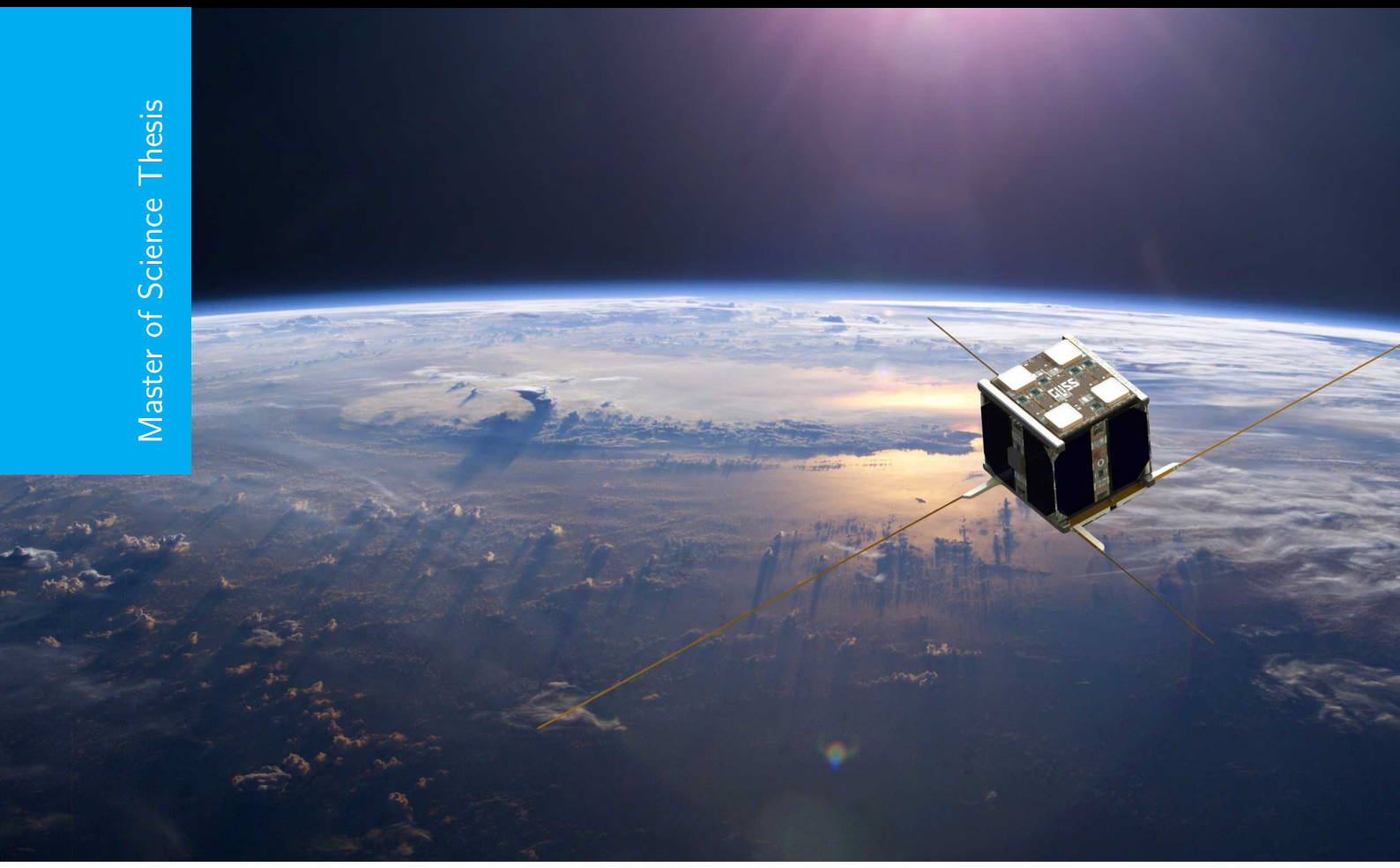
FPGA-based soft error correction for the memory of microcontrollers

A generic methodology to transparent soft error correction for the off-chip memory of microcontrollers operating in space.

Jan van Tuijn

Master of Science Thesis



TU Delft Delft University of Technology

ISISPACE

FPGA-based soft error correction for the memory of microcontrollers

A generic methodology to transparent soft error correction for the off-chip memory of microcontrollers operating in space.

Master of Science Thesis

by

Jan van Tuijn

For the degree of Master of Science in Embedded Systems at Delft University of Technology, to be defended publicly on Wednesday August 10, 2022 at 10:00 AM.

Student number:	4369297	
Thesis number :		
Thesis committee:	prof. dr. ir. G. Gaydadjiev	TU Delft, supervisor
	G. Aalbers	ISISPACE
	Dr. Ir. M. Taouil	TU Delft
	Dr.ir. C.J.M. Verhoeven	TU Delft

Cover image from European Space Agency (ESA)

TU Delft Delft University of Technology

ISISPACE

Abstract

Manufacturers of CubeSats prefer the use of COTS electronic components such as microcontrollers (MCU) and SDRAM but these components are vulnerable to errors caused by radiation. A specific type of error caused by radiation are soft errors which can be corrected by Error Detection and Correction (EDAC) methods. Unfortunately, it is uncommon for MCUs to have an (optimal) EDAC solution integrated in their memory controller. In these cases an external solution is required.

This work proposes a generic methodology towards transparent FPGA-based correction of soft errors that can be applied given a specific microcontroller and its off-chip main memory. To determine how powerful the EDAC solution must be the methodology uses reliability models to evaluate the MTTF of the memory system. Then, the methodology describes how the EDAC logic must be designed in the FPGA fabric such that the FPGA and its EDAC logic are transparent to the MCU. A method to periodically scrub the memory transparently to the processor is also included.

A major trade-off of the FPGA based design is that it requires the clock of the MCU memory controller to be lowered to meet the timing requirements of the SDRAM interface. For some designs it is also necessary to lower the memory capacity or to use more SDRAM devices to store the ECC parity bits.

Lastly, a hardware demonstrator was built to provide EDAC capabilities to the onboard computer developed by ISISPACE. The design consisting of an ARM9 microcontroller, an FPGA and two SDRAM devices is capable of correcting a single error in each byte of the 32-bit SDRAM interface of the MCU and can scrub the entire memory in 33 seconds. Software benchmarks showed that computing performance is about half of the original board as a result of halving the memory controller clock.

Acknowledgements

Looking back I'm very grateful for the opportunities this thesis has provided me and I'm proud off all the things I have accomplished in the past year. When I started my masters I did not anticipate on doing a thesis project of such complexity. The design of a PCB with a microcontroller, FPGA and memory has been an extremely valuable experience and an amazing way to start my career as an embedded engineer.

I would like to thank my supervisor from the TU Delft, Prof. Georgi Gaydadjiev for reaching out to ISISPACE to arrange a thesis internship and his guidance during this project. Many thanks to my supervisor Gerard Aalbers from ISISPACE for his daily support during the internship. I also would like to thank Lisanne Kesselaar for answering all my questions during the design of my PCB. Of course, many thanks to all other people from ISISPACE who helped me during my internship.

Lastly, I want to thank my family for their moral and financial support throughout my studies. I could always rely on them when I needed advice when making important life decisions.

Jan van Tuijn
Rotterdam
September 5, 2022

List of Abbreviations

ADCS Attitude Determination Control System

ASIC Application Specific Integrated Circuit

CHDS Command Handling and Data System

COTS Commercial Off The Shelf

DDR Double Data Rate

DRAM Dynamic Random Access Memory

ECC Error Correcting Codes

EDAC Error Detection and Correction

FF Flip-Flip

FPGA Field Programmable Gate Array

HDL Hardware Description Language

IC Integrated Circuit

JEDEC Joint Electron Device Engineering Council

LEO Low Earth Orbit

LFSR Linear Feedback Shift Register

LUT Lookup Table

MCU Multi-Cell Upset

MCU Microcontroller Unit

MTTF Mean Time To Failure

SBU Single-Bit Upset

SDR Single Data Rate

SDRAM Synchronous DRAM

SECDED Single Error Correction and Double Error Detection

SEFI Single-Event Function Interrupt

SEU Single Event Upset

SoC System on Chip

TMR Triple Modular Redundancy

List of Symbols

CL	Column Address Strobe latency measured in clock cycles (Read latency)
d_x	Data bus width of SDRAM interface x
M	Memory capacity
$t_{refresh}$	Time between two Refresh operations
$p(x)$	The probability an event causes x errors
λ	SEU rate in upsets/device/day
L	Maximum number of soft errors that a SEU can cause in a memory word
n	Length of a code word in number of bits
k	Length of message in number of bits
t	Maximum number of errors that can be corrected in a code word
Δlatency	Latency difference of a SDRAM Read operation when comparing two SDRAM clock frequencies
D	Number of SDRAM devices
D_{max}	Maximum number of SDRAM devices

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays, microcontrollers, also referred to as Microcontroller Units (MCUs), play an important role in both consumer and industrial electronic applications. A microcontroller is an Integrated Circuit (IC) containing a processor and memory, usually combined with different peripherals such as ADCs, communication interfaces and general-purpose IO pins. With many different types of connectivity and relatively high computation power compared to its cost and power usage, microcontrollers are an excellent solution for automation, data acquisition or connectivity devices. This is well reflected by the 4.4 billion Arm Cortex-M based microcontrollers shipped in the fourth quarter of 2020 alone [1].

The on-chip memory capacity of most microcontrollers is not sufficient for many applications. Higher-end microcontrollers provide external-memory controllers to interface with external memories such as NOR/NAND-Flash for permanent storage or volatile memory like SDRAM for the running program, instructions and data.

The use of Commercial Off The Shelf (COTS) microcontrollers and memory ICs has also found its way into the space industry. The CubeSat industry pushes for low cost and fast development time while traditionally electronics in spacecraft use expensive radiation-hardened technology. CubeSats are small form factor satellites weighing typically between 1 and 10 kg, characterized by the use of multiple $10 \times 10 \times 10$ cm cubic units as defined in the 'CubeSat' standard. Figure 1-1 shows a render of the PICASSO CubeSat consisting of three standard cubic units.

While the use of COTS components in CubeSat electronics has its advantages, reliability is still a big concern. The intensive radiation environment in which satellites operate can cause failures to occur, interrupting the service provided by the satellite or in the worst case ending the life of a spacecraft.

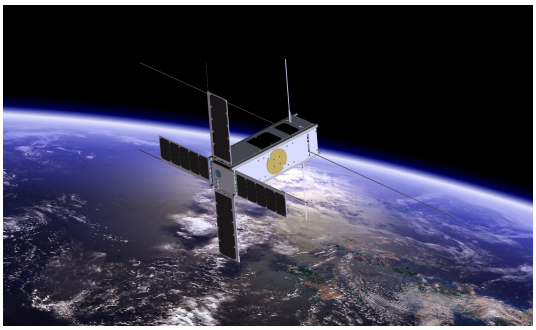


Figure 1-1: Render of the PICASSO CubeSat [2]

Radiation can have two effects on ICs: cumulative effects and Single Event Effects (SEE) [3]. Cumulative effects damage the device which over time causes it to (partially) stop working while SEEs can introduce an error as a result of a single particle event. Radiation errors can be categorized into soft and hard errors [4]. Hard errors are uncorrectable errors caused by permanent damage either accumulated over time or from an SEE. SEEs can cause soft errors that can be corrected by for example repeating the operation, rewriting data, resetting or power-cycling the device.

When a soft error resides in the external memory of a microcontroller it can potentially be corrected. A corrupted bit stored in the memory can be rewritten with the correct bit value if it can be determined what the correct bit value should be. The process of detecting an error and correcting it is called Error Detection and Correction (EDAC). This thesis will focus on EDAC for external SDRAM memory, specifically for Single Data Rate (SDR) SDRAM of COTS microcontrollers to correct soft errors as a result of the radiation present in a space environment. When this work refers to SDRAM it specifically means SDR SDRAM, unless otherwise specified.

1.2 EDAC for external memory

Error Detection and Correction for memory is not a new concept but there are several specific issues regarding the implementation for microcontrollers which are addressed in this section along with a brief introduction of EDAC techniques for memory.

1.2.1 EDAC

Error detection and correction is done by adding redundant information to the data stored in memory, creating a codeword. Error Correcting Codes (ECC) structure codewords in a particular way such that errors can be detected and or corrected [5]. The detection and correction capabilities depend on the minimum Hamming distance [1] between all codewords. A larger Hamming distance allows for larger errors to be detected and corrected but at the expense of more overhead. For example, a simple error correcting code can be created for a single bit message (0 or 1) using the codewords 000 and 111. Any single bit error pattern can

[1] Hamming distance: number of positions that differ between a sequence $\mathbf{x}$ and $\mathbf{y}$ [5].

be corrected in this case. Conversely, the codewords 00 and 11 would only be able to detect errors, it can not be determined whether the corrupted codeword 01 was 00 or 11.

Certain error patterns can also be prevented with additional EDAC features such as scrubbing and interleaving. Scrubbing periodically reads and corrects the memory to prevent the accumulation of errors. Interleaving can be use to transform a burst of errors in a single codeword to single errors in multiple codewords.

1.2.2 Integrated EDAC

EDAC can be implemented in both hardware or software. The advantage of using a memory controller with an integrated EDAC solution is that the encoding, decoding and storage of the parity bits is completely transparent. The programmer of an application does not need to be aware of the existence of the EDAC implementation.

Memory controllers in server or workstation grade computers used in terrestrial applications often use integrated EDAC solutions to improve reliability. A study by Google in 2007 showed that about a third of all their server machines saw at least one error that was corrected by the machine's EDAC solution each year [6] showing the value of having error correcting capable memory systems.

A major disadvantage of integrated EDAC is that its features are fixed. This gives several problems:

1. First, the EDAC capabilities of a specific MCU might not be able to correct enough errors in order to reach the desired level of reliability for a given system. On the other hand, if an EDAC solution is more powerful than is needed, more resources or performance are wasted on its implementation. The lack of flexibility comes at the cost of either reliability or performance.

2. Second, the EDAC features are often not complete. Some only have ECC to correct errors, some include scrubbing while others also support interleaving. Missing features limit the amount of reliability that can be achieved.

An issue specific to microcontrollers is that microcontrollers with EDAC for external memory are hard to find. This makes the lack of flexibility and features of EDAC solutions even more problematic because selecting a microcontroller with the most optimal EDAC solution forces developers to switch between different microcontroller families or vendors. This is often undesirable because it leads to more changes in hardware, rewriting software drivers for the new hardware and the use of new software development tools. All of this increases development time and costs.

1.2.3 Field Programmable Gate Arrays

Adding EDAC capabilities to a microcontroller without integrated EDAC requires an external solution to keep the EDAC transparent to the programmer of the microcontroller. For external digital logic a Field Programmable Gate Array (FPGA) is a common approach. FPGAs are a type of IC consisting of an array of thousands up to millions of logic elements (LE) that can be wired to each other using the programmable interconnect infrastructure as shown in Figure 1-2. Each LE consists of a Lookup Table (LUT) that implements a logic function. The output of the LUT can be wired to a Flip-Flip (FF) to implement synchronous logic. Surrounding the array of LEs and interconnect are many IO banks connecting the FPGA to other external components.

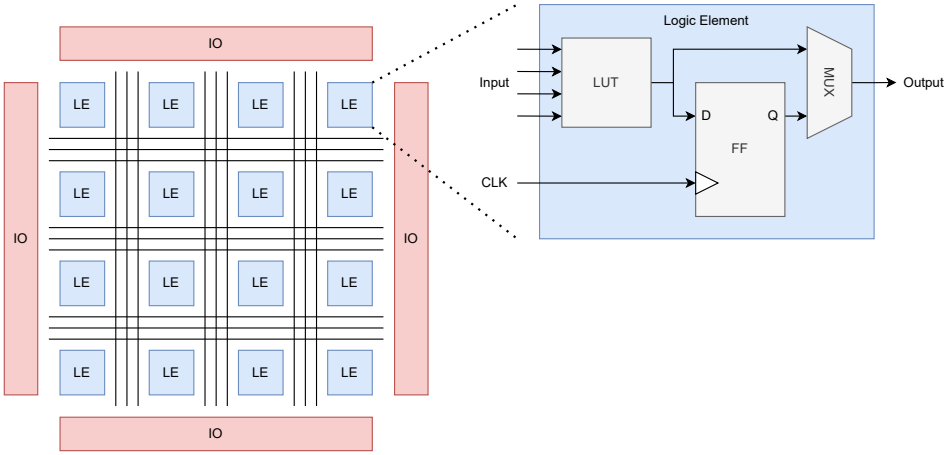


Figure 1-2: Overview of an FPGA. Shown is the array of logic elements connected by a routing networ and IO. Each logic element contains a programmable LUT that implements a logic specific function. The output of a LUT can be connected to a FF or to the routing network.

The architecture of FPGAs allows designers to create large digital circuits that can be reconfigured by reloading the FPGA configuration memory, something that is not possible when implementing a digital circuit in an Application Specific Integrated Circuit (ASIC). The disadvantage of FPGA over ASICs is that FPGAs are slower and can consume more power.

1.3 Problem statement

The main goal of this thesis is:

To develop a proposal for a generic methodology towards transparent FPGA based recovery from soft errors, that can be applied given a specific microcontroller, its off-chip main memory and the parameters of the targeted space mission.

The final design created as a result from applying the methodology to a given system will be able to deal with simultaneous errors up to a pre-defined number by adding error correcting capabilities transparent to the microcontroller. In order to develop a definition of this methodology the following research questions need to be answered:

1. *What type of soft errors can be found in SDRAM of systems operating in low earth orbit and how often do they typically occur?*
2. *Are there affordable error correcting solutions in terms of FPGA implementation costs to cope with these soft errors?*
3. *Can a methodology be derived to integrate together the solutions of the previous question transparently to any microcontroller given a specific space mission?*

1.4 Thesis outline

The content of this thesis has the following outline. In Chapter 2 some background information about SDRAM will be given. Chapter 3 presents the types of soft errors that can occur in SDRAM found in the literature. In Chapter 4 describes several EDAC methods and their implementations. Then several design considerations for the implementation of FPGA based EDAC are discussed Chapter 5. Finally, all information is combined to define the methodology in Chapter 6. Chapter 7 shows the hardware demonstrator that was made for this project.

Chapter 2

SDRAM technology

In this chapter the working principles of (Synchronous) Dynamic RAM are introduced. Understanding how SDRAM operates and how it is accessed is needed later to determine how the logic implemented in the FPGA will interact with the MCU and the SDRAM devices.

In Section 2.1 the DRAM memory cell is introduced as described in [7, 8]. Section 2.2 describes how the memory cells are organized and Section 2.3 summarizes the operations that are required to access the memory. The last section discusses the SDRAM interface.

2.1 DRAM memory cell

Dynamic Random Access Memory (DRAM) refers to memory technology that uses electric charge to store information. Due to leakage the charge of the storage cell can be lost meaning that DRAM cells must be refreshed periodically to prevent the loss of the stored data [7].

In DRAM each bit is stored in a memory cell. Nowadays, the memory cell is constructed using a single access transistor and a single storage capacitor (1T1C) as shown in Figure 2-1. This circuit requires a small amount of silicon area which allows for high density implementations. A single bit of information (0 or 1) can be stored in the memory cell by either charging or discharging the capacitor.

Writing to the memory cell requires changing the charge of the capacitor which is done by enabling the access transistor via the wordline and applying a voltage on the bitline. Similarly, reading is done by enabling the access transistor which discharges the storage capacitor via the bitline. The change in voltage can be sensed to determine if a 0 or 1 was stored. Reading is thus a destructive operation and requires the capacitor to be recharged to its original state.

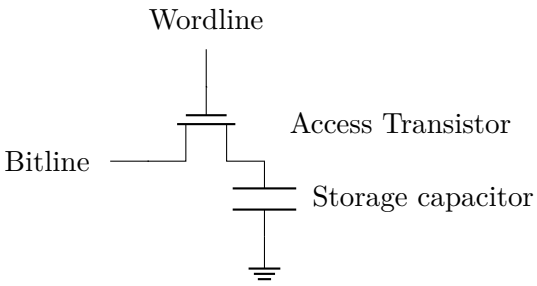


Figure 2-1: Circuit of a 1T1C DRAM memory cell

2.2 SDRAM organization

DRAM is typically organized in an array of memory cells. Rows are constructed using multiple wordlines each connecting the gates of multiple access transistors together. A row decoder translates a row address to a specific row in the memory array. During the activation step a single row is activated allowing multiple cells to be read or written. Columns are created by connecting bitlines together vertically meaning all memory cells in a column share the same sense amplifier. The column decoder selects specific memory cells in each rows based on the address given to the SDRAM.

Figure 2-2 shows simplified block diagram of a 3×3 DRAM array with three rows and three columns. Real SDRAM devices can incorporate thousands of cells in each row and column.

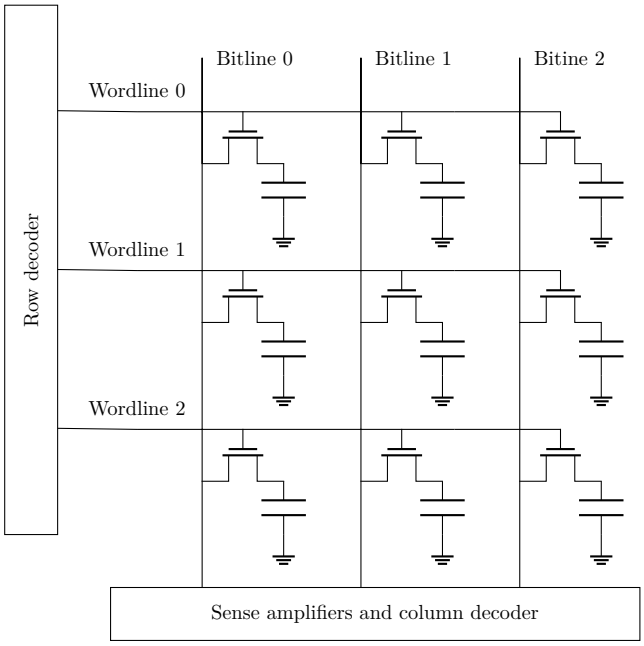


Figure 2-2: Example of a 3×3 DRAM cell array

A modern SDRAM chip consists of multiple arrays which are referred to as banks. Each bank has its own column decoder, row decoder and sense amplifiers. This allows banks to be operated independently such that each bank can be in a different state of a read or write operation. Additionally, multiple rows can be activated at the same time allowing parallel read or write operations. Refresh operations are also performed per row meaning single refresh operation refreshes an entire row. The most common type of DRAM that is used nowadays is Synchronous DRAM (SDRAM) which use a clock signal to synchronize commands, addresses and data on the interface. The advantage of a synchronous design is the interface can be pipelined which improves throughput.

An example of an SDRAM chip is shown in Figure 2-3. This memory device consists of four banks each with 8,192 rows, 2,048 columns and each address holding four bits. The figure also shows that there is a lot of control logic surrounding the memory arrays to process the commands, refresh the memory banks and to decode the addresses.

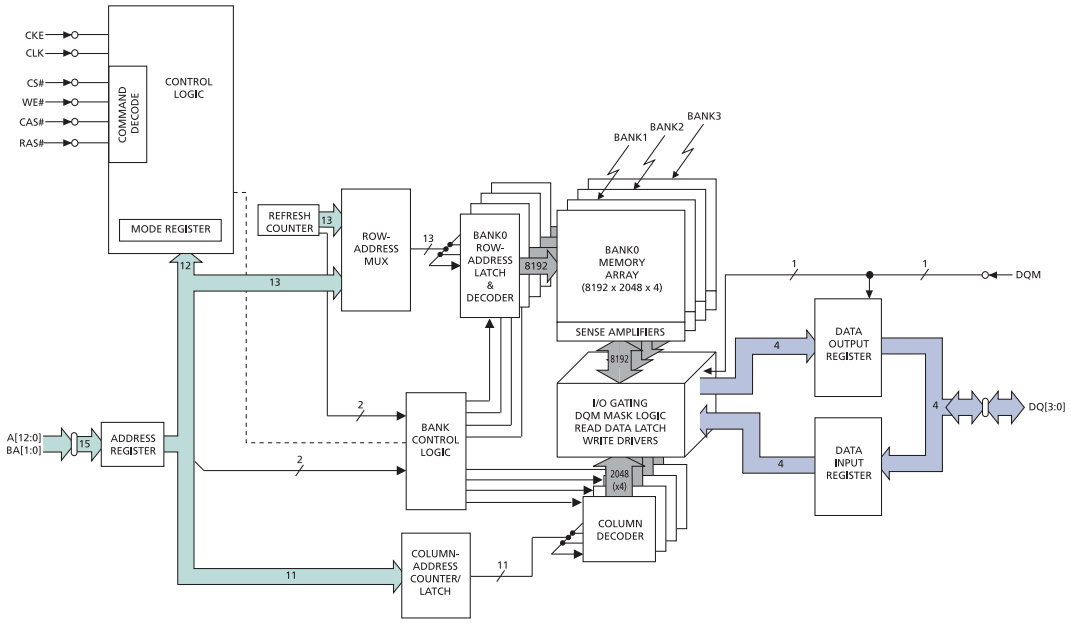


Figure 2-3: Block diagram of a Micron MT48LC64M4A2 256 Mbit SDRAM [9]

2.3 SDRAM operations

Reading and writing to DRAM must be done in multiple steps. In this section the required steps will be discussed from a perspective outside the DRAM together with a brief explanation of what happens internally. The internal steps are discussed in more detail in [7, 10].

Reading from DRAM

Reading a bit from the memory array is a destructive operation. Enabling the wordline discharges the storage capacitor, destroying the information it stored. To restore the information the cell must be charged to its original charge value after it has been read. A read operation involves the following steps:

1. **Precharge**: the bitlines are precharged to a reference voltage and all wordlines are deactivated.
2. **Activation**: the wordline of the given row address is activated allowing the memory cell capacitors to slightly charge or discharge the bitlines depending on the state of the cell. The change of voltage on each bitline is amplified by a sense amplifier (SA). The sense amplifier also acts as a buffer allowing the same cell to be read multiple times without having to perform another precharge and activation step. Because each bitline has an SA, the entire row of memory cells is activated and buffered.
3. **Read**: the column decoder selects the memory cells using the given column address and transfers the amplified signals of these cells to the output register, now as a digital value.

The SA amplifies the bitline to its maximum or minimum voltage while the wordline is still active. This also recharges the storage cell to its original state. Thus the effect of the destructive read is automatically addressed.

Writing to DRAM

The inital steps needed for writing data to the memory are the same as for a read operation, after a precharge and activation step the actual write can take place:

1. **Precharge**: the bitlines are precharged to a reference voltage and all wordlines are deactivated.
2. **Activation**: the wordline of the given row address is activated. Internally the same amplification and buffering of the bitlines are performed as in the activation during the read operation.
3. **Write**: the write data is transferred from the input register to the column decoder that selects the columns of sense amplifiers that need to be overdriven causing the memory cells of the addressed column to be written.

Refreshing

The DRAM memory cell is not ideal and overtime the charge of the storage capacitor will leak away. Therefore, the cell needs to be refreshed regularly. This requires the following steps:

1. **Precharge**: the bitlines are precharged to a reference voltage and the wordlines are deactivated.
2. **Refresh**: the refresh step is essentially an activation step that automatically restores the content of a memory cell. Normally an activation opens a row such that its columns can be accessed. In case of a refresh step each row is incrementally activated, automatically refreshing their state. This process must be repeated endlessly. Typically an SDRAM device must be refreshed entirely every 64 ms.

2.4 SDRAM interface

The design of a typical SDRAM device looks similar to the one shown in Figure 2-3. SDRAM devices like the one shown in Figure 2-3 are standardized in the JESD21-C standard developed by the Joint Electron Device Engineering Council (JEDEC) organization [11]. This standard describes everything from IC packages to IO signals, functional descriptions and timing characteristics. In this section the signals and the timing characteristics of the SDRAM interface will be summarized.

Signals

The signals of SDRAM interface can be categorized into four groups:

- **Data signals** transfer the data to and from the SDRAM. The signals are bidirectional. SDRAM chips exists with 4, 8, 16 or 32 data signals.
- **Data mask signals** are needed when only a part of a memory word needs to be written. When a processor only writes a specific byte instead of a whole memory word the memory controller activates the mask signals of the bytes in the memory word that must be ignored. For example, a 32-bit memory word requires four mask signals such that each individual byte can be masked in case of a write operation.
- **Address signals** are used to select the banks, rows and columns of the memory arrays. Seperate address signals exists for the banks while the row and columns share the same address signals.
- **Command signals** define the commands such a Precharge, Activation, Read or Write etc. Included in this group is also the clock signal.

Table 2-1 shows an overview of all the signals of the SDRAM interface. Excluded from the table are power related signals.

Timing specifications

Operating the DRAM memory cell is an analog process which means it takes time for the Precharging, Activation, Read and Write operations to complete. JESD21-C specifies the minimum and maximum timing specifications that must be met when accessing the SDRAM device but actual values may differ between vendors.

The following timing specifications are the most important to consider when sending commands and data over an SDRAM interface:

- **Precharge period** t_{PR} is the maximum amount of time it takes for a bank to be precharged. When t_{PR} time has passed an Activation or Refresh command can be performed.
- **Activate-to-Read or Write delay** t_{RCD} is the maximum amount of time it takes an activation of a row in bank to be completed. After t_{RCD} activated row can be read or written.

- **CAS latency** CL is the latency of a Read operation in clock cycles. The SDRAM will output the data of a Read operation CL clock cycles after the rising edge at which the Read command was registered.

  Write operations has a latency of zero clock cycles.

- **Row fresh cycle time** t_{RFC} is duration of a refresh cycle. After Refresh command has been registered no other commands must be send for at least t_{RFC}.

Table 2-1: SDRAM signal descriptions

Symbol	Name	Description
CLK	Clock	All signals are registered at the rising edge of this signal.
CKE	Clock enable	Activates or deactivates the clock. Used to power-down the SDRAM.
CS	Chip select	Enables or disables the registering of all other command signals
WE	Write enable	
RAS	Row address strobe	Define the command together with CS.
CAS	Column address strobe	
A[r:0]	Row/column address	Provides the row and column address of Activate, Read and Write operations.
BA[b:0]	Bank address	Provides the bank address of Precharge, Activate, Read and Write operations.
DQ[d:0]	Data	Data bus of $d + 1$ bits.
DQM[b:0]	Data mask	One mask signal for each byte in the data bus.

Chapter 3

Radiation induced soft errors in DRAM

This chapter describes the different types of soft errors that can occur in SDRAM caused by radiation. Knowing what types of soft errors occur in SDRAM is required to determine which EDAC implementations are needed to correct them.

First, a small introduction is given on what radiation does to integrated circuits. Then, in Section 3.2, the chapter will discuss the types of soft errors that can occur in SDRAM. For each type of soft error the rate of occurrence in Low Earth Orbit (LEO) is estimated based on data found in the literature. The estimations are limited to LEO because that is the common orbit for CubeSats missions.

3.1 Radiation in semiconductors

The effects of radiation in semiconductors are well described in [3, 12, 13]. When an ionizing particle strikes a semiconductor it ionizes the device by depositing energy. There are two ways this ionization can happen: direct and indirect ionization. Direct ionization is caused by heavy ions that directly transfer energy to the semiconductor as they pass through the material. Lighter particles like protons and neutrons are usually not capable of directly ionizing the semiconductor material. However, they can create nuclear reactions within the semiconductor material creating heavier particles that are more capable of upsetting the device.

Ionization of semiconductor material requires energy. Therefore, the radiation particle loses energy along its ionization track. The amount of energy that a particle loses per distance is determined by its linear energy transfer (LET). The LET of a particle is expressed in $MeV/mg/cm^2$ and depends on its energy, mass and the density of the target material.

A semiconductor device will collect a certain amount of charge Q_{col} as an ionizing particle passes through it. The amount of charge it takes for a circuit to change state is its critical charge Q_{crit}. Whenever a circuit node collects enough charge such that $Q_{col} > Q_{crit}$, an upset will occur in the circuit. This is referred to as a Single Event Upset (SEU).

3.2 Single Event Upsets in SDRAM

SDRAM devices consists of a large array of memory cells that holds the data and control logic that executes the commands send by the memory controller as was discussed in Chapter 2. Unsurprisingly, several different SEU classes have been identified and their upset behaviour depends on the location of the particle strike. In this section the different classes of SEUs commonly found in the literature are described. Additionally, the rate of occurrence of these SEU classes is estimated based on data found in the literature.

3.2.1 Single-Bit Upsets

The first and most common type of SEU is a Single-Bit Upset (SBU) [13]. When a SBU occurs, a single bit in a memory word is upset by a particle strike causing it to change its current state. Three mechanisms that cause SBUs in a DRAM memory cell are summarized in [12, 14]:

- A strike in or near the access transistor or storage capacitor as shown in Figure 3-1 causing enough relaxation of the stored charge that the information of the memory cell is lost [15].
- A bitline strike, causing a reduction of the signal when the bitline is in a floating voltage state [16].
- Combined cell-bit line failure mode in which the combined charge collected by the memory cell and the bitline is larger than Q_{crit} [17].

SBUs can cause bits to change from $1 \rightarrow 0$ or from $0 \rightarrow 1$. A transition of $1 \rightarrow 0$ can be caused by the collection of charge in the storage capacitor or the access transistor, relaxing the stored charge [12, 14]. A $0 \rightarrow 1$ transition is possible because charge can be shunted into the storage node [12].

It's also possible for SDRAM to internally scramble the data by storing the inverse of some bits (typically, half of the memory cells are inverted). The amount of scrambling can differ per vendor and generation of SDRAM [18]. This means that when a particle strike discharges an inverted memory cell the error will be a transition of $0 \rightarrow 1$.

Technology scaling trend

As semiconductor technology scales down in size it is expected that sensitivity to radiation for electronic devices increases [3, 12]. Surprisingly, this is not the case for DRAM. An explanation for this can be found in how technology scaling is applied to DRAM. Figure 3-2a shows how various parameters of the DRAM cell have changed as technology scaled down. As can be seen, the sensitive junction volume has dropped dramatically compared the cell voltage, especially compared to the cell capacitance. A higher LET is needed for a particle to upset a memory cell with a lower sensitive junction volume. Therefore, the soft error rate per bit has decreased as shown in Figure 3-2b. At the same time, technology scaling has enabled higher density DRAM devices which results in a constant soft error rate per device.

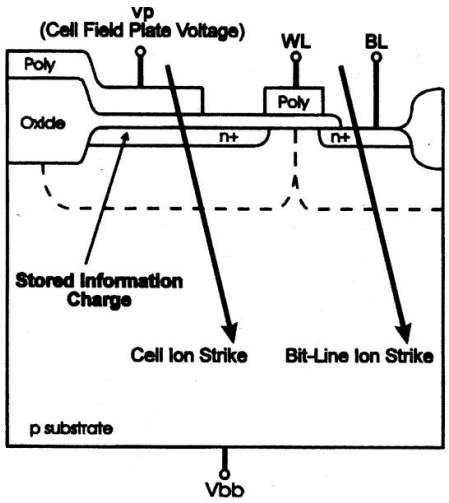
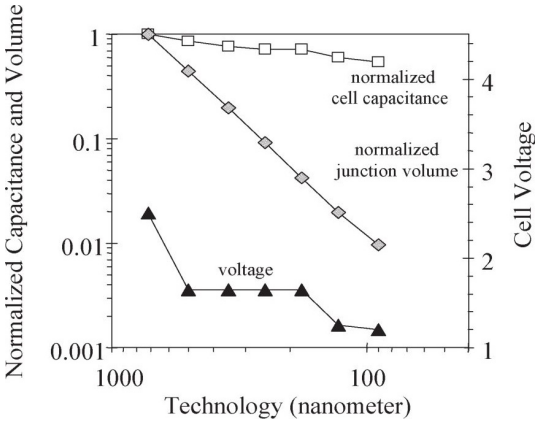
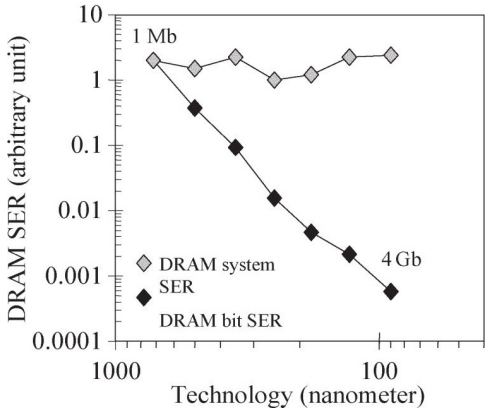


Figure 3-1: Illustration of an ion-strike in a DRAM cell [12]



(a) DRAM voltage, junction volume and capacitance trend as a function of technology.



(b) DRAM soft error rate trend as a function of technology.

Figure 3-2: DRAM scaling trends from [19]

SBU observations in LEO

Table 3-1 shows the SEU rates observed by several space missions for various DRAM devices. No work has been published on SEU rate observations for higher density DRAM devices in similar Low Earth Orbits (LEO) or Polar orbits. The lack of data on more modern high density DRAM devices and the low sample size of the available data makes it difficult to estimate the expected SEU rate in LEO. Therefore, only a worst case upper bound estimate will be used.

The average upset rate for the observations listed in Table 3-1 is 1.68 SEU/device/day. With the worst observation being 6.43 it is safe to estimate a maximum of 10 SEU/device/day. Most of the observed upsets occur when a spacecraft travels through the South Atlantic Anomaly (SAA), this holds for every spacecraft listed in Table 3-1.

Keeping the SEU rate to a constant value for all generations of DRAM requires the assumption that the upset rate per bit drops as devices become more dense while overall device rates

Table 3-1: SEU rates observed in DRAM from various space missions.

Orbit	SEU/device/day	SEU/bit/day	Capacity	Ref.
705 km, 98°	2.7	$6.44 \cdot 10^{-7}$	4 Mb	[20]
350 km, 51.6°	0.10	$2.30 \cdot 10^{-8}$	4 Mb	[21]
707 km, 98.2°	1.49	$8.89 \cdot 10^{-8}$	16 Mb	[22]
707 km, 98.2°	1.38	$2.06 \cdot 10^{-8}$	64 Mb	[22]
707 km, 98.2°	0.82	$1.22 \cdot 10^{-8}$	64 Mb	[22]
707 km, 98.2°	0.11	$1.67 \cdot 10^{-9}$	64 Mb	[22]
820 km, 98.2°	6.43	$9.58 \cdot 10^{-8}$	64 Mb	[23]
LEO	0.10	$1.55 \cdot 10^{-9}$	64 Mb	[24]
LEO	2.98	$2.98 \cdot 10^{-8}$	64 Mb	[24]
LEO	1.97	$2.93 \cdot 10^{-8}$	64 Mb	[24]
ISS	0.03	$2.15 \cdot 10^{-10}$	128 Mb	[24]

remain constant. This assumption is inline with the previously discussed observation of the change in soft error rate and technology scaling.

3.2.2 Multi-Cell and Multi-Bit Upsets

Besides an upset in a single memory cell it also possible for a particle to cause an upset in multiple cells [25]. A single event can cause a Multi-Cell Upset (MCU) due to the diffusion of induced charge by a particle. The diffused charge can be collected by nodes adjacent to the strike location which can lead to multiple upsets [12, 14]. In the literature the terms MCU and Multi-Bit Upset (MBU) are used interchangeably to describe this phenomenon but in this document the definitions in the JESD89A standard [4] are used:

- MCU: a single event upset in which multiple errors occur in different memory words. These cells are physically adjacent but not logically.
- MBU: two or more errors occur in the same word. These cells are physically and logically adjacent.

The number of upsets caused by an MCU or MBU is referred to as its multiplicity [25]. Whether a single particle that upsets multiple memory cells causes an MBU with a multiplicity greater than 1 or an MCU where all upset cells are in different memory words (resulting in multiple SBUs) largely depends on the physical layout of the memory array. Laser testing of a 256 Mb SDRAM device showed that for this particular device memory cells are grouped next to each other in pairs [26] as shown in Figure 3-3. A high energy particle striking this particular device can only cause an MBU of multiplicity 2.

Another work showed that the cell layout differs depending on the manufacturer [27]. This work tested the newer generation of DDR3 SDRAM but similar differences in cell layout could also exist for SDR SDRAM. The tests revealed, as shown in Figure 3-4, that for some devices none of the cells of a word are close to each other while for other devices multiple cells of a word are located next to each other.

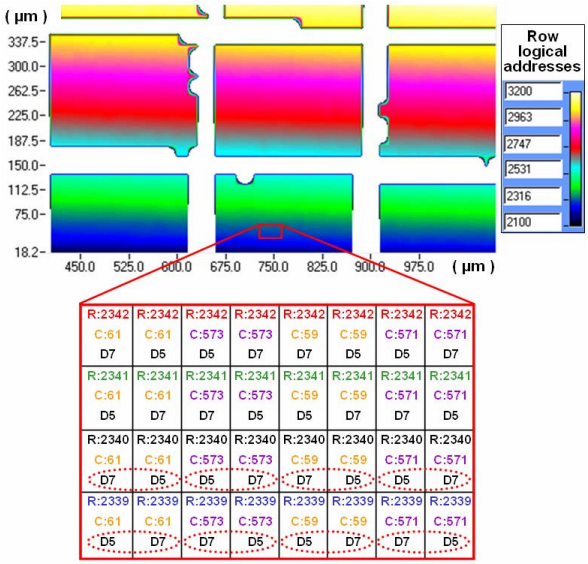


Figure 3-3: Results of laser mapping a 256 Mb SDRAM [26]. The mapping shows cells of the same word are paired next to pairs of a different column.

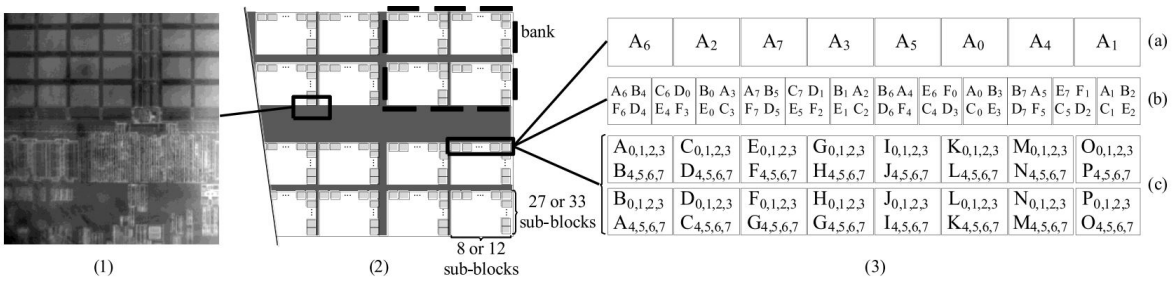


Figure 3-4: Infrared micro-photograph of memory arrays and peripheral circuitry (bottom) (1), schematic representation of a DDR3 SDRAM architecture (2), and representation of the scrambling strategies of the vendors A (a), B (b), and C (c) tested memories (3). [27]

MCU and MBU observations in LEO

MCUs have been observed in SDRAM of several spacecraft orbiting in LEO [21, 28, 22]. The distribution of the multiplicity of these recorded MCUs is plotted in Figure 3-5. The plot shows that SBUs (MCU multiplicity = 1) are likelier to occur than MCUs with multiplicity greater than 2 and the probability of higher multiplicity decreases exponentially. MCUs with multiplicity of 2 happen almost 10% of the time for almost all spacecraft showing that this phenomenon can be a serious problem.

The number of errors a single event can cause is an important parameter when determining the error correcting capability of an EDAC solution. In case the memory cells of memory word in an SDRAM device are all close to each other the probability of a single event causing x errors in that word is similar to the distribution shown in Figure 3-5. Therefore, this distribution is modelled as $p(x) = (1 - 0.8)^{x-1} \cdot 0.8$ and can be used later to calculate the probability of an SEU causing x errors.

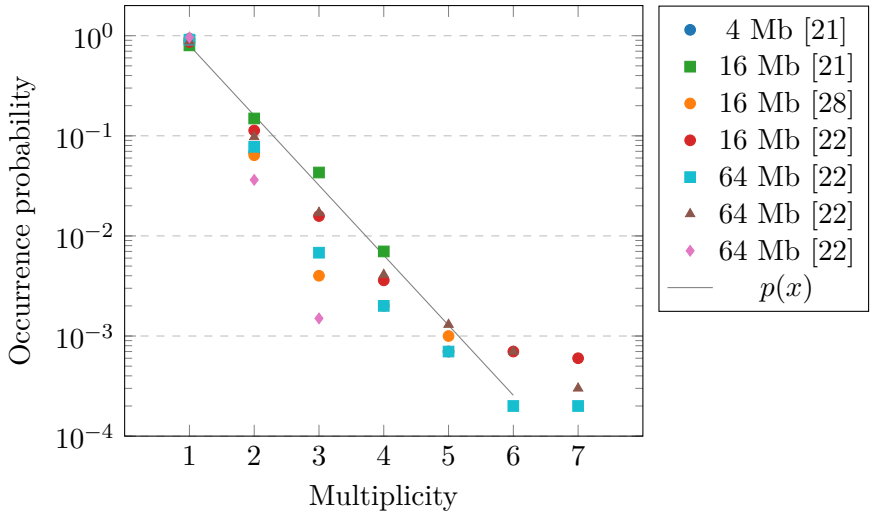


Figure 3-5: MCU multiplicity statistics collected from several spacecraft. With the distribution modelled to $p(x) = (1 - 0.8)^{x-1} \cdot 0.8$.

3.2.3 Single Event Functional Interrupt

DRAM devices also contain logic responsible for decoding the interface commands, refreshing the memory cells, decoding the received address etc. An upset caused by radiation in these parts of the DRAM device is known as a Single-Event Function Interrupt (SEFI) and is generally recognizable by the large number of corrupted memory words in rows or columns [18].

There exist no precise definitions of SEFI types which is why various terms can be found throughout the literature that describe the same or different SEFI events [29, 30, 31]. In [32] the SEFI are categorized into two types:

1. **Transient SEFI**. Transient SEFIs result in errors that disappear at the next access. An upset in the row or column decoder for example, can return the data from a different address but re-accessing the same address will return the correct data if the data wasn't overwritten [32].

2. **Persistent SEFI**. Persistent SEFIs occur when re-accessing the same data does not remove the error. An upset in the refresh circuitry explanation of this SEFI [30]. Fuse-Latch upsets has also been identified as a source of Persistent SEFIs. These Fuse-Latches are responsible for activating redundant rows or columns. Resetting the Mode Register has shown to be an effective removal procedure for this [29]. In the case of Fuse-Latch upsets, data is not necessarily lost. In the worst case a power-cycle is needed to remove the error.

SEFI observations and predictions

SEFIs are expected to becoming more problematic as memory densities and clock frequencies of the synchronous interfaces increase requiring more control logic in each device [12]. Un-

fortunately, few works on actual SEFI rates observations or rate predictions exists. The ones found are shown in Table 3-2.

Estimating a SEFI rate given the amount of data in Table 3-2 is difficult. However, given that the newer generations of Double Data Rate (DDR) memory, like DDR3, are more sensitive to SEFIs and the predicted rates of the DDR3 memories is less than 0.01 SEFI/device/day it safe to assume the SEFI rate for SDR SDRAM is also less than 0.01 SEFI/device/day.

Since devices of 16 Mb can be considered obsolete technology and the other entries in Table 3-2 are worse environments/predictions than LEO in general an upper bound estimation of 0.01 SEFI/device/day is reasonable. The probability distribution of Transient and Persistent SEFIs is not known.

Table 3-2: Observed and predicted SEFI rates

Orbit	SEFI/ device/day	Capacity	SEFI Type	Observation/ Prediction	Ref.
600 km, 29°	$5 \cdot 10^{-6}$	16 Mb	Persistent	Observation	[30]
600-30,000 km, 10°	0.005	512 Mb	Persistent	Observation	[33]
800 km, 98°	0.009	4 Gb (DDR3)	Persistent	Prediction (worst week solar flare)	[32]
800 km, 98°	0.002	4 Gb (DDR3)	Persistent	Prediction (worst week solar flare)	[32]

3.3 Conclusion

Soft errors in SDRAM can divided into three classes. SBU are the most common and in the worst case no more than 10 SEUs per device per day can be expected in LEO but it should be noted that this number is based on a small data set of observed upset rates in satellites. A single event can also cause multiple upsets, in a single memory word (MBU) or in multiple memory words (MCU). MCUs upsetting two memory cells can occur as often as 1 every 10 SEUs but whether those two errors occur in the same word depends on the structure of the SDRAM which can vary between vendor. A third SEU class is the SEFI which is an upset in the control logic of an SDRAM device. The rate of SEFIs is very low for SDR SDRAM and its more problematic for newer DDR memories but it can cause many errors.

This means that an EDAC solution for SDR SDRAM needs to be able to correct multiple errors per memory word in case the SDRAM device(s) suffers from MBUs. If the EDAC solution is capable of correcting t errors per word and an upset creates $t+1$ errors, the errors will remain in the memory which can lead to a failure. The solution must also be able to prevent errors from accumulating given the upset rate of 10 upsets per device per day. The error correcting capability t of the FPGA design must be adjusted to a level such that the reliability of the memory sub-system is sufficient given the upset rate in LEO.

Chapter 4

Error Detection and Correction

In this chapter multiple error detecting and correcting methods will be discussed that can be used to correct the soft errors discussed in the previous chapter. Later this information is used to determine which methods are suitable to be used in the FPGA design.

For the EDAC methods different types of implementations will be discussed and for each implementation an estimation of the time and resource complexity will be derived. The second section discusses supplementary techniques that can be used to prevent errors from becoming uncorrectable. The final section present some models that can be use to determine how much the EDAC methods improve the reliability of a memory system.

4.1 Error Correcting Codes

The process of using ECCs can be described as follows. Messages are mapped to a code word during a encoding process and then transmitted over a communication channel or stored in a storage device in which the code word might become corrupted. The receiver or reader decodes the code word which reverses the encoding process. Depending on the severity of the error and the ECC capabilities the error can be corrected.

This work will focus on block codes as they can be used to encode a block of information. A block code C is a set of M code words $\{\mathbf{c}_0, \mathbf{c}_1, .., \mathbf{c}_{M-1}\}$ and each code word is a tuple $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$ of length n generated by an encoder from a message tuple $\mathbf{u} = (u_0, u_1, ..., u_{k-1})$ of length k. The additional symbols that are added to code word $\mathbf{c}$ are referred to as the parity bits of which there are $n - k$. The received code word $\mathbf{r} = (r_0, r_1, ..., r_{n-1})$ can be modeled as an addition of an error pattern $\mathbf{e} = (e_0, e_1, ..., e_{n-1})$ of length n and the stored code word $\mathbf{c}$ as shown in Figure 4-1. The decoder outputs the message $\mathbf{u}$ if the error pattern is correctable.

How many errors a decoder can detect or correct depends on the *minimum distance* d_{min} of a block code. The minimum distance of a block code C is defined as the minimum Hamming distance of all code words:

$$d_{min} = \min\{d_{\text{Hamming}}(\mathbf{a}, \mathbf{b}) : \mathbf{a}, \mathbf{b} \in C, \mathbf{a} \neq \mathbf{b}\} \tag{4-1}$$

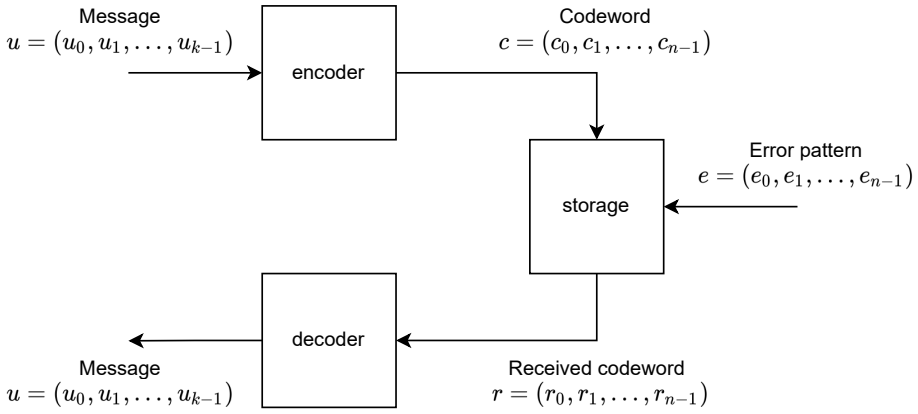


Figure 4-1: Model of the use of ECCs. A message $\mathbf{u}$ is encoded to a code word $\mathbf{c}$. The stored code word can corrupted with an error pattern $\mathbf{e}$. The decoder will restore the error if possible and returns the message $\mathbf{u}$.

The hamming distance is defined as the number of positions at which two code words are not equal:

$$d_{\text{Hamming}}(\mathbf{v}, \mathbf{w}) = d(\mathbf{v}, \mathbf{w}) = |\{i | v_i \neq w_i, i = 0, 1, ..., n-1\}| \tag{4-2}$$

A code C can detect up to s errors if $d_{min} \geq s+1$ and it can correct t errors if $d_{min} \geq 2t+1$. How a mapping of a message $\mathbf{u}$ to a codeword $\mathbf{c}$ ensures a minimum distance d_{min} is not discussed in this report. Instead, in section 4.1.3 a specific type of ECC will be discussed with the property to design code words capable of correcting t errors.

The next sections use the information from [34] and [5] to discuss three types of block codes and their implementations. For each implementation the time and resource complexity is analysed. This information is later used to determine which implementations are optimal for the FPGA design.

4.1.1 Linear block codes

Code words of linear block codes can be generated using a generator matrix. By definition a linear block code C over a field GF(q) is a k-dimensional vector space of k linearly independent vector $\mathbf{g}_0$, $\mathbf{g}_1$, ..., $\mathbf{g}_{k-1}$ meaning that each $\mathbf{c} \in C$ can be written as a linear combination $\mathbf{g}_i$. Thus a generator matrix $\mathbf{G}$ can be constructed using the row vectors $\mathbf{g}_i$ as shown in Equation 4-3.

$$\mathbf{G} = \begin{bmatrix} \mathbf{g_0} \\ \mathbf{g_1} \\ \vdots \\ \mathbf{g}_{k-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \tag{4-3}$$

To simplify retrieving the message part of a code word, a generator matrix $\mathbf{G}$ can be made systematic using Gaussian elimination to achieve the structure $\mathbf{G} = [I_k|\mathbf{P}]$. Then, a code

word $\mathbf{c}$ consisting of the original message $\mathbf{u}$ and the generated parity bits $\mathbf{p}$ is generated as follows:

$$\mathbf{c} = \mathbf{uG} = \begin{bmatrix} u_0 & u_1 & \dots & u_k & | & p_0 & p_1 & \dots & p_{n-k-1} \end{bmatrix} \tag{4-4}$$

Another property of linear codes is the possibility to construct a parity-check matrix $\mathbf{H} = [\mathbf{I}_{n-k} | -\mathbf{P}^\intercal]$ from $\mathbf{G}$ of size $(n-k) \times n$. How this parity-check matrix is used in the decoding process is discussed later.

Encoding

Implementing the matrix multiplication Equation 4-4 requires fixed multiplication and addition arithmetic. Digital systems only contain symbols from GF(2) which greatly reduces the computational complexity of the multiplication and addition. Table 4-1 and 4-2 show the truth tables of addition and multiplication in GF(2). Addition can be implemented with an XOR-gate. Multiplication is implemented with a connection if $g = 1$ or no connection if $g = 0$.

Table 4-1: Addition $c = a + b$ in $GF(2)$.

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

Table 4-2: Multiplication $c = g \cdot a$ in $GF(2)$.

g	a	c
0	0	0
0	1	0
1	0	0
1	1	1

For a systematic $\mathbf{G}$ of size $k \times n$ matrix, the multiplication $\mathbf{c} = \mathbf{uG}$ consists of $n-k$ dot-products and each dot-product requires at most k fixed multiplications and $k-1$ additions. Thus in worst case a k-XOR gate is needed which can be constructed from a tree of XOR gates with a depth of $\log_2(k)$. Thus the total encoding time of a linear block code using matrix multiplication is:

$$T_{matrix\ encoding} \leq \log_2(k) \cdot T_{\text{XOR}} \tag{4-5}$$

The resource usage of this decoder is $k-1$ XOR-gates for each $n-k$ dot-product:

$$A_{matrix\ encoding} \leq (n-k) \cdot (k-1) \cdot A_{\text{XOR}} \tag{4-6}$$

Thus encoding a linear block code can be done with combinational logic only and requires no registers. A diagram of the encoding circuitry is shown in Figure A-1.

Decoding

The previously discussed parity-check matrix has the property that for any valid code word $\mathbf{r}$ the syndrome $\mathbf{s} = \mathbf{r} \cdot \mathbf{H}^\intercal = 0$. A corrupted code word $\mathbf{r} = \mathbf{c} + \mathbf{e}$ is a linear combination of a valid code word $\mathbf{c}$ and an error pattern $\mathbf{e}$. As shown in Equation 4-7 the syndrome $\mathbf{s}$ independent of the received code word $\mathbf{r}$:

$$\begin{aligned}\mathbf{s} &= \mathbf{r} \cdot \mathbf{H}^\intercal \\ &= (\mathbf{c} + \mathbf{e}) \cdot \mathbf{H}^\intercal \\ &= \mathbf{c} \cdot \mathbf{H}^\intercal + \mathbf{e} \cdot \mathbf{H}^\intercal \\ &= 0 + \mathbf{e} \cdot \mathbf{H}^\intercal\end{aligned} \tag{4-7}$$

Thus for decoding, the first step is to determine if an error has occurred by calculating the syndrome which requires a matrix multiplication like the one in the encoding step. The parity-check matrix $\mathbf{H}^\intercal$ is an $n-k \times n$ matrix requiring $n-k$ dot products, each consisting of at most n fixed multiplications and $n-1$ additions.

Next, the syndrome has to be matched with an error pattern. If the syndrome $\mathbf{s} = 0$, the received vector is a valid code word. If $\mathbf{s} \neq 0$, an error has occurred and the error pattern has to be determined.

The syndromes of all error patterns can be stored in a lookup table which is a low complexity method of determining the error pattern. A maximum lookup table stores an error pattern for all 2^{n-k} syndromes while a minimum table only stores the error patterns up to t errors. The respectable sizes of the lookup tables are as shown in Equation 4-8. After the error pattern has been determined the error can be corrected by simply adding the pattern $\mathbf{e}$ to the received code word $\mathbf{r}$, removing the error.

$$\text{maximal LUT size [bits]} = n \cdot 2^{n-k} \tag{4-8}$$

$$\text{minimal LUT size [bits]} = n \cdot \sum_{i=1}^{t} \binom{n}{i} \tag{4-9}$$

The total decoding time complexity is the delay path of the matrix multiplication, the error pattern lookup and the final error correction:

$$T_{matrix\ decoding} \leq \log_2(n) \cdot T_{XOR} + T_{lookup} + T_{XOR} \tag{4-10}$$

The total resource complexity of the decoder is (excluding the memory needed to store the error pattern lookup table):

$$A_{matrix\ decoding} \leq ((n-k) \cdot (n-1) + n) \cdot A_{XOR} \tag{4-11}$$

A complete overview of the decoding circuit is shown in Figure A-2.

4.1.2 Linear cyclic codes

A subset of linear block codes are linear cyclic block codes with the property that a cyclic shift of any code word $\mathbf{c} \in C$ gives another code word $\mathbf{c}' \in C$. Cyclic codes can be described using polynomials. The polynomial representation of a code word $\mathbf{c}$ is $c(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}$.

Similar to the generator and parity-check matrix discussed in the previous section, cyclic codes use a generator polynomial $g(x)$ and a parity-check polynomial $h(x)$. These can be used to generate code words $c(x) = u(x)g(x)$ and the syndrome $s(x) = r(x)h(x)$. However, because of the cyclic property more efficient encoder and decoder structures can be made using shift registers.

Encoding

Simply multiplying $u(x)$ with $g(x)$ results in a non-systematic code word. Creating a systematic cyclic encoding is a slightly more complicated process and requires the following steps [34]:

1. Multiply the message polynomial $u(x)$ with x^{n-k}
2. Calculate the remainder $d(x)$ of the division $x^{n-k}m(x)/g(x)$
3. Calculate $c(x) = x^{n-k}u(x) - d(x)$

All three steps can be combined in a single Linear Feedback Shift Register (LFSR) circuit combined with some switches [34]. During encoding, the LFSR calculates the remainder $d(x)$ during the first k shifts. Then the parity bits are shifted out the LFSR in the next $n - k$ shifts. The total encoding time is therefore equal to n shifts:

$$T_{cyclic\ encoding} = n \cdot T_{clk} \tag{4-12}$$

The encoder requires $n - k$ Flip-Flops and in worst case $n - k$ XOR gates (depending on the generator polynomial $g(x)$):

$$A_{cyclic\ encoding} \leq (n-k) \cdot A_{FF} + (n-k) \cdot A_{XOR} \tag{4-13}$$

Decoding

The error pattern in a code word can also be written as a polynomial giving the received polynomial $r(x) = c(x) + e(x)$. Because the code word $c(x)$ was encoded by multiplying $u(x)$ and $g(x)$ the syndrome of $r(x)$ is simply the remainder of $r(x)$ divided by $g(x)$.

This polynomial division can be done using a LFSR similar to the one used in the encoding process. Besides calculating the syndrome, special decoders such as the Meggitt decoder exist that use the properties of cyclic codes such that only one syndrome for each error pattern $\mathbf{e}$ and all cyclic shifts of $\mathbf{e}$ need to be calculated [5]. This means the error pattern lookup table can be reduced in size with a factor n.

However, a Meggit decoder requires $2n$ shifts which adds a lot of latency. Therefore in this work the LFSR will only be considered to calculate the syndrome and the error pattern will be determined using a lookup table. This means the total decoding time complexity is the number of shifts of the LFSR, an error pattern lookup and an XOR gate for the error correction:

$$T_{cyclic\ decoding} = n \cdot T_{clk} + T_{lookup} + T_{XOR} \tag{4-14}$$

The resource usage of the cyclic decoder is again $n-k$ Flip-Flops and in worst case $n-k$ XOR gates for the LFSR. The error correction requires n XOR gates. The total resource complexity is (excluding the memory for the lookup table):

$$A_{cyclic\ decoding} \leq (n-k) \cdot A_{FF} + (2n-k) \cdot A_{XOR} \tag{4-15}$$

4.1.3 BCH codes

Bose–Chaudhuri–Hocquenghem (BCH) codes are a type of cyclic codes with the two great properties that the two previously mentioned methods lack. First, by design BCH codes guarantee a specific number of correctable errors t for a code words of length $n = 2^m - 1$. Secondly, BCH codes can be decoded algebraically removing the need for a large error pattern lookup table. Because BCH codes are cyclic codes they can be encoded using a LFSR or by transforming the generator polynomial to a generator matrix.

Decoding

The decoding process of BCH codes can be divided into three steps. Since the math on the constructing of BCH codes hasn't been discussed and the math involved in the decoding process is complex the reader is referred to [34] or [5] for more detail. The following is a summary of these works:

1. **Computing the syndrome**. The first step is to calculate the syndrome which can be done with a parity matrix multiplication that can be implemented with an XOR-tree.

2. **Determining the error locator polynomial (ELP)**. The calculated syndromes can be re-expressed in a set of complex nonlinear equations, expressing the unknown error locators X_l for an unknown error $l = 1 \ldots v$. It is difficult to solve a solution for this set of equations. A more simplified method uses the Error Locator Polynomial (ELP) $\Lambda(x)$ shown in Eq. 4-16. The roots of this ELP are the inverse of the v errors locators.

$$\Lambda(x) = \prod_{t=1}^{v}(1 - X_l x) = \Lambda_v x^v + \Lambda_{v-1} x^{v-1} + \cdots + \Lambda_1 x + \Lambda_0 \tag{4-16}$$

   There are multiple algorithms that can be used to determine the coefficients of the ELP of which the most known are the Peterson and the Berlekamp algorithm.

3. **Finding the root of the ELP**. Lastly the roots of $\Lambda(x)$ have to be calculated which returns the error locators and therefore the locations of the errors. This is often implemented with a Chien search algorithm.

The second step in the BCH decoding process is the most time consuming [35]. The solution presented in [35] uses the Berlekamp algorithm which requires t iterations and has a critical path of $(2 + \log_2(m-1)) \cdot T_{AND} + (4\log_2(m) + 2\log_2(t)) \cdot T_{XOR} + (m-1) \cdot T_{OR}$ operations. A complete overview of the time and resource complexity of a BCH decoder using the Berlekamp algorithm can be found in [35].

Constructing codes

For arbitrary generator matrices and generator polynomials as previously discussed, a search must be done on all code words to determine the minimum distance and therefore the error correcting capability t. The method of constructing a t correcting BCH code is found in [34] or [5] but nowadays tools such as Matlab provide simple functions to create codes quickly [36].

The code words created using the BCH method have a specific message length but memory systems store information in bytes or multiple bytes. To prevent memory from being wasted the code words can be shortened. Shortening reduces a (n, k) code word to a $(n - a, k - a)$ code word.

4.2 Triple Modular Redundancy

Another error correcting technique commonly found in space systems is Triple Modular Redundancy (TMR). In a TMR system all processing or storage is done in threefold. A voter is used to determine the correct output and because of the odd number of inputs there will always be a majority that is equal.

TMR allows multiple errors in one copy of the original data but not multiple errors in the same bit position for each copy. A disadvantage of TMR is the large amount of overhead. For each data word two additional copies have to be stored which results in a 200% overhead. This means an EDAC solution using TMR needs three times more memory capacity.

For storage applications no encoding is required because the data is simply stored three times. Decoding a memory word requires a majority voter for each bit in the memory word. Equation 4-17 shows the Boolean equation of a majority voter outputting the corrected memory word v given a word x and its copies y and z. The circuit equivalent of Equation 4-17 is shown in Figure 4-2.

$$v_i = x_i \cdot y_i + x_i \cdot z_i + y_i \cdot z_i \tag{4-17}$$

The total latency of a TMR decoder circuit is:

$$T_{tmr} = T_{\text{AND}} + T_{\text{OR}} \tag{4-18}$$

The resource usage of a TMR decoder consists of the gates shown in Figure 4-2 for all k bits:

$$A_{tmr} = 3 \cdot k \cdot (A_{\text{AND}} + A_{\text{3-OR}}) \tag{4-19}$$

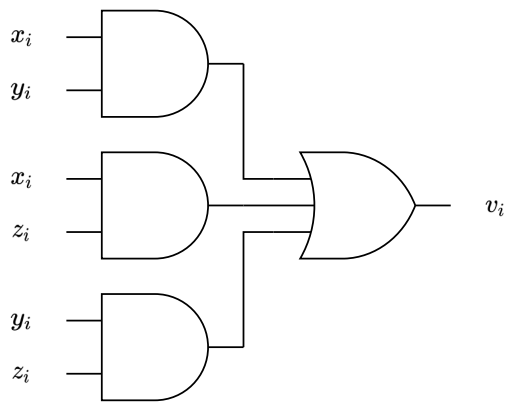


Figure 4-2: Majority voter digital logic implementation of Equation 4-17

4.3 Supplementary EDAC techniques

So far only error correcting methods have been discussed. This section describes two common techniques that supplement the EDAC by preventing certain errors patterns.

Interleaving

With interleaving the content of a memory word is distributed in such a way that each chunk of the memory word is physically separated. The level of interleaving is determined by the size of each chunk and the amount of physical separation determines.

Interleaving is used to transform burst errors (such as single event MBUs) into single errors. When a soft error upsets multiple bits that are physically close to each other, the EDAC solution cannot correct them if the number of errors is greater than the error correcting capability. By interleaving the memory words the burst error will affect multiple code words but the number of errors in each code word will be smaller increasing the chance that the errors are correctable.

Scrubbing

Memory scrubbing is a technique that scans the memory for errors. The read operation of the scrubber uses the implemented EDAC method (ECC or TMR) to detect and correct errors and writes the corrected memory word back to memory. After scanning all addresses of the memory the scrubber will repeat the same memory scan.

The reason to scan the memory for errors in contrast to relying on correcting errors when the processor reads from memory is that its prevents the accumulation of errors. Not all information stored in memory is accessed with the same frequency. When a memory address is not accessed for a longer period the probability that multiple soft errors occur in the same memory word increases. The more likely the error will becomes uncorrectable.

4.4 Reliability

In the previous sections different methods of error correcting have been discussed. These methods can be applied to a memory system to correct errors which improves the reliability. However, the implementation of these EDAC methods comes at the expense of increased latency and decreased throughput because of the time complexity T, the logic resources A and reduced memory capacity because of the $n - k$ parity bits. The selection of an EDAC solution in a memory design is thus a trade-off between the reliability provided by the EDAC solution and the costs of its implementation but an expression of the reliability has not been defined yet.

The reliability of a system can be expressed in its Mean Time To Failure (MTTF). This section will discuss how a MTTF value can be derived for a memory system suffering from soft errors. Testing the MTTF of a memory system in a lab environment using a radiation source or in an actual spacecraft is extremely costly. Therefore, using a model is a preferred method to determine the MTTF. First, Section 4.4.1 discusses which EDAC solutions are required for each type of SDRAM soft error. Then, in Section 4.4.2 a model is defined for each of these cases.

4.4.1 EDAC solutions for SDRAM soft errors

In Chapter 3 four different types of single event soft errors were discussed: SBUs, MBUs, MCUs and SEFIs. In this chapter EDAC solutions were discussed to correct t errors per memory word, to reduce multi-bit errors using interleaving and to prevent the accumulation of errors by scrubbing the memory. With this information it is possible to select which EDAC solutions must be applied to correct each type of error. Table 4-3 shows an overview of the EDAC solutions that must be applied to SBUs, MBUs and MCUs.

In case a SDRAM device is only sensitive to SBUs or MCUs only a single bit error correcting solution is needed ($t = 1$) because an SBU or MCU can only cause single bit errors. To prevent the accumulation of errors scrubbing can be used. Interleaving is not required because it is use to transform multiple errors into single errors and SBU or MCU cannot cause multiple errors. What distinguishes MCUs from SBUs is that a single MCU will cause multiple memory words to become corrupted instead of a just one, increasing the rate of errors.

When MBUs can occur in a SDRAM device the EDAC solution needs to be able to correct up to L errors, where L is the maximum number of errors a single event can cause. A scrubber is needed as well. Interleaving can be used to reduce L or if sufficient interleaving is applied then the MBU can be treated as an MCU, $L = 1$.

Table 4-3: The EDAC solutions needed for each type of soft error excluding SEFIs.

Soft error type	Error correcting capability	Scrubbing	Interleaving
SBU	$t = 1$	Yes	No
MCU	$t = 1$	Yes	No
MBU	$1 < t \leq L$	Yes	Yes

4.4.2 MTTF models

In the literature three works were found that provide expressions for the MTTF of a memory system suffering from soft errors [37, 38, 39]. These models represent the three scenarios of a memory system vulnerable to SBUs, MCUs or MBUs, as was already shown in Table 4-3. The model presented in [37] is the base MTTF model on which the other models [38, 39] expand. All three models assume that the events occur with a Poisson distribution.

The parameters used by the models are listed in Table 4-4. The error rate λ is the overall error rate of a memory system. When, for example, a memory system has four SDRAM devices and the SEU rate of 10 SEUs/device/day is used that was defined in the previous chapter the error rate is $\lambda = 40$ errors/day.

In case of a MCU or MBU an adjusted error rate λ' is used that to either increase or decrease the original error rate λ depending on the characteristics of the soft error. For MCUs the adjusted error rate is higher than the original rate because MCUs cause more errors compared to SBUs given the the same number of upset events.

Table 4-4: Parameters used in the MTTF models described in [37, 38, 39]

Parameter	Description
λ	Error rate.
λ'	Adjusted error rate.
M	Memory size in number of words.
$p(x)$	Probability that an event induces x errors.
t_s	Scrubber interval.
L	The maximum numbers of errors an event can induces in a single memory word.

No EDAC

To compare the MTTF of a system that is not equipped with an EDAC solution against a system that is protected, an MTTF expression for non-EDAC systems is needed. Such an expression $\text{MTTF}_{no\ edac}$ is given in [37] and shown in Equation 4-20.

$$MTTF_{no\ edac} = \frac{1}{\lambda} \tag{4-20}$$

Where λ is the overall error rate of the sytem. In [37] this equation is actually given as $MTTF = 1/\lambda_{bit} wM$ where w is the number of bits per word and λ_{bit} is the error rate per bit. In this work and in [38, 39] the error rate λ is defined for the whole memory. The total memory capacity of a system is wM which means $\lambda = \lambda_{bit} wM$.

MTTF model for SBUs

The first model, presented in [37], considers that a memory system is protected by a single bit error correcting EDAC ($t = 1$), referred to as Single Error Correction and Double Error Detection (SECDED). Double errors can only be detected and not corrected which is why double errors will still result in a failure.

In case of SBUs, a failure will occur when a memory word is accessed that has two single bit errors caused by two SBUs. The mean time to failure of this scenario MTTF_{SBU} is:

$$MTTF_{SBU} = \frac{1}{\lambda}\sqrt{\frac{\pi}{2} \cdot M} \tag{4-21}$$

In this model of a single error correcting code and SBUs, an EDAC solution will fail if two errors occur in the same memory word. For a memory system consisting of more memory words M and the same error rate λ the probability of two errors in the same memory word decreases resulting in a larger MTTF.

Next, in case a scrubber is added that scrubs the entire memory every t_s, the expression of Equation 4-22 can be used [37]:

$$MTTF_{SBU+scrubber} = \frac{2 \cdot M}{\lambda^2 \cdot t_s} \tag{4-22}$$

Here a failure will also occur when a memory word contains two errors similar to Equation 4-21. The probability of this happening decreases when the time between two memory scrubs t_s decreases meaning the memory is scrubbed more often or when a memory system has more memory words M.

The next two models use the same equations as described above but the term λ is replaced with an adjusted error rate λ'. The adjust error rate accounts for the increased error rate due to MCUs or MBUs. The next two sections will show how λ' is derived.

MTTF model for MCUs

In [38] the model of [37] is expanded to memories suffering from MBUs. The work assumes that the memory cells are interleaved such that errors caused by a single event are all in different memory words. This describes the scenario of an MCU and can therefore be used to model the MTTF_{MCU} and $\text{MTTF}_{MCU+scrubber}$.

The work in [38] determined that a lower bound of the MTTF_{MCU} can be derived by using the model presented in Equation 4-21 with an increased error rate λ'. Where λ' is defined as shown below in Equation 4-23. The same adjusted λ' can also be used to calculate $\text{MTTF}_{MCU+scrubber}$ using Equation 4-22.

$$\lambda' = \lambda \cdot \left[1 + \sum_{j=2}^{\infty} (j-1) \cdot p(j) \right] \tag{4-23}$$

MTTF model for MBUs

When a memory device is susceptible to MBUs causing up to L errors in a single event two types of scenarios have to be considered based on the error correcting capability t of the EDAC solution. In the first case $t < L$ meaning a single event can cause an uncorrectable error directly. Secondly, $t = L$ which means only multiple MBUs can cause an uncorrectable error. The case when $t > L$ is not discussed because this is considered over-engineering. While it is possible that a memory word becomes uncorrectable due to the accumulation of errors from multiple events, it is assumed that scrubbing takes care of this scenario.

- $t < L$:

  In this case the EDAC solution is not able to correct all upsets that are caused by SEUs depending on the number of errors the event causes. The memory will contain an uncorrectable error as soon as an event occurs of $t+1$ errors. Therefore the MTTF_{MBU} can be modelled shown in Equation 4-24 which is based on Equation 4-20.

$$MTTF_{MBU} = \frac{1}{\lambda \cdot \sum_{j>t}^{L} p(j)} \tag{4-24}$$

- $t = L$:

  Another variation of [37] presented in [39] considers that MBUs can introduce a maximum of L errors in a memory word and that the EDAC solution can correct up to L errors. The model is based on [37] given that it considers the a failure to occur when to events occur in the same memory word with the addition that the total number of errors by the two events is bigger than L and therefore t. Similar to the MCU model, an adjusted λ is defined to be used with Equation 4-21 to derive MTTF_{MBU}:

$$\lambda' = \lambda \sqrt{\sum_{i+j>L} p(i) \cdot p(j)} \tag{4-25}$$

4.5 Conclusion

Two different EDAC methods were discussed in this chapter. First, ECCs of which three different types of implementations were considered: a matrix multiplication implementation, cyclic codes implemented using a LFSR and an algebraic solution using BCH codes. The second EDAC method is TMR which uses a majority voter to correct errors. For all implementations an expression for the time and resource complexity was derived.

A introduction to two supplementary EDAC solutions was made. Interleaving, which spreads burst errors across multiple code words and scrubbing to prevent the accumulation of errors.

Finally, several analytical models were introduced that provide the basis for analysis of the improvement in terms of reliability (MTTF) when the proposed EDAC methods are applied to a specific memory system.

Chapter 5

Design considerations

Previous chapters provided background information about SDRAM, the type of soft errors that can be found in SDRAM and how EDAC methods can be implemented in logic. The next step is to determine how the three main components, the MCU, the FPGA and the SDRAM can be combined in a design that allows for these EDAC methods to be implemented.

Section 5.1 will discuss how the MCU, FPGA and SDRAM must be connected. Then, Section 5.2 discusses how the insertion of the FPGA effects the timing of the SDRAM interface. Section 5.3 discusses what EDAC features are implementable in the design of the MCU, FPGA and SDRAM.

5.1 Component layout

First, the layout of the three main components, the MCU, FPGA and SDRAM, in the design has to be determined. Three different layout options have been considered. In the comparison the following aspects are evaluated:

1. **Memory capacity**: How the layout deals with storing the parity bits of the EDAC solution and how this impacts the total usable memory capacity for the MCU.
2. **Error correction**: How the layout allows for correcting data when the MCU reads from the memory.
3. **Memory control**: How the layout allows for the FPGA to control the memory interface to implement features such as scrubbing.

FPGA as memory

In the first layout the SDRAM device is removed from the design and instead the FPGA is used as memory as shown in Figure 5-1. FPGAs have many blocks of SRAM (BRAM)

spread throughout the FPGA fabric which can potentially replace SDRAM. The logic in the fabric can be used to correct any errors when the MCU reads from the BRAM. BRAM can be accessed with less latency compared to SDRAM because it doesn't require precharging or activation of rows of memory cells. This means such commands can be ignored by the FPGA and the clock cycles spend on those commands can be used to perform other operations on the memory such as scrubbing.

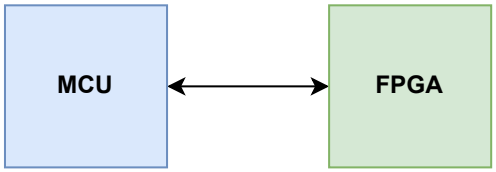


Figure 5-1: Layout with the MCU directly connected to the FPGA with an SDRAM interface. No SDRAM devices are used in this design.

The major drawback of this design is the severely limited memory capacity. When looking at the BRAM capacity provided by small to medium sized FPGAs of Xilinx the typical capacity is between 100 Kb for the lower-end FPGAs and up to almost 100 Mb for the very high-end FPGAs. This is very low compared to the capacity provided by SDRAM chips of which a single device can have up to 512 Mb of memory.

FPGA next to SDRAM

A second option involves placing the FPGA next to the SDRAM as shown in Figure 5-2, attaching the FPGA to the same SDRAM interface that connects the MCU and the SDRAM. Unlike the previous layout which suffered from a low total memory capacity, additional SDRAM ICs can be added to the SDRAM interface to expand memory capacity if more parity bits are needed.

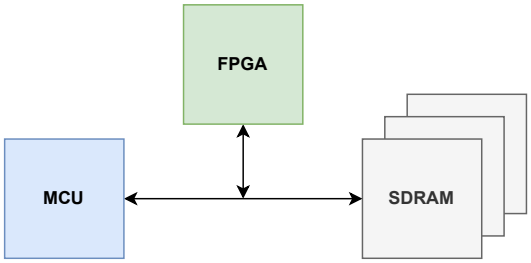


Figure 5-2: Layout with the FPGA next to SDRAM connected to a single SDRAM interface.

The downside of this design is that it's not possible for the FPGA to control the command and address signals of the SDRAM interface. The SDRAM interface is not designed to allow multiple devices driving these signals. This makes it impossible for the FPGA to read or write the memory to perform tasks like scrubbing. For such operations the command and address signals have to be driven by the FPGA.

Correcting errors when the MCU reads from the memory is also difficult in this configuration. When the SDRAM outputs the data of a Read operation, the DQ signals are driven high or low depending on the bit value of the memory word. The FPGA cannot simply overdrive a DQ signal when it detects that a bit of the memory word is incorrect without a complex electrical solution.

FPGA in between MCU and SDRAM

A third approach would be to route all the signals that would normally go from the MCU to the SDRAM through the FPGA as shown in Figure 5-3. The FPGA is placed in between the MCU and the SDRAM with two SDRAM interfaces connecting the MCU and SDRAM to the FPGA. Similar to the previous design the total memory capacity can be expanded by adding more SDRAM ICs to store parity bits. A disadvantage of this approach is that the FPGA needs at least twice as many IO as the previous solutions to implement the two SDRAM interfaces.

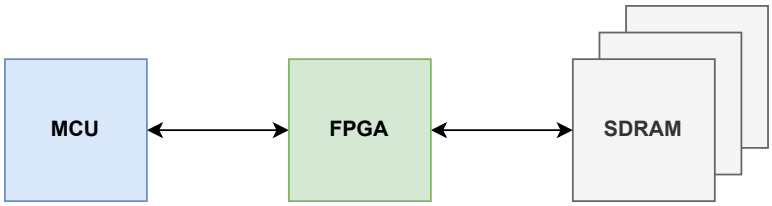


Figure 5-3: Layout with the FPGA in between the MCU and the SDRAM.

5.1.1 Comparison

Table 5-1 show a comparison of the three layouts based on the their characteristics of: memory capacity, memory correction, memory control. FPGA as memory has a big disadvantage because of its low memory capacity and FPGA next to SDRAM is not suitable for EDAC implementations. Therefore, placing the FPGA in between the MCU and SDRAM was chosen for our solution.

5.2 FPGA memory interface architecture

The next step is to analyze how the FPGA must operate such that the SDRAM interface signals going from the MCU to the FPGA are transferred to the SDRAM. The main challenge in this FPGA design is how to deal with Read operations.

As already mentioned in Section 2.4 the SDRAM interface outputs the data of a Read operation after CL clock cycles. Since the SDRAM signals now go through the FPGA, the commands will arrive at the SDRAM with a delay. Further delay is added because of read data going through the FPGA and the latency introduced of any EDAC decoding. The total delay of the Read operation cannot be greater than CL expected by the MCU.

Table 5-1: Comparison of each layout in terms of memory capacity, memory correction and memory control.

Layout	Memory capacity	Memory correction	Memory control
FPGA as memory	Very low capacity	SDRAM interface is routable through FPGA fabric	Complete control
FPGA next to SDRAM	Expendable capacity	Not possible	No control
FPGA in between MCU and SDRAM	Expendable capacity	SDRAM interface is routable through FPGA fabric	Complete control

This problem is illustrated in Figure 5-4 in which the FPGA delays the signals in all directions with 1 clock cycle. In this case the data will arrive two clock cycles too late because the Read command and data were delayed inside the FPGA.

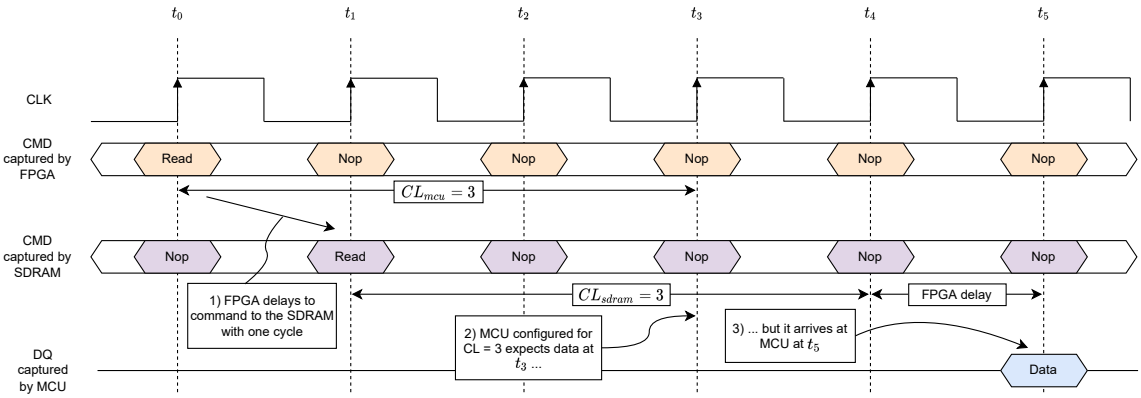


Figure 5-4: Example showing how delaying the SDRAM interface signals in the FPGA cause issues for Read operations. In this case the FPGA delays all signals with 1 clock cycle in both directions. The delay of the Read command and the data causes the data to arrive 2 clock cycles too late at the MCU.

An obvious solution would be to configure the MCU with a higher CL to compensate for the delay introduced by the FPGA. Unfortunately, SDRAM memory controllers are designed to work according to the SDRAM standards which is why they are only configurable for CL values found in those standards. The allowed CL values are 2 and 3 but lower CL values require a lower clock frequency, reducing throughput.

In the logic of the FPGA the memory interface architecture can be implemented with an combinational design or a sequential design. In this section both designs will be analyzed to determine which design is most optimal in solving the Read operation latency problem.

5.2.1 Combinational design

A combinational design only uses logic gates in the path of the SDRAM signals. The advantage of such a design is the issue of the read data arriving one or more clock cycles later as was shown in Figure 5-4 is non-existent as long as the setup and hold time for the capturing Flip-Flops in the MCU and SDRAM are met. Both the launching and capturing Flip-Flops use the same clock so $f_{mcu} = f_{sdram}$.

There are two signal paths groups for which the setup and hold times need to be analyzed shown in Figure 5-5. The first group being the signals going from the MCU to the SDRAM: address, command and data (DQ) for Write operations. The second group consists of the signals going from the SDRAM to the MCU: data (DQ) for Read operations.

A significant amount of delay can be added to the signal paths as their are routed through the FPGA. To meet the setup and hold time of the capturing Flip-Flips it might be necessary to reduce the clock period and therefore the throughput of the MCU memory controller.

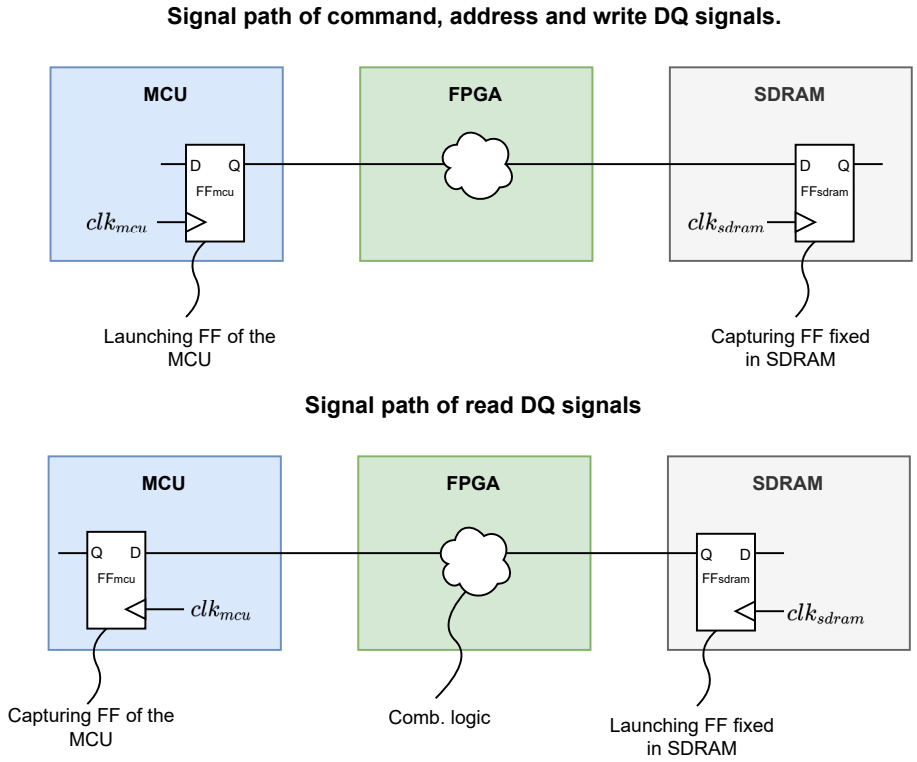


Figure 5-5: Signal path of Read operations of an asynchronous design. Between the launching and capturing Flip-Flips is the combinational logic inside the FPGA.

5.2.2 Sequential design

In a sequential design Flip-Flops are used to registers signals. The SDRAM signals for a synchronous design can be divided into the same two groups as in the combinational design. The major difference now is that one or more Flip-Flops can be added in the paths as shown in Figure 5-6. Its important to note that all command, address and write signals need to have the same number of Flip-Flips in their path because a commands like Activate and Write are accompanied with an address and or write data.

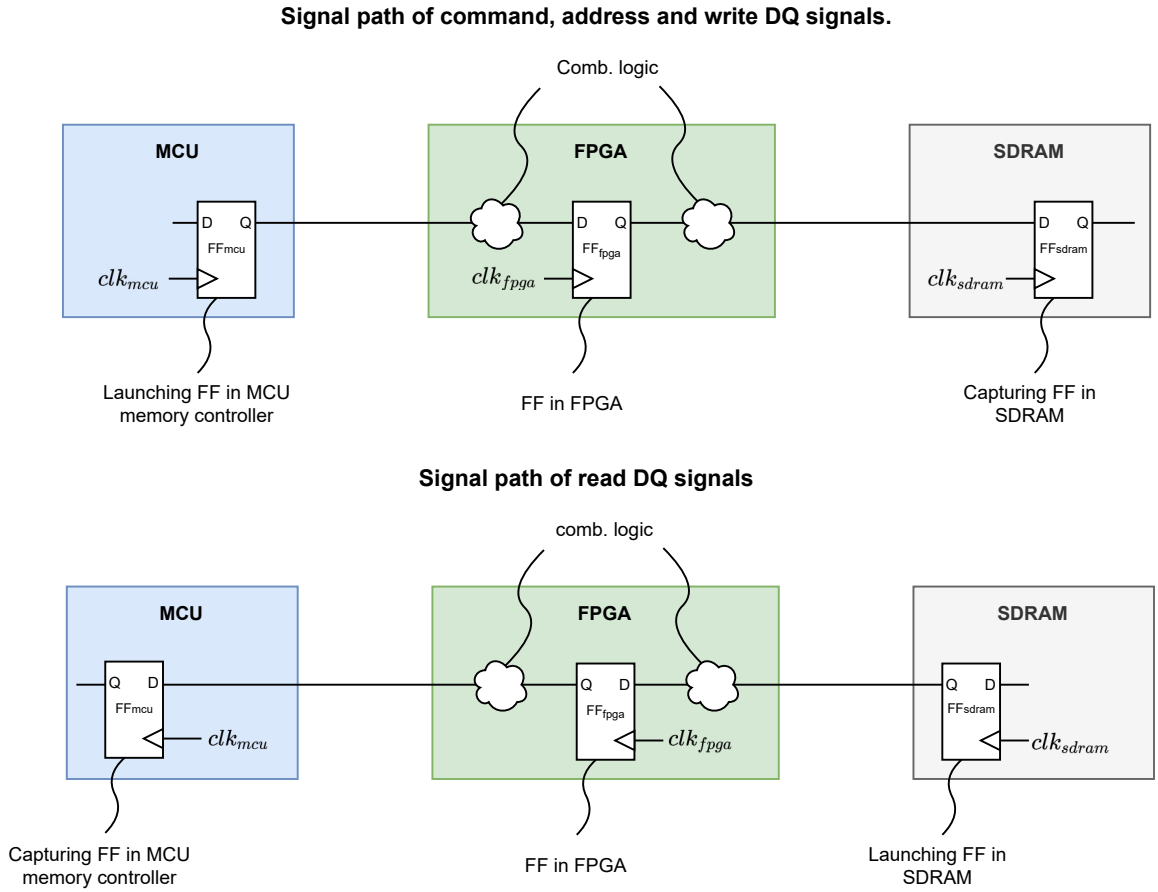


Figure 5-6: Signal path of Read operations of a synchronous design. Between the launching and capturing Flip-Flips is the combinational logic and one or more Flip-Flops (only one shown here) inside the FPGA.

Adding FFs has one major problem as was already illustrated in Figure 5-4. A single FF in the command path results in the Read operation to be delayed with one clock cycle. With an additional FF in the read DQ path the latency of the Read operation as seen by the MCU will be $CL + 2$ instead of CL. Because the MCU cannot be configured to accept a Read latency of $CL + 2$ a solution needs to be found. The next section will explain how using multiple clocks will solve the latency problem.

Multi frequency design

The CL of SDRAM is defined in clock cycles so the CL in units of time depends on the frequency of the clock signal. This can be exploited by using a lower clock frequency at the MCU (clock clk_{mcu}) than at the SDRAM (clock clk_{sdram}) such that the CL in time units of the SDRAM is faster than the CL in time units that the MCU expects. Figure 5-7 illustrates how two clock frequencies with the same CL create a times difference.

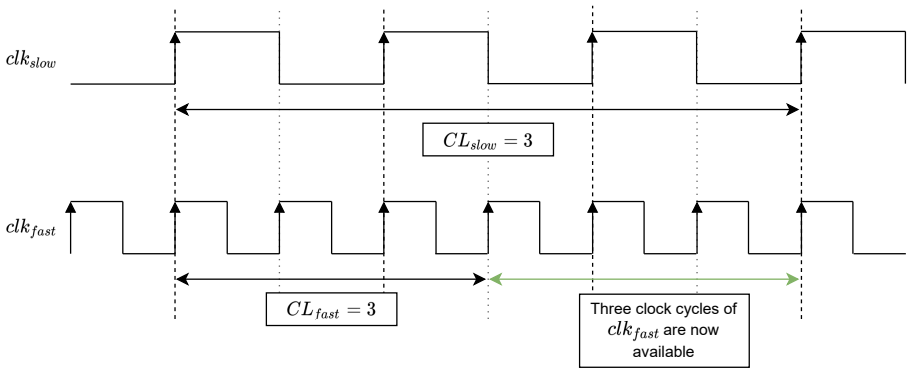


Figure 5-7: Example of two clock signal clk_{slow} and clk_{fast} where $f_{fast} = 2 \cdot f_{slow}$. A $CL = 3$ results in a time difference.

The difference in time created by the two different frequencies can be used such that the MCU has higher latency for Read operations compared to the SDRAM in exchange for a lower memory throughput for the microcontroller. Because the memory clock of the microcontroller has to be reduced to a slower clock frequency the throughput will also decrease. The difference in latency with symbol Δlatency can be expressed in terms of clock cycles of clk_{sdram} as shown in Eq 5-1. The expression is derived from calculating the difference in time of the two CL at different clock frequencies divided by the clock period of the fast frequency (f_{sdram} in this case).

$$\Delta\text{latency}_{clk_{sdram}} = \frac{f_{sdram}}{f_{mcu}} \cdot CL_{mcu} - CL_{sdram} \tag{5-1}$$

An expansion to using two different frequencies is to use a third clock in the FPGA such that the following relations exists: $f_{fpga} > f_{sdram} > f_{mcu}$. Then the latency difference becomes:

$$\Delta\text{latency}_{clk_{fpga}} = \frac{\frac{CL_{mcu}}{f_{mcu}} - \frac{CL_{sdram}}{f_{sdram}}}{f_{fpga}} \tag{5-2}$$

The latency difference Δlatency is equal to the total amount of FFs that can be inserted in the two signal group paths (the command, address and write group and the read group). For example, if Δlatency $= 3$ then 1 FF can be inserted in the path of the command, address and write signal group leaving two FFs to be inserted in the path of the read data.

Note that the latency difference in terms of time is the same for Eq 5-1 and 5-2 is the same but because the $f_{fpga} > f_{sdram}$ the same time period results more clock cycles for f_{fpga} compared to f_{sdram} and therefore more FFs can be inserted which can be beneficial. The downside of this is increased complexity with a design of three different clocks compared to two.

The next two examples will show how the insertion of FFs in the signal paths result in the correct operation of the SDRAM interface because of the latency difference.

Example: two clock frequencies

In this example the MCU and SDRAM are configured with $CL_{mcu} = CL_{sdram} = 3$ and the SDRAM is clocked with a frequency twice as high as the MCU clock $f_{sdram} = 2 \cdot f_{mcu}$. Using Eq 5-1 this design can add a total of 3 FFs of which a single FF is added to command, address and write data signals and the other two are added in the read data path. With the help of Figure 5-8 the timing and states of these FFs will be described to show how the read data will arrive at the MCU at the correct rising edge of the clk_{mcu}.

The Read operation launched by the MCU and captured by the single FF in the FPGA at t_0. Because this FF is clocked with clk_{sdram} the Read operation is captured by the SDRAM one cycle of clk_{sdram} later at t_1. Since the SDRAM has read latency $CL_{sdram} = 3$ the data will be captured three clock cycles of clk_{sdram} later at t_4 by the first FF of the FPGA. Inside the FPGA the read data then captured by the second FF at t_5. Then one clock cycle of clk_{sdram} later the data arrives arrives at the MCU at t_6 which at the rising edge at which the MCU expects the data arrive because of the read latency $CL_{mcu} = 3$ at clk_{mcu}.

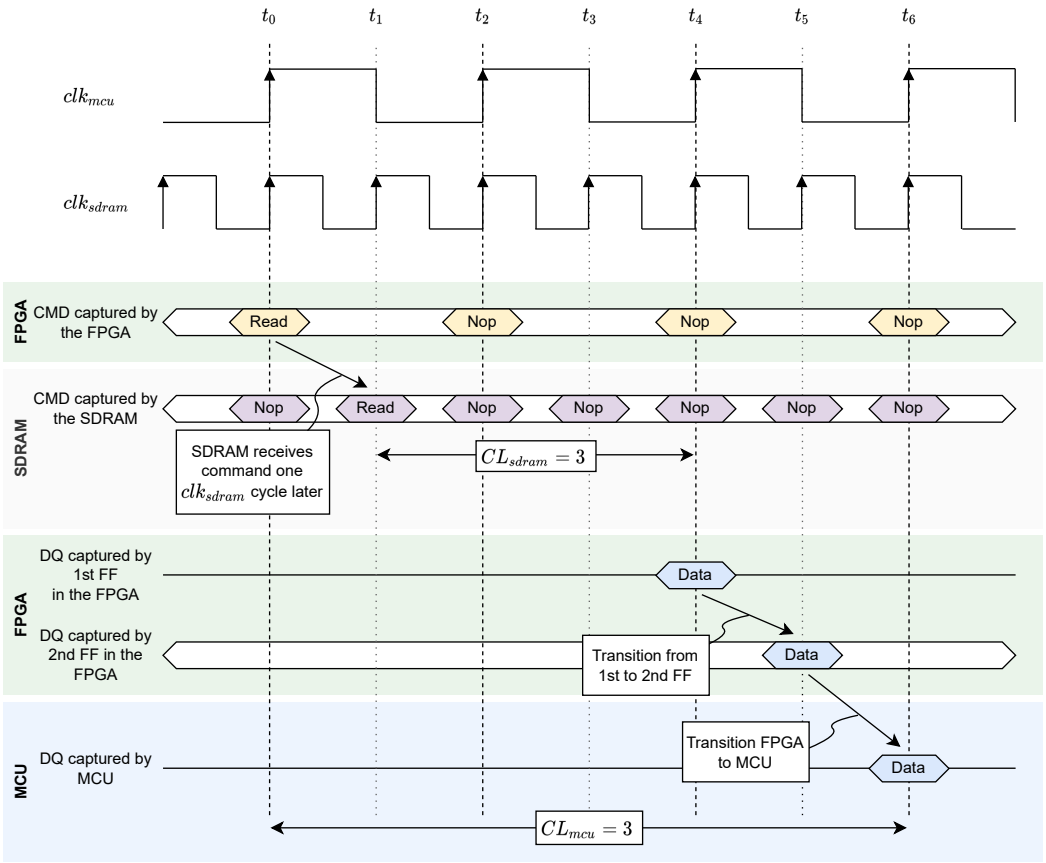


Figure 5-8: Timing diagram showing how a Read operation returns the data to the MCU at the correct rising edge due to the latency difference created by the two different clock frequencies.

Example: three clock frequencies

In this example the read latency is $CL_{mcu} = CL_{sdram} = 3$ and clock frequencies $f_{sdram} = 2 \cdot f_{mcu}$ and $f_{fpga} = 2 \cdot f_{sdram}$. Using Eq 5-2 gives that 6 FFs clocked with clk_{fpga} can be inserted. In this design two FFs are inserted in the command, address and write path. The remaining four FFs are inserted in the read data path. The timing of this design is shown in Figure 5-9

The Read operation passes through two FFs in the FPGA as shown in the upper green box and is captured by the SDRAM at t_2. Then after a Read latency of $CL_{sdram} = 3$ the read data is captured by the FPGA in the first FF at t_8. This data then traverses through the other three FFs in the FPGA read path before being captured by the MCU precisely three clock cycles of clk_{mcu} after the Read operation was captured by the FPGA.

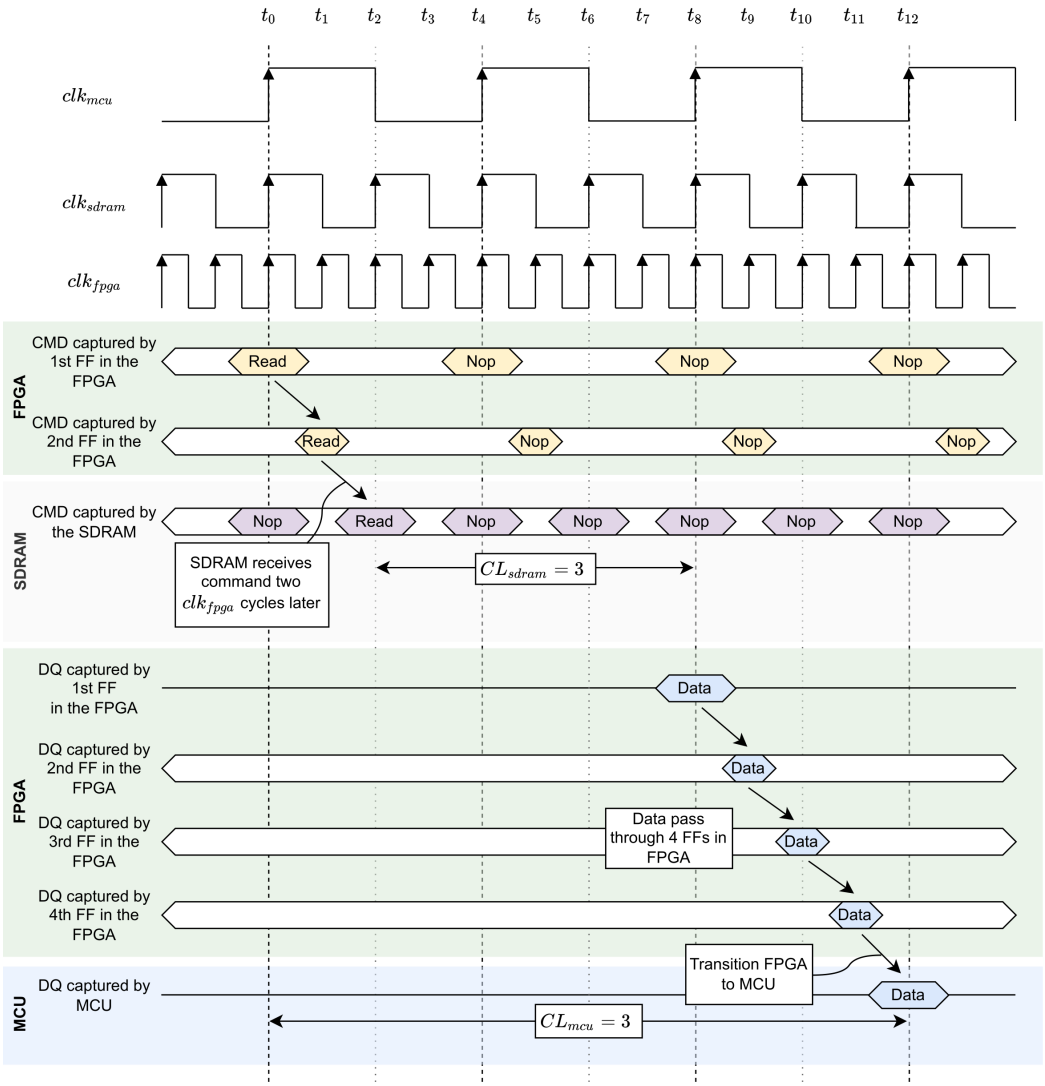


Figure 5-9: Timing diagram showing how a Read operation returns the data to the MCU at the correct rising edge due to the latency difference created by the two different clock frequencies.

5.3 EDAC implementation

This section discusses how the EDAC solutions discussed in Chapter 4 can be inserted in this design. Knowing now that the FPGA is inserted in between the MCU and SDRAM. First the actual error correcting solutions, ECC and TMR are discussed. Then limitations of the design are discussed that prevent some desired features from being implemented. Finally, a solution is given the implementation of a memory scrubber that can periodically scrub the entire memory.

5.3.1 ECC and TMR

In Section 4.1 and 4.2 the working principles of ECC and TMR and their encoder and decoder implementations were discussed. In this section they are compared to determine which solutions are suitable to be inserted in the FPGA memory interface design discussed in the previous section.

1. **Matrix multiplication**. Using a matrix multiplication implemented with an XOR-tree is the fastest method and therefore sacrifices the least amount of throughput of the MCU memory controller because it has a time complexity that scales $\log_2(k)$ for the encoding process and $\log_2(n)$ for the decoding process.

   The resource usage of a this solution is very small. A $n = 64$ and $k = 32$ code word requires a total of 3,072 XOR-gates to encode and decode using Eq 4-11 and Eq 4-6 which is well within the range of even most low-end FPGAs.

   For larger n and higher capable code words ($t \geq 2$) the lookup table will become very large. A compromise would be to only store the error pattern of burst errors reducing the size of the table. Although, without knowledge on the internal layout of the SDRAM chip it is difficult to determine which burst patterns can occur.

2. **Cyclic codes**. While the LFSR allows for an efficient implementation in terms of resources, it is not a suitable method because it requires n clock cycles in both the encoding and decoding stage. To be able to insert n FFs in the read and the write path requires a sacrifice in the clock frequency of the MCU memory controller which makes it a bad design choice.

3. **BCH**. Using the algebraic BCH decoder is only recommended when lookup table of matrix multiplication method is no longer implementable due to its size but an memory efficient ECC solution is still required. A disadvantage is that the implementation is complex and requires t FFs in the second part of the decoding and two more for the first and the second part to reduce the critical path.

4. **TMR**. It is a good solution when a high error correcting capable design is required without comprising in throughput of the MCU memory controller and if adding 200% more memory to store the TMR copies is acceptable.

5.3.2 Limitations

There are more limitations to the EDAC solution in addition to the complexity of the implementation of the ECC encoders and decoders. These limitation are described in this section. For some limitations a solution is proposed:

SEFI correction

In Section 3.2.3 SEFIs were classified into Transient and Persistent SEFIs. For both types the first problem is the detection of a SEFI. If a single SDRAM device is used and a wrong column or row is accessed that has been written previously, the wrong code word will be read but it will still be a valid one, making it impossible to detect an error.

A system using multiple SDRAM devices is still problematic because of the amount of errors that can occur in a single memory word read from the wrong column or row. This is illustrated with an example in Figure 5-10 in which a wrong row is activated in one of the two SDRAM devices. The code word can contain many errors because the data part of the code word originates from a different code word. However, because the SEFI can be detected in this case the microcontroller can be notified that an uncorrectable error has occurred.

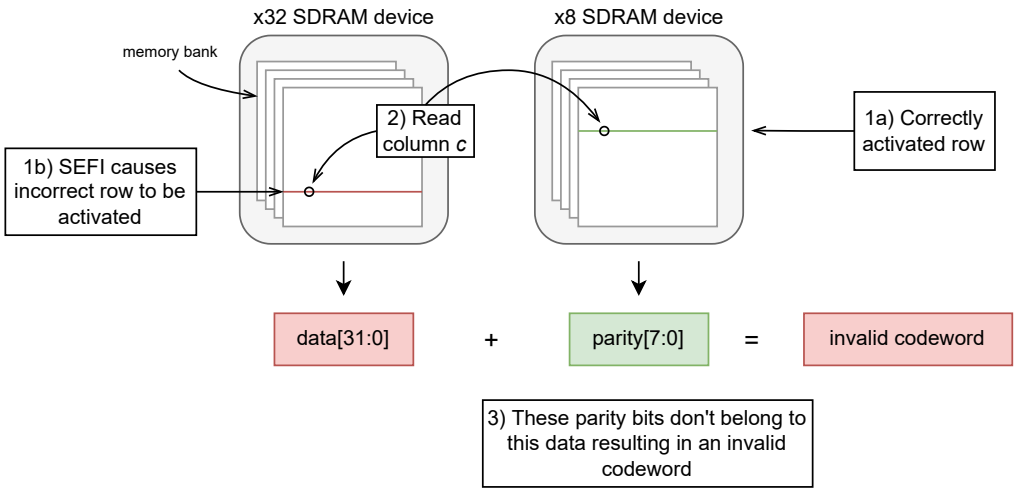


Figure 5-10: A SEFI caused the wrong row to be activated in the left SDRAM device 1). When the same column is read 2) from both SDRAM devices the data bits and the parity bits don't combine into a valid code word with many errors 3).

Moreover, when an error is detected in a memory word it is more likely to be cause by a SBU or MCU. Determining that the errors are caused by a SEFI requires reading multiple memory words, if many consecutive words are corrupted it's likely to be caused by a SEFI. However, in the mean time the corrupted memory words will have arrived at the MCU resulting in a failure.

Using a periodic Mode Register reset to prevent Permanent SEFIs as discussed in Section 3.2.3 is a method that is implementable in theory. According to the JEDEC standard it's allowed to reconfigure the Mode Register but after a Mode Register command no other commands

must be send before t_{RMD} has passed [11]. Such a periodic reset could be incorporated in the memory scrubber that will be discussed in Section 5.3.3.

Read-Modify-Write operations

Processors have byte sized read and write instructions. Therefore, SDRAM memory support byte sized write operations by masking the bytes of a memory word that don't need to be written or written. This is problematic because rewriting a single byte that is part of a multi-byte ECC code word causes the parity bits become invalid. This problem is also referred to as Read-Modify-Write operation (RMW) [40] because a Read operation is required to modify the parity bits and perform the Write operation. The problem of RMW operations is illustrated in Figure 5-11. A single byte of a 40 bit code word is written causing a mismatch between the parity bits and data bits, invalidating the code word.

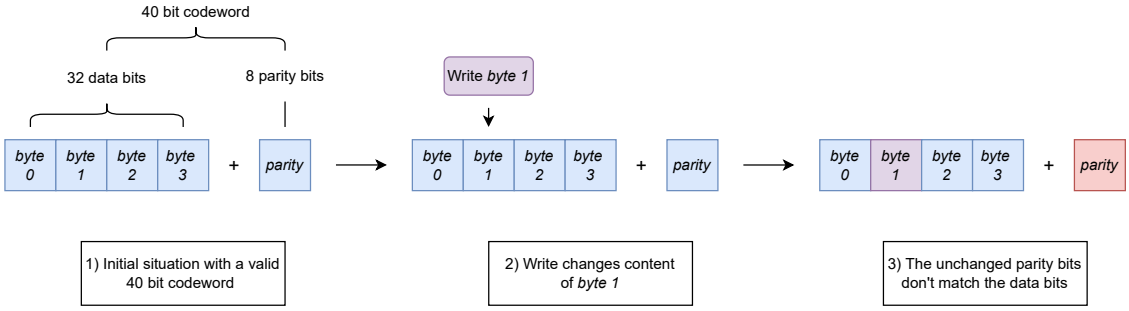


Figure 5-11: Example of a RMW scenario in which a single byte of a 32 bit memory word + 8 parity bits is written. Initially 1) the parity bits correspond to the 32 data bits. Then 2) a single byte of the code word is re-written. This causes the code word to become invalid 3) because the parity bits don't correspond to the new 32 bit data.

Recalculating the parity bits requires an additional Read operation which takes up CL clock cycles. A MCU memory controller does not assume such a large latency of a Write operation so it can perform other memory accesses that will collide with the RMW operation.

There are two ways this problem can be dealt with:

- **Byte-sized ECCs.** The problem of RMW operations can be eliminated by creating a codeword for each individual byte ($k = 8$) of a memory word instead of encoding the entire memory word. The disadvantage of this approach is that smaller code words have a lower code rate meaning more memory is needed for the parity-bits.

- **Cache write-miss policy.** Another method that can force word-sized memory operation is related to the cache write-miss policy. If the cache supports Write-Allocation and a byte-sized write is performed the cache line will be updated and the write will not be immediately performed to the external SDRAM. When the cache line is replaced a whole memory word write operation to the external SDRAM is done instead of a byte-sized write.

Write-back of corrected memory words

Preferably, the FPGA writes back the corrected value of any errors detected in memory words during a Read operation. This immediately removes the error within the SDRAM lowering the possibility of the accumulation errors in a memory word.

The most obvious solution would be to write back the corrected memory word after the Read operation. This requires that the MCU does not access the memory when a write-back needs to be done. The problem is that it cannot be guaranteed that the MCU performs no other operations that will collide with the Write operation. For example if the MCU only performs Read operations before closing the row then there is no time to write back any corrected memory word. Thus removing errors in SDRAM by writing back the corrected value can only be done by scrubbing the memory which will be discussed in Section 5.3.3.

Interleaving

In Section 4.3 it was discussed that interleaving is an effective method to prevent MBUs caused by a single event. Interleaving memory data requires that parts of a memory word are distributed in a different address within a SDRAM device or across different SDRAM devices. However, the implementation of both is challenging as will be explained below. As a result interleaving is not recommended for implementation in the FPGA design aimed in this thesis.

1. **Interleaving within a SDRAM device**. With this approach a memory word is spread across multiple columns, rows or banks within a SDRAM device decreasing the likelihood that the bits of a memory word will be located physically close to each other. An example of this is shown in Figure 5-12 in which 4-bit memory words are interleaved across multiple columns within a row.

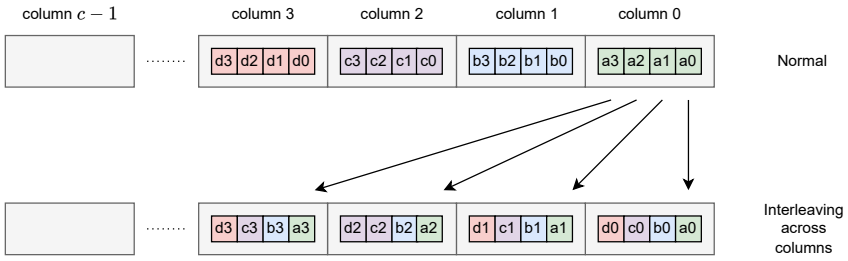


Figure 5-12: Example showing how a 4-bit memory word is interleaved across four different memory words in the same column.

   This method cannot be applied is because it requires multiple Read and Write operations for each memory access by the MCU. The example of Figure 5-12 requires 4 operations to the SDRAM for each memory access by the MCU which is not implementable with the current design.

2. **Interleaving across SDRAM devices**. An alternative is to spread the bits of a code word across multiple SDRAM devices but in the same address for each SDRAM device such that an access by the MCU requires only one Read or Write operation. The disadvantage of this approach is that it requires a very high number of SDRAM

devices in order to sufficiently interleave a code word such that only MBU with a low multiplicity can occur.

An extreme example of memory interleaving is IBM's Chipkill which spreads a 72-bit code word (capable of correcting a single bit) over 4 DIMMs each containing 18 SDRAM devices [40]. Such high amount of SDRAM devices is very impractical for platforms in which MCUs typically operate.

5.3.3 Scrubbing

With the layout configuration chosen in Section 5.1 a fully customized logic can be added to the FPGA to gain full control of the SDRAM sub-system. This provides the opportunity to add a scrubber to design.

The scrubber's task is to perform periodic scrub-cycles in which one or more memory words are read, any errors present in those memory words are corrected (if possible) and the corrected words are written back to the SDRAM. The complete set of commands that need to be executed by the scrubber is as follows:

1. **Precharge**. A precharge is needed to close the row that is currently open.
2. **Activate**. Activates the row that will be accessed during the scrub cycle.
3. **Read**. Perform one or more read operations as part of the scrub cycle
4. **Write**. Perform one or more write operations as part of the scrub cycle.
5. **Precharge**. Close the activated row.
6. **Activate**. Activate the row that was open before the scrub cycle.

The six listed commands required more than six clock cycles because of the CL of the Read command, additional Nop commands between the Precharge and Activate commands to meet the t_{RCD} requirement and between the Activate and Read command to meet the t_{PR} requirement. If the t_{RCD} and t_{PR} periods take two clock cycles a complete scrub cycle will take 12 clock cycles. Another issue is that the scrub-cycle cannot be interrupted by the SDRAM commands from the MCU because the scrubber has activated different rows.

Scrubbing during refresh-cycles

The solution to this problem can be found by manipulating the Refresh commands send by the MCU to the SDRAM. When a Refresh command has been send to the SDRAM only Nop commands are allowed for at least t_{RFC} as shown in Figure 5-13. Another great property of the Refresh command is that it must be proceeded with a Precharge (all banks) command meaning the scrubber doesn't have to issue this command and doesn't have to activate the row that was active before the scrub cycle saving a lot of clock cycles.

Typically SDRAM needs to be completely refresh every 64 ms. If an SDRAM has 8,192 rows per bank this means a Refresh command must be send every 7.8 μs. Analyzing several MCUs showed that the refresh rate is generally configurable allowing a higher refresh rate than needed for the SDRAM. The redundant refresh cycles can be used by the FPGA to access the SDRAM without being interrupted by other SDRAM commands send by the MCU.

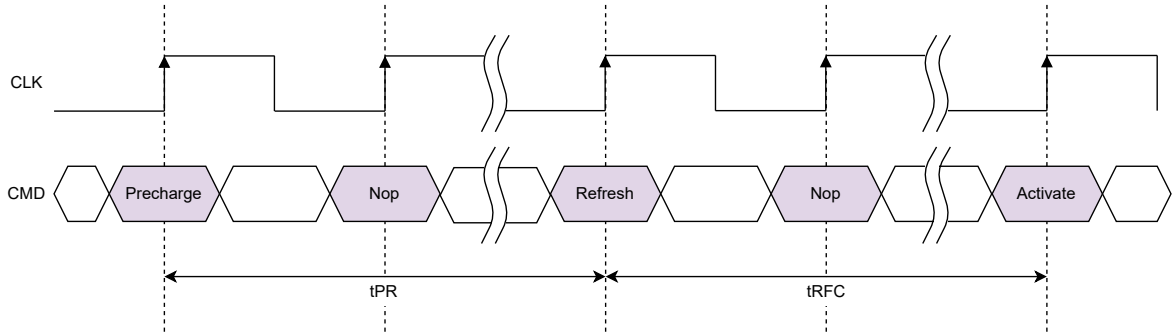


Figure 5-13: Timing diagram showing the only Nop commands (No operation) can be send to SDRAM for at least t_{RFC}. Before a Refresh operation a Precharge operation must be done which takes t_{PR} time to complete.

Figure 5-14 shows the commands that are issued when a Refresh operation issues by the MCU is used to perform a scrub cycle instead. The Refresh operation is replaced with an Activation operation to engage the row that needs to be scrubbed. Then a column is read and rewritten if an error is detected and corrected. Finally, all banks are Precharged and after t_{RP} the MCU can issue an Activation command.

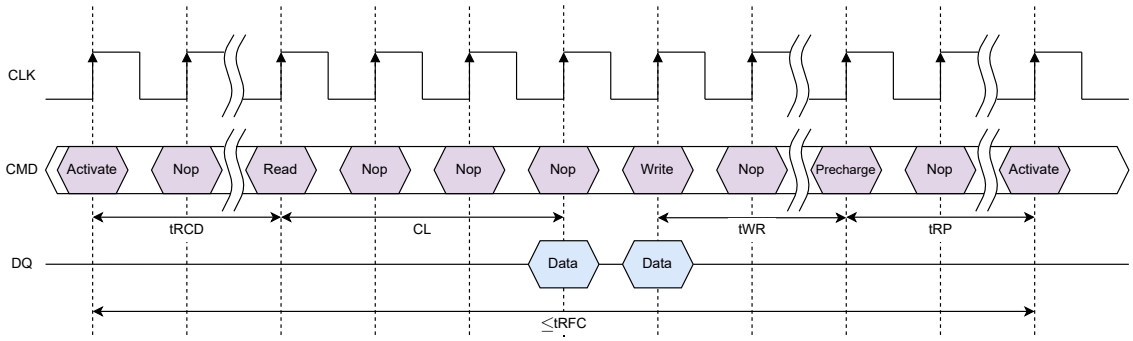


Figure 5-14: Timing diagram showing a scrub cycle being performed in which one memory word is read and the corrected version is written back. The Activation operation at the start replaces the Refresh operation command. The scrub cycle must be completed within t_{RFC}.

5.4 Conclusion

The first part of this chapter determined that the most optimal layout for the MCU, FPGA and SDRAM is to route the SDRAM interface from the MCU through the FPGA to the SDRAM. This way more SDRAM devices can be added and all the signals go through the FPGA logic giving the FPGA complete control.

The delays introduced by the FPGA and the delays of the EDAC implementation increase the latency of a Read operation. The MCU can not be configured to except a larger read latency

so a solution is to either decrease the clock frequency or to use multiple clock frequencies. The latter allows for the inclusion of Flip-Flops in the FPGA logic. In all cases the design will reduce the throughput of the memory controller of the MCU.

In the last part of this chapter the different suggested EDAC implementations were compared to derive the use cases of the implementation. Some limitations of the EDAC implementation in the FPGA are discussed as well. Finally, a method for memory scrubbing was described that abuses the Refresh operation to allow the FPGA to access the memory freely.

Chapter 6

Methodology

This will introduce a methodology for customized FPGA based error correcting for COTS microcontrollers and their main off-chip memory with the goal of increasing the reliability of a satellite. Section 6.1 describes the approach taken by the methodology. The remaining Sections 6.2 to 6.4 discuss the processes of the methodology.

6.1 Methodology approach

The previous chapter discussed the design options that have to be considered when implementing an FPGA based EDAC solution for external memory. However, what hasn't been defined until now is how an optimal design of an FPGA based EDAC solution can be chosen for a MCU and its SDRAM that operate in a radiation environment such that the reliability improves.

When used, the methodology guides towards a design that improves the reliability of the memory to a desired level while making sure that the system does not become over-engineered in terms of reliability with the consequence of lowered memory capacity, latency and memory throughput.

The methodology is described in three steps. First an analysis step, followed by a design exploration and finally the implementation of the design. The motivation for the three steps are described below:

1. **Mission and electronics platform analysis**

   First, the constraints of the design have to be defined using an analysis. These constraints are used to ensure that the EDAC and FPGA design is capable of providing the required level of reliability while minimizing the costs of the implementation.

   The constraints are defined by performing an analysis on the mission of the satellite and its electronics. In the mission analysis the reliability requirement is defined expressed in a MTTF based on, for example, the type or duration of the mission. Using the MTTF

requirement and the MTTF models the error correcting capabilities needed to reach the required reliability can be derived. The electronics analysis is used to define the specifications of the microcontroller's memory controller and the SDRAM such as the required memory capacity and the data bus with of the SDRAM interface.

2. **Design space exploration**

   Given the constraints and specifications defined in the previous step the actual design of the FPGA, the EDAC solution and the memory configuration can be determined. This step consists of several parts. Each part touches several key design aspects that need to be explored in order for the eventual implementation to provide the required reliability.

3. **Implementation**

   The last step is the implementation of the design proposed in step 2. Besides implementing the chosen design some additional control logic, for example to control the bi-directional data bus of the SDRAM interface, is required which is independent of the EDAC solution. Some extra features are also considered in this step which are useful for testing purposes and for the validation of the reliability.

A flow diagram containing all of the steps is shown in Figure 6-1. The steps are described below and their content is discussed in detail in the next sections.

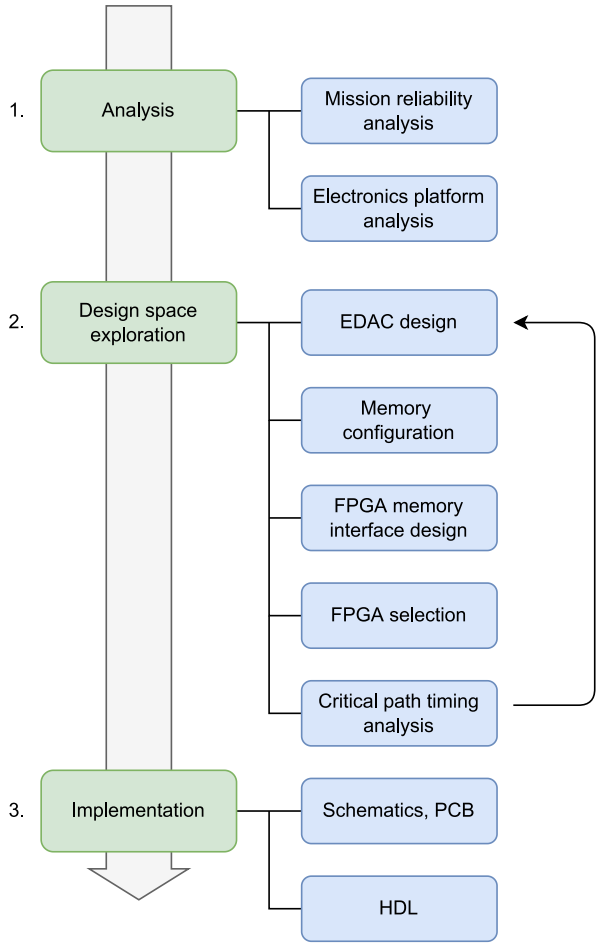


Figure 6-1: Overview of the steps defined in the methodology.

6.2 Analysis

The first step in the methodology is to define the reliability requirement and the specifications of the electronics platform (the MCU and memory). These properties are needed in the second step to constrain the design. There are two different analysis steps that must be performed to define these properties.

The first step focuses on the mission of the satellite to define the required Mean Time To Failure of the MCU and its memory. Given this MTTF, the capabilities of the EDAC solution can be derived with the use of the MTTF models discussed in Section 4.4.2.

The second step analyses several parameters of the MCU and the memory. These parameters are used to determine how the required EDAC solution can be implemented for the specific MCU and memory sub-system.

6.2.1 Mission reliability analysis

The mission reliability analysis involves completing a questionnaire about the mission of the satellite and applying a MTTF model. The end result of this step is the definition of the capabilities of the EDAC solution that must be implemented.

Mission questionnaire

To guide the user of this methodology towards an optimal design requires the user to set a certain level of reliability expressed as a MTTF. However, defining a MTTF for the memory system is difficult. Therefore, in this methodology a questionnaire is included that guides towards a MTTF based on several aspects of a CubeSat's mission in which the MCU and memory operate. The questionnaire consists of several categories. Each category represents a key aspect of the satellite mission and can therefore be used to determine the desired MTTF of the MCU and its memory.

The following assumption is made. The final MTTF value is any MTTF_x that is lower than the $\text{MTTF}_{mission}$. If all MTTF_x are higher than $\text{MTTF}_{mission}$ then the final MTTF value is $\text{MTTF}_{mission}$. The reasoning behind this is that the reliability does not need to be higher than the duration of the mission. This is considered over-engineering.

The questionnaire categories are described below. For each category a description is given and how the MTTF of that category must be defined if it is applicable to the satellite and its mission.

- Mission duration: $\text{MTTF}_{mission}$

  A satellite mission is designed for a certain mission duration $t_{mission}$. This parameter can be used as an upper bound of the MTTF. A design with a higher reliability than the duration of the mission is considered to be over-engineered at the cost of performance and increased design complexity.

  Define $\text{MTTF}_{mission} = t_{mission}$

- System power: $\text{MTTF}_{powered\ on}$

  In some cases CubeSats are too small to include a battery to store energy that can be used to operate the electronics when the satellite is not exposed to sunlight. Such satellites reboot each time the solar panels are exposed to sunlight again and the electronics are therefore only active for periods of $t_{powered\ on}$ and deactivated for periods of $t_{powered\ off}$.

  Define $\text{MTTF}_{powered\ on} = \text{MTTF}_{mission} \cdot \dfrac{t_{powered\ on}}{t_{powered\ on} \cdot t_{powered\ off}}$

  Additionally, a reboot of the microcontroller will also automatically prevent the accumulation of errors because the memory will be reloaded removing any existing errors, essentially creating a scrubber. Therefore, the scrubbing interval t_s can be set to $t_{powered\ on}$.

- Visible ground station passes: $\text{MTTF}_{ground\ station}$

  A critical part of a satellite's mission can be downloading the data generated by the satellite payload. A satellite will not always be connected to a ground station due to a limited number of ground stations. When the satellite is visible to a ground station, it is important the CubeSat is available to upload or download any data.

  The reliability can be derived from the number of ground stations n_{passes} in which data must be reliably transferred. The $\text{MTTF}_{ground\ station}$ can then be derived from n_{passes} and the frequency $f_{ground\ station}$ at which the satellites connects to a ground station.

  Define $\text{MTTF}_{ground\ station} = n_{passes} \cdot f_{ground\ station}$

- Propulsion: $\text{MTTF}_{propulsion}$

  Some CubeSats have a propulsion system to change or correct their orbit. Fuel for these systems is limited requiring increased reliability to prevent the need for additional burns (orbital maneuver) to correct any wrong missed/burns. The required number of successful burns n_{burns} in a time period $t_{burn\ period}$ can be used to determine a $\text{MTTF}_{propulsion}$.

  Define $\text{MTTF}_{propulsion} = n_{burns} \cdot t_{burn\ period}$

- MCU task: MTTF_{task}

  A satellite can contain multiple MCUs each used for a specific task. Commonly, MCUs are used as Command Handling and Data System (CHDS), Attitude Determination Control System (ADCS) or as a platform to control he payload of the satellite. Generally, the CDHS controls and monitors the ADCS. Subsequently, the payload can depend on the ADCS to properly orientate the satellite. This means we can classify the reliability of the MCU task as strong, medium and weak for the CDHS, ADCS and payload respectively.

  Parameters that influence the required MTTF for the task that the MCU performs are:

  - CDHS (strong reliability):

    Often this is the most important component as it controls the ADCS and payload. Because the CDHS is so important it is reasonable to define the MTTF_{CDHS} as a percentage α_{CDHS} of the mission duration $t_{mission}$. For example, if the CDHS must remain operational 80% of the time use $\alpha_{CDHS} = 0.8$.

    Define $\text{MTTF}_{CDHS} = \alpha_{cdhs} \cdot t_{mission}$

  - ADCS (medium reliability):

    Operations of the satellite can depend on proper orientation. Antenna orientation can be important or if a satellite only has solar panels on one side. Defining how long the ADCS must be able to maintain proper orientation $t_{orientation}$ is an MTTF indicator.

    Define $\text{MTTF}_{ADCS} = t_{orientation}$

  - Payload computer (weak reliability):

In this case the MTTF depends on the reliability of the service the satellite provides. Relevant is the how often the payload is turned on how long operates. More on this in the next category.

- Mission type: $\text{MTTF}_{mission\ type}$

  The reliability can also be determined based on the mission type that the CubeSat is performing. Common mission types are listed below and for each type an indication to how the requirement of the reliability can be determined:

  - Telecommunication (strong reliability):

    Telecommunication requires high reliability to provide quality service. The reliability of telecommunication missions can be based on the required uptime of the communication service.

    Define $\text{MTT}_{mission\ type} = t_{uptime}$

  - Earth observation (medium/weak reliability):

    Earth observation often involves monitoring a more slow environment allowing for weaker reliability. Determining a MTTF can be done by defining the minimal duration of a measurement without errors $t_{measurement}$.

    Define $\text{MTT}_{mission\ type} = t_{measurement}$

  - Astronomy (weak reliability):

    Astronomy is classified as weak reliability because monitoring orbital bodies requires a lower sampling rate. Similar to the Earth observation mission type, the measurement duration $t_{measurement}$ can be defined.

    Define $\text{MTT}_{mission\ type} = t_{measurement}$

  - Technology demonstrator (custom reliability):

    In this case any of the above can apply when a satellite is used to demonstrate new telecommunication or sensing technology for earth observation or astronomy.

When the questionnaire is followed, multiple categories might be relevant for a satellite. The final MTTF value $\text{MTTF}_{required}$ that must be selected is the highest MTTF from these categories. If this MTTF is higher than $\text{MTTF}_{mission}$ then $\text{MTTF}_{required} = \text{MTTF}_{mission}$. The $\text{MTTF}_{required}$ is the required reliability of the satellite and is used as a constraint of the FPGA design in the next steps of the methodology.

MTTF model application

With a MTTF value set by the mission questionnaire it is possible to determine what capabilities the EDAC solution must have in order to achieve this MTTF. This is done by applying an MTTF model.

In Section 4.4.2 three models were give to estimate the MTTF of SDRAM susceptible to SBUs, MCUs and MBUs. The sensitivity to upsets can not be easily estimate but Section

3.2.1 concluded that an upper limit of 10 SEU/device/day for spacecraft operating in LEO can be safely assumed.

At the same time it is also difficult to say something about the multiplicity of these upsets and how many errors a single event MBU can cause. The sensitivity to MCUs or MBUs depends on the internal layout of the SDRAM which can only be determine with expensive equipment as SDRAM vendors don't offer this information as was concluded in Section 3.2.2.

To apply any of the three MTTF models it must first be determined how many errors L a single event can cause in a memory word. This leaves us two options:

1. The internal SDRAM layout is known:

   In this case L can be estimated based on how many cells in a memory word are physically located next to each other. For example, the 8-bit SDRAM device shown in Figure 3-3 has a $L = 2$ because its cells are paired together.

2. The internal SDRAM layout is unknown:

   This is a non-ideal, however quite realistic case because a high value for L must be chosen to be safe. Later, it will show that a higher L requires a higher error correcting capability t to achieve the same MTTF. If the L is set too high the solution will not be optimal because it will result in an over-engineered EDAC solution. However, in the worst case it must be assumed that all memory cells of a memory word are physically close meaning a single event can cause many errors in the same memory word giving a large L.

Finally, based on L the MTTF model and an error correcting capability value t can be chosen. Following the steps described below will result in a value for t and possibly t_s (if needed) to reach the required MTTF:

- $L = 1$:

  This means a single event can only cause a single error in a memory word. However, MCUs are still likely to occur but they cause multiple errors in multiple words. The model expressed by Equation 4-23 considers this scenario by account for the increased number of errors per event.

- $L > 1$:

  When a single event can cause multiple errors in a memory word the MTTF depends on the error correcting capability t of the EDAC solution, L and multiplicity proability $p(x)$. Determining t is an iterative process starting at $t = 1$. Increment t until it is high enough to provide the required MTTF. The MTTF model is selected based on the following relationship between t and L and the SEU multiplicity $p(x)$ defined in Section 3.2.2.

  - $t < L$:

    A single event MBU with multiplicity $t < x \leq L$ causes an uncorrectable error. Therefore the MTTF can be modeled as SBU with adjusted rate $\lambda' = \lambda \cdot \sum_{i=t+1}^{L} p(i)$

and Equation 4-20. The adjusted rate only considers MBUs that cannot be corrected.

In theory this also means that implementing a scrubber has no effect. However, if L was determined without knowing the internal SDRAM structure, it is still advised to add a scrubber.

- $t = L$:

  In this case all single event upsets can be corrected except when multiple events accumulate. The model of Equation 4-25 has to be used in this case because this accounts for two MBU event accumulating to an error greater than t.

6.2.2 Electronics platform analysis

The second step of the analysis that needs to be performed focuses on acquiring the relevant parameters of the original electronics platform: the MCU and the memory. These parameters are needed in the second step of the methodology to determine the FPGA and memory design. The following parameters must be defined:

1. Memory specifications:

   M_{cpu} Define the memory capacity that must be available to the processor.

   D_{max} The maximum number of SDRAM devices that can be used.

2. MCU and memory controller specifications:

   d_{mcu} Data bus width of the MCU SDRAM interface. Usually MCUs can be configured for 8, 16 or 32-bit wide SDRAM interfaces.

   t_{RFC} The maximum configuration of the t_{RFC} timing parameter in the memory controller expressed in clock cycles. During this period the MCU will not access the SDRAM.

   $t_{refresh}$ Time between refreshes must be configurable to be able to implement a scrubber. The scrubber requires a lower $t_{refresh}$ compared to a normal memory design.

   clk_{mcu} Define the frequencies that clk_{mcu} can be configured to. In all design options the clock must be lowered so have a flexible clock source is helpful.

   Cache policy Define whether the cache supports Write-Allocation for Write misses when writing to the data cache.

6.3 Design space exploration

Now the parameters have been collected, it is possible to move on to the second step and construct the design. The end result is the configuration of all components of the complete memory system including the FPGA as shown in the overview in Figure 6-2. This includes the configuration of the memory controller of the MCU, the EDAC implementation in the FPGA and the memory configuration of the SDRAM devices.

The design exploration consists of five parts. The first three parts focus on the configuration of the design by defining a set of rules and guidelines. When applied, these will result in a design capable of achieving the required reliability given the set of parameters defined in the previous section. The fourth advises on the selection of FPGA and the fifth part evaluates whether the implementation is feasible by performing a critical path timing analysis.

If the evaluation in the last part concludes the design is not implementable with the current configuration the design space exploration has to be redone by relaxing some of the constraints or selecting a different FPGA with better timing properties. This iterative process can be performed until a working solution is found.

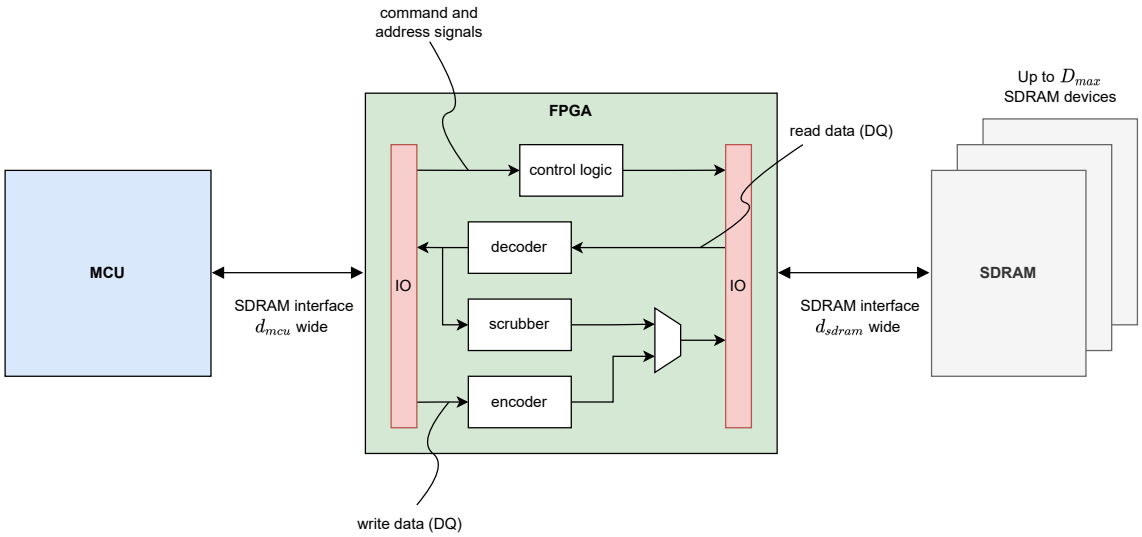


Figure 6-2: Overview of the memory design with the addition of the FPGA in between the MCU and the SDRAM. Shown in the overview are the two SDRAM interfaces the FPGA connects to, one interacting with the MCU and one interfacing with the actual SDRAM.

A. EDAC configuration

The EDAC configuration is a critical part of the design as it is the part that ensures the desired reliability can be achieved. The first part consists of selecting the number of parity bits given the required error correcting capability t and the implementation of the encoder and decoder. For this process the following steps must be followed:

1. From the reliability analysis an error correcting capability requirement t is defined. To be able to correct t errors requires a certain amount of parity bits $n-k$ to achieve a minimum distance $d_{min} \geq 2t+1$.

   The number of parity bits $n-k$ can be derived using the BCH code construction method. This requires the number of message bits k per code word to be determined first which of the RMW problem. If your cache support Write-Allocation for write-misses to the case, code words of size $k = d_{mcu}$ can be used. Else, $k = \lceil \frac{d_{mcu}}{8} \rceil$ because a code word needs to be created for each byte in the d_{mcu}. In case a TMR method is used the number of parity bits is $n - k = 2 \cdot d_{mcu}$.

   For convenience a set of pre-calculated code word sizes of n and k are listed in Table A-1 to A-4 for values error correcting capabilities up to $t = 4$.

2. Select how the encoder and decoder will be implemented. If your processor doesn't support this then byte sized code words must be used ($k = 8$) and an encoder and decoder is required for each byte in d_{mcu}. For encoding, the use of a matrix multiplication is always recommended. When TMR is used this no encoding is required.

   For decoding there are two options: a matrix multiplication with an error pattern lookup table or an algebraic BCH decoder. The matrix multiplication is fast but is limited by the size of the lookup table. Included in Table A-1 to A-4 are the sizes of the lookup tables for range $t = 1$ to $t = 4$. For the BCH code $k = 32$ and $t = 2$ and all codes of $t > 2$ the minimum lookup table needs more than 40 kB of memory which starts becoming a problem for most FPGAs. There are FPGAs with megabits of memory but those are in the high-end segment and overkill for this type of application. Therefore an algebraic BCH decoder is recommended starting from $k = 32$ and $t = 2$ codes.

Next the configuration for the scrubber can be determined. The error correction performed by the scrubber can re-use the decoders inserted in the data read path of in the FPGA because the memory cannot be accessed by the MCU during a scrub cycle. Configuring the design of the scrubber consists of the following steps:

1. Determine the maximum number of memory words accessed during a scrub cycle based on the maximum configurable value of t_{RFC} using Eq 6-1 below. All timing parameters must be expressed in terms of clk_{sdram} clock cycles. When using a sequential design in which $\frac{f_{sdram}}{f_{mcu}} > 1$ then number of memory operations that can be performed by the scrubber is not t_{RFC} but $t_{RFC} \cdot \frac{f_{sdram}}{f_{mcu}}$. The result is divided by two because for each word that is Read operation a Write operation is potentially needed to write back the corrected version.

$$\text{words per scrub cycle} = (t_{RFC} - t_{RCD} - t_{CL} - t_{WR} - t_{RP})/2 \tag{6-1}$$

2. Determine scrub rate meaning the number of scrub cycles per unit time.

$$\text{scrub rate} = \frac{\text{total memory words}}{t_s \cdot \text{words per scrub cycle}} \tag{6-2}$$

3. Next configure the MCU memory controller with a lower $t'_{refresh}$ such that the MCU issues more Refresh operations in the same amount of time.

$$t'_{refresh} = t_{refresh} \cdot \frac{t_{refresh}}{t_{refresh} + \frac{1}{\text{scrub rate}}} \tag{6-3}$$

   Then the scrubber in the FPGA must be configured to perform a scrub cycle instead of a Refresh operation at every $\frac{t_{refresh}}{t'_{refresh}}$ Refresh operation it receives from the MCU.

B. Memory configuration

In this part the configuration of the SDRAM devices is determined. There are three rules that must be followed in order to store the parity bits given a desired available memory capacity M_{cpu}, the maximum number of SDRAM chips D_{max} and the data bus width of d_{sdram}.

1. The first rule is that the complete data bus width in the SDRAM interface between the FPGA and the SDRAM devices d_{sdram} must be wide enough for all the data bits d_{mcu} and all parity bits.

   For example, a design that uses $d_{mcu} = 32$ and TMR for error correction requires $d_{sdram} = 96$. This can be implemented with three SDRAM devices each with a 32-bit data width.

2. A second rule that must hold is that the structure of the SDRAM devices must be the same for all meaning the number of columns, rows and banks must be equal. This combined with the previous rule automatically ensures that there is enough memory for the data bits (usable by the processor) and the parity bits: $M_{total} = M_{cpu} + M_{parity}$.

3. The third rule applies for designs in which a separate encoder is used for each byte in d_{mcu} it is important that the parity bits from each encoder are stored in a part of the memory with a separate mask signal.

   For example, a design for $d_{mcu} = 32$ with four $k = 8$ encoders producing $n - k = 4$ parity bits each requires $d_{sdram} = 48$ which can be implemented using one 32-bit and one 16-bit SDRAM device because this is combined 48-bits. However, a 16-bit SDRAM devices can only mask two bytes meaning the two of the four sets of parity bits must share a mask signal. This introduces a RMW problem and therefore a design that is not byte addressable.

C. FPGA memory interface design

The FPGA memory interface design deals with the issue of the MCU that cannot be configured to accept a higher read latency (CL_{mcu}). The design choice depends on the number of synchronous elements in the encoder and decoder selected in part A. The outcome of the interface design determines the reduction in the memory throughput of the MCU as in all cases a reduction in the clock speed of clk_{mcu} is required.

A major aspect to consider is the use of Flip-Flops in the FPGA IO banks. Although, such IO Flip-Flops have fast clock-to-output times and low delays from to and from the physical pad, no combinational can be inserted after a IO output Flip-Flop or before an IO input Flip-Flop.

When the number of synchronous elements such as Flip-Flops and synchronous BRAM has been determined one of the three designs in the discussed in Section 5.2 can be selected. Below three arguments are given for selecting one of the three memory design interface designs:

- **Combinational**

  A combinational design can only by used if the implementation of the encoder and decoder is also completely combinational, i.e. using a matrix multiplication with XOR-gates. For larger values of n the error pattern lookup table needs to be implemented with BRAM. If the selected FPGA only supports synchronous reads from BRAM a sequential design is required.

- **Sequential dual frequency**

  A dual frequency approach is most suitable for a design that doesn't require many Flip-Flops to be inserted. A frequency ratio $f_{sdram}/f_{mcu} = 2$ with $CL_{mcu} = CL_{sdram} = 3$ gives a Δlatency $= 3$. To be able to add more than 3 Flip-Flops the use of a triple frequency design is recommended to prevent further reduction of the ratio f_{mcu}/f_{sdram}.

  A sequential design is unavoidable when using a BCH decoder or using synchronous SRAM for the error pattern lookup table.

- **Sequential triple frequency**

  As already mentioned a third frequency is recommended if more than 3 Flip-Flops are required to implement the encoder and decoder. There is a practical limit in the ratio of the clock signals clk_{fpga}/clk_{sdram} as Flip-Flops in FPGA fabric generally have a maximum frequency of around 400 MHz. Using such high frequencies also comes at the expense of increased power consumption.

This part of the methodology is what determines the loss of performance of the MCU memory controller. Decreasing the clk_{mcu} frequency by a factor of two for a sequential dual frequency design means the potential throughput of the MCU memory controller is halved. Slower memory means the CPU has to wait longer for memory accesses slowing down the software.

D. FPGA device selection

The next step addresses the selection of an FPGA. Below are the most important aspects that need to be considered:

1. **FPGA resources**.

   The FPGA must of course have enough resources in terms of logic elements (LE) and memory to be able to implement the circuitry of the encoder and decoder and the logic of the scrubber. To estimate the required number of LEs the equations in Section 4.1 can be used. Note that these equations assume 2-input gates while LEs implement gates using LUTs often with 4 (4LUT) or 6 (6LUT) inputs depending on the FPGA. This means a n-wide XOR-tree can be implemented more efficient.

   There are multiple possible implementations of the error pattern lookup table:

   - Distributed BRAM which uses the LE LUTs to implement a memory. This is useful implementation for smaller error patterns.
   - When larger error patterns need to be stored a more efficient approach would be to use the BRAM present in the FPGA. When selecting the FPGA a criteria to check is the total capacity of the BRAM and whether it supports asynchronous read operations. If it only supports synchronous read operations a Flip-Flop must be sacrificed.

2. **IO requirements**.

   The FPGA must have enough high-speed IO that supports the same IO standard as the MCU and the SDRAM. For testing purposes its good to include a separate Chip Select signal for each SDRAM device. This way each SDRAM chip can be accessed independently.

   Additionally, check the distance in the fabric floorplan of the FPGA between IOs that are (in)directly connected. Take this into account when assigning pins. Figure 6-3 shows an example of a good and bad IO assignment.

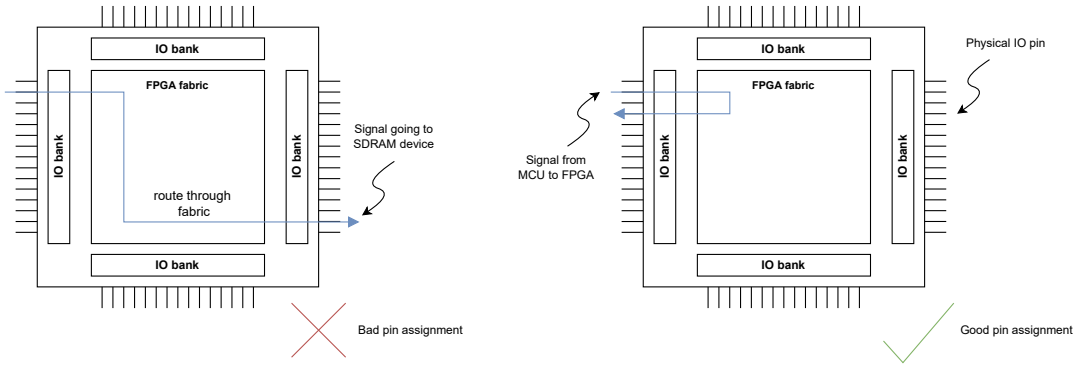


Figure 6-3: Illustration of an FPGA with a bad IO pin assignment (left) and a good IO pin assignment (right). The left is not optimal because the input and output IO are distant from each other requiring a long routing through the fabric. A better approach is to assign the pins such that the input and output are physically close.

3. **Clock distribution**. An important part of the design is the distribution of the clock signals. The design cannot use two clock sources because this causes meta-stability issues which cannot be solved with a re-synchronization circuit because of added latency.

   The design needs to be in sync with the clk_{mcu} outputted by the memory controller of the MCU. When multiple clocks are used it is not advised to generate these clocks in the MCU. Since these clocks might have a phase shift compared to clk_{mcu} which can result in timing issues.

   A good approach is to generated new clocks from the clk_{mcu} clock in the FPGA using a PLL. The FPGA tools of your vendor should know all the delays of the PLL such these can be considered in the timing analysis. Check if the FPGA has a special IO that directly feeds into PLL or other clock manager circuitry.

E. Critical path timing analysis

Before the design is implemented a critical path timing analysis must be performed to determine whether all signals in the design can meet the timing requirements (setup and hold time). Depending on the design of the FPGA memory interface there are four critical paths that must be analyzed:

- Command, address and write data captured by the FPGA (sequential design only).
- Command, address and write data captured by the SDRAM.
- Read data captured by the FPGA (sequential design only).
- Read data captured by the MCU.

For some of these critical paths an estimation of the delay inside the FPGA fabric is needed. For the encoder and decoder circuitry the expression in Section 4.1 can be used but these don't include the routing delay which can be very significant.

A rough estimation can be made by creating a project in the tools of your FPGA vendor with a single input and output connected to each other. Then, assign IOs to the input and output based on the worst case IO pin placement defined in the FPGA selection step. Running a Static Timing Analysis on this path will give a routing delay that can be expected in the final design.

6.4 Implementation

After all design options have been chosen the final step is the implementation of the design. This steps consists of drawing up the schematics of the PCB, designing the layout of the PCB and coding the FPGA logic in a Hardware Description Language (HDL) such as VHDL or Verilog. The engineering practices for these tasks are beyond the scope of this work.

An important step in the implementation is to test if the SDRAM interface is working and the MCU is able to access the complete memory range of M_{cpu}. A recommended approach

for such a test is by implementing an idle FPGA design in which no encoder, decoders or scrubber is present. This is helpful when troubleshooting issues on the PCB. Using an idle FPGA design allows for troubleshooting the PCB without having to worry about any bugs in the EDAC solution.

Described in this step is the control logic that needs to be implemented in addition to the EDAC solution. Some extra features that are useful for testing purposes are also discussed at the end.

6.4.1 Control logic

Until now only the implementation of the logic of the encoder and decoder has been discussed. However, for the FPGA to bridge the SDRAM interface from the MCU to the SDRAM correctly some additional control logic is required. Below is a summary of the different types of control logic that needs to be included.

Bi-directional data bus

The SDRAM interface connecting the FPGA to the MCU and the SDRAM have a bi-directional data bus, referred to as the DQ signals. However, internally FPGAs don't support bi-directional signals. To read and drive the bi-directional DQ signals, the FPGA IO-banks has tri-state buffers that can be used to control the direction of the IO pin.

The state of the tri-state buffer is controlled based on a Read or Write operation performed by the MCU. When the MCU reads from memory the FPGA must enable its tri-state buffers connected to the MCU. This allows the FPGA to drive the DQ signals with the data the FPGA received from the memory. The tri-state buffers must be disabled for all other operations. For tri-state buffers connecting the FPGA to the SDRAM the opposite is true, the tri-state buffer must be enabled for Write operations and disabled for Read operations.

Command control

During normal operation the MCU sends SDRAM commands to the SDRAM using the WE, RAS, CAS and CS signals to control the memory. Some additional logic must be inserted in the paths of those signals to control the commands that are actually being send to the SDRAM devices when using a sequential design with multiple frequencies.

In a sequential design the MCU sends commands every rising edge of clk_{mcu} but the SDRAM is clocked at a faster clk_{sdram}. The control logic is responsible for adding Nop operations additional rising edges the clk_{sdram} has. Another task of the control logic is to allow the scrubber to send commands to the SDRAM.

Scrubber control

The implementation of the scrubber also requires control logic to perform the scrub cycle. Additionally, the control logic needs to keep track of the address(es) that needs to be scrubbed in the current and next scrub cycle. The implementation of the control logic depends on the number of memory words that are scrubbed per scrub cyce and the timing parameters listed in Eq 6-1.

6.4.2 Extra features

Besides the implementation of the logic of the encoder/decoder and control logic two extra features can be added in the FPGA that are not required to have a transparent EDAC FPGA design for external memory but are useful for testing and verification purposes: error injection and error logging.

Error injection

Error injection refers to the method of purposely changing a memory word such that the code word stored in the memory becomes invalid. Depending on how much the code word has been changed the error pattern is either correctable or uncorrectable. There are two ways an error can be injected in the SDRAM.

- **Disabling/masking the parity bit memory**.

  This method requires a special mode in which the parity bits are not updated in the SDRAM devices and the decoder is disabled. When the parity bits can no longer be changed the MCU can rewrite the message part of each code word to introduce an error pattern.

  The advantage of this approach is that error patterns can be read and written quickly through the SDRAM interface by the processor. However this, requires some more complex control logic disable the parity SDRAM or to mask parts of a Write operation containing parity bits.

- **Post-encoding injection**.

  An alternative is to implement some logic that applies an error pattern after the encoder. This simplifies the error injection but requires an external interface to configure the error pattern in the FPGA.

For both methods it is convenient to be able to bypass the decoder. This way the injected error is not corrected when a Read operation is performed allowing the MCU to verify the error was successfully injected.

Error logging

Error logging is a feature that allows for collecting statistics on the errors observed by the EDAC solution. Whenever the decoder detects an error during a Read operation of the MCU or a scrub cycle the error can be logged in a memory.

The error logging can be implemented with multiple levels of complexity. A basic implementation simply counts the number detected errors. More complicated error logging can include saving the addresses of the error, saving the error pattern and tagging each logged error with a timestamp.

Accessing the error logger within the FPGA by the MCU is recommended to be done with a different interface than the SDRAM interface that is already connected to the FPGA, like I2C or SPI. The rate at which errors occur should be low enough to be able to use an interface with a lower throughput compared to an SDRAM interface. Also, using a separate interface allows the logger to operate independent on the SDRAM interface going through the FPGA.

6.5 Conclusion

In this chapter a methodology was defined towards FPGA based error correction for the external memory of a MCU. The methodology describes different steps in such a way that it guides the user towards an optimal design in which the reliability is increased to the desired level while minimizing trade-offs.

To achieve this the methodology includes an analysis step in which the required reliability is determined and the error correcting capabilities that provide this level of reliability. Then together with design parameters of the MCU and memory a design space is explored. After a design is chosen and deemed implementable it can be realized. For this the methodology describes the control logic that needs to be implemented and some additional features for testing purposes.

Chapter 7

Hardware demonstrator

In this chapter a hardware demonstrator of the methodology is discussed. In section 7.1 the original PCB on which the hardware demonstrator is based is introduced. Next, Section 7.2 describes how the methodology was applied to come up with the design of the hardware demonstrator. Section 7.3 presents benchmark results showing how the performance of the MCU is affected by the addition of the FPGA and EDAC solution.

7.1 iOBC

ISISPACE sells and uses their ISISPACE On-Board Computer (iOBC) as a platform to run CDHS software. This board, shown in Figure 7-1 consists of a MCU with an ARM9 processor capable of running at 400 MHz and a single SDRAM device of 64 MB (512 Mb). The size of the PCB is approximately 10×10 cm.

The prototype develop as hardware demonstrator for this project is based on this board with the goal to improve its reliability. The next section describes how the methodology defined in the previous section is applied to this existing board to achieve the reliability desired by ISISPACE.

7.2 Applying the methodology

7.2.1 Reliability requirements

The current iOBC has no EDAC solution for its SDRAM memory. This means it can theoretically have up to 10 upsets per day with its single SDRAM device as was concluded in Section 3.2.1.

ISISPACE doesn't have knowledge of the internal layout of the SDRAM devices they use. Therefore, they are forced to assume that $L > 1$ meaning a single event can cause more errors

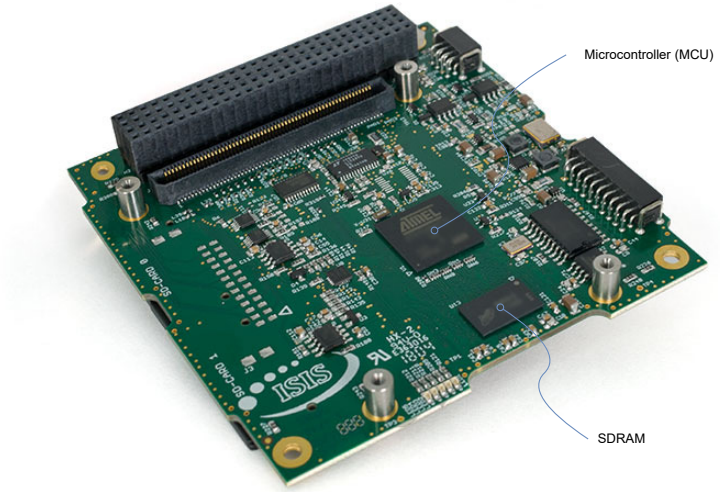


Figure 7-1: Photo of the iOBC developed by ISISPACE.

in a memory word than the error correction capability t. In this case, the higher the error correcting capability the better.

It was decided to implement the following EDAC designs in the hardware demonstrator prototype:

- $t = 1$ + scrubber
- $t = 2$ + scrubber
- TMR + scrubber

7.2.2 Electronics platform

The second part of the first step of the methodology is to analyse the electronics platform. The memory requirements were imposed by ISISPACE and are shown in Table 7-1 below. They required the same amount of usable memory for their CPU. Only one additional SDRAM device is allowed to be placed on the board because of limited board space.

Table 7-1: Memory requirements of the prototype

Parameter	Value
M_{cpu}	64 MB (512 Mb)
D_{max}	2

Table 7-2 shows the MCU and memory controller specifications. Parameters d_{mcu}, t_{RFC} and $t_{refresh}$ are configured in the memory controller of the MCU. The clock clk_{mcu} is generated by a clock generator circuit in the MCU. The cache policy is part of the ARM9 CPU core.

Table 7-2: Memory requirements of the prototype

Parameter	Value
d_{mcu}	$\{16, 32\}$
t_{RFC}	The memory controller can configure t_{RFC} to $\{1, ..., 15\}$ in clock cyles of clk_{mcu}
$t_{refresh}$	The refresh interval is configured using a counter incremented every clock cycle clk_{mcu}. A Refresh operation performed with intervals of: $\frac{\text{Refresh counter}}{f_{mcu}}$
clk_{mcu}	$\{66$ MHz$, 133$ MHz$\}$ The clock generator circuit is actually more flexible and allows other frequencies of clk_{mcu} but this requires lowering the CPU clock frequency which was undesirable.
Cache policy	No support for Write-Allocation for Write misses.

7.2.3 Design

Given the reliability parameters defined in Section 7.2.1 and the parameters of the MCU defined in Table 7-1 and 7-2 the design can be constructed following the parts described in the second step of the methodology.

The critical path timing analysis part of the methodology is not discussed in this section. Instead, on multiple occasions the parts discussed in this section will mention several iterations were needed to reach the final design. In these cases a need for an iteration was based on the result of the critical path timing analysis.

EDAC configuration

As mentioned in the Section 7.2.1 three different designs are to be implemented for the hardware demonstrator: $t = 1$, $t = 2$ and a TMR design. Because the cache of the MCU of the iOBC does not support Write Allocation a separate encoder and decoder is needed for each byte in d_{mcu}.

Using Table A-1 and A-2 the number of parity bits for a $k = 8$ encoder are 4 and 10 for a $t = 1$ and $t = 2$ respectively. With only 12 error patterns for the $t = 1$ design and 153 for the $t = 2$ according to Table A-1 and A-2 a matrix multiplication with lookup table as decoder can be used. For the TMR design the number of parity bits is $2 \cdot d_{mcu}$ as was defined in Section 6.3.

The scrubber of the prototype was not designed with a specific scrub interval t_s but rather to demonstrate its feasibility. The first step is to use Equation 6-1 to calculate the maximum number of words that can be scrubbed in each scrub cycle. However because the ratio of f_{mcu} and f_{sdram} is not known no specific value can be determined yet.

$$\begin{aligned} \text{words per scrub cycle} &= (t_{RFC} - 3 - 3 - 1 - 3)/2 \\ &= (t_{RFC} - 10)/2 \end{aligned} \tag{7-1}$$

For simplicity the scrubber will be designed with $t'_{refresh} = t_{refresh}/2$ meaning every second Refresh operation is ignored by the FPGA and an scrub cycle is performed instead. The original refresh interval is $t_{refresh} = 7.8\ \mu s$. The SDRAM used has 4 banks and 8,192 rows each with 512 columns giving a total of 16,777,216 memory words. This gives the scrub interval in seconds:

$$\begin{aligned} t_s &= \frac{\text{total memory words}}{\text{scrub rate} \cdot \text{words per scrub cycle}} \\ &= \frac{130}{\text{words per scrub cycle}} \end{aligned} \tag{7-2}$$

Memory configuration

The design of the prototype is constraint by the number of SDRAM devices it can use, only two. The highest width SDRAM devices that can be bought are 32-bit wide. This means using two 32-bit SDRAM devices the SDRAM interface can be 64-bit wide, $d_{sdram} = 64$.

With $d_{sdram} = 64$ and the required number of parity bits for each EDAC design it is not possible to use $d_{mcu} = 32$ for the $t = 2$ and the TMR design because those requires more than the available d_{sdram}. Therefore, these designs must use a lower data width of the MCU: $d_{mcu} = 16$. An overview of memory widths d_{mcu} and the number of parity bits for the three EDAC designs is shown in Table 7-3.

Table 7-3: Overview of the memory interface widths d_{mcu} and how many bits are used of d_{sdram} for each EDAC design.

EDAC design	d_{mcu}	parity bits	d_{sdram} used
t = 1	32	16	48
t = 2	16	20	36
TMR	16	32	48

FPGA selection

For the prototype the Microsemi SmartFusion2 M2S025 FPGA was chosen. The primary reason for choice is that the SmartFusion2 family is already used within ISISPACE allowing existing designs to be used as a reference. The FPGA also has enough logic elements and IOs.

The IO pin assignment of the FPGA saw a couple of iterations as a result of the critical path timing analysis that is described in the last part of the design space exploration in the methodology. The analysis showed that the initial IO pin assignment was not optimal and cause large routing delays inside the FPGA fabric. In the final design the IOs of signal groups like the commands, address and data signals were group together to minimize the routing path.

Figure 7-2 shows the clock distribution of the design. The clock clk_{mcu} coming from the MCU is used as a reference clock by the PLL in the FPGA to generate three clock signals. The 66 MHz signal is used to synchronize with clk_{mcu} clock. A 45° phase shifted clock is used to

clock the SDRAM devices. The phase shift is needed to compensate for the routing and IO delay of the FPGA.

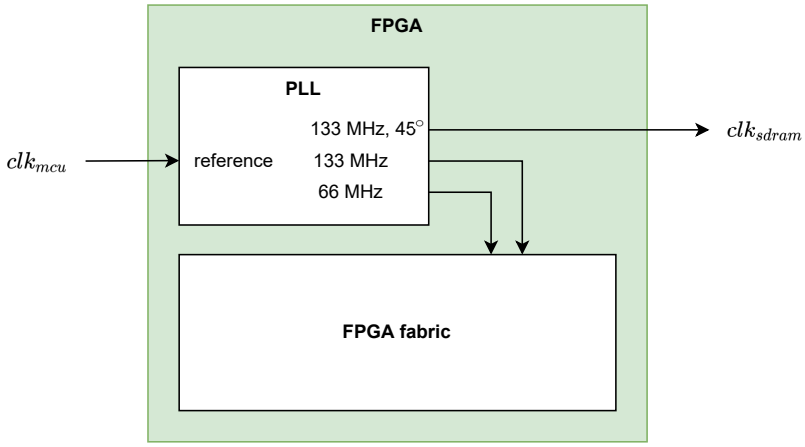


Figure 7-2: Clock distribution overview of the prototype

FPGA memory interface design

For the memory interface design a sequential design with two frequencies was chosen with $f_{mcu} = 66$ MHz and $f_{sdram} = f_{fpga} = 133$ MHz. Using Equation 5-1 this gives Δlatency= 3 meaning a total of three Flip-Flops can be inserted in the signal paths of the SDRAM interfaces. There are two reasons why a sequential design with this configuration was chosen:

- The datasheet of the MCU provided limit information about the timing characteristics of the SDRAM memory controller, especially the maximum clock-to-output delay $t_{co_{max}}$ of the output Flip-Flops. This forced to assume a worst case $t_{co_{max}} < 6$ ns given the memory controller's maximum clock period is 7.5 ns ($f_{mcu} = 133$ MHz) and a setup time of the SDRAM of 1.5 ns.

  To add any logic and allow for IO and routing delay the frequency of clk_{mcu} needs to be lowered. According to Table 7-2 to only available option is to set $clk_{mcu} = 66$ MHz. With a clock period of 15 ns and $t_{co_{max}} = 6$ the remaining 9 ns was deemed plenty to for any control logic or encoding logic and FPGA delays.

- Using a ratio of $f_{sdram} = 2 \cdot f_{mcu}$ allows more words to be scrubbed per scrub cycle. Allowing the memory to be scrubbed in a shorter time period.

One Flip-Flop is inserted in the command, address, DQM and write data path. The remaining two Flip-Flops are inserted in the read data path. This configuration maximizes the decoding time which is needed because decoding is more time consuming than encoding.

For all Flip-Flops in the design the Flip-Flops in the IO banks of the FPGA are used. Static Timing Analysis in the Microsemi FPGA tools showed these significantly reduce the $t_{co_{max}}$ of

the FPGA output Flip-Flops. Without it, the setup and hold times of SDRAM devices could not be met.

An overview of the sequential design with the inserted Flip-Flops is shown in Figure 7-3. Shown in the graph is the single Flip-Flop in the command, address and DQM path. The bi-directional DQ signals are separated in a write path with the encoder followed by a Flip-Flop and a read path with two Flip-Flops and the decoder in between.

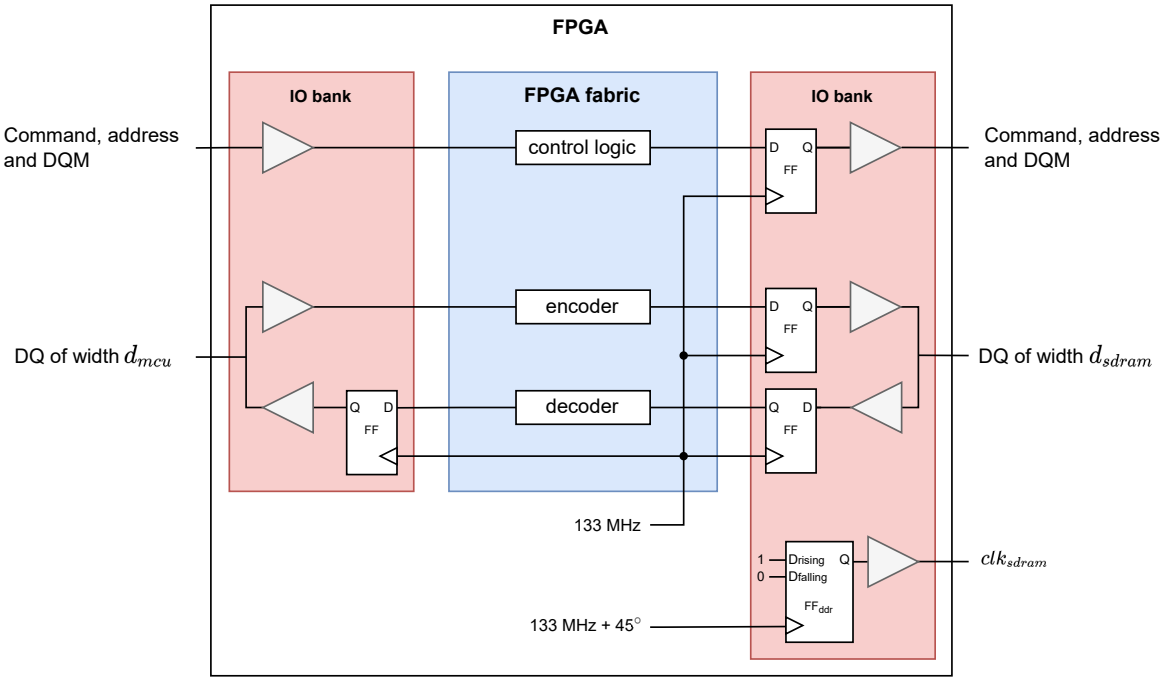


Figure 7-3: Overview of the Flip-Flops inserted in the signals paths in the FPGA. All Flip-Flops are located in the IO banks to improve clock-to-output timing.

Now the ratio between $f_{mcu} = f_{sdram}$ is known the number of words per scrub cycle can be determined and the scrub interval. With $f_{sdram} = 2 \cdot f_{mcu}$ the t_{RFC} configured in the MCU memory controller can be multiplied by two. The final deign of the scrubber was chosen with four words per scrub cycle resulting in an approximate scrub interval $t_s = 33$ seconds using Equation 7-1 and 7-2. Meaning the whole memory can be scrubbed in 33 seconds.

7.2.4 Implementation

With the final design result of the previous section a prototype PCB was developed, manufactured and tested. The design of the original iOBC was modified to add the FPGA and extra SDRAM device. An overview of the connections between the MCU, FPGA and SDRAM is shown in Figure 7-4.

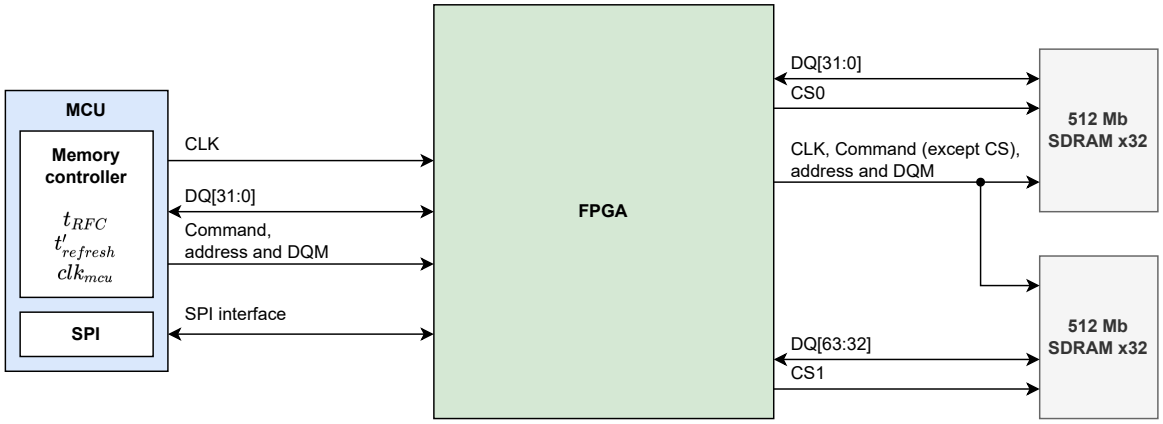


Figure 7-4: Overview of the SDRAM interfaces connecting the MCU, FPGA and SDRAM (2x) in the prototype.

Between the MCU and FPGA there is a standard SDRAM interface with a data width of 32-bit. The memory controller of the MCU sources the reference clock clk_{mcu} and the important parameters t_{RFC} and $t'_{refresh}$ are configured. An SPI interface is added between the MCU and FPGA to configure certain parts of the FPGA design which will be discussed later in this section.

On right side the SDRAM interface between the FPGA and the SDRAM devices is shown. The SDRAMs share the same clock (clk_{sdram}, address and DQM signals but each SDRAM has a separate Chip Select (CS0 and CS1) signal. This is use to inject errors which will be discussed later. Each SDRAM has its own 32-bit data bus connected to the FPGA.

Signal integrity issues

Initial tests of the prototype were performed with an Idle FPGA design in which the design shown Figure 7-3 is implemented but with the encoder and decoder removed. Memory test software was developed to test if the complete memory can be accessed. The software used different access patterns while writing and reading from memory to verify different columns, rows and banks could be accessed. The results of the memory tests showed an issue in which the data written would not always match the data that was read.

Troubleshooting the problem concluded that the issue was related to the timing of the SDRAM signals. This prompted an investigation on the signal quality of the SDRAM interfaces on the PCB. Figure 7-6 shows a capture of the clk_{mcu} signal. The signal is supposed to be a 66 MHz square wave but the actual signal suffers from a lot of over and undershoot.

A temporary fix was implemented by enabling a 50 Ω parallel termination impedance in the FPGA IO. While this improved the quality enough to cause the memory instability to

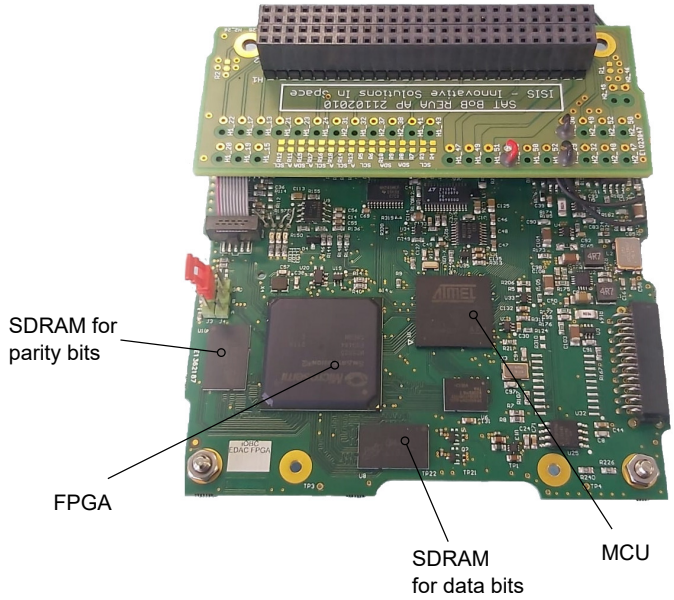


Figure 7-5: Photo of the prototype PCB

go away the overshoot and undershoot still remained visible. Proper electrical termination of the SDRAM interface signals is needed in a future revision such that the signal quality improves to an acceptable level.

Unfortunately, because troubleshooting the issue of the unstable memory took so long only the $t = 1$ EDAC FPGA design was completed and tested on the prototype. Most of the VHDL code of the $t = 2$ and TMR design was finished but due to the lack of time no bitstream was created and tested.

Error injection

To test whether the implemented $t = 1$ EDAC solution works and that it is actually correcting errors an error injection feature was added to the FPGA design. To inject errors three different modes were added to the FPGA. These modes determine how the data flows between the MCU and the SDRAM when the MCU performs a Read or Write operation. The active mode can be changed through the SPI interface.

Table 7-4: FPGA modes used for error injection.

FPGA mode	Data written to SDRAM	Data read by MCU
Default	Encoder output	Decoder output
Data pass-through	MCU write data (SDRAM0 only)	SDRAM0 read data
Parity pass-through	MCU write data (SDRAM1 only)	SDRAM1 read data

To inject errors the FPGA is first put in Default mode. Writing to an address in the SDRAM causes the complete code word to be written. Then by switching to any of the pass-through modes the data can be overwritten without the encoder/decoder intervening to apply an error

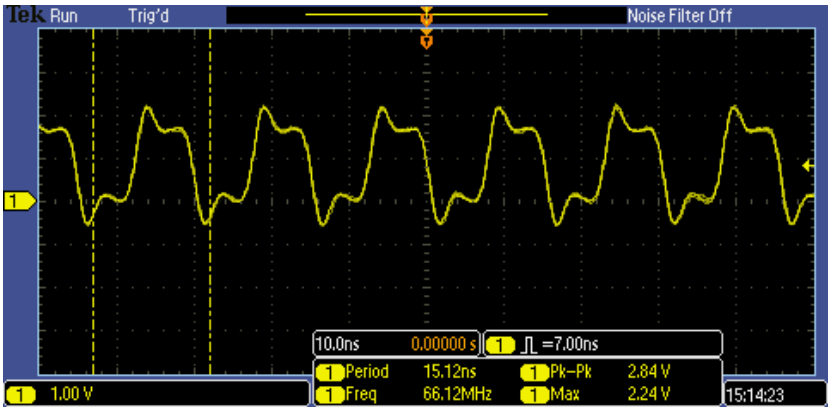


Figure 7-6: Signal capture of the clk_{mcu} going from the MCU to the FPGA. The signal shows significant overshoot and undershoot.

pattern by changing one or more bits. To test if the decoder corrects the error the FPGA is set to the default mode again. If the error pattern contains t or less errors the correct data should be read.

Using these modes the error correction and the scrubber were verified. Software tests were written and these concluded that error patterns within the the capabilities of the EDAC solution were corrected and that the scrubber removed the errors from memory over time.

7.3 Benchmarks

With the prototype working it is possible to run benchmarking software to evaluate the loss in performance compared to the original design of the iOBC because of the lowered clock speed of MCU memory controller clk_{mcu}. The prototype runs the memory at 66 MHz while the original iOBC uses a 133 MHz clock.

The following benchmark suites were used: Embench [41] and CoreMark-PRO [42]. Both suites contain multiple programs each running a specific type of algorithms like image compression, linear algebra and encryption. From the CoreMark-PRO only a subset of all its programs were executed.

The two benchmark suites were executed on an original iOBC and the new prototype running the $t = 1$ EDAC design in the FPGA. Each benchmark was executed with and without the scrubber enabled to see the effect the MCU memory controller performing more Refresh operations and with O0 and O1 compiler optimization. For all programs executed from the benchmark suites the execution time was recorded. The processor of the MCU had its instruction cache enabled and data cache disabled.

The results are presented in Figures 7-7 to 7-10. These graphs show the speedup of the execution times of the new prototype design versus the original iOBC. Both benchmarks are very similar as both show the new prototype running with the lower 66 MHz memory clock is slower. Almost programs run approximately twice as slow with the exception of a few. The linear_alg_mid_100x100_mid, cubic and nbody programs are very arithmetic intense which

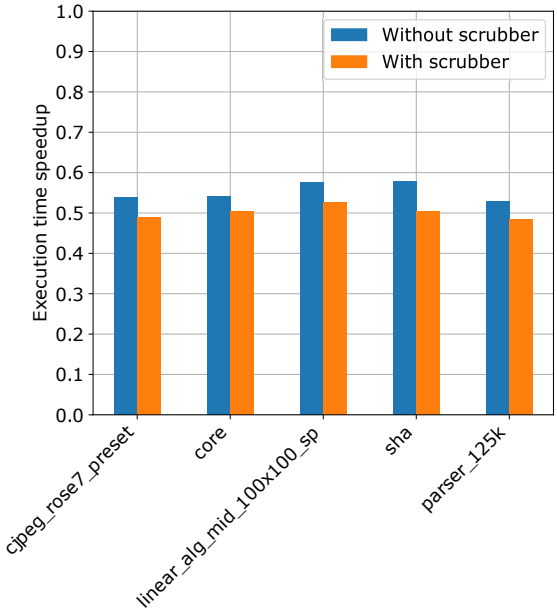


Figure 7-7: CoreMark-PRO benchmarks compiled with -O0

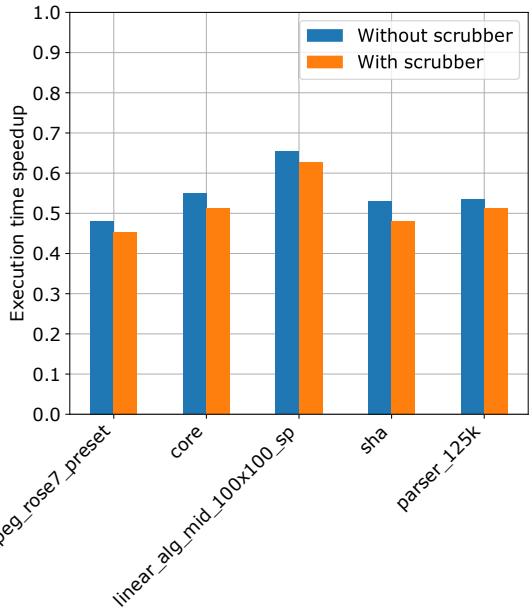


Figure 7-8: CoreMark-PRO benchmarks compiled with -O1

could explain why their suffer less from the reduced memory clock frequency compared to the other programs.

The extra Refresh operations introduced by the enabling of the scrubber show an additional increase of execution time for all benchmarks. All Embench programs run an additional 11% slower on average and the CoreMark-PRO programs 5% and 9% on average for O0 and O1 respectively. It must be noted that this design uses an extreme scrubber implementation that scrubs the entire memory every 33 seconds.

7.4 Conclusion

This chapter discussed the implementation of a hardware demonstrator that was realized following the proposed design methodology. The demonstrator is a prototype based on the original iOBC designed by ISISAPCE which has no EDAC solution in its memory controller. To improve the reliability of the iOBC an external FPGA based EDAC solution has been implemented using the methodology defined in the previous chapter.

The prototype consists of the original MCU, an FPGA and two SDRAM devices. For the EDAC three designs were chosen, all with a scrubber: $t = 1$, $t = 2$ and TMR. Because of issues with the prototype only the $t = 1$ was tested. Memory tests showed the memory is accessible by the MCU and by using error injection the EDAC solution was verified. Errors are corrected when the MCU reads from memory and when the scrubber reads an corrupted address.

Benchmark showed that the lowered clock frequency of the memory controller from 133 MHz to 66 MHz halved the execution time of the benchmark software. The scrubber decreased the

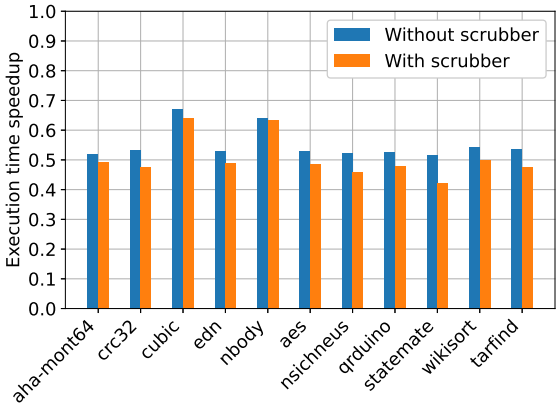


Figure 7-9: Embench compiled with -O0

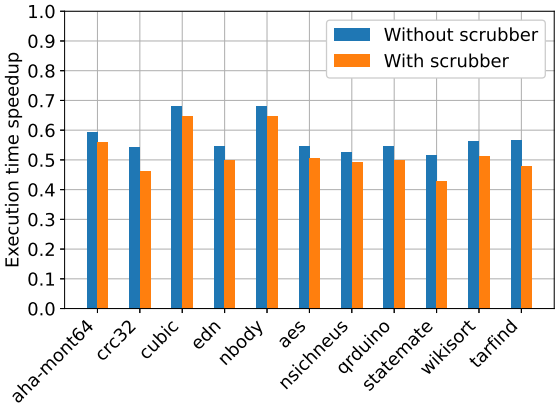


Figure 7-10: Embench compiled with -O1

performance by an additional 5-11% but the prototype uses an extremely fast scrubber that scrubs the entire memory in 33 seconds.

Chapter 8

Conclusions

In this chapter we provide the answers to the research questions defined in the introduction and we describe the main contributions of this project. Lastly, some suggestions for future improvements are made.

8.1 Research questions

This thesis introduced an FPGA-based EDAC solution for the correction of soft errors in the external memory of microcontrollers operating in a space environment. The motivation for this work is that few microcontrollers have an integrated EDAC solution and if they do the implementation can be insufficient. To cope with this problem the following main goal was defined for this thesis:

To develop a proposal for a generic methodology towards transparent FPGA based recovery from soft errors, that can be applied given a specific microcontroller, its off-chip main memory and the parameters of the targeted space mission.

To reach this goal several research questions were defined of which the answers are given below:

- **What type of soft errors can be found in SDRAM in operating low earth orbit and how often do they occur?**

  A literature review showed that soft errors are a common problem for spacecraft including the ones in LEO and that there are multiple types of soft errors than can occur. SEUs can cause one or more errors depending on the energy of a particle and the sensitivity of a DRAM cell to the collection of charge from a particle strike. Soft errors can also occur in the SDRAM control logic causing many errors, these are referred to as SEFIs.

  SDRAM shows a trend in which the sensitivity per cell is dropping but because more cells are put in a device as the technology scales the upset rate per device remains

relatively constant. The upset rate of SDRAM in a LEO environment is difficult and rates derived from observations of actual spacecraft differ a lot. Therefore, only an upper limit estimation of 10 SEU/day/device can be made. This rate is high and shows the need for an EDAC solution to correct the assumed soft errors.

- **Are there affordable error correcting solutions in terms of FPGA implementation to cope with these soft errors?**

  Error correcting is commonly done with ECCs which require encoding to map a message to a code word and decoding to reverse the mapping with the addition of correcting any errors. Several implementations were discussed and the matrix multiplication implemented as an XOR-tree was concluded to be the fastest but requires a lot of memory when the code words get larger. In those cases an algebraic BCH decoder is more efficient. Using a LFSR for cyclic ECCs is not recommended because it typically requires many clock cycles.

  The main challenge in implementing the FPGA based error correction is dealing with the delays caused by routing the SDRAM interface through an FPGA and the logic to encode and decode the ECC code words. The memory controllers of MCUs are designed to work with standard SDRAM that only allows an amount of latency for Read operations. However, with additional delays introduced by the FPGA the read latency needs to be increased which cannot be done in these standard memory controllers. This issue is only solved by lowering the clock frequency of the memory controller. The disadvantage of this is a reduced throughput of the memory.

  A solution to transparently scrub the memory has also been proposed. The solution uses an increased the refresh rate of the SDRAM to force the MCU from not accessing the memory. This allows the FPGA to freely read and write the memory to correct any errors.

  In conclusion there are affordable error correcting solutions for FPGA based error correcting but they do come with trade-offs in terms of performance of the MCU memory controller in order to be implemented.

- **Can a methodology be derived to implement the solutions of the previous question transparently to any microcontroller?**

  A methodology was defined in this work based on the collected knowledge about the type of soft errors in SDRAM, error correction method and interface an FPGA between a MCU and SDRAM. The methodology is defined such that it guides towards an optimal EDAC FPGA design that provides the required reliability while minimizing the trade-offs. To achieve this, three steps have been defined. An analysis step in which the reliability requirements and MCU and memory parameters are collected. Following is a design space exploration to determine the EDAC, FPGA and memory design based on the parameters collected in the previous step. The third step is the implementation of the design.

8.2 Main contributions

The main contributions of this thesis are as follows:

- The proposed methodology is the main contribution of this thesis. The methodology defines the steps that must be followed to design a transparent FPGA-based EDAC solution for the correction of soft errors in the external SDRAM memory of a given specific microcontroller operating in space.
- A hardware demonstrator was made following the methodology to proof the methodology can be applied to a microcontroller such that it provides an external EDAC solution for its memory. The demonstrator was built for the iOBC, an On-Board Computer, developed by ISISPACE. The implemented EDAC solution is capable of correcting a single bit per byte of the 32-bit wide memory interface. Additionally, a scrubber was implemented that scrubs the two 64 MB SDRAM devices in 33 seconds. The performance of the hardware demonstrator is around 50% lower compared to the original iOBC because the memory clock frequency was halved. The addition of the scrubber lowered the performance another 5-11%.

8.3 Future work

- **SEU rate and MBUs**

  Expressing the reliability of a memory system as is done in the work heavily relies on the SEU rate λ and how many soft errors a single event can cause in a memory word. In this work the observations from several spacecraft orbiting in LEO have been used but there are large difference in SEU rates between these spacecrafts which forces the use of an upper limit estimate.

  The number of errors a single event can cause in a memory word is also often unknown because the internal layout of the SDRAM is not known and experimentally acquiring this information is complicated and expensive. However, this information heavily influences how an optimal EDAC solution can be selected.

- **Double Data Rate SDRAM**

  This work focused on Single Data Rate SDRAM which is a relatively old technology. MCUs that are introduced today support newer generations of DRAM like DDR, DDR2 or even DDR3. The working principles of these are similar to SDR SDRAM but because of minor differences in the interfaces the methodology in this work cannot be applied to DDR memory. To make the methodology more future prove the newer DDRx interfaces need to be analyzed in order to define how they can used for an FPGA based external error correction solution for MCUs.

- **Reliability of FPGAs**

  The reliability improved that is claimed in this work because of the error correcting in the FPGA only assumes error occur in the SDRAM. However, it is unlikely FPGAs don't suffer from radiation induced errors as well. A corruption in the logic configuration

of the FPGA fabric could cause the complete memory to become inaccessible for the MCU. The reliability of the FPGA should be considered as well to make a more accurate estimate of the reliability of the complete memory system.

- **BCH codes efficiency**

  To correct errors BCH codes are used in this work. BCH codes allow to construct codes that guarantee to be able to correct t errors which is useful to correct multiple errors in a memory word. However, it is likely that most MCU require to use code words for each byte ($k = 8$). BCH code words for message of $k = 8$ become less efficient when t becomes larger because they are derived from shortened codes.

- **MCU + FPGA System On Chip**

  The motivation for this work is lack of flexibility or lack of EDA features that MCU with an integrated EDAC solution provide and the aversion of engineers to change MCUs. While the proposed solution solves this issue it costs a lot in terms of performance. Caused by lack configurability of the MCU's read latency. A different approach would be to use a System on Chip (SoC) with an MCU and FPGA fabric. This allows a custom memory controller to be implemented in the FPGA capable of handling the increased latency of the decoding. The downside being that it does require a change of MCU platform in case of existing designs.

Appendix A

Error Correcting Codes details

A.1 Encoding and decoding circuits

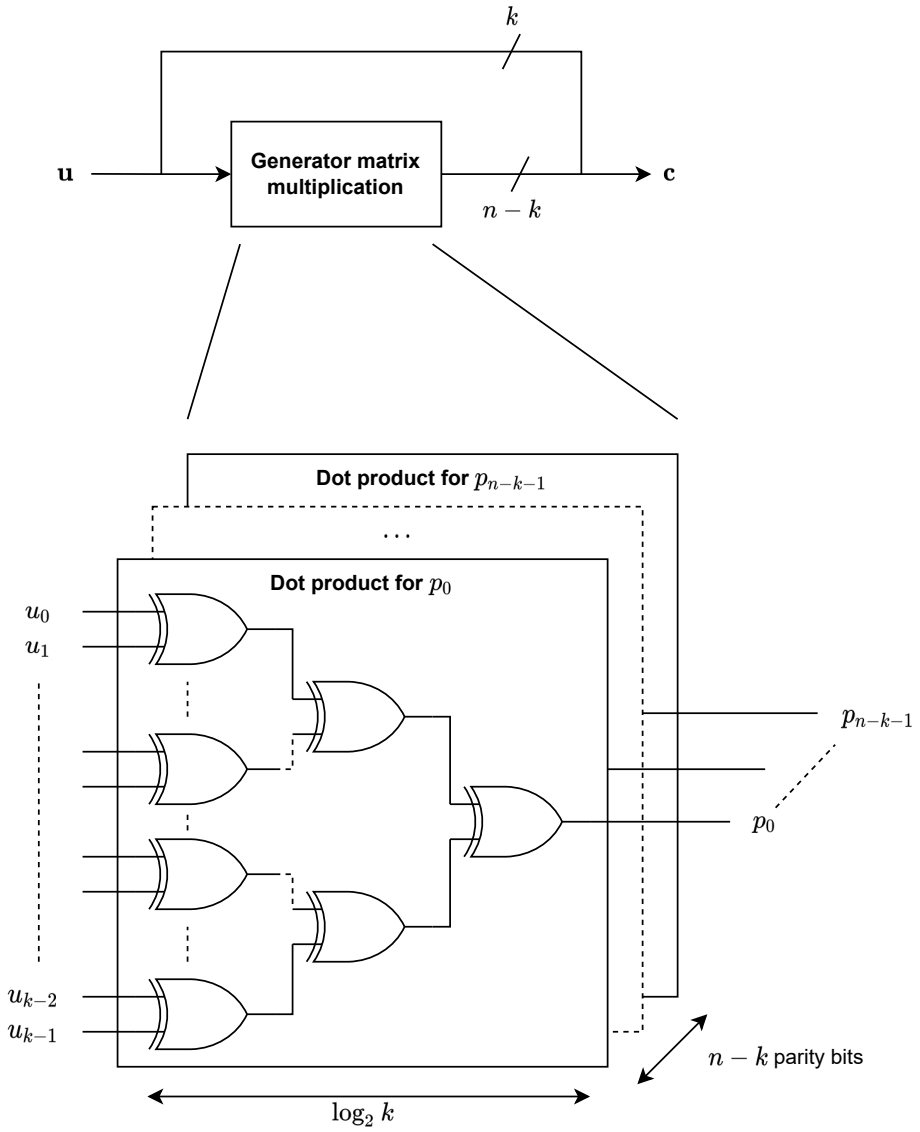


Figure A-1: ECC encoder circuit using generator matrix multiplication. Each parity bit is generated using from a dot product implemented with a XOR-gate tree.

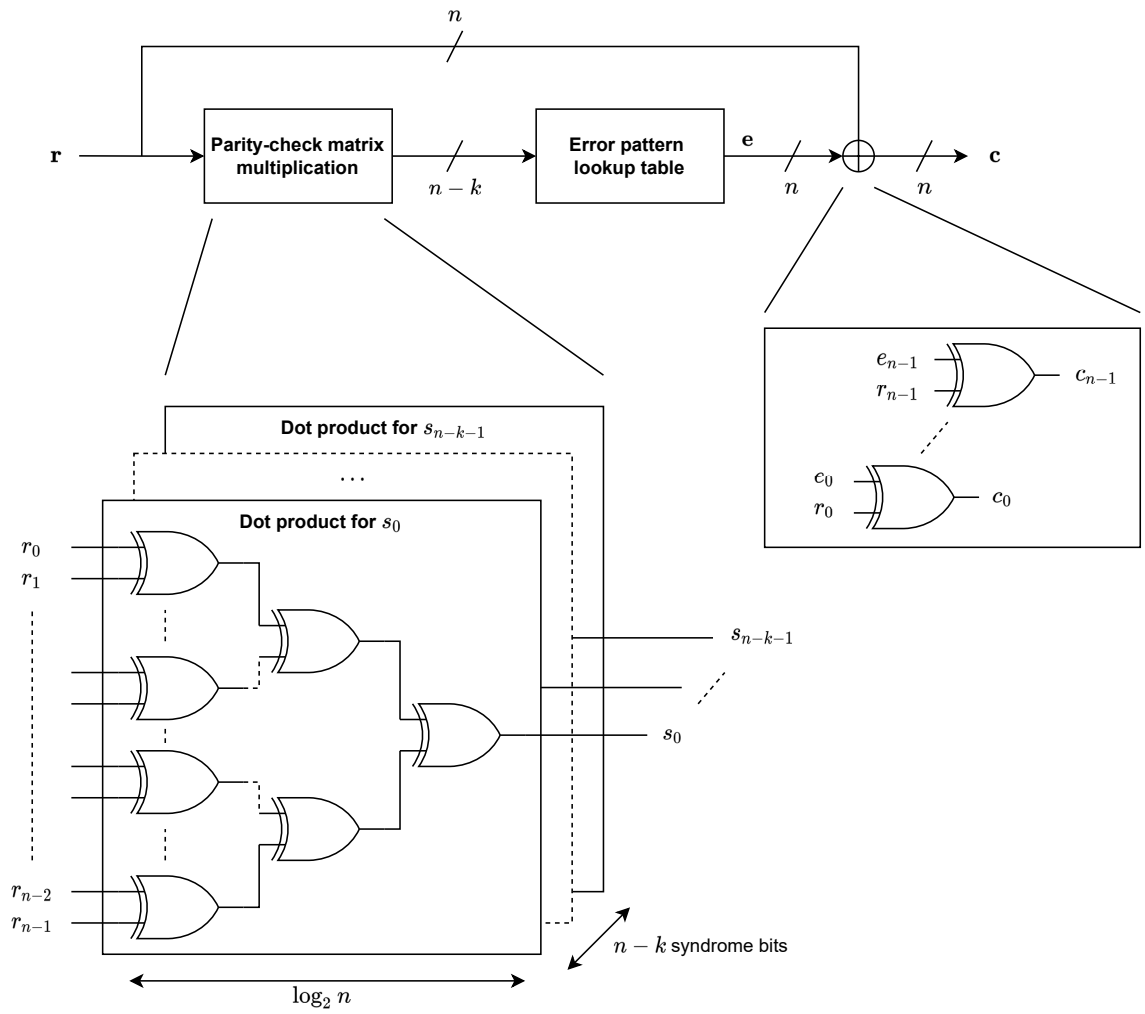


Figure A-2: ECC decoder circuit using parity-check matrix multiplication in the first stage. The error pattern lookup table can be implemented using a memory.

A.2 BCH codes

n	k	R	Error patterns	min. LUT [bits]	max. LUT [bits]
12	8	67%	12	144	192
21	16	76%	21	441	672
38	32	84%	38	1,444	2,432

Table A-1: BCH code size for $t = 1$

k	n	R	Error patterns	min. LUT [bits]	max. LUT [bits]
18	8	44%	153	2,754	18,432
26	16	62%	325	8,450	26,624
44	32	73%	946	41,624	180,224

Table A-2: BCH code size for $t = 2$

k	n	R	Error patterns	min. LUT [bits]	max. LUT [bits]
23	8	35%	1,771	40,733	753,664
31	16	52%	4,495	139,345	1,015,808
50	32	64%	19,600	980,000	13,107,200

Table A-3: BCH code size for $t = 3$

k	n	R	Error patterns	min. LUT [bits]	max. LUT [bits]
28	8	29%	20,475	573,300	29,360,128
40	16	40%	91,390	3,655,600	671,088,640
56	32	57%	367,290	20,568,240	939,524,096

Table A-4: BCH code size for $t = 4$

Bibliography

[1] P. Hughes. The Arm ecosystem ships a record 6.7 billion Arm-based chips in a single quarter. [Online]. Available: https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter

[2] Technology CubeSats. [Online]. Available: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Technology_CubeSats

[3] E. Petersen, *Single Event Effects in Aerospace*. Piscataway, NJ, USA: Wiley-IEEE Press, 2011.

[4] *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*, JEDEC, 2016.

[5] T. Moon, *Error correction coding: mathematical methods and algorithms*. Hoboken, NJ, USA: John Wiley & Sons, 2005.

[6] B. Schroeder, E. Pinheiro, and W. Weber, "DRAM errors in the wild: A large-scale field study," *Commun. ACM*, vol. 54, no. 2, p. 100–107, Feb. 2011.

[7] T. P. Haraszti, *CMOS MEMORY CIRCUITS*. New York: Kluwer Academic Publishers, 2002.

[8] A. K. Sharma, *Semiconductor Memories: Technology, Testing, and Reliability*. Piscataway, NJ, USA: Kluwer Academic Publishers, 1997.

[9] *MT48LC64M4A2 (256 Mb) SDR SDRAM*, Micron, 2015, rev. W.

[10] B. Jacob, S. Ng, and D. Wang, *Memory-Systems: Cache, DRAM, Disk*. Burlington, MA, USA: Morgan Kaufmann Publishers.

[11] *JEDEC Configurations for Solid State Memories*, JEDEC, 1989.

[12] P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583–602, 2003.

[13] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[14] L. W. Massengill, "Cosmic and terrestrial single-event radiation effects in dynamic random access memories," *IEEE Transactions on Nuclear Science*, vol. 43, no. 2, pp. 576–593, 1996.

[15] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, 1979.

[16] T. Toyabe, T. Shinoda, M. Aoki, H. Kawamoto, K. Mitsusada, T. Masuhara, and S. Asai, "A soft error rate model for MOS dynamic RAM's," *IEEE Journal of Solid-State Circuits*, vol. 17, no. 2, pp. 362–367, 1982.

[17] T. V. Rajeevakumar, N. C. C. Lu, W. H. Henkels, Hwang Wei, and R. Franch, "A new failure mode of radiation-induced soft errors in dynamic memories," *IEEE Electron Device Letters*, vol. 9, no. 12, pp. 644–646, 1988.

[18] G. S. Bagatin M., *Ionizing Radiation Effects in Electronics : From Memories to Imagers*. Boca Raton, FL: CRC Press, 2016.

[19] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[20] C. Poivey, J. L. Barth, K. A. LaBel, G. Gee, and H. Safren, "In-flight observations of long-term single-event effect (SEE) performance on orbview-2 solid state recorders (SSR)," in *2003 IEEE Radiation Effects Data Workshop*, 2003, pp. 102–107.

[21] D. Falguere, S. Duzellier, R. Ecoffet, and I. Tsourilo, "EXEQ I-IV: SEE in-flight measurement on the MIR orbital station," in *2000 IEEE Radiation Effects Data Workshop. Workshop Record. Held in conjunction with IEEE Nuclear and Space Radiation Effects Conference (Cat. No.00TH8527)*, 2000, pp. 89–95.

[22] C. Boatella, G. Hubert, R. Ecoffet, and S. Duzellier, "ICARE on-board SAC-C: More than 8 years of SEU and MCU, analysis and prediction," *IEEE Transactions on Nuclear Science*, vol. 57, no. 4, pp. 2000–2009, 2010.

[23] Y. Kimoto, N. Nemoto, H. Matsumoto, K. Ueno, T. Goka, and T. Omodaka, "Space radiation environment and its effects on satellites: analysis of the first data from TEDA on board ADEOS-II," *IEEE Transactions on Nuclear Science*, vol. 52, no. 5, pp. 1574–1578, 2005.

[24] J. J. Schaefer, R. S. Owen, P. M. Rutt, and C. Miller, "Observations from the analysis of on-orbit data from DRAMs used in space systems," in *2009 IEEE Radiation Effects Data Workshop*, 2009, pp. 106–113.

[25] J. D. Black, P. E. Dodd, and K. M. Warren, "Physics of multiple-node charge collection and impacts on single-event characterization and soft error rate prediction," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1836–1851, 2013.

[26] A. Bougerol, F. Miller, and N. Buard, "SDRAM architecture & single event effects revealed with laser," in *2008 14th IEEE International On-Line Testing Symposium*, 2008, pp. 283–288.

[27] P. Kohler, V. Pouget, F. Wrobel, F. Saigné, P. X. Wang, and M.-C. Vassal, "Analysis of single-event effects in DDR3 and DDR3L SDRAMs using laser testing and monte-carlo simulations," *IEEE Transactions on Nuclear Science*, vol. 65, no. 1, pp. 262–268, 2018.

[28] S. Buchner, A. B. Campbell, T. Meehan, K. A. Clark, D. McMorrow, C. Dyer, C. Sanderson, C. Comber, and S. Kuboyama, "Investigation of single-ion multiple-bit upsets in memories on board a space experiment," *IEEE Transactions on Nuclear Science*, vol. 47, no. 3, pp. 705–711, 2000.

[29] A. Bougerol, F. Miller, N. Guibbaud, R. Gaillard, F. Moliere, and N. Buard, "Use of laser to explain heavy ion induced SEFIs in SDRAMs," *IEEE Transactions on Nuclear Science*, vol. 57, no. 1, pp. 272–278, 2010.

[30] K. A. LaBel, P. W. Marshall, J. L. Barth, R. B. Katz, R. A. Reed, H. W. Leidecker, H. S. Kim, and C. J. Marshall, "Anatomy of an in-flight anomaly: investigation of proton-induced SEE test results for stacked IBM DRAMs," *IEEE Transactions on Nuclear Science*, vol. 45, no. 6, pp. 2898–2903, 1998.

[31] R. Harboe-SØrensen, F.-X. Guerre, and G. Lewis, "Heavy-ion see test concept and results for ddr-ii memories," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2125–2130, 2007.

[32] K. Grürmann, M. Herrmann, F. Gliem, H. Schmidt, G. Leibeling, H. Kettunen, and V. Ferlet-Cavrois, "Heavy ion sensitivity of 16/32-Gbit NAND-Flash and 4-Gbit DDR3 SDRAM," in *2012 IEEE Radiation Effects Data Workshop*, 2012, pp. 1–6.

[33] R. H. Maurer, K. Fretz, M. P. Angert, D. L. Bort, J. O. Goldsten, G. Ottman, J. S. Dolan, G. Needell, and D. Bodet, "Radiation-induced single-event effects on the Van Allen Probes spacecraft," *IEEE Transactions on Nuclear Science*, vol. 64, no. 11, pp. 2782–2793, 2017.

[34] S. Wicker, *Error control systems for digital communication and storage*. Englewood Cliffs, NJ, USA: Prentice Hall, 1995.

[35] D. Strukov, "The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories," in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, 2006, pp. 1183–1187.

[36] Generator polynomial of bch code. [Online]. Available: https://www.mathworks.com/help/comm/ref/bchgenpoly.html

[37] A. Saleh, J. Serrano, and J. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *IEEE Transactions on Reliability*, vol. 39, no. 1, pp. 114–122, 1990.

[38] P. Reviriego, J. A. Maestro, and C. Cervantes, "Reliability analysis of memories suffering multiple bit upsets," *IEEE Transactions on Device and Materials Reliability*, vol. 7, no. 4, pp. 592–601, 2007.

[39] P. Reviriego and J. A. Maestro, "Study of the effects of multibit error correction codes on the reliability of memories in the presence of MBUs," *IEEE Transactions on Device and Materials Reliability*, vol. 9, no. 1, pp. 31–39, 2009.

[40] T. J. Dell, *A White Paper on the Benefits of Chipkill Correct ECC for PC Server Main Memory*, IBM, 1997.

[41] Embench™: A modern embedded benchmark suite. [Online]. Available: https://www.embench.org/

[42] Coremark™-pro. [Online]. Available: https://www.eembc.org/coremark-pro/