MASTER

A framework for generating and evaluating error correcting memory controller designs

Visser, Michiel S.

Award date:
2022

A framework for generating and evaluating error correcting memory controller designs

Master Thesis

Michiel Visser
(0956851)

Committee members:
dr. ir. R. (Roel) Jordans
M.L.P.J. (Martijn) Koedam, MSc
prof. dr. ir. C.H. (Kees) van Berkel
dr. A. (Alberto) Ravagnani

February 2, 2022

Abstract

Each year many products are created that rely on microcontrollers to operate. These microcontroller allow the product to implement complex behaviour and new features. Unfortunately, many of these microcontroller designs are vulnerable to errors caused by cosmic radiation. Especially the memory in these chips is vulnerable as it often makes up a large part of the chip and is built to the absolute limits of the chip technology.

In this thesis the design and implementation of two tools for creating and evaluating error correcting memory controllers are described. The first tool is built to generate hardware designs of error correcting memory controller with different error correction codes and memory controller designs. This tool is built to allow for easy expansion of the supported error correction codes by defining these error correction codes in their standard matrix form.

The second tool is a simulator which can simulate complete system designs containing these error correcting memory controllers. In this simulator, errors can be injected into the simulated memory to test the memory controller when errors occur. Using the results of these simulations the error correction effectiveness of a specific error correcting memory controller can be determined.

To show the functionality of these tools, this thesis also uses these tools to experiment with a number of different error correction codes and memory controller designs. The standard parity, Hamming, extended Hamming and Hsiao codes are explored. Furthermore two error correction codes with adjacent error correction by Dutta and Touba, and She and Li are explored. These error correction codes are combined with three different memory controller designs, a basic design, a design with write-back on error support, and a design which automatically refreshes the memory.

The error correction effectiveness and performance overhead of these designs is shown using the simulator built in this thesis. For completeness, ASIC layouts are created of these designs using open-source tools, to determine the critical path delay, area and power usage of these designs.

The results show that the refreshing controller has the best error correction effectivity, followed by the write-back on error controller. For the error correction codes, the Hsiao code performed the best in all test, while the parity code gives great value for a small and simple implementation. The codes with adjacent error correction were only of value in the case where adjacent errors actually occurred, as the other error correction codes performed significantly worse.

Contents

Chapter 1

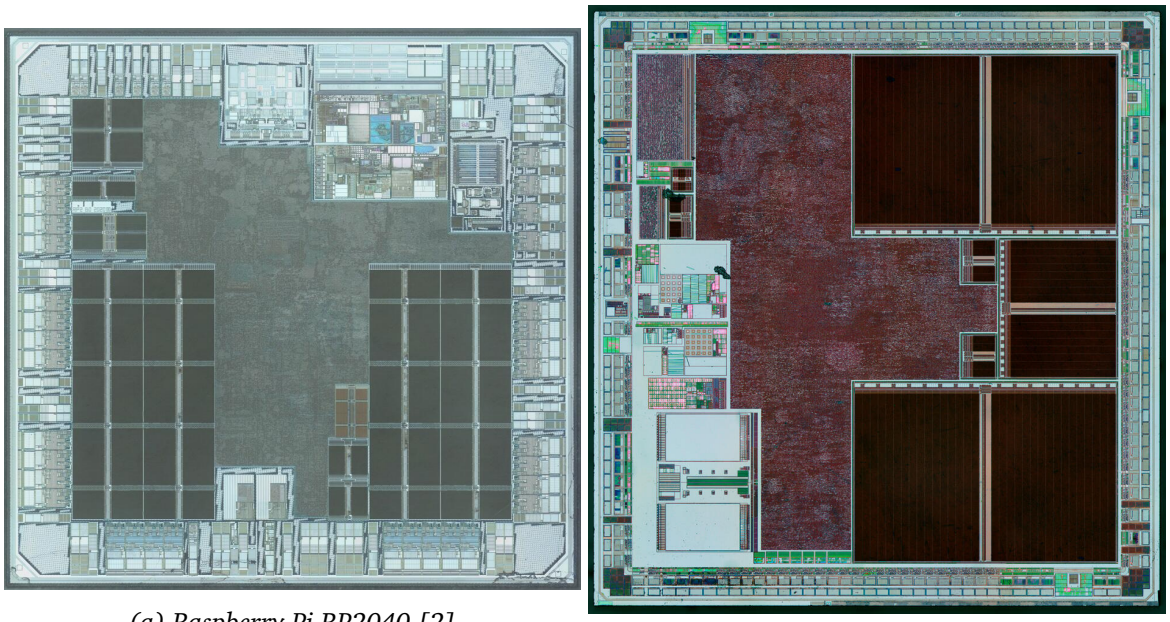
Introduction

Microcontrollers are found in almost every electronic device you have ever interacted with. They are used in almost all consumer electronics, like home appliances, smart home devices and printers. Furthermore, almost all cars contain many tens, sometimes even hundreds of microcontrollers. And a large number of medical and industrial devices are also powered by microcontrollers. In short, they are basically everywhere. To get an impression of how many microcontrollers there are, in the last quarter of 2020 an enormous 4.4 billion ARM Cortex-M microcontrollers were produced [1]. And ARM microcontrollers are not alone, there are a dozen more common microcontrollers brands, and probably another thirty less common microcontrollers brands.

Many of these applications we would consider critical, that is, failures in these systems could lead to serious physical harm, damage to property or to other undesired outcomes, which would be considered as very bad. To prevent the microcontrollers in these critical applications from causing failures, a lot of effort is put into making the software running on the microcontrollers as reliable as possible. Unfortunately, all software methods of improving the reliability rely themselves on the reliability of the underlying hardware. If the reliability of the underlying hardware is not guaranteed, the reliability of the software running cannot be guaranteed.

Guaranteeing the reliability of the hardware is most likely impossible. Microcontrollers can fail both in the logic that makes up the processor and the memory that contains all the instructions and data. These hardware failures can be divided into three categories, manufacturing defects, process variation and degradation, and environmental transients. Manufacturing defects are permanent and will break a specific part of a chip causing it to be non-functional. Some other permanent failures are only experienced when the chip is operating at a high clock speed or high temperature. These failures can be caused by process variation during manufacturing, or excessive heat or radiation during operation, causing degradation to the chips. Finally, there are transient failures caused by the environment.

These transient environmental failures are caused by ionizing or electromagnetic radiation, and only affect the microcontroller for a very short time, almost always less than one clock cycle. Ionizing radiation is often caused by cosmic rays or radioactive impurities in materials used in the chip or packaging, while electromagnetic radiation can be caused by lightning or other electrical systems. Such a radiation induced failure will cause a temporary flow of



(a) Raspberry Pi RP2040 [2]

(b) GigaDevice GD32F103CBT6 [3]

Figure 1.1: Die photos of common microcontroller.

current in a wire inside the chip. This current will either cause a glitch in the signal on the wire, or it can cause a memory element to change state, if the memory element is hit just right. Temporary glitches in the signal on a wire are often not much of a problem, they only cause real problems if they occur just at the clock edge. However, when they occur near a clock edge they mimic the behaviour of a failure in a memory element, since the incorrect value is clocked into a memory element at the clock edge.

Inside these chips there are two types of memory elements. First of all, there are flip-flops, which are used inside the main processor part of the microcontroller. And secondly, there is Static Random Access Memory (SRAM), which is the main memory of the microcontroller, and is used to store the data and instructions that the processors is using. This work focusses on the reducing failures in the SRAM blocks, because they often take up a large portion of the chip, as can be seen in the die photos of the Raspberry Pi RP2040 microcontroller in fig. 1.1a and the GigaDevice GD32F103 in fig. 1.1b. In these photos, the larger solid dark coloured areas are the SRAM blocks, which take up approximately 45% of the main chip area.

Furthermore, SRAM is built closer to the limits of the technology used. Building SRAM cells at the absolute limit of a technology can help reduce the size of the SRAM, since a tiny improvement in a single bit cell is multiplied out to many millions of bit cells. Unfortunately, this also means that the cells themself are more susceptible to environmental influences, since the cells are built with less noise margin to get better density.

To see the impact of these environmental effects, the estimated rate of failure can be calculated. Baumann [4] estimates the failure rate for processors with multi-megabit SRAMs around $50\,000$ FIT. That is $50\,000$ failures every 1 billion device hours, or approximately one

error every two and a half years. For a single microcontroller this error rate is not too terrible, however when looking at the complete population of microcontrollers this adds up quickly. Take for example the 4.4 billion ARM microcontrollers produced in the last quarter of 2020. If all of these microcontroller are in use, there would be approximately 61 errors every second, which would certainly be undesirable. Fortunately, most of these errors would not actually cause an actual failure, however even at one failure per second the impact is still significant.

Another study conducted at Google [5], measuring the error rate in DRAM, which is another type of memory, shows similar error rates of between $25\,000$ and $75\,000$ FIT. While these numbers do not directly apply to SRAM, they show that errors are not a rare phenomenon.

To improve the reliability of microcontrollers we would like them to be resilient against these environmental effects. There are two approaches to achieving resiliency against environmental factors. First of all, there are SRAM bit cell designs which are not built to the absolute limit of the technology, or are even built specially to be hardened against environmental factors. This approach offers up density in the memory for improved reliability. However, there is another approach, which uses the dense SRAM, while also protecting against errors. This approach is adding error correction to the logic that is using the memory. Error correction logic will use extra redundant bits to encode extra information that can be used to correct errors in the data if they occur. Adding these redundancy bits will also increase the size of the SRAM memory, however the redundancy often increases the memory size by a smaller factor than the hardened SRAM cells.

These reliability features are already quite common in the aerospace industry and large server processors. In these industries the need for error correction greatly outweighs the implementation cost. Aerospace applications simply require it, since they have to deal with more radiation and because a failure can be extremely expensive, or could cause harm to many people. In large server processors the error correction is used for caches and the main memory. Since these processors are already expensive, the extra cost of error correction does not impact the price much.

More recently STMicroelectronics started adding error checking and error correction to their STM32G0 and STM32G4 lines of microcontrollers. This shows that even for general purpose microcontrollers adding error detection and correction is becoming more common.

1.1 Problem statement

The goal of this research is to build a framework of tools which can generate and evaluate error correcting memory controllers. These tools should be able to generate a hardware design based on a number of design parameters, such as the error correction code, number of data bits, and the memory controller type. Furthermore, these tools should be able to evaluate the error correction effectiveness and performance of the generated hardware designs. And most importantly, these tools should be easily extended by future researchers so they are able to research different error correction codes or memory controllers, which were not considered in this work.

Finally, to evaluate the usefulness and performance of these tools, they should be used to generate and evaluate a number of error correcting memory controller designs. These designs should be evaluated on error correction effectiveness and performance, but also the implementation area, critical path and power usage, which will be done using available tools. Using these evaluations should allow us to determine the trade-offs between different designs.

1.2 Contributions

In this thesis two major contributions are made. First of all, a tool is designed and implemented that can generate hardware designs for error correcting memory controllers. This tool is designed for easy expansion of the supported error correction codes and memory controller designs to allow further research to be conducted using it. Second, a simulator is designed and implemented that can simulate a complete processor designs, while injecting errors into the memory it is using. Using this simulator a number of statistics regarding error correction effectiveness and performance overhead can be collected. This simulator is built on top of the existing CXXRTL simulator framework from Yosys.

Next to the design and implementation of these tools, they are also used to conduct some research into a number of well known error correction codes and memory controller designs. For these error correction memory controller designs the error correction effectiveness and performance overhead are measured using the simulator. However, a number of other open-source tools are used to determine the area, critical path delay and power usage of the designs when implemented as an ASIC. This research shows the capabilities of the tools created in this thesis, as well as the capabilities of the open-source ASIC flow.

1.3 Document structure

This thesis is made up of four content chapters, followed by three appendices containing additional in-depth information. Chapter 2 explains the required background information on error correction and the hardware implementation of these error correcting memory controllers. Chapter 3 describes the design and implementation of the two tools, and the experimental setup for the research done using these tools. Chapter 4 lists the results of the experiments conducted using the tools, and the performance measurements of the tools themselves. Finally, chapter 5 concludes this thesis and lists a number of areas where future work is possible.

Appendices A and B contain in-depth background information on matrix transformations and error correction codes. Appendix C contains some additional research conducted using these tools, which is less interesting and therefore left out of the main body of this thesis.

Chapter 2

Background and related work

In this chapter the background information on error correction and memory controller design is explained. First of all, the basics concepts and classification of error correction codes are explained. Second, the matrix notation describing error correction codes is introduced and the error correction codes used in this research are listed. Third, the implementation of error correction codes in hardware is explained. Finally, a number of memory controller design choices are discussed, and available memory controller designs are explored.

2.1 Error detection and correction

Numerous papers have been written about error correction codes, both about error correction codes in general, and specifically for memory error correction. Error correction itself goes back all the way to 1950, when Richard Hamming introduced the first error correcting code, which is now known as the Hamming code. However, error correction codes specifically targetted towards memory applications have only become popular in the last two decades.

Error correction is often specified in terms of messages, which are transmitted over a noisy channel, to a different point in space. However, in this research the interest is on error correction for memory applications, where the data is not transmitted, but instead stored in memory for length of time. Fortunately, this distinction between transmission over space or storage over time does not actually matter. Storing a message in memory can be seen as a transmission over time, instead of over space.

2.1.1 Linear Block Codes and Convolutional Codes

In the field of error correction there are two main classes of error-correction codes. First of all, there are the Linear Block Codes, which are error-correction codes that operate on a fixed size message with a fixed number of symbols, the block. Secondly, there are Convolutional Codes, which instead operate on a stream of symbols, without a fixed message length.

Another important distinction between these types of codes is that Convolutional Codes are

inherently sequential, which means that efficiently encoding and decoding using these codes will take many cycles, while Linear Block Codes are more parallel and can be computed for the complete block at once.

In this research error correction is applied to memory applications, where it is infeasible to use these Convolutional Codes due to their sequentialness. Accessing memory should only take a limited number of cycles, often only one or two cycles, which is hard to achieve efficiently using Convolutional Codes. Therefore, in this research the focus is on Linear Block Codes, which can provide encoding and decoding in only the limited number of clock cycles.

2.1.2 Hamming Distance

For error correction codes the Hamming distance is one of the most important parameters, specifically the minimum Hamming distance between two valid messages. The Hamming distance is specified as the minimum number of bits that have to change to get from one message to another message.

Figure 2.1 shows three examples of Hamming distance. In these figures the messages that are considered valid are coloured green, while the messages that are invalid are coloured red. When the Hamming distance is one, as in fig. 2.1a, there are at least two valid messages that only differ by one bit. Changing a single message from valid to invalid does not affect the Hamming distance, since there are still messages that are only a single bit-flip apart.

Figure 2.1b shows a distribution of valid messages with a Hamming distance of two. In this figure moving from any valid message to any other valid message takes at least two bit-flips. Similarly, in fig. 2.1c the Hamming distance is three, so moving from any valid message to any other valid message takes three bit-flips.

In the case where the messages do not use any form of error detection or correction the minimum Hamming distance is one. This means that changing a single bit in the message will result in another valid message. In this case it is impossible to distinguish between a valid message and a message that got changed, since the change will always result in another valid message.

In a messaging scheme with a Hamming distance of two, a single bit error can actually be detected. Take for example fig. 2.1b, if you start at any of the valid messages, and change a single bit, you will end up at an invalid message. Therefore, a single bit error can be detected by checking if a message is considered valid. Unfortunately, this does not allow for error correction, since invalid messages can be created by a bit-flip from multiple valid messages.

To achieve single bit error correction a messaging scheme with a Hamming distance of three is needed. An example of such a messaging scheme is shown in fig. 2.1c. Here any single bit-flip in a valid message will still end up in an invalid message, which means that a single bit error can still be detected, but interestingly the invalid messages can be split into two groups, depending on the valid message that was changed. This means that seeing an invalid message left of the center can be interpreted as "000", while any invalid message right of the

center can be interpreted as as "111". In this messaging scheme any single bit error can be correctly restored into the original message.

The error detection and correction capabilities of a messaging scheme can be generalized based on the Hamming distance of the messaging scheme. A messaging scheme can be k error detecting if and only if the Hamming distance d is at least $k + 1$. For it to be k error correcting the Hamming distance d must be at least $2k + 1$.

2.1.3 Error correction code classification

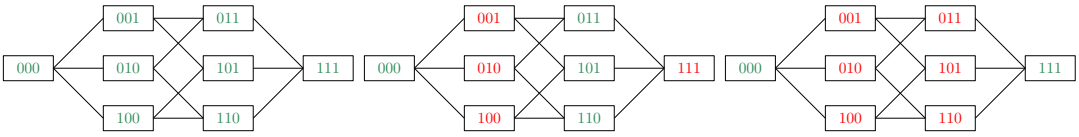
Error correction codes are often classified by the types of errors that it can correct and detected. The Hamming distance can be used to determine the number of errors a code can detect or correct, however this is only true for a number of completely random errors. Some error correction codes can actually detect a subset of those random errors. To classify these types of codes, there are a number of abbreviations.

Error detection capabilities are marked with xED, where x is the number of random errors that can be detected by this code. For one, two and three errors these are marked SED, DED and TED respectively, which stands for Single, Double and Triple. For higher numbers of errors x will often simply be the number.

Error correction capabilities are marked with xEC, where x is the number of random errors that can be corrected by this code. As with the error detection, these are marked SEC, DEC and TEC for one, two and three errors.

More exotic codes have special error detection or correction capabilities, such as adjacent error detection or correction. Adjacent error detection or correction allows the code to detect or correct an error which has flipped x adjacent bits. These codes are marked with xAED and xAEC for detection and correction respectively. Note that the errors these codes target require that all x adjacent bits are flipped by an error. Therefore, a triple adjacent error correcting (TAEC) code cannot correct an error where one bit is flipped, one bit is unaffected, and again one bit is flipped, even though all three bits are adjacent.

For those cases where such a gap in the errors might exist, there are almost adjacent error detecting and correcting codes. These codes are marked with DAAED or DAAEC, when they detect two flips with a single space between them. For higher number of flips, the definition of almost adjacent becomes quite unclear and therefore there are no standard abbreviations.



(a) $d = 1$ *(b)* $d = 2$ *(c)* $d = 3$

Figure 2.1: Examples of Hamming distance where valid messages are in green.

Finally, some error correction code that will be discussed in this chapter might be partially able to detect or correct some number of errors. These will be prefixed with a lowercase p for partial. Specifically, there are a number of codes which can only partially detect double errors, those will be marked with pDED to differentiate them from the actual DED codes.

Next to the type classification, the number of bits used in the code is often presented as (n, k), where n is the total number of bits in an encoded message, and k is the size of the input message. The total number of parity bits used by this code can be calculated as $n - k$.

2.1.4 Parity-check Matrix and Generator Matrix

There exist a large number of Linear Block Codes, all with their own definitions and tricks to improve the error correction capabilities. To ease communication and exchange of these error correction codes they are often expressed as the parity-check matrix. The parity-check matrix, often referred to as $\boldsymbol{H}$, describes the relationships between symbols in the encoded message $\boldsymbol{c}$, specifically

$$\boldsymbol{H} \cdot \boldsymbol{c}^T = \boldsymbol{0}. \tag{2.1}$$

Take for example the parity check matrix

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \tag{2.2}$$

each row represents a relationship between bits in the encoded message,

$$c_1 + c_2 = 0 \tag{2.3}$$
$$c_1 + c_3 = 0. \tag{2.4}$$

If the encoded message does not obey these relationships, the encoded message cannot be a valid encoding, which means that an error has occurred. These matrices can exist over any group of symbols, however for this research it is assumed that the matrices are binary, unless otherwise noted.

Using the parity-check matrix a very simple and general decoding scheme can be realized, the syndrome decoding algorithm. This decoding algorithm relies on the $\boldsymbol{H} \cdot \boldsymbol{c}^T = \boldsymbol{0}$ relationship. If the message arrives without any errors, the calculation $\boldsymbol{H} \cdot \boldsymbol{c}^T$ will actually result in the zero vector, so the data bits can be taken from the encoded message without any changes. Otherwise, if the message did have errors, the encoded message $\boldsymbol{x}$ will be a combination of the original message $\boldsymbol{c}$ and an error vector $\boldsymbol{e}$, which can be expressed as $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{e}$. Multiplying the encoded message $\boldsymbol{x}$ with the parity-check matrix will produce $\boldsymbol{H} \cdot \boldsymbol{e}^T$ as follows

$$\boldsymbol{H} \cdot \boldsymbol{x}^T = \boldsymbol{H} \cdot (\boldsymbol{c} + \boldsymbol{e})^T = \boldsymbol{H} \cdot \boldsymbol{c}^T + \boldsymbol{H} \cdot \boldsymbol{e}^T = \boldsymbol{0} + \boldsymbol{H} \cdot \boldsymbol{e}^T = \boldsymbol{H} \cdot \boldsymbol{e}^T. \tag{2.5}$$

The result of $\boldsymbol{H} \cdot \boldsymbol{e}^T$ is the so called syndrome, which will be mapped to $\boldsymbol{e}$ to determine the bit error locations. For a single-bit error in position i the syndrome will be column i from the original $\boldsymbol{H}$ matrix, which limits checking to just the columns of $\boldsymbol{H}$. If there is more than

one error, for example an error in bit i and j, where $i \neq j$, the syndrome will be the linear combination of the columns i and j from the matrix $\boldsymbol{H}$.

Unfortunately, this does not mean that any number of errors can be detected and corrected with an arbitrary parity-check matrix $\boldsymbol{H}$. The linear combinations of columns will need to be unique to correct errors. If a linear combination can be created from different groups of columns, it can only be used to detect errors, but not correct, since the original combination of errors can differ.

Take for example the matrix $\boldsymbol{H}$ in eq. (2.2), in this matrix all three columns are distinct and non-zero, therefore this code is at least able to correct a single error. However, this code cannot detect or correct more than a single error. This can be seen by taking column 1 and 2, where the linear combination would result in $[\begin{smallmatrix}0\\1\end{smallmatrix}]$, which is column 3 of this matrix. So, if an error occurred in both bit 1 and 2, the decoder would think the error is in bit 3.

Now consider the following different parity-check matrix

$$\boldsymbol{H}' = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \tag{2.6}$$

In this matrix all four columns are distinct and non-zero, therefore this code is able to correct a single error. However, with this new parity-check matrix an additional error can actually be detected. When taking the combination of any two columns the list

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{2.7}$$

of column vectors will be produced. As can be seen these column vectors are not unique, therefore this code cannot actually correct two random errors, however all of these column vectors are unique from the original columns in the matrix $\boldsymbol{H}'$. Therefore, if the syndrome is ever one of these column vectors, there must be two errors in the encoded message, but the decoder does not have enough information to determine where exactly.

Finally, the parity-check matrix $\boldsymbol{H}$ also has a counterpart, which is known as the generator matrix, often referred to as $\boldsymbol{G}$. The generator matrix describes the relationship between an input message and the corresponding encoded message

$$\boldsymbol{m} \cdot \boldsymbol{G} = \boldsymbol{c}, \tag{2.8}$$

where $\boldsymbol{m}$ is the input message and $\boldsymbol{c}$ is the encoded message. The generator matrix can easily be constructed from the parity-check matrix, if the parity-check matrix is in standard form. A parity-check matrix in standard form

$$\boldsymbol{H} = \left[\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{I}_{n-k} \end{array}\right] \tag{2.9}$$

consists of two parts, a sub-matrix $\boldsymbol{A}$ and an identity matrix with size $n - k$, where n is the total length of the encoded message and k is the number of data bits in the message. With the

parity-check matrix in standard form, the corresponding generator matrix in standard form is defined as

$$\boldsymbol{G} = \left[\begin{array}{c|c} \boldsymbol{I}_k & -\boldsymbol{A}^T \end{array} \right], \tag{2.10}$$

where $\boldsymbol{A}$ is the sub-matrix from the parity-check matrix $\boldsymbol{H}$ and $\boldsymbol{I}_k$ is an identity matrix of size k, where k is the number of data bits in the message. For the case of binary codes this can actually be simplified to

$$\boldsymbol{G} = \left[\begin{array}{c|c} \boldsymbol{I}_k & \boldsymbol{A}^T \end{array} \right], \tag{2.11}$$

where there is no longer a negation of the matrix $\boldsymbol{G}$, since that negation does nothing for a binary matrix. As an example, take the parity-check matrix $\boldsymbol{H}$ from eq. (2.2), since it does follow the standard form defined in eq. (2.9), it can be converted to a generator matrix in standard form as defined in eq. (2.10). The resulting binary generator matrix

$$\boldsymbol{G} = \left[\begin{array}{ccc} 1 & 1 & 1 \end{array} \right] \tag{2.12}$$

consists of only a single row, since there is only a single data bit in this code. To generate a valid message for this code, a bit has to be repeated three times. This code is also known as the triple repetition code. This same scheme does unfortunately not apply to $\boldsymbol{H}'$ defined in eq. (2.6), since this parity-check matrix is not in standard form.

At last, a general relation between the generator matrix $\boldsymbol{G}$ and the parity-check matrix $\boldsymbol{H}$ can be constructed. Using eqs. (2.1) and (2.8) shows that

$$\boldsymbol{0} = \boldsymbol{H} \cdot \boldsymbol{c}^T = \boldsymbol{H} \cdot (\boldsymbol{m} \cdot \boldsymbol{G})^T = \boldsymbol{H} \cdot \boldsymbol{G}^T \cdot \boldsymbol{m}^T. \tag{2.13}$$

Since the message $\boldsymbol{m}$ can take any value, the following relation can be derived

$$\boldsymbol{H} \cdot \boldsymbol{G}^T = \boldsymbol{0}. \tag{2.14}$$

Using this relation pairs of generator and parity-check matrices can be verified as this relation should always hold for any code.

Unfortunately, the generator and parity-check matrices are not always in standard form. Using the method described in appendix A, these non-standard matrices can be transformed into standard form.

2.1.5 Error detection and correction codes

During this research a number of error correction codes will be used. Table 2.1 lists all of the codes used, with the error detection and correction capabilities of the specific codes listed. For an in-depth explanation of the specific error detection and correction codes refer to appendix B. This appendix also explains the methods required to construct the generator and parity-check matrices for each of these codes.

The parity, Hamming and extended Hamming codes are the fundamental codes on which the base of error correction is built. The Hsiao code is an alternative for the extended Hamming

Name	Type
Parity	SED
Hamming [6]	SEC
Extended Hamming	SEC-DED
Hsiao [7], [8]	SEC-DED
Dutta-Touba [9]	SEC-pDED-DAEC
She-Li [10]	SEC-DAEC-DAAEC-TAEC

Table 2.1: Error detection and correction codes used in this research.

code, with improved performance when implemented in hardware, due to its more optimal parity-check matrix. Finally, the Dutta-Touba and She-Li codes allow for the correction of errors when they occur in adjacent memory bits. These are useful with memories created on smaller technology nodes, as these might experience these types of error more commonly.

2.1.6 Implementing error correction in hardware

So far, all of these error correction codes have been defined using their respective generator and parity-check matrices. While this way of expressing the error correction codes is useful for theoretical understanding, it does not directly show a hardware implementation. However, actually creating a hardware implementation of these error correction codes can be done systematically from these matrices.

In this case only the typical fully combinatorial implementations of the encoder and decoder are discussed. For some types of error correction codes it might be possible to implement a sequential encoder and decoder, however these are out of the scope of this research. From the generator matrix $\boldsymbol{G}$ the hardware implementation of the encoder can be derived, and from the parity-check matrix $\boldsymbol{H}$ the hardware implementation of the decoder.

Encoder

The goal of the encoder is to implement the encode equation, $\boldsymbol{m} \cdot \boldsymbol{G} = \boldsymbol{c}$ (eq. (2.8)), in hardware, where $\boldsymbol{m}$ is a k-bit binary vector containing the input message and $\boldsymbol{c}$ is a n-bit binary vector containing the encoded message. The generator matrix $\boldsymbol{G}$ is a binary $k \times n$ matrix. From this equation the equation for a specific column in $\boldsymbol{c}$ can be defined as

$$\boldsymbol{c}_x = \boldsymbol{m}_1 \cdot \boldsymbol{G}_{1,x} + \boldsymbol{m}_2 \cdot \boldsymbol{G}_{2,x} + \ldots + \boldsymbol{m}_k \cdot \boldsymbol{G}_{k,x}. \tag{2.15}$$

Since both the generator matrix and the input message are binary matrices, this equation can be simplified by replacing the multiplication and addition by their binary equivalents binary "and" and "xor", which results in

$$\boldsymbol{c}_x = (\boldsymbol{m}_1 \wedge \boldsymbol{G}_{1,x}) \oplus (\boldsymbol{m}_2 \wedge \boldsymbol{G}_{2,x}) \oplus \ldots \oplus (\boldsymbol{m}_k \wedge \boldsymbol{G}_{k,x}). \tag{2.16}$$

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(a) Generator matrix

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(b) Parity-check matrix

Figure 2.2: Matrices for an (8,4) Extended Hamming code.

$$\begin{array}{c} \\ \boldsymbol{m}_1 \\ \boldsymbol{m}_2 \\ \boldsymbol{m}_3 \\ \boldsymbol{m}_4 \\ \\ \end{array} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \begin{array}{c} \\ \boldsymbol{c}_1 \; \boldsymbol{c}_2 \; \boldsymbol{c}_3 \; \boldsymbol{c}_4 \; \boldsymbol{c}_5 \; \boldsymbol{c}_6 \; \boldsymbol{c}_7 \; \boldsymbol{c}_8 \\ \\ \\ \\ \\ \end{array}$$

(a) Annotated generator matrix

$$\begin{array}{c} \boldsymbol{m}_1 \\ \boldsymbol{m}_2 \\ \boldsymbol{m}_3 \\ \boldsymbol{m}_4 \\ \boldsymbol{c}_5 \end{array} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \rightarrow \begin{array}{l} \boldsymbol{c}_5 = (\boldsymbol{m}_1 \wedge 0) \oplus (\boldsymbol{m}_2 \wedge 1) \oplus \\ \qquad (\boldsymbol{m}_3 \wedge 1) \oplus (\boldsymbol{m}_4 \wedge 1) \\ \qquad \downarrow \\ \boldsymbol{c}_5 = \boldsymbol{m}_2 \oplus \boldsymbol{m}_3 \oplus \boldsymbol{m}_4 \end{array}$$

(b) Example derivation of $\boldsymbol{c}_5$

Figure 2.3: Encoder example for an (8,4) Extended Hamming code.

However, since the generator matrix is constant and known when building the hardware, this equation can be simplified even more, to

$$\boldsymbol{c}_x = \boldsymbol{m}_a \oplus \boldsymbol{m}_b \oplus \ldots \oplus \boldsymbol{m}_z, \tag{2.17}$$

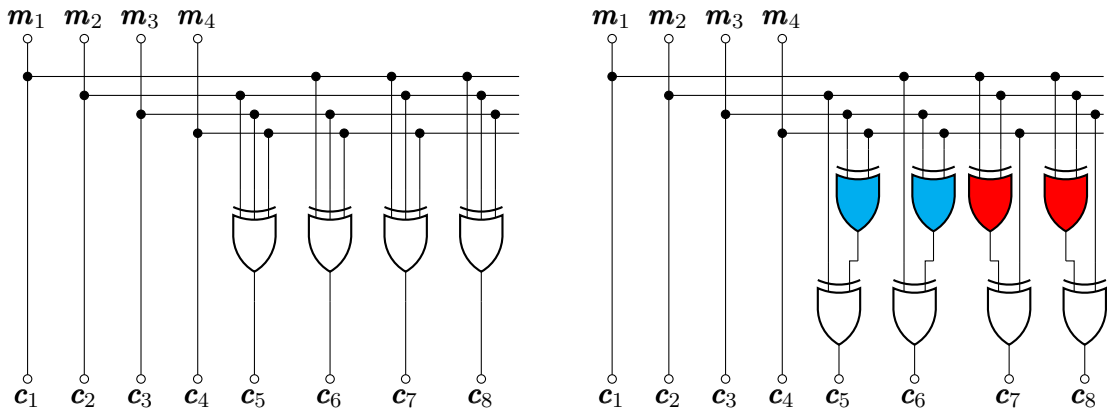
where the columns of $\boldsymbol{m}$ are selected if the matching value in $\boldsymbol{G}$ is one.

Intuitively, this means that we can determine the output bit with index x from column x of the generator matrix and all the input bits. To determine the output bit, one takes all input bits for which their respective row in the generator matrix column is one. As an example, take the generator matrix in fig. 2.2a. Figure 2.3a shows an annotated version of the generator matrix, with the rows marked with the corresponding input bits and the columns marked with the corresponding output bits. As an example, the derivation of the fifth encoded bits is shown in fig. 2.3b.

Using this method all the bits of the encoded message can be determined from the input message, which means the encoded message can be defined as

$$\boldsymbol{c} = \begin{bmatrix} \boldsymbol{m}_1 \\ \boldsymbol{m}_2 \\ \boldsymbol{m}_3 \\ \boldsymbol{m}_4 \\ \boldsymbol{m}_2 \oplus \boldsymbol{m}_3 \oplus \boldsymbol{m}_4 \\ \boldsymbol{m}_1 \oplus \boldsymbol{m}_3 \oplus \boldsymbol{m}_4 \\ \boldsymbol{m}_1 \oplus \boldsymbol{m}_2 \oplus \boldsymbol{m}_4 \\ \boldsymbol{m}_1 \oplus \boldsymbol{m}_2 \oplus \boldsymbol{m}_3 \end{bmatrix}^T. \tag{2.18}$$

From this equation you can clearly see the two parts of a generator matrix in standard form. The first four encoded bits directly map to bits from the input message, which corresponds



(a) wide XOR-gates *(b) 2-input XOR-gates*

Figure 2.4: Encoder implementation using XOR-gates for an (8,4) Extended Hamming code. Coloured gates compute the same result.

with the identity part of the generator matrix. The last four encoded bits are computed from multiple input message bits. These four bits are the additional bits that allow for the actual error correction.

Based on this equation we can now simply implement this as hardware using a number of XOR-gates. Figure 2.4a shows the intuitive design based on this equation. For this design the first four output bits are directly connected to the input bits, and the other four output bits are all connected to the output of an XOR-gate with three inputs. These XOR-gates are then correctly connected to the input bits to match the equation.

Unfortunately, three and more input XOR-gates are often unavailable when building actual chip designs. Therefore, these XOR-gates have to be translated into a tree of 2-input XOR-gates, as shown in fig. 2.4b. For the best performance, the critical path through this circuit should be minimized, which means that the maximum number of gates between input and output should be minimized. To achieve this, the trees of 2-input XOR-gates should be balanced as much as possible.

For this design the critical path delay can be approximated as

$$t_{crit} \propto \lceil \log_2(\text{max inputs}) \rceil, \tag{2.19}$$

where it is proportional to the logarithm base two of the maximum number inputs on the wide XOR-gates, as in fig. 2.4a. This logarithm represents the depth of a balanced tree of 2-input XOR-gates with the same number of total inputs. The maximum number of inputs can be determined by summing up each column in the generator matrix, and then selecting the column with the highest sum. Together this allows us to approximate the critical path delay in the encoder as

$$t_{crit} \propto \left\lceil \log_2 \left(\max_{c=1}^{n} \left(\sum_{r=1}^{k} \boldsymbol{G}_{r,c} \right) \right) \right\rceil. \tag{2.20}$$

Therefore, optimizing the performance of the encoder would require reducing the maximum of the summed columns.

With the performance optimized, one can also look at reducing the total number of gates used, to reduce the area required for the hardware implementation. As can be seen in the example in fig. 2.4b, there might be some XOR-gates that are calculating the exact same result, in this case the two pairs of XOR-gates in the top row, coloured in blue and red. The blue gates are both calculating $\boldsymbol{m}_3 \oplus \boldsymbol{m}_4$, and the red gates are both calculating $\boldsymbol{m}_1 \oplus \boldsymbol{m}_2$. These pairs of gates could be reduced to a single gate with the output connected to both places that requires the result. This type of optimization is often best left to automated tools, since they also take into account more complex issues such as the drive strength of the gates and the actual location on the chip, which impacts the routability.

Decoder

The goal of the decoder is to determine if there were any errors in the encoded message, and then correct them if possible. These operations can be split into multiple steps. First of all, the decoder has to calculate the syndrome, $\boldsymbol{s} = \boldsymbol{H} \cdot \boldsymbol{x}^T$ (eq. (2.5)), where $\boldsymbol{x}$ represents the encoded message with possible errors. The syndrome will be zero when no errors are present in the encoded message. If the syndrome is non-zero, it has to be matched to a known correctable syndrome to determine which bit to flip in the encoded message to restore the correct encoded message. With the flips calculated, they can be applied to the encoded message to correct it. Finally, the syndrome and the flips can be used to determine if there was an error, and if it was correctable or not.

The $(n-k)$-bit syndrome vector $\boldsymbol{s}$ can be calculated using the $(n-k) \times n$ parity-check matrix $\boldsymbol{H}$ and the n-bit encoded input message $\boldsymbol{x}$. Using the same derivation as with the encoder, we can define

$$\begin{aligned} \boldsymbol{s}_i &= \boldsymbol{x}_1 \cdot \boldsymbol{H}_{i,1} + \boldsymbol{x}_2 \cdot \boldsymbol{H}_{i,2} + \ldots + \boldsymbol{x}_n \cdot \boldsymbol{H}_{i,n} && (2.21) \\ &= (\boldsymbol{x}_1 \wedge \boldsymbol{H}_{i,1}) \oplus (\boldsymbol{x}_2 \wedge \boldsymbol{H}_{i,2}) \oplus \ldots \oplus (\boldsymbol{x}_n \wedge \boldsymbol{H}_{i,n}) && (2.22) \\ &= \boldsymbol{x}_a \oplus \boldsymbol{x}_b \oplus \ldots \oplus \boldsymbol{x}_z. && (2.23) \end{aligned}$$

Again, the specific selected input bits are known at the time of building the hardware, as the parity-check matrix is constant and known.

Intuitively, this means that we can derive the i-th bit of the syndrome, $\boldsymbol{s}_i$, from row i of the parity-check matrix $\boldsymbol{H}$, by calculating the exclusive-or of all input bits where the row of the parity-check matrix contains a one. As an example, take the parity-check matrix in fig. 2.2b, which is shown with annotations in fig. 2.5a. The example derivation of $\boldsymbol{s}_1$ is shown in fig. 2.5b.

$$\begin{array}{cccccccc} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3 & \boldsymbol{x}_4 & \boldsymbol{x}_5 & \boldsymbol{x}_6 & \boldsymbol{x}_7 & \boldsymbol{x}_8 \\ \left[\begin{array}{cccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right] & \begin{array}{c} \boldsymbol{s}_1 \\ \boldsymbol{s}_2 \\ \boldsymbol{s}_3 \\ \boldsymbol{s}_4 \end{array} \end{array}$$

(a) Annotated parity matrix

$$\begin{array}{cccccccc} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3 & \boldsymbol{x}_4 & \boldsymbol{x}_5 & \boldsymbol{x}_6 & \boldsymbol{x}_7 & \boldsymbol{x}_8 \\ \left[\begin{array}{cccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array}\right] & \boldsymbol{s}_1 \end{array}$$

$$\boldsymbol{s}_1 = (\boldsymbol{x}_1 \wedge 1) \oplus (\boldsymbol{x}_2 \wedge 0) \oplus (\boldsymbol{x}_3 \wedge 1) \oplus (\boldsymbol{x}_4 \wedge 0) \oplus (\boldsymbol{x}_5 \wedge 1) \oplus (\boldsymbol{x}_6 \wedge 0) \oplus (\boldsymbol{x}_7 \wedge 1) \oplus (\boldsymbol{x}_8 \wedge 0)$$

$$\boldsymbol{s}_1 = \boldsymbol{x}_1 \oplus \boldsymbol{x}_3 \oplus \boldsymbol{x}_5 \oplus \boldsymbol{x}_7$$

(b) Example derivation of $\boldsymbol{s}_1$

Figure 2.5: Encoder example for an (8,4) Extended Hamming code.

Using this method the complete syndrome can be derived as

$$\boldsymbol{s} = \begin{bmatrix} \boldsymbol{x}_1 \oplus \boldsymbol{x}_3 \oplus \boldsymbol{x}_5 \oplus \boldsymbol{x}_7 \\ \boldsymbol{x}_2 \oplus \boldsymbol{x}_3 \oplus \boldsymbol{x}_6 \oplus \boldsymbol{x}_7 \\ \boldsymbol{x}_4 \oplus \boldsymbol{x}_5 \oplus \boldsymbol{x}_6 \oplus \boldsymbol{x}_7 \\ \boldsymbol{x}_1 \oplus \boldsymbol{x}_2 \oplus \boldsymbol{x}_3 \oplus \boldsymbol{x}_4 \oplus \boldsymbol{x}_5 \oplus \boldsymbol{x}_6 \oplus \boldsymbol{x}_7 \oplus \boldsymbol{x}_8 \end{bmatrix}. \tag{2.24}$$

Using these equations for the syndrome, a hardware implementation can be designed using a number of XOR-gates. Figure 2.6 shows this implementation, where multi-input XOR-gates are already expanded into trees of 2-input XOR-gates. The approximate critical path for this section can be derived using the same method as the encoder, and results in

$$t_{crit} \propto \left\lceil \log_2 \left(\max_{r=1}^{n-k} \left(\sum_{c=1}^{n} \boldsymbol{H}_{r,c} \right) \right) \right\rceil. \tag{2.25}$$

Here the summation runs over the rows instead of the columns. Optimizing this part of the decoder would require reducing the maximum number of ones per row. This technique is used in the Hsiao code, which was discussed earlier, to improve the performance over the Extended Hamming code used in this example.

In this design there are also a number of gates that are calculating the same value, which are marked with colours. As an example, the right side of fig. 2.6 shows a simplified implementation using 8, instead of 13, XOR-gates to calculate $\boldsymbol{s}_2$, $\boldsymbol{s}_3$ and $\boldsymbol{s}_4$. Again these optimization only reduce the number of gates, but they cannot reduce the critical path delay, and they are best left to automated tools for the same reasons as mentioned earlier.

After this, the syndrome is used to calculate which bit to flip, and whether an error occurred at all. Designing this part of the hardware requires the parity-check matrix $\boldsymbol{H}$ and a list of all correctable errors. Using the list of correctable errors, a list of correctable syndromes can be calculated following the relation in eq. (2.5). The flip signal for a bit is set when the syndrome matches one of the known syndromes which indicates an error in that specific bit.

The hardware for matching the syndrome consists of a number of $(n-k)$-input AND-gates, with some of the inputs inverted. All of these syndrome matches can be calculated in parallel, so the critical path of this complete section is equivalent to that of one specific matcher. This

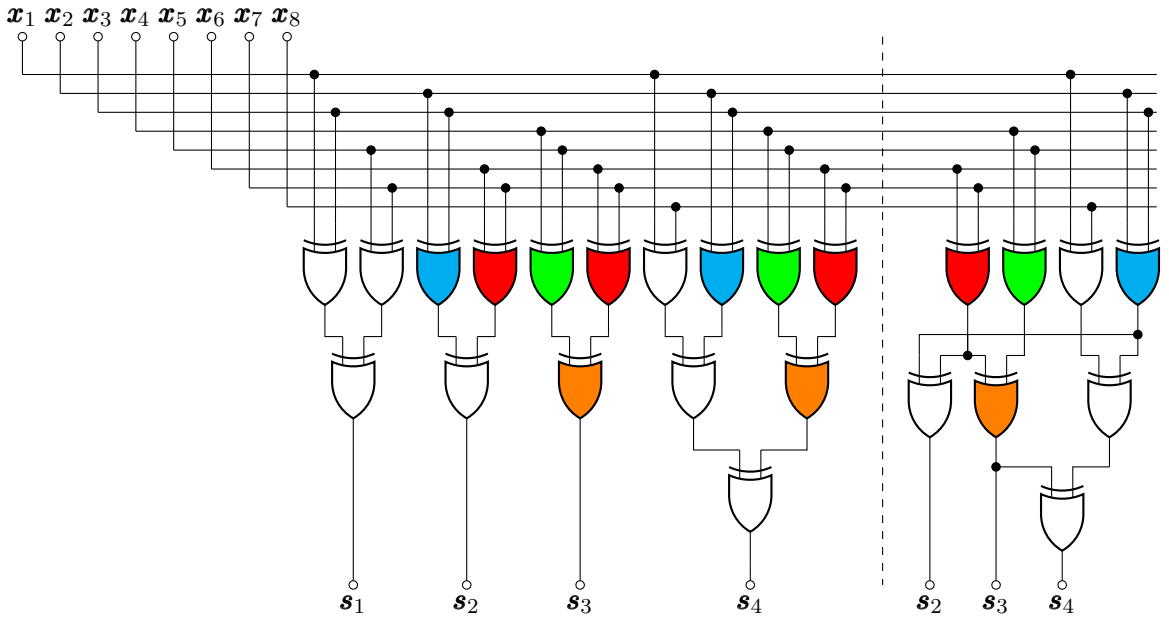


Figure 2.6: Decoder syndrome calculation implementation using XOR-gates for an (8,4) Extended Hamming code. Coloured gates compute the same result.

means that the critical path delay for the syndrome matching can be approximated as

$$t_{crit} \propto \lceil \log_2(n-k) \rceil. \tag{2.26}$$

If there are multiple syndromes that indicate a specific flipped bit, an additional section of OR-gates is required to combine the output of these syndrome matchers. The critical path delay of that section can be approximated as

$$t_{crit} \propto \lceil \log_2(\#syn) \rceil. \tag{2.27}$$

For a simple single error correcting code this results in no additional delay, however with adjacent error correction there might be three or more syndromes influencing a single flip. Finally, checking if an error occurred at all consists of a $(n-k)$-input OR-gate, which results in the same critical path delay as matching the syndromes. Since these parts are parallel, the critical path delay is not increased by this check.

Figure 2.7 shows the hardware implementation of the flip calculation and error detection, for the example Extended Hamming code. In this example no additional OR-gates are required for calculating the flip signals, as this code only support single error correction. Gates that calculate the same values are again coloured in this figure, and the same remarks about optimizations apply here.

Finally, the flips that were calculated have to be applied to the encoded input message to determine the corrected encoded message. Applying these flips simply requires a single XOR-gate for every bit, which will flip the input bit if the flip signal is high. Furthermore, the actual message bits can be extracted from the corrected encoded bits. The position of the message bits are determined by the generator matrix as seen when building the encoder. If

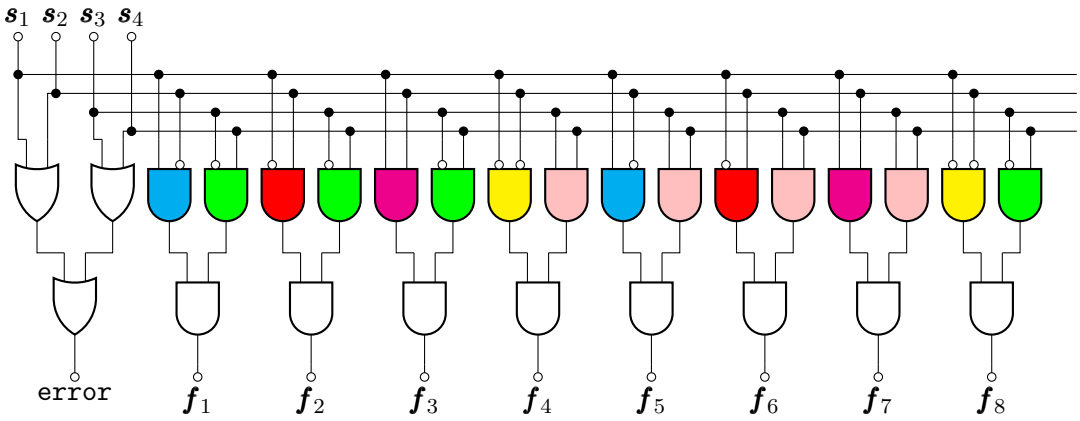


Figure 2.7: Decoder flips and error calculation implementation for an (8,4) Extended Hamming code. Coloured gates compute the same result.

the generator matrix is in standard form, the actual message bits will simply be the first k bits. Applying these flips only adds a single gate between the input and output, therefore the critical path delay though this section can be approximated as $t_{crit} \propto 1$.

Next to flipping the bits, we also have to determine if the error was uncorrectable. Doing this requires checking if there was an error and that there were no bits flipped, which means the error was not corrected. Building a circuit to detect this requires a n-input NOR-gate, and a 2-input AND gate. This means that the critical path delay though this section can be approximated as

$$t_{crit} \propto \lceil \log_2(n) \rceil + 1. \tag{2.28}$$

However, for some error correction codes simpler methods for detecting uncorrectable errors exist. In the case of the Extended Hamming code checking if the last bit of the syndrome is zero can replace checking all flips. And in the case of the Hsiao code checking the exclusive-or of all syndrome bits can replace checking all flips. In those cases the critical path delay though this section reduces to $t_{crit} \propto 1$.

Figure 2.8 shows the implementation of this section, with bit flipping XOR-gates on the left, producing the encoded message vector $\boldsymbol{c}$ and the actual decoded message vector $\boldsymbol{m}$, and the uncorrectable error calculation on the right. Here both the simplified and generate implementations of uncorrectable error detection are shown.

Together all of these sections result in a critical path from the input encoded message, through the syndrome, through the flips to the uncorrectable error output. The delay of the total critical path can be approximated as

$$t_{crit} \propto \left\lceil \log_2 \left(\max_{r=1}^{n-k} \left(\sum_{c=1}^{n} \boldsymbol{H}_{r,c} \right) \right) \right\rceil + \lceil \log_2(n-k) \rceil + \lceil \log_2(\#syn) \rceil + \lceil \log_2(n) \rceil + 1, \tag{2.29}$$

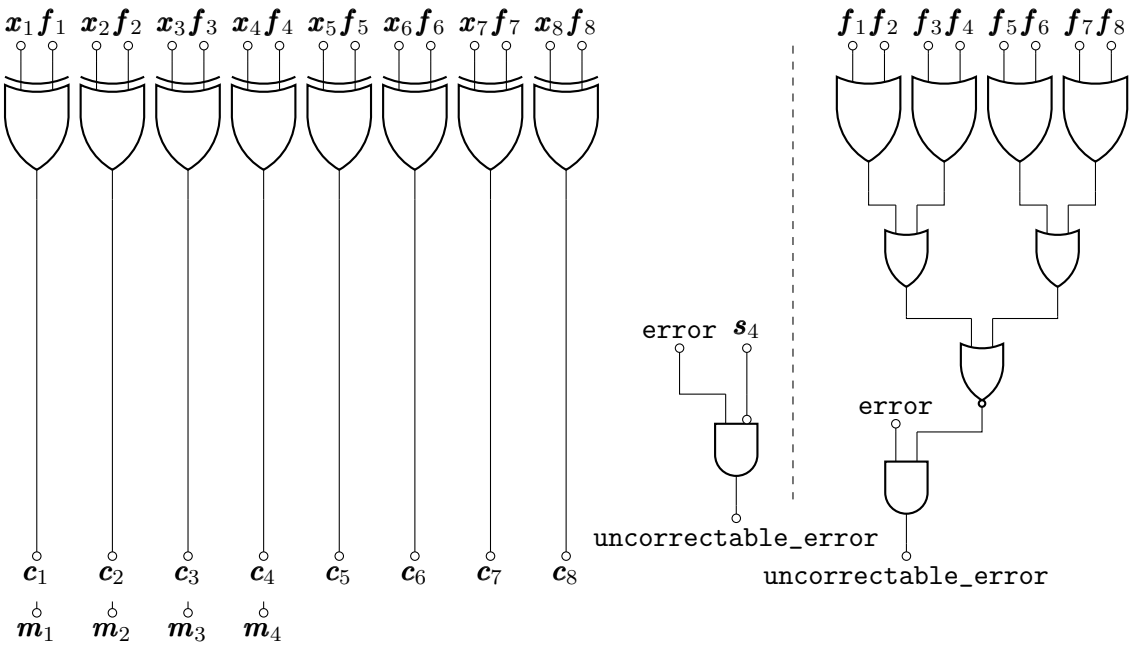


Figure 2.8: Decoder flip and double error detection implementation for an (8,4) Extended Hamming code.

or in the case of the simplified uncorrectable error detection as

$$t_{crit} \propto \left\lceil \log_2 \left(\max_{r=1}^{n-k} \left(\sum_{c=1}^{n} \boldsymbol{H}_{r,c} \right) \right) \right\rceil + \lceil \log_2(n-k) \rceil + \lceil \log_2(\#syn) \rceil + 1. \tag{2.30}$$

From these equations, the best places to target for reducing the critical path delay are the parity-check matrix and possibly simplifying the uncorrectable error detection. These places are targeted by multiple codes, such as the Hsiao code, to improve the performance of the code implementation.

These critical path delay calculations are only an approximation of the real implementation critical path delay. In a real hardware implementation the delay of different types of gates can differ quite a lot and the connections between gates can also introduce extra delay. Therefore, the actual optimization of the implementation is left to automated tools, which can optimize the circuit for the specific gates that are used. However, these calculations do show which places to target when trying the optimize an error correction code during the code design.

2.2 Memory controller design

Next to the selection of the error correction code, the design of the memory controller using the error correction code is also important. The design of the memory controller can greatly impact the effectiveness of the system as a whole. There are three major design choices when

designing a memory controller. First of all, there is the error correction policy, when and how should the error correction be applied. Secondly, updating of partial words is a concern, as this is not easily possible when using error correction. And finally, the memory controller can use a banking strategy to increase the size of the memory or improve the performance.

2.2.1 Error correction policy

The error correction policy determines when and how error correction is applied to memory operations. It determines when the encoded bits are written to the memory, and when the error correction is applied to bit read from the memory.

Read-only correction In the simplest case, the check bits are generated and stored when a write operation is issued to the memory. On a read operation the encoded message is read from the memory and error correction is applied to fix any error that might have occurred. The resulting corrected data is returned to the requester, possibly with some status signals to let the requester known if an error had occurred. At a high level this would be implemented like shown in fig. 2.9a.

While this is simple to implement and has a low performance overhead, this method comes with a major flaw. Bit errors occurring in the memory are only overwritten when a new value is written to a specific memory location. This means that over time multiple error can accumulate in a memory word. The first error might still be correctable by the error correction code on a read operation, however multiple errors are often not correctable.

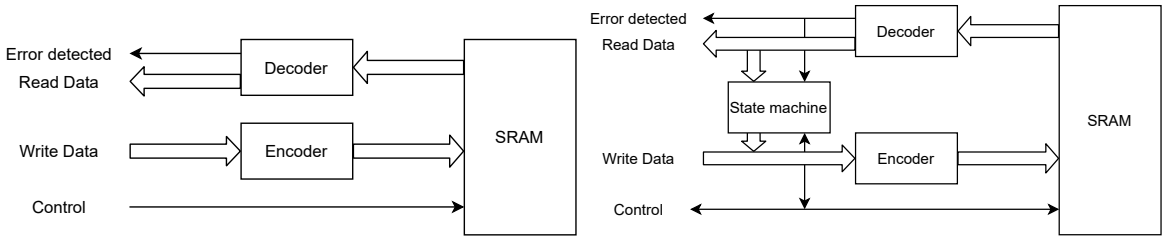
Even though this policy might be flawed, there still are a few cases where it might be good enough. First of all, the system designer might shift the responsibility of correcting the error in memory onto the processor that is using the memory. Using the error status signal the processor can decide what action is necessary to remedy the error in memory. Secondly, the values in memory might be very short lived, meaning that the chance of multiple errors happening, before the memory is overwritten, is very small and therefore an acceptable risk to the system designer. Finally, error correction might be used to correct for manufacturing defects. In this case writing the corrected value back to the memory has no point, as the memory is defective and the same error will occur on the next read operation.

Read and write correction A slightly more complex policy is to update the memory when a correctable error is detected during a read operation. Doing this will ensure that bit errors in the memory are corrected when a read operation is issued. Implementing this policy requires more hardware to execute the writeback operations. Often a small state-machine is added, as is shown in fig. 2.9b. This policy also requires stalling the requests from the processor to allow for the extra write operation to be executed. Another option is to use a dual port memory, allowing the write to take place while other operations continue, however this is often a very expensive solution, especially if errors only occur few and far between.

Implementing this policy reduces the problem of accumulating errors, however it does not

completely fix the problem. This method of removing errors in the memory is dependent on read operations issued by the processor. If a specific memory location is not read or written to in a long time, multiple errors can still accumulate causing the data to be lost.

Even though there are still problems with this policy it is often good enough. As long as the system is accessing memory locations containing useful data regularly, the memory locations will be corrected and accumulation of errors is not a problem. A system can also be designed around the constraints of this policy, by accessing important memory at a regular interval, even if the value is not necessarily needed at that moment.



(a) Simple *(b) Complex*

Figure 2.9: High-level implementation diagram of error correction policies.

Automatic refresh correction Finally, the most complex policy uses automatic memory refreshing to make sure no errors accumulate in the memory, even if memory locations are not used for a long time. This scheme is very similar to the memory refreshing in dynamic memory, however in dynamic memory no error correction is applied. To do this, a state-machine is required which will periodically read from every memory location, correcting any detected errors as it iterates through the complete memory. In [11] Siddiqui et al. show an implementation of such a memory refreshing scheme. Implementing this type of automatic refreshing results in the same high-level implementation diagram as shown in fig. 2.9b, however the state-machine for this policy is more complex than for the previous policy.

While this policy gives the best chance of preventing accumulation of errors in memory, there is still a drawback to this system. With this policy active, a constant portion of the bandwidth available to the memory will be used for refreshing. This means that there is less bandwidth available to the rest of the system, which could impact performance. To improve performance the state-machine used for refreshing can be made configurable, so that it only refreshes part of the memory or refreshes at different rates. Another option is using a multi-port SRAM, which allows the refresh state-machine to use a different port. While this reduces the bandwidth overhead, it greatly increases the cost of the SRAM.

2.2.2 Partial word updating

In embedded systems SRAM with partial write functionality is quite common. Using partial write memories with a larger word size, for example 32-bit, can be written in smaller chunks, often 8 bits. This is often used in microcontroller memories so that the a word can be accessed

every clock cycle, while still allowing updates to a single byte in memory. To support this SRAMs can enable only a subset of all write drivers, updating only those bits where the write drivers are active. However, when using error correction codes to protect the memory this becomes more complex, as writing the check bits requires knowledge of all bits in the protected word, which are not available when partially writing a word.

Avoiding the problem The simplest solution to this problem would be to simply avoid the problem altogether. This can be done by not using a single error correction code for the complete word, but instead using separate error correction for each byte which can be written. Updating the error correction check bits is always possible in the scenario, as all the data that is protected is updated at the same time.

While this is a simple solution, it is often not feasible as error correction for a single byte often uses 4 bits as check bits. In the case where a word would be 32 bits, this would mean an additional 16 bits for error correction, while a error correction code over the full 32 bits often uses only 7 or 8 bits. Theoretically, doing error correction separately for all four bytes would result in slightly better protection, however this is often not needed.

Read-modify-write Another option would be to read the memory location first, modify the specific byte or bytes as required, and then writing the full data back to the memory. In this case the a write to a subset of a word would result in a read of the complete word and then a write of the complete word. Which means that error correction can be applied just as it would have been to a memory without partial writing. In this case the actual SRAM does not even need to support partial writing as it is handled externally.

The drawback of this method is that every partial write operation has to be translated into a read and write operation. This can either be done by a small state-machine connected between the processor and the memory, which would stall requests from the processor while it is executing the read-modify-write operation. Another option would be to completely depend on the processor for this operation by restricting memory operations to complete words only, which would force the processor to do the same operation. Either way, this will increase the cost of partial memory writes, as a single write operation now has to be translated into a read and a write operation.

Split memory for check bits Another possible solution is described in [12] by Li and Huang, where the check bits are stored in a separate dual-port memory. Using a small state-machine and a few registers [12, Fig. 3] they describe an implementation that allows for full throughput of one operation per cycle, while still correctly updating the check bits. To achieve this, the SRAM needs to support write-through operation, that is outputting the written data as if it was a read operation. Fortunately, this is almost always supported in SRAMs.

In this solution the partial write operations are executed on the main memory as normal. After the write operation is executed, the check bit memory will be update on the next cycle using the updated written data. A small state-machine with six states [12, Fig. 5] arbitrates

these memory accesses, and will resolve any problems like reading the same location directly after writing by forwarding the corrected check bits, which were not yet written to memory.

There are two drawbacks to this method. First of all, the implementation of this method is quite expensive using a separate dual-port memory for the check bits. The extra SRAM required is much more expensive than simply adding bits to an already existing SRAM. Secondly, there is a chance that errors will go unnoticed, as the partial write operation does not verify the integrity of the data in the memories. This means that if there is an error in memory before a partial write is executed, the error will be considered correct after the partial write operation, as the check bits are updated to reflect the partially updated data from the main memory.

2.2.3 Banking strategies

Memory banking is a technique frequently used in combination with SRAM. Instead of creating a single monolithic SRAM with the required size, multiple smaller SRAMs, known as banks, are created with a small bit of extra logic to correctly select one of these banks based on the request address. This is done because the speed of an SRAM decreases with an increased size due to capacitance of the bit-lines and word-lines.

The simplest way of implementing an error correcting memory controller using a banked SRAM is to simply consider the complete banked SRAM as a single memory. This is the simplest to implement and will provide predictable performance independent of the specific memory access pattern. However, in a banked SRAM there is also the possibility of reducing the impact of automatic refreshing or read-modify-write cycles by duplicating some of the memory controller hardware. Take as an example a system with two error correcting memory controllers. As long as the memory banks are accessed in an alternating order, the latency of read-write cycles caused by error correction will be undetected, as each bank is only accessed every other cycle. While this increases the cost of the memory controller, it does improve the bandwidth of the memory, especially if errors are corrected regularly, or partial writes are used often.

2.2.4 Memory interleaving

While methods for detecting and correcting multiple bit errors exist, there is also another way of protecting against multi-bit errors, without using more complicated error correction codes. To achieve this resiliency against multi-bit errors, columns of a memory can be interleaved. This means that the bits of a single word are no longer in order in the physical memory cells, instead one or more other memory words are mixed in. If a multi-bit error were to occur in this type of memory it would not be detected as such, instead it would be seen as a single error in multiple words, allowing error correction using only a simple Hamming code. However, this method differs from all the other discussed methods in that it actually requires changes to the SRAM, instead of just the memory controller.

With interleaving the need for an error correction code with adjacent bit error correction is reduced, however it does not completely replace it. Instead, it is often beneficial to combine the properties of both interleaving and adjacent bit error correction to allow for error correction in a much larger range of adjacent bit errors.

2.3 Available error correcting memory solutions

While everything discussed in the previous sections can be used to build an error correcting memory controller, there also exist partial or complete error correcting memory controllers from different vendors. This section lists the available options and their properties.

2.3.1 Xilinx

Xilinx is one of the largest FPGA vendors and in their Vivado Design Suite there is the option to generate an "AXI BRAM controller" [13]. This memory controller is used to connect the FPGA block RAM, which is just SRAM embedded in the FPGA, to an AXI bus. Interestingly, this memory controller does support optional memory error detection and correction.

This memory controller supports error correction only for specific data widths of 32, 64 or 128 bits. For these widths it can do error correction using a Hsiao code and for 32 or 64 bits it can do error correction using an Extended Hamming code. To support partial write operations this controller will use a read-modify-write approach. From [13, p. 61] it seems like the controller will always do a read-modify-write cycle for a write operation, even if the write operation overwrites the complete memory location. This would result in quite significant performance loss when writing if the error correction is turned on. However, this cannot be verified as the source of the AXI BRAM controller is not available. Furthermore, this controller will not take any action when the read data contains an error, instead it just signals the requester.

2.3.2 Altera/Intel

Altera, another large FPGA vendor, does not supply a complete solution for memory error correction. Instead, two basic building blocks are provided, an encoder block `ALTECC_ENCODER` and a decoder block `ALTECC_DECODER` [14]. These two blocks only support the encoding and decoding of an Extended Hamming code, but they do support arbitrary data widths up to 64 bits. This can be used to build an error correcting memory controller, however they are not specifically built for that purpose, instead they are generic building blocks for a system that needs SEC-DED error correction.

2.3.3 Lattice

Lattice, another smaller FPGA vendor, does seem to have some support for error correction on BRAMs, however the information provided is very sparse. In the memory usage guide for the ECP5 series of FPGAs [15] there is a mention of ECC support for BRAMs in the memory IP generator. The error correction code used is a SEC-DED code, however the specific code is not mentioned. This IP only supports data widths up to 64 bits and there is no support for partial writing when ECC is enabled. Overall this is very similar to the Altera approach of only providing a simple encoder and decoder, however here they are automatically connected to the BRAM.

2.3.4 OpenCores

OpenCores is a website where hardware designs can be shared and it has a section for error correction cores. In this section there are a few Hamming code cores [16], [17], a BCH code core [18], a Reed-Solomon code core [19] and others.

The Hamming code cores [16], [17] available are both very similar and provide a simple encoder and decoder block, just like the Altera blocks. These block could be used to build an error correcting memory controller, but need extra circuitry.

The cores that provide the more complex error correction codes [18], [19] are built in a sequential fashion. This means that they only process one or a few bits during each clock cycle. These cores might be suitable for communication applications, however they are not suitable for memory controllers, as they almost always require operations to complete in only a few cycles.

2.3.5 CAST Silicon IP cores

Finally, CAST Silicon sells an IP core for SRAM error correction [20]. This core supports SEC or SEC-DED operation using an unspecified error correction code for data widths up to 256 bits. However, the data width is limited to multiples of 8 bits. Furthermore, it can be configured to support partial writing using a read-modify-write scheme. The core will not automatically repair error in memory, but it does provide an additional bus port for the processor to access some status registers containing the number of error that have occurred and the location of the last error.

Chapter 3

Experimental setup

The goal of this research is to determine the impact of different error correction codes and memory controller designs on the effectiveness, performance, area, critical path and power. However, to be able to determine the impact of these different trade-offs, we would like to test these memory controllers in a real-world scenario. For this a test system was designed, which combines these memory controllers with a real-world system.

3.1 Test system using RISC-V core

The real world system used in this research approximates a microcontroller design, using a relatively simple processor core, with some memory connected directly to it. Furthermore, there is a small peripheral for outputting data to the simulator. Figure 3.1 shows a block diagram of the system design.

The processor core used in this system is a RISC-V core with a custom design, implementing the `RV32I_Zmmul` standard [21]. It is a classic five stage pipelined architecture, with support for handling interrupts and exceptions. Furthermore, it supports pipelined multiplication with a latency of two cycles. Altogether it is a simple but well equipped microcontroller processor. In straight line code the processors is able to retire one instruction every cycle, assuming the instruction fetch can produce an instruction every cycle. This depends on the memory architecture, but is possible in the used design.

The instruction and data bus coming out of the processor core are connected to two memories and a simulation peripheral. For these memory buses the TileLink TL-UL [22], [23] protocol is used. This is a relatively simple bus protocol that allows for full throughput on the memory buses. The Wishbone [24] bus was considered, however the classic specification cannot achieve full throughput, and the pipelined specification requires more complicated arbiters than TileLink. Both the instruction and data bus connect through a decoder and arbiter to both the instruction and data memory. These decoders and arbiters form a crossbar interconnect, this allows both the instruction and data bus to read from both memories and allows for full throughput if both are reading from a different memory. Furthermore, the data bus is also connected to a small peripheral used for communication in the simulation.

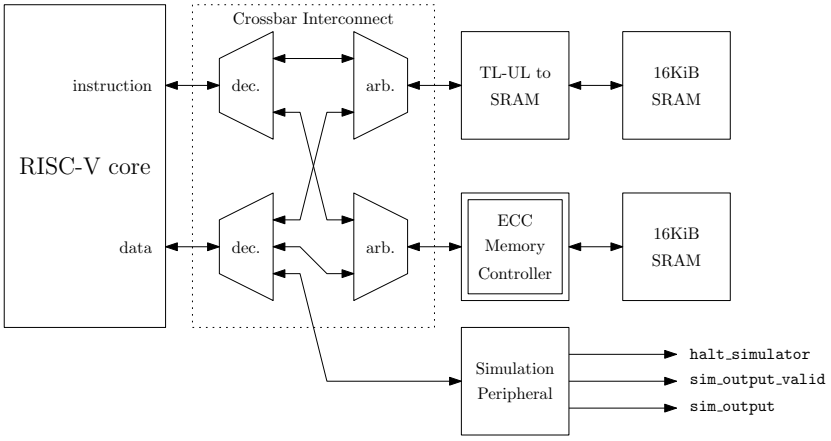


Figure 3.1: Block diagram of the test system with memory controller.

In this test system only the data memory is connected through an error correcting memory controller. The instruction memory is simply connected to the memory bus using a small adapter that converts TileLink operations to SRAM signals. During this research only error correction on the data memory is considered. First of all, only considering the data memory for errors simplifies the handling of the errors when detected. The code can assume that the exception handler code will always be valid and working. Second, in many real world cases the instruction memory is actually a static read-only memory. Either implemented as a mask ROM on the chip, or using a form of flash technology. These technologies are much less susceptible to errors and outside of the scope of this research. And finally, the results of this research can still be applied to the instruction memory, only more care has to be taken when handling uncorrectable errors in the instruction memory. This will likely require a full reset of the system when an uncorrectable error is detected during an exception handler.

The design of the error correcting memory controller consists of a number of submodules, as is shown in fig. 3.2. First of all, there is a TileLink interface, which handles the TileLink protocol specifics, and transforms the requests into a simple request-response interface. The request are passed to a partial-write wrapper, which transforms requests for byte or halfword writes to a pair of read and write operations. This transformation is required, as the TileLink protocol expects support for byte operations, while the error correcting memory does not support this. From there, the requests are processed by the memory controller and transformed into SRAM control signals. The write data going to the SRAM will pass through an error correction encoder, and the read data will pass through an error correction decoder. The specific implementation of these parts are discussed in the following section.

The simulation peripheral allows for easy interfacing with the simulation, which will be discussed in the next section. It has two functions, outputting data from the simulated system to the simulator, and notifying the simulator when the simulation is finished. The simulated program can write to the `sim_output` port, allowing it to transfer bytes of data to the simulator, which will print the output. When the simulated program writes to `halt_simulator`, the simulator will stop the simulation and exit with the exit code supplied by the simulated program. This allows us to exit the simulation when an exception occurs and output the exception code.

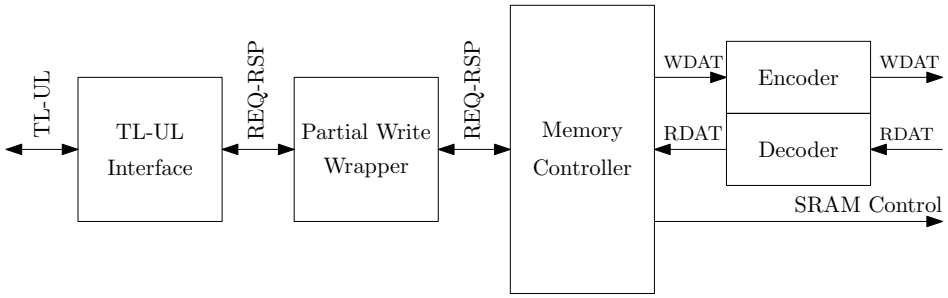


Figure 3.2: Block diagram of the ECC memory controller.

3.2 Generating error correction and memory controller hardware

To build the error correcting memory controllers as discussed in the previous section, actual hardware designs are required. The decision was made to build a custom piece of software to generate these hardware designs, because the designs that are available are not suited for this research. The problem with these designs is that all of them only support (extended) Hamming code or Hsiao code. Furthermore, both the Xilinx and CAST Silicon IP cores design are not available for experimentation, since they are locked to their vendor tool, and require payment respectively.

Before building this tool there were a number of requirements. First of all, this tool should support multiple error correction codes, including the Hamming and Hsiao codes, but not limited to just those two. Secondly, it should support multiple memory controller designs, with a number of the features discussed in the previous chapter. And finally, it should be built to be easily extended. Adding new error correction codes or memory controller designs should be simple, and parts that are reusable for multiple implementations should be factored out. All of these requirements should allow for generating the designs that are required for this research, but should also provide a framework that can be reused for other research in this field.

To build this tool I have chosen to use Python, in combination with the Amaranth [25] (formerly nMigen) hardware design language. This allows building the complete hardware designs in Python, with the ability to change many of the design parameters programmatically. In the end Amaranth can automatically generate Verilog code from the hardware designs described in Python.

3.2.1 Implementation of error correction codes

The implementation of error correction codes in the Python code are all based on a single abstract base class, the `GenericCode`. This base class implements all of the actual hardware generation for error correction codes based only on two matrices and two lists. Any implementation of this base class is required to create a generator and a parity-check matrix when the `generate_matrices` method is called. Furthermore, the implementation is required to

create a list of correctable and detectable errors, either in the constructor, or while generating the matrices. In turn, `GenericCode` provides a complete hardware implementation of the encoder and decoder based on these inputs.

`GenericCode` provides an encoder implementation, `GenericEncoder`, which can be used for any error correction code with a generator matrix. `GenericEncoder` will automatically generate the required xor-trees for implementing the encoder based on the generator matrix provided by `GenericCode`. This implementation should be optimal for most, if not all, error correction codes, which means that no code should have to implement its own encoder hardware, instead it can be inherited from `GenericCode`.

The other important part is the decoder, which is also provided by `GenericCode` as the `GenericDecoder`. This implements an error detecting and correcting decoder based on the matrices and lists of detectable and correctable errors. This decoder uses the generator matrix to extract the data bits and parity bits from the encoded input. It then calculates the error syndrome based on the encoded input and the parity-check matrix. This syndrome is used to determine which bits to flip in the data output to correct for errors. And finally, the syndrome and flips are used to determine whether an error occurred and if it was uncorrectable.

While the `GenericDecoder` implementation will always work correctly for any valid pair of generator and parity-check matrix, it might not always generate the most efficient hardware. This is because some error correction codes use special properties of the syndrome to detect uncorrectable errors faster. Take for example the extended Hamming code, which only needs to check a single bit in the syndrome for double error detection.

To simplify the efficient implementation of such error correction codes, the `GenericDecoder` uses two submodules internally, `GenericFlipCalculator` and `GenericErrorCalculator`. The flip calculator is responsible for calculating which bits to flip in the output data to correct for any errors based on the syndrome. The error calculator is responsible for determining whether an error has occurred and if it was uncorrectable. Any implementation based on `GenericCode` can override the `flip_calculator` and `error_calculator` methods to return a custom hardware implementation for these submodules.

Figure 3.3 shows the relationship between the `GenericCode` class, and the encoder and decoder classes.

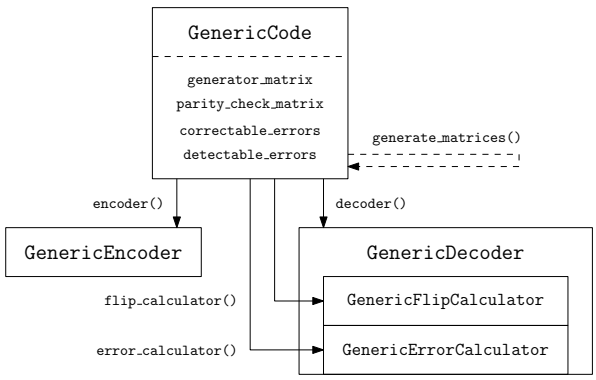


Figure 3.3: Relationship between `GenericCode`, `GenericEncoder` and `GenericDecoder`.

Automatic generation and optimization of the parity-check matrix

For most error correction codes the parity-check matrix is well defined and can be easily generated. However, there are also a number of error correction codes where the properties of the parity-check matrix are well defined, but there is no straight forward method for generating the matrix. They require using search algorithms to find a matrix with the required properties, or in some cases no algorithm is given at all.

To simplify the implementation of such error correction codes, the `BoolectorCode` base class can be used. This class uses the Boolector [26] SAT-solver for finding and optimizing parity-check matrices with the required conditions. For Boolector to find the correct matrices, the code implementation will have to define all properties of the parity-check matrix. However, no implementation of a specific search algorithm is required, skipping most of the error prone implementation work.

Furthermore, Boolector can be used to optimize the generated matrices by defining some optimization goals and then gradually lowering the required value. By default two optimization goals are provided, minimizing the maximum number of ones per row, and minimizing the total number of ones. These optimizations reduce the depth and size of the xor-trees in the decoder, respectively.

Implemented error correction codes

The following eight error correction codes have been implemented in this tool.

- **IdentityCode**, this implementation does not actually do any form of error detection or correction. Instead, the input data is encoded directly without any additional bits, and when decoded there is no check for errors, as there is no way to detect them. This implementation is meant for comparing all other implementations, while keeping as many of the variable constant.
- **ParityCode**, this implementation allows for single bit error detection. While it is not able to correct any errors, it is interesting to compare and might be useful in cases where crashing on an error is a valid option.
- **HammingCode** [6], this implementation provides the simplest single bit error correction code. The parity-check matrix generated by this code simply consists of columns with increasing binary values.
- **ExtendedHammingCode** [6], this implementation provides the error correction code used in many applications. It allows for single error correction and double error detection. The parity-check matrix is generated similarly to the Hamming code, but with an additional row of ones for the parity-check bit.
- **HsiaoCode** [7], similar to the extended Hamming code this implementation allows for single error correction and double error detection. However, this implementation

generates a different parity-check matrix. This matrix is found by using an exhaustive search of all possible valid matrices and selecting one with the lowest maximum number of ones per row.

- **HsiaoConstructedCode** [8], this implementation provides the same Hsiao code, however the method of generating the parity-check matrix is different. The parity-check matrix is generated directly without any searching using the algorithm described in [8].
- **DuttaToubaCode** [9], this code uses `BoolectorCode` to automatically search for a parity-check matrix. This code allows for single error and double adjacent error correction, and some form of double error detection. However, only between 40% and 50% of double errors is detected, all other double errors result in a mis-correction. The automatic optimization used by this code will attempt to reduce the number of mis-corrections, however they cannot be eliminated.
- **SheLiCode** [10], this implementation provides single, double adjacent, double almost adjacent and triple adjacent error correction, however no other error detection. This is also implemented using `BoolectorCode`, since the original paper does not provide a method of generating the parity-check matrix.

Figure 3.4 shows the relationship between `GenericCode`, `BoolectorCode` and all implementations of actual error correction codes.

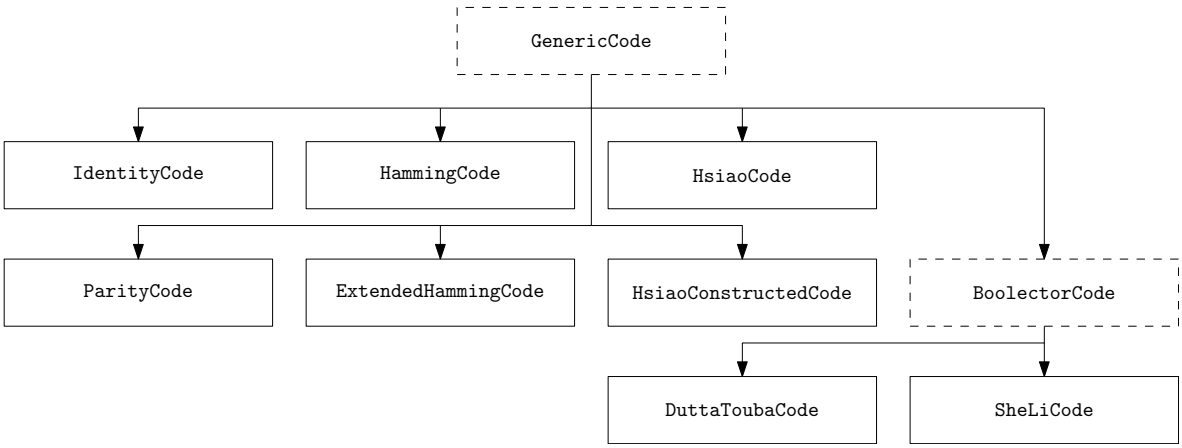


Figure 3.4: Class hierarchy of error correction codes.

3.2.2 Formal verification of error correction codes

During the implementation of these error correction codes, much care has been taken to ensure the correctness of each error correction code. With the use of `GenericCode` the opportunity for errors was already greatly reduced, as most of the hardware implementation only has to be checked in a single place. However, generating the parity-check matrix for an error correction code can also require some more complex algorithms, with their own opportunities for errors.

To verify that the generated parity-check matrices are actually correct, I decided to build a test-bench using formal verification. This test-bench verifies that every error that is specified in the codes correctable errors is actually correctable, and that every error that is specified as detectable is actually detectable. Furthermore, it makes sure that if there are no errors at all the output also reflects this.

This formal verification test-bench has two inputs, a data input and a flip input. The data input is connected to the encoder input. The encoder output is connected to a bank of xor-gates which use the flip input to invert certain bits in the encoded data. Finally, the flipped encoded data is connected into a decoder. The outputs of the decoder are used for a number of formal assertions.

If the flips input is zero, meaning introduce no errors, there are assertions checking that the data input of the encoder matches the data output of the decoder. Furthermore, the decoder should not raise the error or uncorrectable error outputs. If the flips input matches one of the correctable errors there are similar assertions to the no error case, however in this case we check that the error output is raised. Finally, in the case where the flips input matches a detectable error there is an assertion checking that both the error and uncorrectable error outputs are raised.

Using this approach the formal verification tools will check any possible combination of input data and flipped bits, and in the cases where the flipped bits match one of the three cases discussed, it will check all the assertions. In the case where the flipped bits do not match any of the three cases, the output of the error correction code is undefined, so these outputs cannot be checked. In the end, this approach will make sure that every error that is defined as correctable or detectable for a specific code, is actually correctable or detectable.

While this formal verification gives great confidence in the correctness of the error correction code implementations, it does not give complete certainty. This method depends on the correctness of the correctable and detectable errors lists. If either of these lists contain errors or omissions, the verification might succeed, but the operation of the error correction will not be correct. This is a real scenario, which happened during development, and solidifies the need for additional testing using simulation.

For the actual formal verification the SymbiYosys [27] tool was used, in combination with Boolector [26] as verification backend. All the implemented error correction codes were correctly verified in under a minute using these tools.

3.2.3 Implementation of memory controllers

Memory controllers are implemented in a similar way to the error correction codes. They also use an abstract base class called `GenericController`. However, this generic controller implementation only defines a common interface to the memory controller and keeps track of the supplied error correction code, but does not actually implement any hardware. Instead, any controller implementation has to provide their own custom controller implementation.

The memory controller interface consists of two parts, the user facing request and response channels, and the memory interface. The user request and response channels use a ready-valid handshake to allow for back pressure from both the memory controller on the request channel and the user on the response channel. The memory interface is a simple single-port SRAM interface, which consists of a clock, clock enable, address, write enable, write data and read data.

In addition to the memory controllers there are also a number of wrappers. These wrappers are not memory controllers themselves, but instead allow extending the features of a memory controller. A wrapper intercepts the user request and response channels and can add, remove or transform the messages on these channels.

Implemented memory controllers and wrappers

The following three memory controllers have been implemented in this tool.

- **BasicController**, this is the simplest version of a memory controller. It connects the user request to the memory interface through the error correction encoder, and the user response to the memory interface through the error correction decoder. Only a tiny bit of additional logic and a single flip-flop is required to handle the request ready and response valid signals.
- **WriteBackController**, this memory controller extends the basic controller with write-back functionality. When a correctable error is detected on a read operation, the next cycle will be used to automatically correct this error in the memory.
- **RefreshController**, this implementation is not actually a completely new memory controller. Instead, it is a combination of `WriteBackController` with a `RefreshWrapper`. Since this is a common and useful combination, this implementation provides a convenient alias.

The following two memory controller wrappers have been implemented in this tool.

- **RefreshWrapper**, this wrapper will automatically issue a read request every n cycles and the address of this read request is incremented automatically. In combination with the `WriteBackController` this will automatically fix any correctable errors in the memory.

  By default, this wrapper will only use unused cycles to attempt the refresh operations. This allows the processor to always access the memory when it would like to, even when a refresh should happen. As a result, it is possible that the processor uses all of the available memory bandwidth, which would never allow for refresh operations. If the refresh operation is required, the refresh wrapper can be configured to forcefully execute the refresh operation, blocking the processor for the cycle that it is running the refresh.

Uncorrectable errors that are detected by the refresh wrapper are ignored and not signalled to the user of the refresh wrapper. This is because there is no in-band way of signalling these errors, instead requiring an out-of-band method, such as an interrupt. Furthermore, the refresh wrapper might be refreshing unused memory where errors are not important and can be safely ignored.

- **PartialWriteWrapper**, this wrapper will transform a write request with a byte mask into a combination of a full read operation and a full write operation. Since none of the error correction codes support partial writing directly, this wrapper can be used to enable this functionality. All read operations and write operations to the full width of the memory will be executed normally, but partial write operations will result in a read-modify-write operation, to make sure the parity bits are also updated.

3.2.4 Verification of memory controller implementations

For the verification of the memory controller implementations simulation is used. The access patterns of these simulations can be divided into two categories, synthetic and real-world. Using these simulations the correct operation of these memory controllers cannot be guaranteed, however the simulations give great confidence in the correct operation.

The simulations using synthetic access patterns consist of sequential memory access and random memory access. The sequential access simulation is mostly used to check the basic operation of the memory controller, and to make sure that they do not break under constant full load.

The random access simulations are testing completely random inputs to the memory controller. This exercises the both the read and write operation of the memory controller, while verifying the correct operation. Furthermore, it also exercises the flow control on the user interface of the memory controller by randomly lowering the ready signal. These simulations make sure that all requests result in a response, and that the response actually matches the request.

Finally, the real-world access patterns are tested using a simulation of a real-world system running some benchmarking software. The test system, simulation method and benchmarking software will be discussed in the following sections.

Another option for the verification of these memory controller is using formal verification, like was used for verifying the error correction codes. Unfortunately, formal verification is more complicated for sequential designs. Due to my personal inexperience with formal verification this method was not attempted for this research. However, these memory controller designs are a good candidate for formal verification as they are relatively small and well contained.

3.3 Simulation implementation

Two of the goals of this research are to determine the error correction effectiveness and the overhead of the error correcting memory controllers. While a physical experiment is theoretically possible, it would require the production of a large number of chips with different designs and the use of a radiation source to induce errors in these chips. Getting chips produced within the timespan of this research is practically impossible, and furthermore, very expensive. Therefore, computer simulation is a great alternative, which allows for unlimited experimentation with different designs and does not require dangerous or expensive equipment.

For this research the simulation contains the complete test system as described earlier. This enables the comparison of different memory controllers under real world conditions running actual software.

The implementation of the simulation uses the CXXRTL [28] framework which is provided by Yosys [29], [30]. CXXRTL translates a hardware design read by Yosys into a C++ class, which can be compiled into a simulator program. This class allows for controlling the inputs and reading the outputs from the hardware. The hardware design used here only uses the clock as an input, so the simulator only has to toggle the clock signal. After raising and lowering the clock, the simulator can read the outputs from the test system and determine if there is any output or if it needs to halt.

In the simplest case this is all that is required to build a simulator, however in this case we would also like to intercept all the interactions with the data memory. While it is theoretically possible to interact with every wire in the simulated system, it would be needlessly complex and fragile in cases where the design changes slightly. Instead, the black-box functionality of CXXRTL is used to completely replace the data memory. In the test system design the data memory is replace by a black-box module called `sim_dmem`. This module has the same interface as a normal memory, with additional feedback signals from the `error` and `uncorrectable_error` outputs of the decoder. The simulator implements the abstract class provided by CXXRTL for this black-box module, which allows the CXXRTL code to use the provided implementation for the simulation.

The black-box implementation of the data memory provides three functions. First of all, it handles the memory read and write operations which are passed through the input wires using an internal array for storage. Second, it will randomly flip bits in the memory storage array to simulate the memory experiencing an error. And finally, it will keep track of the number of clean, corrected, uncorrectable and undetected errors in read operations for every memory location.

To determine when to flip bits a random number generator that produces integers according to a poisson distribution is used. Such a random number is generated each cycle and determines the number of bits to flip in that cycle. For these simulations the λ parameter of this distribution is often very small, so most of the time no bit is flipped. If a bit has to be flipped in this cycle, the memory location and bit index will be determined using a uniform random

number generator. Furthermore, this simulator also supports generating adjacent errors with a specified probability.

Real-world software

While this simulator is able to run any program on the simulated processor, a program that approximates real-world use of the processor would be ideal. For this the CoreMark [31] benchmark software was chosen, as it runs a number of real-world algorithms, specifically list processing, matrix manipulation, state machine and CRC algorithms. Furthermore, it is relatively easy to get running on a microcontroller by implementing just a few methods for timing operations. This benchmark exercises the complete test system as it contains both CPU intensive and memory intensive processing. Together this should result in a good approximate of real-world use of the processor and memory.

This software is built with a small piece of setup assembly, which is responsible for setting up the environment before calling C code. This setup assembly also configures the processor with an exception handler, which will be called when an uncorrectable memory error is detected. The exception handler will instruct the simulator to halt the simulation with a specific exit code equal to the exception code. In a real application this exception handler would attempt to fix the problem, or in the worst case issue a reset to make sure the operation of the program is correct.

3.4 ASIC layout of designs

The other three goals of this research are to determine the area, critical path and power efficiency of these memory controllers. To determine these properties for the different memory controller designs, a physical layout is needed. To create this physical layout an actual ASIC technology is required, which are often hard to get access to, as they are often under non-disclosure agreements. Fortunately, in the end of 2020, Google partnered with SkyWater foundries and Efabless [32] to provide an open-source ASIC technology. The Skywater 130nm technology [33] can be used together with other open-source layout tools to produce complete hardware layouts.

The OpenLane [34], [35] toolchain combines multiple open-source project, containing OpenROAD [36], [37], Yosys [29], [30], Magic [38] and others, to provide a complete flow from hardware description language to ASIC layout. This tool is used to synthesize, place and route both just the memory controller designs and the complete test system designs containing the memory controller. To get the best performance results, the toolchain is configured to optimize for delay instead of area, producing slightly larger designs, but with much shorter critical paths.

While OpenLane is able to do the layout for standard cell designs, building memory blocks requires a specialized tool. For this an open-source memory compiler called OpenRAM [39],

[40] is used. This tool is able to create memory blocks with custom data widths, number of rows and number of columns.

Using these tools both full system designs and simple memory controller only designs can be synthesized and routed. The layout of the full system design will contain four OpenRAM SRAM blocks, two 8KiB blocks for the instruction memory and two 8KiB blocks for the data memory. Between these SRAM blocks standard cells will be used to implement the processor. For the memory controller designs only standard cells are used to implement the designs.

Chapter 4

Results

In this chapter the results of the simulation and ASIC layout of the error correcting memory controller designs are discussed. First, the error correction effectivity and the performance impact are discussed. Second, the critical path, area and power usage of the ASIC layouts of the designs in isolation are discussed. Finally, these same metrics are also discussed for the ASIC layouts of the complete test system.

4.1 Effectivity and performance results

4.1.1 Simulation setup

Measuring the error correction effectivity and performance of the designs is done using the simulation techniques discussed in the previous chapter. These simulations are running the complete test system with a 32 KiB instruction memory and a 4 KiB data memory, that is, 1024 words of 32 usable bits, the actual memory size will be larger if the error correction bits are counted. The test system in these simulations is running a CoreMark [31] 2k performance run with 10 iterations. Because this configuration of CoreMark only touches 677 memory locations, or only 588 if you ignore locations that are only written, the data memory size was reduced to 1024 words, to show the refresh controller operating in a realistic scenario. Furthermore, the refresh controller is configured with a 128 cycle refresh period, which means one memory location is refreshed every 128 cycles, and the complete memory every $131\,072$ cycles.

For each combination of the eight error correction codes and three memory controller designs a number of simulations are run. Each configuration is run with 0, 1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140 and 150 errors per million cycles. Furthermore, these configurations are also run with both no adjacent errors and 10% adjacent errors. Finally, all of these configurations are run 50 times with different seeds for the random number generator to get a representative random sample of outcomes, except for the case with no errors, as randomness will never influence that result. This means a total of $(17 \cdot 2 \cdot 50) + 1 = 1701$ simulations are run per design configuration, or in total $3 \cdot 8 \cdot 1701 = 40\,824$ simulations for all designs.

There are a number of ways these simulations can exit and produce their final results. First of all, the simulation exits either by instruction of the simulated program, through the simulation peripheral, or when the simulator kills the simulation after it has executed more than the allowed number of cycles.

Since the simulation is known to take exactly $5\,052\,828$ cycles when running without errors, the maximum number of cycles is set at $5\,100\,000$. This maximum allows any valid run to complete in time, while exiting simulations that get stuck in a reasonable time. Simulations that exit in this way are considered *terminated*.

If the simulation exits by informing the simulator through the simulation peripheral, the simulator will receive an exit code from the simulated program. This exit code can either indicate a successful execution, or it can indicate failure, in which case it will supply an exception code to indicate the type of failure. The simulated program will indicate success if it completes running the program, or failure if an exception occurs.

If an exception is indicated, the exception code is checked for the type of exception. If the exception is a load access fault, the memory controller has indicated an uncorrectable error. In this case the simulation is considered *aborted*. Any other type of exception should not occur normally, however if errors completely mess up the flow of the simulated program they might occur. These cases are also considered *terminated*.

If the simulation indicates a successful completion of the program, it might not actually have succeeded in correctly running the program. Therefore, first the output of the simulated program is checked, if this output does not match the expected output the simulation is considered *incorrect*. If both the exit code indicates success and the program output is correct, the simulation is considered *correct*. However, if the simulator detected any mis-corrections in a otherwise correct simulation run, the simulation is considered *cheated*. This is because the simulation output is correct, but there was an undetected error, which in this case did not cause any problems, but could have in other cases, therefore it cannot simply be considered correct.

All simulation which are considered *correct* or *aborted*, have a positive result, as they either complete correctly, or at least detected the an uncorrectable error and exited. All simulation which are considered *incorrect*, *terminated* or *cheated*, have a negative result, as they either incorrectly assumed correct execution, were unable to complete at all, or could have easily resulting in a failure.

4.1.2 Effectivity results

All of the simulation results for the error correcting memory controller designs are shown in fig. 4.1 with no adjacent errors, and fig. 4.2 with adjacent errors. These figures show the distribution of the five possible exit conditions over the range of 0 to 150 errors per million clock cycles.

The first row of fig. 4.1 shows the results for the `IdentityCode`, so no error correction at all.

This row clearly shows that error correction is actually required in an environment where errors can occur. At only 1 error per million clock cycles the percentage of correct executions already drops to only 12% for the `BasicController` and `WriteBackController`, or only 8% for the `RefreshController`, while at 5 errors per million clock cycles there are no correct executions left. All the other executions are either incorrect, terminated or in a few cases cheated.

The second row of fig. 4.1 shows the results for the `ParityCode`. While the drop-off of correct executions exactly matches the `IdentityCode`, the correct executions make way for aborted executions. This is much better than with the `IdentityCode` as the errors are now detected and can be handled. While the `ParityCode` does not have allow for error correction, it does increase the chance of detecting an error to practically 100%.

The third row of fig. 4.1 shows the results for the `HammingCode`, which are much more interesting than the identity and parity codes. These graphs show the drop-off of correct simulation with the increased error rate. The steepness of this drop-off is determined by the memory controller that is used, where the basic controller performs the worst, the write-back controller performs better, and the refresh controller the best. While the basic controller drops to 40% correct simulations at only 20 errors per million clock cycles, the write-back controller manages 46% correct simulations at 50 errors per million clock cycles. Furthermore the refresh controller actually manages to reach 100 errors per million clock cycles before hitting 40% correct simulations.

Furthermore, these graphs also clearly show the lack of decent error detection in the Hamming code. There are large numbers of incorrect, terminated or cheated simulations, because two bit errors have a high chance of being mis-corrected instead of detected. However it does detect some errors, which is due to the default implementation of uncorrectable error detection, that will report an uncorrectable error if the syndrome generated does not match any known syndrome.

The 4th, 5th and 6th row of fig. 4.1 show the results for the `ExtendedHammingCode`, `HsiaoCode` and `HsiaoConstructedCode` respectively. The graphs for these codes are actually identical, as these codes have the exact same error correction capabilities, and the random simulations are seeded deterministically. The correction performance of these codes is basically equal to the Hamming code, which is as expected, as they are all single error correcting codes. However, instead of resulting in failures when the number of errors increase, these codes correctly abort the simulation, notifying the processor of uncorrectable errors. Interestingly, there is a single simulation at 70 errors per million clock cycles, where the result is cheated, which is caused by three errors in a single location, which is read very infrequently. Since errors occur randomly, this is always a possibility, however the chance of this happing is very low. Therefore, these error correction codes are basically perfect in this scenario.

Finally, the 7th and 8th row of fig. 4.1 show the results for the `DuttaToubaCode` and the `SheLiCode`. While these codes have much more extensive error correction capabilities, the performance of these codes is not much better than the previous four error correction codes. This is because the random single bit error injection that the code is subjected to will almost never result in adjacent errors. However, these codes do see a marginal increase in correctly

exiting simulations. Unfortunately, both of these codes do not completely detect random 2-bit errors, which means that there is a decent chance of the simulation ending in either an incorrect, terminated or cheated state.

Random 2-bit adjacent errors

While the simulations shown in fig. 4.1 were only simulated with single bit error injection, the simulator also allows for adjacent bit error injection. Since these types of errors can realistically occur in memories, simulations were also run with an adjacent error probability of 10%. This means that every error that is injected, has a 10% chance of becoming a 2-bit adjacent error.

Figure 4.2 shows the results of all the error correction codes and memory controller designs running under this 10% adjacent error rate. This figure shows some clear trends when adjacent errors are added. First of all, the `ParityCode` has a chance of missing errors, as it is only able to detect single bit errors. This shows up as a few percent of the simulations resulting in termination instead of aborting cleanly. Furthermore, it shows that the Hamming and Hsiao codes are not able to correct errors with any significant error rate. The extended Hamming and Hsiao codes are still able to detect the errors.

More interestingly, the results of the `DuttaToubaCode` and `SheLiCode` are basically unaltered by the added adjacent errors. They perform similarly to the case with only single bit errors, as they are capable of correcting the 2-bit adjacent errors.

Relating simulated error rates to the real world

All of the results discussed are considered in relation to a number of errors per million clock cycles. This unit was chosen as the simulation does not actually consider the clock speed of the design, instead it only considers the number of clock cycles. However, to compare the error rates, a rate in errors per second is much more useful.

As will be seen in following section, the test system with the different error correcting memory controllers will be able to run at frequencies approximately between 75MHz and 95MHz. The conversion from errors per million clock cycles to errors per second would be multiplying by the clock frequency divided by one million. This results in an error rate from zero errors per second to between $11\,250$ and $14\,250$ errors per second.

Looking at research, such as [41], [42], the error rates simulated here are much higher than any real world scenario. The real world error rates are often in the range of one error per one hundred seconds to one thousand seconds, so many orders of magnitude smaller. However, this is to be expected, as the simulated program here only runs for approximately 5 million cycles, which means the error rate has to be much higher to have an effect on this program.

In the real world, all of these error correcting memory controllers would be operating at the left edge of the graphs very close to zero errors per million clock cycles. Simulating these

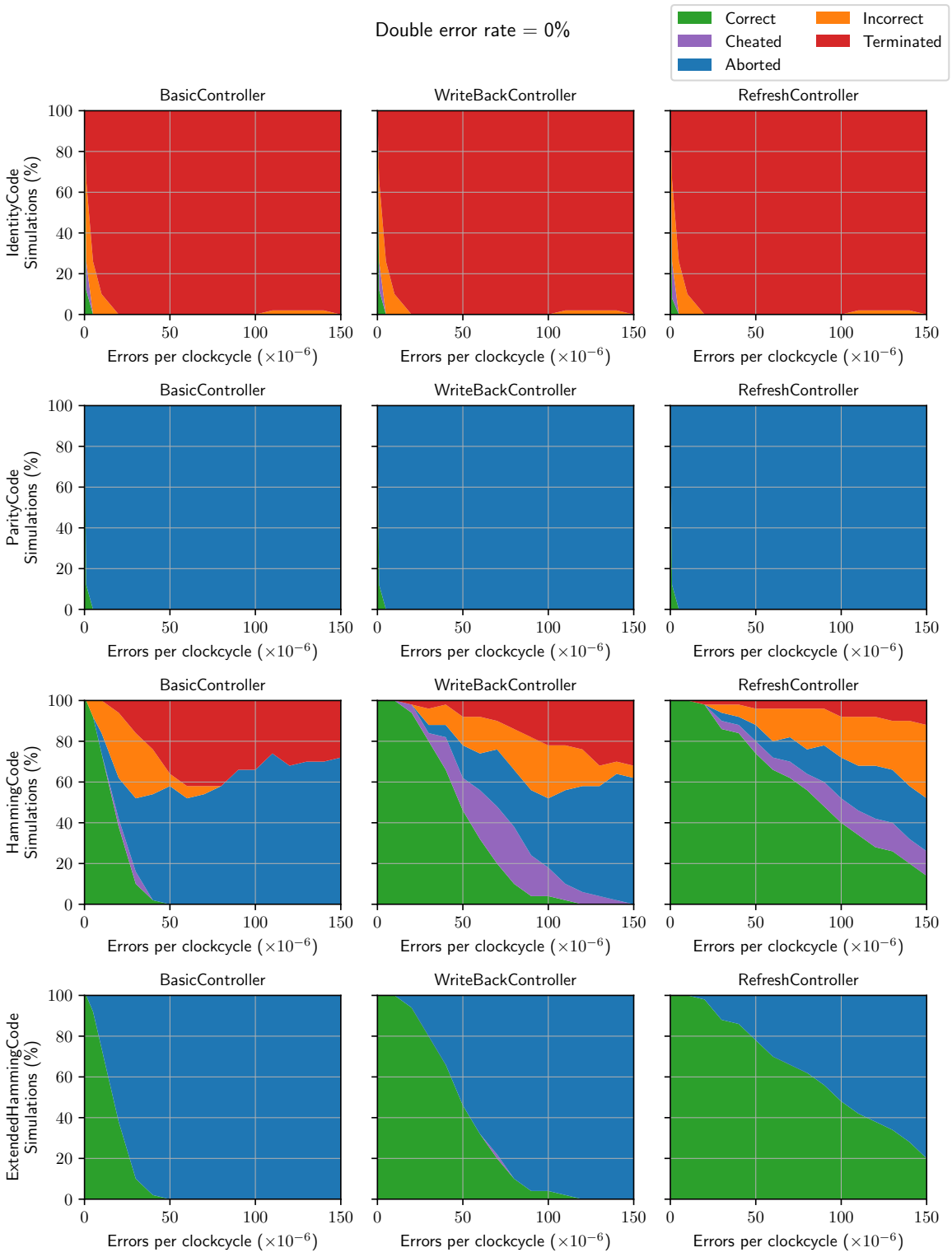


Figure 4.1: Simulation results of all error correcting memory controller designs.

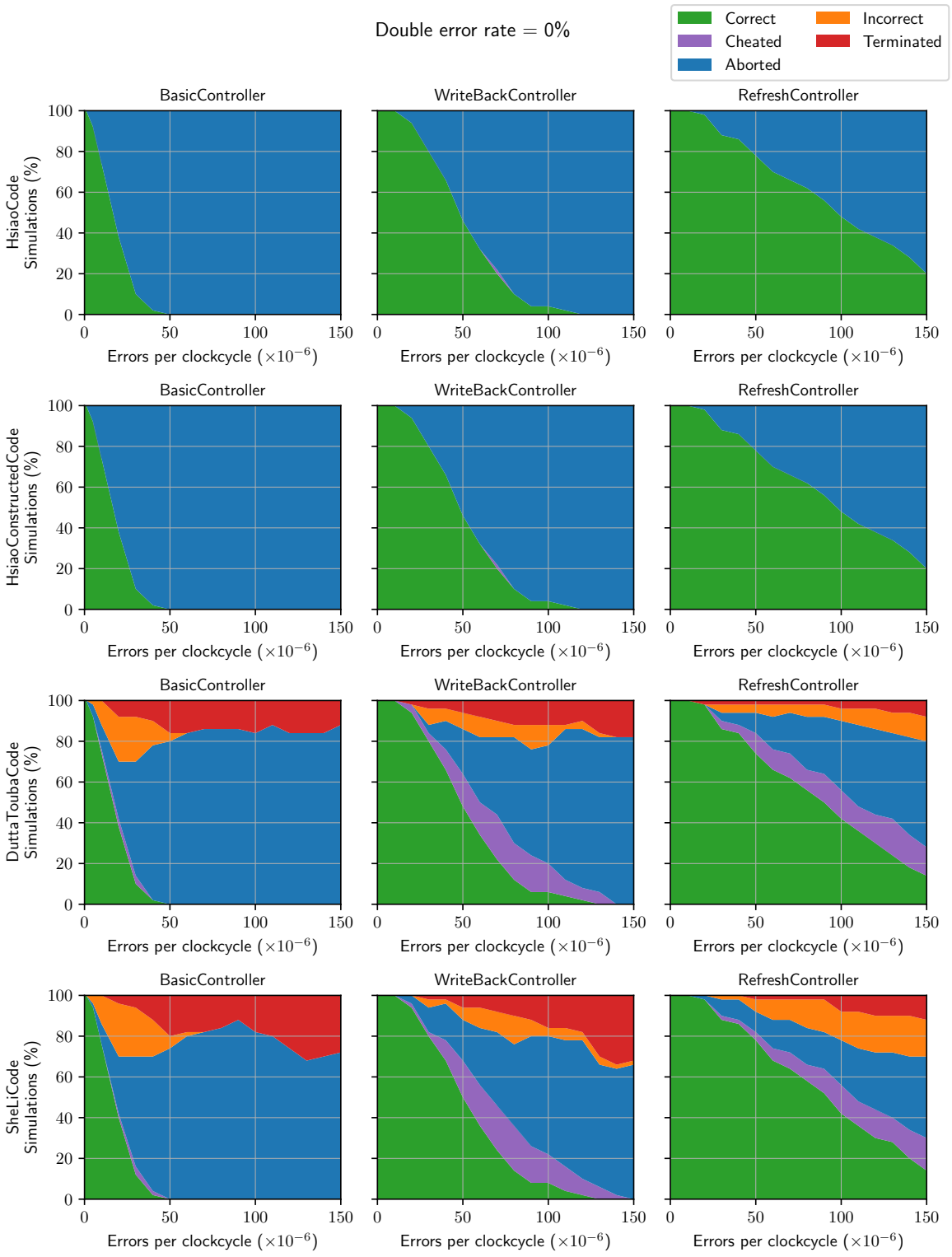


Figure 4.1: Simulation results of all error correcting memory controller designs. (Continued)

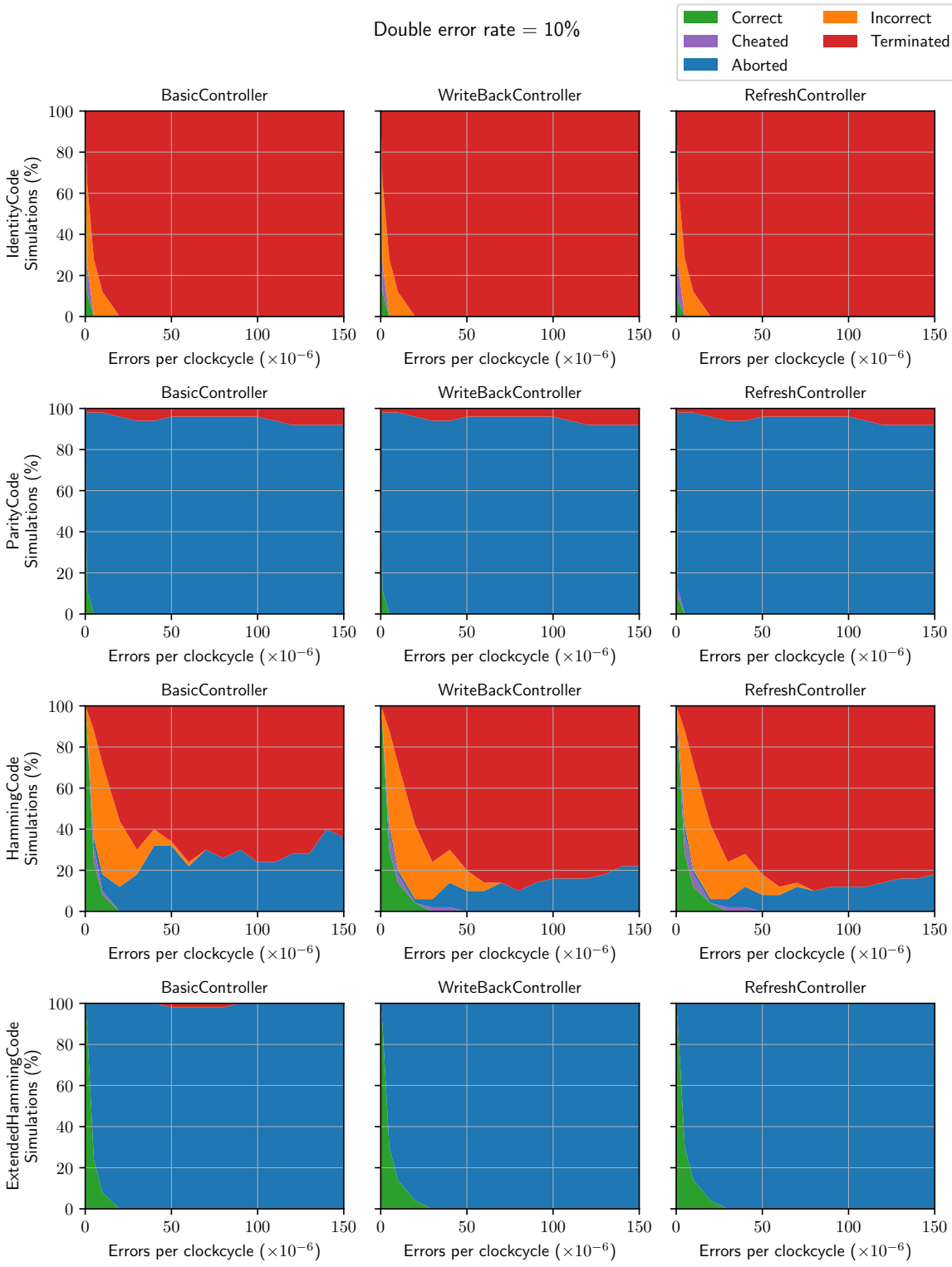


Figure 4.2: Simulation results of all error correcting memory controller designs, with 10% adjacent errors.

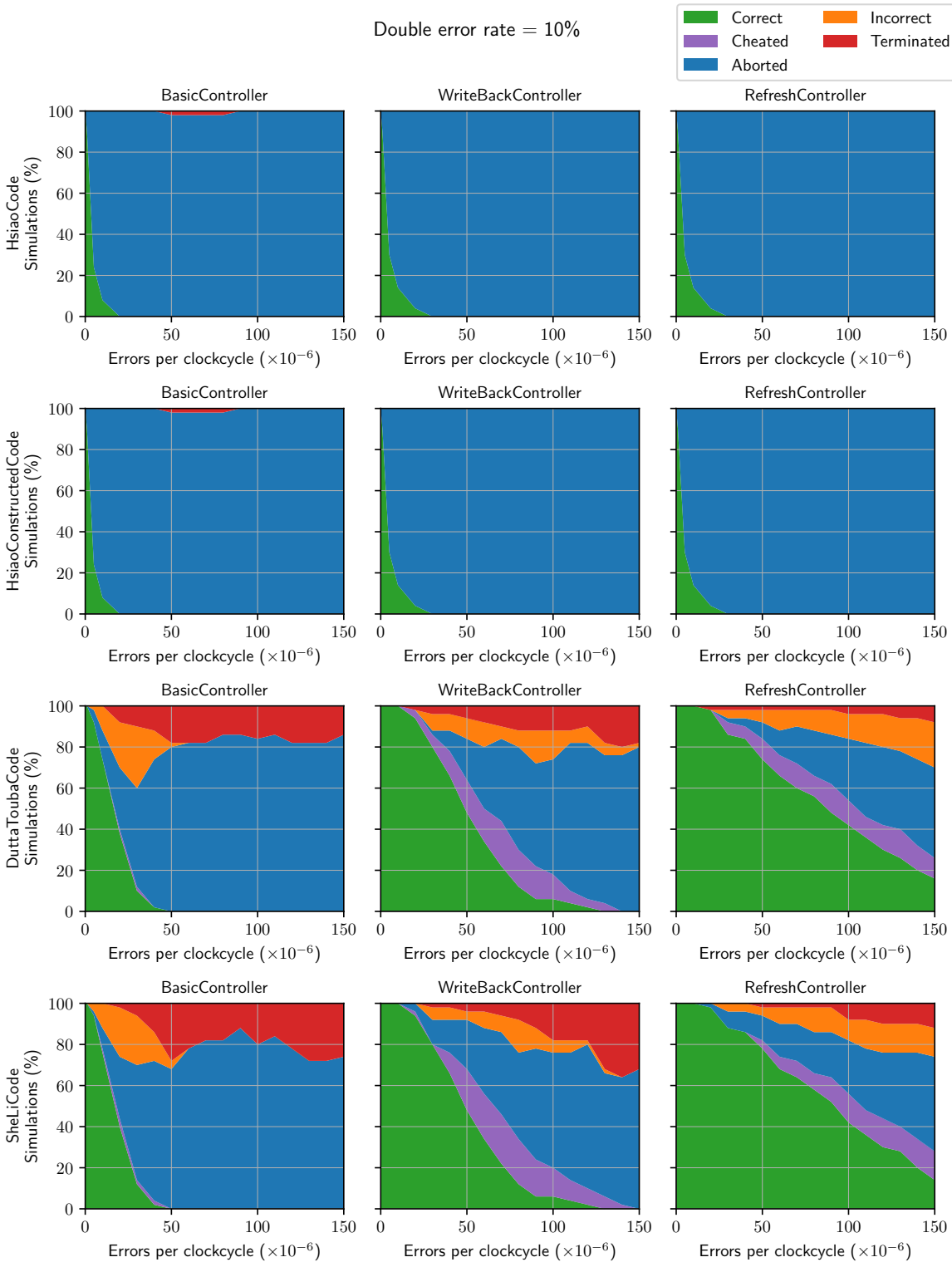


Figure 4.2: Simulation results of all error correcting memory controller designs, with 10% adjacent errors. (Continued)

higher error rates allows us to compare the error correction codes better than when they are all running at basically 100% correctness.

4.1.3 Performance overhead results

In addition to measuring the error correction effectivity, the overhead of these memory controllers can also be measured. The overhead is measured in additional cycles over the base case, where there are no errors at all. Since the number of cycles in the zero error case is fixed at $5\,052\,828$ cycles, the difference can be easily calculated.

Figure 4.3 shows the overhead of each of the different controllers, with lines for each of the error correction codes. Only correct executions are included in this overhead graph, as the number of execution cycles can vary drastically for non-correct executions. Each line shows the average number of additional cycles at a specific error rate.

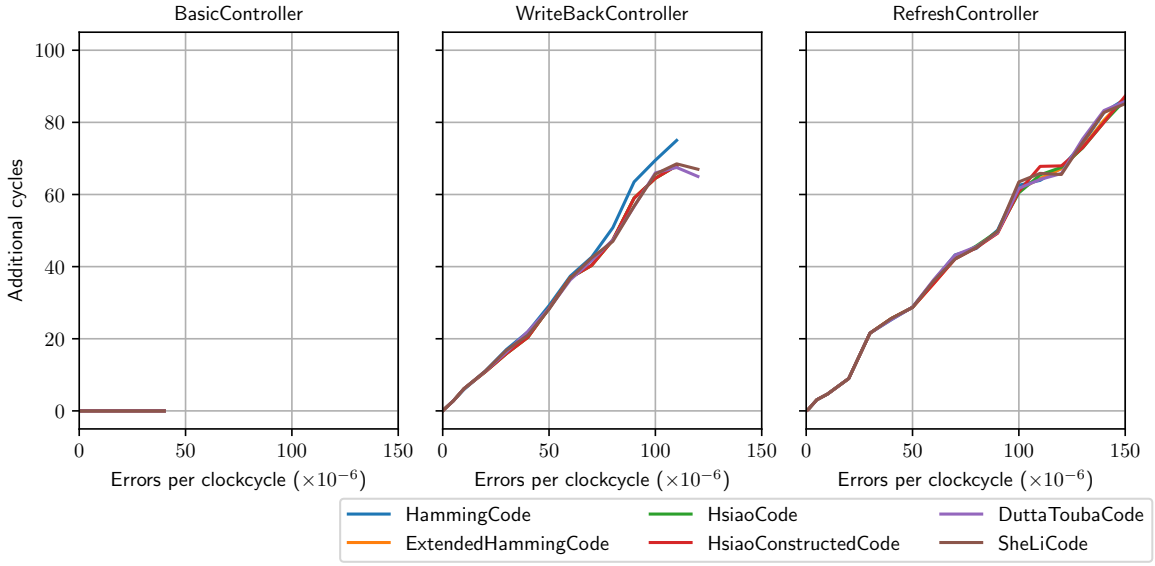
Both fig. 4.3a and fig. 4.3b show very similar results for the no adjacent errors and 10% adjacent errors case. As expected, the `BasicController` does not introduce any additional clock cycles. The `WriteBackController` and the `RefreshController` do both introduce additional clock cycles. These controllers should introduce a single additional cycle for every error that is read, resulting in a linear increase of additional cycles for a linear increase of errors.

However, based on the number of errors per million clock cycles, one might expect a higher number of additional cycles. At 50 errors per million clock cycles, these designs would experience approximately 250 errors during the simulation run. Therefore, one might expect approximately 250 additional cycles, instead the measured number is closer to 30 additional cycles. This is because of two factors, first of all, some errors are simply never detected, as they are affect unused memory. Second, the additional cycles are only incurred if the memory read detecting the error is directly followed by another memory operation. For this specific simulated program the overhead is approximately 0.12 additional cycles per injected error, however this number can vary significantly depending on the type of program that is running. If the processor never accesses the memory in two consecutive clock cycles, the overhead of the memory controllers is reduced to zero, as the additional cycle always overlaps with an unused cycle. However, if the processor accesses the memory in large continuous bursts, it might experience the an additional cycle for every error that is detected.

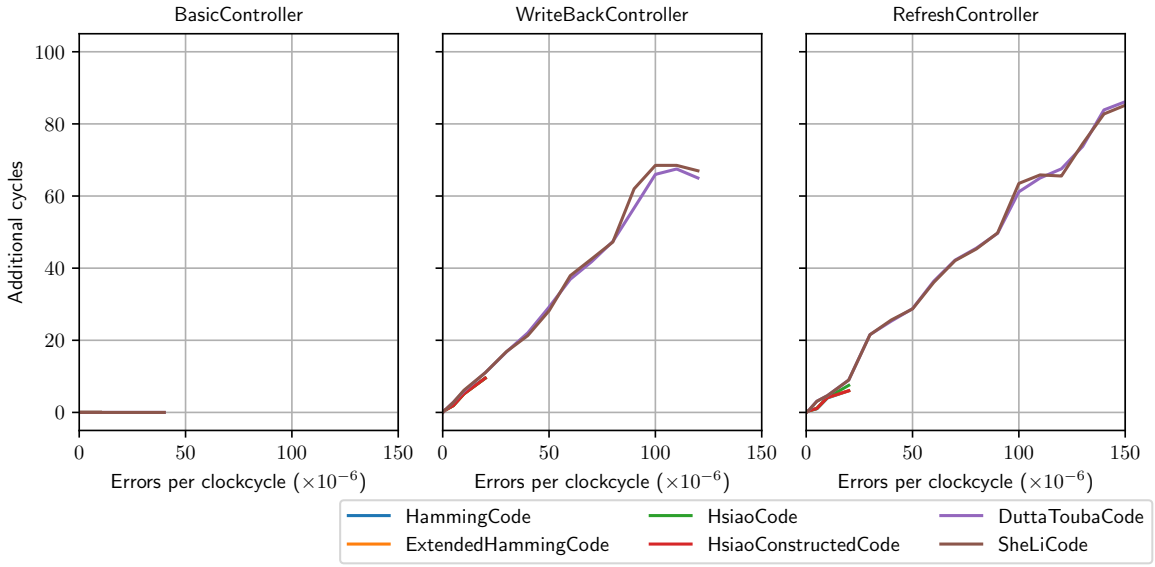
4.2 ASIC layout results for error correcting memory controllers

While the simulations can be used to determine the effectivity and clock cycle overhead of the memory controller designs, it cannot determine the area, critical path delay or power usage. To gather these results an ASIC layout of each of the designs has to be created. All of these layouts are built using the OpenLane [35] flow with the Skywater 130nm PDK [33].

Listing 4.1 shows the configuration parameters used for the OpenLane flow to generate the following ASIC layouts. This configuration enables extra optimizations in the synthesis to



(a) Only single errors



(b) With 10% adjacent 2-bit errors

Figure 4.3: Clock cycle overhead of different memory controller designs.

improve the critical path delay. Furthermore, it will set the placement density to a reasonable 0.45, which is also used in the full system layout. It will reduce the global routing resources by 5%, forcing the global router to put a little more effort in the routing, but reducing the effort for the detailed router, as there is now some additional headroom. And finally, for these layouts the input and output delay is not considered. This is done to show the best-case performance of these designs, and because these designs will likely be integrated in a full system design, which has a lot of opportunity to reduce any input and output to delay to basically zero.

```
# Optimize synthesis for delay
set ::env(SYNTH_STRATEGY) "DELAY 0"
set ::env(SYNTH_SIZING) "1"
# Set the core size and placement density
set ::env(FP_CORE_UTIL) 30
set ::env(PL_TARGET_DENSITY) 0.45
# Decrease the available resources in global routing
set ::env(GLB_RT_ADJUSTMENT) 0.05
# Do not count I/O delay
set ::env(IO_PCT) 0
```

Listing 4.1: OpenLane configuration used for ASIC layout.

4.2.1 Critical path delay

Figure 4.4 shows the critical path delay for each of the different error correction codes combined with each of the memory controllers. Furthermore, these designs all also include a partial write wrapper, as this is commonly included in every error correcting memory controller design. The combination of `IdentityCode` and `BasicController` shows the baseline critical path delay of an error correcting memory controller, consisting mostly of the few gates and flip-flops required for the basic request-response interactions and partial write support. This baseline critical path delay is approximately 2.85ns. The `WriteBackController` has a very similar baseline critical path delay, however, the `RefreshController` has a baseline critical path delay of approximately 3.71ns. This shows that using the `RefreshController` can have an impact on the speed at which the memory controller can operate.

The `HammingCode` and `ExtendedHammingCode` show significantly higher critical path delays. With the `HammingCode` delay between 5.57ns and 6.06ns, and the `ExtendedHammingCode` delay between 5.85ns and 6.33ns. For the `ExtendedHammingCode` with `WriteBackController`, the delay is significantly larger than the other two controllers, however this is likely just the variance in the ASIC layout tools. One might expect the `HammingCode` to be faster than the `WriteBackController` by a greater margin, as the `HammingCode` has a smaller and shallower xor-tree. However, because this implementation of the Hamming code does detect some uncorrectable errors, its advantage is reduced by the extra logic to detect uncorrectable errors.

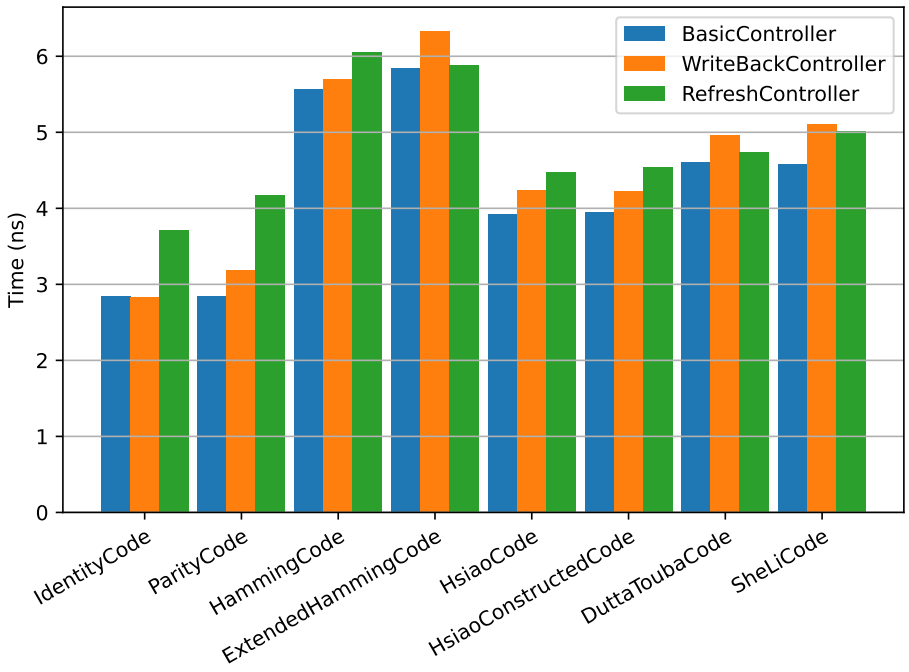


Figure 4.4: Critical path delay for memory controller and error correction tests.

For the `ExtendedHammingCode` detecting uncorrectable errors is simply checking a single bit in the syndrome.

The critical path delays of the `HsiaoCode`, `HsiaoConstructedCode`, `DuttaToubaCode` and `SheLiCode` are all somewhere between the baseline and the Hamming codes. For the two variations of Hsiao code there is almost no difference in the critical path delay, which shows that the actual method of generating the parity-check matrix for these codes does not significantly impact the generated hardware performance. Both the `DuttaToubaCode` and `SheLiCode` have a slightly higher delay than the Hsiao codes, which can be attributed to the increased number of syndromes that can flip a bit, and the slower method of detecting uncorrectable errors.

4.2.2 Cell and chip area

While the critical path delay of the memory controller is very important for performance, the chip area used by the memory controller determines the cost of implementation. This chip area can be determined using two different methods. First of all, the area of the standard cells in the design can be measured, which ignores any routing between the actual cells. This shows small variances between designs more clearly, but lacks any information regarding the routing area. Second, the area of the complete layout can be measured, which includes every part of the design, however is less accurate, since it is determined up front by estimation.

Figure 4.5 shows the standard cell area of the different memory controller designs. In this figure, a clear trend is visible, where the error correction codes with increased complexity use

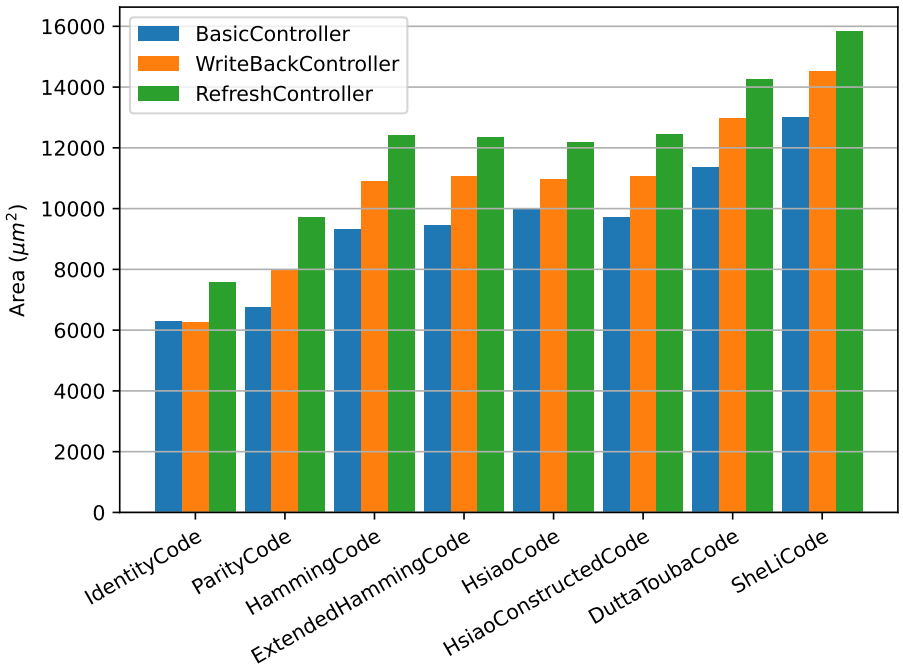


Figure 4.5: Standard cell area for memory controller and error correction tests.

a larger area. Furthermore, the Hamming and Hsiao codes use basically the same area, while performing quite differently with regards to the critical path delay. This figure also shows that the more complex controllers do indeed use more area to implement all their features.

Figure 4.6 shows the chip area used for the complete designs. This figures shows the same trends as fig. 4.5, however the absolute size of the results is multiplied with a relatively constant factor of about $4\frac{1}{2}$.

4.2.3 Power usage

The OpenLane flow is also able to create power usage estimates for the ASIC layouts that is has generated. These power estimates are shown in fig. 4.7, however they vary wildly between different controllers and error correction codes. The figure shows some trend of increased power usage with larger designs, which is expected, as the designs has more cells and wires to drive. Unfortunately, these results are basically unusable, and only give a very rough estimate of 10 milliwatts for these designs.

These problems are likely caused by the immature power estimation in OpenSTA, the analysis tool used by OpenLane for static timing analysis and power estimation. Furthermore, this tool does not support power estimation with simulated activity, so all activity numbers are estimated.

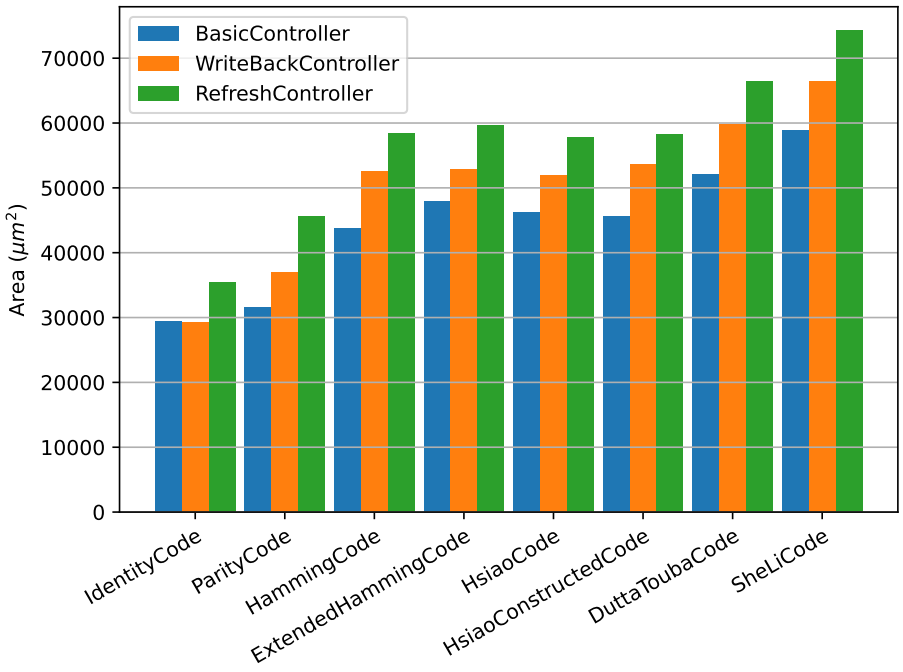


Figure 4.6: Chip area for memory controller and error correction tests.

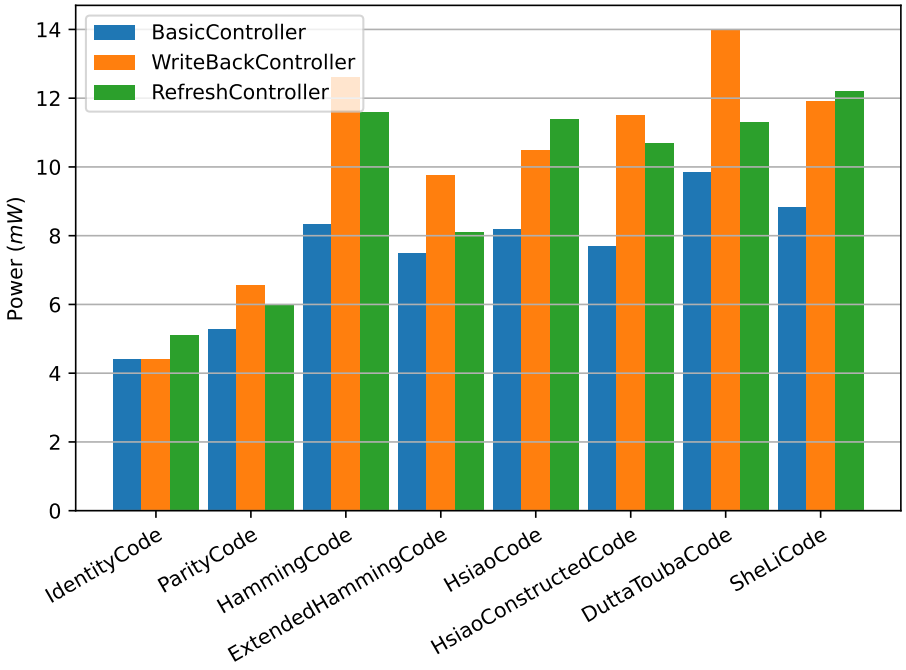


Figure 4.7: Power usage estimates for memory controller and error correction tests.

4.3 ASIC layout results for the test system

Next to the ASIC layouts of just the memory controllers, the ASIC layouts of the complete system can also be created. These can provide the area, critical path delay and power usage for a real world scenario, where the memory controller is implemented in a complete design.

These designed are laid out in a rectangular die, with the SRAM memories in the four corners of the design. On the left side of the chip, there are two 8KiB OpenRAM memories for the instructions, and on the right side there are two 8KiB OpenRAM memories for the data.

Listing 4.2 shows the important configuration parameters used for the OpenLane flow when creating these ASIC layouts. The layout of these designs uses a fixed die area of $1750\mu m$ by $1550\mu m$ for the error correction codes with 0 or 1 parity bits, and $1900\mu m$ by $1550\mu m$ for the designs with 6, 7, or 8 parity bits. Furthermore, the placer is instructed to focus more on routability when placing the cells, as the designs might be quite congested around the memory. Finally, the input-output delay of the design is set to approximately 0.6ns, which matches the output delay of the OpenRAM SRAM blocks.

```
# Optimize synthesis for delay
set ::env(SYNTH_STRATEGY) "DELAY 0"
set ::env(SYNTH_SIZING) "1"
# Set a fixed size for the complete layout
set ::env(FP_SIZING) absolute
set ::env(DIE_AREA) "0 0 1750 1550"
#                 or "0 0 1900 1550" for 38/39/40 bit data memory
# Set the placement density and optimize for routability
set ::env(PL_TARGET_DENSITY) "0.45"
set ::env(PL_ROUTABILITY_DRIVEN) 1
# Decrease the available resources in global routing
set ::env(GLB_RT_ADJUSTMENT) 0.05
# Set I/O delay to 0.6ns, which is the approximate delay of the SRAM
set ::env(IO_PCT) 0.06
```

Listing 4.2: OpenLane configuration used for ASIC layout.

4.3.1 Critical path delay

Figure 4.8 shows the critical path delay for the complete test system containing each possible combination of error correction code and memory controller. This figure shows a baseline critical path delay somewhere between 10.5 and 11.0ns. The baseline critical path is the combination of the critical path from the processor design combined with the common memory controller hardware, which consists of the partial write wrapper and the basic controller im-

plementation. The figure shows no clear trend in critical path delay for the different memory controller implementation, suggesting that most of the controller hardware gets optimized better in the complete design, or is outside of the processor critical path and therefore does not contribute to the total critical path delay. This differs from the previous results where the `RefreshController` on its own had a significantly longer critical path delay than the `BasicController` or `WriteBackController`.

The overall trend for the error correction codes better matches the results of the controller only layouts. The critical path delay of the `ExtendedHammingCode` is still significantly higher than that of the other error correction codes, by approximately 2ns. On the other hand, the critical path delay of the `HammingCode` is much closer to the other codes in the full system layout, while in the controller only layout it was basically equal to the `ExtendedHammingCode`.

For the `HsiaoCode` and `HsiaoConstructedCode` are unsurprising as they both have a slightly higher critical path delay than the `IdentityCode`. Similarly, for the `DuttaToubaCode` and the `SheLiCode`, the critical path delay is again slightly higher than the critical path delay of the Hsiao codes.

While the critical path delays of the ASIC layout for the memory controller only were relatively stable, these results are much more varied. This can be attributed to variance in the ASIC layouts, which increases with larger designs. This variance exists because OpenLane uses tools which have randomized components or components with heuristics, which might change the result quite significantly for only small input changes. To get the best results, many ASIC layouts of the same designs would have to be made. Unfortunately, each of these ASIC layouts can take multiple hours, which means running multiple iterations of the same design was infeasible.

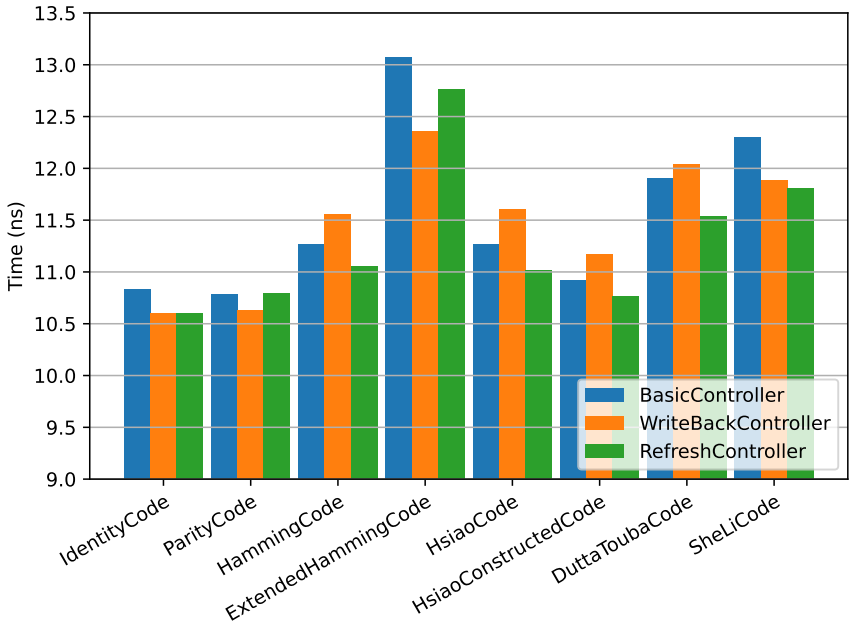


Figure 4.8: Critical path delay for the full test system.

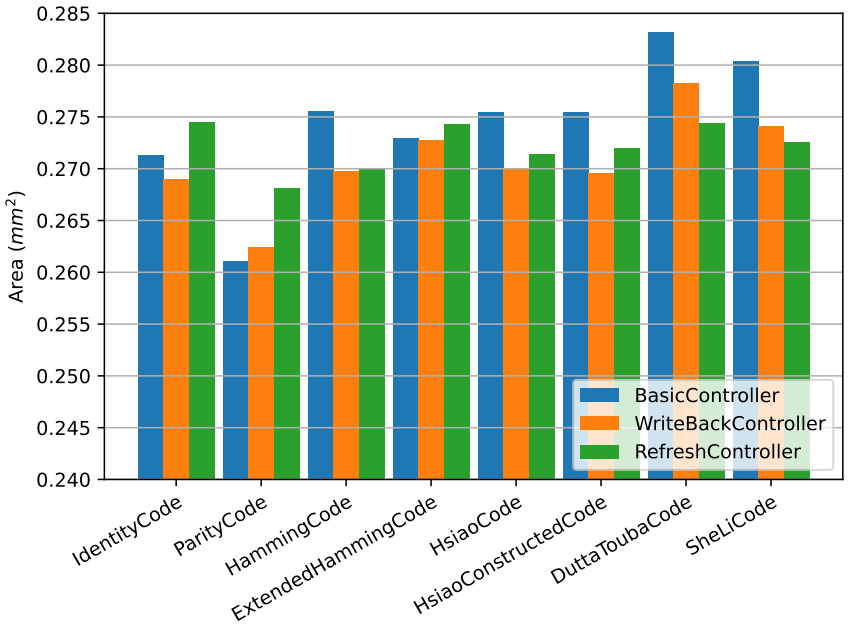


Figure 4.9: Standard cell area for the full test system.

4.3.2 Cell and chip area

The area of the ASIC layout of this complete test system can be used to determine the overhead of the of these different memory controller on the overall design. Because the complete test system consists of both the processor logic and the OpenRAM SRAM blocks, determining the area is slightly more complicated.

First of all, the area of the standard cells implementing the processor design can be measured. This only measures the area of the cells, not any routing, but does include basically everything in the design except for the memories. Figure 4.9 shows the area of the standard cells in the test system with different error correction codes and memory controller. This figure shows that the area fluctuates between 0.26 and 0.28 mm^2 for the different designs.

The cell areas of just the memory controllers, as discussed in the previous section, were only between 0.006 and 0.016 mm^2. This means that the memory controller designs only take up approximately 2.5% to 5.7% of the total design. The `ParityCode` implementation is approximately 0.010 mm^2 smaller the `IdentityCode` implementation, while their controller designs were roughly equal size. This means that size of the memory controller is likely to disappear in the run to run variance of the ASIC layout tools.

The total chip area of these designs is not really interesting, as it was fixed in the configuration. This was required, as the placement of the memory blocks is not automatic. All designs using the `IdentityCode` or `ParityCode` are laid out in a 1750 $\mu\mathrm{m}$ by 1550 $\mu\mathrm{m}$ rectangle, while all other designs use a 1900 $\mu\mathrm{m}$ by 1550 $\mu\mathrm{m}$ rectangle. This results in an area of 2.7125 mm^2 and 2.945 mm^2 respectively.

Approximately 1.8 to 2.0 mm^2 of this chip area is used by the four memory blocks. The specific area for a design can be calculated from the memory block sizes in table 4.2. The remaining 0.9 mm^2 of the chip is used to for all of the standard cells. However, approximating from the cell areas and placement density for these designs, actually only between 0.58 and 0.62 mm^2 is used by the standard cells. This means that approximately 67% of the chip area is used for the memory blocks, and approximately 21% for the logic. The remaining 12% is unused.

Memory overhead

The overhead in memory size for the more complex error correction codes can be theoretically computed, simply using the number of data and parity bits. However, the actual SRAM blocks created by OpenRAM do not necessarily follow this exact trend. Table 4.1 shows the theoretical overhead for the different error correction codes, with the `ParityCode` only having a small 3.13% overhead, but the other codes approaching 25% overhead.

Table 4.2 shows the actual dimensions of 2048 word OpenRAM memories with different word sizes. This table shows that the real overhead is slightly smaller than the estimated overhead from table 4.1. This difference is due to the border area around the actual bit array containing the supporting logic, whose size stays relatively constant. Figure 4.10 shows the actual ASIC layout for a 32-bit and 40-bit OpenRAM SRAM block.

Code	Data bits	Parity bits	Total bits	Overhead (%)
`IdentityCode`	32	0	32	0.00
`ParityCode`	32	1	33	3.13
`HammingCode`	32	6	38	18.75

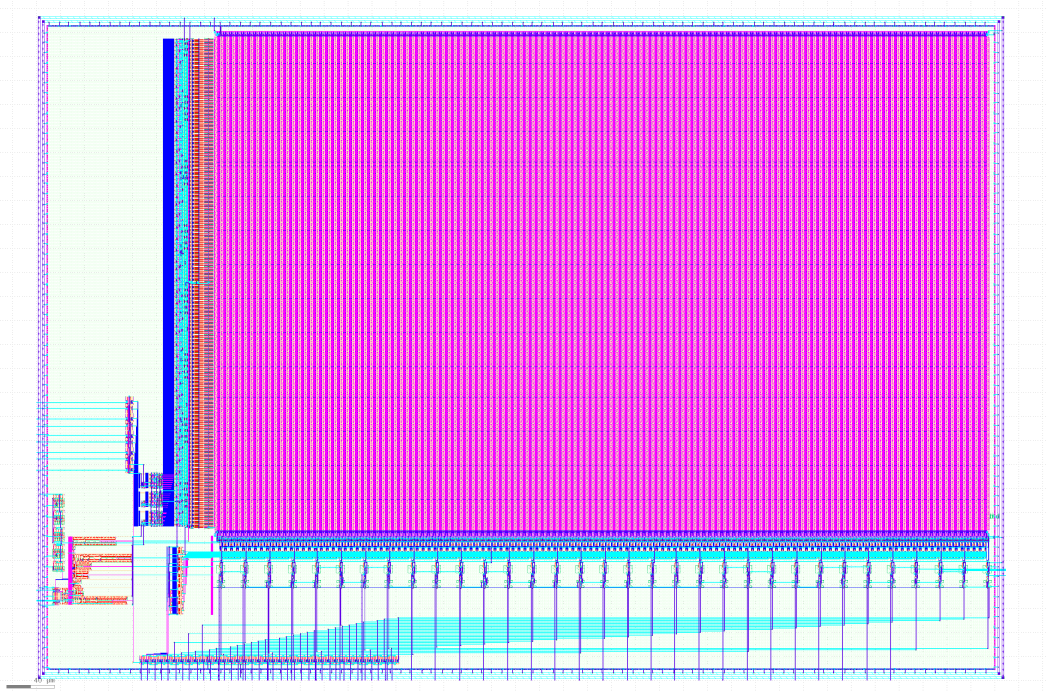
`ExtendedHammingCode`
`HsiaoCode`
`HsiaoConstructedCode`
`DuttaToubaCode`
	32	7	39	21.88

`SheLiCode`	32	8	40	25.00

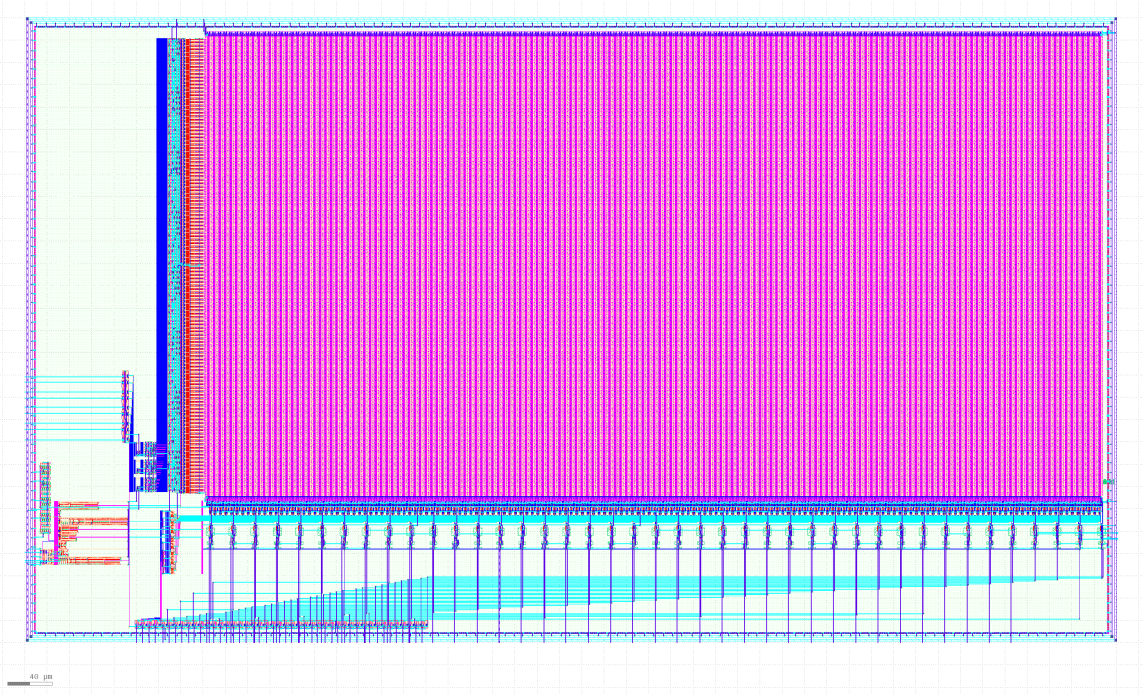
Table 4.1: Theoretical memory size overhead for all error correction codes.

Word size	Width (μm)	Height (μm)	Area (mm^2)	Overhead (%)
32	808.90	553.22	0.44750	0.00
33	829.30	553.90	0.45935	2.65
38	939.46	559.34	0.52548	17.43
39	959.86	559.34	0.53689	19.98
40	981.62	560.02	0.54973	22.84

Table 4.2: OpenRAM memory size and overhead for 2048 word memories with different word sizes.



(a) 32-bit



(b) 40-bit

Figure 4.10: ASIC layout of OpenRAM SRAM blocks.

4.3.3 Power usage

As with the power usage results for just the memory controllers, the power usage results for the complete system are also not very useful. However, they are included here for completeness.

These power results only count the power usage by the standard cells implementing the processor. This power usage is somewhere between 50 and 70 mW for these designs. The power usage of the SRAM memories is not included in these figures, and is likely more than the power usage of the complete processor. However, the OpenRAM tools currently do not supply this information.

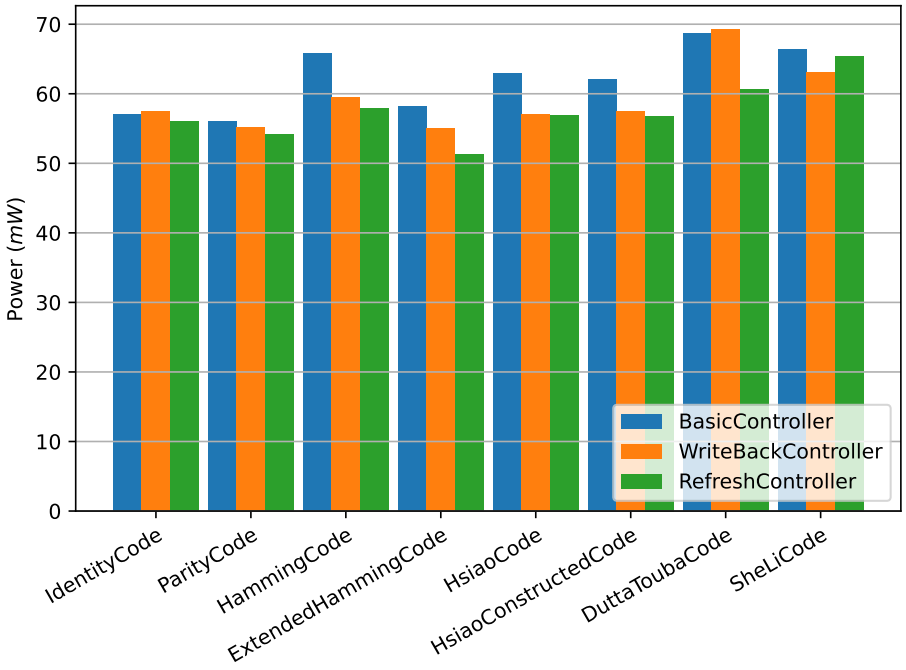


Figure 4.11: Power usage estimates for the full test system.

4.3.4 Final ASIC layouts

Figures 4.12 and 4.13 show the final layouts for `ParityCode` with `WriteBackController` and `HsiaoCode` with `RefreshController`. These figures show two specific layouts, all of the other layouts look very similar. The four corners show the large 2048 word memories, while the center contains the complete processor. The dark-blue areas at the edges of the plus-shaped middle section are mostly empty, while the brightly coloured center contains all the logic.

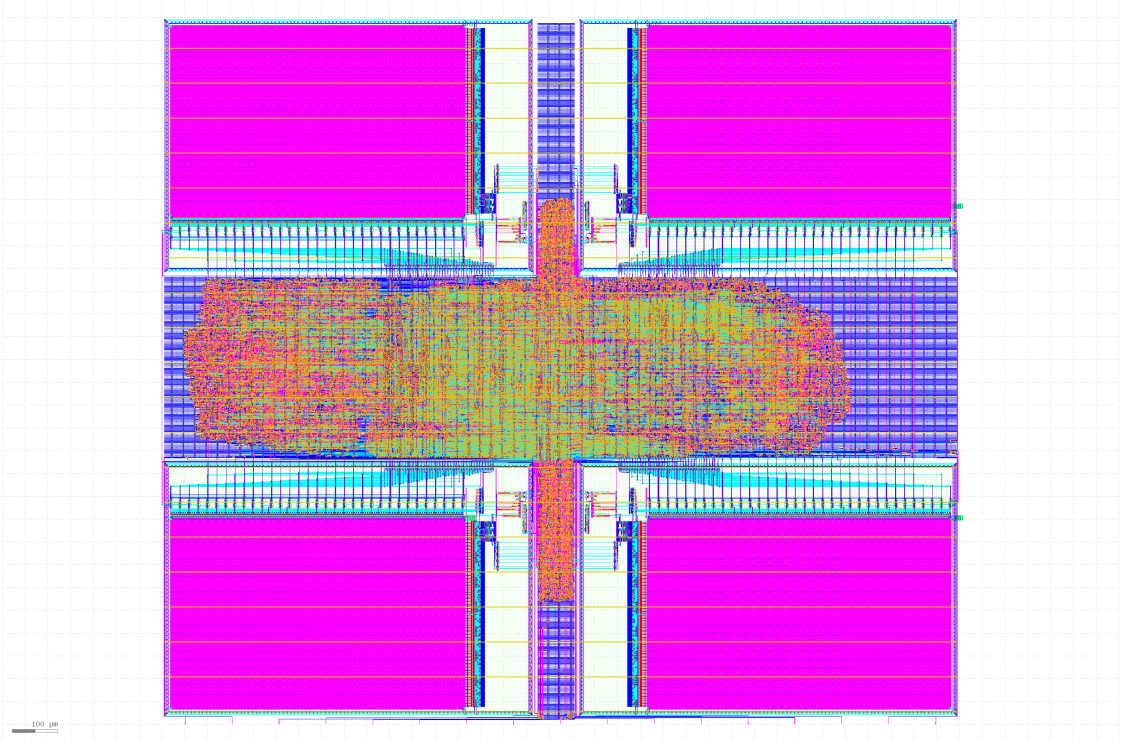


Figure 4.12: Final ASIC layout of the `ParityCode` with `WriteBackController`.

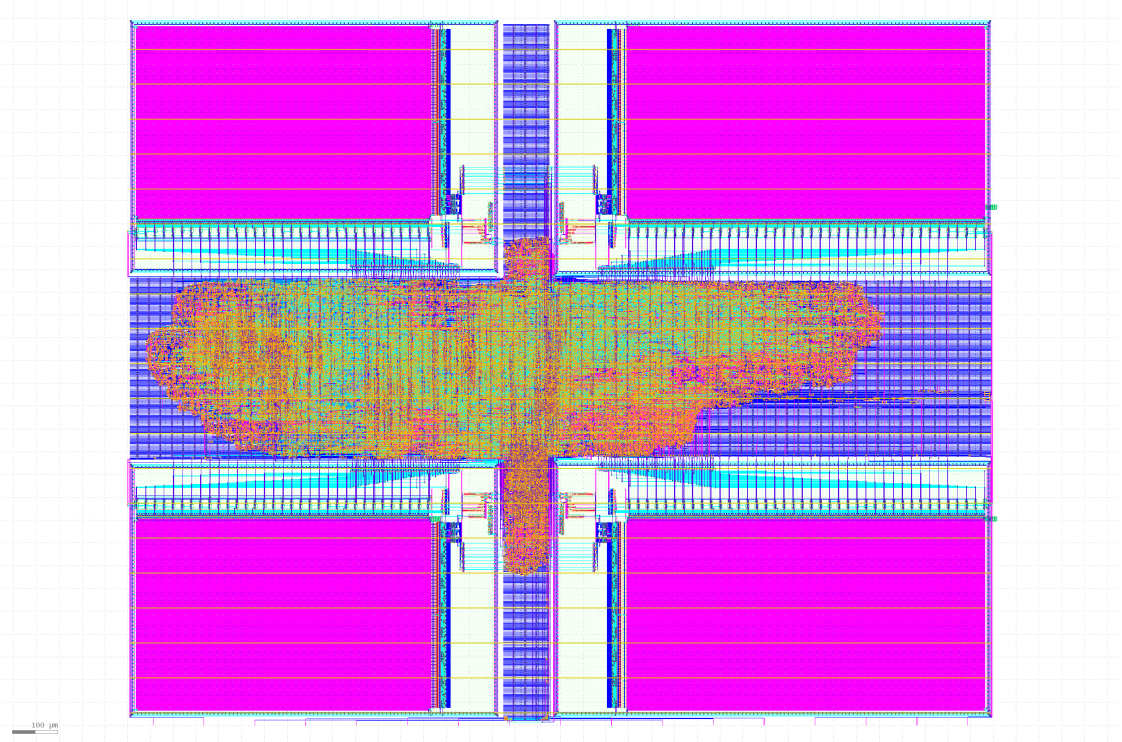


Figure 4.13: Final ASIC layout of the `HsiaoCode` with `RefreshController`.

4.4 Framework performance

In addition to looking at the performance of the generated memory controllers, the performance of the memory controller generator and of the error injecting simulator were both measured. The performance of these tools determines the usability for quickly iterating over multiple error correction code or memory controller designs.

4.4.1 Memory controller generator performance

The memory controller generator consists of two almost separate parts. First of all, the generation of the parity-check and generator matrices, and second, the generation of the actual hardware implementation of these memory controllers. The performance of both parts is measured individually, as in most cases the results of the matrix generation will be cached, incurring the matrix generation time only once, and using the cached matrices for the actual hardware generation. This also allows the user to first dedicate some time to generating highly optimized matrices, after which, these can be used quickly when generating hardware designs.

For all of the following timing results, the respective programs were run on a desktop computer containing an AMD Ryzen™ 9 5950X processor and 32GB of 3600MT/s DDR4 memory, with no other loads running at the time of the measurements.

Matrix generation

Table 4.3 shows the execution time of the matrix generation for all the different error correction codes with different data sizes. This table shows that matrix generation time for the `IdentityCode`, `ParityCode`, `HammingCode` and `ExtendedHammingCode` is fairly constant and negligible, as they are all less than one millisecond.

Interestingly, the `HsiaoCode` implementation shows an odd pattern in the execution times, where they fluctuate a bit around one millisecond for the 8, 16, 24 and 32-bit data sizes, and result in a massive 1.7 seconds for the 64-bit data size. This odd behaviour can be explained by the implementation of `HsiaoCode`, whose runtime is dependent on the number of possible parity-check matrices, which does not monotonically increase with the increase data size. For this reason the `HsiaoConstructedCode` implementation is also provided, which uses a construction method for building the parity-check matrix, instead of exhaustively checking all possible matrices. As table 4.3 shows, this implementation only takes milliseconds to complete in all tested cases.

Finally, the `DuttaToubaCode` and `SheLiCode` implementation use significantly more time to generate their parity-check matrices. Since these codes do not have an easy way to construct them, they are implemented using `BoolectorCode`, which will automatically search for possible parity-check matrices with the right properties. Unfortunately, this means that for the

Code	8-bit (ms)	16-bit (ms)	24-bit (ms)	32-bit (ms)	64-bit (ms)
`IdentityCode`	0.05	0.02	0.02	0.02	0.02
`ParityCode`	0.09	0.06	0.05	0.06	0.07
`HammingCode`	0.22	0.28	0.31	0.36	0.52
`ExtendedHammingCode`	0.26	0.68	0.39	0.51	0.81
`HsiaoCode`	0.12	1.31	0.09	4.61	1735.71
`HsiaoConstructedCode`	0.27	0.40	0.42	0.63	1.61
`DuttaToubaCode`	31.25	687.02	2149.88	11956.70	>5min
`SheLiCode`	2588.84	>5min	N/A	>5min	>5min

Table 4.3: Matrix generation time for different data sizes for all error correction codes.

more complicated codes, this search can quite some time. Finding a valid parity-check matrix for both codes only takes approximately one second, however optimizing the code to find the best parity-check matrix can take much longer. The execution times shown in table 4.3 consider optimizing for both the maximum number of ones per row, and the total number of ones.

An optimal parity-check matrix for the `DuttaToubaCode` can be found relatively quickly up to 32 data bits. However, minimizing the number of mis-corrections can take many hours and will likely not finish within any reasonable amount of time. Fortunately, the process of generating the matrices can be aborted at any point, and will return the best parity-check matrix so far.

For the `SheLiCode` generating the matrices is much harder, as the are many more conditions on the parity-check matrix. Furthermore, the lower bound on the optimization goals is not as tight as with the `DuttaToubaCode`, which means it might spend much more time optimizing, as it has to prove no better solution exists. For the designs with 16 data bits or more, the process of optimizing can take many hours. The design with 24 data bits is marked "N/A", as no valid parity-check matrix for this code with 24 data bits and 7 parity bits exists.

Hardware generation

The performance of generating the hardware is mostly dictated by the the Amaranth [25] framework, which actually generates the Verilog or Yosys ilang file. The execution time of running Python and generating a complete design is measured using Hyperfine [43].

Listing 4.3 shows the execution time for generating just the memory controller example. Generating the hardware design files is significantly faster when generating to the Yosys ilang format, as Amaranth uses an internal representation close to that of Yosys. However, if the designs has to be used with another tool that does not support Yosys ilang, Verilog has to be generated, which takes longer. Either way this only takes a few hundred milliseconds, which is still very fast, even when many designs are generated.

```
Benchmark 1: python -m memory_controller_generator.testbench.example generate test.v
    Time (mean ± s):     310.6 ms ±   8.8 ms    [User: 693.9 ms, System: 1416.2 ms]
    Range (min . max):   303.0 ms . 329.3 ms    10 runs

Benchmark 2: python -m memory_controller_generator.testbench.example generate test.il
    Time (mean ± s):     230.1 ms ±   7.1 ms    [User: 590.9 ms, System: 1338.4 ms]
    Range (min . max):   219.2 ms . 244.9 ms    13 runs
```

Listing 4.3: Execution time generating just the memory controller and error correction hardware.

```
Benchmark 1: python tilelink_soc.py generate test.v
    Time (mean ± s):      1.207 s ±  0.018 s    [User: 1.601 s, System: 1.484 s]
    Range (min . max):    1.185 s .  1.249 s    10 runs

Benchmark 2: python tilelink_soc.py generate test.il
    Time (mean ± s):     763.7 ms ±   2.7 ms    [User: 1175.9 ms, System: 1458.9 ms]
    Range (min . max):   761.3 ms . 769.5 ms    10 runs
```

Listing 4.4: Execution time generating the complete test system.

Listing 4.4 shows the execution time for generating the complete test system. Again, here the Yosys ilang format is significantly faster than generating Verilog. In total generating the Verilog for the complete test system take about 1.2 seconds, which is still very acceptable.

4.4.2 Simulation performance

Finally, the performance of the simulation can be measured. Here the performance of the error injecting simulation is compare to the performance of the same simulation which uses a standard memory. Again, these measurements are done using Hyperfine [43].

Listing 4.5 shows the execution times of both the simple simulation and the simulations with the error injection. These results show that the simple simulator is quite a bit faster, where the simple simulator takes about 3 seconds, while the error injecting simulator takes approximately 5 seconds. This means that the simple simulation runs approximately 1.6 times faster. Both of these simulations ran for $5\,052\,828$ cycles, which means the simple simulator runs at approximately 1.62 megacycles per second, while the error injecting simulator runs at 1.02 megacycles per second.

To determine what is causing this lower performance, the simulator was profiled using the Linux `perf` tool. Listing 4.6 shows an extract of the profiling report, which shows that the actual error injection and memory simulation code is not taking much of the simulation time. Instead, most of the time is still spent evaluating the design. It seems that the error injection code itself does not add much execution time, however using the black-box feature seems to significantly reduce the optimizations in the evaluation of the design.

```
Benchmark 1: ./baseline coremark-perf.bin --stdout
    Time (mean ± s):      3.113 s ±  0.034 s    [User: 3.112 s, System: 0.000 s]
    Range (min . max):    3.080 s .  3.202 s    10 runs

Benchmark 2: ./simulation coremark-perf.bin --stdout
    Time (mean ± s):      4.969 s ±  0.048 s    [User: 4.965 s, System: 0.001 s]
    Range (min . max):    4.917 s .  5.057 s    10 runs

'./baseline coremark-perf.bin --stdout' ran
    1.60 ± 0.02 times faster than './simulation coremark-perf.bin --stdout'
```

Listing 4.5: Performance comparison between plain CXXRTL simulation and CXXRTL with error injection simulation.

```
# Overhead  Samples  Symbol
# ........  .......  ..................................................
    75.70%    14871  cxxrtl_design::p_top::eval
    21.98%     4317  cxxrtl_design::p_top::commit
     1.03%      203  std::poisson_distribution<unsigned long>::operator()
     0.67%      132  cxxrtl_design::bb_p_memory_impl<10ul, 39ul>::eval
     0.32%       62  std::mersenne_twister_engine<...>::_M_gen_rand
     0.19%       37  clk
# Lines with lower sample count removed for brevity.
```

Listing 4.6: Profiling results for a run of the error injecting simulator.

```
Benchmark 1: parallel -j16 './baseline coremark-perf.bin --stdout -s {1}' ::: {0..15}
    Time (mean ± s):      3.380 s ±  0.052 s    [User: 50.947 s, System: 0.043 s]
    Range (min . max):    3.312 s .  3.457 s    10 runs

Benchmark 2: parallel -j16 './simulation coremark-perf.bin --stdout -s {1}' ::: {0..15}
    Time (mean ± s):      5.436 s ±  0.047 s    [User: 83.055 s, System: 0.052 s]
    Range (min . max):    5.367 s .  5.514 s    10 runs

'parallel -j16 './baseline coremark-perf.bin --stdout -s {1}' ::: {0..15}' ran
    1.61 ± 0.03 times faster than 'parallel -j16 './simulation coremark-perf.bin
                                                  --stdout -s {1}' ::: {0..15}'
```

Listing 4.7: Performance comparison of simulations when running 16 threads in parallel.

```
Benchmark 1: parallel -j28 './baseline coremark-perf.bin --stdout -s {1}' ::: {0..27}
    Time (mean ± s):      5.594 s ±  0.058 s    [User: 139.301 s, System: 0.093 s]
    Range (min . max):    5.497 s .  5.687 s    10 runs

Benchmark 2: parallel -j28 './simulation coremark-perf.bin --stdout -s {1}' ::: {0..27}
    Time (mean ± s):      8.850 s ±  0.054 s    [User: 223.303 s, System: 0.098 s]
    Range (min . max):    8.761 s .  8.931 s    10 runs

'parallel -j28 './baseline coremark-perf.bin --stdout -s {1}' ::: {0..27}' ran
    1.58 ± 0.02 times faster than 'parallel -j28 './simulation coremark-perf.bin
                                                  --stdout -s {1}' ::: {0..27}'
```

Listing 4.8: Performance comparison of simulations when running 28 threads in parallel.

Furthermore, all of these measurements assumed no other load on the system, however in most cases the simulator would be run many times in parallel, to reduce the overall runtime of many simulations. The runtime for running both 16 and 28 simulations in parallel was measured, and are shown in listings 4.7 and 4.8.

Interestingly, while running 16 parallel simulations is basically as fast as running a single simulation, running 28 simulations in parallel has a higher execution time. Calculating the number of simulations completed per second, with 16 parallel executions it completes 2.94 simulations per second, while with 28 parallel executions it completes 3.16 simulations per second. So going past 16 of the available 32 threads on this AMD Ryzen™ 9 5950X processor does not significantly increase the number of simulations executed per second. This is likely due to bottlenecks in simultaneous multithreading.

In the end, approximately 3 simulations can be finished every second. Running all the simulations for a single design would require running 1701 simulations, which would finish in approximately 9 and a half minutes. Finishing the simulations for all designs would take approximately 3 hours and 45 minutes.

4.5 Conclusions

In this chapter both the performance of multiple error correction codes and memory controllers, and the execution times of the generation and simulation tools, were discussed. Designs using a parity, Hamming, extended Hamming, Hsiao, Dutta Touba, and She Li code were shown, in combination with a basic, write-back, and refresh controller. All of these designs were simulated to determine the error correction effectiveness and performance overhead, and were laid out using the Skywater 130nm PDK to determine the area, critical path delay and power usage. Finally, the execution times for generating the error correction codes, generating the hardware designs, and simulating the test system with error injection were measured.

In terms of error correction effectiveness, the tested error correction codes can be split into two main groups. The parity, extended Hamming and Hsiao codes are all almost perfect in correcting or detecting errors, while the other codes had between 10 and 40% of simulations missing errors. One of the codes with almost perfect correction and detection capabilities is likely the best choice in designs. However, in the case where adjacent errors are possible, the Dutta Touba and She Li codes might be a better choice, as they are actually able to correct adjacent errors. Furthermore, the refresh controller is clearly the best choice in terms of error correction effectiveness, followed by the write-back controller. The basic controller should not be used in most designs, however in cases with only error detection it might be used,as the other controller additional value is in the error correction.

Using the ASIC layout results the selection of best choices for error correction codes can be reduced even more. The critical path delay results clearly show that the extended Hamming code is quite a bad choice, as it has a significantly higher critical path delay. On the other hand, the parity and Hsiao codes only add a small bit of extra critical path delay. The choice between parity code and Hsiao code depends on the required error correction capabilities and acceptable are overhead. The parity code provides almost perfect single error detection with a small 2.65% memory area overhead, while the Hsiao code provides single error correction and double error detection with a larger 19.98% memory area overhead.

Finally, based on the execution times for generating the error correction code matrices, the code implementations can be divided into two categories. There are implementations that can compute the matrices by construction, which means they are quite fast, while other implementation require searching through all or a large subset of the possible matrices to find a valid and efficient code. Both the Dutta Touba code and She Li code require such a large search and are quite slow, requiring multiple seconds or minutes to find a valid matrix. The `HsiaoCode` implementation also takes a search approach, which is quite fast for 32 data bits and smaller, but which becomes much slower for 64 data bits and larger. In most cases using the `HsiaoConstructedCode` implementation is faster, while producing similar results.

The time it takes to generate a Verilog implementation from the memory controller and test system designs was shown to be 300ms and 1.2s respectively. This is more than fast enough for most experimentation, and is much faster than other steps. Running a single simulation of the test system design with error injection was shown to take approximately 5 seconds,

which is about 1.6 times slower than a simple simulation of the system. However, most of this additional time was not spent in the actual error injection and statistics collection code, instead it was spent in the simulation code, which optimizes worse with black-box memory implementation. Furthermore, on the computer that was used, approximate 3 simulations could finish per second when running with 16 or more simulations in parallel. Meaning that running the simulations for the range shown in this research would take approximately 10 minutes.

Chapter 5

Conclusions and future work

In this thesis the design and implementation of a tool to generate error correcting memory controller, and a tool to simulate systems with errors injected into the memory, were shown. The memory controller generator tool was implemented with a number of error correction codes and memory controller designs, and allows for easy addition of new codes or controllers. The simulator implements simulating a complete test system with errors injected into the memory.

Using these tools a number of experiments were conducted on the parity, Hamming, extended Hamming, Hsiao, Dutta-Touba, and She-Li codes. These codes were combined with a basic, write-back on error and refreshing memory controller. Of these designs the parity, extended Hamming and Hsiao codes stood out with near perfect error correction or detection capabilities. However, during the ASIC layout the extended Hamming code was shown to have poor performance when actually implemented in hardware. The refresh controller showed the best results independent of the error correction code used, but was also the most expensive to implement. Finally, in the case where adjacent 2-bit errors were simulated the Dutta-Touba and She-Li codes stood out, as all other codes struggled to detect or correct errors.

In the end, a combination of the parity code with the basic controller is a great design for low cost and high speed, with the downside of only detecting errors. Another great option was the Hsiao code combined with the refresh controller, which provides great error correction and detection capabilities, still at high speed, but with a more expensive implementation cost. A number of other memory controller designs were tried in appendix C, however these designs were not generally useful, although they might preform better in some specific scenarios.

Finally, the execution performance of both the memory controller generator tool and the error injecting simulator were measured. In most cases calculating the matrices for error correction codes in the memory controller generator was fast, only taking a few milliseconds, however the Dutta-Touba and She-Li codes can take minutes or even hours to completely optimize. With the matrices calculated, generating the actual hardware in Verilog only takes 300 millisecond for the just the memory controller, or 1.2 seconds for the complete test system. Simulating the test system with a program taking approximately 5 million cycles resulted in an execution time of approximately 5 seconds, which equates to simulation speed of 1 million cycles per second. On a large multicore processor 16 simulation could be computed in parallel all at 1 million cycles per second. A simple simulator without the additional error

injection or statistics collection was able to run at approximately 1.6 times the speed of the simulator created in this thesis.

While the tools created in this research, and the experiments done using these tools, already covered quite a large number of error correction codes and memory controllers designs, a number of points still allow for future work. The future work is split up into two sections containing work on improving the tools, and further experimentation using these tools.

5.1 Further work on tools

Both the tools specifically created for this research, and tools created by others used in this research could use additional work. The specific additions and pain points are summarized here.

Improving the error correction code implementations. The current implementations of the `DuttaToubaCode` and the `SheLiCode` are based on the `BoolectorCode` framework. While this does allow for generating and optimizing the matrices for these codes, it is quite slow. Implementations of these code using other optimization methods might improve the execution time and results of these codes.

Decoupling the simulator from the test system. At this point the error injecting simulator is still quite tightly integrated with the specific test system used in this research. This means that reusing it with a different test system would require some work. Fortunately, most of the error injection and statistics collection is contained in the black-box memory module, which should make it easier to extract this functionality into a library. This library could then be used with other test systems to reuse the error injection capabilities.

Adding support for complex errors and interleaving to the simulator. While the current version of the simulator supports adjacent errors, the implementation is quite simple. Future versions of the simulator could allow for the injection of more interesting error patterns, each with their own probabilities. In addition to the expanded error injection, simulation of interleaving could also be beneficial to the applicability of the simulations, as this is often used in real world memories.

Improving the supporting tools. In addition to improving the tools built in this research, the other tools used by this research, especially OpenLane and all the tools it uses, could also use some improvement. While a completely free and open-source flow for ASIC layout is great to use, there are some pain points using OpenLane. Most significant are the problems with routing large designs containing multiple SRAM blocks. Currently it gets stuck on the detailed routing very often, because of too much congestion around the SRAM blocks. There might be some option that I missed during this research, in which case the documentation should

be improved significantly, otherwise the placer and router should keep more space around the SRAM blocks to improve routability. Furthermore, the power analysis currently available in practically useless, and only provides a ballpark figure of power usage. And finally, it would be great if the timing analysis could work across multiple macro blocks, instead of just assuming a specified input or output delay.

5.2 Further research using this framework

In addition to improving the tools in this framework, further research on error correction codes and memory controllers could be done using these tools. A number of research ideas using these tools are listed here.

Additional error correction codes. Next to the currently implemented error correction codes, there are a large number of other error correction codes that were are not implemented yet. For example, the Neale [44], [45] codes which were discussed. Furthermore, there are multi-bit random error correcting codes such as Reed-Solomon (RS) and Bose-Chaudhuri-Hocquenghem (BCH) codes, and Orthogonal Latin Square Codes [46]–[48]. All of these codes could be implemented and evaluated using this framework.

Effects of interleaving. The effects of interleaving memory bits could be researched using these tools, with some additional work on the simulator, as discussed earlier. Since this technique is used quite often in the real-world it could help determine if adjacent error correction codes are needed.

Additional partial write strategies. The current implementation only supports writing partial words using the read-modify-write partial write wrapper. However, other methods are possible as described in background information. Changing the partial write support might improve the performance of the memory controller. Furthermore, a memory controller using per byte parity checks might have interesting performance gains, while not requiring large overhead.

Memory banking strategies. The designs in this research all used only a single large bank of memory, with the memory controller managing all the memory. However, in some cases duplicating the memory controller for different memory banks might help with hiding the additional delay that the memory controller might introduce. This could improve the overall performance of the design, especially with higher load on the memory.

Pipelining and flexible correction delay. All of the designs presented here use a completely combinatorial data path for the encoding and decoding of the data. This results in quite large critical path delays, which might impact the speed at which the complete system can

operate. For systems that are much more pipelined it might make sense to also pipeline the memory controller implementation, reducing the critical path delay. Furthermore, there is a possibility for flexible delays, where the syndrome is calculated quickly to determine if an error happened, but the error correction can take multiple cycles. This can improve the overall critical path delay, while not impacting the system performance unless an error is detected.

Bibliography

[1] R. Haas. "The arm ecosystem ships a record 6.7 billion arm-based chips in a single quarter." (2021), [Online]. Available: https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter (visited on 03/11/2021).

[2] J. McMaster. "Raspberry pi rp2040." (2021), [Online]. Available: https://siliconpr0n.org/map/raspberry-pi/rp2-b0/s1-9_mit20x/ (visited on 01/29/2022).

[3] M. Svarichevsky. "GD32F103CBT6 - Cortex-M3 with serial flash : Weekend die-shot." (2016), [Online]. Available: https://zeptobars.com/en/read/GD32F103CBT6-mcm-serial-flash-Giga-Devices (visited on 01/29/2022).

[4] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, pp. 258–266, 3 2005.

[5] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study," in *SIGMETRICS*, Association for Computing Machinery, 2009, p. 323, ISBN: 9781605585116.

[6] R. Hamming, "Error detecting and error correcting codes," *The Bell Systems Technical Journal*, vol. 29, pp. 147–160, 2 1950. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[7] M. Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *IBM Journal of Research & Development*, pp. 395–401, 1970.

[8] L. Chen, "Hsiao-code check matrices and recursively balanced matrices," Mar. 2008. [Online]. Available: https://arxiv.org/abs/0803.1217.

[9] A. Dutta and N. A. Touba, "Multiple bit upset tolerant memory using a selective cycle avoidance based sec-ded-daec code," in *25th IEEE VLSI Test Symposium (VTS'07)*, 2007, pp. 349–354. DOI: 10.1109/VTS.2007.40.

[10] X. She, N. Li, and D. W. Jensen, "Seu tolerant memory using error correction code," *IEEE Transactions on Nuclear Science*, vol. 59, pp. 205–210, 1 PART 2 Feb. 2012, ISSN: 00189499. DOI: 10.1109/TNS.2011.2176513.

[11] M. S. M. Siddiqui, S. Ruchi, L. V. Le, T. Yoo, I. J. Chang, and T. T. H. Kim, "Sram radiation hardening through self-refresh operation and error correction," *IEEE Transactions on Device and Materials Reliability*, vol. 20, pp. 468–474, 2 Jun. 2020, ISSN: 15582574. DOI: 10.1109/TDMR.2020.2994769.

[12] J.-F. Li and Y.-J. Huang, "An error detection and correction scheme for rams with partial-write function," 2005, pp. 115–120. DOI: 10.1109/MTDT.2005.16.

[13] Xilinx. "AXI block RAM (BRAM) controller v4.0." (2016), [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_0/pg078-axi-bram-ctrl.pdf (visited on 06/19/2021).

[14] Altera. "Error correction code: Megafunctions user guide." (2008), [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altecc.pdf (visited on 06/19/2021).

[15] Lattice Semiconductor. "ECP5 and ECP5-5G memory usage guide." (2015), [Online]. Available: https://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/EH/TN1264.ashx?document_id=50466 (visited on 06/19/2021).

[16] N. De Simone. "Yet another hamming encoder and decoder." (2017), [Online]. Available: https://opencores.org/projects/yahamm (visited on 06/19/2021).

[17] "Configurable hamming generator." (2013), [Online]. Available: https://opencores.org/projects/hamming_gen (visited on 06/19/2021).

[18] M. JiJi. "Configurable BCH encoder and decoder." (2015), [Online]. Available: https://opencores.org/projects/bch_configurable (visited on 06/19/2021).

[19] CAU SSE. "Reed solomon (9,5) encoder/decoder." (2019), [Online]. Available: https://opencores.org/projects/reed_solomon_coder (visited on 06/19/2021).

[20] CAST Silicon IP Cores. "ECC-SRAM: Error correcting code for SRAMs." (2021), [Online]. Available: https://cast-inc.com/peripherals/memory-controllers/ecc-sram/ (visited on 06/19/2021).

[21] Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121." (2019), [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf (visited on 01/12/2022).

[22] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A TileLink case study," in *Proceedings of First Workshop on Computer Architecture Research with RISC-V, Boston, MA USA, October 2017 (CARRV'17)*, 2017.

[23] SiFive, Inc. "SiFive TileLink Specication." (2020), [Online]. Available: https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf (visited on 01/12/2022).

[24] OpenCores. "WISHBONE System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores." (2010), [Online]. Available: https://cdn.opencores.org/downloads/wbspec_b4.pdf (visited on 01/12/2022).

[25] Whitequark. "Amaranth: A modern hardware definition language and toolchain based on python." (2021), [Online]. Available: https://github.com/amaranth-lang/amaranth (visited on 01/09/2022).

[26] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014. DOI: 10.3233/sat190101.

[27] YosysHQ. "SymbiYosys (sby) documentation." (2021), [Online]. Available: https://symbiyosys.readthedocs.io/en/latest/index.html (visited on 06/14/2021).

[28] Whitequark. "Cxxrtl." (2020), [Online]. Available: https://github.com/YosysHQ/yosys/tree/master/backends/cxxrtl (visited on 01/13/2022).

[29] YosysHQ. "Yosys Open SYnthesis Suite." (2020), [Online]. Available: https://github.com/YosysHQ/yosys (visited on 01/13/2022).

[30] C. Wolf and J. Glaser, "Yosys-a free verilog synthesis suite," in *21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[31] EEMBC. "CoreMark - EEMBC Embedded Microprocessor Benchmark Consortium." (2021), [Online]. Available: https://www.eembc.org/coremark/index.php (visited on 01/13/2022).

[32] Skywater Technology. "Google Partners with SkyWater and Efabless to Enable Open Source Manufacturing of Custom ASICs." (2020), [Online]. Available: https://www.skywatertechnology.com/press-releases/google-partners-with-skywater-and-efabless-to-enable-open-source-manufacturing-of-custom-asics/ (visited on 01/14/2022).

[33] T. Ansell. "Skywater open source pdk." (2020), [Online]. Available: https://github.com/google/skywater-pdk (visited on 06/15/2021).

[34] A. A. Ghazy and M. Shalan, "Openlane: The open-source digital asic implementation flow," in *Workshop on Open-Source EDA Technology (WOSET)*, 2020.

[35] A. Gouhar, A. A. Ghazy, *et al.* "OpenLane." (2020), [Online]. Available: https://github.com/The-OpenROAD-Project/OpenLane (visited on 01/14/2022).

[36] T. Ajayi, D. Blaauw, T.-B. Chan, C.-K. Cheng, V. A. Chhabria, K. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Li, Z. Liang, U. Mallappa, P. Penzes, G. Pradipta, S. Reda, A. Rovinski, K. Samadi, S. S. Sapatnekar, L. Saul, C. Sechen, V. Srinivas, W. Swartz, D. Sylvester, D. Urquhart, L. Wang, M. Woo, and B. Xu, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," in *Government Microcircuit Applications and Critical Technology Conference*, 2019, pp. 1105–1110.

[37] J. Cherry, E. Monteiro, M. Liberty, O. Hammad, *et al.* "OpenROAD." (2019), [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD (visited on 01/14/2022).

[38] T. Edwards. "Magic VLSI Layout Tool." (2020), [Online]. Available: http://opencircuitdesign.com/magic/ (visited on 01/14/2022).

[39] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "Openram: An open-source memory compiler," in *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, vol. 07-10-November-2016, Institute of Electrical and Electronics Engineers Inc., Nov. 2016, ISBN: 9781450344661. DOI: 10.1145/2966986.2980098.

[40] M. Guthaus. "OpenRAM." (2021), [Online]. Available: https://github.com/VLSIDA/OpenRAM (visited on 01/14/2022).

[41] M. V. Thun, D. Walz, R. Dumitru, A. Wilson, and T. Farris, "Seu characterization of an embedded 130nm compiled sram in heavy ion and proton environments," 2017, pp. 1–5. DOI: 10.1109/NSREC.2017.8115462.

[42] L. D. van Harten, M. Mousavi, R. Jordans, and H. R. Pourshaghaghi, "Determining the necessity of fault tolerance techniques in fpga devices for space missions," *Microprocessors and Microsystems*, vol. 63, pp. 1–10, Nov. 2018, ISSN: 01419331. DOI: 10.1016/j.micpro.2018.08.001.

[43] D. Peter *et al.* "Hyperfine: A command-line benchmarking tool." (2020), [Online]. Available: https://github.com/sharkdp/hyperfine (visited on 01/27/2022).

[44] A. Neale and M. Sachdev, "A new sec-ded error correction code subclass for adjacent mbu tolerance in embedded memory," *IEEE Transactions on Device and Materials Reliability*, vol. 13, pp. 223–230, 1 2013, ISSN: 15304388. DOI: 10.1109/TDMR.2012.2232671.

[45] A. Neale, M. Jonkman, and M. Sachdev, "Adjacent-mbu-tolerant sec-ded-taec-yaed codes for embedded srams," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, pp. 387–391, 4 Apr. 2015, ISSN: 15497747. DOI: 10.1109/TCSII.2014.2368262.

[46] L. Xiao, J. Li, J. Li, and J. Guo, "Hardened design based on advanced orthogonal latin code against two adjacent multiple bit upsets (mbus) in memories," in *Proceedings - International Symposium on Quality Electronic Design, ISQED*, vol. 2015-April, IEEE Computer Society, Apr. 2015, pp. 485–489, ISBN: 9781479975815. DOI: 10.1109/ISQED.2015.7085473.

[47] P. Reviriego, S. Pontarelli, A. Evans, and J. A. Maestro, "A class of sec-ded-daec codes derived from orthogonal latin square codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, pp. 968–972, 5 May 2015, ISSN: 10638210. DOI: 10.1109/TVLSI.2014.2319291.

[48] T. Shruthymol and Z. C. Reneesh, "Modified orthogonal latin square codes for sec-daec-daaec-taec," *International Journal of Scientific Development and Research*, vol. 1, 7 2016, ISSN: 2455-2631. [Online]. Available: www.ijsdr.org.

[49] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced srams," 2003, pp. 21.4.1–21.4.4. DOI: 10.1109/IEDM.2003.1269335.

[50] D. Radaelli, H. Puchner, S. Wong, and S. Daniel, "Investigation of multi-bit upsets in a 150 nm technology sram device," vol. 52, Dec. 2005, pp. 2433–2437. DOI: 10.1109/TNS.2005.860675.

[51] G. Gasiot, D. Giot, and P. Roche, "Multiple cell upsets as the key contribution to the total ser of 65 nm cmos srams and its dependence on well engineering," in *IEEE Transactions on Nuclear Science*, vol. 54, Dec. 2007, pp. 2468–2473. DOI: 10.1109/TNS.2007.908147.

[52] E. H. Cannon, M. S. Gordon, D. F. Heidel, A. J. KleinOsowski, P. Oldiges, K. P. Rodbell, and H. H. Tang, "Multi-bit upsets in 65nm soi srams," in *IEEE International Reliability Physics Symposium Proceedings*, 2008, pp. 195–201, ISBN: 9781424420506. DOI: 10.1109/RELPHY.2008.4558885.

[53] N. Abramson, "A class of systematic codes for non-independent errors," *IRE Transactions on Information Theory*, vol. 5, pp. 150–157, 4 1959. DOI: 10.1109/TIT.1959.1057524.

[54] D. W. Jensen, "Block code to efficiently correct adjacent data and/or check bit errors," 6 604 222, 2003.

[55] L. J. Saiz-Adalid, P. Reviriego, P. Gil, S. Pontarelli, and J. A. Maestro, "Mcu tolerance in srams through low-redundancy triple adjacent error correction," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, pp. 2332–2336, 10 Oct. 2015, ISSN: 10638210. DOI: 10.1109/TVLSI.2014.2357476.

Appendix A

Transforming parity-check and generator matrices

In eqs. (2.9) and (2.10) the standard form of the parity-check and generator matrices were defined. Unfortunately, in the real world these matrices are not always in standard form. Therefore, it is useful to be able to transform these matrices, while maintaining the properties of their specific error detection and correction codes.

First of all, both the parity-check matrix $\boldsymbol{H}$ and the generator matrix $\boldsymbol{G}$ can be transformed using the elementary row operations. There are three elementary row operations, swapping, multiplication and addition. Row swapping is exchanging the position of two rows in the matrix. Row multiplication is multiplying a row by a non-zero constant. In the case of a binary matrix this operation is not used, as the only non-zero constant is one. And finally, row addition, where a row is replaced by the sum of itself and another row.

All these operations do not affect the relationship between the parity-check and generator matrices. Any combination of operations can be applied to either matrix without changing their relationship. However, operations on the generator matrix do change the way the message is encoded and can require changes in the decoding apart from the parity-checking. Similarly, for the parity-check matrix these changes alter the resulting syndromes, which means the error checking needs to be aware of those changes.

These elementary row operations are not sufficient to transform every parity-check matrix or generator matrix to standard form. To achieve this the column swap operation is also needed, however, this operation cannot be used as freely as the elementary row operations. Swapping columns in either the generator matrix or the parity-check matrix will alter the order of the encoded bits, therefore also requiring a similar column swap in the other matrix. When applying column swap operations to either matrix, the same operations will have to be applied to the other matrix in reverse order.

Using these operations, any parity-check matrix that is not in standard form can be transformed into a standard form matrix. This procedure is very similar to Gauss-Jordan elimination that is used in linear algebra. However, with the definition of the standard form parity-check matrix as defined in eq. (2.9), the procedure has to be altered slightly to get the identity matrix at the right side of the matrix, instead of the left. The following procedure

can be used to transform a parity-check matrix to standard form.

1. Swap rows such that the last entry of the last row is a one, if this is not possible swap a column to achieve the same result. If this is still not possible, the matrix contains a row of all zeros, which means that there is a redundant row, which can be removed.
2. Add the last row to all other rows, such that the last entry in every other row becomes zero.
3. Repeat from step 1, ignoring the last row and last column, as long as there are rows left.
4. Finally, add every row to any subsequent rows to clear out the lower triangle of the identity matrix.

After following this procedure the parity-check matrix is in standard form, after which it can be used to create a corresponding generator matrix. This generator matrix in standard form can then be transformed to a corresponding generator matrix for the original parity-check matrix by reversing the column swap operations applied in procedure above.

Appendix B

Error detection and correction codes

This appendix contains an in-depth explanation of all of the error detection and correction codes used in this research. For each of these error detection and correction codes the actual construction of their parity-check matrices is described. Furthermore, this appendix also contains a few more error correction codes, which were not used in the main research, but which were investigated.

B.1 Parity check code

The parity check code is the oldest and simplest error detection code, which is based on the concept of parity from mathematics. In mathematics, parity is the divisibility of an integer by two, better known as even or odd. This concept of parity is applied to a sequence of bits by summing the bits and determining the parity of this sum. Finally, the parity of the sum is added to the message to create an encoded message that can detect a single bit error.

Based on this textual description of encoding a message, the generator matrix can be defined as

$$\boldsymbol{G} = \left[\begin{array}{c|c} \boldsymbol{I}_k & \boldsymbol{J}_{k,1} \end{array} \right], \tag{B.1}$$

where $\boldsymbol{J}_{k,1}$ is the all-ones matrix with size $k \times 1$. Since the $\boldsymbol{A}$ part of this generator matrix is $\boldsymbol{J}_{k,1}$, which is only a single column wide, only a single bit will be added to the original message. This single bit will be the sum of all the message bits, which calculates the parity.

Based on the generator matrix from eq. (B.1), which is in standard form, the parity-check matrix

$$\boldsymbol{H} = \left[\begin{array}{c|c} \boldsymbol{J}_{k,1}^T & \boldsymbol{I}_{n-k} \end{array} \right] = \left[\begin{array}{c|c} \boldsymbol{J}_{1,k} & \boldsymbol{I}_1 \end{array} \right] = \left[\begin{array}{c|c} \boldsymbol{J}_{1,k} & 1 \end{array} \right] = \left[\begin{array}{cccc} 1 & 1 & \dots & 1 \end{array} \right]_{n \times 1} \tag{B.2}$$

can be derived, which is actually just a single row of n ones. Therefore, checking the validity of an encoded message using eq. (2.1) is as simple as adding all the encoded message bits and checking that the result is zero. If the result of this calculation is one, the code has detected an error. This code is not able to correct the detected error, because the columns of the parity-check matrix are not unique, so any one bit error will result in the same syndrome.

This code does actually do more than just detecting a single error, it can actually detect any odd number of errors. Unfortunately, it cannot detect an even number of errors, so with two errors the encoded message will be decoded incorrectly. Therefore this code is practically just a single error detection code, however theoretically it can detect any odd number of errors.

This parity check code on its own is not very interesting, as it is only able to detect a single error and cannot correct it. However, the concept of parity checks can be extended to create more complex error detection and correction codes.

B.2 Hamming code

In his 1950 paper [6], Richard Hamming was the first to describe a class error correction codes, which are able to correct a single error in a message. This error correction capability is constructed by using multiple parity checks over different subsets of the symbols in the message. These subsets are determined by the binary representation of the index of each symbol.

So, as an example with seven symbols, the first parity check will include all symbols where the lowest bit of the index is a one. That is the symbols at index 1 (001), 3 (011), 5 (101), 7 (111). The second parity check will include all symbols where the second lowest bit of the index is a one. That is the symbols at index 2 (010), 3 (011), 6 (110), 7 (111). The third parity check will include the symbols at index 4 (100), 5 (101), 6 (110), 7 (111).

Using these three parity check the position of any error can be determined. This can be done by checking that the parity of each subset of symbols is even. If the parity of a subset is not even, the symbol containing the error is part of that subset. Since the subsets are chosen based on the index of the symbol, the error index can always be calculated. For example, if the first parity check fails, the error must be at index 1, 3, 5, or 7. If the second parity check is correct, the error cannot be at index 2, 3, 6, or 7, so it can only be at index 1 or 5. Finally, the third parity check also fails, which means the error must be at index 4, 5, 6, or 7. Since only index 1 or 5 were left, the error position can be determined as the symbol at index 5.

Although it is common in computer science to start counting at zero, in this case zero is excluded as an index, this is because it would not be included in any parity check, since the index is zero. Furthermore, if index zero was a valid index, there would be no way to distinguish between an error at index zero and no error at all.

With this code detecting single errors is possible, however, there still has to be an allocation between actual data bits and redundancy bits, such that an arbitrary message can actually be encoded. To achieve this, every subset of symbols should contain one bit which is dedicated to redundancy. This allows the encoder to choose this bit such that the overall parity of the subset is even. In every subset there is only a single position that can be freely chosen without affecting any other subset, which is the position with an index that is a power of two. So, all positions where the index is not a power of two can be used for transmitting actual message symbols.

The relation between the number of symbols in the encoded message n and the number of redundancy symbols m can be defined as

$$n = 2^m - 1. \tag{B.3}$$

The number of actual data symbols k can be calculated from the total symbols n and the redundancy symbols m as

$$k = n - m = 2^m - m - 1. \tag{B.4}$$

Solving $2^m - m - 1 \geq k$ for positive integer values of m gives the minimum number of redundancy symbols required for a message of length k.

Using the textual description of the parity checking procedure, the parity-check matrix for a Hamming code can be constructed by concatenating column vectors containing the binary representation of the index. As an example, the parity-check matrix for a Hamming code where n is 7, is shown in fig. B.1a. Defining the parity-check matrix in this way means that the syndrome of an error will actually be the index at which the error happened. Unfortunately, this is not standard form, however the generator matrix, shown in fig. B.1b, can still be constructed using the procedure discussed in a previous section.

The Hamming code as defined here is only able to either correct a single error or detect two errors, that is, the minimum Hamming distance of this code is 3. This can easily be seen from the parity-check matrix shown in fig. B.1a, where a single error will produce an unique syndrome, but any two errors, will produce a syndrome which is also a valid single error syndrome.

Finally, Hamming codes can actually be extended with a single extra parity check over all the bits in the encoded message. This extra parity check increases the Hamming distance of this extended Hamming code from 3 to 4, giving the code the ability to correct a single error and detect when two error happen, without being able to correct them.

If no errors occur, this parity check will match up with the encoded parity bit. In the case of a single error, this parity check will actually fail, just like in the parity check code. However, in the case of two errors, this parity check will match up, but the other parity checks in the Hamming code will not match up, therefore allowing the distinction between a single error and two errors.

The parity-check matrix for this code can easily be constructed from the original Hamming code by adding a column of zeroes at the end and then a row of ones at the bottom. Take for

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(a) Parity-check matrix

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(b) Generator matrix

Figure B.1: Matrices for (7,4)-SEC Hamming code

example the parity-check matrix shown in fig. B.1a, extending this matrix will result in the parity-check matrix shown in fig. B.2.

$$\boldsymbol{H}_{ext} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure B.2: Parity-check matrix of (8,4)-SEC-DED Extended Hamming code

B.3 Hsiao code

Hsiao codes, as first described in [7], are an interesting evolution on the Extended Hamming code. While Hsiao codes are theoretically equivalent to Extended Hamming codes, they do provide some benefits during the actual implementation in hardware. Therefore, they are often preferred over the normal Extended Hamming code.

Hsiao codes are defined by a number of constraints on the parity-check matrix. These constraints are, as defined in [7], the following:

1. There are no all zero columns.
2. Every column is distinct.
3. Every column contains an odd number of ones.

When the parity-check matrix satisfies the first two constraints it defines a code with Hamming distance 3, like the normal Hamming code. The third constraint forces the code to be a Hamming distance 4 code, like the Extended Hamming code. While the Extended Hamming code used an additional parity bit to get a minimum Hamming distance of 4, the Hsiao code uses the constraint of an odd number of ones in every column. When adding two columns with an odd number of ones, the result will always be a column with an even number of ones, therefore any double can be detected by checking the number of ones in the syndrome.

On its own this choice seems quite arbitrary, however it is important to see that the number of ones in a row of the parity-check matrix relate to the depth of the logic circuit that calculates that bit of the syndrome. With a larger number of ones in a row, the logic circuit will have to be larger and slower to calculate the syndrome bit. Therefore, the goal is to minimize the maximum number of ones in a row, since the row with the largest number of ones dominates the time required to calculate the syndrome. The specifics on how this exactly works are discussed in the later section on the hardware implementation of these error correction codes.

In the Extended Hamming code parity-check matrix this goal is clearly not reached, because of the extra parity check that was introduced to detect double errors, there is a single row with all ones. This means that the Extended Hamming code is the slowest that is possible. With the design of the Hsiao code there are an additional two optimisation constraints with

the goal of improving the speed of the implementation. Those constraints, as defined in [7], are the following:

1. The total number of ones in the parity-check matrix should be minimal.
2. The number of ones in each row of the parity-check matrix should be made equal, or as close as possible, to the average number of ones in a row, the total number of ones in the parity-check matrix divided by the number of rows in the matrix.

These constraints follow from the goal to improve the speed of the implementation.

A method for actually generating such a parity-check matrix efficiently is more complex. Generating the matrix for smaller data sizes can be done using a brute-force approach, checking all possible matrices of that specific size and picking one with optimal results. However this approach is not very elegant, so in a 1986 paper a recursive method for generating these matrices was defined. Unfortunately, this paper was written in Chinese and was largely unknown to the international community. In 2008 the original author of this paper released an English version of this paper [8] with some corrections.

As an example, the parity-check matrix of an eight data bit Hsiao code can easily be generated using a brute-force approach. One of such optimal Hsiao parity-check matrices is shown in fig. B.3 where the first four rows all have 6 ones and the last row 5 ones. This parity-check matrix meets all three constraints, every column is non-zero, distinct and contains only an odd number of ones. The optimisation goals are also met by this matrix. Since it uses all possible columns with a single one and only columns with 3 ones for the remaining columns, it meets the first optimisation goal of a minimum number of ones. Furthermore, since there are 29 ones in this matrix, each row should have $5\frac{4}{5}$ ones on average to balance the rows. Since a fractional number of ones per row is impossible, this is the closest option with an integer number of ones per row.

$$\boldsymbol{H} = \left[\begin{array}{cccccccccccc} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

Figure B.3: Parity-check matrix for (13,8)-SEC-DED Hsiao code (generated)

B.4 Adjacent error correction codes

Both variations of the Hamming code and the Hsiao code are only capable of correcting a single error. However, with the decreasing feature size of modern transistor processes, the chances for multiple errors in a single word are increasing [49]–[52]. The chance of multiple errors depends heavily on the specific memory technology, but with any somewhat recent technologies the chances range from a few percent to over half of all upset events.

Interestingly, these multi bit upsets often cause two neighbouring, or almost neighbouring, bits to flip. This property of these multi bit upsets can be exploited to create so called adjacent error correction codes. These adjacent error correction codes are able to correct two bits, or sometimes even more, but only if the errors happen in adjacent, or almost adjacent bits. Using this technique the redundancy required is similar to that of the extended Hamming code, or Hsiao code, instead of increasing massively as is the case with random two bit error correction codes. Unfortunately, these codes do often make a trade-off, two bit error detection is no longer perfect, instead there is a chance that a random two bit error gets miscorrected as an adjacent two bit error.

B.4.1 Double Adjacent Error Correction (DAEC)

In 1959, Abramson [53] designed the first iteration of this type of code. However, this code was not specifically targeted towards error correction in memory applications, instead is was more focussed on radio transmissions. Because of this, the encoder and decoder that were described in this paper are using a shift-register approach, which takes many cycles to correct a single word. An improved encoder and decoder could be constructed from the parity-check matrix for the Abramson code, however this would still not perform optimally. As an example, take the parity-check matrix shown in fig. B.4. The biggest problem with this code is the all ones row, which means that the parity of all message bits has to be calculated, similarly to the extended Hamming code. Furthermore, this code can protect one data bit less for the same number of parity bits, compared to the extended Hamming code or Hsiao code. And finally, this code cannot detect non-adjacent two bit errors in almost all cases. In more than 90% of the possible non-adjacent two bit errors, the result is a miscorrection.

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure B.4: Parity-check matrix for (15,10)-SEC-pDED-DAEC Abramson code [53, Fig. 3]

B.4.2 Improved Double Adjacent Error Correction (DAEC)

In 2007, Dutta and Touba [9] proposed a new adjacent error correction code specifically targeted towards memory error correction. To overcome the limitations of the Abramson code [53], this code was designed similarly to the Hsiao code, also using only columns with an odd number of ones. This code is defined with four constraints on the parity-check matrix. These conditions as given in the paper [9] are the following:

1. No all 0 columns. (i.e. no linear dependencies involving 1 column)
2. All columns are distinct. (i.e. no linear dependencies involving 2 columns)

3. No linear dependencies involving 3 or fewer columns. (This condition is partly redundant, as the previous two conditions already require no linear dependencies involving 1 or 2 columns)
4. No linear dependencies involving columns C_i, C_j, C_k, C_m, where $m > k > j > i$, such that $j = i + 1$ and $m = k + 1$.

These conditions restrict all linear dependencies with 3 or fewer columns, meaning this code can correct single errors and detect two bit errors. However, some linear dependencies of 4 columns are allowed, meaning that this code cannot correct every two bit error, instead the restricted linear dependencies of 4 columns are chosen such that two bit adjacent errors can be corrected.

Finally, there is one optimisation goal for this code. The code should minimise the number of linear dependencies involving columns C_i, C_j, C_k, C_m, where $m > k > j > i$, such that $j = i + 1$ or $k = j + 1$ or $m = k + 1$. This optimisation goal represents the cases where a random two bit error overlaps in syndrome with an adjacent two bit error, causing the random two bit error to be miscorrected into the adjacent error. Therefore the goal is to minimise the cases where this misprediction happens.

Actually generating parity-check matrices that conform to these conditions is not easy. First of all there is a choice of which columns to use, in the case of this paper the columns that were used contained only an odd number of ones, just like the Hsiao code. Secondly, the order of the columns is also important, since this code is specifically looking at adjacent bits. Together this leads to the claim of the authors that finding the optimal matrix for a specific number of data bits is NP-complete. Instead the authors describe a pseudo-greedy search algorithm to find these matrices quickly, but not necessarily optimally. As an example a parity-check matrix for 16 data bits is shown in fig. B.5.

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure B.5: Parity-check matrix for (22,16)-SEC-pDED-DAEC code [9, Fig. 1]

B.4.3 Double Almost Adjacent (DAAEC) and Triple Adjacent Error Correction (TAEC)

In their 2012 paper [10], She and Li present are more complex adjacent error correction scheme based on a 2003 patent [54]. This error correction scheme provides multiple different adjacent error corrections for two bit adjacent errors, three bit adjacent errors and two bit almost adjacent errors. The two bit almost adjacent error correction can correct two errors

where there is a single unchanged bit in between the flipped bits. To provide these extended error correction capabilities, this code does require one extra parity-check bit in comparison to the Hsiao code, or the previously discussed code by Dutta and Touba [9].

The definition of a parity-check matrix for this type of code is actually quite simple. For the code to have all the specified properties, all of the following should be unique:

1. All single bit error syndromes (columns in $\boldsymbol{H}$)
2. All two bit adjacent error syndromes (xor of adjacent columns in $\boldsymbol{H}$)
3. All two bit almost adjacent error syndromes (xor of almost adjacent columns in $\boldsymbol{H}$)
4. All three bit adjacent error syndromes (xor of three adjacent columns in $\boldsymbol{H}$)

Even though the definition of the code like this is quite simple, generating a parity-check matrix that conforms to these conditions is much harder. Both the paper [10] and patent [54] suggest using an exhaustive search to find suitable matrices and picking the best matrix from the list of suitable matrices. Another possible method suggested by [10] is using a genetic algorithm to evolve a suitable matrix, however this suggestion is not worked out any further.

Unfortunately, the example parity-check matrix that is shown in [10, (4)] does not conform to the actual definition of this code. This can easily be seen from [10, Table 1] containing all syndromes, where the syndrome for the error (d8, d9) is the same as the syndrome for the error (d14, d15, c0). This means that this parity-check matrix cannot determine the different between these two errors, therefore it cannot correct both errors correctly. Interestingly, moving the identity part of this matrix to the front does make this a valid code. This is likely a mistake of the authors when they took the parity-check matrix from the patent [54], where the identity part of the matrix is in the front.

A different, but correct, parity-check matrix can be generated using an exhaustive search and is shown in fig. B.6. No only is this parity-check matrix actually correct, it is also more efficient to implement. This is because there are fewer ones in each row, and fewer ones overall.

In a 2015 paper [55], Saiz-Adalid et al. describe a similar type of code with the same properties, which they call a 3-bit burst error correction code. The parity-check matrices presented in this paper do correctly follow the conditions as specified, confirming that these matrices

$$\boldsymbol{H} = \begin{bmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

Figure B.6: Parity-check matrix for (23,16)-SEC-DAEC-DAAEC-TAEC code (generated)

do exist. However, they also present a slightly simpler code, which does not have the two bit almost adjacent error correction. They show that leaving out this correction features makes it possible to reduce the number of redundancy bits to the same number as a normal extended Hamming code in most cases. As an exception they were unable to find a (22,16)-SEC-DAEC-TAEC code.

B.4.4 Scalable Double or Triple Adjacent Error Correction and x-Adjacent Error Detection

In their 2013 [44] and 2015 [45] papers, Neale et al. introduced two similar error correction codes. These error correction codes are able to correct two or three adjacent errors, while also being able to detect up to x adjacent errors. The number of detected adjacent errors can be tweaked by increasing or decreasing the number of check bits used in the code. Next to the increased possibility of detecting adjacent errors, this code also reduces the miscorrection chance when more check bits are used.

These codes are defined using the following requirements for an x adjacent error detecting and y adjacent error correcting code.

1. All columns must be non-zero.
2. All columns must be distinct.
3. The xor-result of any two columns must not equal any of the individual columns.
4. The xor-result of any n-adjacent columns, where $2 \leq n \leq y$, must be distinct.
5. The xor-result of any n-adjacent columns, where $y < n \leq x$, must not equal any of the error correcting syndromes.

Using these requirements, the error correction code will be able to correct all single bit errors and y-bit adjacent errors, and detect all x-bit adjacent errors. However, this code also cannot always correctly detect a random two-bit error, but the chances for correctly detecting these errors is increased when increasing the y-bit adjacent error detection.

Constructing the parity-check matrix for these codes is a combination of exhaustive search and direction construction. The parity-check matrix of this code always follows the same structure, on the top there is a row of sub-matrices which are found using the search algorithm, while on the bottom there is a row of identity matrices. This can also be written as

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{H}_1 & \boldsymbol{H}_2 & \boldsymbol{H}_3 & \cdots & \boldsymbol{H}_{\lceil n/L \rceil} \\ \boldsymbol{I}_L & \boldsymbol{I}_L & \boldsymbol{I}_L & \cdots & \boldsymbol{I}_{L \times (n \bmod L)} \end{bmatrix}, \tag{B.5}$$

where $\boldsymbol{H}_i$ is a sub-matrix, which has to be found using a search algorithm, and $\boldsymbol{I}_L$ is the $L \times L$ identity matrix. In the case of the last identity matrix, a number of columns at the end might be trimmed off.

Using only the identity matrix part of the parity-check matrix the code is able to detect up to $2L - 1$ adjacent errors. This is because the linear combination of those identity columns

will never result in a zero column. However, if error correction is required, the number of detectable adjacent errors is reduced, because correctable errors should have unique syndromes. The size of the identity matrix L is defined as

$$L = \left\lceil \frac{yAED + xAEC + 1}{2} \right\rceil, \tag{B.6}$$

where $yAED$ is the number of adjacent errors to be detected, and $xAEC$ the number of adjacent errors to be corrected. The total number of check bits m can be calculated using the size of the identity matrices L and the requirement that all columns must be unique. The number of options available for each column the $\boldsymbol{H}_i$ sub-matrix, should be at least as large as the number of columns in $\boldsymbol{H}$ sharing the same column of the identity matrix. This requirement can also be written as

$$2^{m-L} \geq \left\lceil \frac{k+m}{L} \right\rceil. \tag{B.7}$$

The sub-matrices $\boldsymbol{H}_i$ are constructed from two column vectors, one for the odd numbered columns $\boldsymbol{h}_{i,o}$ and one for the even numbered columns $\boldsymbol{h}_{i,e}$. The sub-matrix $\boldsymbol{H}_i$ can be written as

$$\boldsymbol{H}_i = \begin{bmatrix} \boldsymbol{h}_{i,o} & \boldsymbol{h}_{i,e} & \boldsymbol{h}_{i,o} & \boldsymbol{h}_{i,e} & \cdots \end{bmatrix}. \tag{B.8}$$

The number of columns in this sub-matrix is determined by the value of L. To satisfy to constraint 2, every column should be distinct, the value of $\boldsymbol{h}_{i,o}$ for all i should be distinct and the value of $\boldsymbol{h}_{i,e}$ for all i should be distinct. If this is not the case, the final parity-check matrix will contain duplicate columns.

To satisfy constraint 4, extra conditions are needed on the values of $\boldsymbol{h}_{i,o}$ and $\boldsymbol{h}_{i,e}$. For the case that the code is a 2-bit adjacent error correcting code, the xor-result of two adjacent columns should be distinct. Therefore, the xor-result of $\boldsymbol{h}_{i,o}$ and $\boldsymbol{h}_{i,e}$ for every i should be distinct. Furthermore, the xor-result of the last column of $\boldsymbol{H}_i$ and the first column of $\boldsymbol{H}_{i+1}$ should be distinct. Depending on the value of L the last column of $\boldsymbol{H}_i$ can either be $\boldsymbol{h}_{i,o}$ or $\boldsymbol{h}_{i,e}$. Without these constraint the syndrome of a 2-bit adjacent error could overlap with the syndrome of a 2-bit adjacent error shifted by a multiple of L bits.

Similarly, for the case that the code is a 3-bit adjacent error correcting code, additional conditions are required. First of all, the xor-result of three adjacent columns within $\boldsymbol{H}_i$ should be distinct, fortunately this is already required by the conditions on $\boldsymbol{h}_{i,o}$ and $\boldsymbol{h}_{i,e}$ for constraint 2. However, for the case where the 3-bit adjacent error overlaps between two $\boldsymbol{H}_i$ matrices, additional constraints are required. One of these cases requires the xor-result of $\boldsymbol{h}_{i,o}$, $\boldsymbol{h}_{i,e}$ and $\boldsymbol{h}_{i+1,o}$ to be distinct. In the other case the xor-result of the last column of $\boldsymbol{H}_i$, $\boldsymbol{h}_{i+1,o}$ and $\boldsymbol{h}_{i+1,e}$ has to be distinct.

Using all these conditions a search algorithm can be constructed to find $\boldsymbol{h}_{i,o}$ and $\boldsymbol{h}_{i,o}$ for all i. In the paper [44] a two stage randomised search is suggested, however other search strategies are also possible. Example matrices were found for both the 2-bit and 3-bit adjacent error correction scenario, shown in fig. B.7 and fig. B.8 respectively.

$$\boldsymbol{H} = \left[\begin{array}{cccc|cccc|cccc|cccc|cccc|ccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}\right]$$

Figure B.7: Partiy-check matrix for (23,16)-SEC-pDED-DAEC-5AED code [45, Fig. 3]

$$\boldsymbol{H} = \left[\begin{array}{cccc|cccc|cccc|cccc|cccc|ccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}\right]$$

Figure B.8: Parity-check matrix for (23,16)-SEC-pDED-TAEC-4AED code (generated)

Appendix C

Variations on the refresh controller

One of the goals of this research is to build a framework which allows for easy experimentation with different error correction codes or memory controller designs. During the design of the refresh controller a number of slight variations were considered, however for the main part of this research only a single design was chosen. In this chapter a number of these variations are implemented after all, and they are evaluated using most of the same metrics as the main results. However, for these designs no full system ASIC layouts were made, as those take a lot of time.

C.1 Designs and motivation

The refresh controller design has been designs with a number of configuration options, which allow for small tweaks in the behaviour of the refresh controller. First of all, the refresh period of the controller is configurable. This allows the designer to make a trade-off between better error correction and less performance impact. Second, the refresh controller can be configured to forcefully refresh the memory exactly on the configured period. This makes sure that the refresh operations are actually happening, especially if the memory bus is very active. And third, the address generation of the refresh controller can be adjusted to allow for refreshing only a part of the complete memory.

One of the designs which will be tested is a refresh controller with the forced refresh enabled. The refresh period of this controller is 128 cycles, the same as the normal refresh controller designs, and the address generation traverses the complete memory. This design will be called `ForceRefreshController`.

Another designs, called `ContinuousRefreshController`, will be configured with a refresh period of zero cycles. The refresh will not be forced, and the address generation traverses the complete memory. This design will attempt to refresh the memory on every unused cycle, very rapidly refreshing the complete memory.

Finally there are two designs which make use of the configurable address generation. During the development of these memory controller, a large number of simulations which resulted in a crash were caused by errors in the programs stack memory. This is quite logical, as

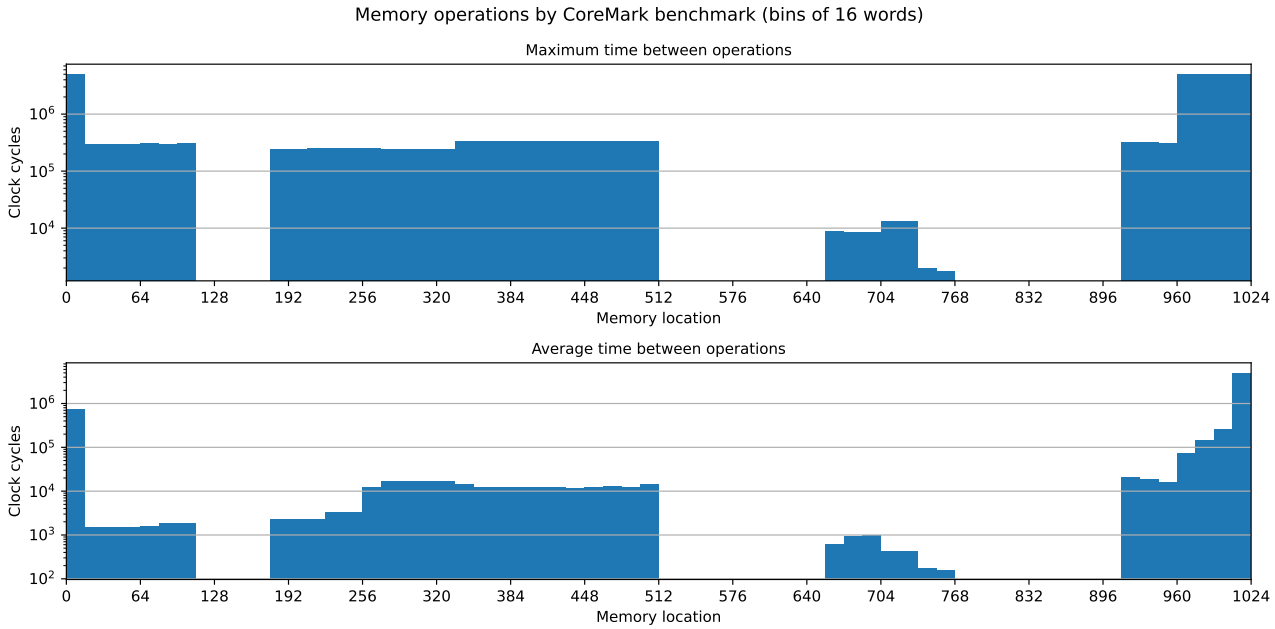


Figure C.1: Maximum and average time between memory operations in bins of 16 memory words.

the programs stack determines the return addresses for functions. If these get corrupted by memory errors, the program might jump into unrelated code, the data memory or even address space without any memory attached.

To quantify if the stack was indeed more vulnerable to errors, the time between memory operations for each memory location was measured. Figure C.1 shows both the maximum and average time between operations for bins of 16 memory locations. Note that the number of clock cycles between operations is shown in a logarithmic scale. This figure shows that the first 16 words and the last 64 words have a maximum number of clock cycles between operations of approximately 5 million cycles, which is basically the total runtime of the CoreMark benchmark.

Based on these results two designs were created. The first design, the `TopRefreshController`, only refreshes the top 128 words of memory, protecting the stack of the program. The second designs, the `TopBottomRefreshController`, protects the top and bottom 64 words of memory, protecting the memory location with large times between operations.

C.2 Effectivity and performance results

C.2.1 Effectivity results

Using the same simulation setup used in chapter 4, these designs are evaluated on their error correction effectiveness and performance impact. Figure C.2 shows the simulation results for

the different refresh controller variations. In this figure the original refresh controller is also included for reference.

The results for the `ForceRefreshController` shows very little difference with the original `RefreshController`. This is as expected, since the only difference between these controllers is whether they forcefully refresh the memory. In this specific benchmark the memory bus is not congested, therefore both refresh controller will refresh at basically the same moment, possibly a few cycles later for the original refresh controller. However, these results show that enabling this option does not negatively impact the error correction performance of the controller.

For the `TopRefreshController` and the `TopBottomRefreshController` the results are less positive. Both perform worse than the original refresh controller, while refreshing their specific regions eight times as much. This is result is likely due to the error still causing problems in the other data, that the program was unable to finish correctly. Interestingly, the `TopBottomRefreshController` does reduce the number of terminated simulations for this specific program significantly. This does seem to indicate that the control flow of the program is preserved better, while results still end up incorrect. These types of designs which only partially refresh the memory might be interesting in special cases where the important data is stored in a specific region.

Finally, the `ContinuousRefreshController` shows a shockingly good result, with practically 100% correct results for this specific range of error rates. As expected, refreshing all memory location at a very high rate does improve the error correction effectivity of this controller. For this specific benchmark the memory bus had quite a lot of available cycles, allowing this controller to refresh many memory locations quickly. However, when running a program that accesses the memory more frequently, the error correction effectiveness of this controller might be reduced. Furthermore, in a real world implementation this controller will likely increase the power usage of the design significantly, as the memory is now accessed on every single clock cycle.

Figure C.3 shows the results for these controllers when the simulation is also injection 2-bit adjacent errors with a 10% chance. As expected, these show poor performance on most of the error correction codes, with the adjacent error correction codes performing practically the same as without any adjacent errors.

C.2.2 Performance overhead results

Using the same method as in chapter 4, the number of additional cycles used by these memory controllers is computed. Figure C.4 shows the number of addition cycles for each of the memory controllers, both with only single errors and with 10% 2-bit adjacent errors.

This figure shows very interesting results for the `ForceRefreshController`. The number of additional cycles for this controller starts at $6\,063$, while the other controllers all start with no overhead. This is because this controller will interfere with regular memory operations to execute the refresh operations. With this controller refreshing every 128 clock cycles, and

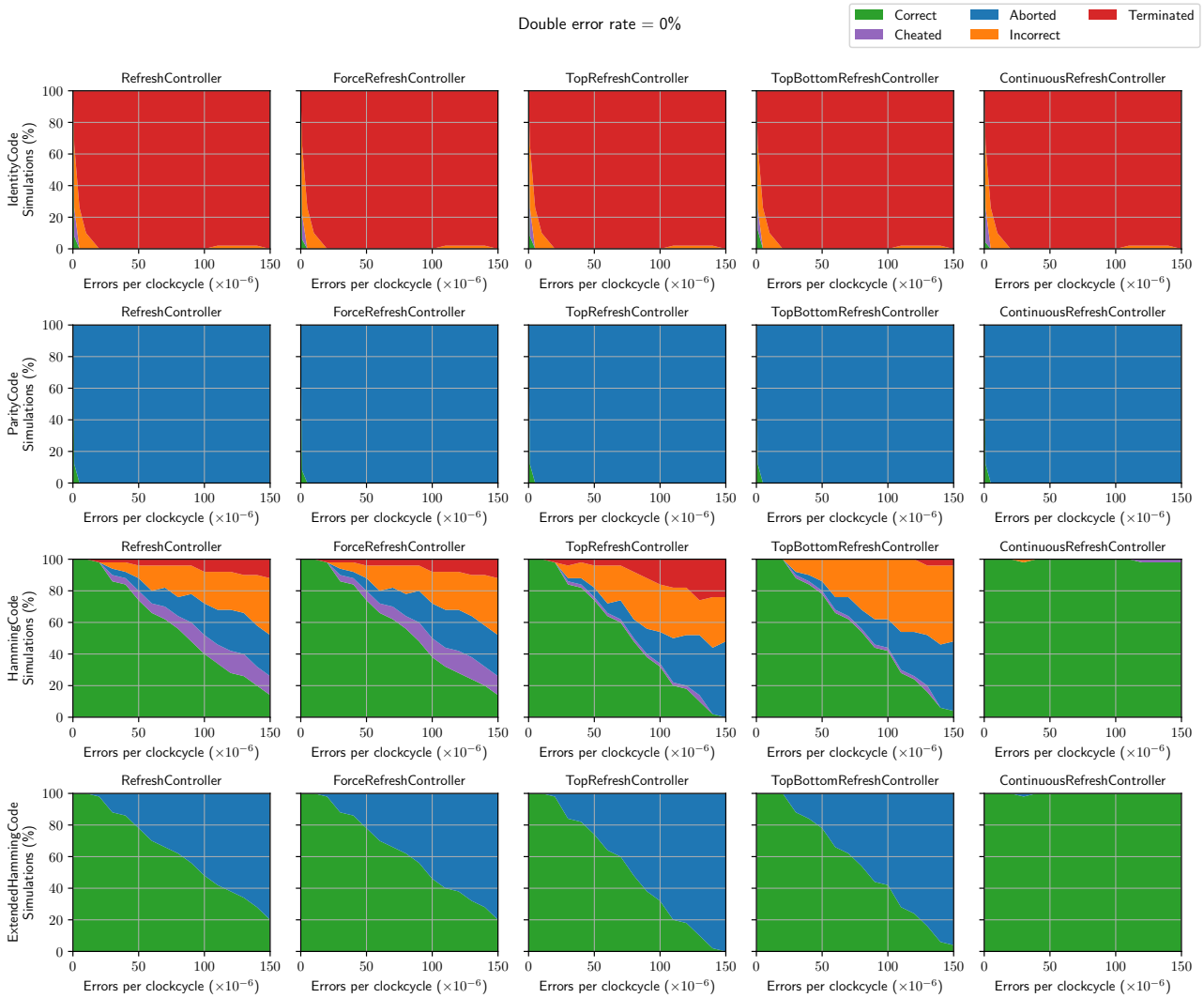


Figure C.2: Simulation results for the refresh controller variations.

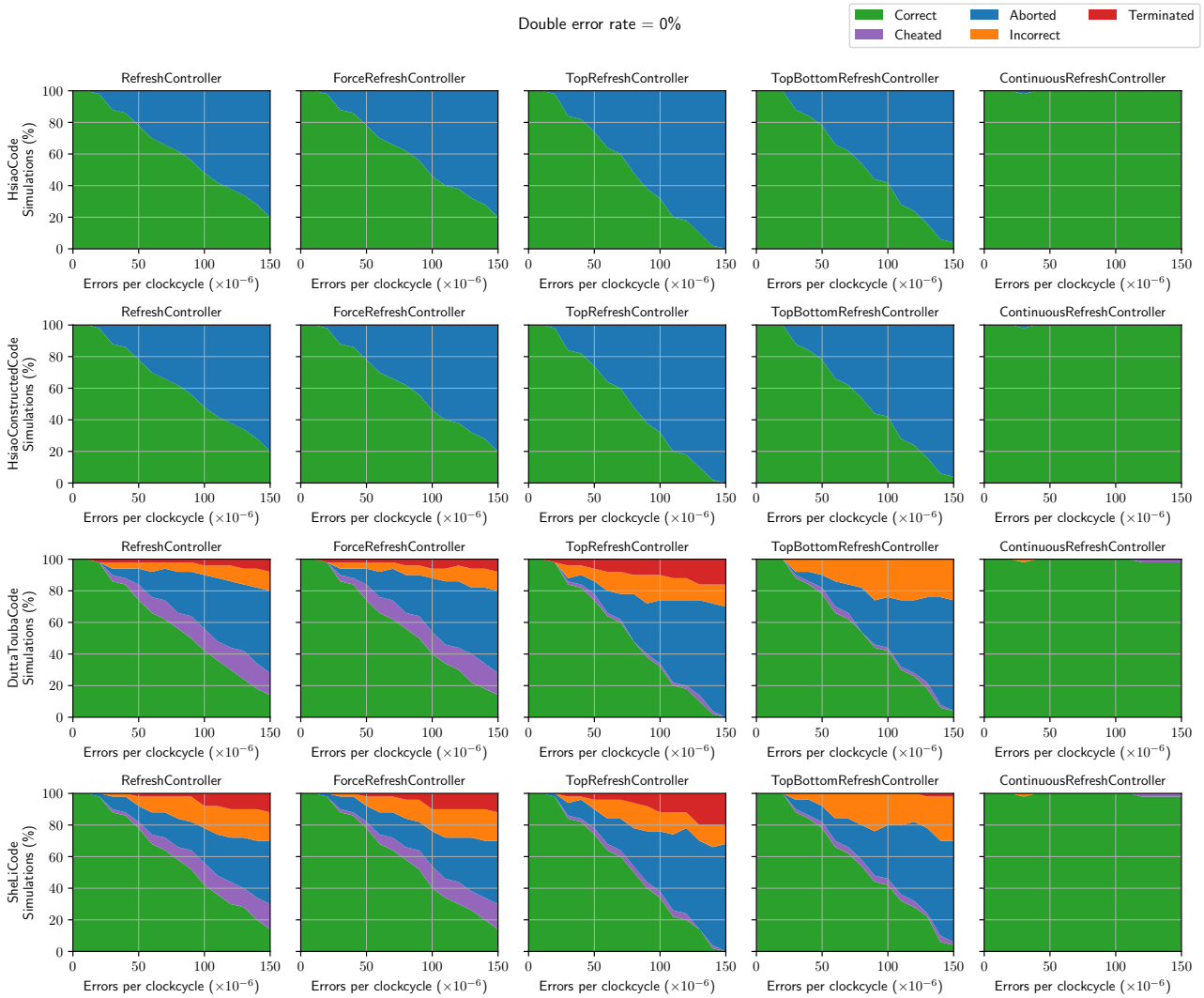


Figure C.2: Simulation results for the refresh controller variations. (Continued)

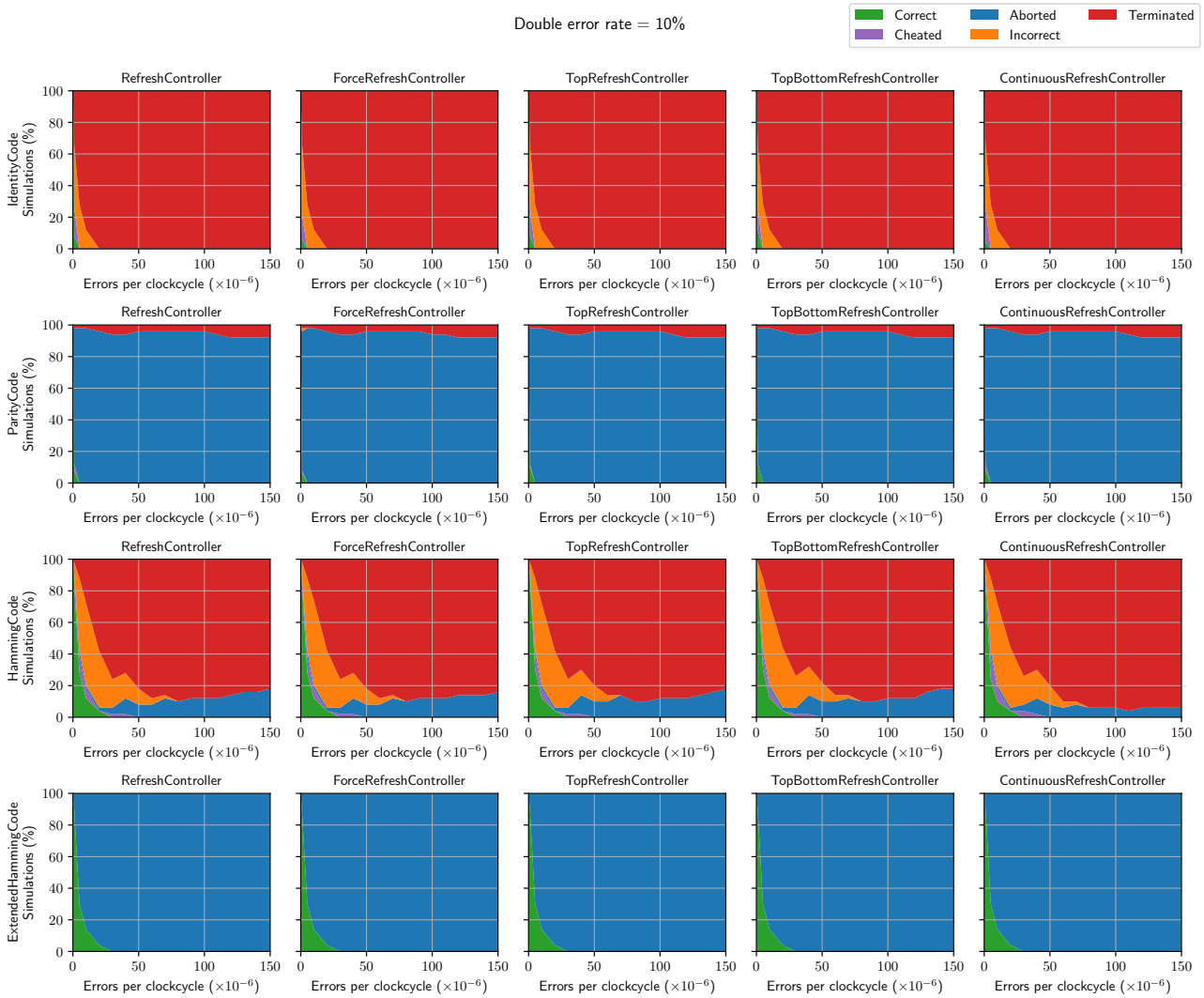


Figure C.3: Simulation results for the refresh controller variations, with 10% adjacent errors.

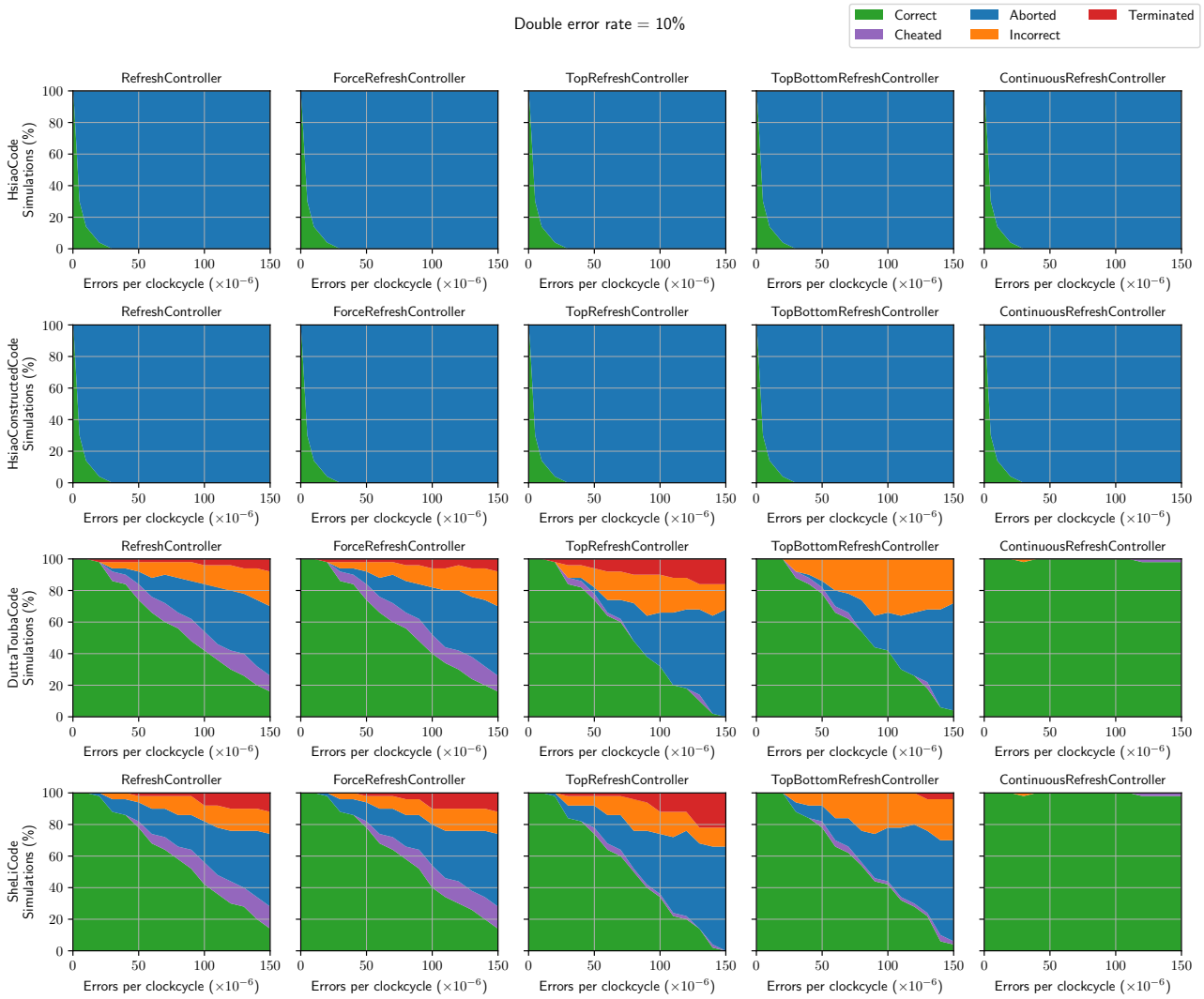
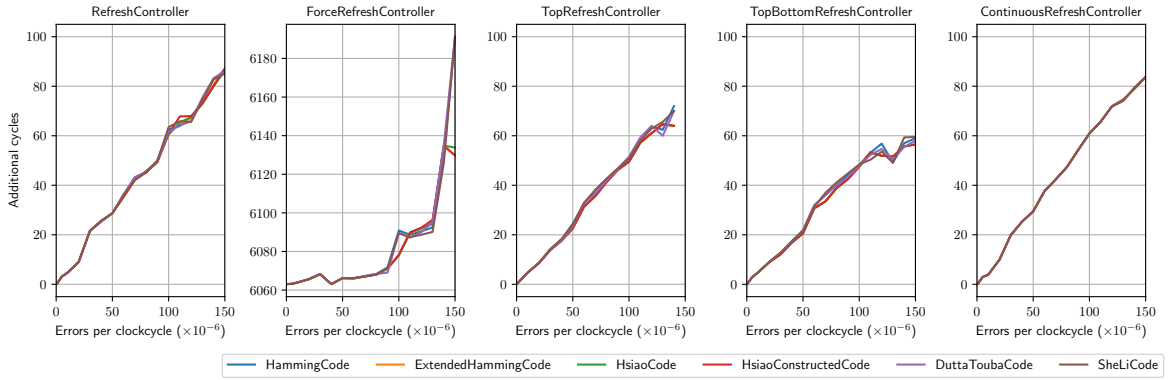
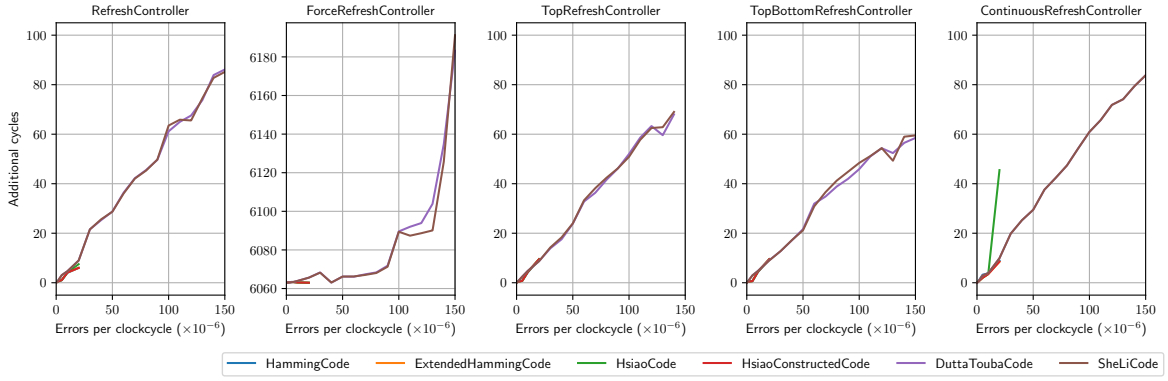


Figure C.3: Simulation results for the refresh controller variations, with 10% adjacent errors. (Continued)



(a) Only single errors



(b) With 10% adjacent 2-bit errors

Figure C.4: Clock cycle overhead of different refresh controller variations.

the program normally taking $5\,052\,828$ cycles, approximately $39\,475$ refresh operations are executed. So in this case only 15% of the refresh operations interfere with regular memory operations, however this percentage can change drastically with different programs.

Furthermore, this controller also has an increased overhead with an increased error rate due to the internal write-back controller. However, this controller does not follow the normal linear increase in additional cycles with the increased error rate. Instead, these additional cycles follow a much more curved line. The likely cause of this difference is that the additional delay of write-back controller desynchronizes the program from the refresh interval, reducing the chance that repetitive memory accesses hit the refresh cycle every time.

The `TopRefreshController` and `TopBottomRefreshController` both show a slightly lower additional number of cycles than the regular refresh controller. This reduction is due to the lower number of total errors hitting a region that is refreshed. However, this reduction is not a factor of eight, as might be expected since the region is eight times smaller. Instead, a large number of these additional cycles are caused by the internal write-back controller which is correcting the non-refreshed regions as they are accessed by the processor.

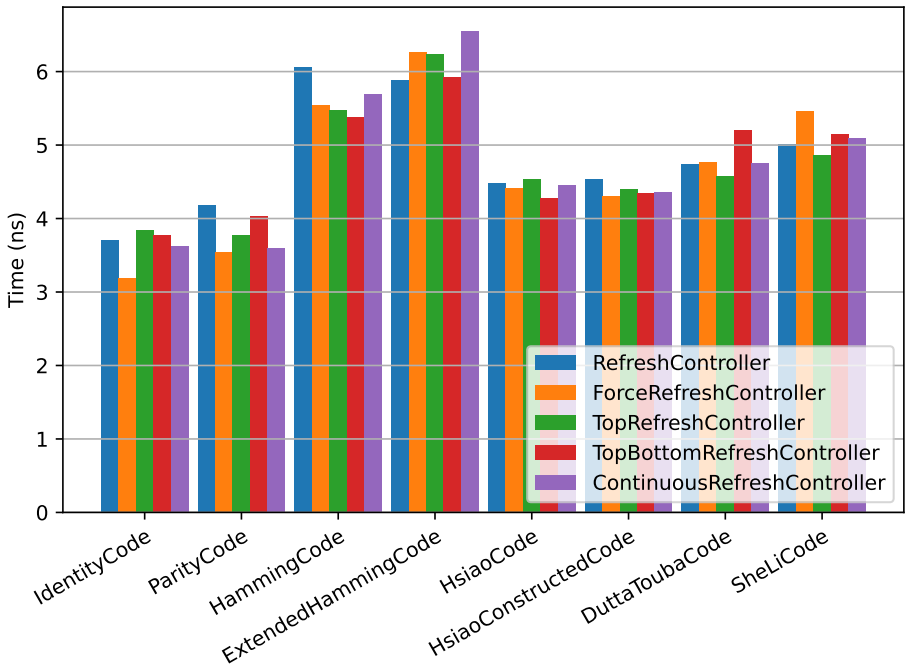


Figure C.5: Critical path delay for refresh controller variations.

C.3 ASIC layout results for error correcting memory controllers

In the same way as in the previous chapter, ASIC layouts of just these controller designs are created and evaluated. OpenLane is configured using the same configuration as specified in listing 4.1.

C.3.1 Critical path delay

The critical path delay for the ASIC layouts of these different refresh controller variations are shown in fig. C.5. These results do not show significant differences in the critical path delay for the variations of the refresh controller. Furthermore, the same trends per error correction code are visible as shown in fig. 4.4.

C.3.2 Cell and chip area

Figures C.6 and C.7 show the standard cell and chip area for these refresh controller variations respectively. Again, all of these results do no show significant difference between the different refresh controller variations. Furthermore, here also the same trends per error correction code are visible as shown in figs. 4.5 and 4.6.

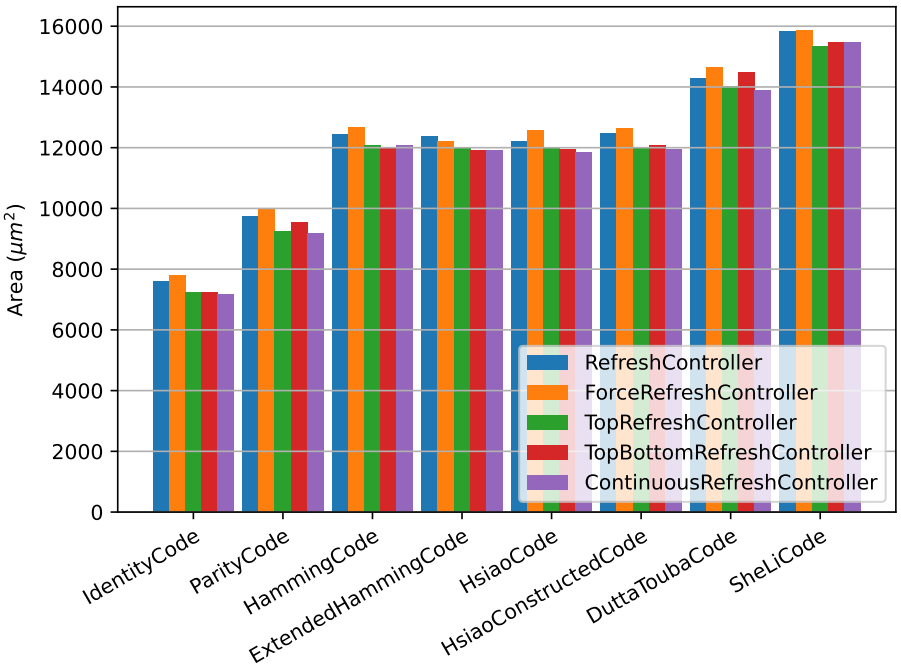


Figure C.6: Standard cell area for refresh controller variations.

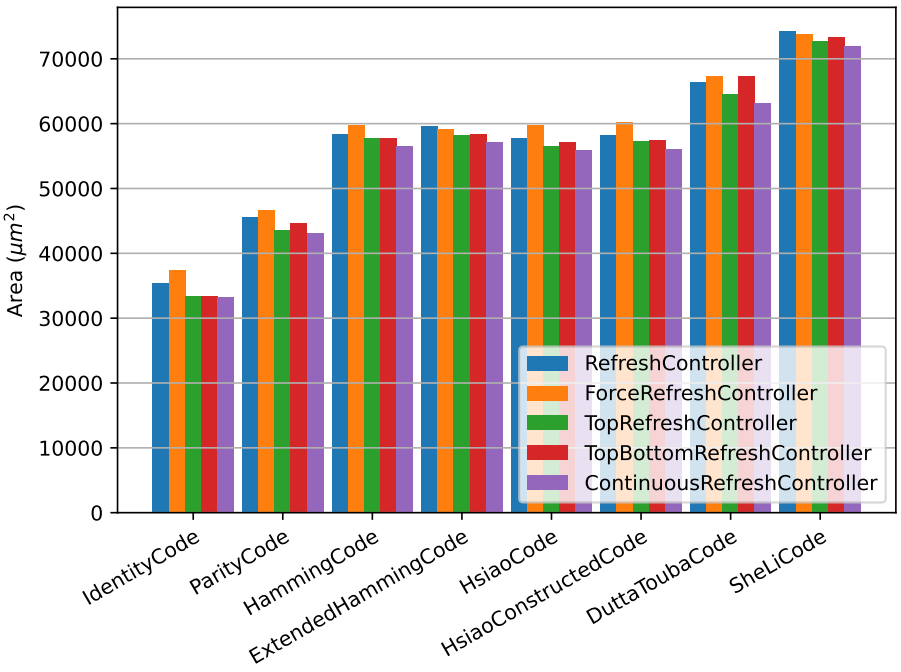


Figure C.7: Chip area for refresh controller variations.

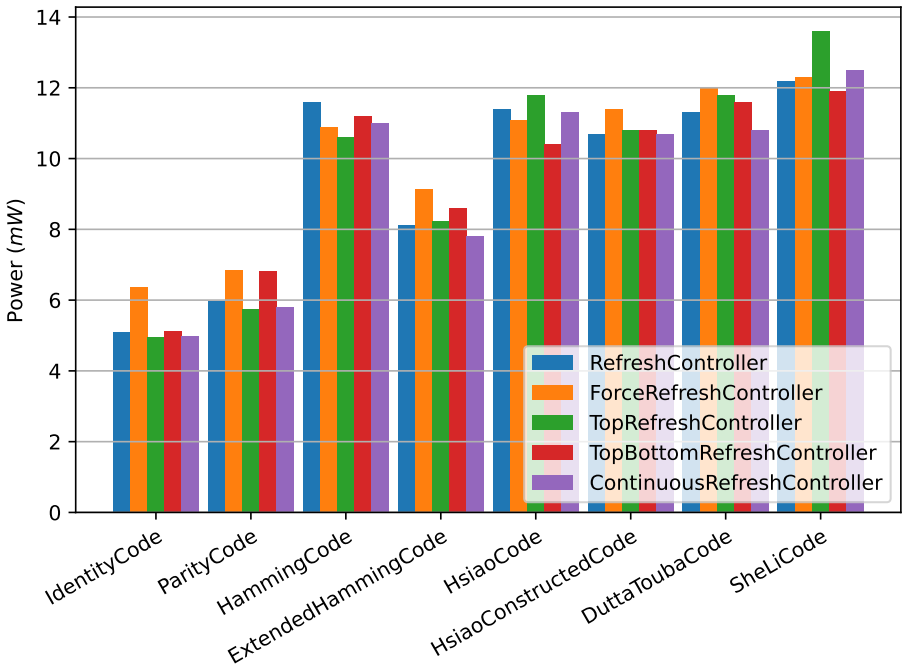


Figure C.8: Power usage for refresh controller variations.

C.3.3 Power usage

The power usage of these refresh controller variations are shown in fig. C.8. All of these results do not show any significant difference between the different refresh controller variations. Unfortunately, the power estimation tools are not able to determine increased activity for the `ContinuousRefreshController`. A large portion of this additional power will also be used in the memory blocks, which are not modelled here.

C.4 Conclusions

This chapter has shown some variations on the refresh controller, which can be created using the memory controller generation tool created for this research. All of these variations have been evaluated using the same methods as used in the previous chapter, with the exception of full system ASIC layouts.

The `ForceRefreshController` design has shown similar error correction effectiveness as the original refresh controller, however it has a larger performance overhead on the running program. This designs can be useful in specific scenarios where the memory bus is very busy or when the refresh should happen exactly at the same moment every time.

The `TopRefreshController` and `TopBottomRefreshController` designs have shown worse

error correction effectiveness for this specific benchmark program. However, they show off the ability for the refresh controller to be configured with only specific refresh areas, instead of the complete memory. If the running software is designed around these smaller refresh regions, it might improve the overall success rate of the program, while only refreshing a small region of memory.

Finally, the `ContinuousRefreshController` has shown great error correction effectiveness, by refreshing as often as possible. While this design performs great in simulation there might be some concerns implementing it in real hardware, as it will increase the memory activity drastically. However, in cases where this increase memory activity does not matter, but the best error correction effectivity is required, this designs might be useful.

In the end, these designs might not be directly useful for any specific design, however this chapter has shown the versatility of the tools created in this research. While only editing a handful lines of code, many different and interesting memory controller designs can be explored. And using the error injecting simulator allows for easy comparison against other designs.

Appendix D

Code of implementations

The code for all of the different tools created in this thesis can be found on GitHub. The code is split up into three separate repositories, containing the memory controller generator, the RISC-V processor used in the test system and the test system itself. Each of these repositories are linked with the exact final commit used in the creation of this thesis.

Memory controller generator
`https://github.com/msvisser/memory-controller-generator/tree/2ffd83095fc125988d7ed20131cbb53d9ed01bbe`

RISC-V processor and peripherals
`https://github.com/msvisser/riscv-tilelink/tree/7a02d6163bd4971b7bde1991d608606e36290be8`

Complete test system and simulator
`https://github.com/msvisser/tilelink-ecc-top/tree/a9853f1a11e4fc33caa56d4be9ec2b3faf47431d`