2016

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

Technolution B.V.
Burgemeester Jamessingel 1
2803 WV Gouda
The Netherlands
http://www.technolution.eu/

MSc THESIS

A fault tolerant memory architecture for a RISC-V softcore

Adam Alexander Verhage

Abstract

Technolution B.V. is developing a custom Reduced Instruction Set Computer (RISC)-V based softcore for implementation on a Field Programmable Gate Array (FPGA). Previously, the softcore used the memory residing on the FPGA only, which is very limited in capacity and limits scaling. To solve this problem, a connection is made from the softcore to the Dynamic RAM (DRAM) interface of the FPGA applied. A cache structure is added to mitigate the performance impact. It makes efficient use of the limited, yet fast memory on the FPGA.
This thesis report describes the prior art, design considerations, implementation details and results for the realized cache structure and the connection to DRAM. The cache structure is the main topic of this thesis, which has been made as fast and efficient as possible, fault tolerant and configurable in capacity. It consists out of a data cache and an instruction cache with slight different parameters. For fault tolerance, both a light and heavy protection scheme are discussed and proposed. Only the light protection scheme is currently implemented and verified. The capacity is configurable as 4, 8, 16 or 32 kB for both types of cache. The connection to DRAM on the other hand makes use of the widely accepted standard of the ARM^{TM}Advanced eXtensible Interface (AXI) bus.

CE-MS-2016-16

Technolution B.V.

Faculty of Electrical Engineering, Mathematics and Computer Science

A fault tolerant memory architecture for a RISC-V softcore

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Adam Alexander Verhage
born in Capelle aan den IJssel, Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A fault tolerant memory architecture for a RISC-V softcore

by Adam Alexander Verhage

Abstract

Technolution B.V. is developing a custom Reduced Instruction Set Computer (RISC)-V based softcore for implementation on a Field Programmable Gate Array (FPGA). Previously, the softcore used the memory residing on the FPGA only, which is very limited in capacity and limits scaling. To solve this problem, a connection is made from the softcore to the Dynamic RAM (DRAM) interface of the FPGA applied. A cache structure is added to mitigate the performance impact. It makes efficient use of the limited, yet fast memory on the FPGA.

This thesis report describes the prior art, design considerations, implementation details and results for the realized cache structure and the connection to DRAM. The cache structure is the main topic of this thesis, which has been made as fast and efficient as possible, fault tolerant and configurable in capacity. It consists out of a data cache and an instruction cache with slight different parameters. For fault tolerance, both a light and heavy protection scheme are discussed and proposed. Only the light protection scheme is currently implemented and verified. The capacity is configurable as 4, 8, 16 or 32 kB for both types of cache. The connection to DRAM on the other hand makes use of the widely accepted standard of the ARM^{TM}Advanced eXtensible Interface (AXI) bus.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2016-16

Committee Members :

Advisor:	Arjan van Genderen, CE, TU Delft
Chairperson:	Stephan Wong, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft
Member:	Frank van Eijkelenburg, Technolution B.V.

Dedicated to my wife Madeleine, and to my parents

Contents

List of Figures

List of Tables

List of Acronyms

5-ED Quintuple Error Detection

ABM AHB Bus Matrix

AHB Advanced High-performance Bus

AMAT Average Memory Access Time

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BCH Bose-Chaudhuri-Hocquenghem

BFM Bus Functional Model

CA Command-Address

CAM Content Adressable Memory

CRC Cyclic Redundancy Check

DBI Data-Bus Inversion

DEC Double Error Correcting

DEC/TED Double Error Correcting/Triple Error Detecting

DED Double Error Detecting

DEDS Double Error Detection Signal

DDR Double Data Rate

DMR Double Modular Redundancy

DRAM Dynamic RAM

EDAC Error Detection And Correction

EDC Error Detecting Code

ECC Error Correcting Code

FF Flip-Flop

FIFO First In First Out

FIT	Failure In Time
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HPMS	High Performance Memory Subsystem
IP	Intellectual Property
ISA	Instruction Set Architecture
L1	Level 1
L2	Level 2
LLM	Lower Level Memory
LPDDR	Low-Power DDR
LSB	Least Significant Bit
LRU	Least Recently Used
LUT	Look-Up Table
MBU	Multiple Bit Upset
MCB	Memory Controller Block
MFU	Most Frequently Used
MTTF	Mean Time To Failure
NMR	N-way Modular Redundancy
PCM	Parity Check Matrix
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RMW	Read-Modify-Write
RS	Reed-Salomon
SBU	Single Bit Upset
SBC/DBD	Single Byte error Correcting/Dual Byte error Detecting
SDRAM	Synchronous DRAM
SEC	Single Error Correcting
SEC/DED	Single Error Correcting/Double Error Detecting

SEC/DED/SBD Single Error Correcting/Double Error Detecting/Single Byte error Detecting

SED Single Error Detection

SEE Single Event Effect

SEFI Single Event Functional Interrupt

SEL Single Event Latchup

SEMU Single Event Multiple bit Upset

SET Single Event Transient

SEU Single Event Upset

SoC System on Chip

SNC/DND Single Nibble error Correcting/Double Nibble error Detecting

SRAM Static RAM

TED Triple Error Detecting

TMR Triple Modular Redundancy

UART Universal Asynchronous Receiver/Transmitter

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

WT write-through

List of definitions

Codec Logic to encode and/or decode a certain EDAC scheme.

Fault tolerance The ability to let faults not become errors in the execution flow.

Softcore Processor described in a Hardware Description Language (HDL) which can be mapped either to an FPGA or ASIC.

Acknowledgements

While I am writing this, it is exactly one year ago I started working on this thesis project at Technolution B.V.. During this year, several people guided me through the process. To those, and some others, I want to express my gratitude.

First, I want to thank Technolution for offering the possibility to do my MSc thesis project within their company. They provided me of an interesting assignment, including some freedom to adjust the project according to my own interests. During this project, I did not only learn a lot in a technical perspective, but also how it is like to be part of a company. Both experiences are expected to be very useful in my future career.

Second, I want to express my gratitude more specifically to several people I met within Technolution. First of all, I need to mention Frank van Eijkelenburg as my daily mentor. He helped me by sharing his experience, mainly concerning the VHDL environment. Additionally, he often motivated me and guided me into the next step. I also thank Jonathan Hofman, being the project supervisor. He inspired me to do what I have done within the project and provided me with useful insights and clear inputs to improve the quality of my work. Third, I also want to thank Wietse Heida as a fellow student for his input and introductory help in the first 8 months.

Third in this acknowledgment list, I want to name, and express my gratitude to, some people at the TU Delft. Of course, this mainly concerns my advisor at the TU Delft: Arjan van Genderen. He guided me through the formal procedures of the master thesis, but also provided me with valuable feedback, especially at the end concerning this thesis report. Furthermore, I want to thank Stephan Wong for chairing the thesis committee, and Rene van Leuken for also taking part.

A special thanks goes to my parents, without them I would never have come to the end of this master studies. They supported me from the beginning of my life in any possible way and still do. Also my parents-in-law are very supportive for which I am grateful.

Last but not least, I want to thank my fantastic wife, Madeleine. There are no words for describing what she means to me in every way.

Adam Alexander Verhage
Gouda, The Netherlands
December 7, 2016

1 Introduction

This chapter introduces the main topics of the thesis. First, the problem will be explained briefly. Second, the project goals will be addressed. The chapter ends with an outline for the rest of the thesis.

1.1 Problem statement

Technolution B.V., hereafter called the employer, is constructing a softcore based on the RISC-V Instruction Set Architecture (ISA) [15]. Thanks to the open and license-free ISA the employer is able to define its own proprietary softcore to use in various projects, modifying and improving the core at any point when needed. The softcore will be deployed in a Field Programmable Gate Array (FPGA) since the quantities needed are small, i.e. an Application-Specific Integrated Circuit (ASIC) would be too expensive.

The problem of the current softcore implementation is that the storage of both data and instructions, needed to run software, is integrated as a part of the softcore. Since it is deployed on an FPGA, which includes limited internal storage, this architecture demands considerable resources for the built-in memories. Also, this scheme limits the practical use and extendibility.

The first step to a more mature memory architecture is to connect the softcore to an external memory device and use that as the main memory instead of the built-in memories. In this thesis this will be Double Data Rate (DDR)3 Dynamic RAM (DRAM). The FPGA to use features an integrated memory controller to interface the DRAM which can be interfaced by FPGA logic, as is described more concisely in e.g. Section 2.5.1.2. Nevertheless, accesses to DRAM incur considerable latency compared with those to the integrated memory. To improve the performance of the softcore while using the DRAM as the main memory device, a cache structure is added. The cache stores recently used data and instructions in the quickly accessible integrated memory while keeping the capacity independent on the total memory needed. An additional constraint added by the employer is a hit time of one clock cycle while running the same frequency as the softcore without the cache structure (which is 62.5 MHz).

Employers softcore is aimed towards achieving fault tolerance for several situations. Introducing fault tolerance in a softcore means it becomes able to withstand a certain amount of faults without letting them become errors. Hence, this means that also the cache has to be made fault tolerant, just like the core is made fault tolerant in a thesis project partly adjacent to this thesis [8]. Ideally, the level of this fault tolerance is configurable in order to provide the right level of fault tolerance for several different applications. Since no particular application is defined yet and to make the softcore more versatile, the cache should have a configurable capacity.

As mentioned before, the interconnection to the Lower Level Memory (LLM), being the DRAM, was not implemented at the start of this thesis project. Implementation of this, including the interconnecting bus, is also part of this project, as well as modifying or creating a test method for verification. Options to add fault tolerance for the DRAM and its interface are explored, but not implemented. This is because the DRAM controller itself is out of the scope of this thesis project. Hence the DRAM controller is a standard component in the targeted FPGA.

1.2 Project goals

The primary goal of the project described in this thesis is to design and implement the fault tolerant cache structure. The secondary goal is to connect the softcore through the cache to the integrated DRAM controller. Together they solve the problems stated in Section 1.1. Therefore the research goals and belonging sub-goals are:

1. Implement a fault tolerant and configurable cache structure for the RISC-V softcore:
   (a) Implement an efficient cache structure with one clock cycle hit time,
   (b) Make the cache capacities configurable,
   (c) Make the cache memory fault tolerant without loosing the configurability.
2. Connect the softcore to the integrated DRAM controller:
   (a) Implement a suitable interface onto the cache memory,
   (b) Obtain the necessary resources for the DRAM controller and interconnection,
   (c) Connect the existing softcore to the cache structure and DRAM controller.

Note that, as with every new implementation, all parts have to be tested and verified individually. Also, during the process knowledge is gathered which is not directly used, this will be documented as future work. These two additional objectives are also elaborated in this thesis report.

The goals mentioned are confined by regular feedback from the employer during the entire project. Some initial goals, not mentioned here, were omitted due to the non-permitting time limit and delays during implementation and verification. Note that these omitted goals can be seen as future work (see Section 6.2 for a more extensive description) and are also taken into considerations in the literature survey (Chapter 2). These (sub)goals are:

- Configurability in fault tolerance: add a higher level of fault tolerance and make it selectable;
- Multi-level cache.

1.3 Outline

Chapter 2 describes the prior art in cache memories, fault tolerance and other related subjects. Secondly, in Chapter 3 the intended design will be discussed in detail by elaborating on the requirements set and applying the knowledge gathered in Chapter 2. Consequently in Chapter 4 the implementation details of the two main goals will be discussed. Following Chapter 5 elaborates on the verification performed on the implemented work and presents the results of tests performed to measure performance and other relevant metrics. Last part, found in Chapter 6, processes the findings from Chapter 5 and proposes future work.

Background: literature survey 2

This chapter summarizes the relevant information found during the literature survey performed for this thesis. It starts however with a more extended project description including the detailed requirements, which lead to the searching criteria and a categorization of the literature found. Based on this, a framework for the remaining parts of the chapter is formed and used to discuss the found literature in a logical order.

2.1 Project description

The project as briefly described in Chapter 1 is about designing and implementing a fault tolerant memory architecture meant for application in a specific softcore microprocessor (called softcore hereafter). This includes the parameterizing and design of a cache structure as the most important factors to ultimately speed up the execution of the softcore. The design will be implemented in VHSIC Hardware Description Language (VHDL) targeting a specific Field Programmable Gate Array (FPGA), while keeping the design as less vendor specific as possible.

Two starting points are defined beforehand: First, any optimization possible should be done towards a minimum of occupied resources i.e. area, as requested by employer. This is taken into account in the quest of relevant papers. Second, the memory architecture will ultimately reside on a flash-based FPGA, namely the IGLOO2 family from Microsemi [16]. Therefore, this (kind of) FPGA will be taken as reference while e.g. exploring the possibilities for fault tolerance.

2.1.1 Categorization

The remaining part of the chapter will go through the literature investigated by the author, all related to one or more of the sub-goals set in Section 1.2 and the definitions in Section 2.1. The author has chosen another categorization than imposed by the goals to divide the literature survey in segments. The following topics will be discussed:

- **Section 2.2: Parameterizing the cache**
  Since one of the main goals of the project is aimed towards an efficient cache architecture, the first topic to discuss is the classical parameterizing of a cache structure. This part will differ from the other parts in the sense that the knowledge discussed here is widely available and common sense to memory architecture designers. Therefore this part mainly resides on one source only, namely the famous book of J. L. Hennessey and D. A. Patterson named "Computer Architecture: A Quantitative Approach"[1].

- **Section 2.3: Fault tolerance - background**
  Secondly the topic of fault tolerance will be introduced focusing on application in memory architectures. Commonly used techniques will be explained and discussed in more detail using the various research papers found. In this section it will become clear what kind of protection and which commonly used technique is suited for the application given.
- **Section 2.4: Fault tolerance - techniques**
  In the third part a more in-depth discussion will arise about more advanced and/or new techniques to detect and/or correct soft errors. Also for this part it holds that most of the techniques are aimed at memory infrastructures, or more specifically the cache structure, while some others are applicable in a broader context.
- **Section 2.5: FPGAs - cache memories and fault tolerance**
  Since the memory architecture will reside on an FPGA, the fourth part will discuss the particular research found on FPGA architectures. Topics included are the parameterizing of cache memories in FPGAs, fault tolerance specific to FPGAs and other memory architecture related techniques which are only applicable for an FPGA.
- **Section 2.6: Verification, Dynamic RAM (DRAM) and buses/arbiters**
  The second main goal, being the realization of the interface to the DRAM memory controller, turns out to be standardized and therefore does not have to be discussed as extensively as the cache memory and fault tolerance. This topic is combined with the closely related topic of the interconnection, which is realized using a bus structure and its arbitrating logic.
- **Section 2.7: Verification**
  To test the working and verify the fault tolerance, verification methods are needed. Simulation of the VHDL design is not discussed since this includes basic knowledge. Instead, methods are explored to verify the working while running on the FPGA. Only one method is found which will be discussed briefly.
- **Section 2.8: Reduced Instruction Set Computer (RISC)-V softcore**
  Seventh part will conclude the main part of the literature study and is aimed for gaining insight leading to designing and implementing the cache structure in such a way it will fit the current softcore. It will provide just a brief overview of the status of the RISC-V core of Technolution B.V. at the moment this thesis project is active.

At the end of the chapter, in Section 2.9, the most relevant information found is summarized and translated to the current project, which forms the basis for the design discussed in Chapter 3.

2.2 Parameterizing a cache structure

This section discusses the general design decisions for a memory architecture, focusing on the cache structure. For every parameter a short explanation is added and for several parameters the impact of that parameter on e.g. the performance is reviewed. All information covered in this section can also be found in [1], unless stated otherwise. According to the definitions in Chapter 1 it is assumed that the Lower Level Memory (LLM) consists of Double Data Rate (DDR)3 Synchronous DRAM (SDRAM) which will be referenced as Dynamic RAM (DRAM) for short. The cache will be split in an instruction and data cache (see **??**) and will consist of one level.

2.2.1 Performance metrics: caches

To understand the different optimization issues faced during parameterizing the cache, it is important to understand some of the performance metrics relevant to cache structures. Therefore the most important metrics are explained here.

2.2.1.1 Hit time

The hit time resembles the time needed to return the requested data when it is already loaded into the cache (i.e. a cache hit). When the cache consists of multiple levels (see Section 2.2.2.2), hit times generally differ between them. The hit time is assumed to be constant for a given system and may be used as a performance metric. Therefore it may be presented as a design requirement, like in the current project where the hit time of the cache should not exceed one clock cycle. Hit time is often referred to as access time.

2.2.1.2 Hit-/miss rate and categorization of misses

The hit and miss rate are fractions indicating how many accesses to the cache incur a hit or miss respectively. The term miss rate is used most frequently, but both terms are used in different literature. The following relation holds:

$$\text{hit rate} = 1 - \text{miss rate}$$

Misses can be categorized as follows:

- **Compulsory** misses are unavoidable as they occur at the first access. Any part of the cache (typically a predefined quantity of bits called a block) which is not written before will always result in a (compulsary) miss since it cannot return any data.

- **Capacity** misses occur when the cache cannot contain all blocks needed for the running program. In this case it has to replace non-empty blocks of the cache which may contain data useful in a later stage of the program. When the evicted blocks are retrieved later on the latter misses are called capacity misses.

- **Conflict** misses form the last category. This one relates to the associativity (see Section 2.2.2.2): when the cache is not fully associative, blocks may be discarded and retrieved later on due to the mapping of multiple blocks to the same set.

This categorization will be used to indicate which kind of misses can be reduced by certain parameterizations

2.2.1.3 Miss penalty

The miss penalty is the time needed to retrieve a block which is not yet present in the cache (i.e. a miss). Since a single level cache structure is used, the data needs to be fetched from DRAM directly. The miss penalty for this cache therefore is relatively large and is probably not constant, as opposed to the hit time. For simplicity however, the miss penalty could be considered constant, like in [1]. This constant number could be represented by the average time needed to fetch the requested block from DRAM.

2.2.1.4 Average Memory Access Time (AMAT)

AMAT is used in [1] to optimize for, being the most representative metric for realistic cache performance. It combines the previously mentioned metrics which may be misleading when considered on their own. AMAT is defined as follows:

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \cdot \text{Miss penalty}$$

In [1] it was described extensively in which ways to optimize for this metric, by means of reducing the hit time, miss rate and/or miss penalty. Most of these optimizations also apply for this project and therefore the sections describing them, found in Appendix C of [1], can be taken as a guideline to improve cache performance for a given design.

2.2.2 Parameters

The following part will go through a brief description of all parameters relevant to the current project:

- Capacity (Section 2.2.2.1),
- Associativity (Section 2.2.2.2),
- Block size (Section 2.2.2.3),
- Replacement policy (Section 2.2.2.4),
- Write miss policy (Section 2.2.2.5),
- Identification of data (Section 2.2.2.6),
- Multi-level cache structure (Section 2.2.2.7).

2.2.2.1 Capacity

First parameter to discuss is the capacity (i.e. size) of the cache. When choosing a cache too small, the miss rate will increase due to capacity misses, while an over-sized cache may suffer in terms of hit time (see Section 2.2.1.2) and takes considerable resources. Note that the access time to DRAM is order 10 to 100 bigger than the access time to an average cache structure (in ASICs). This means that the capacity of the cache structure has a large effect on system performance.

Another reason for carefully sizing the cache is called cache thrashing. Thrashing occurs when the upper-level memory (i.e. the cache) is much smaller than what is needed for a program. This will result in a cache moving data (due to misses) from and to the DRAM for a large amount of time, therefore not utilizing the data before it is replaced again. For instance, in a loop which needs an amount of instructions two times bigger than fits in the cache, every instance will be replaced before being used again, supposing consecutive instructions being used.

In general, it is hard to predict whether a certain cache capacity is adequate or most efficient. Therefore optimizing for this metric ideally is done by simulating the processor executing a representative benchmark suite using different cache capacities.

2.2.2.2 Associativity

The associativity of a cache is another parameter to determine during the design phase. It includes the following options:

- Direct mapped (1-way),
- Set-associative (n-way),
- Fully associative.

In short, the associativity indicates how much locations every piece of data (block) can go within the cache. The space in which one block may reside, is called a set.

For direct mapped or 1-way cache, every block has only one possible position in the cache, i.e. the set is as large as the block size. This means that another block mapped to the same set directly shifts out the previous one, regardless of any condition e.g. the block was written very recently. For set-associative or n-way cache, all blocks can go in n positions, which means that during a read n positions have to be checked for the requested block. In caches, most common configurations are 2-, 4- or 8-way set associative caches.

The last category, i.e. fully associative, just holds one set. Hence the whole cache space needs to be checked when searching a certain block, therefore this is only applied in (very) small caches otherwise the access time (and hence hit time) would explode. Solutions exist for this problem, however they are not efficiently applicable to an Field Programmable Gate Array (FPGA). When the associativity is bigger than one, several replacement policies can be applied as further discussed in Section 2.2.2.4.

One of the metrics directly influenced by the associativity is the access time. For an Application-Specific Integrated Circuit (ASIC) a tool exists to estimate the access time for given parameters (called CACTI [17]). Figure 2.1 from [1] indicates the related trade-offs for cache design in an ASIC.

2.2.2.3 Block size

Block size, also referenced by page- or line size, is the predefined smallest amount of data to be processed at once. The cache typically does not write nor read smaller quantities than the block size.As a minimal block size for an efficient cache, the word size used by

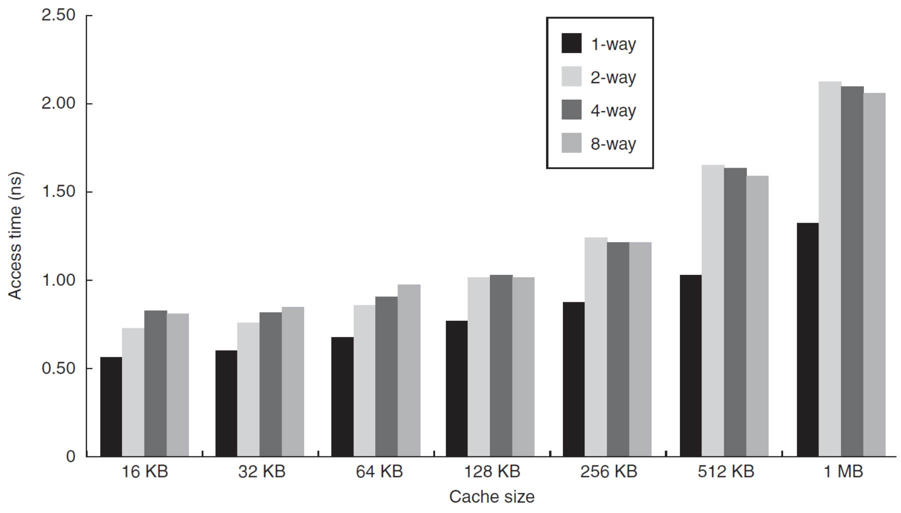


Figure 2.1: Cache access times (in a specific ASIC) for several capacities (cache size) and associativities, taken from [1] (p.294).

the processor or softcore should be considered, which is 32 bits for this project. However, the cache performance e.g. measured by AMAT may improve further when the block size is increased. One aspect to consider is the overhead of storage needed to store (part of) the address of a block, called the tag. The tag increases when the block size decreases and hence more storage overhead is inferred at equal cache capacity. Also, since the total amount of blocks residing in the system increases, the address space needs to be extended, which incurs another overhead in decoding logic.

On the other hand, when the block size increases to more than the word size, the cache takes more data than requested by the processor, e.g. it stores four words of 32 bits if the block size is 128 bits while returning just one word (32 bits). Although this extra data may be beneficial when data locality is employed, in many situations parts of the fetched data are not needed, i.e. needless data is loaded into the cache structure.

It may become clear by now that the block size is a parameter which has to be optimized, ideally even for specific programs and varying combinations with other parameters e.g. the capacity. For further reference, in Figure 2.2 the miss rate is plotted against the block size for different cache sizes (in ASIC). The figure shows how the block size can be optimized for a given program and architecture.

2.2.2.4 Replacement policy

A few options arise for the replacement policy, dictating which block needs to shift out when a new block is coming in and the set is filled with other data. Note that these options only arise when the cache is fully or set-associative (see Section 2.2.2.2). The options are:

- Least Recently Used (LRU),
- Random,

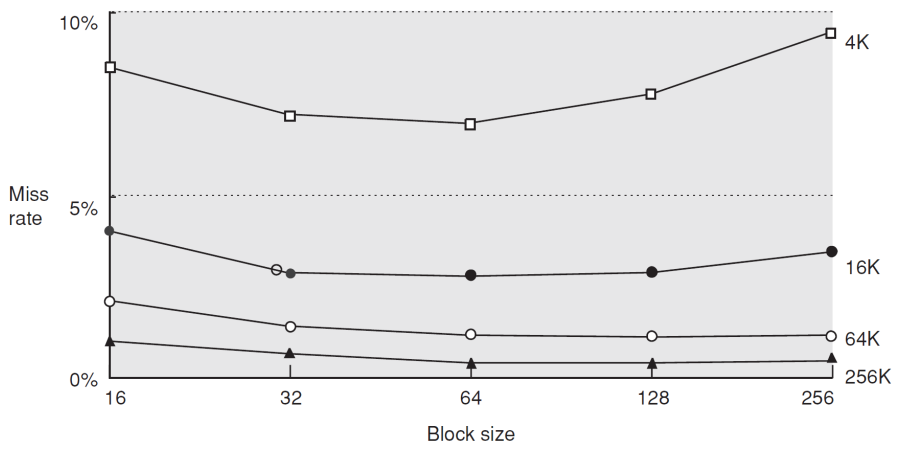


Figure 2.2: Miss rate versus block- and cache size for a specific program and architecture, taken from [1] (p.C-26). Both block sizes and cache sizes are in bytes.

- First In First Out (FIFO).

The considerations for these options are the following: LRU takes, as its name suggest, the least recently used item to be shifted out. This is in general the best option, however it is relatively expensive to implement since extra hardware is needed to keep track of the LRU block.

Random replacement on the other hand is cheap in implementation but performance will be less than LRU replacement in an unpredictable way.

FIFO policy will only look at the order the data came in. Since usage of the entries will differ individually, the behavior differs from LRU, although it can be seen as an estimation while being easier to implement. In practice it outperforms random for small cache sizes, while LRU is the most optimal replacement policy in any configuration (see [1], p.C-10).

LRU is therefore the best choice regarding performance, while random may be preferred in a simple or minimal area implementation. FIFO may be selected as a trade-off between area and performance.

2.2.2.5 Write miss policy

This parameter has, in general, just two options: write-through and write-back. Applying the write-through policy means writing the requested blocks not only to the cache but also at the same time to the LLM, incurring a high bandwidth requirement. Write-back on the other side will write the block to cache memory exclusively, incurring the need for extra facilities (usually a status bit) to keep track on whether the data is overwritten (called dirty) or not. Hence the LLM still contains the non-modified data, therefore care has to be taken making sure the copy in the LLM is not used before it is updated. The update is usually performed when the block shifts out of the cache.

A third option was introduced in [18] and is called eager write-back, which forms a compromise between the two earlier stated policies. The common solution to meet the high bandwidth requirements and blocking of the cache caused by waiting for stores is the addition of a write buffer. Such a write buffer can be used both for write-through and write-back oriented cache structures to reduce contention during writes. However, the authors claim that in many situations the write buffer will not suffice, therefore eager write-back was introduced as a modification on the write-back policy.

The idea proposed in [18] is to write specific dirty lines to the memory whenever the bus is free, balancing the memory bandwidth and hence improving overall system performance. The information of the LRU replacement policy (in a four-way set associative cache) is taken as an indication which lines should be eagerly written back. Hence the least recently used lines are unlikely to become dirty again before being replaced. Therefore the most recently used elements are most suitable to apply the eager write-back. A so-called eager queue is added to the design in addition to the write buffer. The eager queue holds the attempted eager write-backs which are added when the write buffer is full and checks the entry before moving it into the write buffer when an empty slot is available.

Summarized, the eager write-back scheme tries to optimize performance by efficiently using the bandwidth to the LLM. On turn, it adds complexity to the design, not only by logic elements, but also memory for the eager queue. Therefore this scheme should only be applied when bandwidth is a limiting factor in the system.

2.2.2.6 Identification of data

The identification needed to access the right piece of data can either be done by reading its tag or by its block ID, i.e. the location within the cache. However, the first option assumes a simultaneous read and processing of all tags, which means it has to be based on Content Adressable Memory (CAM) to be efficient. This fully associative memory is very hard to implement on an FPGA like previously mentioned (assumed it is not built-in already as a dedicated block). In normal non-CAM memory the identification practically needs to be done by using the block ID, which is predefined by the memory organization and acts as an address. Using CAM may increase cache performance since the blocks can be accessed by their tag, i.e. the associativity may be increased without the large overhead in access time. For this thesis however no CAM memory is available and therefore identification by the block ID is the only realistic option remaining.

2.2.2.7 Multi-level and split cache

Multi-level caches are often used in ASIC processors but generally do not favor FPGA designs. For instance, in a two-level cache, Level 1 (L1) is optimized for providing low access times, while Level 2 (L2) is maximized in capacity to reduce capacity misses. Note that implicitly, L1 has to be close to the core and implemented in the fastest type of memory, which is also very expensive. L2 in general resides on cheaper yet larger memory which is more remote from the core since performance is not the main criterium. In an FPGA this difference in memory types is barely present while the distance to the core is generally dictated by an optimization algorithm, diminishing the main benefits of a multi-level cache. Therefore in this project a single level cache is implemented.

The (L1) cache either can be split in a data- and instruction cache or unified for storing both. Note that L2 usually only consists out of a data cache. In practice, the unified structure incurs an increased rate of thrashing which was introduced in Section 2.2.2.1, therefore split cache is found to be superior (see [1] for further reference).

2.3 Fault tolerance - background

This section discusses various researches about gaining fault tolerance in memory systems and in cache structures specifically. It differentiates the several bases for fault tolerance and those most applied in practice in more detail. This section will not yet consider novel techniques and those which are more involved, these will be discussed in Section 2.4.

In this section, firstly bases and most common techniques are introduced. Note that these techniques are referenced in many of the papers included in this chapter. Consequently various papers will be reviewed dealing with the considerations involved while applying the basic techniques.

2.3.1 Bases to obtain fault tolerance in memory structures

In essence fault tolerance in a memory architecture means being able to continue working correctly in the presence of faults. Therefore, the faults induced may not propagate to become an error or even a failure. To clearly differentiate faults, errors and failures, the definitions from [19] are used:

- Fault: causes a problem in the system,
- Error: state between fault and failure,
- Failure: surfacing of an error ('what you see').

To make an architecture fault tolerant, the very first step is to detect any error occurring in the data being used. Optionally, after detection the error may be corrected. When no correction mechanism is present, the processor needs to take care of the fault in another way, otherwise it may still become a failure. E.g. a restart of the system may be needed or a rerun of (a part of) the program to prevent the failure from happening. Both are costly measures in terms of performance (time spent to complete the program) and energy.

Although error correction is always more expensive in terms of implementation cost than detection, it has a clear main advantage over detection-only. Especially in the case of a non-negligible error rate many of the restarts can be prevented by run-time correction of an error. Dependent on the application domain the focus may be either on error detection (e.g. in high-security) or error correction (e.g. for space applications).

There are two main directions to follow (bases) in order to achieve fault tolerance in a given memory system, which will be explained in Section 2.3.1.3 and Section 2.3.1.4. Prior to that, some definitions are introduced to differentiate fault causes, fault effects and to define which forms need the most attention in this project.

2.3.1.1 Causes of faults

Two main causes for faults can be differentiated: natural and human attacks. The first (natural) is often referred to as Single Event Effects (SEEs). Several papers elaborate on this cause in their introduction, hence SEE is often part of the background for these papers. To summarize the information found in these introductions and other sources, [4] was chosen as a base together with the SEE background report of Microsemi [2].

SEEs can be described as device failures induced by a single radiation event and can be either "soft" or "hard", referring to a temporal or permanent failure respectively. For commercial terrestrial applications the soft SEEs are of most concern [4], hard SEEs mainly occur at manufacturing time. Given the background of this thesis project of using an FPGA, the hard errors originating from manufacturing are already taken care of: devices with hard errors from manufacturing may not be shipped. Other occurrences of permanent faults (e.g. Single Event Latchups (SELs)) are rare and will not be taken into account in this project. Soft SEE is often referred to as soft error or Single Event Upset (SEU), these terms will be used interchangeably in this report.

The (natural) causes of (soft) SEEs are mainly due to the following two sources:

- Ionizing particles with high energy. These originate either from manufacturing e.g. α-particles (originating from chip packaging) and ^{10}Boron isotopes (present in the silicon itself), or external sources e.g. cosmic radiation (mainly protons and again α-particles).
- Atmospheric neutrons generated by cosmic rays.

The second group of causes, apart from natural SEEs, mainly occurs in security applications. In these circumstances, a System on Chip (SoC) may be attacked by humans to force the system going in an erroneous/vulnerable state. Several known options to do this are:

- A glitch in either:
  - Clock,
  - Power supply,
  - Data (by corruption),
- A temperature change out of the specified range,
- Exposure to light, laser and other forms of radiation,
- Placement in a magnetic field.

Faults occurring from these sources are less predictable and often they incur (very) large sets of faults at once. For instance, by laser a very high amount of energy may be inserted on a specific part of the SoC, able to flip multiple bits or logic gates randomly around the location of incidence. Since the impact is expected to be high, for security applications it is common practice to aim for detection of such attacks. After detecting such a threat the execution may be terminated to prevent attackers e.g. from obtaining sensitive information or further disturbing the process.

2.3.1.2 Effects of faults and main issues

Irrespective of the cause, fault effects can be divided in the following categories:

- Single Bit Upset (SBU),
- Multiple Bit Upset (MBU), also referred to as Single Event Multiple bit Upset (SEMU),
- Single Event Functional Interrupt (SEFI),
- Single Event Transient (SET).

SEEs due to natural causes are predictable (to a certain extent), while fault attacks by humans are not. Therefore natural SEEs in general involves more SBUs compared to MBUs. For fault attacks such an assumption cannot be made. Later in this report it will become clear that MBUs require more involved actions to deal with them and therefore this difference is important.

Continuing the list, SEFI refers to an error in a critical control register, for instance the configuration memory in an Field Programmable Gate Array (FPGA) which determines the functionality of the device. In [2] it is stated that this kind of error can be very damaging for an application, even with fault mitigation techniques. This is because the functionality of the device changes immediately and mitigation techniques could be optimized only for minimizing the time to repair the error rather than prevention. However, the IGLOO2 family of Microsemi uses flash memory instead of the vulnerable Static RAM (SRAM) memory for configuration [16]. Flash memory has by design considerably increased resistance against soft errors when compared to SRAM. This is also proven during tests performed by Microsemi. The test presented by Microsemi targets neutron flux resistance, which is the most energetic source for radiation [4]. In Figure 2.3 it can be seen that at 60,000 feet (approx. 18,000 meter) the neutron flux is at a maximum. The tests performed resulted in no single error found at comprehensive fault testing for altitudes up to 60,000 feet, while SRAM based devices showed multiple errors [6].

Based on this test, it is assumed the flash-based FPGA will not produce any errors in its configuration memory when applied in avionic and space applications. Also it is deemed very unlikely that the configuration may be changed by a human attack, however no literature was found on this topic and therefore no substantiated conclusion could be drawn. The range of the neutron flux included in Figure 2.3 is one to ten MeV, which is the vast majority of the spectrum as shown in [2].

Finishing the list at the beginning of this section, SET is of less importance for this thesis project. The reason for this is that it refers to errors in combinatorial circuits, which is the main concern of the partially adjacent project of bringing fault tolerance to the softcore. A SET refers to one or more voltage pulses propagating through the circuit, which does not necessarily incur an SEU.

2.3.1.3 N-way Modular Redundancy (NMR)

NMR is the first of the two base techniques for fault tolerance. The term implies multiplication (to a total of N instances) of either the whole architecture or parts of it,

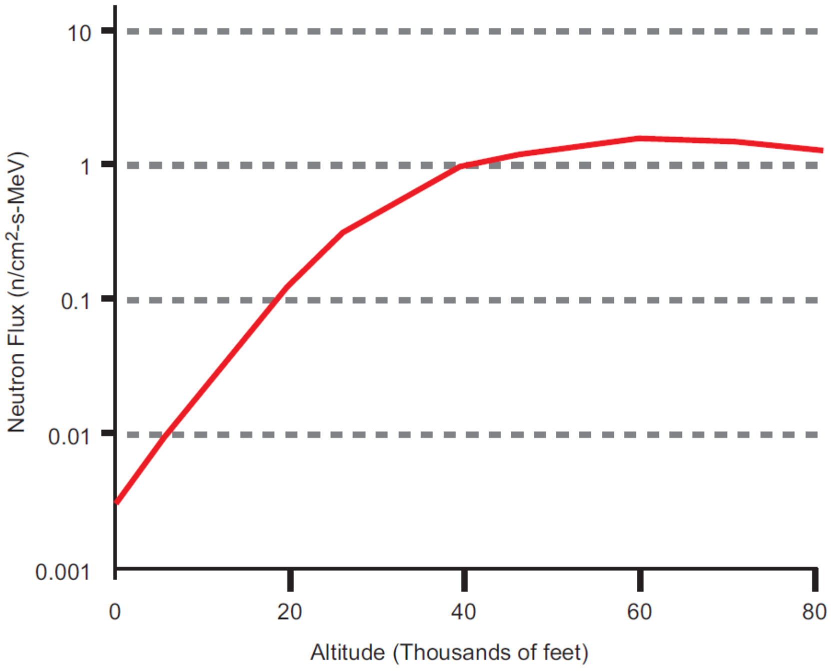


Figure 2.3: Neutron flux (range 1-10 MeV) versus the altitude (range 0-80,000feet) [2].

appended with some regulating logic. When an error occurs in one implementation, it could be detected by comparing the output with that of its copy or copies: any inequality highlights an error. When more than one copy is added (i.e. N is three or more) the possibility emerges to correct the error by majority voting. This technique uses the assumption that when e.g. two of three instances give the same result, the third was erroneous and the result of the other two is correct and can be passed.

The most popular form of NMR is Triple Modular Redundancy (TMR) which uses three instances of the same design. Using TMR, a majority voter can be used to select the right output, hence the design is error correcting in the case that just one error occurs. Note that at multiple errors causing three different outputs the voter cannot decide which output is correct, and when two copies produce the same error the voter will miscorrect the output, although these occasions are rare in typical usecases.

Double Modular Redundancy (DMR) is another special case of NMR using two instances of the same design. This version is only able to detect errors, since no decision can be made when the two outputs are different and thus an error occurred. 5-way modular redundancy is only used in high security applications. Its added value is only in the case of an attack introducing either identical errors in two instances or differing errors in three instances. In these cases TMR would not suffice, while 5-way modular redundancy is able to correct the errors in these situations without any fault propagating to the output.

The largest drawback of NMR in general is its large overhead in terms of area and power. The design has to be duplicated N times and additional circuitry is needed to control the outputs. NMR appears to be a fairly good option for combinatorial circuits, but it is not popular for memory architectures regarding their size and deviating performance requirements. For memory architectures the most commonly used direction for fault tolerant design is EDC and ECC.

2.3.1.4 Error Detecting Code (EDC) and Error Correcting Code (ECC)

Second base technique makes use of, as the name suggests, (extra) code called the check code appended to a certain data quantity. This extra bit or extra bits are dedicated to detect and/or correct errors when present.

The simplest form of detection-only code, categorized under EDC, is the well-known parity bit. Such parity can be appended to any quantity of data, e.g. per byte, word or block. In theory, a single parity bit is able to detect an uneven amount of errors in its domain or data quantity covered. In the case of an even amount of errors occurring, the parity matches with a correct code, i.e. the errors are not detected. Therefore one bit parity is only applied in case of low error probability, i.e. chances of double errors are negligible and single error rate is low. This situation normally means a safe environment (i.e. terrestrial and no attacks possible), but may also occur when small data quantities are taken, although in the latter case it may be more efficient to store larger quantities in a higher quality code. Multiple bits parity can provide a better error detection, but is never able to correct an error and is therefore barely applied in practice.

ECC on the other hand is able to recover the erroneous data up to a predefined error count without harming the execution flow. As stated before, correction always comes at the price of additional performance and area overhead compared to EDC. Most popular class of ECC is Single Error Correcting/Double Error Detecting (SEC/DED) code. To implement this scheme, extended Hamming code [20] can be used, Hsiao [12] or other codes as will be discussed and compared later. More advanced codes could provide Double Error Correcting/Triple Error Detecting (DEC/TED) or even higher detecting/correcting capabilities, traded against additional performance- and area overhead. In this report, the term Error Detection And Correction (EDAC) will be used as synonym for both EDC and ECC.

In general, EDAC is represented either by the (n,k,r) or (n,k) notation substituted with code name (e.g. (22,16,6)/(22,16) Hamming code). In this notation n represents the total number of bits in the codeword (data + check), k the number of data bits and r=n−k the number of check bits. Since r is trivial when both n and k are known it is often left out of the notation. Every specific code can be represented by its Parity Check Matrix (PCM), often designated by H. It dictates the way the data- and check bits are arranged and has dimensions r · n (see for instance [**?**]).

To summarize and compare both bases for fault tolerance, ECC and TMR are compared in Table 2.1. Tabulated are the most common techniques applied. The table is taken from [9].

Scrubbing A technique commonly used in conjunction with EDAC is called scrubbing. On a regular basis, all or specific parts of the memory are read in order to check for faults. When an error is found, it is detected and can be corrected when it is covered by the error correcting capability of the applied code. The idea behind it is that the number

Table 2.1: Comparison between the most common ECC techniques and TMR [9].

	Hamming (SEC/DED)	TMR	RS (DEC/TED)	BCH (DEC/TED)
Area	Small overhead to implement. Varies depending on the number of data bits (7-32%).	+200% plus the voting and correcting logic. No. voters proportional to the number of units.	Varies depending on the number of data bits (13-75%).	Varies depending on the number of data bits (13-75%).
Performance	May be affected by the coder-decoder functions. Proportional to no. bits to be corrected.	High performance. Voter is the only source of delay, hence almost constant delay.	Lower performance than BCH and much lower compared to Hamming and TMR.	Higher performance than RS but much lower than Hamming or TMR.
Error correcting capability	Limited: only corrects one single incorrect bit per word.	Corrects up to n errors in an n-bit word as long as the errors are located in a distinct position/unit.	Able to handle multiple errors: efficient for correlated errors (e.g. in a burst).	Able to handle multiple errors: efficient for uncorrelated errors (e.g. random errors)
Realization	Binary based. Simple to implement.	Simple to implement.	Symbol based. Complex to decode and implement.	Binary based. Complex but simpler than RS to decode and implement.

of errors grows over time when data is stored on the same position. Note that without scrubbing the data is only checked during accesses (also called passive scrubbing) which is possibly very irregular in time taking one element in the memory. In order to prevent multiple errors to accumulate over time, especially in larger caches, cache scrubbing is a well-known and commonly applied technique.

Physical interleaving Another technique which is particularly useful in combination with EDAC is a physical interleaving of memory cells. MBUs generally occur on adjacent memory cells due to an impact with high energy. In a typical ordering of bits this would incur multiple-bit errors in one block, a type of error which is hard to cope with. In an interleaved memory structure however adjacent cells belong to different blocks. This minimizes the chance for multiple errors in one block significantly, enabling a SEC/DED code to detect and correct all errors even in the case of multiple bits flipping around one position.

2.3.2 Surveys and comparisons of fault tolerant techniques

This section will go through the papers which survey and/or compare the most prevalent fault tolerance techniques.

The differences between parity and SEC/DED code for both tag- and data arrays are addressed in [21]. Main question arising in this work is: what is the best technique to protect a cache architecture against soft errors and/or transient faults?

Several combinations of parity and SEC/DED code are applied to tag- and data arrays respectively. Conclusion in this paper is that for very small error rates or relatively small cache memory parity code is sufficient to protect the cache. However, for noisy environments (e.g. space) or large caches, cache scrubbing or SEC/DED code is claimed to be essential. Therefore the conclusion can be drawn that for the current project SEC/DED code is required and cache scrubbing may be a valuable addition.

Another investigation focuses on SEUs occurring in outer space environments [22]. The cache memory is considered but also main memory and logic circuits. Downside of this work is its age (1993), although the basic techniques are still the same today. The techniques reviewed are TMR, shadowing (also known as DMR) and EDAC combined with scrubbing.

The authors conclude several things based on probability calculations performed. First, even in space applications, TMR and DMR are too costly to implement (note: at that time). Second, also EDAC should not be preferred for cache memory due to its large overhead. Therefore, parity is proposed instead, combined with cache scrubbing.

Note that in those days caches were still small and adding extra bits was (very) costly. Therefore the preference for parity instead of EDAC is likely to vanish when the technology scaling since then is considered, which reduces the relative costs significantly. Still, the paper clearly shows that even the basic fault tolerant techniques come with a non-negligible cost.

The next research [23] aims for the lowest possible area overhead while maintaining sufficient fault tolerance. The authors plead that the tag array may also contain bit errors. When these remain unsolved, they incur chaotic memory references which are hard to cope with. Therefore the tag arrays have to be protected with EDAC just like the data.

The same paper proposes a new technique called parity caching. It consists out of a small cache containing check codes for the Most Frequently Used (MFU) blocks only. The idea behind this is that the most errors will occur on the MFU blocks. Therefore those blocks favor protection with EDAC. To mitigate performance impact an additional, small and fully associative cache is used to store the additional check bits. In this scheme, rest of the data and status bits are protected by a parity bit per byte. Cache scrubbing is considered to be useful for this scheme featuring low capability check codes.

Another technique discussed is shadow checking (or DMR) as it is the most area-efficient alternative of NMR. It is used to check the data on a read hit, instead of using EDAC for this purpose. Also for this scheme the authors choose to apply it on the MFU blocks instead of protecting the whole cache space, differentiating from typical DMR.

In short, the authors propose to use selective protection and partial checking as techniques to minimize the area while maintaining sufficient fault tolerance. Tests show that implementing the proposed techniques, incurring a small increase in area, a relative high level of fault tolerance can be obtained. However it is questionable whether the risk of taking an uncorrectable or even undetectable error for the part not in the MFU blocks is worth the savings in area for this thesis project.

A research performed more recently [3] provides more insight in the occurrences of multi-bit errors, in particular the nature of them and the dependency on technology node. Various technology nodes were considered, ranging from 180 nm down to 40 nm. Also different types of implementations were tested, including flip-flops, logic and SRAM, of which the last is most relevant for this thesis project. All the combinations are tested while a neutron beam was applied to the circuits.

First results which are of interest for the current project are the results displayed in Figure 2.4. It shows that SRAM in 65 nm, as used in IGLOO2, has the smallest occurrence of SEUs across different technology nodes. Furthermore it is shown that there

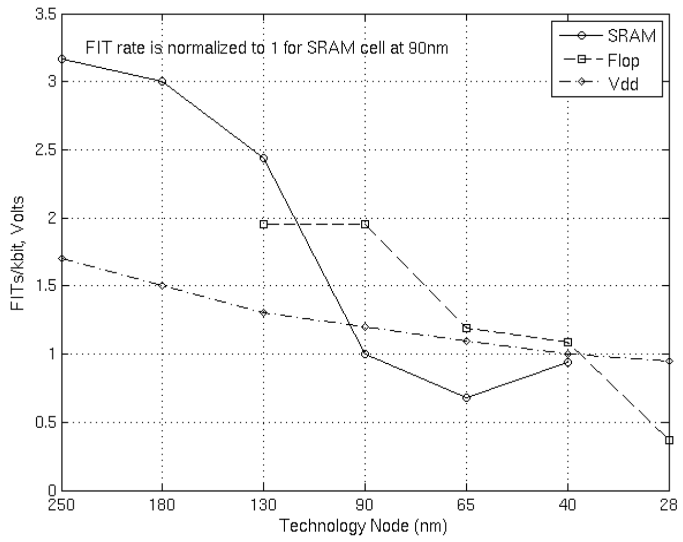


Figure 2.4: SRAM and flip-flop (FF) SEUs as a function of technology node, taken from [3]. Additionally, the voltages, which are specific to the technology node, are included to the figure.

are multiple occurrences of multi-bit errors in various patterns, most of them occurring in adjacent cells (see Figure 2.5, showing results for 90 nm). In 65 nm technology more than 20% of the errors in SRAM are multi-cell errors and approximately 5% incorporated three cells or more (see Figure 2.6). This means that for a considerable amount of errors SEC/DED code would not be sufficient when a traditional bit ordering scheme is applied. Therefore physical interleaving is suggested by the authors as the most appropriate solution, which is also featured in the SRAM chips on the IGLOO2 as will be shown later.

The next research also states that MBUs are tackled by physical interleaving [24].

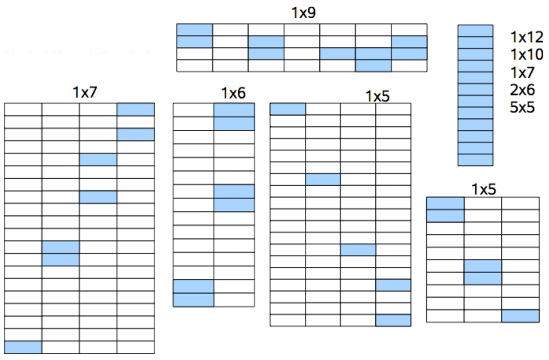


Figure 2.5: Multi-cell upset patterns for 90 nm (#occurrences × #error bits) [3]

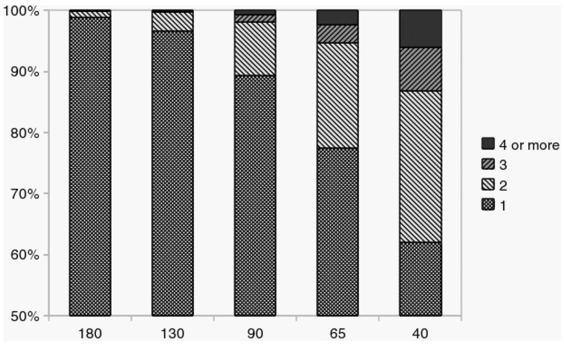


Figure 2.6: Multi-cell error percentages by technology [3]

However, the researchers state that physical interleaving combined with SEC/DED code may be insufficient because MBUs may still occur. Therefore several Double Error Correcting (DEC) codes are discussed and compared to each other, as well as to Hsiao SEC/DED code [12] which is claimed to be the de-facto standard for its limited impact on cache area and performance.

The authors mention the simplest solution, which is the shortening the codeword length i.e. the data quantity on which the ECC is applied. However this means that in total more check bits are needed and hence the overhead increases. Analysis is done for a full-blown processor design which is not comparable to the softcore used in the current project. For the DEC codes it is concluded that there are several trade-offs to make before one can be chosen. The most promising DEC code, according to the authors, is a form of linear sum code [25]. Hsiao SEC/DED code on the other hand seems to be the most efficient in area and performance metrics compared to other SEC/DED codes and performs relatively well compared to the DEC codes. Since the design in the current project is simpler and smaller and therefore inhibits a lower chance on MBUs, these results suggest that implementing Hsiao SEC/DED code may be a good option.

A research which gained considerable impact is [10], which was referenced multiple times in the set of papers discussed in this report. Originating from 2005 it provides an extensive investigation about fault tolerance for terrestrial servers and workstations, i.e. an environment in which soft errors are very likely. Both SRAM and Dynamic RAM (DRAM) are considered and experimental data for cell upsets are provided.

One of the conclusions drawn based on these data is that the probability of two nearest neighbour cells being upset by a single particle strike is ten times less than a single cell upset, and equivalently for three cells it is 100 times less. A table is provided (shown in Table 2.2) comparing the overhead in bits for SEC/DED, Single Nibble error Correcting/Double Nibble error Detecting (SNC/DND) and DEC/TED for different amounts of data bits (granularity). SNC/DND takes nibbles (four bits) as smallest components instead of bits. This means that when multiple errors occur in one nibble, both may be corrected. However, when two errors occur in adjacent nibbles, they are not correctable. As a conclusion the author claims that 72 bit code for 64 data bits, providing SEC/DED

Table 2.2: Comparison of ECC schemes, taken from [10]

	SEC/DED			SNC/DND			DEC/TED		
Data bits	check bits	overhead	total bits	check bits	overhead	total bits	check bits	overhead	total bits
16	6	38%	22	12	75%	28	11	69%	27
32	7	22%	39	12	38%	44	13	41%	45
64	8	13%	72	14	22%	78	15	23%	79
128	9	7%	137	16	13%	144	17	13%	145

protection, is relatively efficient and practical to implement.

Other conclusions include that for smaller on-chip caches parity might be sufficient and that scrubbing, even once per 24h, is probably overkill, but penalty in performance and power is nil. Furthermore again the conclusion is drawn that physical interleaving virtually eliminates the threat of MBUs in large memory arrays. TMR is also referenced for application in FPGAs, specifically for mission-critical avionic and space applications.

The author concludes the work with the statement that ECC (typically SEC/DED or SNC/DND) combined with interleaving and scrubbing is the most cost-effective for protecting large memory arrays. Note that also this paper points towards SEC/DED e.g. Hsiao code, combined with interleaving and optionally scrubbing being a sufficient EDAC for the current project.

Also in [26] SNC/DND is considered, however it is declared only to be applicable for hard disks and main memory for their complexity, likewise are DEC/TED, Reed-Salomon (RS)- and Bose-Chaudhuri-Hocquenghem (BCH) code. For cache memories, the authors only consider parity and SEC/DED code feasible since they inhibit low latency decoding. This paper therefore confirms the thought formed by papers discussed before that SEC/DED Hsiao code could be the most optimal solution for this thesis project.

Next paper [27] investigates the need of scrubbing in caches of microprocessors and originates from 2004. Scrubbing was introduced in Section 2.3.1.4. It is able to diminish temporal double-bit errors when the cache is only protected against single-bit errors (e.g. by SEC/DED code).

In the paper it is analysed what the Mean Time To Failure (MTTF) is for double-bit errors in cache memories. The conclusion for computers in normal execution (i.e. only including radiation at earth surface) is that scrubbing is only required for very large caches, i.e. hundreds to thousands of megabytes. However, as stated in the paper, the Failure In Time (FIT) per bit is approximately hundred times higher on 10 km altitude (typical for airplanes) and also technology impacts this metric, on which the calculations are based. Focusing on double-bit errors, it is stated that unmodified data can be re-fetched from Lower Level Memory (LLM), but only when the error is in the data portion.

Since scrubbing takes some resources, i.e. at least some energy and area, it can also become a disadvantage. For example in the case of a double-bit error residing in data which is not used again by the processor, in this case the scrubbing is useless and a waste of energy and resources. Note that passive scrubbing takes place when new data comes into the processor, the only difference with active scrubbing as discussed here is that there is no upper bound on the interval between ECC updates. In order to compute the FIT rate, 8-bit ECC code was taken for 64 bits of data, which also impacts the final conclusion since better protection will reduce the FIT rate and vice versa.

Concluding this paper, it forms a base for deciding whether scrubbing should be applied in this thesis project or not when protecting for natural cause SEEs. When comparable protection would be applied as in this paper (SEC/DED for 64 bits of data) and considering the cache size to be in the range of tens of kilobytes, scrubbing would not be necessary for the current project, even at 10 km altitude and beyond.

As opposed to most other papers, in [28] a more powerful ECC scheme was chosen. It is effectively implemented for SRAM applications, although the application domain is an Application-Specific Integrated Circuit (ASIC). DEC is proven to reduce the uncorrected error count by more than 98% compared to 44% for SEC/DED ECC during heavy-ion-induced soft error testing. A parallel implementation is constructed for BCH code, which is considered as being the most effective DEC code. DEC/TED can be obtained by adding an overall parity bit to DEC BCH code, making the extra overhead relatively very small.

This research points in the direction of BCH code for DEC and DEC/TED implementations which are claimed to be needed for providing high security (i.e. in case of SEEs due to human attacks). Note that physical interleaving was left out of the research which could alter the conclusions drawn.

The research done in [29] is special since it describes the implementation of the LEON processor, a softcore which is comparable in terms of features and performance to the one implemented by Technolution B.V.. The LEON processor also features fault tolerance for application in space applications. Design decisions made for the LEON case therefore indicate what choices are appropriate for the design considered in this project.

The decisions made include the choice for one (or two when no physically interleaved memory is used) parity bit for each tag or data word in (single level) cache and (39,32) SEC/DED BCH code for external memory (which is DRAM in this case, just as in this thesis project). Also a high-speed Advanced Microcontroller Bus Architecture (AMBA) [30] AHB bus was used for transfers between the caches and external memory controller and a low-speed AMBA APB bus for attaching on-chip simpler peripherals. Note that in later revisions of AMBA, AXI was introduced as successor of Advanced High-performance Bus (AHB). AMBA buses will be discussed later in Section 2.6.2.

From this paper the conclusion can be drawn that the AMBA buses are suitable for connecting caches. Furthermore, the cache could be protected with a single parity bit per word while LLM features SEC/DED per word, which is less protective than concluded from other papers but may be an appropriate choice for this thesis project as a light protection scheme.

A dated 'state-of-the-art review' of ECCs is found in [31], of which Hsiao (the inventor of the Hsiao code) is one of the authors. Hsiao codes are indicated as odd-weight column codes and are widely implemented those days by IBM and the remaining computer industry worldwide for their improvements in speed, cost and reliability of the decoding logic over Hamming codes.

A first note on the SEC/DED codes is that memory systems should provide a certain distance between adjacent bits in a word to make multiple errors in the memory correctable (i.e. physical interleaving). Single Error Correcting/Double Error Detecting/Single Byte error Detecting (SEC/DED/SBD), Single Byte error Correcting/Dual Byte error Detecting (SBC/DBD) and DEC/TED are introduced as more powerful codes which are supposed to be necessary for maintaining the high level of reliability. In this work, the word 'byte' does not have to refer to a quantity of eight bits but may also represent another quantity of bits. For SEC/DED/SBD and SBC/DBD, it means that they are able to detect/correct multiple errors, as long as they are in the same byte or pre-defined quantity of bits. These codes, together with SEC/DED, are compared based on their overhead in check bits and implementation aspects.

Three memory systems containing four megabytes were considered, first with parity per byte, second with (72,64) SEC/DED code and the last with (80,64) DEC/TED code. However, also hard errors are considered which are ignored for the current project, making SEC/DED/SBD and SBC/DBD irrelevant as concluded from the paper. The amount of uncorrectable errors is reduced by 15 and over 84 times compared to parity by using SEC/DED and DEC/TED respectively. An overview of overhead in bits is shown in Table 2.3 which is composed using information found in [31] and extends the comparable Table 2.2 by including parity.

Table 2.3: Comparing bit overheads for different EDACs. The parity entry represents a parity bit per byte (8 bits).

Data bits	16	32	64	128	256
Check bits - parity	2	4	8	16	32
Check bits - SEC/DED	6	7	8	9	10
Check bits - DEC/TED	11	13	15	17	19

From this paper several options can be disregarded as candidates for the current thesis project, which are SEC/DED/SBD and SBC/DBD. The reason is that they are claimed to be particularly beneficial for mitigating hard errors, which are not taken into account in this thesis project. Furthermore, they add considerable bit overhead. Another conclusion can be drawn by using Table 2.3: parity for larger data quantities is quite expensive in bit overhead, while DEC/TED and SEC/DED become more attractive for increasing data quantities. Lastly, Hsiao code is still regarded as being a good candidate to apply in this thesis project.

2.3.3 Conclusion

In this section the concepts of fault tolerance are introduced and the causes of faults distinguished. Also the most prevalent fault tolerance techniques are explained and discussed. Subsequently, research examining and comparing these techniques were described, giving an initial indication which technique(s) are standard in industry and which are probably the most efficient in terms of overhead and implementation costs. Note that only well-known techniques are included in this section.

It appears that Hsiao SEC/DED code is a good candidate for protecting (cache) memory in low-security applications in which multi bit errors will barely occur. To increase the effectiveness of a SEC/DED code, physical interleaving is a valuable addition. For higher levels of fault tolerance DEC/TED code and others can be used, however the additional overhead and implementation costs grow quickly when moving to these solutions. Therefore this may only be applied for special cases like environments prone to human attacks or may be considered for longer-term storage, e.g. in the LLM.

2.4 Fault tolerance - techniques

In this section two main topics will be included. First, the techniques encountered in Section 2.3 will be explained in detail. Second, more advanced and/or less known techniques and optimizations will be discussed and examined for this thesis project.

2.4.1 Common techniques explained

Firstly the most commonly used fault tolerance codes will be explained in more detail. This part will provide more details on Hamming code, Hsiao code and Bose-Chaudhuri-Hocquenghem (BCH) code, using the original works in which they were proposed (when possible).

2.4.1.1 Hamming code

Being the oldest commonly-used technique, the well-known Hamming code is also relatively simple to understand. The code is named after the author of the original paper [20], Richard W. Hamming, who laid the foundations of Error Detection And Correction (EDAC) and introduced Single Error Detection (SED) code, Single Error Correcting (SEC) codes and Single Error Correcting/Double Error Detecting (SEC/DED) codes. At that time (1950), no Error Correcting Codes (ECCs) existed yet, only Error Detecting Code (EDC) had been employed, mainly in communication channels. The term parity check originates from the same paper and the concept of traditional Hamming code is introduced. In this section it is attempted to summarize the explanation in order to understand the principles of Hamming code.

Parity checks and SEC Important to note at this point is that parity checks form the base of any traditional EDAC code. For Hamming SEC code the scheme is as follows: the first parity check takes all uneven bit positions (e.g. 1, 3, 5, 7, 9, ...) to determine its parity. Second one starts checking at the second position, takes two into account, then skips two and repeats this (e.g. 2, 3, 6, 7, 10, 11, ...). Third parity bit takes the fourth bit (2^{3-1}), takes four bits and skips four repeating this also (4, 5, 6, 7, 12, 13, 14, 15, ...). This pattern is continued as far as the codeword permits. Accordingly the check bits are placed in the codeword in such a way that they are independent on each other and together represent the bit positions where an error has occurred (for a simple illustrative example see [20]).

SEC/DED code The code is extended to SEC/DED code by adding an extra bit which contains the parity check over all previous positions. This is also called modified Hamming code, while SEC is often referenced as conventional Hamming code. The idea is that when a double error occurs, the SEC part will indicate the error while the overall parity check does not indicate an error, hence there must have been two errors. In the case of one error both the parity bit and the SEC part indicate an error. Note that in the presence of more than two errors there are no facilities to detect these errors. These can be either misinterpreted as single bit errors (hence it will miscorrect) or as another valid codeword, which is not the intended codeword. Note also that the number of redundant bits (i.e. check bits) in Hamming code increases with $\log_2(n)$. For an illustration on this subject see Figure 2.7.

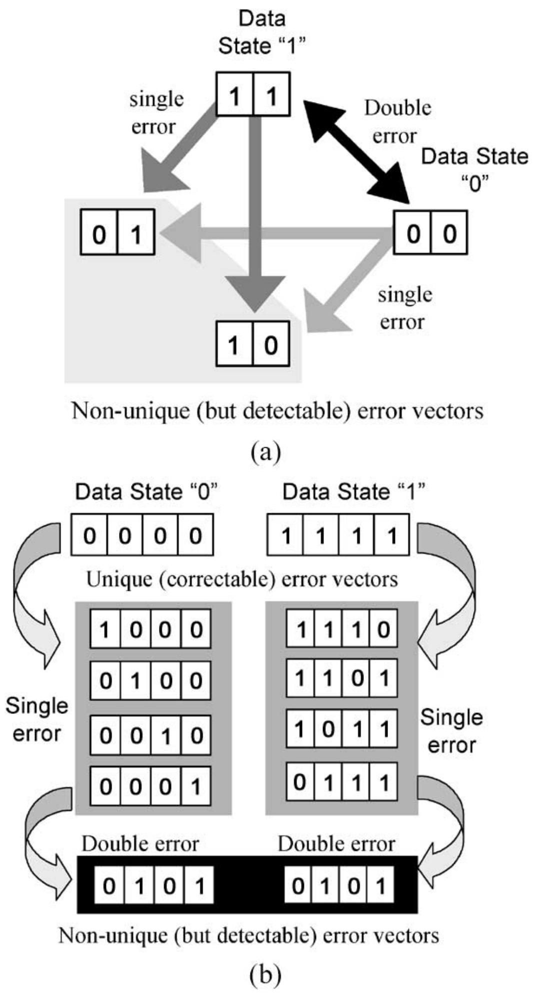


Figure 2.7: Parity code is able to detect one error as illustrated in (a). SEC/DED code is illustrated in (b). Figure taken from [4]

Hamming distance and example Another part of the paper introduces a metric often used to indicate the strength of a code: the distance of a code, known as Hamming distance. In short, it refers to the number of bit positions in which codewords in a set differ from each other. When considering all codewords within a code, the minimum (Hamming) distance is the minimum number of differing positions between all combinations of codewords. The minimum distance is often referred to as the Hamming distance of a code.

As an example, the code consisting of codewords (001;010;100;111) has Hamming distance two, since every codeword differs in two bit positions of any other codeword. The Hamming distance of a code indicates the capability for detecting and correcting errors. Given the example code, (000) is not part of the code, so when this codeword is received an error has occurred: the error is detected. However, this error cannot be corrected since either of the codewords (001;010;100) differ in only one bit position.

For correcting a single error a Hamming distance of three is needed, e.g. the code (000;111) will correct (001;010;100) to (000) since it differs in one bit position opposed to (111) which differs in two bit positions. Note that if two errors occur, the codeword will be miscorrected to (111). Hence a larger Hamming distance increases the capability of a code to detect and correct errors. The designer may decide whether to implement for detecting or correcting more errors, e.g. a code with Hamming distance five can either be used for double error correction, single error correction plus triple error detection or quadruple error detection.

2.4.1.2 Hsiao code

Subsequently Hsiao code can be addressed, which is also named after the author of the paper in which it was proposed [12]. In [24] the code is referenced as the de-facto standard for cache memories. [32] mentions Hsiao code together with Hamming as the most commonly applied EDAC codes for cache memories These are just two examples out of many papers which apply and/or reference this code.

As a matter of fact, Hsiao code is just an improvement on both conventional and modified Hamming code. The main difference is the way in which double errors are detected, as reviewed in e.g. [5]. While Hamming adds an extra check bit in order to do that which is the parity over all data and parity bits, Hsiao code applies the property of odd weight columns. Every column in the Parity Check Matrix (PCM) (see Section 2.3.1.4) contains an uneven amount of ones. Therefore the modulo two sum of any even amount of odd-weight columns always is an even-weight vector. Similarly, the modulo two sum of any uneven number of these columns always is an odd-weight vector.

It improves on the Hamming code by a simpler hardware implementation and an improved error detection rate. This is achieved by minimizing the number of ones in the PCM. One of the main contributions is eliminating the all-one row which was used for double errors in the Hamming code: the extra parity bit accounting all other bits. Since ones in the PCM implicitly require extra hardware for en-/decoding, the implementation with modified PCM is more efficient in terms of area, power and latency.

Hsiao code belongs, just as Hamming code, to the class of SEC/DED codes, hence as such it is not suited to detect triple errors by nature. However it was shown that triple errors are either detected correctly or miscorrected with better results than Hamming code.

Summarized, Hsiao code, in comparison with Hamming code, can be implemented more efficiently in terms of en-/decoder costs. Also it provides better EDAC performance which makes this code favorable over Hamming code in case a SEC/DED code is needed.

2.4.1.3 BCH code

As third and last general coding technique the class of BCH codes is taken into account. BCH codes form a class which can be configured such that it provides a higher level of EDAC than the latter two options which belong to the class of SEC/DED codes. Put differently, SEC/DED codes realize a maximum Hamming distance of four while BCH codes theoretically cover all Hamming distances.

BCH code was firstly described in [33] by A. Hocquenghem and shortly after independently published by Bose and Ray-Chaudhuri in [34], hence the name has become BCH. Only [34] is considered here since [33] is written in French.

The class of codes is described in [34] as a systematic method for t-error correcting (n,k) binary group codes. The t could be an arbitrary number within the limits of an n-bit code ($n = 2^{m}-1$, m being a positive integer). In this code the number of data bits is represented by $k \geq 2^{m}-1-m\cdot t$. Examples satisfying these equations are (63,51) for t=2 and (127,99) for t=4. The details of constructing an arbitrary t-error correcting code and the mathematical proof of the theory will not be discussed here.

BCH codes can be considered as a generalization of Hamming codes. Multi-error correcting BCH codes are hard to implement in hardware as comes forward from different papers. Therefore an implementation is expected to be too complex and time-consuming in the context of this thesis project, but may be a good option to add higher levels of fault tolerance when needed in future applications.

2.4.2 Other techniques proposed

Although the common techniques are well-known and applied for many years now, research is still done to further improve these techniques/codes or even propose new ones. In this section the most interesting ones are listed and summarized.

The first improvement to address is called PSP-cache, as documented in [35]. PSP-cache is short for Per-Set Protected cache. This name indicates the main idea of this technique: generate and store the check code for EDAC per cache set instead of per block or line. This implies that the cache has to feature set-associativity (see Section 2.2.2.2). In this thesis project, only the data cache features this in the form of two-way set-associativity.

The assumption is that a full set is read (i.e. checked) at once in parallel at every access in order to minimize hit time. Therefore the whole set could be used as one piece of data to employ EDAC, achieved by putting the EDAC circuitry before the way selection logic. Accordingly the check codes are stored per set instead of per block. The technique is applicable for any sort of EDAC code, e.g. parity or SEC/DED code. In the paper parity protection is recommended for write-through caches and SEC/DED code for write-back caches (see Section 2.2.2.5).These EDAC measures may further be improved by applying interleaving.

Advantage of this technique is a reduction of both area and energy, dependent on the level of associativity and EDAC type.However note that the covered number of bits by one codeword is larger. This means an increased error probability and hence a stronger code should be used to maintain the same protection level, although the authors claim that this technique does not incur a lower error coverage. A disadvantage which is subscribed by the authors is the access time which may increase by using the technique.

Concluding, PSP cache could decrease area in this project without incurring too much extra latency. However it is only applicable if the design is set-associative and accesses all of its ways simultaneously, which is only the case for data cache. Last but not least, it has to be further examined what the implications are for the level of fault tolerance.

A technique called Memory Mapped ECC is proposed in [32]. The basic idea is to reduce the area needed for EDAC in the Static RAM (SRAM) cells used by the cache.

While error detection is essential for every cache access, error correction on the other hand is very infrequent and scarce. Remember that error correction requires more check bits (and hence, area) compared to error detection. To save costly SRAM space, the extra check bits needed for error correction can be stored in Lower Level Memory (LLM), i.e. Dynamic RAM (DRAM). To fetch the extra check bits it takes considerable time compared to SRAM, but since this is only needed when an error occurs it has no impact on performance during normal operation. Note that an efficient writing scheme to LLM becomes essential since the extra check bits have to be updated for every update in cache. This problem could also be solved by only protecting dirty bits in a write-back scheme with extra check bits. Since the number of check bits stored in SRAM (i.e. within the core) is reduced, benefits show in energy consumption and area used.

For implementing higher levels of fault tolerance, this technique would fit the project. Because the number of (costly) SRAM cells available on an Field Programmable Gate Array (FPGA) is limited, a decrease of SRAM usage is preferable. However, the large delay induced by writing the modified check bits after every modification might be dramatic for the performance.

In [11] the direction is taken to propose a new EDAC code. The newly proposed SEC/DED code is based on Hsiao code [12], rather than defining a new class. As stated in Section 2.4.1.2, the number of ones in the PCM is proportional to the amount of hardware needed for encoding and decoding the codeword. The quantity of hardware needed to implement the encoder and decoder is in its turn proportional to the area overhead, latency and energy. The new code therefore minimizes the number of ones in the PCM even further.

Although for most standard sizes of data bits the number of ones were indeed reduced, the proposed code uses more check bits compared to traditional Hsiao code, as shown in Table 2.4. Related work is found in [36] which claims a successful implementation of this code and thereby proves that the decoder design is simpler than for Hsiao code. However, for 32 data bits and less the original Hsiao code uses a lower amount of gates (i.e. area), while for larger data arrays the new code is an improvement indeed.

Table 2.4: Comparison between proposed [11] and Hsiao code [12].

Data bits	No. check bits		No. ones	
	proposed	Hsiao	proposed	Hsiao
16	8	6	48	54
32	9	7	102	103
64	10	8	216	216
128	11	9	461	481

No other work using this code was found, which means this code is not attractive or mature enough to be applied. This may be due to the increased number of bits which is not easily justified by the relative small decrease of ones in the check matrix and therefore a decrease in en-/decoder logic. Also for the project it seems risky to choose for this code while the profits seem to be small. Therefore normal Hsiao stays the preferred code for SEC/DED protection.

As mentioned in Section 2.4.1.3, BCH code provides the ability of delivering high levels of fault tolerance. But implementing BCH en-/decoders is known to be hard and inefficient. Effort was made in [28] to improve the applicability of Double Error Correcting (DEC) BCH code by proposing a simpler implementation scheme. It includes aligning the word to block sizes of typical memory word widths and an implementation with minimized decoding latency is demonstrated.

Indeed the authors show a relatively simple way how to implement BCH en-/decoders. Compared to SEC/DED en-/decoders, the encoding is almost identical with respect to performance (latency) while decoding is considerably slower (between 55% and 69%). Still, this means a large improvement in the decoding latency compared to prior implementations. Furthermore it is shown that the encoding latency includes the generation of the syndrome, needed to detect an error. Therefore the DEC BCH code is almost as fast as SEC/DED in the case of no error occurring.

Drawbacks include the considerable increase in area overhead.Unfortunately, this is disadvantageous for the implementation in an FPGA considering its limited resource count. Also energy consumption will increase by increased area usage. However the efficient implementation of BCH code is still an interesting option for higher levels of fault tolerance while maintaining a low latency. For this thesis work it is therefore not applicable, but it may be for future work.

Energy is not one of the constraints set by the employer to optimize for, however since energy closely relates to area (and in some sense latency) the research performed in [13] is relevant for this thesis project. The trade-offs between (soft) error correcting capabilities ('reliability') and energy consumption/efficiency are examined in this paper.

An adaptive error coding scheme is proposed based on whether a block is present in the LLM or not. The relevant advantage of this for the project is the saving of access time by processing EDC instead of ECC for the blocks also present in LLM.

Another technique discussed in the paper is the early write-back scheme, which was referenced before in this report (see Section 2.2.2.5 and [18]). It decreases the number of dirty lines, which is more advantageous in this setting since ECC is only applied to dirty data.

Last technique proposed is physical bit interleaving which is put to the test by implementing it on an older technology running at a low voltage, hence being highly susceptible to soft errors. The results (see Table 2.5) are interesting since it tells that bit interleaving alleviates the burden of a more powerful error detection and -correction circuits.

Table 2.5: Effect of adding bit interleaving on MBUs [13]. The columns for multi-bit errors are left out since they were always zero. The numbers represent the occurrence of soft errors.

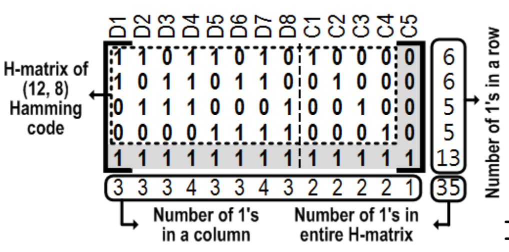
	No interleaving		With interleaving	
Error type	Single-bit	Two-bit	Single-bit	Two-bit
bzip2	716	9	734	0
gap	656	7	670	0
gcc	1246	20	1286	0
gzip	663	8	679	0
mcf	2232	32	2296	0
parser	789	10	809	0
perlbmk	1159	14	1187	0
twolf	1400	19	1438	0
vortex	1450	20	1490	0
vpr	767	10	787	0

Concluding this paper with regards to this thesis project, physical bit interleaving is proven to be (very) beneficial for the level of fault tolerance when combined with EDAC code. Furthermore the idea to only protect data with EDC when it also resides in the LLM is an interesting one which might be beneficial for the future addition of high protective fault tolerance. However, for SEC/DED capabilities belonging to low protective fault tolerance, it is not worth te effort since only one bit extra is needed to make SEC/DED from Double Error Detecting (DED) code. The early write-back scheme is not applicable for this thesis project since it requires a write-back scheme, where write-through (WT) is chosen for this project.

In [5] the main difference between extended Hamming SEC/DED code and Hsiao SEC/DED code is reviewed (see Section 2.4.1.2). Disadvantages of the Hsiao scheme compared to Hamming are the higher complexity of the Double Error Detection Signal (DEDS) generation circuit and extra input in the error locator. On the other hand, the Hamming scheme incurs a higher latency in check bit generation.

As a solution a check bit pre-computation method is proposed to solve the extra latency for Hamming check bit generation. However, an initial straight-forward solution is not valid for calculating the syndrome, hence it is assumed the check bit- and syndrome generation are combined. This is fixed by switching the base principle to Hsiao and adapt it in such a way the ECC circuit is simple as the extended Hamming code and the latency for computing check bits remains the same. In order to achieve this, extra ones have to be added to the PCM, but in contrast with the general assumption it decreases rather than increases hardware cost.

In Figure 2.8 up to Figure 2.13 the differences and modifications are visualized.


H-matrix of (12, 8) Hamming code
Number of 1's in a row
Number of 1's in a column
Number of 1's in entire H-matrix

	D1	D2	D3	D4	D5	D6	D7	D8	C1	C2	C3	C4	C5	
	1	1	0	1	1	0	1	0	1	0	0	0	0	6
	1	0	1	1	0	1	1	0	0	1	0	0	0	6
	0	1	1	1	0	0	0	1	0	0	1	0	0	5
	0	0	0	0	1	1	1	1	0	0	0	1	0	5
	1	1	1	1	1	1	1	1	1	1	1	1	1	13
	3	3	3	4	3	3	4	3	2	2	2	2	1	35

Figure 2.8: PCM of (13,8) extended Hamming SEC/DED code. [5]

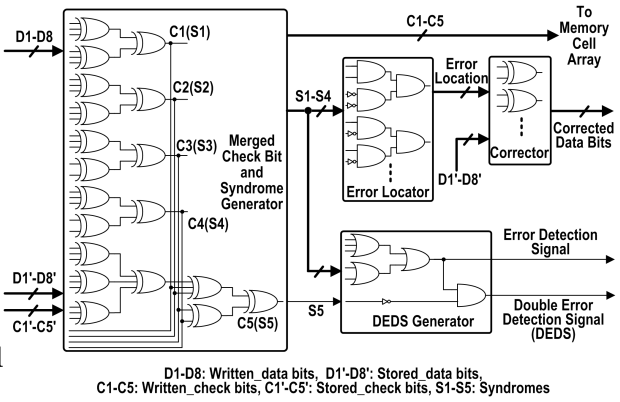
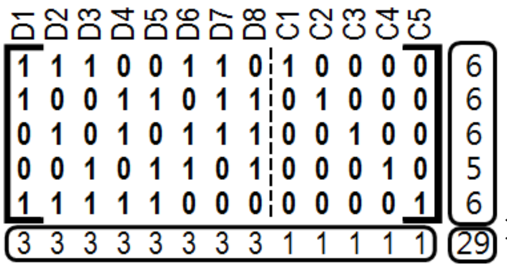

D1-D8
C1(S1)
C2(S2)
C3(S3)
Merged Check Bit and Syndrome Generator
C4(S4)
D1'-D8'
C1'-C5'
C5(S5)
C1-C5
To Memory Cell Array
S1-S4
Error Location
Corrector
Error Locator
D1'-D8'
Corrected Data Bits
Error Detection Signal
S5
DEDS Generator
Double Error Detection Signal (DEDS)
D1-D8: Written_data bits, D1'-D8': Stored_data bits,
C1-C5: Written_check bits, C1'-C5': Stored_check bits, S1-S5: Syndromes

Figure 2.9: EDAC circuit of (13,8) extended Hamming SEC/DED code. [5]



D1	D2	D3	D4	D5	D6	D7	D8	C1	C2	C3	C4	C5	
1	1	1	0	0	1	1	0	1	0	0	0	0	6
1	0	0	1	1	0	1	1	0	1	0	0	0	6
0	1	0	1	0	1	1	1	0	0	1	0	0	6
0	0	1	0	1	1	0	1	0	0	0	1	0	5
1	1	1	1	0	0	0	0	0	0	0	0	1	6
3	3	3	3	3	3	3	3	1	1	1	1	1	29

Figure 2.10: PCM of (13,8) Hsiao SEC/DED code. [5]

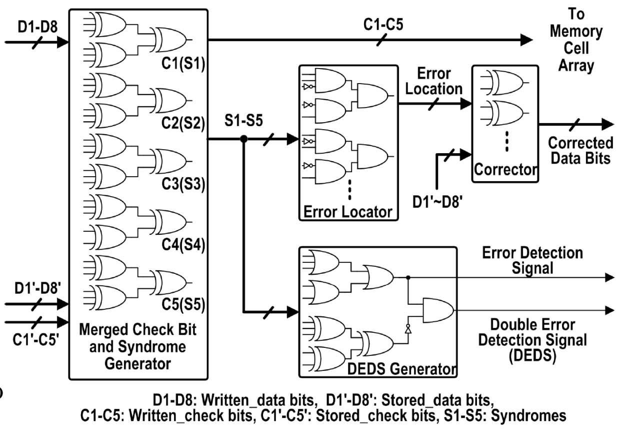
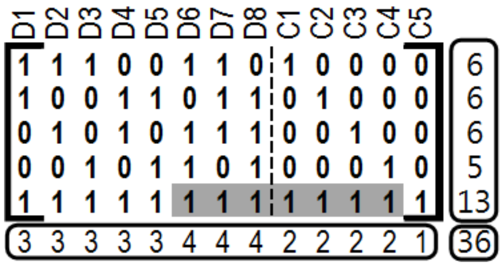

D1-D8
C1(S1)
C2(S2)
C3(S3)
C4(S4)
C5(S5)
D1'-D8'
C1'-C5'
Merged Check Bit and Syndrome Generator
C1-C5
To Memory Cell Array
S1-S5
Error Location
Corrector
Error Locator
D1'~D8'
Corrected Data Bits
Error Detection Signal
DEDS Generator
Double Error Detection Signal (DEDS)
D1-D8: Written_data bits, D1'-D8': Stored_data bits,
C1-C5: Written_check bits, C1'-C5': Stored_check bits, S1-S5: Syndromes

Figure 2.11: EDAC circuit of (13,8) Hsiao SEC/DED code. [5]

For the project it means yet another option for implementing a SEC/DED code. However, being an optimization for both extended Hamming and Hsiao code, it can also be added later on as an optimization to the EDAC, which is mainly Hsiao SEC/DED in this thesis project.

2.4.3 Conclusion

In this section, details of Hamming code, Hsiao code and BCH code are discussed which provide insight in their mutual differences, difficulties and possibilities. Additionally, various researches have been listed and discussed which propose new ways to provide EDAC or improve existing schemes.



D1	D2	D3	D4	D5	D6	D7	D8	C1	C2	C3	C4	C5	
1	1	1	0	0	1	1	0	1	0	0	0	0	6
1	0	0	1	1	0	1	1	0	1	0	0	0	6
0	1	0	1	0	1	1	1	0	0	1	0	0	6
0	0	1	0	1	1	0	1	0	0	0	1	0	5
1	1	1	1	1	1	1	1	1	1	1	1	1	13
3	3	3	3	3	4	4	4	2	2	2	2	1	36

Figure 2.12: PCM of (13,8) proposed SEC/DED code. [5]

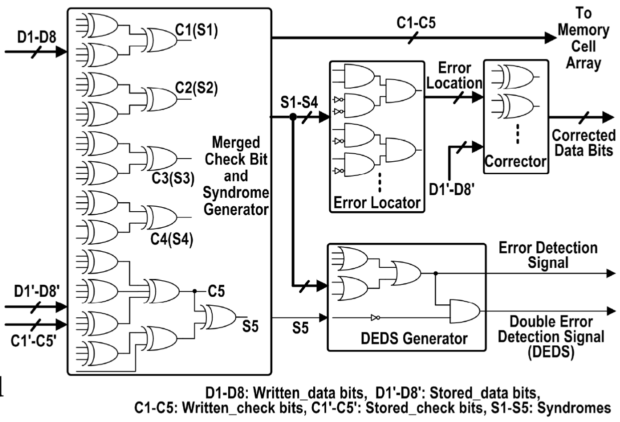

D1-D8
C1(S1)
C2(S2)
C3(S3)
C4(S4)
Merged Check Bit and Syndrome Generator
C5
S5
D1'-D8'
C1'-C5'
C1-C5
To Memory Cell Array
S1-S4
Error Location
Error Locator
D1'-D8'
Corrector
Corrected Data Bits
S5
DEDS Generator
Error Detection Signal
Double Error Detection Signal (DEDS)
D1-D8: Written_data bits, D1'-D8': Stored_data bits,
C1-C5: Written_check bits, C1'-C5': Stored_check bits, S1-S5: Syndromes

Figure 2.13: EDAC circuit of (13,8) proposed SEC/DED code. [5]

In general, the techniques discussed are promising but at the same time they invoke an increased complexity while the profits in general are not expected to be high. Although inspiring, the ideas mentioned in this section are not directly integrated into the design, but rather are kept open as optimization options. For instance, the paper reviewed last [5] proposes an optimization for Hsiao SEC/DED code, which decreases complexity for the DEDS circuit.

A short overview of suggestions found in the papers discussed is given in Table 2.6.

Table 2.6: Overview of proposed fault tolerance techniques (see Section 2.4.2).

Name technique and reference	Main goal
Per Set Protected (PSP) cache [35]	Reduce area and energy
Memory Mapped ECC [32]	Reduce check bits stored in SRAM
SEC/DED with minimal ones PCM [11]	Reduce hardware cost and latency
Parallel DEC code design [28]	Improve applicability BCH code
Data cache improvements [13]	Reduce energy consumption
Efficient SEC/DED code [5]	Reduce area overhead and latency

2.5 Techniques and parameters used in FPGAs

In this section the focus is on techniques and (cache) parameters specifically designed or optimized for application on Field Programmable Gate Arrays (FPGAs). Considering the characteristics of FPGAs it should be clear there will be differences compared to designing for an Application-Specific Integrated Circuit (ASIC). For instance, note that an FPGA has predefined Static RAM (SRAM) blocks fixing a certain lay-out and reduced options for implementation, while an ASIC design should be capable to contain an arbitrary size of SRAM cells in any kind of ordering. This and other differences are taken into account in the work cited in this section. First the relevant characteristics of the targeted FPGA for the project are discussed, second other literature found will be addressed.

2.5.1 IGLOO2: useful characteristics

Since the targeted FPGA is known from the beginning of the project, the design incorporates the techniques implemented in the device to obtain the most efficient solution for this device particularly. The device family selected is the IGLOO2 series by Microsemi [16] which features flash-based FPGAs. Flash-based FPGAs functionally reside between SRAM-based and anti-fuse-based FPGAs. It is not as fast and efficient as an SRAM-based FPGAs, making it inefficient to use for (partial) reconfiguration on the fly or fast prototyping. On the other hand the configuration is more resilient against radiation and other possible errors than SRAM, traded against an increased configuration time. Anti-fuse based FPGAs are not reconfigurable at all while providing the highest form of fault-resilience.

2.5.1.1 SRAM cells: two types and physically interleaved

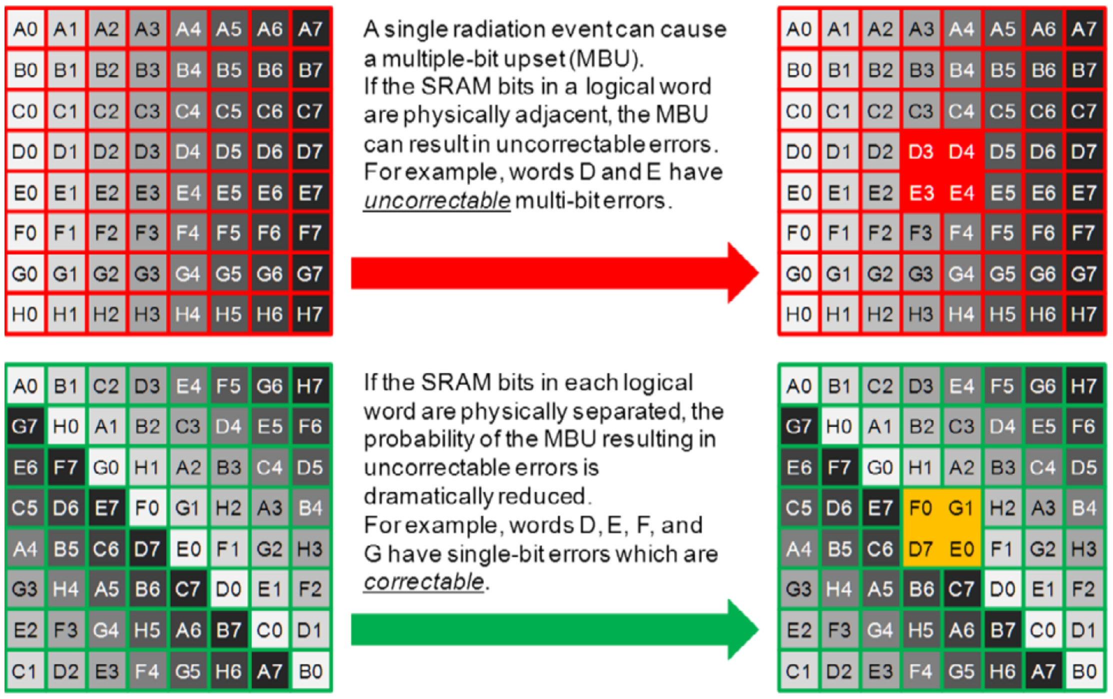
The configuration memory is flash-based, however SRAM cells are incorporated into the design to provide fast memory arrays to the designer. Two kinds of SRAM are distinguished, namely micro or μ-SRAM and large or L-SRAM. The main differences are the size, as could be expected based on their names, and the port configuration. μ-SRAM is configured as 64×18 arrays and incorporates two read ports and one write port, summing up to three (unidirectional) ports. 11 to 240 blocks of this type are incorporated for different models of the IGLOO2 family.

Similarly 10 to 236 blocks of L-SRAM are integrated. This type has two ports which can be either configured as two (bi-directional) read/write ports or two dedicated ports for writing and reading respectively. L-SRAM is configured as 1024×18 arrays. Total size of the two types of memory is a range from 12.672 to 276.480 cells (bits) for μ-SRAM and from 184.320 to 4.349.952 cells (bits) for L-SRAM. This means that for the largest IGLOO2 model 33.75 kB of μ-SRAM is available and 531 kB L-SRAM.

To improve the resilience against errors, the SRAM arrays are implemented as physically interleaved (see Section 2.3.1.4 for an introduction on this topic). In short, it means that adjacent cells/bits in codewords are physically placed apart from each other, as explained in the interim radiation summary on this product family [6]. This is illustrated and explained in Figure 2.14. The same survey claims that no multi-bit errors have occurred during any test performed by Microsemi, which is mainly due to the physical interleaving. Note that these claims correspond to several papers mentioning that physical interleaving is effective mitigating MBUs. The tests performed included radiation testing for both ground level and avionic applications.

2.5.1.2 DRAM controller

Another relevant feature of the IGLOO2 is the integrated Dynamic RAM (DRAM) controller which is able to interface either Low-Power DDR (LPDDR), Double Data


A single radiation event can cause a multiple-bit upset (MBU). If the SRAM bits in a logical word are physically adjacent, the MBU can result in uncorrectable errors. For example, words D and E have _uncorrectable_ multi-bit errors.

If the SRAM bits in each logical word are physically separated, the probability of the MBU resulting in uncorrectable errors is dramatically reduced. For example, words D, E, F, and G have single-bit errors which are _correctable_.

Figure 2.14: Illustration of mitigating MBUs by physical interleaving of memory cells [6].

Rate (DDR)2 or DDR3 DRAM. Microsemi differentiates two types: the so-called Fabric or F-DDR, as part of the High Performance Memory Subsystem (HPMS), and M-DDR controller. The presence of either is dependent on the model chosen. F-DDR is included in just a few of the models making M-DDR the more obvious choice to work with, although implementation details are not known. Differences between the two types are found mainly in the interfacing which will not be discussed here in more detail.

The bus width modes supported by the DRAM controllers also depend on the model, including the options ×8, ×9, ×16, ×18, ×32 and ×36. Both DRAM controllers incorporate the option to enable Single Error Correcting/Double Error Detecting (SEC/DED) code for data protection, regardless of the memory type attached to it. When this option is enabled the DRAM protection is assumed to be adequate, diminishing the need for additional protection. Note that Error Correcting Code (ECC) DDR memory has to be used in this case, which includes e.g. extra memory chips to store the check bits.

2.5.1.3 Internal buses

Just like other FPGA families, IGLOO2 contains a framework for standard buses defined by ARM, all falling under the Advanced Microcontroller Bus Architecture (AMBA) family. The buses are used throughout the structure of IGLOO2 as can be seen in Figure 2.15. It is assumed the application notes and configuration guides provided in the tab 'Documentation' on [16] provides all information needed to use the appropriate buses in implementation. Further explanation on the different kinds of AMBA buses will be given in Section 2.6.2.

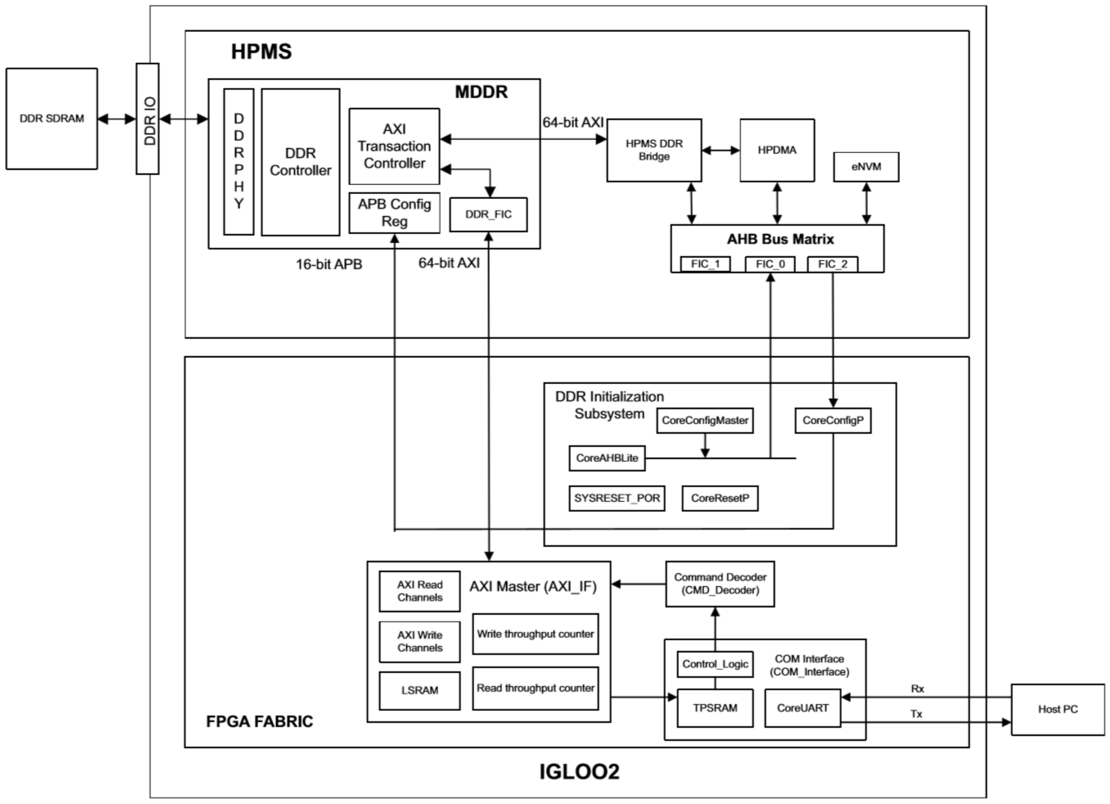


Figure 2.15: Top level view of the IGLOO2 architecture, including the AMBA buses AXI, APB and AHB-Lite. Taken from [7].

2.5.2 FPGA-related research

In [37] fault tolerance techniques seen earlier are discussed and compared for application in SRAM-based FPGAs. Since the fault tolerance discussed in this paper focuses on the different memories present in the FPGA, and the IGLOO2 uses SRAM arrays as well, the paper also applies to the current project.

An elaborate explanation is included on the options and their application on FPGAs, including parity, Triple Modular Redundancy (TMR), ECC and combinations of them. Scrubbing is also discussed and used in conjunction with all other options mentioned before. The techniques are compared based on the number of sensitive bits in a design (including the memory protection logic) and the number of critical failures i.e. Single Event Functional Interrupts (SEFIs). The cost function is chosen to be the area overhead, i.e. used slices and block RAMs, while performance is not considered.

Conclusion drawn is that TMR combined with scrubbing provides the highest grade of fault tolerance for memory. TMR is preferred over Error Detection And Correction (EDAC) because the latter always has a single point of failure on its input and output port signals, even when the EDAC circuitry is duplicated. The authors therefore implicitly state that TMR can be configured such that it does not contain a single point of failure, which is not proved. Hence TMR uses a voter, which easily introduces a single point of failure. Further, it is noted that there are often cheaper alternatives for TMR whose reliability is almost as good, e.g. Complement Duplicate (CD) with duplication (better known as Double Modular Redundancy (DMR)) or a SEC/DED scrubber with triplicated logic.

Although the paper extensively investigates the trade-off points, it has one major weakness when mapping to this thesis project. It does not consider physical interleaving. Probably because an efficient implementation in an FPGA was not feasible at that time. Nevertheless, the information given in the paper gives useful insight when high levels of fault tolerance in an FPGA implementation is required, which is not the main focus for this thesis project but is a candidate for future work.

The next paper is one of the several that elaborates on reconfigurable cache in FPGAs [14]. Although the IGLOO2 is not suitable to reconfiguration on-the-fly, the parameters used and benchmarked are interesting for the current design.

Both direct mapped and two-way set associativity are included in the reconfiguration options and the block size being one, two, four or eight words, summing up to eight options. Parameters chosen statically are a write-through (WT) policy, a Least Recently Used (LRU) replacement policy for two-way set associativity and a capacity of eight kB. These parameters are very similar to the ones chosen in this thesis project: WT, LRU for the data cache (two-way set associative) and a configurable capacity of 4, 8, 16 or 32 kB. The block size in this project is chosen to be four words (128 bits).

Testing with 2048 consecutive memory positions readings show a decreased miss rate for an increasing number of words per block. When randomly accessing 2048 memory positions, the configuration of one word per block has the worst hit rate, but for the other options no particular behavior was observed. For matrix product calculations the miss rate also decreases for increasing number of words per block, and no significant difference is appreciated between direct mapped and set associative.

For access time, also an important metric for cache performance, no actual tests were performed, although calculations show the difference for number of words per block.The calculations assume a fixed miss rate. Results show an increased transfer time for an increased number of blocks, but when the matrix product miss rates are incorporated (not shown in the paper) differences are small, see Table 2.7.

Concluding the paper, it indicates that little gain is to be expected when optimizing the parameters of block size and set associativity. Note that the cache configuration was very similar to the cache implemented in this thesis work. The paper also demonstrates that reconfigurable cache could be beneficial, although it is expensive in latency and resources (i.e. area). The authors refer to their continued work on this subject which is discussed next.

The next paper [38] continues the work of [14]. In [38], the reconfigurability is extended with four-way set-associative cache in addition to the direct mapped and two-way set-associative structures in [14] while remaining parameters are the same. Some modifications in implementation are applied, e.g. switching to a more efficient and complex bus structure.

Table 2.7: AMAT calculated using miss rate data from [14] for matrix multiplication. Hit time is assumed to be 1 ns, miss penalty being $10 + 2 \cdot$ (no. words per block) ns.

Words per block	Miss rate	Miss penalty [ns]	AMAT [ns]
1	0.030	12	1,360
2	0.026	14	1,364
4	0.021	18	1,378
8	0.012	26	1,312

The results in the paper show that an increased associativity and block size result in an increased maximum frequency and lower access times. Again, AMAT is not considered which makes this result questionable. Hence, how would an increased associativity improve the access time and maximum frequency while it probably complicates the design? This paper adds little useful information to [14] except for the contribution that four-way set associative cache in an FPGA may reduce mean access time.

An implementation description for a fully functional SEC/DED protection scheme in an FPGA based on Hsiao code is found in [39]. Hsiao code was chosen instead of Hamming due to its favorable recovery capability from multiple errors.

A (22,16,6) Hsiao code was implemented using an H-matrix constructed such that the column vectors corresponding to useful information bits all differ. The syndrome is calculated by recalculating the control bits from the data read from the cache, which is done using ten XOR gates divided in four levels. The recalculated control bits are subsequently compared with the fetched control bits by using the syndrome equations resulting in the so-called syndrome bits. The syndrome decoder reconstructs the data word using the syndrome bits, which is XOR'ed accordingly with the fetched data word. These operations yield the place(s) of an error, hence the error (in case of one error) can be corrected by flipping the corresponding bits.

This scheme was implemented on a Xilinx FPGA and the behavior under injected errors was verified. The paper provides a clear insight in the implementation of an actual Hsiao en-/decoder on an FPGA like the test platform of this thesis project. Therefore this work is used as a starting point for implementing the Hsiao SEC/DED en-/decoders.

A follow-up paper, written by the same authors, extends on the latter by comparing two implementations of the Hsiao coding on two different FPGAs [40]. Also it is claimed that a system using ECC protected memory is approximately 5% slower compared to one using parity protection. An improved Parity Check Matrix (PCM) was proposed for the same (22,16,6) Hsiao code resulting in slightly lower hardware cost. It resulted in one less XOR gate (total: nine) used for generating the control bits and fewer NOT gates needed for the syndrome decoder.

Concluding the two papers, they show that a Hsiao protected cache is well implementable in FPGA technology, even in dated FPGAs as used in these papers. Both propose a different PCM of which the latter seems more efficient, however both are (22,16,6) Hsiao codes with data words of just 16 bits which is not suited to the project.

In [41] an automatic cache generator (output in Verilog) for FPGAs is developed.

Tests are performed to examine the impact of several cache parameters on performance metrics and area. The targeted FPGA is an old model of Altera's Stratix family. The write policy is write-through, the word size set to 32 bits and the cache contains a single level, these are parameters similar to the ones chosen in this thesis project.Differentiated parameters are:

- Associativity: direct-mapped, two-way set associative with LRU policy and fully associative,
- Capacity: 128 up to 16K bytes,
- Address width: dependent on the required number of addressable blocks,
- Block size: 32, 64, 96, 128 and 256 bits,
- Read and write latency: resp. 2 and 3 clock cycles for full associativity, resp. 1 and 2 clock cycles for direct mapped and two-way set associativity.

For fully associative cache Content Adressable Memory (CAM) is used.The area occupied by logic elements increases linearly with cache capacity, not fitting in the chosen FPGA up from 8K bytes. Together with the minimum read latency of two clock cycles, which incurs a (very) low performance, the large size of the CAM in an FPGA supports the claim that CAM is hard to implement in an FPGA. Hence, based on the results in this paper, it can be concluded that CAM memory is too slow for constructing an Level 1 (L1) cache and takes too much area overhead per bit for constructing an Level 2 (L2) cache or single level cache. Since no other efficient way to implement fully associative caches is found, implementing full associativity is not an option for this thesis project.

Other results of the paper include the proportional increase of logic elements with the data and address width, which may be of lesser concern since modern FPGAs in general have plenty of logic elements. Performance metrics show that CAMs and comparators in particular slow down the caches at increasing size, hence only an increase in address width slows down the direct mapped and two-way set-associative cache structures. The paper shows that fully associative caches by using CAM in an FPGA for usable cache sizes is not tractable. Furthermore it shows that for direct mapped and set associative caches the address width is an important metric when considering performance in FPGAs.

As a conclusion of the paper applied to this thesis project, the address width may be of importance for the performance. However, the address width is predefined in the system and therefore this can be regarded as an optimization.

2.6 DRAM and buses/arbiters

In this section the Dynamic RAM (DRAM) and its interfacing will be reviewed.

2.6.1 DRAM technology

At the start of the project, the employer inspired also fault tolerance measures for the Lower Level Memory (LLM) which is the DRAM. However, in Section 2.5.1.2 it is discussed that the IGLOO2 DRAM controller provides an option to add Single Error Correcting/Double Error Detecting (SEC/DED) protection. Therefore this will be used as the basis for fault tolerant DRAM. Furthermore, the DRAM controllers are standard components which is prohibitive for adding fault tolerance.

In this section other aspects of DRAM fault tolerance will be discussed to show the state of the art and its possibilities. One of the main topics to clarify was requested by employer: whether there is a possibility to also protect the address - and command lines.

In [42] a Double Data Rate (DDR)4 design is proposed which incorporates Double Error Correcting (DEC) logic. Various properties of DDR4 are discussed and implemented, the latter being out of scope for this thesis project since off-the-shelf devices will be used for DRAM.

In general, DDR4 adopts a lower supply voltage to reduce power consumption compared to DDR3 while capable of an increased data rate without increasing the burst length. Furthermore it includes Cyclic Redundancy Check (CRC), Command-Address (CA) parity scheme and Data-Bus Inversion (DBI) as the most relevant (new) techniques for the purpose of fault tolerance. The first, CRC, is able to detect an error for the data being transferred, which is particularly useful when using standard memory without Error Correcting Code (ECC) support. The second, CA parity scheme, adds a parity bit to the combination of command and address in order to signal any single bit error. DBI is another technique which helps to improve the data signal integrity by signaling the DRAM whether received data has to be inverted or not before it is stored.

For this summary on DDR4 techniques an additional resource was consulted namely the main page for DDR standards by JEDEC [43].

In [44] however it is stated that CA parity was already incorporated in the DDR3 standard, and even in DDR2 as an optional feature.

More interesting statements arise in this paper, firstly by the research performed on DRAM modules showing a large difference in fault rates for different (anonymised) DRAM vendors. Permanent fault rates vary over a factor of 1.9 among vendors and transient faults vary over a factor of six, on average the varying factor is four which is significant. Also it is shown that not all DRAM modules experience more permanent than transient faults, opposed to the common assumption.

Another interesting topic is the effect of CA parity applied in DDR3 memory, being quantified in this paper. Regarding to the authors, a key feature of both DDR3 and DDR4 is the ability to add a parity check to the command and address bus, while in DDR2 this was optional. The rate of CA parity errors was compared to the number of detected but uncorrectable data errors: CA parity accounted for 75% in number of errors compared to the detected but uncorrected data from ECC. This means that adding CA parity detects a relatively large amount of errors which would otherwise be undetected and may harm the execution flow. Therefore CA parity is regarded a valuable addition to the DDR standard.

Third topic ought to be relevant for this thesis project is the effect of altitude on the fault rate. During an altitude difference of almost 2000 meters, the factors for Static RAM (SRAM) transient fault rates in Level 2 (L2) and Level 3 cache vary with factors of 2.3 and 3.4 respectively. This is mainly attributed to the increase in particle flux at increasing altitude.

For the project, the most prevalent information coming forward out of the two papers discussed is about the CA parity scheme present in DDR3 and 4 (and optionally in DDR2). This means that in this thesis project no attention will be paid to this aspect of making the DRAM fault tolerant. Another remarkable claim from [44] is that fault rates may vary significantly per module, which may be worth some research before the decision is made for a specific module.

2.6.2 Buses and arbiters

To add peripherals to the softcore which may access the memory structure, an interconnect is needed including arbitrating logic. It was found in [16] that IGLOO2 adopts a high-speed Advanced Microcontroller Bus Architecture (AMBA) bus either configured as 2×32-bit Advanced High-performance Bus (AHB)-Lite or 64-bit Advanced eXtensible Interface (AXI) bus. IGLOO2 also adopts the energy-efficient Advanced Peripheral Bus (APB) 3 bus which is meant for connecting multiple peripherals to the high-speed buses. The following sections will introduce the key concepts of the three types of buses, based on documentation from ARM which is the company behind the AMBA bus standards.

2.6.2.1 APB 3

The specification of APB 3 is found in [45], the introduction will be summarized here. The APB 3 standard is introduced in the AMBA 3 protocol family. It provides a low-cost interface which is optimized for minimal power consumption and reduced interface complexity compared to the other protocols within the family (e.g. AXI). APB interfaces the low-bandwidth peripherals which do not require high performance, therefore APB is not pipelined. Every transfer takes at least two cycles which illustrates the low performance. APB 3 is able to interface both AHB-Lite and AXI.

2.6.2.2 AXI

AXI in its original form is defined in [46]. It is designed for high performance systems working at high frequencies. Therefore it includes several features to make it suitable for high-speed interconnect. For instance, it is able to issue multiple outstanding addresses and out-of-order transaction completion. The protocol is burst-based and typical configuration includes multiple masters and slaves.

For AXI a revision was announced in 2011 known as AXI4 which involves minor changes in its possibilities and processing. The old AXI standard is renamed to AXI3. All differences and the full specification of AXI4 are reported in [47]. Since the modifications do not involve features needed for this thesis, they are irrelevant and no distinction between the two versions will be made. The implementation is compatible with both standards. IGLOO2 implements the old AXI(3) standard.

2.6.2.3 AHB-Lite

In [48] the specifications are found for AHB-Lite. It is positioned as a high-performance bus while eliminating some features of AXI in order to lower the complexity and power consumption. It supports just one bus master and includes support for burst transfers and single-clock edge operation. The common slaves are internal memory devices (e.g. cache structure) and external memory interfaces (e.g. DDR controllers). Low-bandwidth peripherals could be added as AHB-Lite slaves, however for system performance it is recommended to use the APB 3 standard for this. Bridging between the APB and AHB-Lite bus is done using a slave called APB bridge.

With the move from AXI3 to AXI4, the successor for AHB-Lite was added to the AXI standard known as AXI4-Lite. However this standard seems not to be supported by the IGLOO2 at the moment of writing.

2.6.2.4 Implementation in IGLOO2

Microsemi has summarized its implementation of AMBA buses for IGLOO2 in [49] which is called AHB Bus Matrix (ABM). It provides a non-blocking, AHB-Lite based multi-layer switch and supports four master and eight slave interfaces. The arbitration rules are adjustable in order to give certain peripherals a higher priority. An overview of the interconnecting systems in IGLOO2 was shown in Figure 2.15.

2.7 Fault tolerance verification

Functional testing has to be performed for any design and therefore multiple methods are found to do this efficiently. However, testing whether a design is truly fault tolerant within the requirements is another kind of testing not performed often. Errors need to be generated which is not trivial to achieve in common simulation/verification suits. Researches in this section propose ways to test the fault tolerance of the design by fault injection.

Two distinct techniques are proposed in [50] to study the effects of Single Event Upsets (SEUs) while executing a benchmark application. The techniques are called Coded Emulated Upset (CEU) and Field Programmable Gate Array (FPGA) Autonomous Emulation, which are recommended to consider as being complementary. By using them together all memory elements in the processor are accessible to inject faults.

For CEU a hardware implementation is needed and is used for fault injection in register files and cache memories. FPGA emulation requires a VHSIC Hardware Description Language (VHDL) description and allows fault injection for all processor flip-flops. Largest drawback of CEU is that it cannot access hidden flip-flops, but note these are not of interest for the current project. Besides the need of special hardware for CEU, another drawback is that certain memory locations can only be accessed in selected clock cycles. The emulation method in this paper works by replacing flip-flops for a structure capable of inducing faults, this method is very fast in comparison with CEU.

Next paper [51] is based on the previous one by improving on the emulation possibilities (CEU), optimizing for fast execution and thereby adding support for complex circuits like microprocessors. Another work shows that simulation-based fault injection is relatively slow in execution, supporting the choice for emulation-based fault injection.

Two techniques exist to perform fault injection during FPGA emulation: either by using reconfiguration, which takes considerable time, or by circuit instrumentation, which is the addition of some hardware components to provide external controllability and observability. The latter technique was already used in the previous paper for the Autonomous Emulation technique, however an improvement was implemented such that no interaction is needed between the emulation circuit and host computer. This is achieved by adding the required functionalities in the FPGA and speeds up the fault injection by three orders of magnitude.

A dual implementation is proposed: one is the Gate Level (GL) which provides very high accuracy (i.e. it has a small time quantum) and in which faults can be injected by means of instrumented gates. The other is Register Transfer Level (RTL) which executes a lot faster (i.e. larger time quantum of one clock cycle), in which the flip-flops are instrumented to load the state of GL implementation. The idea is to run the RTL implementation for default and switch to GL for fault injection during just one clock cycle. Thereafter it switches back to RTL implementation for a high performance while giving the fault a chance to expand.

The fault can either be:

- Silent: it disappears after a while,
- Failure: a wrong output occurs,
- Latent: the output is correct however the internal state is not the same as if there were no fault.

The platform used for validating and implementing this scheme was the LEON2 processor, which is comparable to the softcore used in this thesis project. It was configured as having 2K bytes of instruction cache, 2K bytes data cache and 256K bytes on-chip SRAM memory. A small Content Adressable Memory (CAM) memory is added to keep track of faults in the memories. This technique seems to be a fast and accurate way to validate the fault tolerance, however a large FPGA is required to fit in the system with its additional components.

2.8 Reduced Instruction Set Computer (RISC)-V softcore

This section will talk about the Reduced Instruction Set Computer (RISC)-V Instruction Set Architecture (ISA) in general and the implementation of the specific softcore used for this thesis project.

2.8.1 RISC-V ISA

The general design parameters of the RISC-V instruction set can be found in [52]. RISC-V stands for the fifth (Roman: V) version of an open ISA designed by UC Berkeley

[15]. It provides several standards including integer (I), multiply/divider (M), atomic read/modify/write (A) and single- (F) and double precision (D) floating point modules. These can be added to the base ISA which can be configured as a 32 (RV32), 64 (RV64) or 128 (RV128) bit system, all based on little endianness. In addition any other module could be added for which [52] provides the required information. The architecture is based on the load-store principle, which means all data to process needs to be loaded in registers first before they can be used and written back by a store.

2.8.2 State of the softcore

The softcore was initiated by Peter Verbrugge[1] during his graduation work (BSc, not published). He was followed up by another graduating BSc student, Daan Ravesteijn[2], who mainly re-factored the softcore and added some key features like interrupt handling. Also this work is not published.

The third student to work on this project was Wietse Heida[3] as a graduate project for his MSc Embedded Systems at TU Delft. His project is partially adjacent to this thesis project. He added fault tolerance for the functional part of the processor, i.e. the whole pipeline structure. Also he refactored the softcore another time. His work can be found on the TU Delft repository [8]. This thesis project therefore is the fourth in row working on the same softcore, however is the first which is not directly related to the basic working of the softcore itself.

Due to the fact no modifications are made to the working of the softcore, details of the softcore are not relevant and will not be discussed here. The structure to connect the cache and Lower Level Memory (LLM) to will become apparent in Chapter 3.

The core is based on the RV32I ISA and implements a (classical) five-stage pipeline inducing the following stages:

- Instruction fetch (IF),
- Instruction decode (ID),
- Execute (EX),
- Memory access (MEM),
- Writeback (WB).

2.9 Summary and conclusion: literature survey

In this section a summary follows for the literature study as described in this chapter. This will be a coarse overview since conclusions are already to be found in the corresponding sections and even per paper or set of papers reviewed. In this continuous processing of gathered information, the requirements set in Chapter 1 are taken into

[1]Peter.Verbrugge@Technolution.nl
[2]Daan.Ravesteijn@Technolution.nl
[3]Wietse.Heida@Technolution.nl

account. This is done in order to let the literature survey serve as a starting point for the design strategy which can be found in Chapter 3.

First the cache parameters will be regarded, second the fault tolerance and third the three remaining topics, namely fault injection, Dynamic RAM (DRAM) and buses/arbiters.

2.9.1 Cache parameters

In Section 2.2 all relevant cache parameters are introduced and specified to some extent to this thesis project. Note that the requirements set in Chapter 1 are partially originating from the knowledge gathered in this survey. For instance, at the start of the project, employer suggested to implement two levels of cache. Although in Application-Specific Integrated Circuits (ASICs) this would be the logical choice, in Field Programmable Gate Arrays (FPGAs) there is barely any motivation for this scheme. Other requirements are verified to be the right decisions. For instance, employer also devised a split cache. According to literature this is indeed the best option, incurring both the instruction- and data cache.

Other considerations for deciding other parameters of the caches are mainly found in literature aimed towards FPGA implementations. In Table 2.7, compiled using results from [14], it is seen that the block size does not have a large effect on performance in the configuration that was tested. Also the level of associativity does not seem to make a large difference as concluded from the same paper and its follow-up paper [38]. Main contribution from [41] is that address width can play an important role in cache performance on an FPGA, although this is a fixed parameter for this thesis project.

2.9.2 Fault tolerance

To introduce fault tolerance in the cache structure, the survey is split into two parts. First describes the basic techniques and reviews a number of papers referencing and comparing them, as well as the motivation behind the fault tolerance: Single Event Effects (SEEs). Second provides more details for the most prevalent techniques and also searches for improvements or complementary techniques to achieve the same result in a more efficient way.

2.9.2.1 Bases of fault tolerance

The two main drivers to add fault tolerance are natural causes e.g. by (increased) radiation and human attacks. Since the second is less predictive and requires high levels of fault tolerance, this thesis and survey mainly focuses on the first: SEE due to natural causes. IGLOO2 is used as the reference FPGA being a model specifically designed for environments with increased radiation or high security. Therefore the information provided by Microsemi (manufacturer of IGLOO2) is essential, in particular the claim that no Multiple Bit Upset (MBU) occurs at high levels of radiation [6]. This eliminates the need for high levels of fault tolerance to protect against multi-bit errors.

The two bases to add fault tolerance in memory systems are N-way Modular Redundancy (NMR) and Error Detecting Code (EDC)/Error Correcting Code (ECC)). The first is inefficient with respect to area and therefore it is not recommended for cache implementations.Error Detection And Correction (EDAC) code exists in many forms of which the most apparent are parity code and Single Error Correcting/Double Error Detecting (SEC/DED) code. Parity is only suitable for situations requiring a low level of fault tolerance and/or application on small parts of data. SEC/DED is able to correct one error or detect two, which makes a good combination with one property of the IGLOO2: physical bit interleaving in the Static RAM (SRAM) blocks.

Subsequently, many papers point towards Hsiao code being the most efficient SEC/DED code to implement. Bose-Chaudhuri-Hocquenghem (BCH) codes are suited to add higher levels of fault tolerance, e.g. by constructing an Double Error Correcting/Triple Error Detecting (DEC/TED) BCH code. However this coding scheme requires a significant increase in complexity of en-/decoders while higher levels of fault tolerance always require a larger overhead in bits and latency.

2.9.2.2 Advanced fault tolerance

Many papers are reviewed for this second part about fault tolerance. Besides a deeper understanding of the most prevalent EDAC schemes, new techniques and improvements are revealed. However, most of them seem not to be suited for a prompt implementation in this thesis project due to their complexity and/or scarce benefits. An overview is given in Table 2.6 while in Section 6.2 this list is reviewed for future improvements.

2.9.3 Remaining topics

Since relevant research appears to be scarce for the last three topics, these are therefore combined into one section. DRAM, buses/arbiters and fault tolerance testing are concluded in succession.

2.9.3.1 DRAM

The built-in features of IGLOO2 and Double Data Rate (DDR)3 will be used for making the DRAM fault tolerant. IGLOO2 includes the option to enable SEC/DED protection for the data [16] while DDR3 includes Command-Address (CA) parity [44]. A higher level of fault tolerance for DRAM is left out of consideration.

2.9.3.2 Arbitration/buses

No new techniques are considered for the topic of arbitration and buses, but the standard for high-performance buses is explored being the Advanced Microcontroller Bus Architecture (AMBA) family of ARM. Advanced Peripheral Bus (APB) is suited for connecting peripherals while Advanced eXtensible Interface (AXI) is the standard to choose for memory transactions. The IGLOO2 family of FPGAs supports these buses by Intellectual Property (IP), just as the FPGA of the development kit. Therefore, an interconnect including arbitration is easily generated by the software of the FPGA vendor.

2.9.3.3 Testing

While several options exist for fault injection, only one seems to be feasible for a softcore on an FPGA.The emulation technique described in [51] seems to be a workable solution. Therefore it is recommended to use the emulation technique to verify the fault tolerance of the cache structure and softcore on the FPGA. However, this incurs a lot of work which means it is not feasible to integrate the technique into this thesis project. Therefore simple techniques will be applied like hard-coded saboteurs. Emulation may be performed in future work.

3 Design considerations

The goal of this chapter is to provide and explain the design strategy used to establish the cache structure for the RISC-V softcore. Basis for the proposed design strategy are both the requirements set in Chapter 1 and the literature study found in Chapter 2. The design considerations in this chapter primarily address the configurability options of the cache in both capacity and level of fault tolerance.

Section 3.1 describes the specific requirements taken in consideration, mainly dealing with fault tolerance and scalability of the cache structure. It includes specifications which come forward from the choices made in order to implement the cache structure and therefore is an addition to Chapter 1. Secondly in Section 3.2 the design is discussed, mainly based on the preceding section. In Section 3.3 an overview of the system and the cache structure is given, as well as several diagrams to further explain the cache structure design. Note that the overview and part of the diagrams are placed in appendices A and B. Last part, found in Section 3.4, summarizes the findings and concludes this chapter.

3.1 Goals and requirements for the configurable fault tolerant cache: revisited

In Chapter 1 and Section 2.1 the base requirements and some details are given. However, during the design and implementing phases, several requirements were added, modified or further specified by employer. Therefore the requirements are revisited in this section, starting with the access of single bytes, followed by configurability for both capacity and fault tolerance. After this, considerations on fault tolerance will follow and the section ends by discussing the relevant considerations regarding Dynamic RAM (DRAM), buses and test methods.

3.1.1 Enabling byte accesses for data

At the initial refinement of requirements, it was not yet clear that some instructions only use one byte (8 bits) or half-words (16 bits) of data. Due to the new insight, per-word (32 bits) protection schemes are not an option for the data cache. A per-word protection scheme would incur a Read-Modify-Write (RMW) sequence in case just a part of the word is changed. Since an RMW sequence is costly in latency and also complexity (inducing increased area), a per-byte protection scheme is chosen for implementation. Note that this dilemma only applies to the data cache, since instructions cannot be written by the softcore, i.e. instruction cache is a read-only memory to the softcore. In conclusion, the fault tolerance measures for the data cache are byte-based instead of word-based to prevent losing performance when a byte or half-word is written. The instruction cache on the other hand remains word-based.

3.1.2 Configurability: capacity and fault tolerance

Employer envisions a configurable cache structure to optimize for several applications since these are not known beforehand. In Chapter 1 it is not yet defined clearly in what extent it has to be configurable. Therefore this part clarifies on this behalf.

3.1.2.1 Configurable capacity

First scalable aspect of the cache is its capacity. Not only in order to facilitate various applications with their own requirements, but also to be able to place multiple instances of the softcore including their cache structures on a single Field Programmable Gate Array (FPGA). Since the applications are not known during this thesis project, it is hard to decide any lower or upper limit with respect to that. For instance, cache trashing for relatively small caches, which has to be prevented from occurring, is dependent on the program to run. Therefore a lower limit is not easy to decide upon in a qualitative way. Employer has set this minimum to 4 kB which should be adequate to prevent cache trashing in standard applications. An upper limit could be determined by the maximum amount of data fitting in the L-SRAM of the largest type of IGLOO2. However employer decided to set the maximum statically to 32 kB. Furthermore the restriction is adopted that the amount of data bits has to be a power of two. This results in the configuration options for capacity being either 4, 8, 16 or 32 kB for both the instruction and data cache.

3.1.2.2 Configurable level of fault tolerance

Secondly, the level of fault tolerance for the cache structure can be made scalable. Note that for any level of fault tolerance, two schemes have to be determined: one for word-based or parts close to this size (such as tags), and one for byte-based. When the same scheme is applied for both, in reality the byte exhibits a higher protection level than the word and the relative overhead is much larger.

As explained in the literature survey (see Section 2.3.1.1), natural causes are mainly Single Event Upsets (SEUs) while human attacks normally do more damage at once, including Multiple Bit Upsets (MBUs). However, in the latter situation of human attacks it is acceptable to just stop the execution, i.e. only detect possible attacks. In that case, there is no need for Error Correcting Code (ECC) but instead the focus is totally on Error Detecting Code (EDC). Note that detection and correction can be traded against each other in any Error Detection And Correction (EDAC) code.

Employer assesses the situation of human attacks covered when deploying quadruple or quintuple error detection per word, which means a minimal hamming distance of five or six (generally used for Double Error Correcting/Triple Error Detecting (DEC/TED) code) respectively. For byte protection Triple Error Detecting (TED) code is considered as a comparable level of fault tolerance. As mentioned before, TED can be configured from Single Error Correcting/Double Error Detecting (SEC/DED) (Hsiao) code.

The option of this so-called high level of fault tolerance, or heavy protection, is left out of the implementation. This is due to lack of time and the difficulties expected when implementing the complex ECC schemes needed. However, for a future addition of this option to the cache, this knowledge may be very useful. Therefore, it is included in this report up to the design phase.

The other cause of Single Event Effects (SEEs) is natural. This level of fault tolerance is referenced as low or light protective. The occurring errors for this category are of a different kind. Additionally, in case of an SEU, the softcore should be able to continue execution correctly. This incurs the need of ECC in addition to (a weaker form of) EDC. Minimum level of error protection set by employer is SEC/DED protection per word, which means a hamming distance of four, which can be provided by e.g. extended Hamming code or Hsiao code.

Since SEC/DED (or just Single Error Correcting (SEC)) code applied on single bytes is already considered high protective and incurs a relatively high overhead, parity is taken to serve as EDC per byte. Since EDC cannot correct an error, the only valid option to correct the detected errors is using a write-through (WT) policy (see Section 2.2.2.5). This means that there is always a copy of all entries in the Lower Level Memory (LLM), which is assumed to be protected by ECC.

In addition to high and low security as just described, an extra option for an intermediate level of fault tolerance is proposed. Also this scheme is not implemented due to limitations in time, but included in the report since it may turn out useful for future modifications.

Note that the quantuple error detection needed for the high level of security can be easily adjusted to serve as DEC/TED code since the required Hamming distance is the same. This means that little effort has to be put in this solution when the high level of security is successfully implemented.

3.1.3 Limiting factors for the fault tolerant cache: timing and capacity

For all three levels (low, high and intermediate) of fault tolerance mentioned above, a more specific design is described next in this report. However, first the optimization goals and requirements are discussed needed to achieve an appropriate design strategy. Two possibly limiting factors are identified being the timing constraint set by employer and the capacity in the FPGA. The latter is implicitly set by the employer by choosing a specific FPGA family.

3.1.3.1 Timing

As is mentioned several times before in this report, one clock cycle hit time is one of the main constraints. This means that latency is an important aspect to take into account for the cache structure as a total. However, in case of a correction this requirement may be released since this event is expected to be rare and does not impact performance in a run without errors.

In agreement with employer, the constraint is set that error checking/detection must happen within a single clock cycle. This means that check bits needed to detect errors must always be included into the cache structure for fast accessing. Note that fetching from DRAM always takes multiple cycles and hence is no option for error detection. On the other hand, check bits needed for adding error correction capabilities do not necessarily need to be included into the cache. Since DRAM capacity is probably much larger than needed by programs run on the system, DRAM can be used to store additional check bits.

The latter idea originates from [32] and is applicable for this thesis project, although architecture and context differ. In case of a correctable error, the extra check bits stored in the DRAM are fetched and used to correct the erroneous word. This implies storing extra check bits on every write. In [32] this is incurred only for writes from the processor which makes an entry dirty (write-back), but in this project WT is chosen, translating to storing extra check bits for every single entry. Also, the method applied in the paper incurs an uncertainty factor, which has to be solved by worst-case scenario thinking. Then, the approach would come close to the alternative solution of storing extra code for every entry, so also coming from the LLM during reads and not only those (over)written by the core.

Looking at the proposed solutions, this technique is only applicable to the extra option being DEC/TED. The low level of fault tolerance providing SEC/DED does not favor this scheme since Double Error Detecting (DED) needs just one bit less than SEC/DED protection. Hence, storing one bit in the DRAM to fetch when a single error occurred is not efficient with respect to resource usage. The high security solution on the other hand, providing Quintuple Error Detection (5-ED) which is detection-only, the optimization can also not be applied. This is because all detection has to be done for all accesses, which makes the cache useless when (part of) the check code is in the DRAM. The only option which favors this approach is the third option: intermediate. TED can be obtained by from a typical SEC/DED code which is then stored in the cache. The Double Error Correcting (DEC) capability can be added by an additional SEC/DED code, which may be stored in the LLM. Together they provide DEC/TED with only the overhead of TED within the cache.

3.1.3.2 Capacity

Another aspect that may be restrictive while designing the cache structure is its capacity within the FPGA. In this thesis project it may be limited by the amount of L-Static RAM (SRAM) devices incorporated in a specific IGLOO2. This type of SRAM is needed for implementing the cache in IGLOO2 because the other type available (μ-SRAM) is too small in total capacity. Other drawbacks are its additional latency for both reading and writing [53] and the restrictive port configuration (two read ports and one write port).

Ideally, the cache should use its space efficiently, occupying no more L-SRAM devices than needed. For small caches in particular this becomes challenging since an L-SRAM device on its own is not configurable and stores 2.25 kB (or 18 kb). In order to employ these memory quantities as efficient as possible, there might be an optimization step following on a final implementation, however this is regarded future work.

The smallest IGLOO2 device features eleven blocks of L-SRAM, summing up to 24.75 kB in total [49]. Although the smallest cache configuration features just 16 kB of memory in total, the implementation probably does not fit in this smallest device due to the extra capacity needed to store tags and check bits. However, note that this smallest device may be disregarded due to other missing features. The second model in the IGLOO2 range already features 21 L-SRAMs for a total of 47.25 kB which is assumed to be adequate to fit at least the smallest configurations.

3.1.4 DRAM: no extra fault tolerance

Another initial goal proposed by the employer is increasing the level of fault tolerance for DRAM. As a starting point, the memory controller inside the IGLOO2 (MDDR) incorporates SEC/DED protection by default, as discussed in the literature survey (see Section 2.5.1.2). Since the cache structure as implemented in this thesis project accounts for a considerable amount of work, it is decided to leave this assessment out of this thesis project. This means that the fault tolerance for DRAM is kept on the default level as provided by Microsemi. A follow-up project may aim for extra protection in DRAM by implementing for instance Double Modular Redundancy (DMR) by connecting two DRAMs and check their output against each other.

3.1.5 Number of additional bits needed

The requirement to use the cache efficiently incurs considerations about the additional bits needed for proper cache operation. Here, initial calculations are applied while in Chapter 4, describing the implementation details, the exact figures for the final version are presented. The tag accounts for the main part of the additional bits, which quantity is defined by the following equation:

$$\text{No. tag bits} = \text{no. address bits} - \text{no. offset bits} - \text{no. index bits}$$

The number of address bits is determined by the amount of memory needed to be addressed. However, since various applications are expected to run on the softcore, this may range from a minimum of about 32 MB up to 4 GB for specific (video) applications. In practice the current implementation of the System on Chip (SoC) uses 32 bits for addressing bytes, meaning that it is able to address 4 GB of external memory (byte addressed). To keep the design consistent, also the cache structure has to work with these 32 bits. However, since the last two bits for byte access are not used (byte enable signals are used instead), effectively 30 address bits are relevant.

Since the size of the cache is relatively small (8 to 32 kB) compared to the addressable DRAM range (4 GB), the overhead induced by the tag memory per block becomes relatively large. Due to the block size of four words (128 bits) there are two bits for offset, decreasing the number of index bits at unchanged cache size. Note that the number of offset bits is defined as being $\log_2(\frac{\text{block size}}{\text{word size}})$ and the number of index bits being $\log_2(\frac{\text{no. blocks}}{\text{associativity}})$. Hence the index bits for instruction cache (being direct mapped) counts one more than for data cache (being two-way set associative) when they adopt the same number of blocks. Note that the tag also needs to be protected since even a single error may result in chaotic execution. Since the tag size differentiates for varying cache sizes, a range of widths has to be supported by the en-/decoders.

3.2 Design description for the cache structure

In this part more details are provided of the final design of the fault tolerant and configurable cache structure. Firstly the light-weight protection scheme is discussed, followed by the measures to take to obtain a high level of fault tolerance. Third the additional option for intermediate protection level is explained. Note that the latter two are not included in the implementation phase. Fourth section discusses the implications on cache parameters. Concluding part considers DRAM, buses and test methods.

3.2.1 Light-weight protection: parity and SEC/DED code

Light-weight protection in this thesis project is defined as the potential to detect and subsequently correct one error or to detect two concurrent errors per word. First the chosen technique, able to provide this protection scheme, is reviewed and supported. Then, the case of byte accesses is discussed, which needs another approach. Finally the selected solutions are summarized.

3.2.1.1 SEC/DED scheme for protection per word: Hsiao code

In the first use-case protecting against SEUs, SEC/DED code protection per word is requested and implemented in this thesis project. In order to detect up to two errors, standard Hamming code can be used, however extended Hamming code (providing SEC/DED) just adds one extra check bit incurring negligible area overhead. In the case of applying SEC/DED code, Hsiao SEC/DED code is more efficient than extended Hamming code. This holds both in terms of hardware resources and latency, while the number of check bits remains the same [12].

An alternative for light-weight error detection is to add one or more parity bits per word. However parity protection cannot provide the same level of error coverage as DED code, except when for any single bit a parity bit is added (making it similar to DMR). For example, when two bits are erroneous, an appended parity bit will not report any error. DED code detects any possible combination of a double error.

Although parity may be a cheaper option with respect to area and decoding latency, employer assesses parity per word as too weak for the envisioned applications. When the physical interleaving is considered which is present in the L-SRAM devices in IGLOO2, one parity bit per half-word (16 bit) should be adequate to provide light-weight error protection. Note that the L-SRAM blocks in IGLOO2 are 18 bits wide, therefore 16 bits are expected to be stored on one row within this block. Since Microsemi claims no multi-bit errors occur in a single L-SRAM it is assumed parity protection is adequate. Due to these considerations, for byte protection in the data cache employer agrees on using parity bits to keep the overhead small.

3.2.1.2 Correction in case of a parity error

In order to protect the data bytes, which are independently accessible, one parity bit per byte is used as substantiated directly above. However, in this scheme a method must be added for correcting a single error since the ability to correct is one of the constraints for this level of fault tolerance.

One option is adding SEC code, e.g. per word. This is not an elegant solution since it would still incur RMW operations for any change in single bytes.

The only feasible and efficient solution found is adopting a WT policy instead of the commonly applied write-back scheme. In this situation all data to modify is also written directly in the LLM, which may be seen as a back-up instance. The main disadvantage for this solution is that every store operation initiated by the processor incurs a write to LLM, which means a considerable delay. To minimize the performance impact of this, a FIFO buffer is included: every store will be fed to the buffer after which the processor may continue execution. This incurs another potential problem, since the update of a certain data could remain in the buffer for a longer time e.g. when the bus remains occupied for a while. Then, if a correction is inferred, it may be replaced by an old element from DRAM while the updated instance still resides in the buffer.

Various solutions to this problem are examined, like adding a dirty bit (like in the write-back policy) indicating whether a byte or word is being changed since the last load from LLM. However this is a rather complex solution to implement efficiently, questions arise like when to reset the dirty bit without incurring extra latency? An alternative solution is to assign priority to writes over reads. Since the DRAM controller processes one request at a given time, it makes no sense to send multiple requests at the same time, which means the priority has to be assigned within the cache structure. In short it means that when a write operation to LLM has started, no read can interrupt it. Also, when multiple write requests are queued in the write buffer, any read request must wait until all writes are completed, signaled by the write buffer being empty. This can be done by storing the read request and not giving an acknowledgment until the read has proceeded. Thereby, the softcore cannot continue execution, hence no other request can be made.

Note that when the requested contents for a read are available in the cache (i.e. a hit occurs), no access to LLM is needed and therefore execution proceeds, even when a write is being processed from the write buffer.

3.2.1.3 Summary low security: Hsiao, parity and write-through

Taking the above mentioned considerations into account, the options chosen for this project are the following. For quantities of bits larger than a byte, Hsiao SEC/DED code is selected, up to the word size of 32 bits. Thereby including all tag widths, tags appended with a valid bit in the instruction cache and the 16 valid bits in the data cache. SEC/DED code is able to detect up to two errors or correct one, just as the name suggests. The check bits will be stored inside the cache structure (i.e. not in the LLM).

For the data cache, needed to be byte accessible, one parity bit is applied per byte as an alternative to the SEC/DED code per word. In order to correct an error found by the parity bits, a WT policy is adopted together with a First In First Out (FIFO)-buffer to mitigate the performance loss associated with this scheme. When an error is detected, the correcting logic may take one or more additional clock cycles in order to correct the error if possible.

3.2.2 High level of protection: TED and 5-ED code

For a high level of protection, only the detection capabilities need to be extended since correction is not required. Note that all detection measures need to be stored inside the cache structure. Hence, extra check bits in DRAM are only helpful for correcting any detected but uncorrectable error, while the high level of fault tolerance does not include correction at all. This means the full bit overhead of the high level of fault tolerance measure are inside the cache. Also it is expected that the detection in one clock cycle will become a constraint which is hard to comply with when more advanced codes are applied. Hence, complex decoders are generally needed for high detection capabilities, especially when implemented by Bose-Chaudhuri-Hocquenghem (BCH) codes like in [54].

An alternative was found at [55], lowering the en-/decoding latency by using two complementary Hsiao-code-like SEC/DED codes together providing DEC/TED code. A disadvantage is that this technique requires 16 check bits for 32 data bits incurring a relatively large overhead in bit storage. Still this code is the recommended approach for protecting 32-bit instructions and tags since it seems to have the highest chance to satisfy the constraint of one clock cycle hit time. This code subsequently can be configured to provide 5-ED. For bytes inside the data cache Hsiao code will be used again, but now configured as TED code. In this way the codes used for SEC/DED code can be re-used for TED, minimizing the additional implementation effort.

3.2.3 Intermediate solution: DEC/TED code

The technique proposed before, namely storing check bits for detection in cache and for correction in DRAM, could not efficiently be integrated for both low and high level of protection. However, it might prove to be very beneficial in the intermediate solution providing DEC/TED. For a high level of protection an alternative is recommended using two complementary SEC/DED codes [55]. By storing the first seven bits in cache structure, up to three errors can be detected (TED) with relatively simple hardware. This is because it is very similar to the Hsiao code in the light-weight protection solution. By adding the other nine bits, which can be stored in DRAM since they are only needed for correction, DEC/TED protection is realized while the bit overhead in the cache is minimized.

3.2.4 Cache parameters: associativity and block size

Conventional cache parameters such as associativity, block size and capacity are best optimized by a design space exploration for a given application. However, the application is not known during this thesis project and therefore the concept of configurability is applied for the capacities of the caches, enabling the possibility of optimization afterwards. Another approach is to keep the design simple or to use best practices. In this thesis project, the architecture overall is kept relatively simple while the capacity is configurable.

The basic parameters include the block size of 128 bits which is equal to four times the word size. Reason for this is that the typical configuration for the DRAM controller involves 128-bit wide burst size for highest efficiency. Furthermore, the instruction cache is direct mapped and data cache two-way set-associative. The cache exists out of a single level and the data cache adopts a WT policy, incurred by the parity protection per byte in data cache as explained before in Section 3.2.1.2. The instruction cache is read-only and therefore does not employ any write-back policy.

For the 128 bit block size, it holds that the cache both profits from temporal and spatial locality, but also suffers in performance when spatial locality is (nearly) absent. In the latter case data is stored which is not used before being replaced, which is a waste of storage space. Furthermore both instruction and data cache are assumed to reside in L-SRAM on an IGLOO2 FPGA. Also they adopt the same level of EDAC, i.e. the low level of fault tolerance is selected to implement for both caches, each in their own way as described before. Instruction and data cache do not need to have equal capacity and are enabled to be scaled independently. In general, instruction cache has equal or smaller capacity than data cache, but the decision to scale them is left to the designer optimizing for a specific application.

The minimum size for both caches is set to four kilobytes (kB), as adduced by employer. The largest supported capacity is 32 kB. This was verified to be feasible in IGLOO2, although the smallest version of IGLOO2 may not be able to fit in any configuration of the cache structure. The cache structure supports all sizes with a power of two within the range mentioned. The configuration options are summarized in Table 3.1, including the high level of protection proposed to add to the cache structure in future work.

Table 3.1: Summary on configuration options proposed for the fault tolerant configurable cache structure. Note that only the lightweight protection scheme is implemented in this thesis project.

Type of data	Instructions; tags with valid bits		Data	
Data quantity	20-32 bits		8 bits	
Protection level	Low	High	Low	High
EDAC	SEC/DED	5-ED	SEC	TED
Technique	Hsiao [12]	Double SEC/DED [55]	Parity (+ WT)	Hsiao [12]
Capacity [kB]	4 - 32			

3.2.5 Design considerations for DRAM, buses and test methods

Although the focus is mainly on the cache structure, this thesis project also incorporates other objectives which are addressed in this section. First the DRAM structure will be discussed briefly, subsequently the topic of buses is revisited and lastly the test method.

3.2.5.1 DRAM protection

As discussed in Section 3.1.3, the default protection scheme is used for this thesis project in terms of DRAM protection. This means that the SEC/DED option of the MDDR subsystem in IGLOO2 must be enabled when it will be generated. In this thesis project it is not enhanced by another functional block to increase the level of fault tolerance. Note that DDR-3 memory includes also Command-Address (CA) parity to protect the command and address bits, using this type of DRAM will therefore also enhance fault tolerance without any additional work. Note that another FPGA is used for prototyping which does not have the option of SEC/DED protection in the DRAM controller, therefore the porting to IGLOO2 is left as future work.

3.2.5.2 Errors in buses

The set of buses present in the IGLOO2 is discussed in the literature study (see Section 2.6.2). For this thesis project these readily available buses are integrated. Extra protection is not needed since the probabilities for errors in a bus are very small and the data inhibits EDAC measures already. Therefore, if any error occurs in the bus, this is corrected when the data is checked at the next access.

3.2.5.3 Test methods

Another part of the literature study (see Section 2.7) indicates that testing the fault tolerance of a cache memory in FPGAs is not as straight-forward as for combinatorial logic. The only way which seems feasible according to the scarce resources found is emulation [51]. This method is expected to incur a lot of work. Therefore the same techniques are applied as before on the same platform [8], which is based on saboteurs. Note that during implementation of the design using VHDL, testcases are created and executed to test and debug the individual parts, already providing a certain level of verification for the parts. Emulation is only required when testing the implementation while running on the actual FPGA.

3.3 System overview and diagrams

To indicate clearly the location of the proposed design into the existing SoC, first an overview of the system is provided and discussed briefly. Additionally, several state diagrams show the working principle of the cache. What follows is a block diagram of the complete system including a brief explanation. Lastly an interface diagram of the system is shown.

3.3.1 Schematic system overview

In Figure 3.1, the structure of the complete system (i.e. SoC) is shown. In its current form it is designed and implemented by Wietse Heida[1] and Daan Ravesteijn[2]. The figure

[1]Wietse.Heida@Technolution.nl

[2]Daan.Ravesteijn@Technolution.nl

is taken from the work of Wietse [8], with addition of the red lined square which indicates the part relevant to this thesis project.

Figure 3.2 shows the cache structure in more detail, including the surrounding components and interfaces. Here it can be seen clearly what the two interfaces are for the cache subsystem within the SoC.

Firstly it adheres to the `tl_data_bus` in order to communicate with the core. This bus provides 32 bit transfers and implements one transfer per clockcycle.

The second interface is specific to this project, being the interface towards DRAM which is provided by an AXI bus. This will be the interconnect for the cache towards the MDDR DRAM-controller, which is a built-in component of IGLOO2. Also connected to the AXI bus is an Advanced Peripheral Bus (APB) bridge to provide an interface for the peripherals to be added by employer. Arbitration on this bus is done by the so-called AXI-interconnect which is generated from Intellectual Property (IP) tools.

3.3.2 Flow diagrams for the working of the cache

To clarify the working of both caches their behavior is captured in flow diagrams. State diagrams are not applicable here since the flow may occur in one clock cycle. The first flow diagram depicts the general working of the WT cache as shown in Figure A.1, found in Appendix A. Note that for the read-only instruction cache a simpler flow diagram arises which is not included separately, without the possibility to write and without the emptying of the buffer. The states shown in Figure A.1, except for 'idle', are worked out in the following flow diagrams to provide an extended insight in the working of the cache. Note that the diagrams do not reflect the implementation as discussed in Chapter 4 since in hardware parallelism is deployed for e.g. the detection and correction of errors.

Figure A.2 shows the diagram for performing a read. After receiving the data from the cache memory at the indicated address, firstly error detection and possibly correction is performed. When it turns out the error in the requested word is uncorrectable, the block is re-fetched from LLM. When no error occurs or an error was corrected, the valid state of the block (instruction cache) or requested word (data cache) is checked subsequently. If not valid, the block can not be used and the data has to be fetched from the LLM. Ultimately, a valid block is checked for its tag matching with the requested address, i.e. indicating whether it is the right block. If the tag matches a cache hit occurs and the data is fed through back to the core. Additionally the Least Recently Used (LRU) bit is updated for the set-associative cache, i.e. the data cache. Otherwise the data has to come from LLM (cache miss).

A similar flow diagram arises when considering the event of a write incurred by the processor, as shown in Figure A.3. Note that this diagram only applies to the data cache since the instruction cache is read-only. At a write request from the processor the data has also to be written to LLM, both in case of a miss and hit. This means the data to write is placed in the write-buffer (not indicated in the figure) and subsequently written to the LLM. Before storing the byte(s), check bit(s) are added.

The other possibility to write the cache is by the LLM, shown in Figure A.4. This scheme occurs when the cache does not have the requested data i.e. in case of a miss. The block received from the DRAM is fed through to the core selecting the right word.

Concurrently the block is stored together with its check bits into the cache for future reference. This is done on the LRU position after which the LRU bit is updated.

3.3.3 Block diagram: connecting components of the cache

In order to acquire a better understanding of the components to implement, also a block diagram is constructed as can be seen in Figure B.1, found in Appendix B. All components and signals directly related to the cache structure are depicted. The central component, called 'cache controller: signal routing' serves as interconnect for all other components. Simple logic like selectors/multiplexers are included, combined with signal routing to interconnect the components. Note that the data/instructions are read per word and written per block, therefore four times as many encoders are needed compared to the decoders. Per block only one pair of tag and status bits is needed, so for these only one encoder and decoder are provided. The x and y depicted in the figure are there to compensate for the extra tag bits needed at larger cache widths.

To make clear how the storage is used to store all necessary bits, Figure 3.3 and Figure 3.4 give insight to this for instruction and data cache respectively. Every single rectangle represents one bit while a red instance depicts a check bit.

3.3.4 Interface diagram: connections inside the structure

While the block diagram gives a good indication of the components to implement, it does not provide information about the interfaces. Therefore also the interface diagram is constructed, to be seen in Figure 3.5. The interfaces are clear: the `tl_data_bus` will serve as connection towards the core of the processor while AXI acts as bus towards the DRAM controller. The widths of the buses are provided in the picture.

3.4 Conclusions: design of the configurable fault tolerant cache

This chapter describes the constraints and considerations needed for designing the requested configurable fault tolerant cache structure. The design for the cache structure consists out of an instruction and data cache, able to be integrated with the softcore of employer (Technolution B.V.). Since the data cache must be able to handle multiples of bytes efficiently, the data cache is read and written per byte, while instructions (32 bits) are not subdivided.

The cache is designed to be configurable in the following two aspects and within the following ranges:

- capacity: 4/8/16/32 kB
- fault tolerance: low and high protective

The capacity range is verified to fit on the FPGA selected for implementation. This range does not reflect the technical borders of a specific device, but originates from the employer. The range holds both for the instruction and data cache.

The fault tolerance is achieved by using EDAC, both ECC and EDC. For the low protective version of fault tolerance, possible errors will be corrected to a certain extent, hence ECC has to be used. Heavy protection requires detection only which means that EDC suffices. A tabulated summary on following information is shown in Table 3.1.

For the low protective fault tolerance scheme, two techniques are introduced.

First, in order to detect two errors and correct one error, SEC/DED code is selected. More specifically, Hsiao SEC/DED is found to be the most efficient way to detect two errors or correct one error for bit quantities larger than 8 bits. These are the instructions (32 bits), tags in the data cache, tags appended with a valid bit in the instruction cache and the valid signal (16 bits) in the data cache.

Second, for data (8 bits per entry) one parity bit will be added for detection of one error. To correct an error in a data byte it is re-fetched from LLM, which means a WT policy has to be applied. To minimize the latency in case of writing, a write buffer is included.

The high protective fault tolerance scheme also incurs two techniques in this design, although this part is not implemented.

First, for detecting up to three errors within a byte, TED code may be used. Technically the coding can be exactly the same as for SEC/DED code, only the codecs have to be adjusted. Therefore also Hsiao was selected as the preferred technique for TED code. This scheme may be applied on the byte storage in the data cache.

Second, for detecting up to five errors in data quantities of more than eight bits, 5-ED code is chosen. Selection of the technique is restricted by the latency requirement of one clock cycle hit time. This results in the choice for an unusual technique, being two complementary Hsiao/Hamming codes. It may be applied to the instructions (32 bits), tags in the data cache, tags appended with a valid bit in the instruction cache and the valid signal (16 bits) in the data cache.

Another part of this chapter defines the basic cache parameters. The block size is 128 bits i.e. four 32-bit words because of the efficiency in burst width in the connection to the DRAM. Data cache is two-way set associative while instruction cache is direct mapped.

Above summaries can be seen as the base for the implementation, described in the next chapter. Together they adhere to the requirements stated by the employer and are aimed to achieve an efficient, fault tolerant and configurable cache structure.

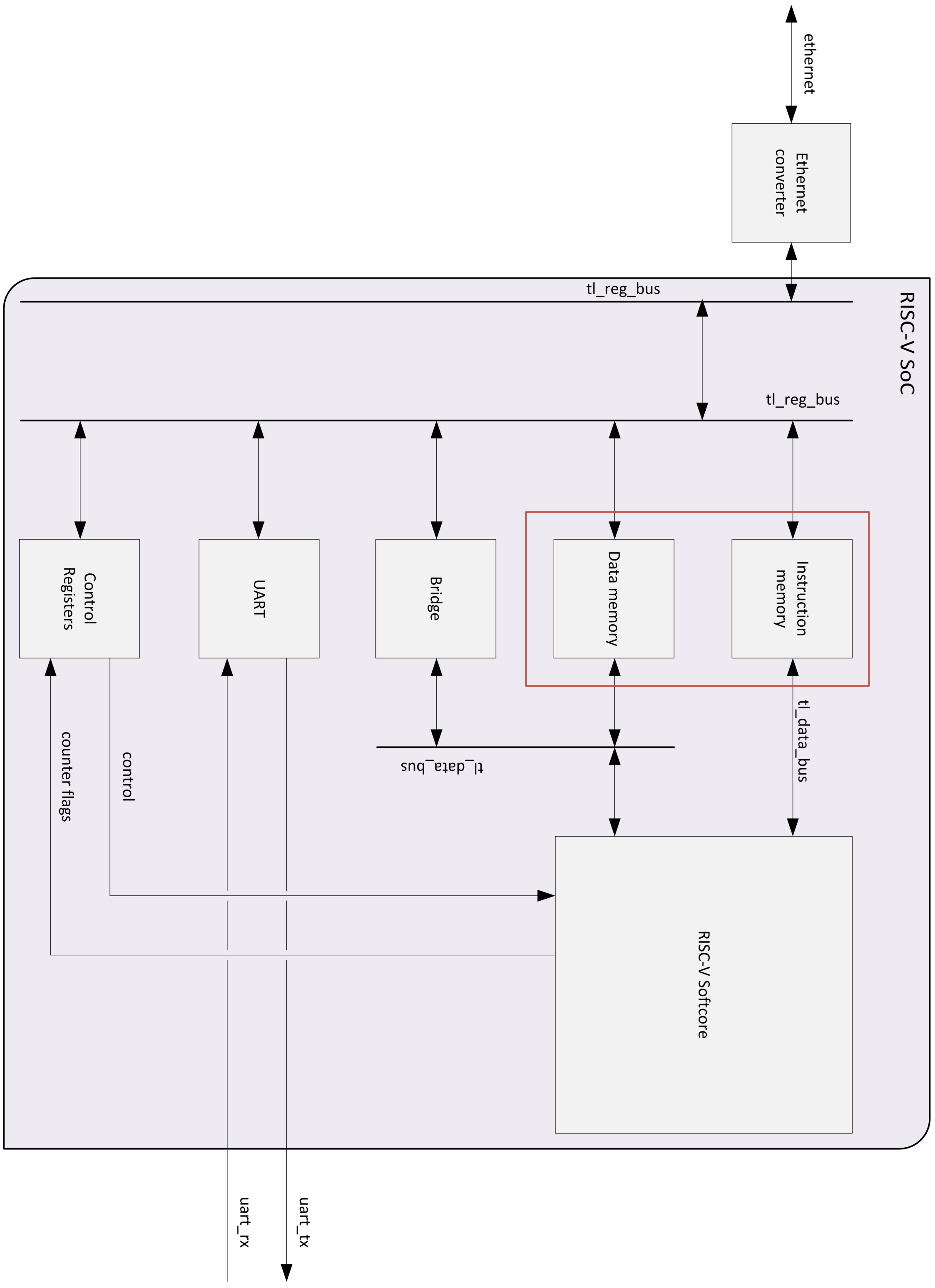


Figure 3.1: An overview of the RISC-V based SoC as implemented during foregoing projects. Picture taken from Wietse's work [8].

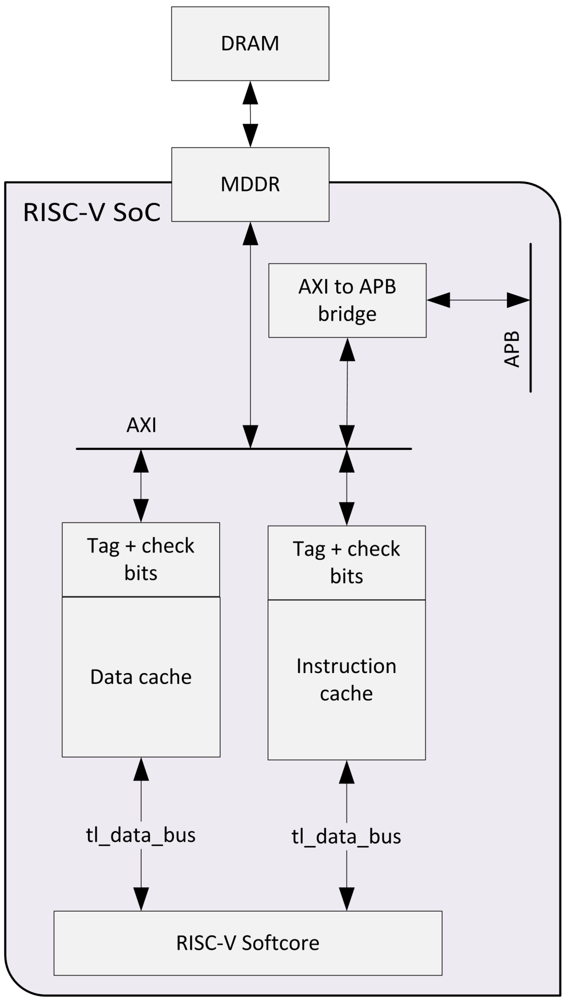


Figure 3.2: Part of the system relevant to this thesis project. It shows a part of the SoC, including a cache structure and the appropriate interfaces to (pre-defined) MDDR DRAM controller by an AXI interconnect.

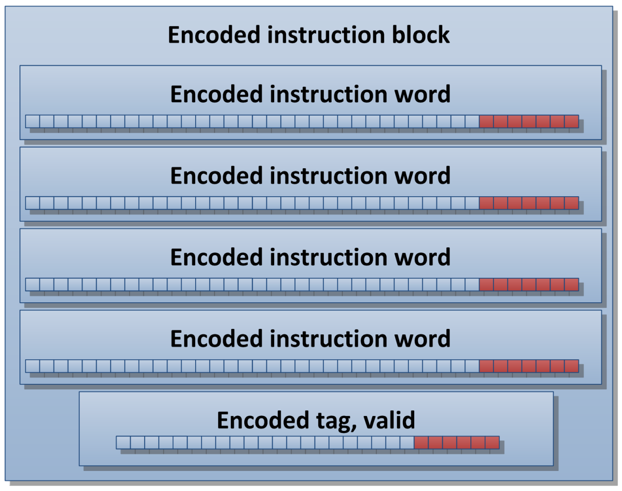


Figure 3.3: Structure of the instruction cache. Necessary data is stored in a single memory instance.

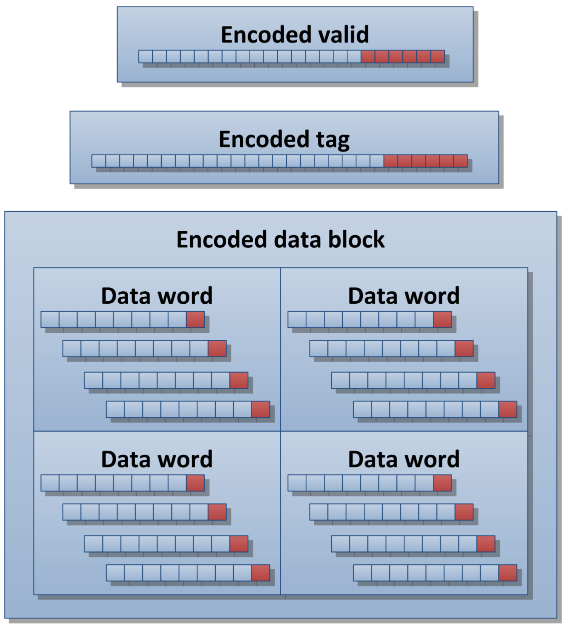


Figure 3.4: Structure of the data cache. Necessary data is divided among three memories for each of the two ways (six memories in total).

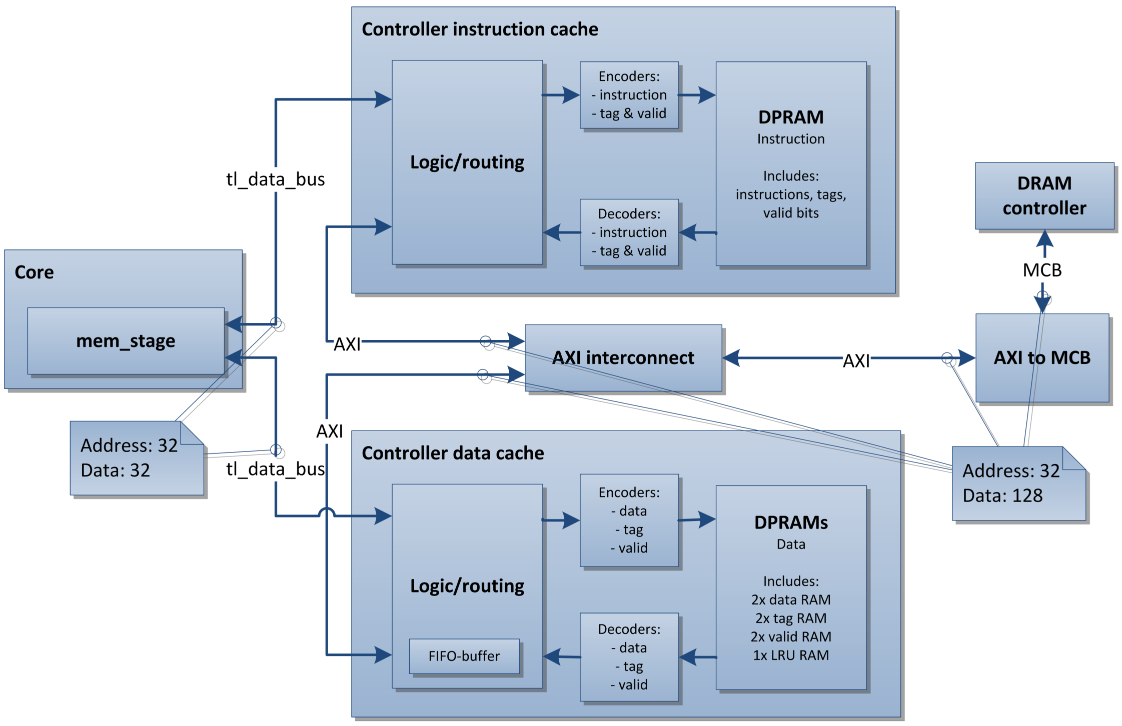


Figure 3.5: Interface diagram for both caches showing the interconnection. The relevant parameters are included in the figure.

Implementation 4

In this chapter, the implementation details are discussed. Three main topics are distinguished:

1. fault tolerance: implementing codecs,
2. cache controller: managing loads and stores of instructions and data,
3. completing the memory structure: connecting the caches to DRAM by AXI.

As defined in Section 2.1, the implementation is written in VHDL while the targeted technology is an FPGA. However, for efficient prototyping and testing, an SRAM-based FPGA is used instead of the flash-based IGLOO2. The platform used is called the GigaBee [56], which features a Xilinx Spartan 6 FPGA, two banks of 128 MB DDR3 SDRAM and additional hardware meant for easy prototyping.

4.1 Realization of fault tolerance: codecs

In this first section, the relevant implementation details about the EDAC measures are discussed. Codec is short for the combination of encoder and decoder. All logic needed for the EDAC is implemented as codec, which means there is a single implementation for both encoding and decoding. Although part of the logic is not used when acting as either encoder of decoder (as will be clear by exploring e.g. Figure 4.1), the synthesis tool removes the obsolete logic when optimizing the design. Hence there should be no additional resource overhead on the FPGA by choosing this strategy.

4.1.1 Parity codec

The simplest EDAC measure to take, is to implement parity code. In this thesis project, it is used to provide EDC per byte in the data cache. One parity bit is added to every byte of data for a total storage requirement of nine bits per data byte. The parity bit is the result of an exclusive-or (`XOR` for short, binary operator) over all data bits within the byte [20].

To make the parity codec more versatile, the codec is made generic in input size. Therefore, one generic input has to be provided indicating the data width. Internally the codec uses a `for-generate` loop to minimize hardware costs: only in range of the data width the `XOR` tree expands. Subsequently, resulting parity bit is processed (using another `XOR` tree) with the input parity bit to act as decoder: the error out signal is raised when the parity input and the calculated parity do not match.

The inputs and outputs represented in the implementation are found in Table 4.1.

Table 4.1: Parity codec parameters, inputs and outputs.

Signal name	In/out/par	Width/type	Description
`g_data_width`	par	*integer*	Generic: width of input `data_in`
`data_in`	in	`g_data_width`	The data to process
`par_in`	in	1	Parity bit to check (decoder)
`par_out`	out	1	Generated parity bit (encoder)
`error_out`	out	1	Parity error signal (decoder)

4.1.2 Hsiao codec

The other codec implemented in this thesis project is based on Hsiao code, which can be used for providing either SEC/DED or TED code. Hsiao SEC/DED code is used to provide light-weight EDAC for instructions and combinations of tag and status bit(s), i.e. all data portions larger than eight bits. TED is planned to be used for providing heavy protection to data bytes in future work, and as detection mechanism in the third intermediate option being DEC/TED code. Since these codecs are strongly related, both are implemented, although only the SEC/DED code is applied in this thesis project.

Also for this scheme, the width is a generic input value. However, it has to adhere to pre-specified widths for which the codec is implemented. The data widths currently supported for both the SEC/DED and TED codec are listed in Table 4.2.

Table 4.2: Supported widths for the Hsiao SEC/DED and TED codecs, including the number of check bits needed and total width.

Data	8	18	19	20	21	22	23	24	25	26	27	28	29	32	34
Check	5	6	6	6	6	6	6	6	6	6	7	7	7	7	7
Total	13	24	25	26	27	28	29	30	31	32	34	35	36	39	41

In order to establish the configurability, this time two additional generic values are required: the width of data and the width of the check code. In theory, the check width can be calculated from the data width, but VHSIC Hardware Description Language (VHDL) does not allow for calculated input widths, hence an extra generic value is needed. The complete list of generics, inputs and outputs for the SEC/DED codec can be seen in Table 4.3.

Figure 4.1 shows the structure of the Hsiao SEC/DED codec and thereby the relation of the inputs and outputs listed in Table 4.3. The structure of the `XOR`-tree connected directly to the incoming data is defined by the Parity Check Matrix (PCM). The inputs of the `XOR`-tree for a specific check bit are defined by the location of ones in the corresponding row in the PCM. The PCMs used are listed in Appendix C.

Note that this codec, as well as the parity codec, is implemented mainly in combinatorial logic. Therefore, all actions will occur within one clock cycle. This is only required for detecting any possible errors, i.e. a hit in one clock cycle. However, it can be conducted from the structure that just a little latency overhead is expected producing the corrected data. Therefore, it is chosen to also correct the error in one clock cycle instead of adding complexity for a little gain in latency.

Table 4.3: Hsiao SEC/DED codec parameters, inputs and outputs.

Signal name	In/out/par	Width/type	Description
`g_data_width`	par	*integer*	Generic: width of data
`g_check_width`	par	*integer*	Generic: width of check code
`data_in`	in	`g_data_width`	The data to process
`check_in`	in	`g_check_width`	Check code to check with (decoder)
`corr_data_out`	out	`g_data_width`	Corrected data out (decoder)
`check_out`	out	`g_check_width`	Generated check code (encoder)
`error_out`	out	1	Error signal (decoder)
`uncor_err_out`	out	1	Uncorrectable error signal (decoder)

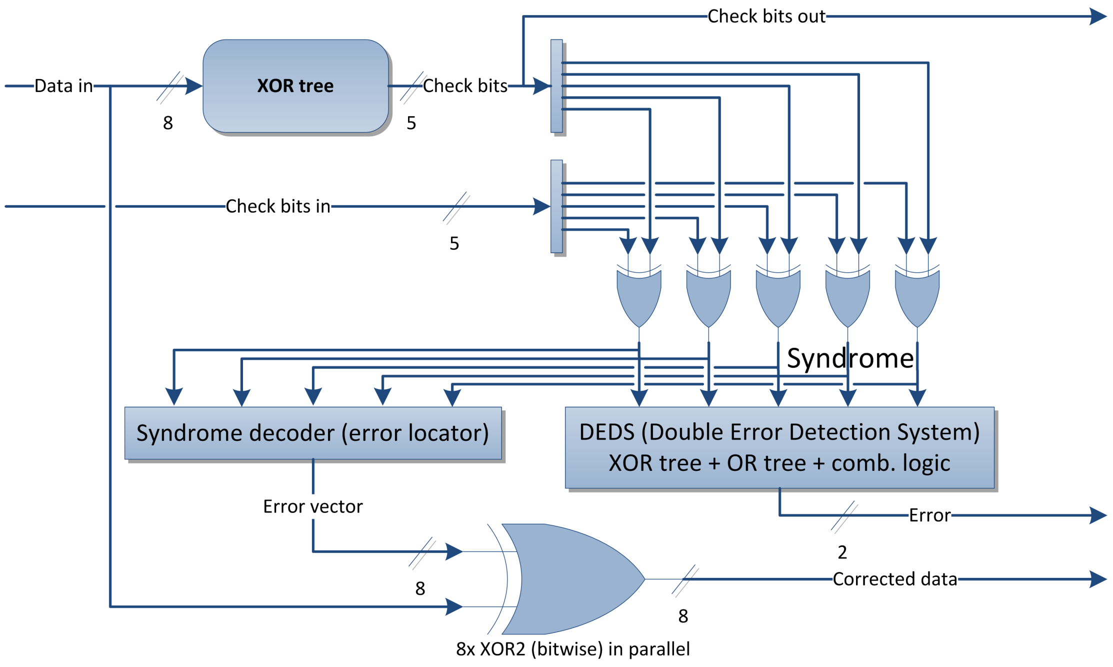

Check bits out
Data in
8
XOR tree
Check bits
5
Check bits in
5
Syndrome
Syndrome decoder (error locator)
DEDS (Double Error Detection System)
XOR tree + OR tree + comb. logic
Error vector
Error
2
8
Corrected data
8
8x XOR2 (bitwise) in parallel

Figure 4.1: Structure of the Hsiao SEC/DED codec. This picture shows the (13,8) implementation.

The structure of the syndrome decoder is shown in Figure 4.2. Also the syndrome decoder (error locator) coheres to the PCM: now the columns are used. Every signal in the error vector is the result of an `AND`-tree of any syndrome signal, of which the ones corresponding to a zero in the PCM are negated. For the (13,8) Hsiao SEC/DED codec this results in two possible circuits with just two logic levels. Note that the exact implementation is decided by the synthesis tool provided by the FPGA vendor. Either one of the implementations is needed eight times, which are all connected differently to the incoming signals as mandated by the PCM. The output of the syndrome decoder is the error vector. It features the same width as the original input data and indicates all errors by providing a one on erroneous location(s). This means that when no error occurs the error vector is equal to the null vector. To correct any error, the incoming data is processed together with the error vector by a bit-by-bit exclusive or operator, flipping the bits in which an error has occurred.

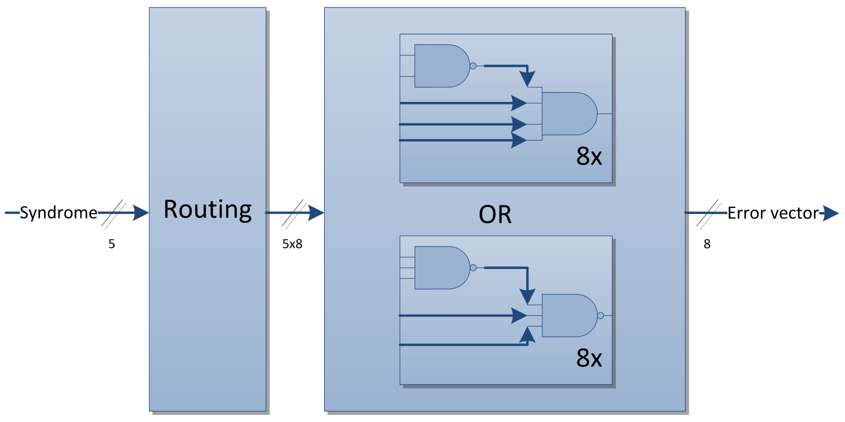


Figure 4.2: Structure of the syndrome decoder (error locator) of the (13,8) Hsiao SEC/DED codec.

In Figure 4.3 the structure of the Double Error Detection Signal (DEDS) generation circuit is shown. The trees inside this structure are just sequences of the indicated logic function, until just one signal is left.

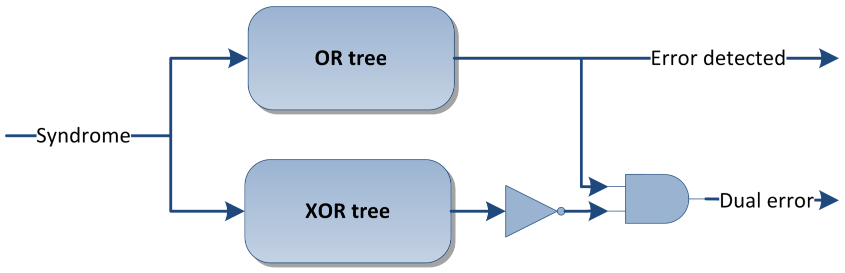


Figure 4.3: Structure of the DEDS of the (13,8) Hsiao SEC/DED codec.

The structures shown in Figure 4.1, Figure 4.2 and Figure 4.3 are implementations for a (13,8) Hsiao SEC/DED codec. Note that all other widths just differ in the first `XOR` tree and syndrome decoder since these are defined by their specific PCMs. Remaining parts just scale up as data- and check width increase.

For TED implementation, the first `XOR`-tree as seen in Figure 4.1 is identical to the SEC/DED implementation, for all data widths. Also the syndrome is generated in the same way, but the remaining logic found in the SEC/DED codec is minimized to an `OR` operation on all syndrome bits. This operation signals any error without calculating its position nor the number of errors.

4.2 Cache controllers: managing loads and stores

This second section on implementation details explains the working of the cache controllers, both for instructions and data. Since the instruction cache is direct-mapped and accesses are always quantized per word, this part of the cache structure is less complex. Therefore, the instruction cache controller is the first to discuss. Second the modifications needed for the data cache are reviewed.

4.2.1 Instruction cache: fault tolerant, configurable capacity and direct-mapped

The instruction cache is decided to be direct-mapped. Also it is read-only since instructions may never be written by the processor. For all design considerations see Chapter 3. This section starts with discussing the interfacing types, which are equal for both the instruction and data cache, except for the fact that the instruction cache is read-only. Related to this, the input and output signals are addressed. Second the configurability in capacity is discussed together with the implications of the configurability and fault tolerance. Lastly, the general working of de cache is briefly explained.

4.2.1.1 Interfaces of both caches: `tl_data_bus` and AXI

As is made visible in Figure 3.5, both caches interface the core and the DRAM controller. The core is interfaced by the so-called `tl_data_bus` which is developed by the employer. It implements a simple 32-bit bus capable of a maximum throughput of one transfer per clock cycle. Therefore it does not wait for an acknowledgment to proceed, but checks for this in the next cycle and stalls on its next request when the acknowledgment for the previous is not asserted.

The bus is defined by two interfacing types or channels, one for the request and one for the response. The request type (`t_tl_data_bus_req`) features a 32-bit address for both read and write requests, and one bit for write- and one bit for read requests. Since the address is shared, only one request (read or write) can be made at a time. For writing, also 32-bit data signal is included and a byte-enable (four bits wide). The response type (`t_tl_data_bus_resp`) on the other hand has an acknowledge bit which signals that the request is handled properly. Also a not-acknowledge signal is included (one bit), as well as a 32-bit data signal to return data on a read request.

At the other side of the cache structure, there is the connection to the DRAM, or the LLM in general. The modifications needed for implementing the AXI interface are discussed in a separate section (Section 4.3). Since both caches need to read from the LLM, both cache controllers act as AXI master devices. To the core (by `tl_data_bus`), the cache acts as a slave device.

Another interface is that of a `tl_reg_bus`, which is very similar to `tl_data_bus` and carries the same signals. This bus is used for verification possibilities by connecting registers to it, as is discussed in Section 5.1.2. The cache is acting as a master since the registers are placed inside the cache structure, i.e. they have to be addressable through the cache controller.

Next, there are some generic inputs. One is purposed for indicating the desired capacity, which is discussed next. Two other are meant for the configuration of the standard dual port Random Access Memory (RAM) component: `g_read_first_b` and `g_storage`. First changes the behavior of the dual port RAMs used as cache memory: when true any read is processed before a write and for false vice versa. Second one is meant for the synthesis tool: to what kind of memory should this dual port RAM be mapped? Options are block (RAM), distributed or auto (self-deciding). Since the memories are relatively large, block has to be chosen to prevent timing issues when synthesizing the design.

The complete list of all inputs and outputs is presented in Table 4.4, while the parameters are listed in Table 4.5. Note that the discussed interfaces are included per channel instead of per signal. The signals (channels) belonging to the AXI interface are addressed in Section 4.3.

Table 4.4: Inputs and outputs of the instruction cache.

Signal/channel name	In/out	Width/type	Description
`data_clock`	in	1	Clock from the core
`data_reset`	in	1	Reset from the core
`data_req`	in	`tl_data_bus_req`	Request channel from the core
`data_resp`	out	`tl_data_bus_resp`	Response channel to the core
`regs_req_instr`	in	`tl_reg_bus_req`	Request channel to registers
`regs_resp_instr`	out	`tl_reg_bus_resp`	Response channel from registers
`m_axi_aclk`	in	1	Clock from LLM
`m_axi_aresetn`	in	1	Reset (active low) from LLM
AXI write address	in&out	*AXI channel*	Write address channel
AXI write data	in&out	*AXI channel*	Write data channel
AXI write response	in&out	*AXI channel*	Write response channel
AXI read address	in&out	*AXI channel*	Read address channel
AXI read data	in&out	*AXI channel*	Read data channel

Table 4.5: Parameters of the instruction cache.

Parameter name	Width/type	Description
`g_depth_bits`	*integer*	Generic: cache depth (2^x)
`g_read_first_b`	*boolean*	Generic: read priority
`g_storage`	*string*	Generic: FPGA storage to use
`c_m_slave_base_addr`	32	Generic: AXI slave base address
`c_m_axi_burst_len`	*integer*	Generic: AXI burst length
`c_m_axi_id_width`	*integer*	Generic: AXI ID width
`c_m_axi_addr_width`	*integer*	Generic: AXI address width
`c_m_axi_data_width`	*integer*	Generic: AXI data width

4.2.1.2 Configurability in size

To make the cache capacity configurable, the generic value `g_depth_bits` is applied on several places within the implementation. Since the width (i.e. block size) is set statically to 128 bits, as well as the associativity to direct-mapped, the capacity should be altered by configuring the depth. The cache depth parameter indicates the number of blocks in the cache. The generic input is an integer, indicating a number which serves as the power with base two. Since the block size is 128 bits, the range to be supported is 8 up to 11, resulting in $2^8 = 256$ up to $2^{11} = 2,048$ blocks, equal to 4 to 32 kB.

Note that the tag width is also dependent on this generic value: when the depth increases, the tag width decreases and vice versa. The tag accounts for the part of the block address which is not used by locating it in the cache memory. The address itself is defined as being byte-based, with the two LSBs never being used (i.e. byte offset within a word). Furthermore, since the blocks are 128 bit, the two bits next to them are also obsolete (i.e. word offset within a 128-bit block). Hence these parts of the address are represented by the order of storing the bytes and words respectively inside the block. Taking these considerations into account, the tag width becomes $(32-4)-$`g_depth_bits`, resulting in a maximum of $28 - 8 = 20$ bits for the smallest cache and $28 - 11 = 17$ for the largest possible instruction cache.

When implementing the configurability, main difficulty is found to be the tag to store, which has to be protected and still being scalable in size. The component in VHDL to interface the on-board dual port RAM on FPGA is called `dpram` or dual port RAM. It is a standard component and is scalable in both directions (i.e. width and depth). To store the instructions in `dpram`, the requested depth of the cache is fed through to this standard component in order to scale up the memory capacity.

Since the tag is also stored in the same `dpram` instance, the width of this dual port RAM is also dependent on the depth of the cache. The valid bit is concatenated with the tag before adding fault tolerance, since protecting the valid bit individually would incur a high overhead. Together with the tag, the concatenation results in input widths for the codecs of 18 up to 21 bits. Hence, the codecs for the tags have to be generated separately for any configuration, which also means the codec has to support all the widths needed. The check width is six bits for all configurations, resulting in a total storage constraint of 24 up to 27 bits per tag/valid pair. Per instruction block, one tag/valid pair has to be stored. This results in a total size of $(35 - \texttt{g\_depth\_bits}) \cdot 2^{\texttt{g\_depth\_bits}}$, translating to $(35 - 8) \cdot 2^8 = 6,912$ up to $(35 - 11) \cdot 2^{11} = 49,152$ bits. The encoded instructions (seven bits check code per word) take up $4*(32+7)*2^{\texttt{g\_depth\_bits}}$ translating to $156 * 2^8 = 39,936$ up to $156 * 2^1 1 = 319,488$ bits. The total storage capacity needed for the instruction cache in different configurations count up to $39,936+6,912 = 46,848$ bits for the smallest configuration and $319,488 + 49,152 = 368,640$ bits for the largest instruction cache. In Table 4.6 the required memory capacity for the instruction cache is tabulated, together with the overhead induced by adding fault tolerance and the share of the tag/valid pair.

4.2.1.3 General working of the instruction cache

The structure of the instruction cache is represented in Figure 3.3 found in Section 3.3.4. Every block that is written into the cache is encoded by four Hsiao SEC/DED encoders, connected to the data-in ports of the single dual port RAM. Also the tag together with the valid bit are encoded with a Hsiao SEC/DED encoder.

The four instructions, together with the corresponding tag, valid bit and five pairs of check bits are stored together into one cache line, resulting in cache lines of 180 to 183 bits. Since just one instantiation of dual port RAM is used, the total line is read at

Table 4.6: Insight in the minimal memory usage of the instruction cache per capacity.

Configuration				
Capacity (kB)	4	8	16	32
`g_depth_bits`	8	9	10	11
No. blocks	256	512	1,024	2,048
Storage: instructions				
Instruction size (b)	32	32	32	32
Check size (b)	7	7	7	7
No. instructions stored	1,024	2,048	4,096	8,192
Storage instructions (b)	32,768	65,536	131,072	262,144
Storage check bits (b)	7,168	14,336	28,672	57,344
Total (b)	39,936	79,872	159,744	319,488
Overhead FT (%)	21.9	21.9	21.9	21.9
Storage: tag/valid pair				
Tag/valid size (b)	21	20	19	18
Check size (b)	6	6	6	6
Storage tag/valid (b)	5,376	10,240	19,456	36,864
Storage check bits (b)	1,536	3,072	6,144	12,288
Total (b)	6,912	13,312	25,600	49,152
Overhead FT (%)	28.6	30.0	31.6	33.3
Storage: combined				
Instructions + tag/valid (b)	38,144	75,776	150,528	299,008
Total check bits (b)	8,704	17,408	34,816	69,632
Total instruction cache (b)	46,848	93,184	185,344	368,640
Total instruction cache (kB)	5.72	11.38	22.63	45.00
Total overhead FT (%)	22.8	23.0	23.1	23.3

once. This means also a total of five Hsiao SEC/DED decoders are needed to check and correct the contents of that line.

In order to clarify the general working of the instruction cache, the execution flow is described. At a read request from the core, first it is determined whether the requested entry is loaded in the cache (i.e. a hit or miss). To determine upon a cache hit in the instruction cache, the following is checked:

- Is the entry valid?
- Does the stored tag match the corresponding address part of the request?
- Are there no uncorrectable errors in both the requested instruction and the tag/valid pair?

When it hits, the read data is put directly on the response line to the core together with an acknowledge. Otherwise, the request is forwarded to the LLM. After the LLM responded with the requested data (within a block), this data is both written in the cache (while being encoded) and fed through to the core together with an acknowledge.

Note that the functional working coheres to the flow diagram for a read (Figure A.2), although the order of processing differs.

4.2.2 Data cache: fault tolerant, configurable in capacity and two-way set associative (with write buffer)

Since the data cache is largely based on the instruction cache, it has several similarities. This part about the data cache controller mainly focuses on the differences. Similarities include the set of interfaces (`tl_data_bus` and AXI, see Table 4.4) and the basic functionality of the cache. Differences to discuss are the following:

1. handling stores,
2. byte processing,
3. two-way set associativity.

These three topics are treated separately in the following sections.

4.2.2.1 Handling stores: FIFO write buffer

One of the main differences with the instruction cache is that the data cache by default is able to accept stores (i.e. write requests). Since a WT policy is chosen, any modification in data incurred by the store is also applied in the LLM. In order to continue execution during a write, without waiting for the write to complete, a FIFO write buffer is added. To prevent reading old values from the LLM, the writes get priority over the reads. This means that the write buffer is emptied before any read request from the LLM is permitted. As a consequence, reads may experience a large latency when the cache has taken many write requests in its buffer. This may be tuned by regulating the buffer depth included as an additional generic, which is the only addition to the interface of the instruction cache controller. This generic is of the natural type.

An alternative solution would be to add a dirty bit which indicates whether an entry is still in the buffer. However this bit then needs to be fault tolerant, since an error could possibly lead to reading in an old value, messing up the execution. This bit cannot be stored together with the data, nor the combination of tag and valid since RMW operations would be incurred. Furthermore a dirty bit would add extra complexity to the controller.

Yet another option would be to check for every read whether the entry exists in the write buffer. If it is, the read either has to wait or, if possible, use the entry from the write buffer. Note that it is expected that most of the reads do not interfere with writes in the write buffer. This is more true due to the fact that when an entry is still in cache, it will not request a read from LLM. On the other hand, the performance would suffer when the read-out of the buffer takes a relatively large amount of time. Therefore the solution requires a fast read-out of all entries. Ideally, the buffer may be implemented as a full-associative memory, however this is hard to realize in an FPGA and requires a lot of on-board resources.

Seen the possibilities, it is chosen to leave the dirty bit out and opt for the simplest scheme including the write buffer, namely the write priority over reads.

The FIFO buffer is featured by a standard component available at the employer. The buffer depth is chosen to be 32 by default. With this depth, it is expected the buffer will not fill up quickly. When the buffer is full, the request is stored and no acknowledge is given before the request has been written to the buffer. Put differently, the request has to wait until one other write operation is completed, as if the buffer is full.

Since both the data to write and their belonging addresses are stored in the buffer, they have to be protected. Therefore, additional codecs are placed in the data cache structure to add Hsiao SEC/DED protection to these entries. For data the standard (39,32) SEC/DED protection is applied, just as for the instructions within the instruction cache. The address bits needed (30 bits) are concatenated with the byte-enable bits (four) for a total of 34 bits. Since the values in the buffer are stored for a smaller time interval than with cache entries, the employer agreed upon a (41,34) Hsiao SEC/DED scheme. This scheme is marginally less secure than its (39,32) counterpart, which is set as minimum for the cache storage itself.

Total minimal storage required for the 32-entry buffer is $39 + 41 = 80$ bits per entry, resulting in $80 * 32 = 2560$ bits.

4.2.2.2 Byte processing and storage used

Another structural difference compared to the instruction cache is the ability to process individual bytes of data, since stores might incur quantities of data which are smaller than a word. This is due to the byte-enable field in the `tl_data_bus` and the design of the core using this functionality. To efficiently write any combination of single bytes within a word without an RMW operation, the data has to be stored per byte, hence also fault tolerance is implemented per byte. To keep the overhead for check bits small, it is chosen to take parity protection per byte to provide EDC for one error. A WT policy ensures that the data can be retrieved when an error occurs. A modified instance of the dual port RAM is implemented to provide byte access to the 32-bit port on the core side, while the LLM interfaces the cache by a 128-bit port without byte enable support. Existing dual port RAMs feature either two fixed 32 bit ports with byte enable support or two ports with configurable widths without byte enable support. Since a parity bit is appended to every byte, the actual modified dual port RAM handles nine bits as one byte. This means the modified dual port RAM has one 36-bit port supporting byte-enable (per nine bits) and one 144-bit port which does not support byte-enable. Note that the (de-)coders are directly connected to this memory, resulting in the 32-bit and 128-bit ports which exclude the parity bits.

Also for the data cache a minimal storage requirement can now be calculated, in the same way as is done for the instruction cache. Additionally, the valid is stored separately as a 16-bit signal together with its six check bits. Also the LRU bits are included, although this accounts for a very small part of the cache. The overview of the memory requirements for the data cache are tabulated in Table 4.7.

4.2.2.3 Two-way set associativity

Last main difference is the associativity: while the instruction cache is direct-mapped (i.e. one way), the data cache adopts two-way set associativity. As explained in Sec-

Table 4.7: Insight in the minimal memory usage of the data cache per capacity.

Configuration				
Capacity (kB)	4	8	16	32
(g_depth_bits)	8	9	10	11
No. blocks (total)	256	512	1,024	2,048
Storage: data				
Data size (b)	8	8	8	8
Check size (b)	1	1	1	1
No. data bytes stored	4,096	8,192	16,384	32,768
Storage data (b)	32,768	65,536	131,072	262,144
Storage check bits (b)	4,096	8,192	16,384	32,768
Total (b)	36,864	73,728	147,456	294,912
Overhead FT (%)	12.5	12.5	12.5	12.5
Storage: tag				
Tag size (b)	20	19	18	17
Check size (b)	6	6	6	6
Storage tag (b)	5,120	9,728	18,432	34,816
Storage check bits (b)	1,536	3,072	6,144	12,288
Total (b)	6,656	12,800	24,576	47,104
Overhead FT (%)	30.0	31.6	33.3	35.3
Storage: valid				
Valid size (b)	16	16	16	16
Check size (b)	6	6	6	6
Storage valid (b)	4,096	8,192	16,384	32,768
Storage check bits (b)	1,536	3,072	6,144	12,288
Total (b)	5,632	11,264	22,528	45,056
Overhead FT (%)	37.5	37.5	37.5	37.5
Storage: LRU				
LRU size (b)	1	1	1	1
Storage LRU (b)	128	256	512	1024
Storage: combined				
Total word + tag + valid (b)	42,112	83,712	166,400	330,752
Total check bits (b)	7,168	14,336	28,672	57,344
Total data cache (b)	49,280	98,048	195,072	388,096
Total data cache (kB)	6.02	11.97	23.81	47.38
Total overhead FT (%)	17.0	17.1	17.2	17.3

tion 2.2.2.2, it means that every line can reside in multiple, in this case two, locations. In order to implement this, the storage (dual port RAM) and the belonging logic (e.g. the codecs) are duplicated. Although, the capacity of both memories is halved, keeping the total capacity in line with the intended capacity indicated by the generic value (`g_depth_bits`). When checking for a hit at a load, both ways have to be checked on validity, tag match and correctness of data.

Since the cache is set associative, a replacement policy has to be implemented, as explained in Section 2.2.2.4. LRU is chosen since it is the most efficient policy, although for two-way set associativity it incurs an extra status bit (LRU-bit) to indicate which entry is used last. Since the LRU-bit is updated every time the relevant cache entry is used (either a load or store), it is stored separately, apart from the tag/valid pair and the data. Hence RMW operations are incurred when the LRU-bit would be stored at the combination of tag and valid, since these are only modified when the total line is being replaced. The same holds when the LRU-bit is stored together with a part of the data, since most data will not be updated at every operation. Note that the LRU-bit does not have to be fault tolerant, since a corrupted value will result in a performance penalty. The LRU bit is used every time a new line has to be stored.

4.3 DRAM controller and its interface: MCB and AXI

Connecting the DRAM as LLM via the memory controller has taken a considerable amount of time, therefore a separate section is devoted to this topic. First a brief overview is provided about the interface of both cache controllers: the AXI protocol. Second, the interface to the memory controller is regarded. Lastly, the way of initializing the DRAM is addressed.

4.3.1 AXI: interface and arbitration

The AXI bus is a complex interface compared to the `tl_data_bus`. Hence, it features a lot of configuration options to optimize the performance. Above all, it provides the advantage of being a widely adopted standard. Thereby, if the interface is implemented correctly according to the standard, the structure can be connected to a range of other IP from many manufacturers. Refer to Section 2.6.2.2 for introductory information about the bus and references to the standard.

Relevant for this thesis project is to know that the AXI bus includes five channels in total. These channels are divided over both read and write connectivity.The total set of signals featured by these channels is not discussed in detail in this thesis. For further reference see [46] (AXI3 only) and [47]. Both caches act as an AXI master featuring a data with of 128 bits and only supporting single beat bursts. This means that, per transaction, a maximum of 128 bits can be transferred. Narrow bursts are also implemented in order to write specific bytes or even a single byte.

With the cache structure featuring an AXI-interface, the components directly connected to it can be generated by specific tools, e.g. provided by the manufacturer of the FPGA used. In this case Xilinx IP is used to provide an interconnect for the caches. Both are equipped with an AXI master interface, while the structure features just one slave: the DRAM controller. Note that another option would be to instantiate two memory controllers, one for instructions and one for data. Besides that this scheme would incur more overhead by using two memory controllers, it would also not fit any other FPGA equipped with just one DRAM device. Therefore, to preserve generality, an interconnect is invoked which connects both cache controllers to the same DRAM controller.

The interconnect also takes care of the arbitration, required to handle the requests in a fair way. By default, this is implemented as Round Robin: all ports have equal priority. The interconnect is configured such that the first port can only read from DRAM, hence the instruction cache is connected to this port. The other port is able to both read and write, suitable to the data cache. Other configuration options such as an additional read and/or write FIFO buffers are left untouched.

While the language preference in the Xilinx tool is set to VHDL, the interconnect provided by the Xilinx tools is written in Verilog, together with a VHDL top-level. This makes the project mixed-language, which complicates simulating the system. For newer series of FPGAs, or those from other manufacturers, there might be newer software which provides full VHDL output which would remove this complication.

4.3.2 Memory Controller Block (MCB)

To connect the combined AXI master interface of the interconnect to the DRAM, a memory controller is required. For the (prototype) FPGA used, Xilinx provides a so-called MCB. This block translates to the integrated memory controller on the FPGA. However, the generated code outputted by the Xilinx tools uses a library which is protected by license. Since the employer does not own this license, the code could not be used, at least not in simulation. However, the employer has solved this problem before by using another instantiation of the memory controller. This one is written/generated in VHDL and is largely based on the files which can be generated from the Xilinx tools. Also a Bus Functional Model (BFM) is provided to perform simulations. This specific code, made available by the employer, is used to interface the DRAM.

The MCB is interfaced by the Xilinx-specific MCB-bus, while the cache structure is based on AXI. In order to connect the cache structure to the MCB, an AXI to MCB bridge is added to the implementation, which is found in the generated files by the Xilinx tool.

4.3.3 Initializing the memory

The MCB supports different data widths and port configurations. However, for 128-bit data width, only one configuration option is available which provides a single (read/write) port. At the phase of integrating the structure in the existing SoC and subsequent testing on the FPGA, this property incurs a problem. The original memory, which is replaced by the cache structure, is dual port RAM. One of the ports is connected to the core while the other leads to an interface which can be accessed externally. The latter port is used to fill the memory while the SoC is on the FPGA. Note that in simulation there are other alternatives to fill the memory.

The cache controllers provide just one port each at the core side. The other port of the internal dual port RAM is used to interface DRAM. Therefore the memory (here: DRAM) could not be filled through the cache structure like it is done with the original dual port RAM. Since the extra connection to the memory is essential for loading applications and belonging data, three options are considered to add the extra port. These options are shown in Figure 4.4 and are discussed subsequently.

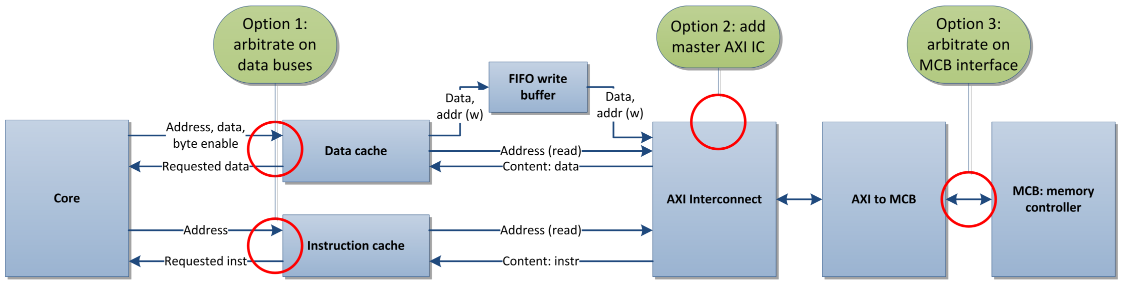


Figure 4.4: The three identified solutions to provide an extra port to the memory.

Options one and three are comparable: they involve the addition of either two (option one) or one (option three) arbiter(s). An arbiter is a functional block which handles multiple (here: two) masters connected to one slave. The main difference between the two options is the interface to arbitrate: either the `tl_data_bus` for option one or the MCB bus for option three.

Option one is special since it also fills the cache memory, which may or may not be desirable. The disadvantage of this option is its performance since it uses the cache infrastructure. Option three on the other hand would provide a direct access to the DRAM and may be qualified as the more logical place to arbitrate and inject the initializing instructions and data.

Both options however incur extra work: implementing an arbiter. The `tl_data_bus` specific to the employer has no arbiter currently implemented. Also, this option was not qualified as favorable by the employer. Arbitrating the MCB bus is the preferred approach.

Since such an arbiter is not part of the problem statement, only one attempt is made to implement a simple arbiter for the MCB bus, to keep the costs in time low. MCB and the belonging bus are Xilinx specific, therefore the information resources are limited to those provided by Xilinx and (un)official forums. This complicates the implementation of this new block. Note that an arbiter is included in the MCB which is only activated when multiple ports are connected. This is no option for the chosen 128-bit width configuration.

The implementation made is tested in simulation and picks the first request coming in. It also features a single slot to hold a request when the previous is not finished yet. However, this simple implementation failed during tests on the FPGA, which led to the decision to go for option two.

For option two no new arbiter needs to be implemented, since the idea is to extend the arbiter inside the AXI interconnect. This interconnect is generated for different input counts and data widths, as discussed in Section 4.3.1. The difficulty for this option is the translation from employer's `tl_reg_bus` to AXI. Although AXI may be an even more extended bus compared to MCB, the resources and documentation is plenty due to the wide applicability of this bus.

The implementation of this bus translator, or bridge, is realized using the implementation of the cache controllers. Hence, the cache controllers themselves act as `tl_data_bus` slave and as an AXI master. Note that the `tl_data_bus` is a modified `tl_reg_bus` with increased timing requirements, which means a `tl_data_bus` slave should also be suitable to serve as a `tl_reg_bus` slave. Modifying the cache controller therefore resulted in a working `tl_reg_bus` to AXI bridge, both tested in simulation and FPGA. The AXI interface is replaced by one including three slave inputs. The result is represented in Figure 4.5.

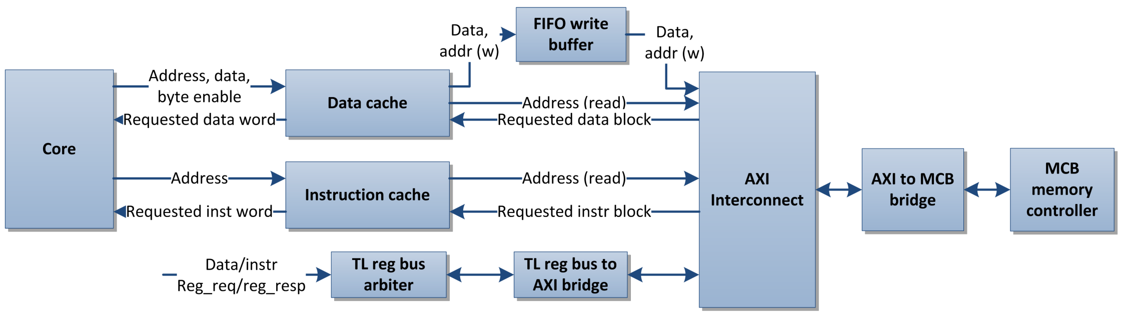


Figure 4.5: Worked-out AXI arbiter solution providing an extra port to the memory meant for initializing.

4.4 Summary: implementation details of both cache controllers and DRAM interface

In this chapter, the implementation details for both cache controllers and the DRAM interface are discussed. Also the solution to fill the DRAM on the FPGA is addressed. However, the chapter started with the implementation of the codecs.

The parity codec implementation for the data cache is straight-forward: a `XOR` is applied to the combination of all input bits (which is generic in size) resulting in the parity bit. Another `XOR` generates the error out signal, which is asserted when the generated parity bit and the input parity bit do not match.

The Hsiao SEC/DED (and TED) codecs for all data quantities larger than eight bits require more work to successfully implement them. Base for the codecs are the PCMs, which can be found in Appendix C. Due to the fact that every cache size implementation resembles another PCM, the structure is only partially configurable, that is within the predefined values. The codecs are fully combinatorial, just like the parity codec. The structure of a (13,8) Hsiao SEC/DED codec is shown in Figure 4.1.

The cache controllers are comparable in functionality, although the additional features of the data cache make the controller more complex in implementation. Basically, both controllers have to check whether a hit or miss occurs. In case of a miss, the missing data/instruction is fetched from the LLM. Interface going out of the controllers is an AXI master bus, for which the required functional block is included into the controllers. Ingoing bus is the `tl_data_bus`, a relatively simple proprietary bus from the employer.

Three conditions are checked in order to determine a hit or miss: the tag/address, the validness and whether an uncorrectable error has occurred. Since the tag scales according to the cache depth, different codecs are invoked for every depth.

In the data cache, also writes have to be performed. For this, a write buffer is included with configurable depth. Furthermore, data bytes may be written individually. Since the data cache is two-way set associative and features a WT policy, additional bits are stored in the form of LRU data. These bits are not protected since this would incur a relatively large overhead, and there is no harm for the execution flow when these would be erroneous.

The connection to the DRAM controller requires a connection from both AXI masters, provided by the cache controllers, to the MCB. To arbitrate the AXI signals, generated logic is applied in the form of an AXI interconnect. Subsequently, a generated AXI to MCB bus bridge is included into the implementation.

Problem encountered when implementing the logic unto the FPGA is the missing port to the memory, which was provided earlier by a dual port RAM. Three options are examined, of which an `tl_reg_bus` to AXI bridge is chosen and implemented. This bridge is added to the AXI interconnect and thereby arbitrated between the cache controllers.

Verification and results 5

This chapter discusses the way the implementation is verified and shows the results obtained. First the verification is addressed with regards to simulation. After this, the verification on the Field Programmable Gate Array (FPGA) device is regarded. Since in the current state of the project the verification fails for certain programs, the problems encountered are also discussed. Third, some results are presented. The occupied resources, clock analysis and performance metrics in the form of Dhrystone results are included in this section. Lastly the chapter is summarized and concluded.

5.1 Verification by simulation

As is a common practice when working with VHDL, every component is tested individually by using a tailored testcase. Also combinations of components are tested, up to the total system. Simulation tool used is ModelSim by Mentor Graphics, version 10.3d (Altera Starter Edition).

This part tells about the simulations run for all components involved in this thesis project. First the verification of the codecs is briefly discussed, after that the cache controllers including their interface to the Dynamic RAM (DRAM) controller. Lastly, the simulation run for the total System on Chip (SoC) is explained briefly.

5.1.1 Simulating the codecs

To verify the correct working of the codecs, a custom testfile is constructed for every codec instance. First test within the testcases is to check the working as an encoder. In order to check the outgoing check bits, the Parity Check Matrices (PCMs) are used to generate some correct check codes manually. Three sample inputs are chosen featuring a full-one input, a full-zero input and a checkerboard pattern input. In the testcase, the outgoing check bits are checked against these expected check bits.

Remaining tests all target the decoding capability. Note that for the parity and Triple Error Detecting (TED) codecs, the verification targets the detection capabilities only. For the Single Error Correcting/Double Error Detecting (SEC/DED) codecs, also the correcting feature is put to test.

In order to verify the detection (and correction for SEC/DED) of a single error, one of the bits in either the input or check bits is modified from one to zero or vice versa. Thereby, an erroneous combination of input and check bits is inserted in the codec, which is verified to output an error signal. For SEC/DED, the corrected code is verified. Faults were injected both in the input and in the check bits.

Both Hsiao codecs should also detect two concurrent errors. A similar strategy is applied: two bits are flipped in the input, check bits or one in both. The testcase checks whether the codecs report an error. Furthermore, it is checked that the SEC/DED codec flags an uncorrectable error.

Three errors have to be detected by the TED codec only. In this case, three bits are modified in either the input, check bits or a combination of those. The TED codec successfully detects these errors.

All described tests are performed successfully without unexpected behavior. This means that the codecs deliver their advertised detecting and correcting capabilities.

5.1.2 Simulating the cache controllers

A `tl_reg_bus` Bus Functional Model (BFM) instance as well as a Memory Controller Block (MCB) BFM are used for verifying the cache structure. They respectively generate traffic on the input and act as a (slave) DRAM controller. Note that the `tl_reg_bus` is equal to the `tl_data_bus`, except for the timing requirements as explained in Section 4.2.1.1. However, no BFM for the latter is available, therefore the `tl_reg_bus` BFM is used. The Advanced eXtensible Interface (AXI) to MCB bridge is also included in the simulation, in order to test the complete chain from processor input to DRAM (controller).

Two other components are included solely for verification purposes: the saboteur and a register instance. The latter includes both counter registers and a control register. These are discussed first, subsequently the testing of loads and stores (data cache only) functionality is explained.

5.1.2.1 Saboteur: injecting faults

The saboteur used in this thesis work is the same as applied in [8]. Although, not all functionality is used. Some error vectors (indicating an error location with a one) are defined statically and serve as an input for the saboteur. Which error vector is connected to the saboteur is decided by an additional two-bit signal (`insert_fault_data_i` and `insert_fault_instr_i`). In this way, the level of fault injection can be regulated.

First option ("00") for both signals indicates no error at all: the saboteur is disabled. This is the default value. Second option ("01") implements an error pattern which should be fully correctable for the SEC/DED protected parts and detectable for the parity-protected part. Third option ("10") features an error pattern which is uncorrectable for all codecs, but still detectable. Note that all cache accesses should generate a miss in this case, since a miss is also triggered by uncorrectable errors. Fourth option ("11") integrates a case which is undetectable and uncorrectable, featuring two errors per byte and three errors per SEC/DED protected part.

5.1.2.2 Registers: counters and control

As discussed before, the saboteur is regulated by a certain signal for every cache controller. To set this signal externally, it is controlled by a control register.

Counter registers count the number of cycles that the incoming signal is high. Several of these counters are included in the register instance to track the behavior of the caches. These are, divided in their categories:

1. cache performance:
   (a) hit count,
   (b) miss count,
2. general (functional) errors:
   (a) simultaneous read and write at input (not supported),
   (b) "others" case visited in any of the processes (AXI or saboteur),
3. error detecting/correcting performance:
   (a) no. errors detected:
       tag (tag/valid for instruction cache),
       valid (data cache only),
       data,
   (b) no. errors corrected:
       tag (tag/valid for instruction cache),
       valid (data cache only),
       data (instruction cache only).

5.1.2.3 Verifying load requests from the cache structure

For reading (i.e. load), there are two scenarios which have to be tested: when the cache misses and when the cache hits. However, there are multiple ways the cache may miss during a read, strongly related to the miss categories explained in Section 2.2.1.2. The conditions which are checked in the cache are listed in Section 4.2.1.3.

First is the non-validness of the entry, this occurs when the cache is cleared and the specific entry is not read nor written before. For the data cache, this occurs two times for every set since it is two-way set associative. Second is a non-matching tag within the set. This means another entry is loaded into that cache position, or two distinct cache entries in the case of the two-way set associative data cache. Third is a non-correctable error. This is triggered by setting the control register, regulating the saboteur input, to "01" and "10" subsequently.

For every case, the number of hits, misses and (if applicable) errors detected and corrected are checked. No anomalies have been shown by executing this tailored testcase. This means that the controller works as expected, and that the fault tolerance is also working as advertised.

5.1.2.4 Verifying store requests to the cache structure

The data cache on the other hand also has to be checked whether it writes (i.e. stores) the values in a correct way. Also in this case it may be a miss or hit. However, in this case, the data and valid state are not checked, hence they will be overwritten. Although, the error state of the valid is checked since it may be updated: the old value should not contain uncorrectable errors.

When a miss occurs, the Least Recently Used (LRU) bit is used to determine the way to write in. Otherwise, the entry is written in the way which reported a hit. For a hit, the tag remains the same while the valid signal is updated. At a miss, also the tag and valid have to be written to the new location.

Several write sequences are included in the testcase. They cover the situations which may occur in practice, also including writes with selective byte-enables. Also for these parts in the testcase, no anomalies are experienced when running the testcase.

5.1.3 Simulation of the SoC: problems encountered

Now the cache structures are simulated individually, the next step is to simulate the SoC including both caches. For this, the framework and testcases are used which were applied in [8]. The dual port Random Access Memories (RAMs) are replaced by the caches and the DRAM port is added. Also, the interconnect is included to arbitrate the two caches to one DRAM controller signal.

However, the softcore as used in the previous projects (referenced in Section 2.8) appears to be erroneous in combination with the cache structure. Due to the fact that the core has to wait longer than one clock cycle for its instruction (at a miss), the execution flow is interrupted. It is out of the scope of this project to discuss exactly what goes wrong, since the error is found to be in the fetch stage, one of the five stages in the pipeline.

One of the options is to examine the working of that part and fix the bug within the softcore. However, the employer at that time was already working on a complete new version of the softcore, called Frenox. Therefore, the best option was to switch to this new version, although it was not yet completely tested nor run on an FPGA. This process delayed the workflow of the project. No so-called top-level (to place the softcore on the FPGA) existed yet and some modifications were needed to connect the Frenox to the existing SoC environment.

After moving to the Frenox core, the verification suite testcase is run on the simulated Frenox SoC including the cache structure. This testcase checks whether all instructions work as expected and originates from the creators of the Reduced Instruction Set Computer (RISC)-V Instruction Set Architecture (ISA) [57]. Note that the instruction cache structure is tested functionally already by running the testcase: the instructions have to be read via the instruction cache. Furthermore, a few data entries are being stored and loaded which is an initial test for the data cache. Note that both also rely on the connection to DRAM. The verification suite is applied as an initial test to check the basic functionality.

Closer to a real-life program is the Universal Asynchronous Receiver/Transmitter (UART) testcase. It includes printouts processed by the processor, outputted via UART. The UART is implemented by a built-in component (available from the employer) into the SoC, accessible externally and by the softcore. The program includes a much larger amount of instructions compared to the latter verification testcase and uses the data memory extensively. During this test, it shows that the data cache contains a functional error.

Currently, the data cache includes dual port RAMs for both ways, split into data, valid and tag. One of the ports of each dual port RAM is connected to the softcore, while the other is assigned to the DRAM. Therefore, the address of the second port (to DRAM) is controlled in such a way that the requested data is written to the right position once the data has arrived. However, for the softcore, a problem emerges on the first port. For reading, the request to the dual port RAM has to be made in the same clock cycle that the request from the softcore comes in. If not, the hit time cannot be just one clock cycle. Therefore, the incoming address is fed through directly from the request input to the dual port RAM address port.

However, when writing from the core, first it has to be checked whether the entry is already in the cache or not. This means the tag and LRU memory are read directly. However, the tag, valid, LRU and data are written in the next cycle, after it was decided it was a hit or miss (by checking tag and valid) and the LRU state is known. Since this is done on the same port as reads from the softcore, the address may be changed meanwhile for a read. Hence, when a hit occurs on the write request, an acknowledge is given in the next cycle, allowing the softcore to put another request (e.g. a read) on the input. When a load to a dual port RAMs is performed in the same clock cycle, it will be on the wrong address. Also the read will return a wrong value, i.e. the one that is written simultaneously (assuming a write-before-read policy).

A solution to this problem might be to use the port assigned to the DRAM also for reads from the core. Hence, only the write port is used at the DRAM readback, and only in case of a read miss. Since reads from DRAM always come after checking the cache entries, there can never be a conflict between a read from the core and a write incurred by a read from DRAM. This is important since they share their input address to the dual port RAM. Unfortunately, the time did not permit implementing this solution. Note that it also involves some extra logic since the port used for DRAM accesses is 144 bits wide instead of 36 bits at the core side. Also note that the write from the core could not be merged with the write from DRAM incurred by a read. This is due to the fact that the 144 bit port does not support byte enables. Although note that no timing conflicts would be inferred there either.

To verify the working without this error, a quick fix is applied: delaying the acknowledgment at a write with one additional clock cycle. This means that, during the clock cycle after the write request, the same address is still on the bus. Therefore, the address is also still on the dual port RAM which has to be written, hence the data is written to the right address. With this scheme, the SoC including the cache structure is verified to work properly, even for the more complex programs like the UART testcase and the Dhrystone benchmark. The Dhrystone benchmark will be further discussed in Section 5.3.3.

5.2 Checking behavior on the FPGA

When a simulation turns out to be successful, the design can be synthesized for placement on the FPGA. This is done first for the caches individually, which is not further discussed since this configuration does not have any practical use. However, for this structure it is verified that the cache structures are able to reach the 62.5 MHz requested by the employer, with a maximum of about 70 MHz.

Testing on the FPGA is most interesting when the full SoC is concerned, since this will be the configuration to use in practice. The verification testcase, as mentioned in Section 5.1.3, is also implemented on the FPGA previously for the old RISC-V core. It runs, after some functional modifications to convert to the Frenox core, without any problem for all cache configurations. However, the UART testcase does not output anything, while the Frenox SoC without the caches turns out to work correctly.

For running on the FPGA it holds that any error encountered is very hard to trace since the intermediate signals cannot be seen. A tool enables this (called ChipScope for Xilinx devices), however this could not be put to work for the SoC including the caches. Therefore, reproducing the error in simulation is the only option to find the error. Doing this led to finding the error described before in Section 5.1.3. As discussed there, a quick fix is implemented, showing correct results in simulation. However, the SoC equipped with this fix does not work on the FPGA: the verification suite still works as before, but there is no output on the UART. Some time is invested to find the error, checking many aspects, but the error was not found. Therefore it is also unknown how to fix the error. Note that the target FPGA (the IGLOO2) needs another top-level (the part that is also not simulated). Implementing a new top-level and using other tools may already fix the problem, since in simulation no error was found.

5.3 Results

As became apparent from the two previous sections, the SoC is not working correctly on the FPGA. Therefore, no program can be run on FPGA that outputs a performance metric, such as Dhrystone or Coremark.

To indicate the performance, it is decided to use simulation for that. This is done by running a benchmark in simulation, which is the Dhrystone benchmark. Dhrystone is a little synthetic benchmark, including computational loads with mathematical and other operations while only using integer operations [58]. Compiler settings and other considerations are untouched from the work described in [8].

Another result related to performance is the verification that a cache hit returns the data in the next clock cycle. This is checked by investigating the waveforms generated by running a simulation. For a miss on the other hand, the structure takes 14 cycles to get the right data from the memory in simulation. Note that this includes the full process of an AXI transfer: it is seen in simulation that this process takes about five to six cycles.Since in simulation a BFM is used, this may differentiate from real-life performance. However, it is expected that in reality the process takes at least an even amount of time, so that the structure may take more than 14 cycles in case of a miss. Thereby, simulating the Dhrystone benchmark provides an insight in real-life performance. Probably, the results from simulation must be seen as a best-case scenario.

With these considerations in mind, the Dhrystone benchmark is run in simulation in order to give an indication of the performance. Results of this will be elaborated on at the end of this section. First, the results on area (occupied resources on FPGA) and clock performance are presented and reviewed. Note that energy is not discussed

Table 5.1: Comparison of occupied resources for the total SoC by different cache configurations. Note that also the DRAM controller is included in this SoC.

Configuration					
Instr. cache (kB)		4	4	32	32
Data cache (kB)		4	32	4	32
Write buffer entries		32	32	32	32
Occupied resources					
Slice Look-Up Tables (LUTs):	Total	10726	10982	10766	10971
	Logic	9967	10153	9958	10147
	Route-thrus	360	430	408	424
Occupied slices		3794	3824	3836	3935
LUT Flip-Flop (FF) pairs		12535	12650	12593	12763
RAMB16BWERs		19	27	34	42
RAMB8BWERs		8	12	7	11

separately since it largely depends on the FPGA used. Regarding a specific FPGA, the energy usage is influenced heavily by assumptions made for e.g. toggle rates, indicating how many state changes are experienced by any logical circuit. When these assumptions are made, the energy usage for the prototype FPGA can be estimated by using the Xilinx Power Estimator [59].

5.3.1 Area: occupied resources on FPGA

To estimate the area occupied by the addition of the cache structure, the resource usage results outputted by Xilinx ISE are used. The version used is ISE 14.5, settings used are tabulated in Appendix D, which differ from the settings used in [8]. These settings originate from the current state of the softcore project at the employer.

First an overview is given on the differences encountered when the configurable parameters of the cache are changed. Second, insight is given into the amount of extra resources needed with respect to the stock Frenox SoC. Third it is shown what impact it has when minimizing the write buffer depth. Fourth, an overview is provided comparing the calculated minimum required storage with the actual storage used in block RAM. Lastly a comparison is drawn including the Frenox and (old) RISC-V core as used in the partly adjacent project [8].

5.3.1.1 Resource counts: capacity configuration

First, the differences incurred by the configurability options of the caches are reviewed. They are shown in Table 5.1. Note that resource counts which are (almost) equal are left out of the comparison. The relative increases are also represented in Figure 5.1

Mainly the block RAM (indicated with RAMB16BWER and RAMB8BWER) counts are increased when the caches grow. This is also what could be expected since block RAMs are generally used in FPGAs to store larger quantities of bits. Furthermore, it is seen that a large data cache incurs more slice LUTs in the form of logic, while for both

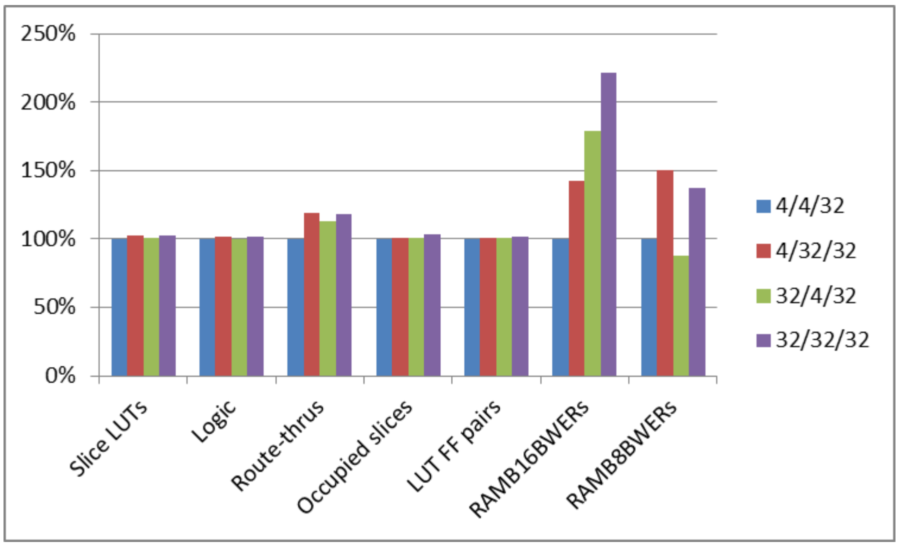


Figure 5.1: Relative resource counts when changing the capacity configurations. First number stands for the kilobytes of instruction cache, second for kilobytes of data cache and third for write buffer entries (90 bits each). The smallest configuration is taken as 100%, being four kilobytes of instruction and data cache, and 32 write buffer entries.

types of caches hold that the route-thrus are increased for larger capacities.

For the occupied slices, LUT FF pairs and total count of block RAMs it seems they scale nicely with the total cache capacity. It can further be concluded that the instruction cache takes more use of RAMB16BWERs where the data cache uses additional RAMB8BWERs. This coheres to the fact that the data cache is divided into multiple smaller memories, while the instruction cache just instantiates one large memory in the VHDL implementation.

5.3.1.2 Resource counts: addition of caches to stock Frenox

Another comparison can be drawn with reference to the stock Frenox SoC without caches. This shows the increase of logic needed to implement the cache controllers and the codecs. Also, by taking 16 kB for all memories, the overhead in bit storage is indicated by these results. The table showing this is Table 5.2. Note that also for this table, the parameters are omitted which do not show a significant difference.

It becomes clear that the cache structure takes a lot more resources compared to the Frenox SoC, even when the memories are equally sized. Of course, there are some legit reasons for this. With respect to occupied memory resources (e.g. RAMs), the cache has to store tags, status bits (valid, LRU) and check bits additionally. Also the write buffer adds to the required memory count.

For logic, it can be seen that the cache controllers require a relatively large share of the resources. Note that also the generated parts, which are expected to be quite complex such as the AXI interconnect, are included in the version with the caches. As a grand total, the increase in resource count on the prototype FPGA is averaged at about

Table 5.2: Resource counts comparison for adding the cache structure to the Frenox SoC.

		Configuration		
		w/o cache	w/ cache	increase (%)
Instr. cache/memory (kB)		16	16	-
Data cache/memory (kB)		16	16	-
Write buffer entries		n/a	32	-
Occupied resources				
Slice Registers:	Total	5264	8571	63
	Flip Flops	5260	8544	62
Slice LUTs:	Total	6458	10852	68
	Logic	5711	9946	74
	Memory: shift register	5	86	1620
	Route-thrus	428	506	18
Occupied slices		2132	3805	78
MUXCYs		1312	2256	72
LUT FF pairs		7283	12545	72
bonded IOBs		41	90	120
RAMB16BWERs		18	24	33
RAMB8BWERs		3	8	167

65-70 %. This is a significant share for adding a cache structure, therefore it should be decided by the employer for any application whether the DRAM access is needed. If not, a lot of resources can be saved by not including the cache structure, but hence that the implementation then has to be modified in order to include fault tolerance. Of course, it may also be possible to only add DRAM without a cache memory applied. However, in Section 5.3.3 it will be shown that performance suffers a lot in this case. Note that the DRAM only configuration is not optimized in any way.

5.3.1.3 Resource counts: addition of the write buffer

In order to see the difference in resource counts due to the write buffer, the default value of 32 is compared with an implementation featuring a buffer depth of two entries. Since one entry caused problems while running the verification suite on the FPGA, two is taken as a minimal working configuration.

Note that every entry is 80 bits wide. For 32 entries, the total memory requirement is therefore 2560 bits, while two entries just take 160 bits. The FPGA storage to use is set to "auto", i.e. the synthesis tool decides whether to implement it as a slice LUT or place it in a block RAM. In Table 5.3 the synthesis results for changed numbers of occupied resources are shown. In Figure 5.2 the relative resource counts are visualized.

The numbers show that the buffer takes more block RAMs than could be expected based on the total bits stored. The difference of three RAMB8BWER blocks account for nine kb each, i.e. 27 kb. Note that the required 2560 bits (less than three kb) should fit easily in just one instance of this block. When decreasing the number of write buffer

Table 5.3: Resource counts comparison for two configurations of write buffer depth.

Configuration					
Instruction cache (kB)		4	4	32	32
Data cache (kB)		4	4	32	32
Write buffer entries		2	32	2	32
Resource counts					
Slice Registers:	Total	8641	8573	8646	8578
	Flip Flops	8614	8546	8619	8551
Slice LUTs:	Total	10815	10726	10977	10971
	Logic	9940	9967	10124	10147
	Memory: dual port	314	234	314	234
	Memory: single port	78	79	80	80
	Route-thrus	397	360	373	424
Occupied slices		3828	3794	3867	3935
LUT FF pairs		12625	12535	12751	12763
RAMB16BWERs		19	19	42	42
RAMB8BWERs		5	8	8	11

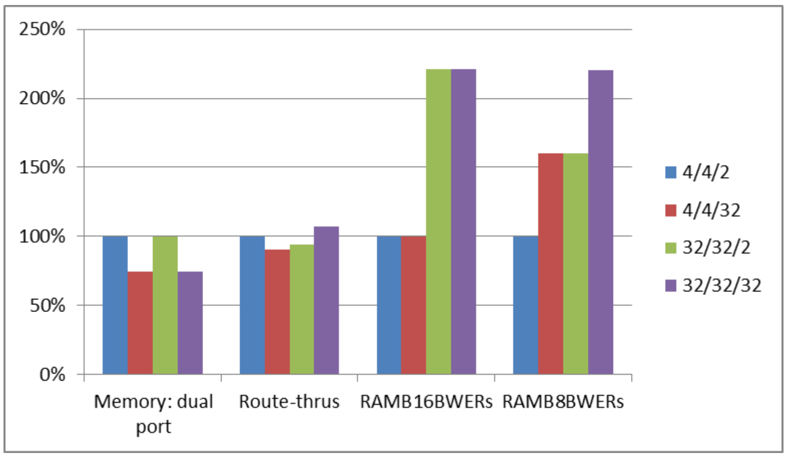


Figure 5.2: Relative resource counts for the most significant differences when configuring the write buffer depth. Legenda: first number indicates kilobytes of instruction cache, second kilobytes of data cache and third the number of buffer entries.

entries from 32 to two, the increase in dual port memory, which is a slice LUT in the prototype FPGA, is exactly 80 entries. This means that the small write buffer is most likely fully implemented in this type of memory, saving costly block RAM instances.

5.3.1.4 Resource counts: comparison of calculated minimum and real memory usage

Another comparison to make is with respect to the calculated minimum required memory usage. As seen with the buffer, the block RAMs often cannot be used fully resulting in

Table 5.4: Overview of memory usage as calculated (minimum) and implemented on the prototype FPGA.

	Configuration										
	Frenox w/ cache							Frenox stock		RISC-V	
Instr. cache/mem. (kB)	4	32	4	32	16	4	32	64	16	64	16
Data cache/mem. (kB)	4	32	32	4	16	4	32	128	16	128	16
Write buffer entries	32	32	32	32	32	2	2				
Occupied block RAM counts											
RAMB16BWERs	19	42	27	34	24	19	42	98	18	98	17
RAMB8BWERs	8	11	12	7	8	5	8	3	3	3	3
Comparisons											
Occupied mem. (kB)	51.8	107	74.3	84.4	63.0	48.4	104	224	43.9	224	41.6
Calculated mem. (kB)	12.1	92.7	53.4	51.3	46.8	11.7	92.4	192	32	192	32
Overhead (%)	330	15.3	39.0	64.4	34.8	312	12.0	16.6	37.1	16.6	30.1

inefficient usage of the available memory, which is relatively scarce on FPGAs. All configurations discussed before are included in Table 5.4. The second RISC-V entry in the table (configuration 16/16 kB) is taken from [8], while the first is generated from the last version of the RISC-V softcore project. Furthermore, note that RAMB16BWERs feature 18 kb each (or 2.25 kB) while RAMB8BWERs include 9 kb of storage (or 1.125 kB).

When looking at efficiencies overall, it turns out the addition of the cache structure does not make the memory usage less efficient. But note that the cache has to store more bits for every instruction or data word, therefore requiring increased amount of memory resources in total. Efficiencies for small caches are dramatic at using four times as much memory than actually needed. This is probably due to the fact that the synthesis tools takes at least one block RAM for every instantiation, which means for data cache at least thirteen RAMs has to be used. Hence, to achieve byte enable support the data memory is implemented as four smaller RAMs. Featuring two instances of these (one for every way), the data takes eight RAMs already. Two for tag, two for valid and one for LRU brings the total on thirteen.

Most efficient, in terms of block RAMs, is the implementation including the minimized write buffer. Note that this takes some more instances of dual port memory as seen in Table 5.3.

5.3.1.5 Resource counts: Frenox vs. (old) RISC-V core

For reference to the old softcore of the employer used in [8], a similar configuration is compared to the reported results found in referenced thesis project. See table Table 5.5. It can be seen that Frenox uses more resources than the old RISC-V softcore for an equal configuration of instruction and data memory. Since the core itself is out of the scope of this thesis project, the differences will not be discussed in more detail. However, note that the differences may also (partly) be caused by the different settings of the synthesis tool (Xilinx ISE), as found in Appendix D.

Table 5.5: Comparison between the Frenox core and (old) RISC-V softcore. The RISC-V results are taken from [8]. Both SoCs feature 16 kB of data and instruction memory.

	Frenox	RISC-V
Slice Registers	5264	4534
Slice LUT	6458	5155
Occupied slices	2132	1894
RAMB16BWERs	18	17
RAMB8BWERs	3	3

5.3.2 Timing analysis

While the clock performance of the cache structure individually is verified to meet the contraint of 62.5 MHz, topping at about 70 MHz, the results for the total SoC differ a lot. It turns out the synthesis tool for configurations just reaches 31.25 MHz, which is half of the frequency required.

Reason for this large decrease is one of the connections made by the softcore, as is seen by using PlanAhead integrated in the Xilinx ISE tool. It shows that, after a read hit in the data cache is signaled which results in an acknowledge, the core features a path that outputs the same signal to the data cache again in the form of a write. Technically, this means a Read-Modify-Write (RMW) sequence, which is known to take considerable time in an FPGA.

For the smallest configuration, featuring four kB of instruction, four kB of data cache and two write buffer entries, the longest path is about 30.2 ns. For other configurations this number changes slightly towards 31 ns. A longest path of 30.2 ns translates to a maximum frequency of about 33 MHz. It is expected that when the critical path in the softcore is removed, and any other RMW sequence (if present), the frequency may be increased up to the targeted 62.5 MHz at optimal settings.

5.3.3 Dhrystone results

Since the SoC including the cache structure contains an error, benchmarks like Dhrystone do not display any result from the FPGA. Therefore, it is decided to run the Dhrystone benchmark in simulation to provide an insight in the performance figures between different configurations. However, in simulation it takes a very long time to run for e.g. one second. The default number of runs (2,000,000) is therefore not feasible: it would take approximately 140 days in the current setup. Therefore, a small number of runs is taken as default to compare the performance for different configurations. The results for this will be discussed first. After that, an insight is provided into the performance of the Frenox core versus the old RISC-V implementation, and how the numbers obtained from the FPGA compare to those from simulation.

5.3.3.1 Dhrystone figures for the cache (obtained by simulation)

Simulating a Dhrystone benchmark takes considerable time. Therefore, the default run count is set to 100. In order to get an indication of how the numbers change for larger run

Table 5.6: Dhrystone results for different (cache) configurations, obtained by simulation. Note that the first entry without cache structure is only including the connection to DRAM, while the second only features on-board memory.

Configuration	With cache				Without cache	
Instr. cache/mem. (kB)	4	32	32	32	n/a	64
Data cache/mem. (kB)	4	32	32	32	n/a	128
Write buffer entries	32	32	32	32	32	n/a
LLM: DRAM	Yes	Yes	Yes	Yes	Yes	No
Variables						
Frequency (MHz)	62.5	62.5	62.5	62.5	62.5	62.5
Number of runs	100	100	400	10000	100	100
Run time (us)	1760	1759	6876	170715	11901	1634
Results						
Dhry/sec	568182	568505	581734	585772	84027	611995
Dhry/(s*MHz)	909	910	931	937	134	979
DMIPS/MHz	0.52	0.52	0.53	0.53	0.08	0.56

counts, one simulation is included with 10,000 runs through the Dhrystone code. To see the performance impact by adding a cache structure, the benchmark is also simulated once while both caches are functionally disabled. This means that only the DRAM remains as single memory device. Disabling the caches functionally is done by setting the fault injection parameter for both caches to uncorrectable errors, making sure every cache access will result in a miss. Functionally, this is equal to a system without any cache memory. Since the write buffer cannot be disabled in such a way, this part is left untouched. The second entry in the following results without cache is the stock Frenox SoC with its integrated memories.

The results are to be seen in Table 5.6. From the table, it can be concluded that the different cache configurations do not introduce a large difference in performance for this specific benchmark. This may be due to the fact the Dhrystone benchmark is relatively small and may fit into the smallest cache configuration.

The number of runs influence the results a little more, but not significantly: just by a few percent. Comparing to the Frenox SoC excluding the cache, no big difference is observed which also indicates that the Dhrystone benchmark in total probably even fits into the smallest instruction cache.

A large performance gap is seen for the Frenox SoC without caches and without on-board memory, only featuring the DRAM. There it can be seen how large the performance increase could be when adding a cache structure: the performance is just 14% compared to the SoC including the caches.

Note that the implemented solution to the earlier described problem, i.e. waiting an extra cycle at a write hit, could influence performance for the structure including the data cache. However, it is expected this influence is small.

Table 5.7: Dhrystone results comparing (old) RISC-V softcore with Frenox and simulation with FPGA results.

	Configuration				
	Frenox with cache	Frenox without cache			RISC-V
Sim./FPGA	Sim.	Sim.	FPGA	FPGA	FPGA
Freq. (MHz)	62.5	62.5	62.5	62.5	83.3
No. runs	10000	100	10000	2000000	2000000
	Results				
Run time (us)	170715	1634	161799	32098265	24200000
Dhry/sec	585772	611995	618051	623087	826446
Dhry/(s*MHz)	937	979	989	997	992
DMIPS/MHz	0.53	0.56	0.56	0.57	0.57

5.3.3.2 Comparison of Dhrystone figures to RISC-V and Frenox on FPGA

To place the Dhrystone figures obtained by simulation into context, Dhrystone runs are also performed on the FPGA for the stock Frenox SoC without caches. For reference, also the result reported in [8] is taken into account, which represents the (old) RISC-V core. All results are gathered in Table 5.7.

It is seen that there is little difference in the actual performance per MHz. This means that the simulation is representative for the Frenox SoC. Note that it can not be verified that also the interface to the DRAM is adequately represented in simulation with respect to timing, since no successful run could be done on FPGA including the DRAM.

5.4 Verification and results: summary and conclusion

This chapter described the verification of the codecs, cache controllers and SoC, as well as the results for resource utilization on the FPGA, timing analysis and the Dhrystone benchmark. As a summarizing conclusion, these will be reviewed briefly.

5.4.1 Verification

Verification is done both in simulation and on the prototype FPGA, being a Xilinx Spartan 6. In simulation, all codecs are tested separately on their encoding and decoding capabilities. Furthermore, the cache controllers are tested on their working and fault tolerance. The latter is done by including a saboteur and a control register to set the level of (static) fault injection. For both, additional counter registers are used in order to check the behavior.

The SoC is also tested in simulation including the caches and DRAM interface, connected to a BFM. When running the UART testcase, an error condition emerged for the data cache. A quick fix was applied which delays any write hit with one additional clock cycle. A better solution is proposed but not implemented due to limiting time.

The verification on FPGA includes a verification suite which is a very simple and tiny program, checking the working of all instructions as well as the loading and storing of some values. This suite is executed correctly for the SoC including the caches and DRAM connection. However, any other testcase or benchmark using the UART output failed: no output was observed on this port. Therefore, the envisioned Dhrystone and Coremark benchmarks could not be run on FPGA.

5.4.2 Results: area, timing and Dhrystone

The results section started with investigating the occupied resources on the prototype FPGA.

First it was verified that the configuration does not have a larger than expected impact on resources, and insight was given in the numbers for different configurations.

Second, it is shown that adding any cache configuration requires quite a lot of extra resources on the FPGA. Compared to the Frenox SoC excluding the cache memory structure, the increase is about 65 to 70 percent in total.

Third, the write buffer is being minimized to two entries in order to compare the resource utilization. Results show that a write buffer with 32 entries is placed in block RAM in an inefficient way, occupying more than three times as many memory than needed. When just two entries are used, the memory is placed in slice LUTs, namely dual port memory.

Fourth, an overview is provided of the storing efficiencies by comparing the occupied block RAM storage with the calculated required storage. It turns out that small caches are less efficient, but larger cache configurations score equal or even better in efficiency compared to the implementations without the cache structure, hence only including the on-board memory.

Lastly, the Frenox softcore is compared to the old RISC-V implementation with respect to resource counts. It turns out that the Frenox softcore, as is used to connect the caches to, is slightly larger than the old RISC-V implementation.

For timing, in the verification phase it was verified that 62.5 MHz could be reached with the individual caches only. However, integrated in the SoC, this frequency cannot be reached. Even half of it, i.e. 31.25 MHz, is hard to achieve on the prototype FPGA. The longest path is investigated showing an RMW sequence beginning and ending in the data cache, via the softcore. When this is solved, it is expected the frequency can be increased.

The Dhrystone could not be run on FPGA for the SoC including the caches. However, simulation is able to give an indication of the performance by decreasing the number of Dhrystone runs.

Simulation results show a little decrease in performance compared to the Frenox softcore with its integrated memory. This could be expected since DRAM accesses take up a lot of time. The latter is also confirmed by a very low score for running the softcore with DRAM only.

As a last result, the Dhrystone benchmark is also run on the FPGA for the softcore without the caches. These results show that little difference is experienced moving from simulation to the FPGA. Note that it can not be said how well the DRAM access times in simulation match those on the FPGA.

6 Conclusion: discussion and future work

In this concluding chapter, first the work done is summarized and concluded. Second, future work is proposed to fix, optimize and/or extend the current state of the product.

6.1 Summary and conclusions

This part will summarize the thesis work by first briefly restating the problem. After that, the structure of the report is followed to indicate the most important contributions towards the final product.

6.1.1 Problem restatement: connecting a softcore to DRAM and implementing a cache structure

The goal of this thesis project is to connect the existing softcore, based on Reduced Instruction Set Computer (RISC)-V, from Technolution B.V. to the integrated Dynamic RAM (DRAM) controller on the provided Field Programmable Gate Array (FPGA). Furthermore, to build a cache structure in order to mitigate the performance impact due to the move from on-board memory to DRAM as the main memory location. This cache structure has to be configurable in capacity, feature fault tolerance and maintain a hit time of a single clock cycle at 62.5 MHz.

6.1.2 Investigated literature: problem tailored to Advanced eXtensible Interface (AXI), Hsiao and parameters

The literature survey provided a lot of insights, e.g. into ways of adding fault tolerance to a memory architecture. One of the main findings, which is also applied in the final product, is the base for deciding upon a fault tolerance scheme. The survey brought forward that Error Detection And Correction (EDAC) code is the most prevalent form of fault tolerance when memory architectures are concerned. From the range of EDAC codes, parity code is the most simple and provides error detection for one error only. Hsiao Single Error Correcting/Double Error Detecting (SEC/DED) code is the most efficient to implement in logic when looking at Error Correcting Codes (ECCs).

Another insight gained is the application possibilities of the most prevalent standardized bus technology, the AXI protocol by ARM^{TM}. Since this bus is supported by both the manufacturer of the prototype FPGA and the targeted FPGA, it is the best option to integrate into the structure.

More general, the literature survey described which parameters would fit the cache structure, also based on requirements set by the employer. These considerations did also conclude that an FPGA base exhibits some restrictions in the possibilities, e.g. sizing the memory.

6.1.3 Design considerations: defining the capacities, fault tolerance measures and parameters

The design phase considered the information from the literature study and the requirements put by the employer. It resulted in the design of the final product which implements a connection to DRAM and two cache structures: one for data and one for instructions. For both cache structures, the capacity is configurable, featuring the options of 4, 8, 16 and 32 kilobytes for both. Furthermore, the block size is 128 bit while the word size is 32 bit. The latter originates from the softcore the caches connect to as slave devices. 128 bit block size is practical for optimal throughput to the DRAM. Therefore, the AXI interface, for which the cache controllers act as master devices, is 128 bit wide with respect to data width.

The instruction cache is a read-only, direct-mapped cache structure. Data cache on the other hand is also writable and two-way set associative, meaning that any entry from DRAM may reside on two locations. For writing the data, an Least Recently Used (LRU) policy is chosen, replacing the least recently used entry when a new entry has to be written. Data also may be written per byte.

Due to the latter fact, the data inside the data cache structure is protected per byte. Parity is chosen due to the large relative overhead the other schemes would incur. Since the parity scheme is only capable of detecting an error, correction is done by re-fetching the entry from DRAM. For this, a write-through (WT) scheme is required, which means that every write is also directly fed through to the DRAM.

All other information bits residing in either one of the cache structures is protected by Hsiao SEC/DED code. The information concerned are:

1. the tag/valid combination in the instruction cache as well as the instructions themselves,

2. the tag, valid and write buffer entries in the data cache structure.

6.1.4 Realization: implementing the cache controllers and DRAM interface

Both cache controllers are implemented like described in the design part. Also the needed codecs, i.e. encoders and decoders, are implemented for all required widths and integrated subsequently. The write buffer uses a First In First Out (FIFO) component made available by the employer. For the DRAM interface, several components have been generated in the tooling provided for the prototype FPGA. These components include an AXI to Memory Controller Block (MCB) bridge and an AXI interconnect. The latter takes care of the arbitration between three master devices: the two cache controllers and one port to initialize the memory externally. The AXI to MCB bridge is needed since the integrated DRAM controller on the prototype FPGA only accepts the Xilinx-specific MCB bus. The MCB itself, acting as the DRAM controller, is instantiated by a modified component from the generated sources which can be obtained from the FPGA tooling.

6.1.5 Verification and results: significant area usage, adequate speed and remaining problem

The last main part of the report describes how the implementation is verified and indicates the (relative) performance on the prototype FPGA. Starting from the codecs, via the controllers to the total system, the total implementation is being verified. This is done primarily in simulation using MentorGraphics ModelSim 10.3d (Altera Starter Edition). The cache structures and total system are also tested on the FPGA. For the cache structures only, it is verified they are able to reach 62.5 MHz as requested by the employer, while the hit time was verified in an earlier stage to be one clock cycle. However, for the total system just over 30 MHz could be achieved, due to an Read-Modify-Write (RMW) sequence made possible by the provided softcore. Note that the provided softcore differentiates from the one that resulted from other student-driven projects. The new softcore called Frenox is a rebuilt version of the softcore implemented before. The old version does not work in combination with the cache structure due to an erroneous implementation of the fetch stage.

Although the new softcore is verified to work, the Frenox including the cache structure, and hence the DRAM connection, does not work as expected. The problem arises that the Universal Asynchronous Receiver/Transmitter (UART) does not output any data, while in simulation this part of the System on Chip (SoC) works as it should. This problem is not solved in this thesis project and will be further discussed in Section 6.2.

Results that could be gathered are mainly related to area and performance. Note that these results are highly dependent on the FPGA chosen. The first is based on the resource count reported by Xilinx ISE (version 14.5) after generating the program file to place on the FPGA. Results with respect to this parameter show that the relative resource count is fairly high compared to the Frenox SoC with integrated memory only: a 65-70 % increase overall. However, it is noted that this figure also includes all generated components such as the DRAM controller and the AXI interconnect. These are expected to be quite complex, taking up a considerable share of resources. Also the codecs take their share: the original SoC does not include fault tolerance for its memories. Also, with respect to the occupied memory, the cache structure has to store additional bits, when compared to the same functional capacity used as plain internal memory. These additional bits include the tag, valid, LRU, buffered entries and check bits. When sizing the cache capacities, the memories scale as expected. However, it is seen that small caches can not make efficient use of the integrated block Random Access Memories (RAMs) on the prototype FPGA. Therefore, larger capacities result in a higher efficiency with respect to used block RAMs than small configurations. Also the write buffer is inefficiently mapped to block RAMs when the default depth of 32 is chosen. For a minimal value of two entries, these block RAMs are released and the buffer moves to slice Look-Up Tables (LUTs), which might be advantageous when a small FPGA model is targeted.

Performance measurements for the SoC including caches and DRAM are performed in simulation, using the small synthetic benchmark called Dhrystone. The default number of runs was lowered to make simulation times feasible. Note that, related to this, it is shown that the difference when using a higher number of runs is negligible. In simulation, the SoC including the caches and DRAM interface performs almost equal to the SoC with integrated memory only. This is due to the small size of the Dhrystone benchmark, fitting in the caches easily. When the cache functionality is disabled through fault injection, it turns out the performance suffers a lot when only the DRAM can be used as the main memory device. For the Frenox SoC without the cache structure and DRAM connection, it is verified that the simulation results cohere to those obtained from the FPGA.

A remaining problem not solved properly, resides in the data cache. It is fixed provisionally by a quick fix changing the hit latency for a write from one to two clock cycles. A small performance decrease could be expected by this change, depending on the program run and the buffer depth.

6.2 Future work

As also becomes apparent from the previous section, there is some work left to finalize the cache structures including the DRAM interface. Also, the author wants to propose future work to expand and/or optimize the product, e.g. optimizing the performance. The following recommended improvements will be explained briefly subsequently:

1. fix the error in the data cache,
2. add a 'heavy' protection scheme,
3. optimize read/write transactions (coalescing),
4. removing any RMW sequence in the softcore,
5. moving to the target FPGA (or any other more modern model).
6. find and fix the problem that no UART output is shown,
7. select and implement a more relevant benchmark,
8. optimize Xilinx ISE settings.

6.2.1 Fix: error in the data cache

The error found in the data cache is provisionally fixed for now since time did not permit implementing a neat solution. However, as described at the end of Section 5.1.3, a better solution is available. In short, the ports of the dual port RAMs within the VHDL implementation have to be reassigned. This is impeded by the fact that for the data dual port RAM, one port is 144 bits wide while the other is just 36 bits. It is expected this could be fixed within a reasonable time when the designer understands the structure of the cache controller.

Advantage of fixing this error is that the hit time for writing to the data cache can be brought back from two clock cycles to only one, as was enabled by including a write buffer. However, care has to be taken that in the case of simultaneous reading and writing, the write will be processed first. This might not be supported by every FPGA model.

6.2.2 Extension: 'heavy' protection scheme

Up till the design phase, described in Chapter 3, three fault tolerance schemes were discussed. Two of them were envisioned by the employer, due to time constraints only one scheme ('light') was implemented. The second is the 'heavy' protection scheme. It includes Triple Error Detecting (TED) code for protecting data bytes and Quintuple Error Detection (5-ED) for all other information bit quantities. It rises the resilience to soft errors to such a high level that human attacks are very unlikely to not being detected. This is also due to the design of the Static RAM (SRAM) devices in the targeted FPGA being the IGLOO2 of MicroSemi.

A third option may also be considered, especially when the heavy protection scheme is implemented. It uses the same EDAC code as the heavy scheme, but applies it as SEC/DED for bytes and Double Error Correcting/Triple Error Detecting (DEC/TED) for any other information bit quantity. Additional optimization, which can be done for this scheme, is to place the bits needed for correction into the DRAM to save memory occupation within the FPGA at the price of some additional logic.

Note that, when implementing the heavy protection scheme, it might also be needed to add protection to the DRAM. With the IGLOO2, the attached DRAM is optionally protected with SEC/DED code per word. This suits to the light protection scheme, but might be inadequate for the heavy scheme, which should be decided by the employer. Since the DRAM controller is usually a pre-defined block, this might be hard to realize in an FPGA.

Another note is that currently, the buses used are not fault-tolerant themselves since this lies out of the scope of this project. However, also this option is probably critical when moving to the heavy protection scheme.

6.2.3 Optimization: coalescing reads to and writes from DRAM

As was noticed at the beginning of Section 5.3, the AXI protocol takes a considerable time of handling one load from or store to DRAM. However, the AXI protocol is burst-based. This means that it is designed to support multi-beat transactions, with a beat being a single data transaction. In this thesis project, only one beat (transaction of data) is done for every burst. When this number is increased, less overhead in latency will be incurred for every data transaction. Therefore, the throughput of the AXI bus could be improved a lot by combining reads and/or writes.

For writes, it is even worse since they originate directly from the softcore, providing a number of bytes up to 32 bits in total. Therefore, in the best case only a quarter of the data transferred by a single burst in AXI is useful data. Write coalescing would therefore be very helpful to improve useful AXI throughput, even without increasing the burst size (i.e. number of beats per transaction).

6.2.4 Fix/optimization: remove RMW sequence(s) within the softcore

When integrating the SoC with the cache structure and DRAM connection, it became apparent that the timing was much worse than when only the cache structure was concerned. See also Section 5.3.2. It was observed that this was due to a RMW sequence introduced by the softcore provided. Since the softcore itself is out of the scope of this thesis project, no further analysis is done how to solve this problem. However, if this problem could be fixed, the clock frequency may be increased quite a lot. Note that it could not be checked whether any other RMW sequence occurs: the reported one introduces the longest path. After fixing, it should be checked whether there are more RMW sequences.

6.2.5 Extension/optimization: move to the targeted FPGA (or any other modern model)

The FPGA used for prototyping is the quite old (2009) Spartan 6 from Xilinx. Since then, many FPGAs have been released featuring all kinds of improvements and optimizations. Specifically, the targeted FPGA, the IGLOO2 from MicroSemi, is more modern and is already expected to show a significant leap with respect to performance. Also, the tooling will be more up-to-date, which means more optimized modules like DRAM controllers might be included.

The reason why this FPGA is not used is two-fold. First, the IGLOO2 is flash-based, which means configuration times are relatively long. This makes a flash-based FPGA unsuitable for prototyping. Second, the Spartan-6 was available during the entire project within Technolution, as well as the knowledge about this and other Xilinx devices. Moving to the IGLOO2 would incur another tooling set and additional problems may be found since standardized components available within Technolution are targeted and tested on Xilinx devices.

This recommendation is logical in a sense that the employer indicated the IGLOO2 as the target device, hence some assumptions are also made based on this given fact. But it is also recommended to move to a more comparable FPGA in terms of possibilities and speed, in order to get a better insight into the limitations and bottlenecks encountered in more modern FPGAs. Ideally, this would be an SRAM-based FPGA for fast prototyping. Furthermore, if possible, it should come from MicroSemi to get used to their tooling.

6.2.6 Fix: make the UART output work on the Spartan 6

When the employer decides that they want to continue working with the Spartan 6 prototyping board, it should be clear the error has to be found which was described in Section 5.2. When moving to another prototyping FPGA, the problem may be fixed by just re-implementing a top-level for that specific architecture.

The UART output is essential to output e.g. benchmark results from a Dhrystone run. Unfortunately, it is not yet clear where the error may be located since in simulation the SoC works as expected, including the UART output.

6.2.7 Extension/optimization: select and implement a more relevant benchmark

Since the Dhrystone benchmark is very small, it fits even in the smallest cache memory. Therefore, the benchmark results may be not representative enough for real-life applications. In this thesis project, also the Coremark benchmark [60] was envisioned to be included for the results, however time ran out when trying to adapt the benchmark for the Frenox SoC. Some additional work is needed to map the benchmark to the RISC-V architecture of the core. Note that also the Coremark benchmark should output its results via UART. Hence, when the Spartan 6 is used, the error has to be found first why it does not output any UART signals.

With the results of that benchmark, it also may become clearer what to optimize for in future work on the Frenox softcore as well as the cache structure.

6.2.8 Optimization: reviewing the Xilinx ISE settings

For this thesis project, no attention has been paid to optimizing the Xilinx ISE settings. These settings may have a significant effect for e.g. the occupied resources. The settings applied in this thesis project are found in Appendix D and are taken from the RISC-V project in its most current state available at the employer. Note that in [8], different settings were applied. There, some optimizations were disabled explicitly such as register balancing and duplication. Therefore, it is assumed that the settings were optimized after the work from [8] had finished. This means that there might be little to gain with respect to the settings as they are applied already.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] Microsemi corp. Single Event Effects (SEE). Visited on January 27$^{\text{th}}$ 2016. [Online]. Available: http://www.microsemi.com/products/fpga-soc/reliability/see

[3] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *Reliability Physics Symposium (IRPS), 2011 IEEE International*, April 2011, pp. 5B.4.1–5B.4.7.

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

[5] S. Cha and H. Yoon, "Efficient implementation of single error correction and double error detection code with check bit pre-computation for memories," *JSTS: Journal of Semiconductor Technology and Science*, vol. 12, no. 4, pp. 418–425, 2012.

[6] Microsemi corp., "IGLOO2 and SmartFusion2 65nm commercial flash FPGAs: Interim summary of radiation test results," Tech. Rep., October 2014.

[7] ——, "Application note AC424: IGLOO2 - Optimizing DDR Controller for Improved Efficiency - Libero SoC v11.6," Tech. Rep., October 2015.

[8] W.F. Heida. Towards a fault tolerant RISC-V softcore. Visited on November 7$^{\text{th}}$ 2016. [Online]. Available: http://resolver.tudelft.nl/uuid:cee5e97b-d023-4e27-8cb6-75522528e62d

[9] V. Castano and I. Schagaev, *Resilient Computer System Design.* Springer International Publishing, 2015.

[10] C. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 397–404, Sept 2005.

[11] P. Lala, "A single error correcting and double error detecting coding scheme for computer memory systems," in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, Nov 2003, pp. 235–241.

[12] M. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, July 1970.

[13] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "Soft error and energy consumption interactions: a data cache perspective," in *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, Aug 2004, pp. 132–137.

[14] A. Gil, J. Benitez, M. Calvio, and E. Gomez, “Reconfigurable cache implemented on an FPGA,” in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, Dec 2010, pp. 250–255.

[15] UC Berkeley. The RISC-V instruction set architecture. Visited on January 12th 2016. [Online]. Available: http://riscv.org/

[16] Microsemi corp. IGLOO2 FPGAs. Visited on January 12th 2016. [Online]. Available: http://www.microsemi.com/products/fpga-soc/fpga/igloo2-fpga

[17] J. H. A. Shyamkumar Thoziyoor, Naveen Muralimanohar and N. Jouppi. CACTI. Visited on February 4th 2016. [Online]. Available: http://www.hpl.hp.com/research/cacti/

[18] H.-S. Lee, G. Tyson, and M. Farrens, “Eager writeback-a technique for improving bandwidth utilization,” in *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, 2000, pp. 11–21.

[19] I. C. Society, P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014.

[20] R. W. Hamming, “Error detecting and error correcting codes,” *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950. [Online]. Available: http://dx.doi.org/10.1002/j.1538-7305.1950.tb00463.x

[21] S. Miremadi and H. Zarandi, “Reliability of protecting techniques used in fault-tolerant cache memories,” in *Electrical and Computer Engineering, 2005. Canadian Conference on*, May 2005, pp. 820–823.

[22] S. Karp and B. Gilbert, “Digital system design in the presence of single event upsets,” *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 29, no. 2, pp. 310–316, Apr 1993.

[23] S. Kim and A. Somani, “Area efficient architectures for information integrity in cache memories,” in *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, 1999, pp. 246–255.

[24] D. Rossi, N. Timoncini, M. Spica, and C. Metra, “Error correcting code analysis for cache memory high reliability and performance,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[25] T. Fuja, C. Heegard, and R. Goodman, “Linear sum codes for random access memories,” *Computers, IEEE Transactions on*, vol. 37, no. 9, pp. 1030–1042, Sep 1988.

[26] I. Lee, M. Sullivan, E. Krimer, D. W. Kim, M. Basoglu, D. H. Yoon, L. Kaplan, and M. Erez, “Survey of error and fault detection mechanisms,” 2012.

[27] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt, "Cache scrubbing in microprocessors: Myth or necessity?" in *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*, ser. PRDC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 37–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=977407.978748

[28] R. Naseer and J. Draper, "Parallel double error correcting code design to mitigate multi-bit upsets in SRAMs," in *Solid-State Circuits Conference, 2008. ESSCIRC 2008. 34th European*, Sept 2008, pp. 222–225.

[29] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 2002, pp. 409–415.

[30] ARM Ltd. Advanced Microcontroller Bus Architecture (AMBA) specification - homepage. Visited on January 18th 2016. [Online]. Available: http://www.arm.com/products/system-ip/amba-specifications.php

[31] C. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, March 1984.

[32] D. H. Yoon and M. Erez, "Memory mapped ECC: Low-cost error protection for last level caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 116–127. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555771

[33] A. Hocquenghem, "Codes Correcteurs d'Erreurs," *Chiffres (Paris)*, vol. 2, pp. 147–156, September 1959.

[34] R. Bose and D. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019995860902874

[35] H. Farbeh and S. Miremadi, "PSP-Cache: A low-cost fault-tolerant cache memory architecture," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–4.

[36] M. Anwar, P. Lala, and P. Thenappan, "Decoder design for a new single error correcting/double error detecting code."

[37] N. Rollins, M. Fuller, and M. Wirthlin, "A comparison of fault-tolerant memories in SRAM-based FPGAs," in *Aerospace Conference, 2010 IEEE*, March 2010, pp. 1–12.

[38] A. Santana Gil, F. Quiles Latorre, M. Hernandez Calvino, E. Herruzo Gomez, and J. Benavides Benitez, "Optimizing the physical implementation of a reconfigurable cache," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, Dec 2012, pp. 1–6.

[39] O. Novac, S. Janos, S. Vari-Kakas, and K. Che-Soong, “Reliability increasing method using a SEC-DED Hsiao code to cache memories, implemented with FPGA circuits,” 2011.

[40] O. Novac, J. Sztrik, and C. Grava, “Comparative study regarding two implementations of an SEC-DED code with FPGA circuits,” in *Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on*, June 2015, pp. 1–4.

[41] P. Yiannacouras and J. Rose, “A parameterized automatic cache generator for FPGAs,” in *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, Dec 2003, pp. 324–327.

[42] K. Sohn, T. Na, I. Song, Y. Shim, W. Bae, S. Kang, D. Lee, H. Jung, S. Hyun, H. Jeoung, K.-W. Lee, J.-S. Park, J. Lee, B. Lee, I. Jun, J. Park, J. Park, H. Choi, S. Kim, H. Chung, Y. Choi, D.-H. Jung, B. Kim, J.-H. Choi, S.-J. Jang, C.-W. Kim, J.-B. Lee, and J. S. Choi, “A 1.2 V 30 nm 3.2 Gb/s/pin 4 Gb DDR4 SDRAM with dual-error detection and PVT-tolerant data-fetch scheme,” *Solid-State Circuits, IEEE Journal of*, vol. 48, no. 1, pp. 168–177, Jan 2013.

[43] JEDEC. Main memory: Double Data Rate (DDR)3 & DDR4 Synchronous DRAM (SDRAM). Visited on January 20th 2016. [Online]. Available: https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram

[44] N. DeBardeleben, S. Blanchard, V. Sridharan, S. Gurumurthi, J. Stearley, K. Ferreira, and J. Shalf, “Extra bits on SRAM and DRAM errors - more data from the field,” *Silicon Errors in Logic - System Effects (SELSE-10), Stanford University*, April 1, 2014 2014.

[45] ARM Ltd., “AMBA® 3 APB™ Protocol Specification v1.0,” Tech. Rep., August 2004.

[46] ——, “AMBA® AXI™ Protocol Specification v1.0,” Tech. Rep., March 2004.

[47] ——, “AMBA® AXI™ and ACE™ Protocol Specification, year = 2011,,” Tech. Rep.

[48] ——, “AMBA® 3 AHB-Lite™ Protocol Specification v1.0,” Tech. Rep., June 2006.

[49] Microsemi corp., “IGLOO2 FPGAs Product Brief,” Tech. Rep., October 2015.

[50] M. Valderas, P. Peronnard, C. Ongil, R. Ecoffet, F. Bezerra, and R. Velazco, “Two complementary approaches for studying the effects of SEUs on digital processors,” *Nuclear Science, IEEE Transactions on*, vol. 54, no. 4, pp. 924–928, Aug 2007.

[51] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil, “Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection,” *Computers, IEEE Transactions on*, vol. 61, no. 3, pp. 313–322, March 2012.

[52] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[53] Microsemi corp., "IGLOO2 FPGA and SmartFusion2 SoC FPGA DS0128 Datasheet, Revision 9," Tech. Rep., December 2015.

[54] S. G. T. D. Priya Mathew, Lismi Augustine, "Hardware implementation of (63, 51) BCH encoder and decoder for WBAN using LFSR and BMA," *International Journal on Information Theory (IJIT)*, vol. 3, no. 3, pp. 1–11, July 2014.

[55] I. Koren and C. M. Krishna. (2005) Two-bit correcting Hamming Code. [Online]. Available: http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/TwoBit/description.htm

[56] T. E. GmbH. (2015) Trenz Electronic TE0600 Series (LX45, LX100, LS150). [Online]. Available: http://www.trenz-electronic.de/products/fpga-boards/trenz-electronic/te0600.html

[57] UC Berkeley. Verification suite - RISC-V. Visited on November 29th 2016. [Online]. Available: https://riscv.org/software-tools/riscv-tests/

[58] A. R. Weiss, "Dhrystone benchmark," *History, Analysis,Scores and Recommendations, White Paper, ECL/LLC*, 2002.

[59] X. Inc. Xilinx Power Estimator (XPE). [Online]. Available: https://www.xilinx.com/products/technology/power/xpe.html

[60] S. Gal-On and M. Levy, "Exploring coremark–a benchmark maximizing simplicity and efficacy."

Appendices

Flow diagrams A

Figure A.1 shows the general flow diagram as discussed in Section 3.3.2. Figure A.2 projects the flow diagram in case of a read. Following Figure A.3 depicts the flow diagram for writing from the core (processor). Lastly Figure A.4 is the flow diagram for writing from the Lower Level Memory (LLM).

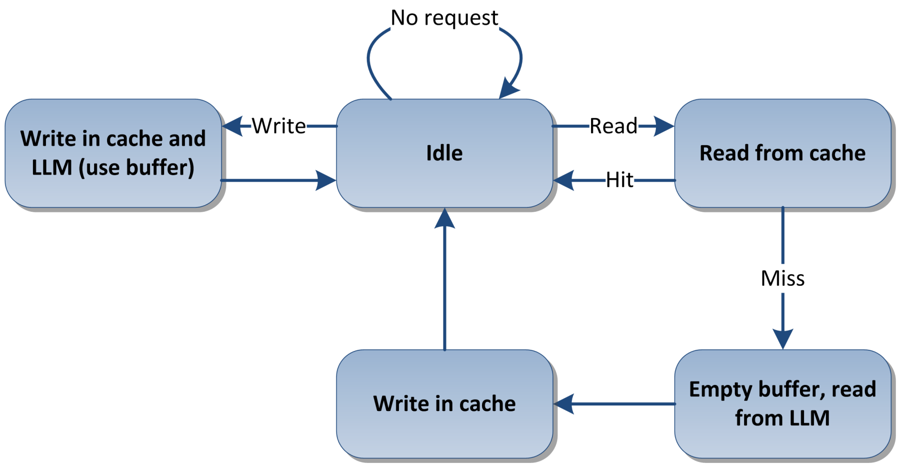

No request
Write in cache and LLM (use buffer)
Write
Idle
Read
Hit
Read from cache
Miss
Write in cache
Empty buffer, read from LLM

Figure A.1: General flow diagram of the working of the WT cache (high level, from the viewpoint of the core).

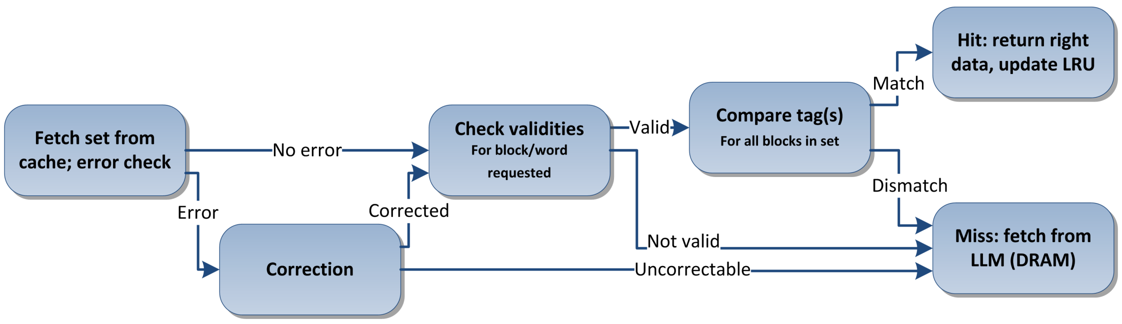

Fetch set from cache; error check
No error
Error
Correction
Corrected
Check validities For block/word requested
Valid
Not valid
Uncorrectable
Compare tag(s) For all blocks in set
Match
Dismatch
Hit: return right data, update LRU
Miss: fetch from LLM (DRAM)

Figure A.2: Flow diagram for performing a read.

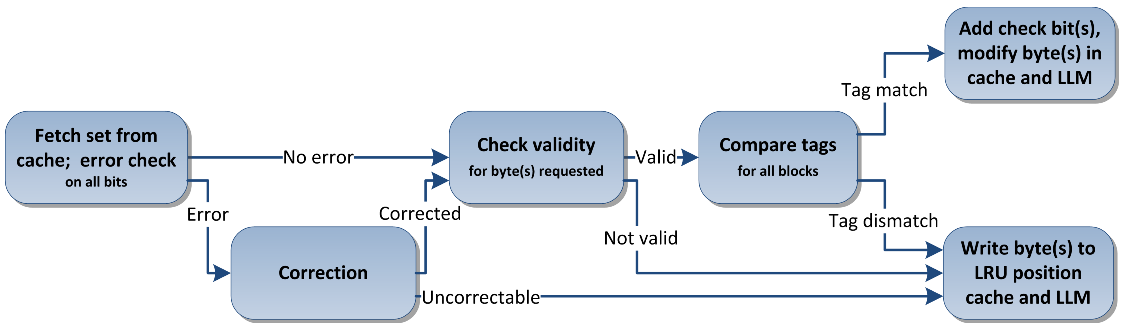

Fetch set from cache; error check on all bits
No error
Error
Correction
Corrected
Uncorrectable
Check validity for byte(s) requested
Valid
Not valid
Compare tags for all blocks
Tag match
Tag dismatch
Add check bit(s), modify byte(s) in cache and LLM
Write byte(s) to LRU position cache and LLM

Figure A.3: Flow diagram for modifying (writing) a value, incurred by the processor.

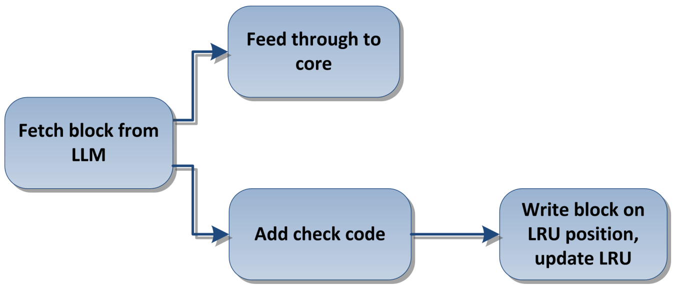


Figure A.4: Flow diagram for loading in data from the LLM.

B Block diagram

Figure B.1 shows the block diagram as discussed in Section 3.3.3.

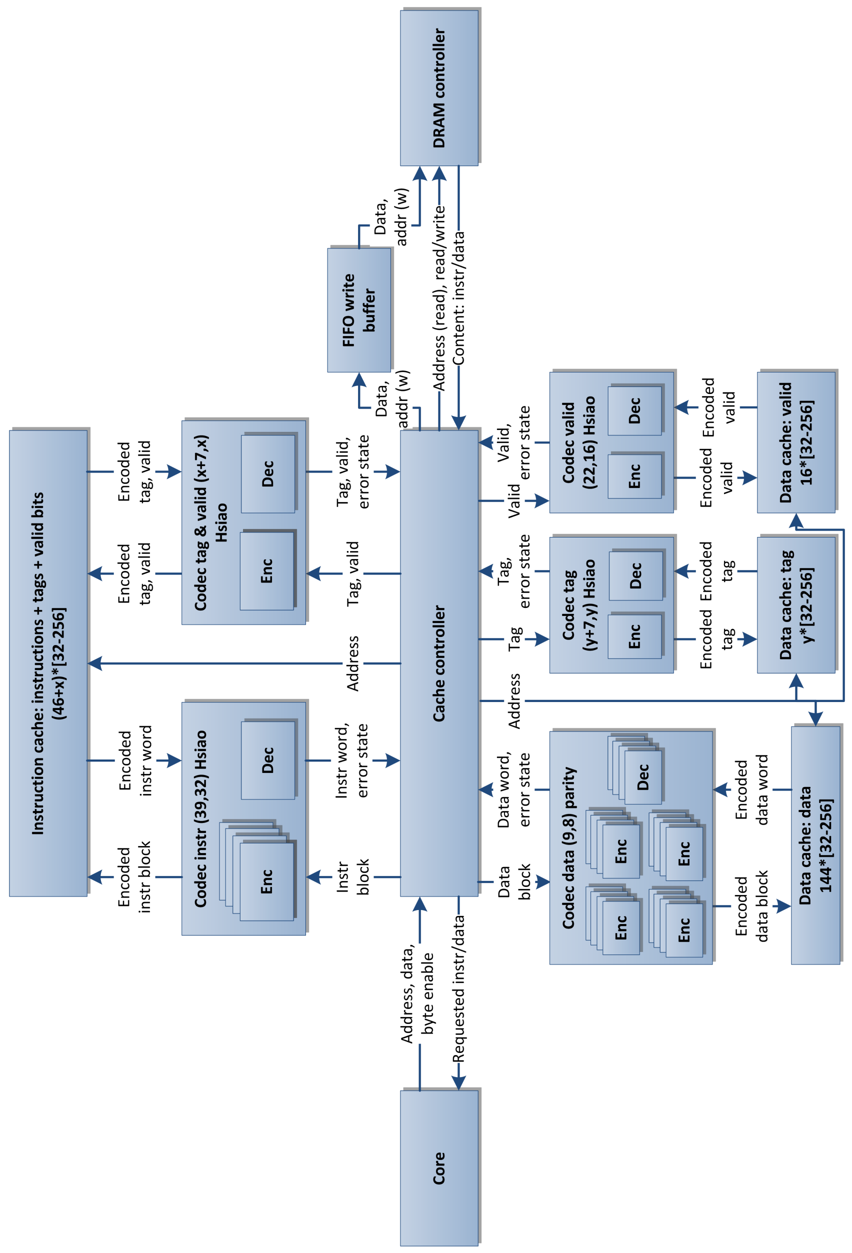


Figure B.1: Block diagram for the complete cache structure, including all signals.

Parity Check Matrices (PCMs) C

The PCMs of EDAC codes define them: these are needed to implement the codecs. In this appendix the PCMs are listed as used to implement all needed codecs. Below and to the right of these tables are the number of ones which indicates the hardware cost and also whether the code is balanced. The column marked with ”s” lists the number of ones including the check bits while the column marked with ”c” does not include the check bits. Remaining numbers to the left and above the matrices are the indexes.

The listed PCMs are:

1. Hsiao (13,8): Table C.1
2. Hsiao (24,18): Table C.2
3. Hsiao (25,19): Table C.3
4. Hsiao (26,20): Table C.4
5. Hsiao (27,21): Table C.5
6. Hsiao (28,22): Table C.6
7. Hsiao (29,23): Table C.7
8. Hsiao (30,24): Table C.8
9. Hsiao (31,25): Table C.9
10. Hsiao (32,26): Table C.10
11. Hsiao (34,27): Table C.11
12. Hsiao (35,28): Table C.12
13. Hsiao (36,29): Table C.13
14. Hsiao (39,32): Table C.14
15. Hsiao (41,34): Table C.15

Table C.1: PCM of Hsiao (13,8) code (original [12]).

	7	6	5	4	3	2	1	0	4	3	2	1	0	s	c
4	1			1	1		1	1	1					6	5
3	1	1				1		1		1				5	4
2	1	1	1	1			1				1			6	5
1		1	1	1	1	1						1		6	5
0			1		1	1	1	1					1	6	5
	3	3	3	3	3	3	3	3			Check			29	24

Table C.2: PCM of Hsiao (24,18) code (modified from (25,19)).

								1																		
	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1										1						10	9
4	1	1	1							1	1	1	1	1	1					1					10	9
3				1	1	1				1	1	1				1	1	1			1				10	9
2	1			1			1	1		1			1	1		1	1					1			10	9
1		1			1		1		1		1		1		1	1		1					1		10	9
0			1			1		1	1			1		1	1		1	1						1	10	9
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3			Check				60	54

Table C.3: PCM of Hsiao (25,19) code (modified from (26,20)).

									1																		
	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1											1						10	9
4	1	1	1							1	1	1	1	1	1						1					10	9
3				1	1	1				1	1	1				1	1	1				1				10	9
2	1			1			1	1		1			1	1		1	1		1				1			11	10
1		1			1		1		1		1		1		1	1		1	1					1		11	10
0			1			1		1	1			1		1	1		1	1	1						1	11	10
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3			Check				63	57

Table C.4: PCM of Hsiao (26,20) code (modified from (27,21)).

										1																		
	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1	1											1						11	10
4	1	1	1	1							1	1	1	1	1	1						1					11	10
3	1				1	1	1				1	1	1				1	1	1				1				11	10
2		1			1			1	1		1			1	1		1	1		1				1			11	10
1			1			1		1		1		1		1		1	1		1	1					1		11	10
0				1			1		1	1			1		1	1		1	1	1						1	11	10
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3			Check				66	60

Table C.5: PCM of Hsiao (27,21) code (modified from (28,22)).

	2										1																		
	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1	1											1	1						12	11
4	1	1	1	1							1	1	1	1	1	1					1		1					12	11
3	1				1	1	1				1	1	1				1	1	1		1			1				12	11
2		1			1			1	1		1			1	1		1	1		1	1				1			12	11
1			1			1		1		1		1		1		1	1		1	1	1					1		12	11
0				1			1		1	1			1		1	1		1	1	1							1	11	10
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	5	Check						71	65

Table C.6: PCM of Hsiao (28,22) code (modified from (29,23)).

		2										1																		
	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1	1											1	1	1						13	12
4	1	1	1	1							1	1	1	1	1	1					1	1		1					13	12
3	1				1	1	1				1	1	1				1	1	1		1	1			1				13	12
2		1			1			1	1		1			1	1		1	1		1	1	1				1			13	12
1			1			1		1		1		1		1		1	1		1	1	1						1		12	11
0				1			1		1	1			1		1	1		1	1	1		1						1	12	11
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	5	5	Check						76	70

Table C.7: PCM of Hsiao (29,23) code (modified from (30,24)).

			2										1																		
	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1	1											1	1	1	1						14	13
4	1	1	1	1							1	1	1	1	1	1					1	1	1		1					14	13
3	1				1	1	1				1	1	1				1	1	1		1	1	1			1				14	13
2		1			1			1	1		1			1	1		1	1		1	1	1					1			13	12
1			1			1		1		1		1		1		1	1		1	1	1		1					1		13	12
0				1			1		1	1			1		1	1		1	1	1		1	1						1	13	12
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	5	5	5	Check						81	75

Table C.8: PCM of Hsiao (30,24) code (modified from (31,25)).

				2										1																		
	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1	1											1	1	1	1	1						15	14
4	1	1	1	1							1	1	1	1	1	1					1	1	1	1		1					15	14
3	1				1	1	1				1	1	1				1	1	1		1	1	1				1				14	13
2		1			1			1	1		1			1	1		1	1		1	1	1		1				1			14	13
1			1			1		1		1		1		1		1	1		1	1	1		1	1					1		14	13
0				1			1		1	1			1		1	1		1	1	1		1	1	1						1	14	13
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	5	5	5	5	Check						86	80

Table C.9: PCM of Hsiao (31,25) code (modified from (32,26)).

					2										1																		
	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1	1											1	1	1	1	1	1						16	15
4	1	1	1	1							1	1	1	1	1	1					1	1	1	1			1					15	14
3	1				1	1	1				1	1	1				1	1	1		1	1	1		1			1				15	14
2		1			1			1	1		1			1	1		1	1		1	1	1		1	1				1			15	14
1			1			1		1		1		1		1		1	1		1	1	1		1	1	1					1		15	14
0				1			1		1	1			1		1	1		1	1	1		1	1	1	1						1	15	14
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	5	5	5	5	5	Check						91	85

Table C.10: PCM of Hsiao (32,26) code (custom made with compliance to [12]).

						2										1																		
	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1		5	4	3	2	1	0	s	c
5	1	1	1	1	1	1	1	1	1	1											1	1	1	1	1		1						16	15
4	1	1	1	1							1	1	1	1	1	1					1	1	1	1		1		1					16	15
3	1				1	1	1				1	1	1				1	1	1		1	1	1		1	1			1				16	15
2		1			1			1	1		1			1	1		1	1		1	1	1		1	1	1				1			16	15
1			1			1		1		1		1		1		1	1		1	1	1		1	1	1	1					1		16	15
0				1			1		1	1			1		1	1		1	1	1		1	1	1	1	1						1	16	15
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	5	5	5	5	5	5	Check						96	90

Table C.11: PCM of Hsiao (34,27) code (modified from (39,32) [12]).

							2										1																		
	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0	s	c
6	1	1	1	1	1	1						1					1			1		1					1	1							12	11
5					1	1	1	1	1	1	1	1	1			1			1			1							1						13	12
4				1					1					1	1	1	1	1	1	1	1			1	1					1					13	12
3			1					1			1		1	1								1	1	1	1	1	1				1				12	11
2		1	1			1				1			1					1	1	1	1		1	1		1						1			13	12
1	1						1			1	1	1		1	1	1	1	1								1							1		12	11
0	1	1		1	1		1	1	1						1						1		1		1		1							1	13	12
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	Check							88	81

Table C.12: PCM of Hsiao (35,28) code (modified from (39,32) [12]).

								2										1																		
	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0	s	c
6	1	1	1	1	1	1	1						1					1			1		1					1	1							13	12
5					1		1	1	1	1	1	1	1	1			1			1			1							1						13	12
4				1						1					1	1	1	1	1	1	1	1			1	1					1					13	12
3			1			1			1			1		1	1								1	1	1	1	1	1				1				13	12
2		1	1				1				1			1					1	1	1	1		1	1		1						1			13	12
1	1					1		1			1	1	1		1	1	1	1	1								1							1		13	12
0	1	1		1	1			1	1	1						1						1		1		1		1							1	13	12
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	Check							91	84

Table C.13: PCM of Hsiao (36,29) code (modified from (39,32) [12]).

2										1																											
	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0	s	c
6	1	1	1	1	1	1	1	1						1					1			1		1					1	1							14	13
5					1			1	1	1	1	1	1	1	1			1			1			1							1						13	12
4				1							1					1	1	1	1	1	1	1	1			1	1					1					13	12
3			1				1			1			1		1	1								1	1	1	1	1	1				1				13	12
2		1	1			1		1				1			1					1	1	1	1		1	1		1						1			14	13
1	1					1	1		1			1	1	1		1	1	1	1	1								1							1		14	13
0	1	1		1	1				1	1	1						1						1		1		1		1							1	13	12
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	Check							94	87

Table C.14: PCM of Hsiao (39,32) code (original [12]).

		3										2										1																			
	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0	s	c
6	1	1	1	1	1	1	1	1							1					1			1		1						1	1	1							15	14
5					1			1	1	1	1	1	1	1	1	1			1			1			1					1				1						15	14
4				1								1					1	1	1	1	1	1	1	1			1	1		1	1				1					15	14
3			1				1				1			1		1	1								1	1	1	1	1	1	1	1				1				15	14
2		1	1			1		1		1			1			1					1	1	1	1		1	1		1								1			15	14
1	1					1	1		1				1	1	1		1	1	1	1	1								1									1		14	13
0	1	1		1	1				1	1	1	1						1						1		1		1				1							1	14	13
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	Check							103	96

Table C.15: PCM of Hsiao (41,34) code (modified from (39,32) [12]).

		3										2										1																					
	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0	s	c
6	1	1	1	1	1	1	1	1							1					1			1		1						1	1		1	1							16	15
5					1			1	1	1	1	1	1	1	1	1			1			1			1					1				1		1						16	15
4				1								1					1	1	1	1	1	1	1	1			1	1		1	1			1			1					16	15
3			1				1				1			1		1	1								1	1	1	1	1	1	1	1	1					1				16	15
2		1	1			1		1		1			1			1					1	1	1	1		1	1		1										1			15	14
1	1					1	1		1				1	1	1		1	1	1	1	1								1				1							1		15	14
0	1	1		1	1				1	1	1	1						1						1		1		1				1	1								1	15	14
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	Check							109	102

Xilinx ISE settings D

In Table D.1 the parameters are shown used for generating the bit files for the FPGA. These settings are taken from the most current status of th RISC-V project at the employer.

Table D.1: Settings for running Xilinx ISE applied for all generated results.

Process	Setting	Value
Synthesis	Bus Delimiter	()
	Keep Hierarchy	Soft
	Optimization Goal	Speed
	Optimization Effort	High
	Register Balancing	Yes
Map	Global Optimization	Speed
	Generate Detailed MAP Report	True
Place & Route	Place & Route Effort Level (Overall)	High
	Enabl Multi-Threading	4
Generate Programming File	Set SPI Configuration Bus Width	4