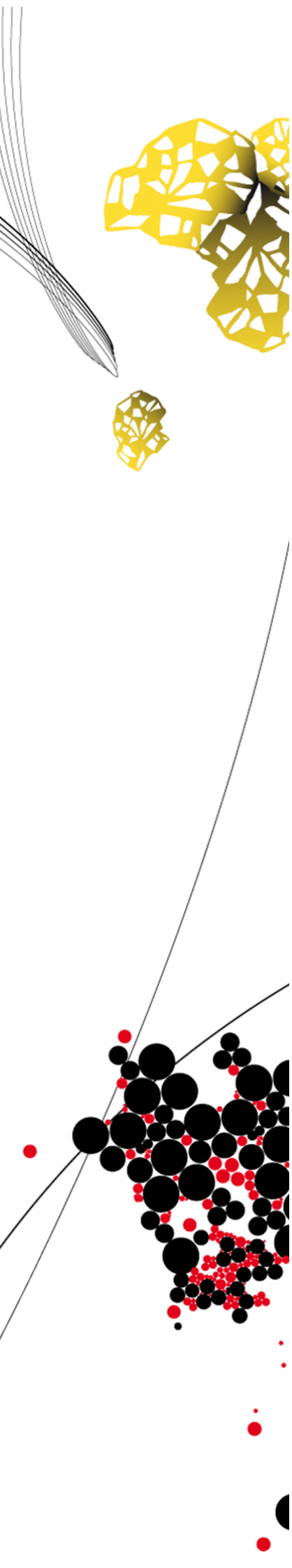

UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Designing a RERI-based error logging system for RISC-V cores

Michiel Koenderink

M.Sc. Thesis - Embedded Systems
March 2025

Supervisors:
dr. ir. M. Ottavi
dr. ir. G. K. Rauwerda
dr. ir. A. Chiumento
B. Endres Forlin

Computer Architecture for Embedded Systems (CAES)
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

In recent years, the adoption of RISC-V cores in advanced systems has grown significantly. These cores are employed in several areas, including environments with a higher risk of hardware errors, such as space. Critical systems must be able to detect and resolve as many errors as possible to maintain reliable operation. Error detection, logging, analysis and resolution keep systems operational while collecting important diagnostic information. The RISC-V organisation proposed a specification for formatting error information, known as RERI. However, extensive and large nature of this format can be impractical where time and resources are scarce. Furthermore, no dedicated framework around this error logging format has been specified or introduced yet. This work builds on the initial RISC-V RERI specification by implementing an adapted version called ”RERI-Lite”. Developed primarily for research use in radiation beam experiments, this system addresses the needs of smaller-scale applications with high error rates. This thesis will focus on the implementation of a RERI-Lite based system and it will compare RERI-Lite to the standard RERI format. It demonstrates how the lighter, more flexible design of RERI-Lite can improve performance in resource-constrained contexts. Finally, the philosophy behind the error logging framework is examined, illustrating how it fits into broader system reliability goals.

Contents

List of acronyms

CE	Corrected Error
DUE	Detected Uncorrectable Error
ECC	Error Correction Code
EDAC	Error Detection And Correction
ELS	Error Logging System
FPGA	Field-Programmable Gate Array
IC	Integrated Circuit
IO	Input/Output
LUT	Lookup Table
RAS	Reliability, Availability and Serviceability
RERI	RAS Error Record Register Interface
SDC	Silent Data Corruption
SECDED	Single Error Correction Double Error Detection
SEE	Sigle Event Effect
SET	Single Event Transient
SEU	Single Event Upset
SoC	System-on-Chip
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver/Transmitter
WDT	Watch Dog Timer

List of terms

Error Logging System The entire framework of a system that handles: monitoring error detection units, storing error information, analysing error information and reporting error information.

RERI-Lite (format) The adjusted format of an error record for the storage of error information that is proposed in this thesis. This format is based on the format described in the RISC-V RERI Architecture specification [1].

standard RERI (format) The format of an error record for the storage of error information. This format is described in the RISC-V RERI Architecture specification [1].

Chapter 1

Introduction

The attention for the open-source RISC-V ISA is increasing in the industry [2]. RISC-V is a very promising candidate for various application domains, such as industrial automation and healthcare. Another interesting application domain is space. The European Space Agency (ESA) has selected the RISC-V for use in space [3]. This growing adoption highlights the critical need to address hardware errors that jeopardize Reliability, Availability and Serviceability (RAS). Hardware errors can cause serious problems in a RISC-V System-on-Chip (SoC). In radiation-intense environments, such as those found in space, systems become more susceptible to Single-Event Upsets (SEUs) and other radiation-induced faults, which can disrupt normal operations or even cause mission failure [4]. While various mitigation strategies (e.g., error detection and redundancy) exist, it is impossible to fully prevent hardware errors from occurring.

To enhance RAS a dedicated subsystem can monitor, store, analyse and report hardware errors. Such an Error Logging System (ELS) enables targeted error handling and corrective action. This capability is also beneficial in radiation testing environments, where multiple errors may accumulate and affect a device's functionality [5]. An Error Logging System (ELS) could also be used to improve standardized error reporting in radiation testing by providing more information about errors, which could be useful for a more extensive analysis.

To prevent the system behaviour from changing or breaking down, the errors caused by the radiation should be handled. This is traditionally done by either overwriting single frames of programming data at a time or by otherwise stopping all processing and completely overwriting the programming data. Creating a new subsystem to document the errors during the tests could prevent the need to stop all processes. However, it is important that an error logging system is also able to protect itself against errors, since an unreliable error logging system could cause even

more damage to the data and behaviour of the main system. So, the new error logging subsystem should not be more susceptible to hardware errors than the main system itself. Finally, to be able to use it on different types of SoCs, a general technique should be created to execute the necessary actions to handle uncorrected errors.

The RISC-V foundation worked on a specification for the logging format of hardware error information, called RAS Error Record Register Interface (RERI) [1]. However, as will be argued in this thesis, this format is not ideal for every system. The proposed format requires a lot of memory and creates overhead. That is why systems that have limited resources or require quick decisions based on the detected errors, might need an alternative. Although the specification is comprehensive and flexible enough to also be included in embedded systems, many fields of the standard format will have to be left partially unused or unimplemented. This could lead to many different implementations of RERI records in embedded systems, making their compatibility with other applications difficult.

For example, if a system wants to integrate components from different manufacturers that made changes to the RERI format, it will have to know what information is (no longer) available. This requires custom compatibility patches to integrate components. Using multiple RERI variants will make the integration more complex and it is likely to lower the efficiency of the system. The lack of a general standard will especially decrease the performance of systems such as the ELS, which require compatibility with as many components as possible. That is why a single simplified version, based on the standard RERI format, that is dedicated to embedded systems would be a good addition to the original specification.

So, the main contribution of this research will be to create a new version or updated version of the RERI format that is dedicated to embedded systems and to design a general subsystem for RISC-V SoCs to log and resolve hardware errors that are detected. This new error record format will be referred to as the RERI-Lite (format), while the original error record format specified by the RISC-V foundation will referred to as the standard RERI (format).

1.1 Research goal

As mentioned in the introduction, there is a need for a subsystem in RISC-V SoCs that is able to log and analyse hardware and/software errors in a system. The main

goal of this thesis is as follows:

"Design and implement a system that can systematically detect, log, analyse and resolve hardware errors in a RISC-V SoC."

This main goal has been divided into multiple smaller research questions and goals. To design a complete system, the following research questions and objectives will need to be completed:

1. *What types of hardware errors exist and how can they be detected?*
2. *What type of information about the errors is generally available and is it usable for error analysis?*
3. *How can the error information be stored and made accessible for later use in an efficient way?*
4. *How can the error information be analysed in a general way?*
5. *Can errors be handled in a general way?*

By answering these research questions, a final design for an error logging system can be created. The final goal of this thesis will be to implement and test that error logging system design.

1.2 Report organization

In the following chapters, the research goals will be systematically tried to be answered.

- Chapter 2 gives information about different types of errors and the RISC-V computer architecture to provide some necessary base knowledge for the thesis.
- Chapter 3 presents some research works that are related to the subject of this thesis.
- Chapter 4 explains all the theoretical considerations that were made and how a final design for the error logging system was formed.

- Chapter 5 explains how the design from Chapter 4 was implemented and tested in simulations and experiments.
- Chapter 6 analyses the performance of the RERI-Lite-based system and compares it to the standard RERI format.
- Chapter 7 discusses the validity and completeness of the work.
- Chapter 8 answers the main thesis goal and the research questions. It will also recommend any necessary or optional future work on this subject.

Chapter 2

Background

This chapter will provide the necessary knowledge to understand all the topics that will be covered in this thesis. The focus will be on answering the first two research questions that were stated in Section 1.1. First, it will determine the hardware errors that exist and how these errors can be detected and corrected. Next, it will discuss the information or data that will usually be available about these errors as well. This chapter will also give some important information about the RISC-V instruction set architecture and how it handles error information.

2.1 Radiation induced hardware errors

Radiation is one of the biggest causes of failures in electronic systems. Energetic charged particles, such as protons and heavy ions can bombard electronic systems and cause errors. The particles can cause all sorts of harmful effects in an Integrated Circuit (IC), which are called Sigle Event Effects (SEEs) [6]. These SEEs can lead to the loss of information or failures in systems. The SEEs can change the state of a latch or memory cell or cause temporary voltage spikes at internal circuit nodes called Single Event Transients (SETs). There exist destructive and non-destructive SEEs [4]. Destructive SEEs are also called hard errors and these errors change component states that cannot be changed back or they damage the physical hardware of the ICs. There are also non-destructive SEEs, which are called soft errors. These changes caused by these errors can be reverted and systems are able to recover from these errors.

2.1.1 Error type overview

The standard taxonomy for error detection has three different categories [7]. The first category is Silent Data Corruption (SDC), which contains the errors that are not detected. The second category, called Detected Uncorrectable Error (DUE) has errors that are detected, but these errors can not be corrected. The third category is Corrected Error (CE), which are errors that are detected and they can be corrected as well.

If a hardware error occurs in a system, it will depend on the available detection and correction methods in the system that will determine in which category the error will end up. Figure 2.1 shows a possible way to classify errors in a system [7].

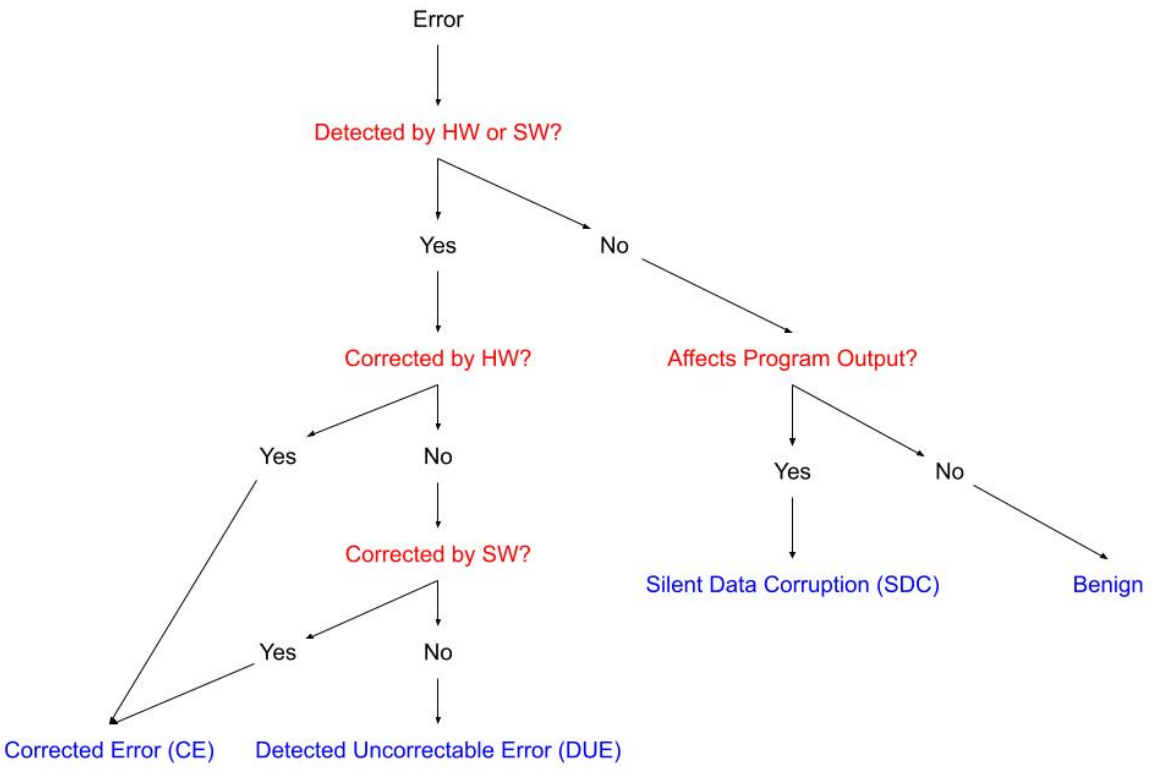

Figure 2.1: A simplified taxonomy of error outcomes based on the error taxonomy from [7].

Hardware errors can occur in many different locations, which will influence the severity of the error. A simple bit flip in a data memory could overwrite a completely unused or irrelevant data value, while that same bit flip in the instruction memory could cause the system to overwrite important data, repeat or skip instructions or even fully break down. An overview of important components and locations within

the computer architecture of RISC-V based systems was made and it is shown in Table 2.1. This overview identifies certain possible errors that could occur in these components and how they will influence the behaviour of the system.

processor part	component	Detection methods	Main result/problem	Possible causes
-	wires/flipflops/(DE)MUX/LUT	redundancy	Wrong logic output	SEU/SET/timing errors
CPU core	whole core		Wrong calculations/data results	-
			Execution of incorrect instructions	-
			Core stalling/repeating	-
			Excess to/overwriting wrong memory parts/data	-
			Wrong/malicious communication to external connections	-
	control unit	redundancy	wrong output signals for CPU and BUS	instruction decode
				wrong flags
				clock glitches
	program counter	redundancy	skipping/repeating parts of the instruction memory	wrong selector
				incorrect counter increas
				incorrect branch calculation
	ALU	Redundancy	Wrong data/output	wire changes
				corrupted logic gates
	FPU	redundancy	Wrong data/output	wire changes
				corrupted logic gates
	CFU	redundancy	Wrong data/output	wire changes
				corrupted logic gates
	Instruction cache/register	redundancy & ECC	Altered instructions	Bit flips in the instructions
			Failing instructions	Bit flips in the instructions
	Registers	redundancy & ECC	Wrong data used for instructions	Read/write switched
				Read/write old data
				Bit flips
				Incorrect register selection
	Data cache	redundancy & ECC	wrong data output	Incorrect address selection
				Read/write switched
				Read/write old data
				Bit flips
	sign exted	redundancy	wrong data output	Bit/signal flip
	CLINT/PLIC	redundancy	Core stalling	wrong interrupt code/signal
			unnecessary fixes	false interrupt triggers
	MMU	redundancy	using wrong data	
	Load/store unit	redundancy	Wrong data/output	wrong register/data selection
	PMP		illegal access	permission setting changed
			locked out of acessable regions	permission setting changed
Bus	address bus	redundancy	data stored in wrong location	wrong address read due to bit/signal change
				wrong address write due to bit/signal change
	data bus	redundancy	wrong data stored	SET/SEU in data
	control bus	redundancy	data not written	switch in read/write mode
			data written instead of read	switch in read/write mode
External connections	JTAG/UART	redundancy	Unable to communicate or receive/transmit information	Protocol errors
				Signal interruptions/disconnections
			receive/transmit incorrect information	SET/SEU in data

Table 2.1: A table with an overview of possible error locations in the system architecture and the possible problems and causes that these errors can have.

As can be seen in the table, the most important part of the system is the core. There are a lot of components here that are very important. The core has a lot of different components and it can often control the entire system and its processes. It usually has the ability to influence system components outside the core as well. This is why errors in the core can do significantly more damage to the data or the system itself than in other system units. For example, an error that changes an (executed) instruction can change the settings of the system, access illegal data or instructions and change the entire system process by reading wrong instructions. Executing such corrupted instructions can change the entire process in the system or even break it. Depending on the application of the system, it could lead to catastrophic results.

Another problem with errors in the core is that it could cause irreparable errors. If an error occurs, it can sometimes be repaired by the core while the system is still running. For example, by reloading corrupted data. However, if the core itself has completely broken down, it might prevent the core from executing the necessary instructions to fix itself. In such a case, the only option might be to completely reset the system. That is why it is very important to prevent these errors from happening and to correct any errors as quickly as possible if they occur.

2.2 Detecting and correcting errors in hardware

For systems to be able to recover from soft errors, Error Detection And Correction (EDAC) is used. There are many different methods and techniques to detect and correct errors. However, these techniques focus on certain types of errors and these techniques all have their own advantages and disadvantages. There is not a single best way to detect and correct every possible error type that exists. Systems require a combination of methods to proof their system against all possible error types. So a general system to log all errors would need to be compatible with as many of these detection units as possible.

Some detection methods can immediately correct the detected errors. One important method is Error Correction Code (ECC). ECC can detect bit changes by adding extra bits to the data. There are several ECC algorithms, such as parity check and Hamming code [4], [8], [9]. Algorithms like the parity check can only detect a change in the original data, while Hamming code can be used to detect and correct multiple bit changes. However, the ability to detect and correct multiple bit changes requires additional bits for the check, which results in larger memory usage.

Hamming codes can be extended with additional parity bits and that is often known as Single Error Correction Double Error Detection (SECDED) [10]. SECDED is an ECC that is frequently used for embedded memory applications, since it is easy to implement and it has little influence on the latency and space of the memory system.

It is also possible to try and prevent double errors in memory elements by adding scrubbing to the system. Scrubbing works by periodically reading the memory and repairing any single bit errors that are present. While this is a good way to prevent unusable data, it will cause a lot of overhead and in cases where the data is relatively quickly used, it will be less useful to add scrubbing, since its effectiveness will be

lower. A method has been proposed to determine what the mean time until a double error is, depending on the system variables and environment [11]. That calculation can be used to determine if it is necessary to add scrubbing to the system.

ECC and SECDED add redundancy to the data, but another important method to detect errors is to add redundant hardware to a system [12], [13]. By using multiple instances of the same hardware, functional units can be duplicated and system operations can be done multiple times. If the results deviate, it means that an error occurred, so this is how the hardware of functional units can be checked for errors.

There are several approaches to adding hardware redundancy as well. Redundancy can be added from transistor level to system level and all these different approaches have their own trade-offs. The most popular method is Triple Modular Redundancy (TMR), which uses three instances of a functional unit. It allows the hardware to compare the 3 results and immediately choose the most frequent result as the correct outcome. This method does not require a new operation to find the correct result and that will speed up the process.

Unfortunately, some hardware errors are undetectable. Most errors can be prevented by adding more redundancy and checks, but even these backup systems can theoretically fail. A balance needs to be found where the measures against hardware errors do not cause more problems than the original errors can cause.

2.3 Detecting and correcting errors in software

The error logging system is mainly focussed on hardware errors, however, software errors can also occur in its systems. Software errors are often handled by the operating system, however they can be useful for the error logging system as well. While it is not necessarily the focus of the error logging system to deal with these type of errors, some applications might want to use the error logging system to log and export these types of errors as well.

Another advantage of monitoring software errors inside the error logging system is that the occurrence of large amounts of software errors or very specific errors can indicate an hardware error as well. If specific tasks like an address read or write keep failing, it might indicate that there is an hardware error that is causing it. For example, a hardware error in the bus or address calculation might cause the software to access illegal addresses or invalid data. If these errors were not found by

hardware detection methods, the software errors will still give the error logging system an indication that an error is present. By analysing the software error data, the error logging system could try to narrow down the location of hardware errors, but this analysis would likely be very complex.

This complexity might actually make the performance of the error logging system worse. There are many types of software errors, with different information. Also access to the error data will depend on the specific system application or manufacturer. Using the available data in a meaningful way will be very difficult. That is why monitoring and usage software errors will be ignored for now. It will broaden the scope of this thesis too much, but this feature could be added in future versions of the system.

So, the error logging system will not focus on software errors. However, it would be possible to add specific hardware units that can only monitor for software errors and log the error data in the error logging system. The RERI and RERI-Lite error records that will be explained in Section 4.4 could be used for that.

2.4 Error information

Various systems and error detection methods were looked at to answer research question two, ”What type of information about the errors is generally available and is it usable for error analysis?”. Examples are the architecture of RISC-V systems such as the NEORV32 [14] and OpenTitan [15] or the available manuals with the architecture of various Field-Programmable Gate Arrays (FPGAs) from companies like AMD [16] and Intel [17]. The focus lies on their implementation of build-in error detection mechanisms, such as their available ECC signals.

It is hard to find very specific data that is always available. It depends largely on the type of error and on the specific implementation of a company. For example, the provided build-in ECC signals of AMD [16] and Intel [17] FPGAs would be different, but the information would be very similar. The AMD implementation had two signals: one for a single bit corrected error and one for a double bit detected error. However, the Intel implementation would give a double bit flag, that signals four different states, of which one state was for a single bit corrected error and one for a double bit detected error. The information is mostly the same, but the way that the information is presented differs. This problem makes it very difficult to make an error logging system compatible with all types of error detection systems.

However, the information that usually seems to be signalled by most error detection implementations was that an error had occurred and if that error had already been corrected or not.

Besides, not all information necessarily needs to be provided by the error unit itself. Information and data are also provided that are not always directly provided by the error detection units, but can relatively easily be gathered from the system itself. Examples for this are the error address, which would be available on the bus or some sort of timestamp that the system (or the ELS itself) is using.

Also, some additional information can be hardcoded when the error detection units are connected to the error logging system. When the detection units are added to the system, it is possible to provide some extra information for the error logging system. Examples are a unique ID for the error detection unit or specific error codes for the types of errors that this unit can detect.

In addition to this data, there was not much overlap in the data that could be provided by standard error detection techniques. Additional information would usually depend on the specific implementations of error detection methods, so basing the error logging format on it will likely result in the data fields remaining unused in most applications. For other error information, the use of customizable information fields would likely be more efficient.

2.5 RISC-V RERI

As mentioned in Section 1, the RISC-V foundation created its own specification for the storage of information about errors in a systematic way [1].

The RISC-V RERI Architecture Specification describes a RERI bank that can log information about errors. One RERI bank can store up to 63 different error records. The first 64 bytes are used to store general information about the error bank, such as the bank ID, the amount of records in the bank and an overview of the valid records inside the bank. After that, every 64 bytes will form an error record. Each error record is made up of 8 registers of 64 bits. Table 2.2 shows an overview of the RERI bank layout.

The first register of an error record is the ”control” register, which is used to control the reporting of that specific error record. The ”status” register will provide

general information about the error, such as the error code, error type, its priority and if the record is valid. The ”addr_info” register will give information about the location of the error. The ”info” and ”suppl_info” registers are customizable and can be used to store all sorts of available information about the error. The ”timestamp” register can be used to store any sort of time reference used in the system from the moment the error was written to the bank. The last two registers are reserved for future standard use by the specification. Figures 2.2 and 2.3 show the format of the control and status registers from the RERI specification [1] as an example of the individual fields in these register formats.

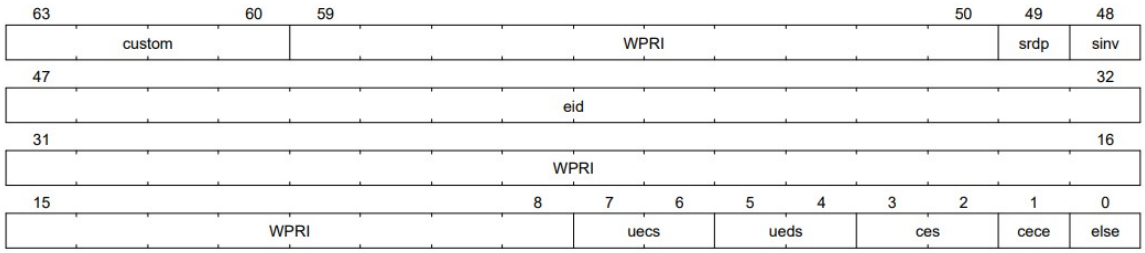

Figure 2.2: The format of the RERI control register from [1].

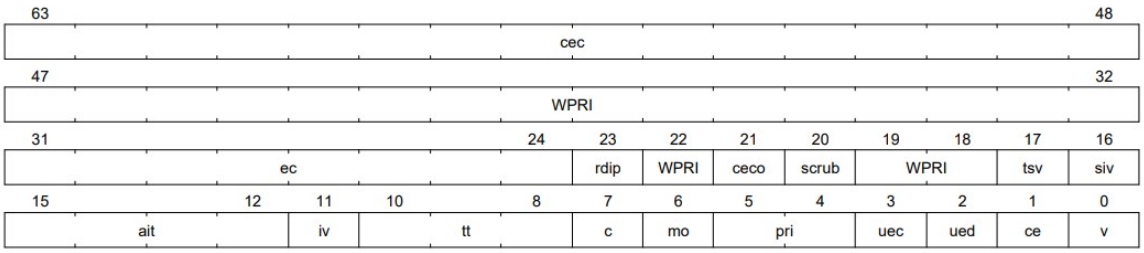

Figure 2.3: The format of the RERI status register from [1].

The records are connected to a dedicated system component that can log errors to it. If a component detects a new error and wants to write it to the RERI bank, it should use a procedure to determine if a previous error in that record should be overwritten or not. If a component is expected to detect a lot of (different) errors, it is likely that error information will be overwritten and lost.

Offset	Name	Size	Description
0	vendor_n_imp_id	8	Vendor and implementation ID.
8	bank_info	8	Error bank information.
16	valid_summary	8	Summary of valid error records.
24	Reserved	32	Reserved for future standard use.
56	Custom	8	Designated for custom use.
64 + 64 * i	control_i	8	Control register of error record i.
72 + 64 * i	status_i	8	Status register of error record i.
80 + 64 * i	addr_info_i	8	Address-or-info. register of error record i.
88 + 64 * i	info_i	8	Information register of error record i.
96 + 64 * i	suppl_info_i	8	Supplemental information register of error record i.
104 + 64 * i	timestamp_i	8	Timestamp register of error record i.
112 + 64 * i	Reserved	16	Reserved for future standard use.

Table 2.2: An overview of the RERI error bank format from table 2 of the RISC-V specification [1]. The size of the registers is given in bytes.

Chapter 3

Related Work

This chapter will give several research works and projects that are related to the work of this thesis. These works give new and unique methods and techniques to detect hardware errors in systems and monitor the state of a system. First some research will be given that is specifically for RISC-V systems and after that, some other unrelated works will be provided that give some other interesting angles and aspects on the error logging topic.

3.1 Error reporting in RISC-V specific systems

Error logging in RISC-V systems has not been explored much yet. RISC-V recently formed a task group to create the RISC-V RERI Architecture Specification [1]. This describes a register bank format to store information about hardware errors. However, this specification does not describe the framework around the register bank. The specification leaves a lot up to the implementation and its design is rather large for most applications, but it is an important format to consider while designing a new error logging system, since future RISC-V works are likely to follow this RERI standard.

The paper [18] proposes a more complete error logging system for RISC-V. It describes the architecture of a hardware and software interface called ”ENGAGE” to collect and store the information of errors. However, this system was not designed with the RERI specification in mind, since no specification proposal was published by the RISC-V RERI taskgroup at the time.

FIRECAP is also interesting, because it is an IP embedded in a SoC that allows the probing and recording of processor resources [19]. The approach in this work

uses watchpoints, triggers and circular buffer recording to identify the failure reason of processor-based SoCs.

Another technique to addresses microprocessor errors due to radiation induced Single Event Upset (SEU) faults in SoCs is also proposed [20]. This work uses a machine-learning algorithm to analyse the traces of the errors. All these works are focused on RISC-V based SoCs, which makes these works extra useful.

3.2 Other error detection and reporting mechanisms

However, there is also a lot of relevant work that is not specific to the RISC-V architecture. Examples are the design and implementation of error correcting and reporting mechanisms in systems with different architectures, such as the products from Intel [17], AMD [16] and ARM [21]. These systems already exist and form a proven method to detect and log radiation-induced hardware errors in systems.

Another completely different methodology to diagnose radiation-induced faults in microprocessors is proposed in the paper [22]. It uses the hardware trace infrastructure to reconstruct the error that occurred. The follow-up paper [23] describes how this method was tested under a laser and what the experimental results were.

There are also methods to adapt the system to its environment [24]. This paper shows that by using a model, the system can adapt to the environment by changing its configuration to the one with the highest availability in that situation.

Finally, there is a paper that describes several design concepts that are important for making FPGAs fault tolerant [25]. These concepts can be applied to new systems as well.

Chapter 4

System design

This chapter will describe the design of the ELS and its main components. It will also explain the reasoning and considerations that were made led to the important design decisions that were made.

4.1 Error Logging System

The ELS will need several components and functions to form a fully functional system. It is important to consider how the different parts of the system depend on each other and how they can interact with each other. Various phases can be considered in the process of handling a single error. Figure 4.1 gives an overview of the main stages that will be considered for the ELS. This flow also considers important stages outside the ELS itself, such as the detection of errors. Although not every stage in this process chain will be implemented or created for the ELS, it is still important to consider these phases, so it allows for a smooth integration and future additions to the system.

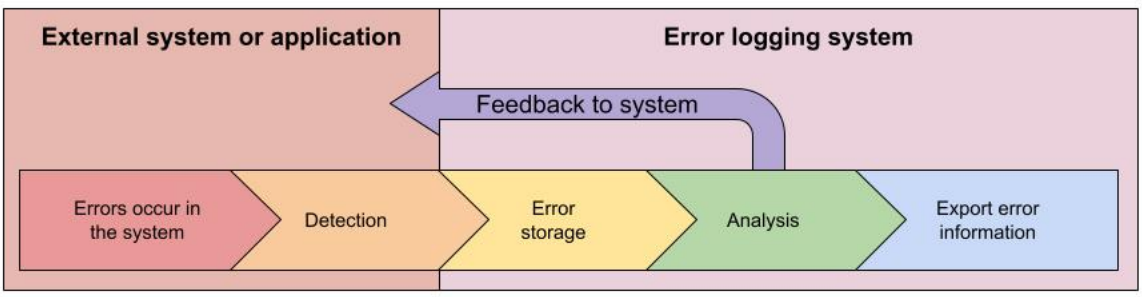

Figure 4.1: An overview of the main stages in the error logging system.

As can be seen in Figure 4.1, the process starts on the left with the introduction

of an actual error within the system. Chapters 1 and 2 introduced various causes that can introduce errors in a system. These errors do not necessarily occur inside the ELS, but they are the main trigger for the system and the error handling process to start.

The second phase in the process is ”detection”. This stage contains all the implemented error detection methods that are present within the system. Section 2.3 mentioned multiple older and newer methods that can be used to detect the different types of hardware errors. These detection methods will not be implemented by the error logging system itself, but it is assumed that systems that add error logging already have detection methods implemented. The focus of the ELS lies on the compatibility with as many different detection methods as possible, so the ELS can use all the detection methods that are already present. Only the signals from these detection methods with the relevant information about the detected errors and system state will be necessary as an input to the error logging system.

Although it was mentioned that the ELS uses the already existing detection methods inside the main system, the ELS should also add detection methods to its own units. In order to ensure its own reliability and integrity, it is important to know if errors occurred inside the logging system and if they influenced the behaviour of the ELS. More about the detection of errors inside the ELS itself will be discussed in Section 4.3.

The detection of errors inside or outside the ELS will trigger the third phase called ”error storage”. This is where the main implementation of the ELS itself starts. This stage contains the interpretation of signals from the error detection methods, the formatting of the error data, storing the data in storage elements and providing a way to make the data accessible. This stage has a lot of different considerations. The RERI format from Section 2.5 was modified and used as the basis for the error storage. This concept is an integral part of the ELS and was very important for other design decisions, so the modifications made for the ELS will be discussed in Section 4.4.2. Once error data is stored, it can be used in the next phase.

The next phase in the error handling process is the analysis. This stage is very flexible and its implementation could be made as simple or complex as desired by the main system. The most basic function of this stage is to retrieve information about errors from the error storage, interpret it and then format the data for export to an external unit and possibly send a command or task to the main system if an error needs to be solved. A lot of considerations and trade-offs can be made in this

stage and it will be highly depended on the implementation and application of the main system. This will be further discussed in Section 4.5.

The last stage in the error logging process is ”Export error information”. This refers to a data connection with an external unit to which the data of errors or information about the systems can be exported. This could be in the form of an external storage unit or a console that will simply output the data. More details about this will be given in Section 4.7.

Now that an overview of the system data flow has been given, the design of the ELS will be described. After that, the individual stages and their design considerations will be explained in more detail in the following sections. The more detailed explanation will follow the same order as the data flow.

4.2 Design overview

Now that the process flow of the ELS has been discussed, the system units that will be needed for these process phases will be combined into a general design of the entire ELS. This design can be seen in Figure 4.2. The colours in the figure represent the different phases in the error handling process as shown in Figure 4.1.

Figure 4.3 shows how the ELS should be placed in relation to the external main system and other external devices or systems that can be used to store the data of the systems, observe these systems or even control them. The red feedback arrow indicates that this is a planned addition in the future, but it will not be added to the current implementation. The orange parts are not necessarily always available, but to make the UART connection of the error logging system useful or to be able to determine whether the error logging system is operating properly, an external system with at least one of these options would be necessary.

Of course, the external system could theoretically be the main application in cases where the main system is able to display or store the data output from the ELS. The downside of using this configuration is that it is not necessarily reliable, since reported errors indicate that the system has problems, so any data stored or displayed might be unreliable as well.

The next sections will go into more detail about all the components in the design and the choices that were considered.

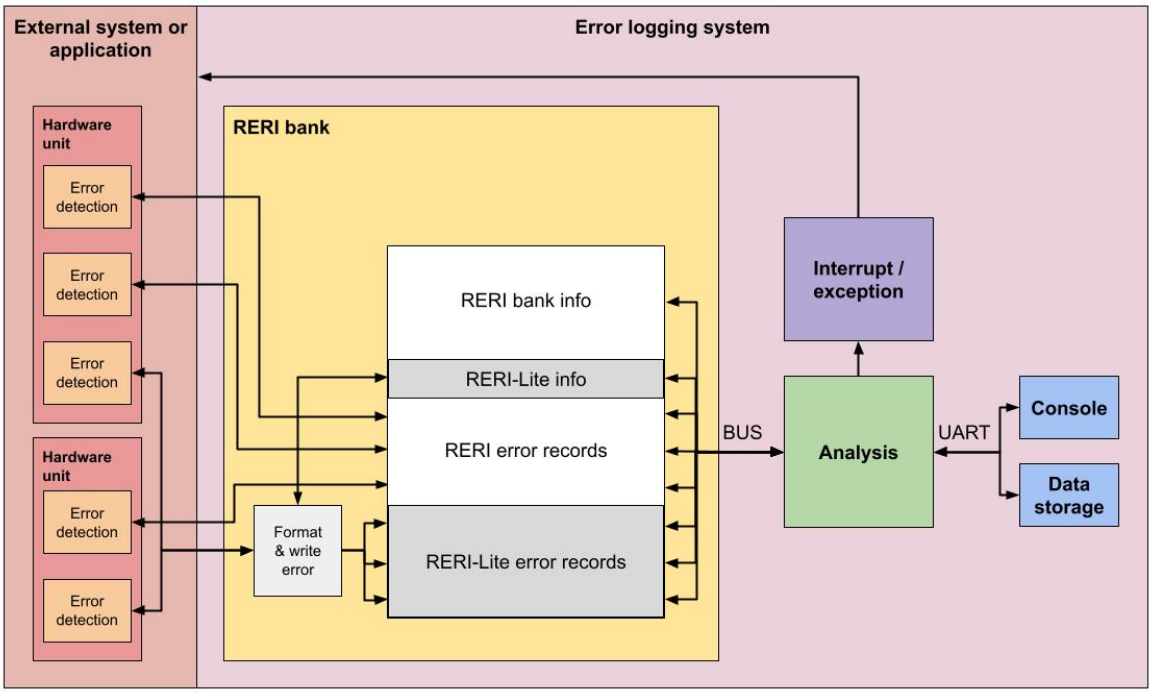

Figure 4.2: A general overview of the ELS and the integration of RERI-Lite in the standard RERI format.

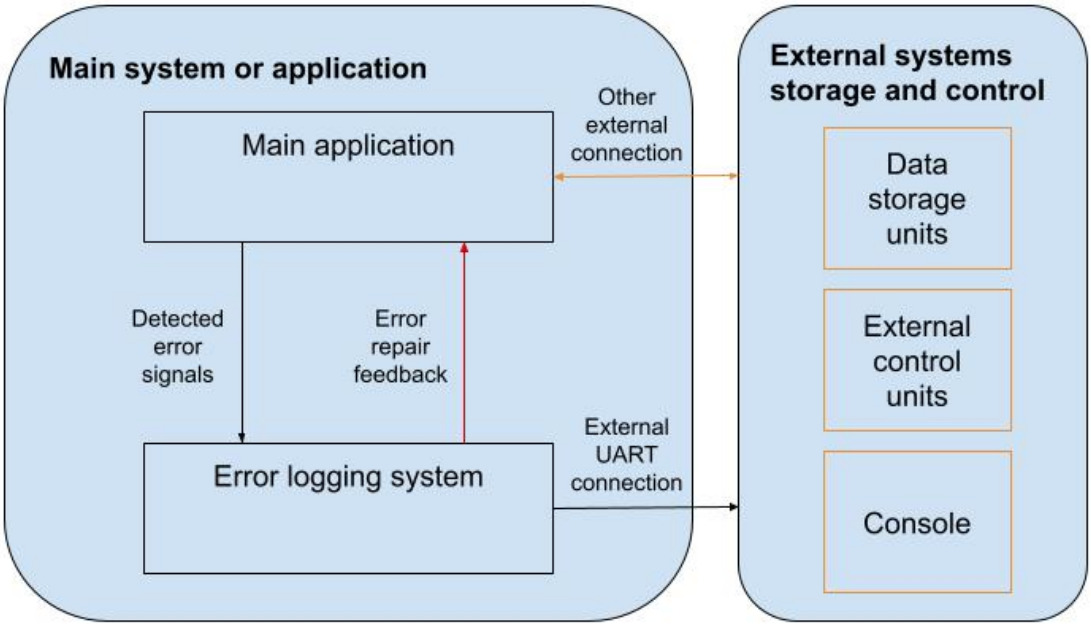

Figure 4.3: An overview of how the ELS should be integrated in other external systems and applications.

4.3 Error detection

There are several different types of possible error signals. The ELS was made in a way to make it as easy as possible to remove, modify or add different error signals and error information to the system. It should be possible to easily make the system compatible with whatever type of error or information the main system to which the ELS capability is added wants to log.

While there is no systematic or standard way to provide information about detected errors in SoCs, there are several errors that usually have the same information available about them. The most frequent and important error types and the information that is usually available will be discussed.

To be able to analyse and log the information of errors, the first step is to detect that errors have actually occurred in the main system. As discussed in Section 2, there are various types of hardware errors and multiple ways to detect them. The important errors that are usually present in most systems will be discussed next.

4.3.1 External errors

Most of the errors that will be logged in the ELS will likely have their origin outside the ELS. Since the (external) main systems to which error logging will be attached are likely to be different, the connection between the ELS and the external system will have to be flexible.

The information that is known in most cases, is based on the detection method. The information that is almost always known is the error type and if the error was corrected or not. Examples of error types are single ECC error, double/mulitple ECC errors, bus protocol errors, modular redundancy and Watch Dog Timer (WDT). There is also information that is not necessary provided by detection methods, but that could be accessed in most main/external systems and cases. For example, in the case of an ECC error, the read/write address can sometimes be accessed (with some changes) and a timestamp could be added to the error depending on whatever type of timestamp the external main system uses.

The Section 4.4.3 will go into further detail about how all this information will be formatted and stored and the format will be optimised by taking into account how likely it is that certain information will be present.

4.3.2 Errors within the Error Logging System

The main system that error logging will be attached to can always change and its components are not always the same. This will likely result in different error types and detection methods for every single application. When it comes to errors within the ELS, the different error types and detection methods can be completely fixed. No major changes will be made to the components that make up the system.

The part of the ELS that is most likely to cause errors is the error storage part. It will contain a lot of data storage elements where bit flips can occur and those could cause problems in the error analysis and could even result in the ELS unintentionally breaking the external main system. Fortunately, this can be relatively easily prevented with the use of SECDED.

With SECDED, as the name suggests, it is only possible to detect double errors, but not correct them. This means that the corrupted data can no longer be used and this could cause problems in the system. That is why it is an option to add scrubbing to memory, so the chance that a single error will turn into a double error can be reduced. The ELS can listen to the error flags of SECDED units in the same way that it would listen to error detection units in the external systems. The main difference will be that these units will always be present and can get a special custom error record.

Scrubbing was not added to the ELS implementation design. The reason for this was the fact that the overhead for the system and the implementation time would likely be large. Also, the error records in the RERI-Lite banks are supposed to be processed and removed as soon as possible and the records will not stay untouched for very long. That is why the choice was made to not add it to this first implementation of the ELS and make it an optional improvement for future development of the system.

However, the memory part is not the only part of the ELS where errors can occur. The ELS also contains a BUS and errors can occur during BUS transactions. If an error occurs, the BUS might be able to throw an error by setting one of the flags. The ELS can always check its own BUS, so just like the error logging SECDED units, the BUS could get its own error record for the possible errors of the BUS type used by the ELS.

Finally, every part with signals and logic in the ELS can generate errors, but it is not equally important for every part of the ELS to be protected. For example, the analysis will (in the future) become the most complicated part with a lot of logic, which makes it more susceptible to errors. The analysis is also very important, since it planned for the system to provide feedback to the external main system in the future. Errors here can (incorrectly) influence the behaviour and results of the external system. Preventing errors here will be the most important.

One way to do this could be to use modular redundancy. By duplicating the hardware of the (analysis) process, the process can be carried out two or more times. By comparing the results, it can be determined if an error occurred in one of the processes. If an error is detected, the process can be repeated to find the correct solution. Alternative, TMR can be added. In this case the process is carried out three times and in case of an error, the processes will vote on the results and the most frequent result will be chosen as the correct one. This will speed up the process, but the downside of this solution is that it will require additional hardware and resources.

The error types that were mentioned included ones for future development versions of this system with its proposed expansions and improvements. However, once the ELS and its future expansions are implemented, the main structure of the ELS and the necessary detection methods are not likely to change unless the whole concept is changed. This means that the future development changes are not likely to result in major problems with these error types. The concept of using the ELS for other logging tasks will be further explained in Section 4.4.3.

4.3.3 Information unrelated to errors

An advantage to using the error records and analysis in a general and customizable way is that the records can be used for other functions than reporting errors as well. For example, if a system wants to log generic information that is unrelated to errors, it is possible to do this by creating a new 'error' format that just uses the register fields for other data. The error analysis can then just ignore any possible actions for this error type and just export the data in the record. While this is not necessarily the most efficient data logging option, it can still be useful in systems that do not need to log a lot of generic data and where the ELS is alreayd implemented. In these cases, the ELS can be used instead of an additional external connection.

4.4 Error storage

This section will answer the third research question: ”How can the error information be stored and made accessible for later use in an efficient way?” It will propose an adjusted version of the RERI format that is smaller and more flexible and explain how it is supposed to work.

4.4.1 RERI downsides

There are aspects of the RERI specification format that make it less ideal for certain types of systems or applications. It is not specified in the RERI specification how the information should be written to or read from these records. Currently, it is up to the specific implementations to determine that mechanism. Also, if a new error occurs, it will overwrite the previous error. In some cases, a system would not want any error information to be overwritten and lost. So, a more flexible version of the RERI specification could be better in some cases.

Furthermore, the largest problem with the RERI specification is that it is relatively large. It uses 8 64-bit registers to log a single error. A lot of the registers are (partially) used for future or custom use. This means that large parts of the registers are usually not used by the error records or will always contain the same redundant data. This makes the efficiency of the records very low. In systems with a lot of errors or systems that do not have much memory, it will require a lot resources if the error bank is implemented according to the specification. Specially in combination with the fact that every record is assigned to a specific error detection unit, which means that even units that do not generate many errors, will constantly use up that entire error record.

Another negative effect that this large error record size has is the amount of bus transactions that will be necessary to write or read the entire record. If standard RERI is fully implemented, not only would it waste memory, it could also slow down any processes that use or interact with these records. For example, reading a RERI record will require multiple bus operations to retrieve the error information. Specially in the process of analysing and reporting error information on a system with high error rates, speed will be important. The longer it takes to handle critical errors, the more time these errors have to propagate. This can cause additional damage to the system or its data. It could also increase the chance that errors are occurring faster than the system is able to handle them. This could lead to full error records and that

can result in the loss of valuable error data.

A final problem with the RERI specification is that it was made for 64-bit systems. This will immediately make the implementation of such a system harder on 32-bit system types. While it is possible to just use two 32-bit registers for a single RERI register, the format will be even less efficient in its implementation. Specially for record entries such as the address, since a 32-bit system usually does not use more than 32 bits for the address, which means that half of the address record will not contain useful information.

Although the standard RERI specification provides full flexibility in which fields are implemented, if each design selects a custom implementation with different features, designing for RERI-based systems would be difficult as each ELS would be fully custom, and each monitored element could present a different interface. Especially for systems such as an ELS, creating new variants that are deviating from the original standard are problematic. Not only does this require additional implementation work, these modifications and specializations will most likely decrease the performance of the systems as well. Therefore, a smaller standard dedicated to embedded systems is required.

4.4.2 RERI-Lite

To address the problems of RERI mentioned in Section 4.4.1, this thesis proposes modifications to the original RERI specification. A new standard format will be created that is more optimized and flexible for embedded systems. This format will be referred to as RERI-Lite.

When it comes to the bank information for this RERI-Lite part, the necessary information can be stored inside the original 64 bytes of standard RERI bank information. In this section, there are registers available for future standard use and custom use. The concept is to use the last 24 bytes of the RERI bank information part to give any information that will be necessary in the future for the RERI-Lite part of the error bank. The information part might become useful in the future if more complex features of the RERI-Lite records and analysis will be added. Examples could be the addition of an used error records counter or the highest priority error in the bank.

The addresses after the standard RERI error records could then be used for the

RERI-Lite records. The specific format of these records will be explained in Section 4.4.3, but their main design philosophy is to decrease the size of these records and have less unused data storage elements. It will also reduce the amount of necessary overhead. For example, to read an error record, less bus operations will be necessary. This is expected to speed up the processes of the ELS and reduce the amount of necessary resources.

The idea behind the addition of RERI-Lite is to make it compatible with the standard RERI format as well. This will make it easier for systems (or components) that follow the standard RERI format to be used in the ELS as well. By combining the formats, applications can use either or both ways to log errors. An overview of how the RERI-Lite parts will be able to fit in the format of the standard RERI bank and the new address offsets can be seen in Table 4.1.

Error bank layout		
Offset (bytes)	**Registers**	**Size (bytes)**
0	Bank info	40
40	RERI-Lite info	24
64 + 64 * 0	RERI error record 1	64
...	...	61 * 64
64 + 64 * 62	RERI error record 63	64
4096 + 16 * 0	RERI-Lite error record 1	16
...	...	(n-2) * 16
4096 + 16 * (n-1)	RERI-Lite error record n	16

Table 4.1: An overview for the concept of combining an RERI-Lite format inside a standard RERI error bank.

4.4.3 Error records

This section first describes the format of the generic RERI-Lite error record and will then go into some of the error formats that were specifically made for certain errors. It will describe the individual fields within the format and what it will represent. It will also describe how the records work and how new ones should be added.

The generic record format shown in Figure 4.4. The four 32-bit registers that make up the error records are: ”*information register*”, ”*additional error info*”, ”*address*” and ”*timestamp*”. Its size is 4 times 32 bits, resulting in a maximum of 128

bits of data per error record. This is only a fourth of the 512 bits used in the full standard RERI error records. Also, a 32-bit compatible format was chosen for the error records. This decision was not only made because the NEORV32 used in the beam experiment is a 32-bit system architecture, but also because it is easier to implement a 32-bit format in a 64-bit system than vice versa, so it will be easier to implement this system in other 32- and 64-bit systems. The colours indicate the usage of the register (fields). Green means that every record uses that register or field. Orange means that it is optional. If a register or field is red or gray, it means that it is not used or unspecified.

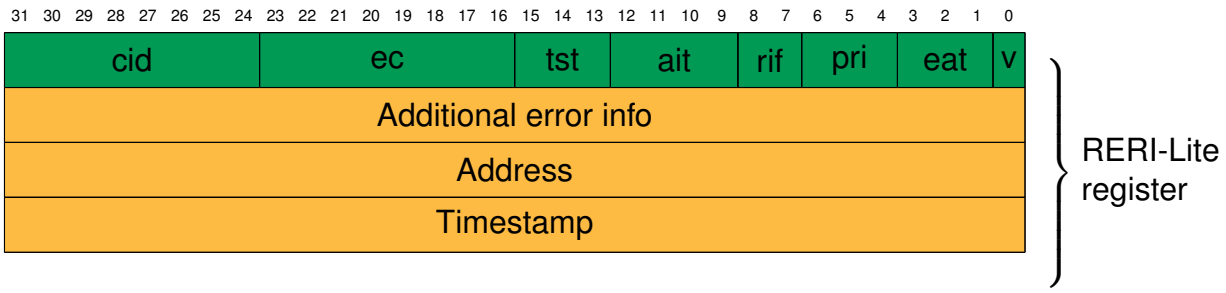

Figure 4.4: The generic error record format.

The first register of 32 bits is used to provide the most important and general data that is usually available for every error type. It will also provide information about the use of the other three optional and customizable registers. Some of the fields use the same names, encodings and functions as the standard RERI specification. A function description of every register and its fields will now be given.

The first bit is used as the Valid or "*v*" field. It follows the standard RERI specification [1] and indicates if this specific record in the error bank is valid or not. A '1' indicates that the record is valid and a '0' means that the record is not valid.

The fields "*eat*", "*rif*", "*ait*" and "*tst*" all indicate how the other three registers are used and what format they use in this record.

First, the Extra Adress Timestamp or "*eat*" field indicates if and which of the other three registers are used by the error record. For example, "100" indicates that only the Extra error info register is used. The use of the Address register would be indicated by "010" and the use of the Timestamp register is indicated by "001". Of course combinations can be made as well, so "111" would mean that all the registers are used and the error record has its maximum size of 128 bits.

The "*rif*" (Register Information Format) field indicates the register format that is used by the error and it uses 2 bits. "00" indicates that the additional error info register is not used at all. The code "11" indicates that the full register is used as one single field. In case it is not using the full 32 bits for one data element, the code "01" is used. A field in the "*additional error info*" register itself will indicate the format of the additional info register. This format is used by record types like the type 4 ECC format. This format uses the first 4 bits of the "additional error info" register as an indicator of the amount of used bytes and custom format in that register. The next 4 bits are used for the (custom) data itself. This format will later be explained in more detail.

The Address-or-Info-Type "*ait*" and the TimeStamp-Type "*tst*" contain encodings that indicate the type of addresses and timestamps that are stored in their respective registers. The "*ait*" field already existed in the standard RERI specification and its encodings are reused and shown in Table 4.2. The "*tst*" field is based on the same concept and its timestamp encodings are shown in Table 4.3. Encoding 3 for the "*tst*" field is used as a custom format and will be discussed later on. While these tables show the encodings for this specific implementation, they can always be used for other purposes and custom uses, but it should always be checked if the overwritten format does not cause problems in the analysis. For example, in the future with error feedback, if an address is wrongly interpreted, it might cause an incorrect repair of the system. However, for data export and post analysis, it should never be a problem.

Encodings	**Binary**	**Description**
0	0000	None. The contents of the address register are unused.
1	0001	Supervisor Physical Address (SPA).
2	0010	Guest Physical Address (GPA).
3	0011	Virtual Address (VA).
4-15	XXXX	Component-specific address or information

Table 4.2: Address-or-information type encodings from the RISC-V RERI Architecture Specification [1].

The "*ec*" field is for the Error-Code of the error record. It follows the same standard as the "*ec*" field in the RERI specification and uses the error code encodings as described in that specification. It can be customized, since the original encodings already had encodings above 26 reserved for future standard use and custom use.

Encodings	Binary	Description
0	000	None. The contents of the timestamp register are unused.
1	001	Local cycles counter
2	010	Global counter
3	011	ECC cycle counter
4-7	1XX	Encodings reserved for future custom timestamp types

Table 4.3: Timestamp type encodings for the ”tst” field, using the same concept as the ”ait” field encodings.

The ”*cid*” field stands for Component ID field. It should contain an 8-bit unique ID for the component that reported the error. The component that this refers to can be as large with as many types of errors as the specific application wants, but it should be either provided by the component that provides the error signals or one should be assigned when the component is connected to the write unit of the error logging system.

In a system with 64-bit registers, the 32 unused bits in the ”*information register*” and ”*additional error info*” can either be used for additional custom information or left unused. The other two registers ”*address*” and ”*timestamp*” can just be upgraded to 64 bits, since a 64-bit system will likely have 64-bit addresses and timestamps and other data as well. An overview of the error encodings can be found in the Appendix in Figure A.2.

Next some custom error formats that were implemented in the experimental version of the ELS will be explained. This will also show how the customization of these records can be used to create new records for other error types and very specific functions. These custom records were specifically created to be used in a radiation beam experiment.

The first custom error record is for error type 1. This record is meant for most SECDED ECC errors. The format for this error record is shown in Figure 4.5. This format can be used for two different errors. Single bit errors and double/multiple bit errors. There is usually no difference in the available error information, so the only difference will be in the error code, one for each type.

An option for this error record would be to use the additional error info register as a counter. In cases where many of these ECC errors are expected from the same component, it would be possible to count the amount of (duplicate) ECC errors and only send the data to a record in specific cases. This could reduce the possibility of

the ELS getting overwhelmed with error records because of a single error. Specially if these single errors can automatically be fixed or if an unfixable error keeps reappearing.

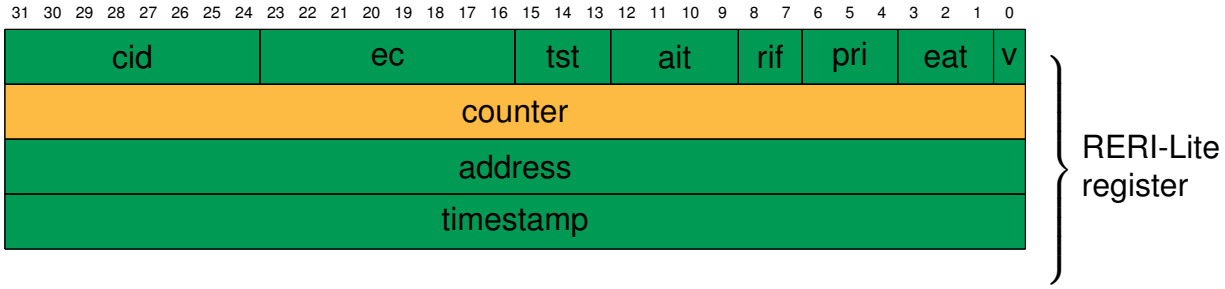

Figure 4.5: The error record format of a type 1 SECDED error.

The second custom error record is for error type 2. This record is meant for bus errors. The format for this error record is shown in Figure 4.6. The additional error info register is used to store the data that was on the response bus. The requested address and a timestamp of the error can be stored in the address and timestamp registers. Based on the type of address and timestamp used by the systems, the ”*tst*” and ”*ait*” fields should be set with the correct type.

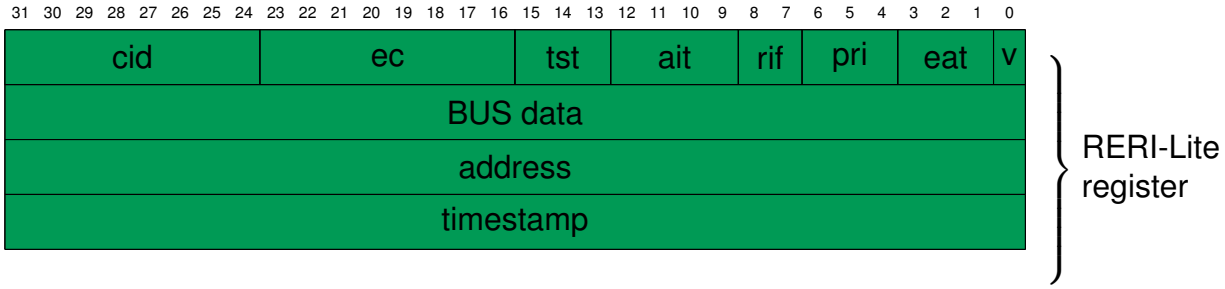

Figure 4.6: The error record format of a type 2 bus error.

The third custom error record is for sparrow data. This record was supposed to be used for a component in the beam experiment setup. This component would provide 32 bits of data to the ELS. The record would store this in the Additional error info register. Note that ”*rif*” field would be ”11” in this case, since the entire register is used for one data value of 32 bits. Similarly the ”*eat*” field would be ”100”.

The format of error type 4 is shown in Figure 4.8. It was supposed to be used for logging the ECC data of a specific beam experiment setup. It is similar to the first error type, which is also used for ECC errors, but it has been slightly customized, so it can log more data for a specific test setup during the beam experiments. The

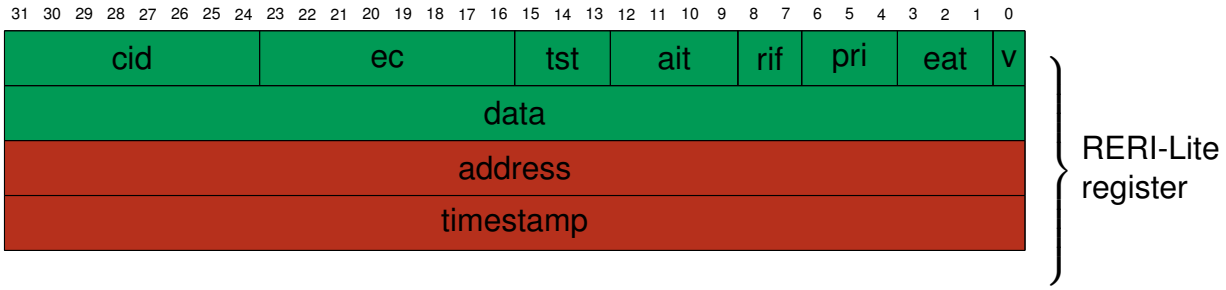

Figure 4.7: The error record format of a type 3 Sparrow error, which is only used to log data and not an actual error.

records were eventually not used and tested in the beam experiment, but it still gives an important example of the customization of these RERI-Lite error records.

In this case, the additional error info register uses a custom format, so the "*rif*" field uses "01". This means that the first four bits in that register (the "*byte*" field) will indicate the custom format for the entire register. So since this is the first custom format, it got code "0001", but it is possible to add 16 new formats. If it is necessary to create extra custom formats in the future, more bits can be added to this field. Also, in combination with the specific error codes and component ID's, it would be possible to reuse the same codes, while an analysis in real-time or afterwards can still distinguish the different formats and decode the information within the error record. However, this would require extra logic and might slow down the analysis, so just using a unique byte code is for now preferred.

Other three fields in the format are specific data that were supposed to be logged during the experiment. The "*ecc*" field would indicate if there was a single or double bit error, the "*bf*" field would give a single bit of data from the bloom-filter. Finally, the "*xor*" field would indicate if there were differences in the ECC flags. To make sure that it was not needed to log this record every cycle, even if no errors were detected, a counter would be added to the ECC component that would count the amount of cycles until a difference was found by the "*xor*" data. This way it would be possible to determine how many cycles were correct before a difference was detected. This kept the necessary data, while lowering the amount of necessary error messages.

This RERI-Lite format is more compressed and flexible than the standard RERI format. This new RERI format is only a fourth of the standard format size and has a lot of room for customization. It allows error records to only use parts of the register that will actually contain useful information. Much of the (often) unused fields have been removed and the fact that the records are not dedicated to a specific error

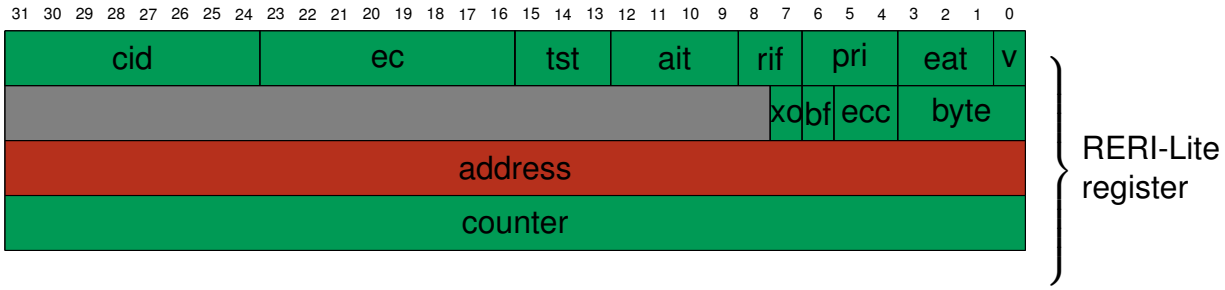

Figure 4.8: The error record format of a type 4 ECC error.

detection unit will allow more freedom in the choice of the error bank size. It was also made with 32-bit systems as the focus, but as discussed, it should be easy to implement it for 64-bit systems as well. This makes the system more easily compatible with both system types. Analysis or reporting units can use this flexibility to ignore unused parts of the record and to speed up their processes, resulting in better performance. In addition, systems with limited available resources can opt for RERI-Lite to save memory space. This makes RERI-Lite better for embedded systems, compared to the standard RERI format.

4.5 Error analysis

If a system wants to keep its availability and reliability as high as possible, it is necessary to resolve any detected errors that are problematic in real-time. The data of the errors can of course be analysed after it is exported to an externally connected device, either in real-time or at a later point in time, but it would require another connection to the main system and add a significant delay to the handling of errors.

The longer it takes for errors to be resolved, the more likely it becomes that the error will cause the following operations to fail or be incorrect as well. Or in critical systems, it can stall the operations and cause important deadlines to be missed or important data to be lost. To prevent this, the ELS should (in the future) be able to perform its own error analysis and be able to repair the system and prevent further damage to the system.

One option would be to have the analysis be done by a processor. This is the most flexible option and allows for a more extensive analysis of error information. However, it will introduce some new problems and trade-offs.

This option will either require the use of one or more of the processors on the

main system, or a dedicated processor. Using processors from the main system has the advantage that the analysis of resources and forcing repairs and changes to the system to resolve the potential errors will be easier. However, not every system to which error logging might be added will have a processor available. Even in systems where one is present, it might not be usable by an external application like the ELS or it could slow down the main systems operations significantly.

The logging system would also require a much more extensive integration with the main system and would make it harder to add or remove it from any system. This would reduce the modularity of the ELS. Finally, in case of critical errors in that processor, the analysis could come to a wrong conclusion about an error and provide a wrong solution. This will cause the system to fail the repair or even cause itself more harm.

The next option would be to create a new dedicated processor to the ELS and have it execute the analysis. However, adding such a processor for just the error analysis will require a lot of hardware, which might not always be available. Another important aspect to consider for the use of a processor in the analysis, is the timing. A processor will likely take multiple cycles to be able to analyse any error. As mentioned before, it is important to reduce the time it takes to fix errors as much as possible.

The last option is to use a state machine. It is easy to implement and control and will not require as much overhead and resources as the other solutions. It will reduce the chance of new critical errors happening in the analysis and it will be easy to add, modify or remove more steps to the analysis process in the future.

Other advantages of the state machine are that it will be easier to skip different parts of the error analysis based on settings, this can be used in the future to change the complexity of the analysis while the ELS is in use. Since the analysis speed is an important factor for the effectiveness of any type of error recovery, it would be best for the analysis to have a quick way of accessing the required data. A potential way of speeding this process up in the future will be presented in Section 4.8.

In the end the choice was made to use a state machine for the analysis of the ELS.

4.6 Error recovery

Eventually the goal of the ELS will be to automatically recover any system from errors that would normally break the system. It should be able to to this on its own and in real-time without the need of an external process or analysis. Due to time constraints, the focus of this thesis switched to a design and implementation that just handled the logging of errors. A design without error feedback was created and implemented. However, since error recovery was meant as a major function of the ELS, the design did already account for it and considered some important aspects that will be important when adding it in the future.

Error recovery will depend highly on the external main system and the type of errors that can be detected there. There are various options that could be used to influence the external system. The first one is to create and implement a completely new unit inside the external system. However, since these systems can vary a lot, it will be difficult to create a general system. It would require a lot of customization for every new system that it gets added to. This will come at the cost of the modularity of the ELS and that is not ideal.

Another option is to signal the external system with flags or error codes and then create interrupts for the process that is running. It can also be done by using another connection to a processor like accessing available ports like a UART connection on the external system and write new data or instructions directly to the system memory or processor. This will require any processor to be able to identify this feedback and switch to the execution those new instructions. Another problem is that the necessary instructions to fix it will depend on the system architecture.

However, the main problem with these options is that they can only be done in systems where some sort of processor core is available that can handle these inputs. For example, simple input and output systems without any sort of instructions or adjustable process will be much harder to repair. Without a single processor or other control unit to steer all the other parts of the system, it will be hard to make any changes or repairs in the erroneous components. Also, in systems with only one core, the repair will have to entirely suspend the process that is running in the main system. Depending on the application, this might be problematic. Specially since the ELS will not be able to determine if the running process is more important than the potential repair. In cases where a lot of minor errors are detected, the repair feedback could actually stall important processes. This is why it might be necessary to address the components with errors directly in the systems with one or less cen-

tral control units.

Error recovery for the main external system was of course the main goal of the ELS. However, error recovery should also be implemented for the ELS itself. This error recovery will be much easier, since everything will be internally. Any recommended changes decided in the analysis unit can directly be communicated to the designated component. Also, the available error detection units and information will always be the same, so there is no need to generalize this process. That is why it is possible to tightly integrate it with the individual components inside the ELS. The recovery system needs to be made and implemented once and there will be no need for customization afterward.

The repairs that will be necessary inside the error logging system will be very straightforward. As discussed in Section 4.3.2, the main error detection will be SECDED, bus errors and modular redundancy of the analysis. bus errors and modular redundancy errors can be handled easily. These processes can be repeated until they succeed. In the case of SECDED, the single bit errors can also be easily repaired. However, if the bus transaction and the analysis results keep failing, or if double bit errors are detected, there are not much other options than to log this problem and the error data and then skip this particular error record or error analysis. If the same type of problems keep occurring, the only real solution would be to reset the error logging system.

4.7 Error export

While error recovery is meant as an important feature of the ELS in the future, in this thesis the focus was put on logging the error data. The main reasons for this were the fact that the ability to log the errors would allow the system to be used in the UT radiation beam experiment. The other reason is that it will be necessary to log the error data anyway if any type error recovery system wants to be tested in the future. Without the ability to check what errors the logging system is detecting and how it is handling those in the physical system, it will be hard to test whether or not the recovery system is actually working properly.

4.7.1 UART protocol

The connection that was chosen for the communication between the ELS and an external device was a Universal Asynchronous Receiver/Transmitter (UART) protocol. UART can be used as a serial communication protocol for the transfer of data between devices [26]. As the name suggests, the communication is asynchronous, so the sender and receiver do not need a shared clock. Instead, the protocol relies on a predetermined baud rate to synchronise the data speed. Baud is the number of symbols transferred per second, so 1 baud is equivalent to one bit per second. Equation 4.1 can be used to calculate how many bits per second are transferred. In this equation, the output rate of the message is in Hz.

$$Baud = number\ of\ bytes * output\ rate\ of\ the\ message \tag{4.1}$$

In the UART protocol, the data (bits) are transmitted in frames. These UART frames consist of a start bit, one or two stop bits, the data bits and an optional parity bit. An example of the signal for such a frame is shown in Figure 4.9. Equation 4.2 shows the calculation for the total amount of bits in a frame. One of the most used frame configurations is 8N1. This uses 1 start bit, 1 stop bit, no parity bit and 8 data bits. This results in a total frame size of 10 bits. So, for each byte of data, 10 frame bits are transmitted. This makes the relation between baud rate, frames and data bits very easy.

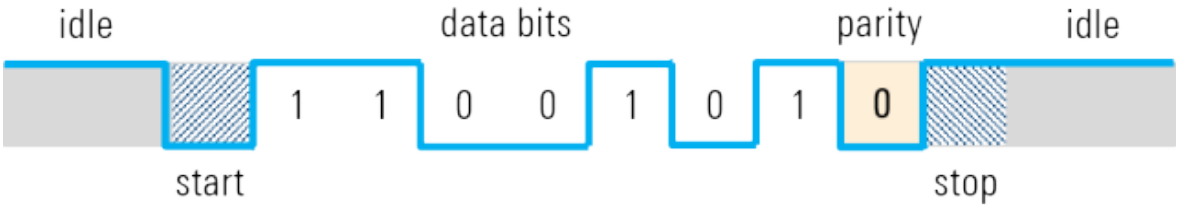

Figure 4.9: An example of the signal for a single UART frame from [27].

$$Bits_{frame} = bits_{start} + bits_{data} + bits_{parity} + bits_{stop} \tag{4.2}$$

The baud rate in bits per second (bps) is the rate of symbols transferred over the connection per second. This baud rate is not fixed, but both the sender and receiver

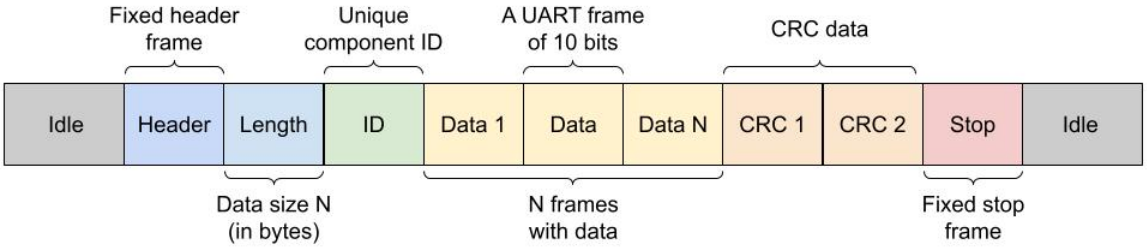

Figure 4.10: The formation of an UART message in the beam experiment protocol.

should have the same configuration of baud rate and message frame structure. The standard baud rates that are often used are: 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800 and 921600. The best choice for this baud rate will depend on the amount of data that needs to be transmitted and the accuracy of the used systems. These transmission rates will be discussed further in Section 4.7.3.

4.7.2 Experiment protocol

The UT radiation beam experiment uses a specific UART protocol to communicate with the setups. This UART implementation has a specific format and those implementation details will be explained here. The UART implementation can be found here [28].

The beam experiment uses the 8N1 frame configuration without an additional parity bit. The setup is configured to receive messages consisting of multiple UART frames. One such message contains a header (1 frame), the message length (1 frame), a component/setup ID (1 frame), CRC data of the message (2 frames), a stop frame (1 frame) and a variable amount of frames for the data (depends on the message length). Figure 4.10 shows how the formation of the UART signal for such a message.

Every message will require a fixed amount of 6 frames and a variable number of frames for the actual data that needs to be transmitted. This requires additional overhead and time in the ELS to be able to transmit data. By adding extra data in a single message, for example by combining multiple error records in a single message, the amount of relative overhead can be reduced. The next Section 4.7.3 will determine if such measures are necessary to reach a high enough UART throughput for the expected amount of error data.

4.7.3 Bandwidth limitations

It is important to determine that the UART connection is able to handle the amount of data that needs to be transmitted. Using the equations in Section 4.7.1, a tool was made in Microsoft Excel, to calculate the transmission data limits for different configurations of the UART frames, experiment protocol messages and baud rates. It can also calculate how much data needs to be transmitted for the range of expected error rates in the ELS. By changing the parameters of the UART configuration and the error rates, the tool will determine if the bandwidth of the various configurations will be sufficient.

Figure A.1 in the Appendix shows the calculation results of the Excel tool with the default experiment message and the 8N1 UART frame configuration. The table on the top right shows this configuration. All the fields that have an orange background are meant to be variables that can be changed by the user of the tool. For example, the parity bit can be set to two and that will result in a total frame size of 11. Since all the fields are connected, it will then do all the calculations with this new frame size and determine the new data requirements and limits. Other fields that can be changed are the amount of data frames that are necessary for the different error record messages and the expected range of error rates. The tool will also make sure that the data size is not greater than the length field can indicate, which is 256 (2 to the power 8 options). So, if a data size of more than 256 bytes is chosen, length should use at least 2 frames to give the size of the message.

As can be seen in the table at the bottom, the baud rates of 38400 and higher will be able to handle the entire range of expected error rates. The baud rates of 9600 and 19200 would still be able to handle the lower error rates, which will most likely be enough, but it is better to keep a safe margin. The higher baud rates of 230400 and 460800 will need much more precision from the systems, so a baud rate of 38400 or 115200 will be the best and safest option. Eventually the 115200 baud rate was chosen for the beam experiment setup. This means that every error record can be transmitted in a separate message.

Figure 4.11, shows the bit rate limits for various UART baud rate configurations. The line in red shows the expected required bit rate that is necessary to send all the UART messages with the error data. As can be seen, only the two lowest baud rates will be insufficient to transmit all the error messages in case of a high error rate of more than 350 errors per minute. It will require a much higher error rate to reach the limit of the 115200 Baud rate, so this seems to be a safe option for the UART settings. Table 4.4 shows whether the various baud rates will have a high enough

bit rate to handle the error messages of different error rates.

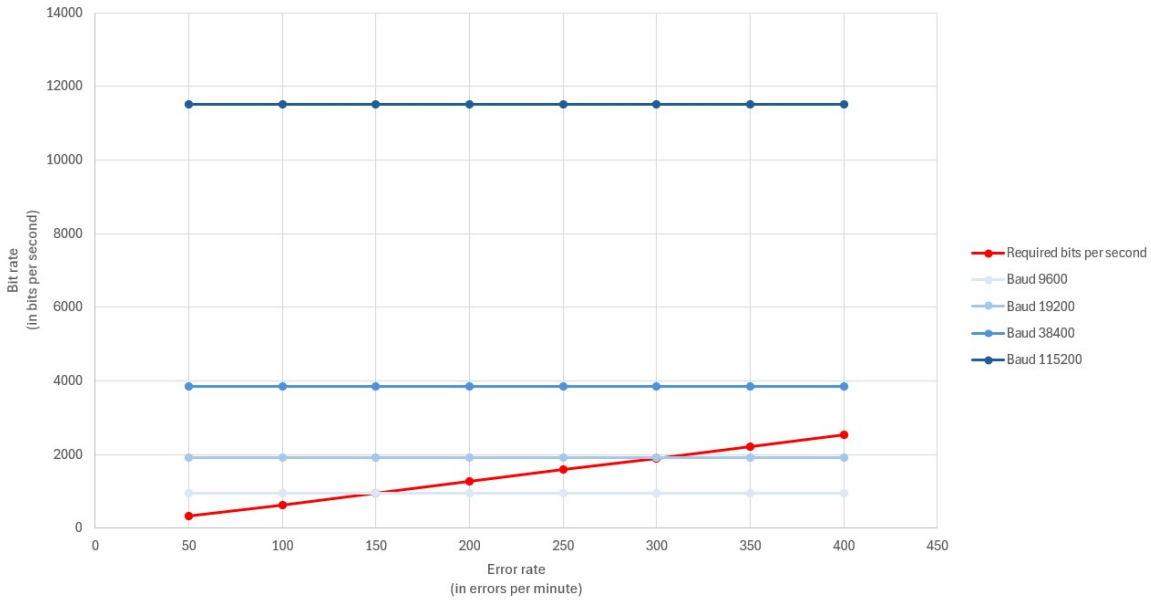

Figure 4.11: A graph created by the Excel tool to show the bit limits for various UART baud rate configurations versus the required bit rates based on various error rates.

Output (bits/s)		**Check for the different error rates (errors/min)**							
baudrate	**Data bits**	*50*	*100*	*150*	*200*	*250*	*300*	*350*	*400*
9600	**768**	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
19200	**1536**	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
38400	**3072**	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
115200	**9216**	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
230400	**18432**	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
460800	**36864**	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

Table 4.4: This table shows if the UART bit rate will be high enough to handle the expected bit rate of the error messages for different error rates.

4.8 Optional extensions

There are still various features that could be added to the system to improve its capabilities and compatibility with other systems. It was not possible to integrate all these concepts and improvements in the final design, due to the time limit of this thesis and the beam experiment. The choice was made to create a design that was

manageable within the time limit and that was usable for the radiation beam experiment. However, some of the possible development areas and concepts that the ELS already considered or accounted for in the current design will be discussed here.

Some simple and useful improvements to the error logging system would be to add more error types to the system. These new error types are easy to add and there are more specific error types that can still be added, such as an error type for a WDT.

When it comes to completely new features, the first option would be to use an error priority system. As mentioned in Section 4.4.3, a 3 bit register field is already present in the generic error record format, but it currently does not have a function. The improvement would be to actually use this priority in the ELS. For example, by giving every single error type its own optional priority, the error bank and analysis could decide to handle these errors first and overwrite lower or unimportant priority errors in the bank. For example, this could prevent the analysis from wasting time on handling already corrected errors instead of using that time for a newer critical error that could cause additional problems in the future.

Such a priority system could use both a predetermined priority based on the error type and a dynamic priority based on the system settings, system state or the timing and amount of various error types. Another possible extension to this priority system could be to provide an easy way to make the most useful information about the highest priority error quickly accessible to the error analysis.

A possible implementation of this could be the addition of the proposed ”output selection” unit in Figure 4.12. By immediately putting the relevant error information on dedicated signals accessible to the analysis unit, the analysis unit will not have to wait for all the bus reads and this could speed up the error processing and could decrease the amount of necessary clock cycles to process and fix problematic errors. This will especially make a difference if feedback to the external main system is implemented, since it reduces the chance of the system breaking down due to an uncorrected error.

As explained in Section 4.3.2, scrubbing was not added to the design and implementation of the ELS, due to the low chance of double bit errors. However, since the chance is not zero, it would be useful to add scrubbing as an optional feature to the ELS as an improvement. In some system applications the priority might be to never lose error logging data due to a double error, even if the chance is very low. If

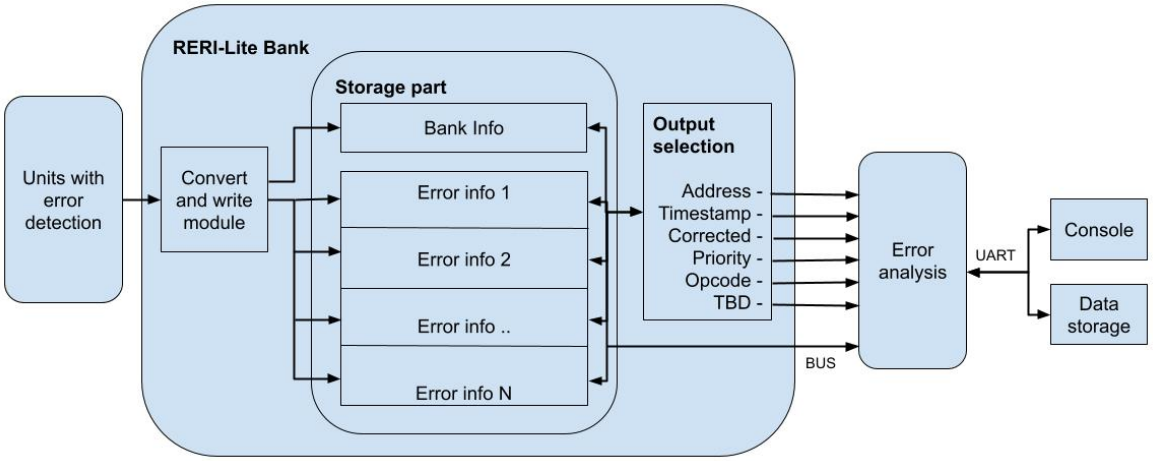

Figure 4.12: A possible future extension where data can be automatically presented and bus cycles can be prevented.

a system has a lot of hardware resources left for an ELS, it might be worthwhile to add it to the system.

It would also be worthwhile to add some sort of timer or counter to the ELS. Not all systems to which error logging will be added might have a timestamp available for error records, but in those cases the one from the ELS itself could be used. The advantage of using such a timestamp would be that the order of the recorded errors can always be reconstructed. Since the errors are not always written to the records in order, due to cleared or overwritten error records, the order might be hard to determine without some sort of timestamp. Especially if a priority system is added in a future version, the order of error handling will change and it will be even harder to determine the original detection order of the error records.

The next major improvement for the ELS would be to add the possibility of a more complex error analysis. While some aspects would depend on the type and amount of expected errors in the system, a more general analysis can still be done. With the addition of a priority, the order of analysis would already become more dynamic and based on the priority, the analysis might skip less important steps of the analysis. A corrected single bit ECC error should not cause any problems in the system and does not need a fix. However, it could be useful to use it for the calculation of system statistics to determine the general health of the system. For example, if an abnormal number of single bit ECC errors are detected, it might indicate that something else is causing problems in the system.

Also, in case the error bank is almost full with records, it might be useful to skip

(parts of) the analysis of less important or corrected errors and just export them, to make sure that the error record bank will not completely fill up.

Related to the addition of all these analysis options, another possible improvement for the future would be to add the option to control the ELS over the UART. As the analysis options might become more complex and situational in the future, it might be desirable for a way to change the system from the outside, even after the initial setup and during runtime. This could be realised by setting up a UART connection in the other direction as well and using simple commands to change the system settings.

Chapter 5

System implementation

This chapter will explain how the final ELS design that was described in Section 4.2 was implemented and how the implementation was tested and validated in both simulations and physical setups.

5.1 Implementation details

While the architecture of the system is according to the design in Section 4.2, some implementation details and choices had important consequences for the behaviour of the system and future development. The implementation of the system will be explained in the order of the data flow and will start with the top-level integration between the error logging system and the external main system to which it will be added.

5.1.1 System overview and VHDL hierarchy

A Gitlab repository was made [29], so the necessary code can be easily cloned and integrated into other projects. All the necessary code for the ELS can be found in the ”IPs” directory. This directory contains multiple subdirectories. The main one is the ”rtl” directory, which contains all the major entities that make up the entire ELS. The other subdirectories are meant for files that are necessary for specific implementations. In this case, there are directories for the ”ARTY” and the ”SF2” setups. These directories contain files like the integrated ELS in other top-levels or constraint files. More about these specific setups will be discussed in Section 5.3.

The code implementation of the ELS was mainly done in the VHDL language. A VHDL top-level was created in a way to make it as easy as possible to add it to

any other system top-level and to need as little integration specific changes. The top-level design can be seen in Figure 5.1.

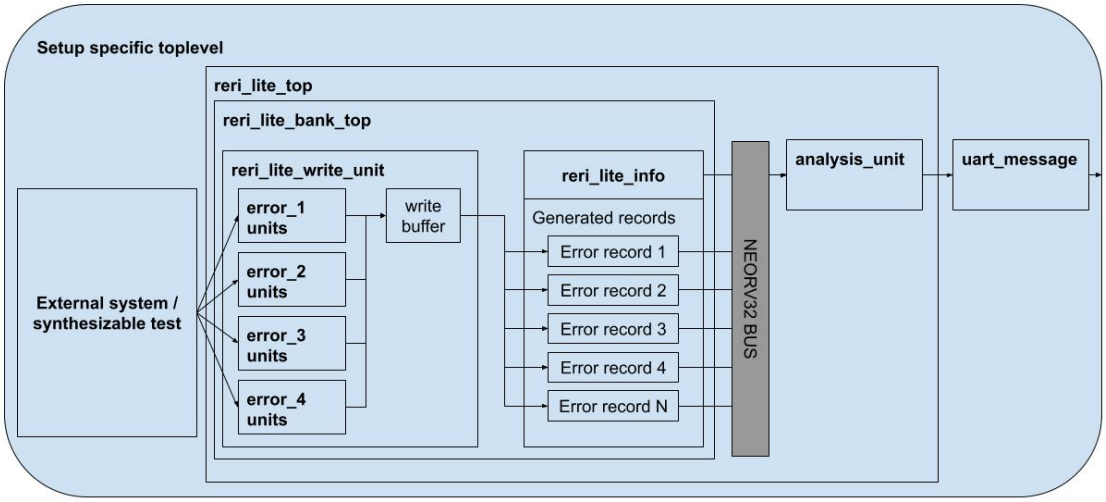

Figure 5.1: An overview of the VHDL top-level hierarchy and how the entities relate to each other.

The setup specific top-level block represents the top-level of whatever project the ELS gets added to. This has most likely some sort of system or application that will act as the external system to provide error signals. This external system can also be replaced by a synthesizable test unit to test the error logging system. This option will be used in Section 5.3 as well. Finally, the ”uart_message” unit is not included in the ”reri_litetop” entity, since it was provided and is technically a part of the beam experiment setup. However, this UART protocol could of course be used by other projects, so it was added to the top-level in such a way that it is easy to add this specific UART module to other projects as well.

5.1.2 Write unit

The first important entity to discuss is the ”reri_lite_write_unit”. It will generate a variable amount of units for each error type, depending on the amount of available error signals for that specific error type. These units can be custom made for every available error type and will convert the error signals and information to one of the error record formats discussed in Section 4.4.3. These units will constantly monitor the error signals and if errors are detected, they will form an error record and send it to the error bank. The number of each specific error unit can be changed in the

top-level parameters or in the ”reri_lite_package” file with necessary constants and types. These error units and the signals coming from the external system will have to be manually connected in the top-level unfortunately, since this will be different for any setup.

Another important aspect of this entity is the logic. It will constantly check all the error units for new error records. If a new record is available in a unit, it will set this on the output to the error bank. A small buffer of size 1 is added in case two error detection units want to write an error record at the same time. If that happens, the buffer will make sure that the record gets written to the error bank in the next clock cycle. The buffer was only made size 1 to make sure that the system does not become needlessly large. The choice was made to leave the buffer at size one and increase the size if experiments indicated the need for a larger buffer.

5.1.3 Error storage

The next part of the system that was implemented was the error storage. This is basically the RERI-Lite bank and it currently contains an entity for bank info about the RERI-Lite part and a variable amount of RERI-Lite error records. Originally the standard RERI bank implementation was also created, however, this was later removed due to complications and time constraints.

Register generation

The first implementation of the error bank (which had the RERI-Lite format added to a standard RERI error bank instance) used the ”Register Tool” from opentitan [30]. This was a very easy tool that just required a Hjson file to describe the amount, type and default values of registers. The tool would then be able to generate a Verilog top-level and all the other required files for these registers. At the time it was seen as a good way to quickly create the large RERI bank implementation and it also meant that the registers and bus were already used and tested before. The downside was that the generated top-level was in Verilog instead of VHDL. However, a design can be made with mixed Verilog and VHDL languages. A quick test was done with a simplified version of the register bank and no problems were encountered at this time. So, the choice was made to try the implementation with the register generation tool.

Unfortunately, the integration of these generated files would at later stages in the implementation process cause problems. This was due to the registers becoming

more complicated as other parts of the RERI bank were added. This required more complicated register types and variables and the top-level would require more Verilog dependencies. At some point, the design and testing tools like Questasim and Vivado would encounter various errors, that could not be resolved. The most likely reason was that one or more of the new dependencies were causing errors in mixed language designs. Several work-around solutions were tried to get the generated design to work, however, they would usually cause new problems further in the design, so a working version could not be achieved with this method.

New register implementation

Since it was not possible to get the generated implementation to work in a reasonable time, it was decided to drop the generation tool and switch to a different register implementation, based on the one used in the NEORV32 [14]. This one was used in earlier UT projects and already included SECDED. That was very useful, but the downside was that this custom implementation cost a lot more time than the generated implementation. It took a lot of time to manually create and adjust the implementation for every single register defined in the RERI bank. That is why the choice was made to not implement an instance of the standard RERI bank and only create the RERI-Lite part. It would have been nice to show how the RERI-Lite format can be integrated inside the standard RERI format, but this did not have the priority, since the standard RERI registers were not supposed to be used for the beam experiments and they were not necessary to test the concept of the RERI-Lite part.

Another consequence of the switch from the register generation was that the TL-UL bus generated by the tool was no longer usable either. This meant that a switch to the NEORV32 bus was needed as well. This required additional changes in the analysis, since that entity accounted for the TL-UL bus protocol, which was slightly different.

5.1.4 Analysis unit

The next part of the implementation is the ”analysis_unit”. This will load, interpret and transmit the data inside the error records. This unit is also where the feedback to the core will be determined. As mentioned in Section 4.5, the analysis unit was implemented as a state machine. All the parts of the analysis process can have multiple different states, but they can mostly be divided in four main parts or process

stages.

The first main part are all the states that are related to the communication with the error bank. This part will check if any error records are valid and in case a valid one is detected, it will initiate bus transactions to retrieve the data from the error record. These states will set the bus signals, wait for a response and make the received bus data available for the analysis.

The second part is the actual analysis part. Currently, this analysis is not very complex, since there is not much to analyse yet. However, for now it will mainly get the necessary information like the component ID and data length to be able to form the UART message for the export of the error record data.

The third part does not currently have an implementation, however, this is where the ELS would communicate with the external main system to add error recovery. As discussed in Section 4.6, this part could be implemented in several different ways. No final choice has been made for this, but new states can easily be added here to add the functionality. It will also be possible to add different states for different methods and enable and disable certain states based on the settings of the ELS, depending on whatever the external system requires.

The last part of the analysis process contains the states to write the error data for export to the ”uart_message” unit, so it can be transmitted. This part will also finish up the analysis by resetting all the analysis states, signals and variables. It will also send a reset signal to the error bank to make sure that the record will be removed from the error bank or at least made invalid, to prevent the analysis from reading the same error record again. Once this is done, the analysis unit will repeat the entire process and search for a new valid error record.

While system feedback is not yet implemented, some options have already been discussed in Section 4.6. The feedback part of the analysis can be easily added as a new state in this analysis unit. All the different types of feedback processes for various external system types could also get their own state to make the system more general and modular again.

5.1.5 External connection

An early version of the error logging system used an online example UART for the export of error record data. This was just to test how the ELS would work with the

UART and how it would influence the behaviour. It also helped to form the necessary framework for any UART connection inside the ELS. It was useful to determine how the signals should travel through the hierarchy and what would be necessary to implement any type of UART to the system. In a later stage of the development, this UART was replaced by a different UART that would be used by the UT radiation beam experiment. Since the framework for an UART implementation was already in place, it was easy to switch them. However, there were some slight modifications necessary in the analysis, due to differences in the used UART protocol, as discussed in Section 4.7.2.

Another important implementation detail that needs to be mentioned is that the provided experiment UART was using bits instead of bytes for the length at the time of the implementation. The result is that the implementation currently deviates from the theoretical description, but this first needs to be changed in the experiment UART before it can be changed in the ELS. This error in the UART was discovered during the implementation and was resolved by providing the message length in bits. With this change, the implementation does work as expected, but it cannot reach its theoretical maximum amount of bytes and data size for one message. Since it does not seem to be necessary to reach the maximum at the moment, it is not important to focus on this problem and once the UART is updated, it should be fairly easy to update the analysis to give the message length in bytes instead of bits.

5.1.6 General remarks

Due to the many changes caused by implementation problems, some of the names and hierarchy choices are no longer as logical as they were during the initial implementation. One example is that the ”reri_lite_write_unit” is part of ”reri_lite_bank_top”, but it is not really part of the error bank. This was caused by a work-around regarding the TL-UL bus, that was used to generate the registers. Although this does not necessarily cause problems for the functionality or efficiency of the system, it can make it more complicated when trying to understand the ELS. Issues like this do currently not have priority, but it would be good to try and fix some of the name and hierarchy issues in future versions.

5.1.7 Implementation cost

The design was synthesized and implemented in Vivado. An implementation run was executed for an Arty A7-35T board. The results of this implementation run are shown in Tables 5.1, 5.2 and 5.3. It shows the usage of slice logic, memory DSP blocks and Input/Output (IO) blocks. As can be seen, the system does not use DSP blocks and two IO connections are used. This is as expected, since the ELS is not processing any signals and the two IO pads are used to transmit and receive UART messages. The component and memory usage of the ELS will be evaluated in Chapter 6.

Logic type		**Amount of RERI-Lite records**				
		1	**4**	**8**	**16**	**32**
Slice LUTS	Total	216	451	1929	3468	7226
	As logic	215	447	1921	3452	7194
	As memory	1	4	8	16	32
Slice registers	Total	254	534	2041	2638	6769
	As flip flop	248	528	2035	2632	6763
	As latch	6	6	6	6	6
Muxes	F7	0	0	50	20	122
	F8	0	0	8	10	8

Table 5.1: A table with an overview of the slice logic usage for the implementation of the error logging system with various amounts of RERI-Lite records. The LUTs implemented as memory are all shift registers.

Records	**RERI**			**RERI-Lite**		
Amount	**Bytes**	**RAMB36**	**Utilised**	**Bytes**	**RAMB36**	**Utilised**
1	64	4	8%	16	1	2%
4	256	16	32%	64	4	8%
8	512	32	64%	128	8	16%
16	1024	-	-	256	16	32%
32	2048	-	-	512	32	64%
64	4096	-	-	1024	-	-

Table 5.2: A table with the (expected) amount of used RAMB36 blocks for the standard RERI and RERI-Lite implementations on an ARTY A7. The required RAMB36 resources of the standard RERI implementation are estimated.

Site type		Used
DSP		0
Bonded IOB	total	2
	Master pads	1
	Slave pads	1

Table 5.3: A table with an overview of the DSP and IO usage for the implementation of the ELS.

5.2 Simulation and validation

This section will explain how the implementation was checked and validated. The simulations were mostly executed in the Vivado simulator, but some additional manual checks were performed in Questasim and Libero.

The simulation and validation were done in multiple ways. The newly written code was almost always simulated and checked manually at first and systematically checked later. This was mostly due to the fact that the system functionalities, components and even more importantly the error formats were regularly changed and this often required several major changes to the test-bench. For example, switching from the generated registers to a custom-made register-bank changed a lot of the names and hierarchies of the register-bank files and the new bank used a completely different bus with a different protocol. Adjusting the original test file needed a lot of time and it was important to quickly determine if the switch would solve the critical Vivado errors, so a quick manual check on the simulation was performed instead. At a later point, when the ELS was less likely to need major changes, the tests were updated to work with the new code to make sure that all aspects of the system were working properly.

The testing unit for Vivado and the tests that it contains will first be explained. The testing unit is a VHDL file that contains multiple processes. These processes will drive all the signals that are expected to be generated by any external system. The test creates a clock signal and a clock cycle counter. Depending on the error types that are enabled, it will at certain preset clock cycles simulate the error signals of different error types. This triggers the creation of error records and the analysis process. Error types are generated in different processes, so any error type can be easily disabled or new ones can be added. There is also a process that will monitor the UART port to reconstruct the UART messages and see if the ELS was able to generate the messages as expected and the amount of correctly received messages.

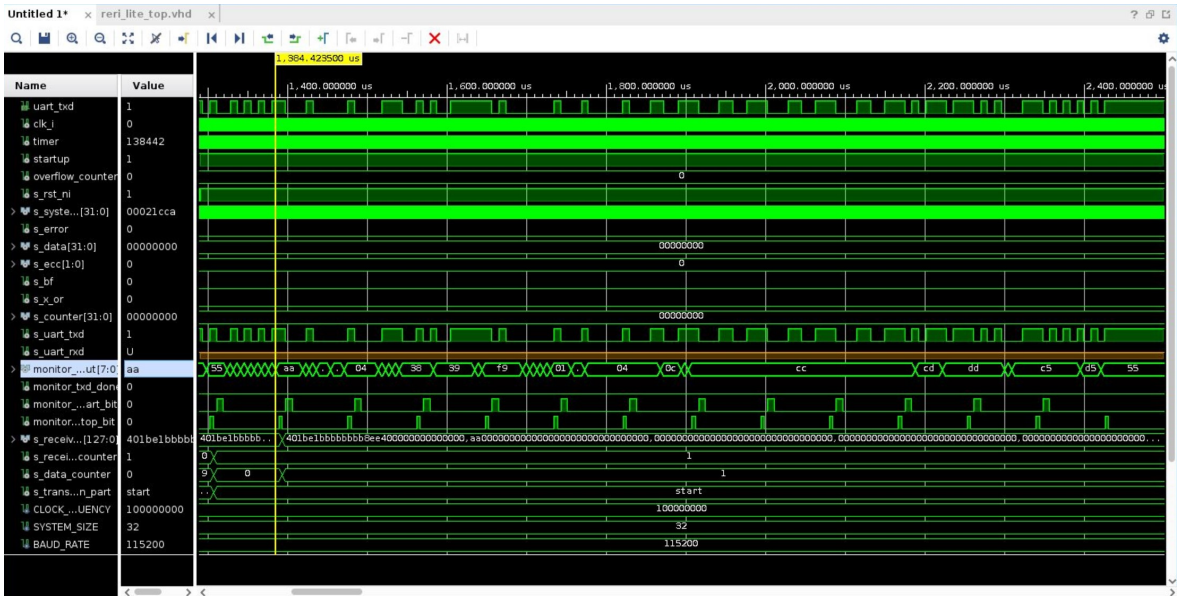

Figure 5.2: A screenshot showing the simulation of important signals from the test unit in Vivado.

Figure 5.2 shows a screenshot from the Vivado signal simulation of this test. The monitor signal that is selected shows the hexadecimal data that was detected in the UART transmission signal. The yellow marker shows the start of one of the error transmissions over the UART. The byte before this point was ”55”, which is the stop byte of an error message. The first received byte of data after the yellow marker is the header byte of ”aa”, followed by all the data inside the error record, including four transmissions of the ”cc” byte, which is data of one of the fixed simulated error signals. At the end of the message, the stop byte of ”55” is visible again.

Multiple error signals were generated to check whether the system was able to handle the following situations and corner cases:

- Trigger the reset signal and check if the error records and states are correctly returned to the default settings.
- Use multiple detection units of the same error type.
- Generate two error signals at the same detection unit at different points in time.
- Generate two error signals at the same detection unit in two consecutive clock cycles.
- Generate a new error signal after the record reset of a processed error to check if a reset record will be reused correctly.

- Generate two or more error signals at different detection units in the same clock cycle to test the write unit buffer.
- Generate more error signals than available error records before the records can be handled to check the overwrite mechanic of the error records.
- Generate more error signals at the same clock cycle to test a write unit buffer overflow.

Another version of the test unit was made that was synthesizable. This was done by replacing the clock signal and removing the UART port monitor, since the UART would be monitored by an external device. Also, only one process with some basic error signals was used to make sure that the system worked. So, no complicated error generation was done in the synthesized version of the test. The UART data would be received and stored in ASCII characters. This data had to be converted from ASCII characters to hexadecimal format first in order to check it. This check was done manually, so the simpler test made it easier to keep track of the UART output and to see if the received UART data were as expected.

The simple synthesizable test would just reset the ELS and trigger a total of four errors from type 3 and 4 at three different clock cycles. It also gave every single error a specific cycle counter or data value, so the specific error would be easy to identify in the resulting log file. The expected error messages should contain ”BBBBBBBB”, ”CCCCCCCC”, ”DDDDDDDD” and ”EEEEEEEE”.

5.3 Case study implementation

This section will describe the various test setups. Two main test setups were created on two different FPGA boards.

5.3.1 Arty A7

The first setup was a configuration on the Digilent Arty A7-35T variant [31]. Since Digilent retired this product, the board files were no longer available in Vivado. This meant that these files had to be manually added to the Vivado setup for the generation of the bitstreams. The system configuration used the synthesized version of the test script to generate errors. The bitstreams were uploaded to the board using the

Vivado Lab 2024.1 edition.

The setup used a Pmod USB UART [32] to be able to connect an external device to the board. On the external device, the SSH and telnet client PuTTY [33] was used to setup a connection and log the received UART data to a log file. This file could afterwards be read in Visual Studio Code (VSC) and converted to a hexadecimal format by using the extension ”Hex Editor” from Microsoft [34].

5.3.2 SmartFusion 2

The second test setup used the SmartFusion 2 (SF2) board. This is the same board that is used in the UT radiation beam experiment. The setup is very similar to the Arty setup from Section 5.3.1. Besides some minor settings differences, like the different clock speed on the SF2 board, the main difference was that the synthesis of this setup had to be done in Libero from Microchip [35]. The method of programming the board was also different. Libero would not create the bitstream for the board, but create a job file that could be uploaded to the SF2 by using the FlashPro Express program [36].

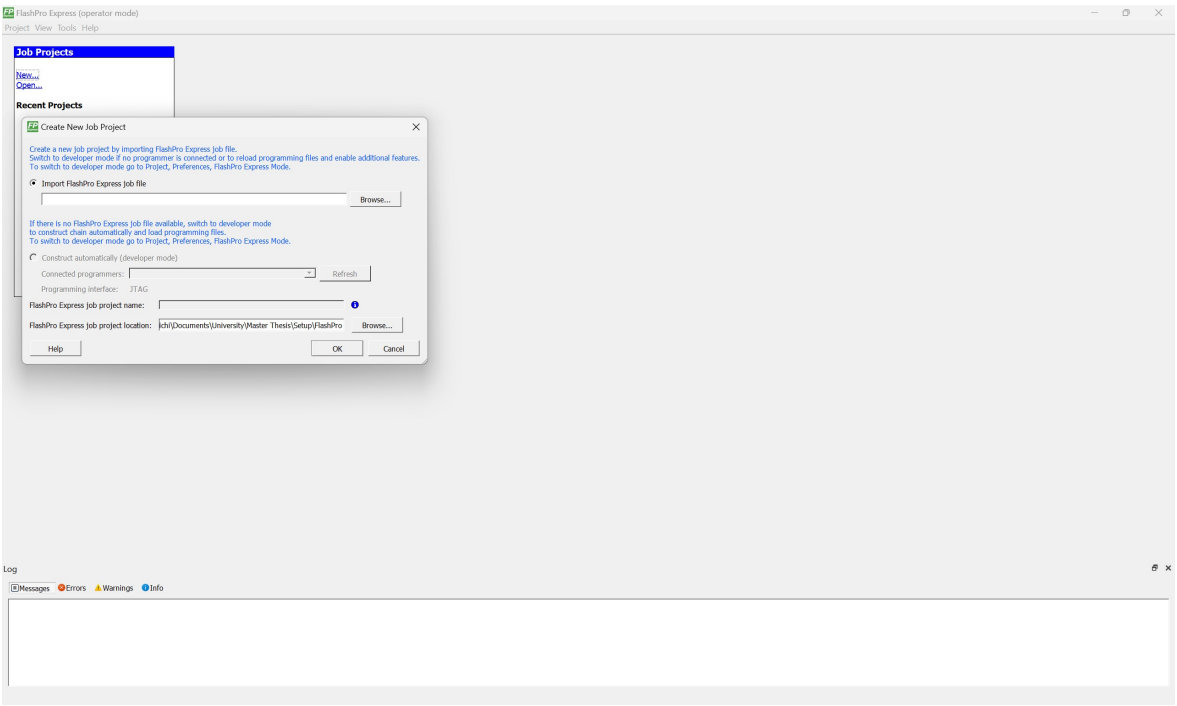

Figure 5.3: A screenshot of the FlashPro Express program [36].

This SF2 setup is targeted to be used for the UT radiation beam experiment,

but in that setup, the synthesised test unit was replaced by more complex external systems. In the experiment setup, the ELS was supposed to be connected to a NEORV32 SoC [14], the ”Sparrow” setup and the ”PDS” setup.

5.4 Experimental results

The PuTTY connection was setup and as soon as the bitstream was uploaded, the UART connection started to receive data. The converted PuTTY log file is shown in Figure 5.4.

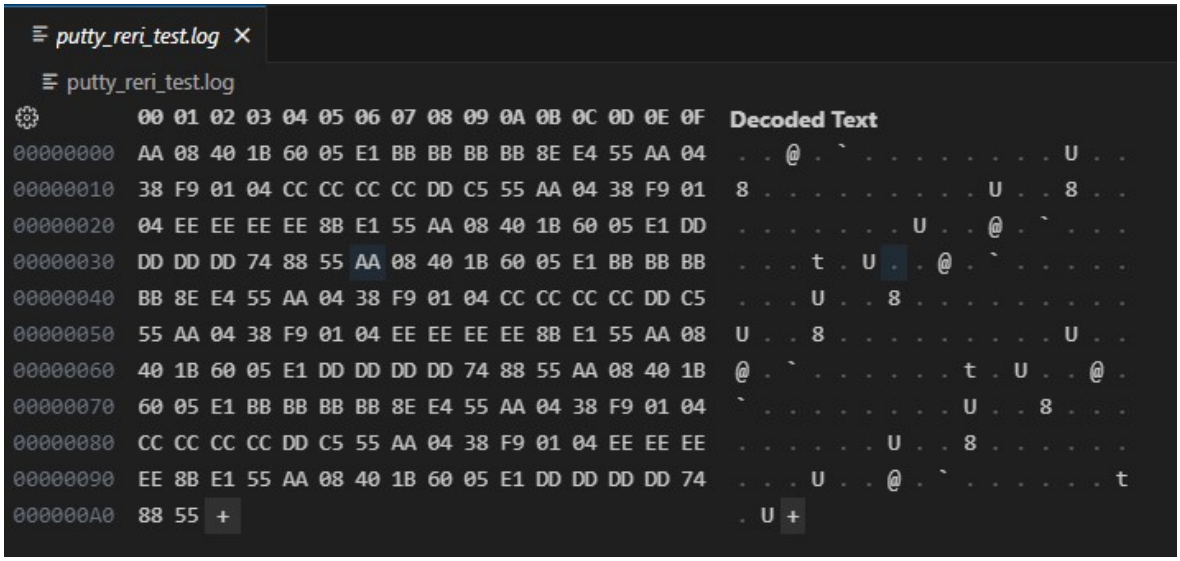

Figure 5.4: A screenshot of the PuTTY log file of the Arty setup test converted to a hexadecimal format.

As can be seen in the log file, the expected results with ”BBBBBBBB”, ”CCCCCCCC”, ”DDDDDDDD” and ”EEEEEEEE” (see Section 5.2) can be clearly found in the results. The messages contain all the expected frames. They start with the ”AA” data, which is the header frame, followed by either ”04” or ”08”, which is the assigned Component ID for error types 3 and 4. The next length frame is either ”38” or ”40”, because the two error types have a different amount of data that needs to be transmitted (see Section 4.4.3). The frames after that represent the data in the information field, followed by the specified ”BBBBBBBB”, ”CCCCCCCC”, ”DDDDDDDD” and ”EEEEEEEE” values in the additional error info register or timestamp register. The messages are concluded by two frames with the CRC data of the message and a stop frame that is always ”55”. Finally, a new message starts with ”AA”. As can be seen, the messages will repeat after a while. This is also as expected, since the

clock cycle counter in the synthesizable test (which triggers the error signals) will overflow after a while and that causes the test process to start over.

Chapter 6

System evaluation

This chapter will evaluate how the final ELS design that was implemented in Section 5 performs. The main aspects that will be looked at are the system timing, memory usage and power usage. It will also compare the implementation to a standard RERI based system.

6.1 Timing

It is important to quickly handle any errors in the system. The longer an error remains unresolved, the more damage it could cause to the system since the error can propagate. The ELS should be able to resolve detected errors as soon as possible. That is why it is important to analyse the amount of necessary clock cycles for the critical processes in the ELS.

The entire process of the ELS can currently be split in four main categories: ”monitoring”, ”controlling”, ”analysing” and ”UART communication”. These important categories can have several sub-processes and the required time for these processes can depend on the error type and other system variables. An overview of required clock cycles for the various processes is shown in Table 6.1. The clock cycle data was gathered in a Vivado simulation. Some simulation screenshots of the processes can be found in Figures A.3, A.4, A.5, A.6, A.7 and A.8 in the Appendix. To be able to evaluate the RERI-Lite format, this overview also shows the expected amount of clock cycles that a standard RERI system would require.

”Monitoring” is the time it takes from the detection of an error signal until the error information is collected and sent to the error bank. If multiple errors are detected within the same clock cycle, the time it takes the system to send the information to

Process	Subfunction	Clock cycles	
		RERI	RERI-Lite
Monitoring errors	Detect error signals	3 + N	3 + N
Control error record	Write valid record	33.5	9.5
	Reset record	33.5	9.5
Analyse errors	Read error record	149 - 150	14 - 42
	Check error records	3.5	3.5
	One bus transaction	9	9
UART communication	Create UART transmission	1	1
	Transmit UART frame	8681	8681
	Transmit full error record	607639	86806 - 190972

Table 6.1: A table with the (expected) amount of clock cycles used to execute parts of the error logging operations. N represents the number of additional errors that were detected in the same clock cyle, since every additional error delays the timing by one clock cycle. The UART communication clock cycles assumes a 115200 baudrate and a 100 MHz clock frequency.

the error bank will increase. This is caused by the fact that it is not possible to write to multiple error records of one error bank in the same clock cycle. So, for every additional error the required time will be increased by one clock cycle.

”Controlling” consists of the processes that change the information within the error records. Currently, there are two possible processes that can change the information. One is to write the information of a new error in an error record and the other is to reset the information within a record to the default values.

”Analysing” contains the processes that the analysis unit needs to obtain and interpret the information within the error records. The analysis unit needs to check for valid error records and it needs to read the entire record. Bus transactions are an important part of the error record reading process.

”UART transmission” entails the creation of a UART message and the time it takes to fully transmit the entire message with the information about a single error. The required amount of time and clock cycles will depend on the baud rate that the UART uses, but this is not important if RERI and RERI-Lite are expected to use the same UART settings.

As can be seen in the overview of Table 6.1, the RERI-Lite format is expected to perform much better than the standard RERI format in processes that require in-

teraction with the error records. Since the error record format is only a fourth of the standard RERI format, it requires less write and read bus operations to complete the important processes. Additionally, the RERI-Lite format is more flexible, so the system can ignore the unused parts of the error records. This can save a lot of unnecessary read operations and bus transactions. Reading an entire RERI record (assuming the use of a 32-bit width bus) will require 149 - 150 clock cycles to complete. In contrast, the RERI-Lite version will use a maximum of 42 clock cycles and the amount can even be lowered to 14 clock cycles if the error type uses only one of the registers.

The same concept applies to the UART communication process. The standard RERI format will always have to transmit the information of the entire record, while the RERI-Lite format can ignore the unused parts and lower the required amount of UART frames. The reduced error record size is not only useful for faster error transmissions, but its reduced size can also prevent the loss of error data. As discussed in Section 4.7.3, the UART bandwidth is limited. In systems where a low baud rate is used, where the ELS does not have access to the entire UART bandwidth or where an extremely high number of errors per minute is expected, the large standard RERI records can quickly overwhelm the UART connection and cause important error information to be lost. The RERI-Lite format will require a lot less UART frames on average, so the number of RERI-Lite errors that can be handled by the UART connection will always be much higher.

6.2 Memory usage

Another aspect of the RERI-Lite format to evaluate is the memory usage. Table 5.2 shows a comparison between the required memory usage of the implemented RERI-Lite format and the expected memory usage of the standard RERI format for various amounts of error records.

As can be seen in Table 5.2, the RERI-Lite format scales linearly and the format is expected to use only a fourth of the memory required by the standard RERI format. The result is that the RERI-Lite format can have four times the amount of standard RERI records with the same available memory. As can be seen, the ARTY A7 was able to implement 32 RERI-Lite error records, while the standard RERI format would only be able to implement 8 records. So systems that implement the RERI-Lite format can have a larger error buffer.

An increase in the number of error records allows the system to better handle peaks in the number of detected errors. It allows the system to save error records longer and decreases the chance of needing to overwrite unhandled errors or ignoring new errors. This gives the system more time to handle an error. Especially in systems where a large fluctuation in the amount of errors is expected, it will be useful to create a large error buffer. So in systems with limited available memory for an ELS, the RERI-Lite format can be used to increase the buffer size of the error logging system, without the need for more memory.

Other advantages of reducing memory usage are the decrease in area and overhead. The error data needs to be protected, so the memory uses ECC to detect and correct errors, which increases the area and overhead of the system. Also, the larger the area used, the higher the chance that errors will occur, which will add more overhead to deal with the errors and it could lower the reliability of the system. Finally, as the system gets larger and the area increases, the time it takes signals to reach their destination will increase too. This can slow the system down and reduces the performance of the ELS. It is important for any application to find a good balance between the amount of error records and the area that is used by the system. The lower memory usage of the RERI-Lite format makes it ideal for a better trade-off between the used area and performance of the system.

To conclude, the RERI-Lite format will be much faster in handling errors and the format will reduce the chances of error propagation or information loss. These aspects are critical for an ELS, which will likely make systems with the RERI-Lite format perform better, compared to systems with the standard RERI format.

6.3 Power

A power estimation was performed for the RERI-Lite implementation on an ARTY A7. Since no specific environment is known for the applications with error logging, the default Vivado settings were used to get an indication of the power usage of the ELS. The settings used for the estimation are shown in Table 6.2.

The result of the Vivado power estimation tool is shown in Table 6.3. The table shows the power estimations of various amounts of RERI-Lite error records. It was not possible to do an accurate power estimation of a comparable standard RERI based system, without having to fully implement it, so it is not possible to evaluate how the RERI-Lite records would perform compared to the standard RERI records.

Ambient Temp (C)	25
ThetaJA (C/W)	4.8
Airflow (LFM)	250
Heat Sink	Medium
ThetaSA (C/W)	4.6
Board Selection	Medium
Number of Board Layers	12 to 15
Board Temperature (C)	25

Table 6.2: A table with the settings for the Vivado power estimation.

Number of error records	**1**	**4**	**8**	**16**	**32**
Total On-Chip Power (W)	0.591	1.158	2.68	4.611	9.312
Dynamic (W)	0.532	1.098	2.617	4.542	9.217
Device Static (W)	0.059	0.06	0.063	0.069	0.095
Effective TJA (W)	4.8	4.8	4.8	4.8	4.8
Max Ambient (C)	97.2	94.5	87.2	78	55.5
Junction Temperature (C)	27.8	30.5	37.8	47	69.5
Confidence Level	Low	Low	Low	Low	Low

Table 6.3: A table with the Vivado power estimation summary for various amounts of RERI-Lite error records.

The estimated power and temperature data of Table 6.3 have been plotted in Figures 6.1 and 6.2. As can be seen, most of the power and temperature estimations for the system seem to be scaling linearly with the increase in error records. The data could only be gathered up to 32 error records, since the ARTY A7 was unable to fit more records. So, more data points might be necessary to determine if the power actually scales linearly at high amounts of error records.

Furthermore, no matter how the component and power usage scales, the RERI-Lite format will always scale better than the standard RERI-format. The results of the RERI-Lite component and power usage also give an indication of how well the standard RERI format would perform. Table 6.4 shows the number of on-chip components that are used by the ELS. The standard RERI format will require 4 times the memory of the RERI-Lite format. It is expected that the component and power usage of an ELS with standard RERI records will be comparable to 4 times as many RERI-Lite records. For example, when implementing a system with 4 or 8 standard RERI records, the system component and power usage would be comparable to

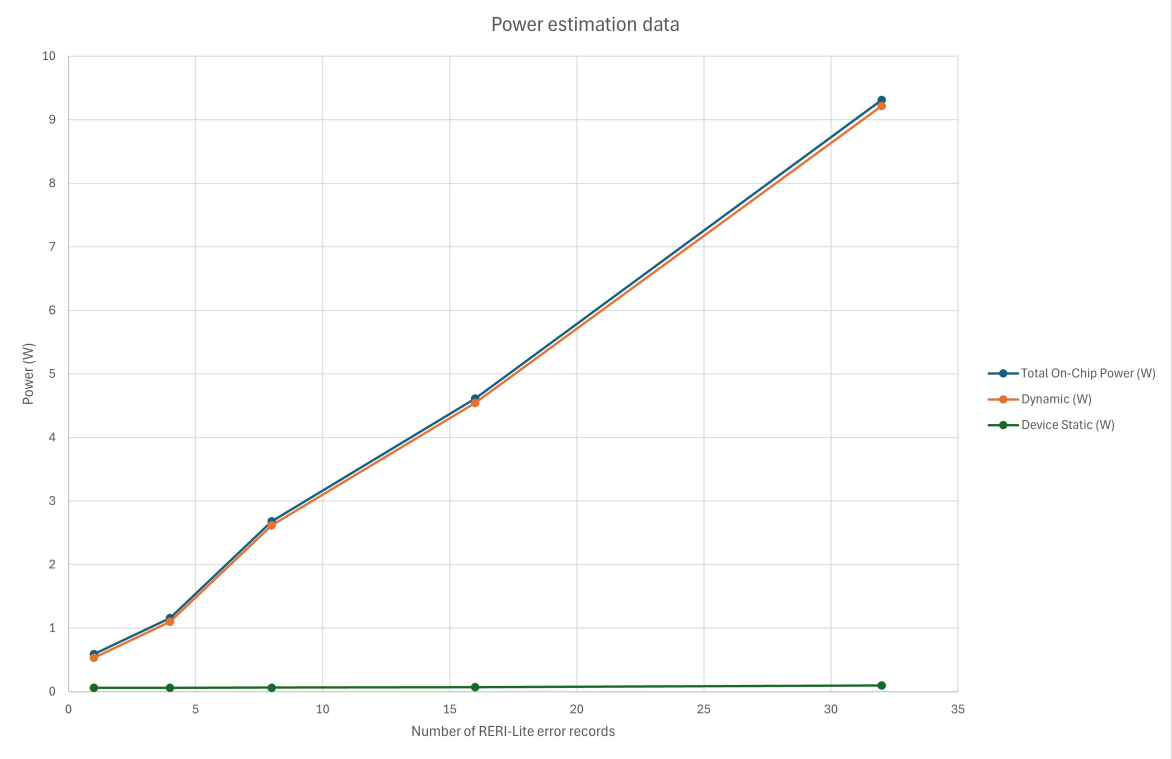

Figure 6.1: A plot of the estimated power data from Table 6.3.

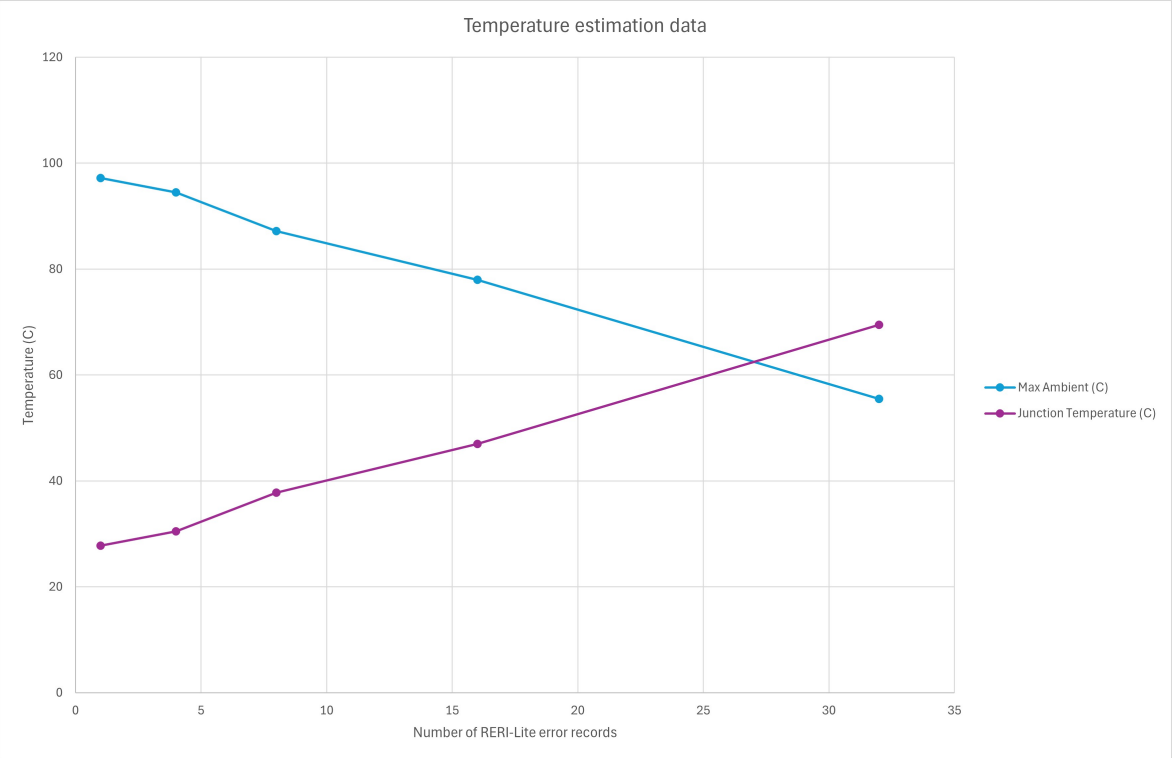

Figure 6.2: A plot of the estimated temperature data from Table 6.3.

implementing 16 or 32 RERI-Lite records. This means that the RERI-Lite format is likely to scale 4 times better than the standard RERI format if the power usage does scale linearly.

On-chip components		**Amount of RERI-Lite records**				
		1	**4**	**8**	**16**	**32**
Slice logic	Total	592	1175	4130	6866	15936
	LUT as logic	215	447	1919	3452	7194
	CARRY4	20	20	44	68	372
	Register	254	534	1627	2638	6769
	BUFG	1	1	1	1	1
	Others	27	48	70	117	246
	F7/F8 muxes	0	0	58	30	130
	LUT as shift register	1	4	8	16	32
Signals		420	812	3232	5554	13200
Block		1	4	8	16	32
I/O		2	2	2	2	2

Table 6.4: An overview of the used on-chip components in the error logging system that are relevant for the power usage of the system.

Chapter 7

Discussion

This chapter will go over the most important achievements and problems that were encountered. It will also discuss how the encountered problems influenced the decisions and results in this thesis.

The experiment on the ARTY was able to give a proof of concept for the design of the ELS and it showed that all the components can work together properly. This ELS does not only specify a format for error data in RISC-V systems, but also creates a framework around it. It is able to handle the data from the error detection units by storing the data, by analysing it and by exporting it. The system can also be expanded in the future to allow for error correction feedback to the external system where the error occurred.

The new RERI-Lite format that was used in the ELS also reduced the required resources and added more flexibility. The system does not use dedicated error records for detection units, so the amount of records can be based on the available space. Also, every error record is at maximum a fourth of the full standard RERI error record size. This does not only allow for more error records in general, but it also lowers the amount of required bus transactions to retrieve the error data. This will speed up the error handling as well. The record format was also created with both 32-bit and 64-bit systems in mind. The format focuses on 32-bit systems, but as explained in Section 4.8, it can be quickly adjusted to a 64-bit format. The RERI-Lite records were also made customizable, so that new or adjusted detection methods with other error data can easily create their own record to store the necessary data in a convenient way.

The ability to lower the amount of necessary resources, reduce overhead and customize error record types for new hardware errors shows the potential for such a new RERI-Lite based ELS. Since the system was made separately from the RISC-

V-based systems that it was designed for, it will be easy to add, modify or remove the ELS to or from other systems. All these advantages could make this system very useful for very specific or small applications.

The ELS still has a lot of possible improvements left in general. However, most of these were already expected to be left for future development, as discussed in Section 4.8. Adding these features in the future can make the ELS even more customizable and useful for specific applications.

Still, some other problems were encountered and added to the list of improvements during the implementation of the system. The most important complication during the thesis was the unexpected behaviour of the ELS on the SF2. It is at this moment still unknown why the ELS does not work on the SF2 board. As explained in Section 5.4, the messages received over the UART connection could not be explained. The problem probably lies in the integration with the SF2 specifically. It is not likely that the problem is caused by a major oversight in the system design or implementation itself, since the results on the ARTY setup were exactly as expected.

Due to the problems with integrating the ELS on the SF2, it was not possible to finish the integration of the ELS into the beam experiment setup. So, for future work, it would be good to solve the SF2 integration problems. The system can then be tested under the radiation beam as well. This will be useful to test if the system is also able to operate and execute its main task in practise in an environment similar to one where it is supposed to operate. It will specifically be useful to see if the ELS will be overwhelmed by the amount of radiation-induced errors and how reliable the system itself will be under radiation conditions.

Another problem that occurred during the development of the ELS were the simulation issues that would be caused by the error bank register generation, as mentioned in Section 5.1.3. A lot of time and work was spent on figuring out the generation of the registers, the use of the bus and the creation of a combined VHDL and Verilog system. Especially since the errors could not be resolved in a reasonable time, it was necessary to switch to the custom creation of the error bank registers and a new bus. It also required a lot of additional time to adjust all the other parts of the ELS to the changes that this switch brought along as well. This valuable time could have been spent fixing the SF2 integration problems instead.

Chapter 8

Conclusion and future work

This chapter will try and answer the research goals and questions that were given in Section 1.1. This section will answer the main research question and the goal of this thesis. It will also look at the individual sub-goals that were set for this thesis. Finally, it will give some recommendations on possible areas to further improve or research in the future to continue the development of the error logging system.

8.1 The main thesis goal

The main goal of this thesis was: *"Design and implement a system that can systematically detect, log, analyse and resolve hardware errors in a RISC-V SoC."*

By answering the smaller research questions in Chapter 4, a final design for such an ELS was given in Section 4.2. This design was implemented to achieve the main objective of this thesis. However, one part of this goal, the "error resolving" part of the system, has not been fully designed and implemented yet.

During the design and implementation phase of the ELS, the focus of this thesis was to obtain a working prototype of the ELS that could be used and tested in a UT radiation beam experiment. While some important aspects for resolving errors were considered, providing feedback to the system was mostly ignored in the design and implementation to save time and to try and get a basic working version. That is why the important design decisions for the feedback part of the system have not yet been made.

The new system was tested on an ARTY and is confirmed to work as expected under normal conditions. This means that the detection, logging and analysis aspects of the research goal were reached. So, the most important part of the goal,

the implementation and testing of a general ELS was reached. As explained in Section 5.4, due to the limited time of this thesis, it was not yet possible to use the ELS in a working setup under the radiation beam. Performing such an experiment will be left to future work.

Furthermore, this thesis proposed a new version of the RISC-V RERI specification that is smaller and more flexible. As shown in Chapter 6, this format is expected to perform better in the important aspects of speed, memory and area usage. Although reduced memory and area usage are mostly important for system applications that have little resources available, the increase in speed will be important for all ELS. The system will require less clock cycles to gather and analyse information about new errors with the RERI-Lite format. This will increase the response time to errors and that can prevent errors from propagating further and causing new problems in the system. This makes this RERI-Lite format specifically useful for embedded systems.

In conclusion, many of the objectives for this thesis were achieved. This work created a core design for an ELS that uses the newly proposed RERI-Lite error record format. Although there are still some important aspects and improvements left, it forms a good basis for future development of this system. Section 8.3 will go into a little more detail on the most important recommendations for the future development of this system.

8.2 The sub research questions

The next section will cover all the different aspects of this thesis in a little more detail. This section will go over the answers and solutions that were found for the individual research sub-questions and sub-goals of this thesis.

8.2.1 Error taxonomy and detection

Many different hardware error types exist, which can all cause a wide range of problems in any system. This is why the first sub-question that was looked into was: "*What types of hardware errors exist and how can they be detected?*"

An overview of errors and possible causes was given in Section 2.1. Some errors remain undetected and can cause an Silent Data Corruption (SDC). The errors that

are detectable can be split into two groups. The Detected Uncorrectable Error (DUE) and the Corrected Error (CE). The severity of any error will depend on the location of the errors and whether the error can be detected and corrected. A simple bit flip can simply change unused data and have no impact, but it could also change important system settings or values and cause a complete breakdown of the system.

As mentioned in 2.3, there is also a wide range of methods and techniques to detect these errors and to reduce the effects they have on systems. Most of these techniques work by adding some sort of redundancy to the system. This can be done in the form of redundant data, such as one or more parity bits to data values or by adding redundancy to entire components of a system. This redundancy can also be added in space (by duplicating the hardware of the system) or in time (by executing the process multiple times). A simple check or comparison can then usually detect any errors and methods that use more resources for this redundancy can sometimes use a voting system to immediately choose the correct result.

Unfortunately, all of these detection and correction methods have trade-offs. The more reliable they make a system, the more redundancy they usually need, which will require more resources and it can also slow systems down in some cases. So, it is necessary to find a good balance between the reliability of the system and the performance of the system.

8.2.2 Error information

For an ELS, it is important to gather as much relevant error data as possible. So, the next sub-question that was answered was: "*What type of information about the errors is generally available and is it usable for error analysis?*"

As described in Section 2.4, it was difficult to determine what information would generally be available about all detectable error types. However, there was some information about errors that was often available or could be obtained in other ways. The conclusion was that the most important data that were usually available contained information about the type of error (such as an error code), some sort of location (such as an address or system component), some sort of time reference and if the error has already been corrected or not.

Although it is sometimes possible to retrieve more information about the errors, additional information does not seem to be available in general cases. This meant

that it was not very efficient to dedicate specific register fields to this type of information inside the error records, but a more flexible option like the customizable register in the new error record format was a better solution. This still allows for the use and logging of additional data without sacrificing the efficiency of the error records too much.

8.2.3 Error information storage

The storage of the error information was also a major part of this thesis. The sub-research question "*What type of information about the errors is generally available and is it usable for error analysis?*" covered this aspect of the error logging system.

As discussed in Sections 2.4 and 4.3, the information that is available about errors differs for every error type and detection method. However, as concluded in 8.2.2, it would usually be possible to determine the following information about any error: what the error type or code is, a component ID, some sort of priority or importance, the address where the error is located and a timestamp. Most of the other data on errors will be completely dependent on the error detection unit. So, the ELS focused on an error record format that had entries for the most frequent data, with room for customization for additional information about these specific errors.

Some of this information can only be determined during the integration of the error detection units and the ELS, such as a unique component ID. This will require some manual integration and settings for error detection units to be able to provide the correct or full error information.

As described in Sections 4.4 and 5.1.3, an adjusted error bank format was proposed, based on the RISC-V RERI Architecture Specification format, with multiple customizable error records. Its format is highly customizable and focused on systems that have limited error information or resources available, to make this ELS compatible and efficient for more and smaller scale applications, such as embedded systems.

In Chapter 6, it was concluded that the newly proposed RERI-Lite format is expected to perform better than the standard RERI format when it comes to speed, memory and area usage. Especially the timing aspect is of importance, since an ELS should be able to quickly access and use the available error data. As the gathered data indicated, the amount of necessary clock cycles for an error record read,

write or transmission are expected to be at maximum a fourth for the new RERI-Lite format. Due to the flexibility of the records, it can be even lower if an error record does not use the entire record. This can significantly improve the performance of ELSs.

8.2.4 Error analysis

The next sub-question was: "*How can the error information be analysed in a general way?*" The goal of this question was to give the ELS additional uses, besides just logging errors. Analysing the error data allows for error logging features such as providing error feedback to the core of a system.

As discussed in Sections 4.5 and 5.1.4, the analysis can be made as complex as desired by the application in which the ELS is used. However, a general state machine structure was created, such that it can be expanded with more complexity in further developed versions. By using a state machine, it will be easy to add or switch between various parts and options of the analysis process. Systems that do not have compatibility with some of the more complex analysis features can skip these states, based on the settings. This will provide a general way to analyse the errors written in the error bank that allows for much customization in future versions, especially when new features like error priorities are fully implemented.

8.2.5 Handling the errors

The final sub-question that was necessary to create a complete error logging system was: "*Can errors be handled in a general way?*"

Technically, the current design is able to analyse and export the data to an external system for later analysis. So, in that sense the error is handled in a general way. However, this question can be considered as a two-part problem, since the system has two different options to handle an error record. One option is to just export the data to an external device, while the other option is to try and resolve any problems caused by that error. The main objective of this sub-question was to find a general method for both these options. Mainly in Section 4.6, some possibilities were discussed, but no final decision or design was implemented or tested. Since error recovery is very dependent on the external system implementation, it is very hard and complicated to resolve errors in the external system. However, the problem

could likely be solved, but it would require more time and effort to design a general method.

So, it can be concluded that this sub-question has not been fully answered yet, but this aspect of the question was mostly ignored due to a lack of available time. It would be best to look further into this aspect and try to find a more satisfying solution to it.

8.3 Future work

A lot was achieved with the implementation of the ELS, however, there is always something that can be added or improved. Section 4.8 mentioned a number of possible expansions and current shortcomings of this system. Examples are adding scrubbing or a faster way of presenting the most important error record information.

Another possible improvement would be the addition of a working feedback loop to correct errors. Due to time issues and the focus on trying to get a working beam experiment version, this feedback loop was not implemented in the test version. However, adding this feedback loop would significantly improve the use of this ELS. Right now, the main function of the ELS is to just log the error data, but adding the feedback will allow the analysis unit to have more functionality and that could actually improve the RAS of systems to which the ELS is added.

Furthermore, an improvement that will also be easy and important to implement is to add priorities to the errors. This will work well with the addition of the analysis feedback, since a priority system would most likely allow for a much more efficient analysis of any error records.

However, the final and most important recommendation would be to get the ELS compatible with the SF2 setup or a different setup that could be used in future UT radiation beam experiments. This will finally give a proof of concept with errors that are actually randomly generated by radiation instead of predetermined errors. This would make sure that the ELS can handle the randomness and amount of these radiation induced errors as well. It will also test how susceptible the ELS itself is to radiation-induced errors, which will be a very important aspect to investigate for the reliability of the ELS.

Bibliography

[1] "RISC-V RERI Architecture Specification," Sep. 2023. [Online]. Available: https://github.com/riscv-admin/ras-eri

[2] P. R. Nikiema, A. Palumbo, A. Aasma, L. Cassano, A. Kritikakou, A. Kulmala, J. Lukkarila, M. Ottavi, R. Psiakis, and M. Traiola, "Towards Dependable RISC-V Cores for Edge Computing Devices," in *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Jul. 2023, pp. 1–7, iSSN: 1942-9401. [Online]. Available: https://ieeexplore.ieee.org/document/10224862/references#references

[3] G. Furano, S. Di Mascio, A. Menicucci, and C. Monteleone, "A European Roadmap to Leverage RISC-V in Space Applications," in *2022 IEEE Aerospace Conference (AERO)*. Big Sky, MT, USA: IEEE, Mar. 2022, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/9843361/

[4] "ECSS-Q-HB-60-02A – Techniques for radiation effects mitigation in ASICs and FPGAs handbook (1 September 2016) | European Cooperation for Space Standardization." [Online]. Available: https://ecss.nl/hbstms/ecss-q-hb-60-02a-techniques-for-radiation-effects-mitigation-in-asics-and-fpgas-handboo

[5] H. Quinn, "Challenges in Testing Complex Systems," *IEEE Transactions on Nuclear Science*, vol. 61, no. 2, pp. 766–786, Apr. 2014, conference Name: IEEE Transactions on Nuclear Science. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6786369

[6] R. Velazco, P. Fouillat, and R. Reis, Eds., *Radiation Effects on Embedded Systems*. Dordrecht: Springer Netherlands, 2007. [Online]. Available: http://link.springer.com/10.1007/978-1-4020-5646-8

[7] V. Sridharan, D. A. Liberty, and D. R. Kaeli, "A Taxonomy to Enable Error Recovery and Correction in Software," in *Workshop on Quality-Aware Design*, 2008.

[8] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. John Wiley & Sons, Dec. 2020, google-Books-ID: XB0JEAAAQBAJ.

[9] *Error Correction Codes for Non-Volatile Memories*. Dordrecht: Springer Netherlands, 2008. [Online]. Available: http://link.springer.com/10.1007/978-1-4020-8391-4

[10] H. Zhou, "SECDED code and its extended applications in DRAM system," *Applied and Computational Engineering*, vol. 6, pp. 505–511, Jun. 2023. [Online]. Available: https://www.ewadirect.com/proceedings/ace/article/view/2177

[11] S. Mukherjee, J. Emer, T. Fossum, and S. Reinhardt, "Cache scrubbing in microprocessors: myth or necessity?" in *10th IEEE Pacific Rim International Symposium on Dependable Computing, 2004. Proceedings.*, Mar. 2004, pp. 37–42. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1276550

[12] Krištofík, M. Baláž, and P. Malík, "Hardware redundancy architecture based on reconfigurable logic blocks with persistent high reliability improvement," *Microelectronics Reliability*, vol. 86, pp. 38–53, Jul. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271418301847

[13] H.-T. Li, C.-Y. Chou, Y.-T. Hsieh, W.-C. Chu, and A.-Y. Wu, "Variation-Aware Reliable Many-Core System Design by Exploiting Inherent Core Redundancy," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2803–2816, Oct. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7959084/

[14] S. Nolting and A. T. A. Contributors, "The NEORV32 RISC-V Processor," Nov. 2024. [Online]. Available: https://github.com/stnolting/neorv32

[15] "Open source silicon root of trust (RoT) | OpenTitan." [Online]. Available: https://opentitan.org/

[16] "AMD64 Architecture Programmer's Manual, Volume 2: System Programming, 24593," Jun. 2023.

[17] "Intel® 64 and IA-32 Architectures Software Developer Manuals," Dec. 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[18] N. Canino, S. Di Matteo, D. Rossi, and S. Saponara, "HW-SW Interface Design and Implementation for Error Logging and Reporting for RAS Improvement," *IEEE Access*, vol. 12, pp. 60 081–60 094, 2024, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/document/10508585

[19] S. Thomet, S. De-Paoli, J.-M. Daveau, V. Bertin, F. Abouzeid, P. Roche, F. Ghaffari, and O. Romain, "FIRECAP: Fail-Reason Capturing hardware module for a RISC-V based System on a Chip," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct. 2021, pp. 1–6, iSSN: 2765-933X. [Online]. Available: https://ieeexplore.ieee.org/document/9568317

[20] S. Thomet, F. Ghaffari, S. De Paoli, J.-M. Daveau, F. Abouzeid, and O. Romain, "Observation framework of errors in microprocessors with machine learning location inference of radiation-induced faults," *Microelectronics Reliability*, vol. 137, p. 114667, Oct. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271422001913

[21] "Arm Architecture Reference Manual for A-profile architecture," Apr. 2023. [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest/

[22] M. Peña-Fernandez, A. Lindoso, L. Entrena, and M. Garcia-Valderas, "The Use of Microprocessor Trace Infrastructures for Radiation-Induced Fault Diagnosis," *IEEE Transactions on Nuclear Science*, vol. 67, no. 1, pp. 126–134, Jan. 2020, conference Name: IEEE Transactions on Nuclear Science. [Online]. Available: https://ieeexplore-ieee-org.ezproxy2.utwente.nl/document/8915730

[23] M. Peña-Fernández, A. Lindoso, L. Entrena, I. Lopes, and V. Pouget, "Microprocessor Error Diagnosis by Trace Monitoring Under Laser Testing," *IEEE Transactions on Nuclear Science*, vol. 68, no. 8, pp. 1651–1659, Aug. 2021, conference Name: IEEE Transactions on Nuclear Science. [Online]. Available: https://ieeexplore-ieee-org.ezproxy2.utwente.nl/document/9381897

[24] M. Möstl, A. Dörflinger, M. Albers, H. Michalik, and R. Ernst, "Self-Adaptation for Availability in CPU-FPGA Systems Under Soft Errors," in *2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Jul. 2019, pp. 9–16, iSSN: 2471-769X. [Online]. Available: https://ieeexplore-ieee-org.ezproxy2.utwente.nl/document/8792930

[25] H. Quinn, D. Roussel-Dupre, M. Caffrey, P. Graham, M. Wirthlin, K. Morgan, A. Salazar, T. Nelson, W. Howes, E. Johnson, J. Johnson, B. Pratt, N. Rollins, and J. Krone, "The Cibola Flight Experiment," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 1, pp. 3:1–3:22, 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2629556

[26] "UART Baud Rate and Output Rate," sBG Systems. [Online]. Available: https://support.sbg-systems.com/sc/kb/latest/technology-insights/uart-baud-rate-and-output-rate

[27] “Understanding UART,” rohde & Schwarz. [Online]. Available: https://www.rohde-schwarz.com/us/products/test-and-measurement/essentials-test-equipment/digital-oscilloscopes/understanding-uart_254524.html

[28] “DCS Group GitLab uart_message,” Sep. 2024. [Online]. Available: https://gitlab.utwente.nl/dcs-group/soc-fpga-setups/cores/-/tree/master/uart_message?ref_type=heads

[29] “DCS Group / SoC FPGA Setups / RERI-lite · GitLab,” Jul. 2024. [Online]. Available: https://gitlab.utwente.nl/dcs-group/soc-fpga-setups/reri-lite

[30] “reggen & regtool: Register Generator - OpenTitan Documentation.” [Online]. Available: https://opentitan.org/book/util/reggen/index.html#running-standalone-regtoolpy

[31] “Arty A7 - Digilent Reference.” [Online]. Available: https://digilent.com/reference/programmable-logic/arty-a7/start

[32] “Pmod USB UART - Digilent Reference.” [Online]. Available: https://digilent.com/reference/pmod/pmodusbuart/start

[33] “Download PuTTY - a free SSH and telnet client for Windows.” [Online]. Available: https://www.putty.org/

[34] “microsoft/vscode-hexeditor,” Nov. 2024, original-date: 2020-05-11T22:44:34Z. [Online]. Available: https://github.com/microsoft/vscode-hexeditor

[35] “Libero® IDE | Microchip Technology.” [Online]. Available: https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-ide

[36] “FlashPro Express | Microchip Technology.” [Online]. Available: https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/programming-and-debug/flashpro-express

Appendix A

First appendix

	Error report size (bytes):							Total byte	Total bits
Error type	Header	Length	ID	CRC	Stop	Fixed size	Data size	Per report	Per report
Sparrow	1	1	1	2	1	6	5	11	110
DSP	1	1	1	2	1	6	6	12	120
BUS	1	1	1	2	1	6	16	22	220
Average	1	1	1	2	1	6	9	15	150
Min	1	1	1	2	1	6	1	7	70
Max	1	1	1	2	1	6	16	22	220
Test	1	1	1	2	1	6	32	38	380

Frame size (bits)	
Start	1
Data	8
Parity	0
Stop	1
Total	**10**

Error rates and reports over tim			Requried data per error message (top in bytes below in total bits)						
			Sparrow	DSP	BUS	Average	Min	Max	Test
errors/mi	errors/se	reports	5	6	16	9	1	16	32
50	0.83333	1	91.6667	100	183.333	125	58.3333	183.333	316.667
100	1.66667	2	183.333	200	366.667	250	116.667	366.667	633.333
150	2.5	3	275	300	550	375	175	550	950
200	3.33333	4	366.667	400	733.333	500	233.333	733.333	1266.67
250	4.16667	5	458.333	500	916.667	625	291.667	916.667	1583.33
300	5	5	550	600	1100	750	350	1100	1900
350	5.83333	6	641.667	700	1283.33	875	408.333	1283.33	2216.67
400	6.66667	7	733.333	800	1466.67	1000	466.667	1466.67	2533.33

Output limits per sec				Check for the different error rates (in errors per minute)							
baudrate	Total bits	Data bits	Data bytes	*50*	*100*	*150*	*200*	*250*	*300*	*350*	*400*
9600	960	768	96	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
19200	1920	1536	192	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
38400	3840	3072	384	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
115200	11520	9216	1152	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
230400	23040	18432	2304	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
460800	46080	36864	4608	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

Figure A.1: A screenshot of the tables in the Excel tools to calculate the data output limits and required data output for various UART configurations and error rates.

2.7. Error Code Encodings

Table 6. Error code encodings

Encoding	Description
0	None
1	Other unspecified error occurred
2	Corrupted data access (e.g., attempt to consume poisoned data) error
3	Cache block data (e.g., ECC error on cache data) error
4	Cache scrubbing detected (e.g., ECC error on cache data) error
5	Cache address/control state (e.g., parity error tag or state) error
6	Cache unspecified error
7	Snoop-filter/directory address/control state (e.g., ECC error on tag or state) error
8	Snoop-filter/directory unspecified error
9	TLB/Page-walk cache data (e.g., ECC error on TLB data) error
10	TLB/Page-walk cache address/control state (e.g., ECC error on TLB tag) error
11	TLB/Page-walk cache unspecified error
12	Hart state error (e.g., ECC error on CSRs or x/f/v registers)
13	Interrupt controller state (e.g., ECC error on interrupt pending/enable state) error
14	Interconnect data (e.g., ECC error on data bus) error
15	Interconnect other (e.g., parity error on address bus) error
16	Internal watchdog error
17	Internal datapath, memory, or execution units error (e.g, ALU datapath parity)
18	System memory command/address bus error
19	System memory unspecified error
20	System memory data (e.g., ECC error in SDRAM or HBM) error
21	System Memory scrubbing detected error
22	Protocol Error - illegal input/output error
23	Protocol Error - illegal/unexpected state error
24	Protocol Error - timeout error
25	System internal controller (power management, security, etc.) error
26	Deferred error pass-through (e.g., forwarding poisoned data) not supported
27	PCIe/CXL detected (e.g., logged into PCIe AER, CXL.mem error log, etc.) errors
28 - 63	Reserved for future standard use
64 - 255	Designated for custom use

Figure A.2: An overview of the error code encodings from the RISC-V RERI Architecture Specification.

Figure A.3: A screenshot with the clock cycle timing of the write unit process.

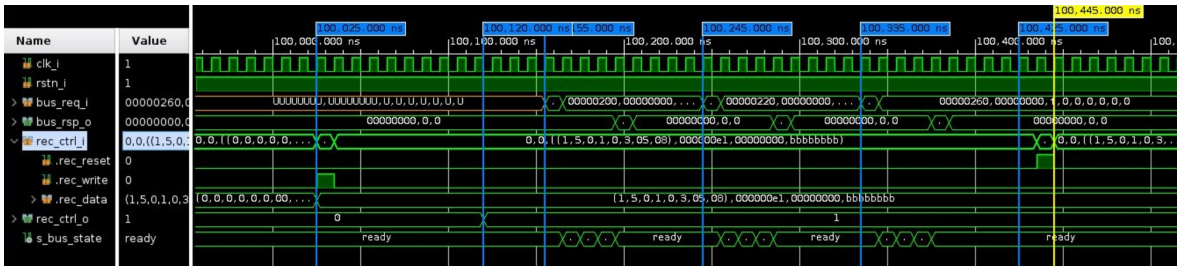

Figure A.4: A screenshot with the clock cycle timing of the record write process.

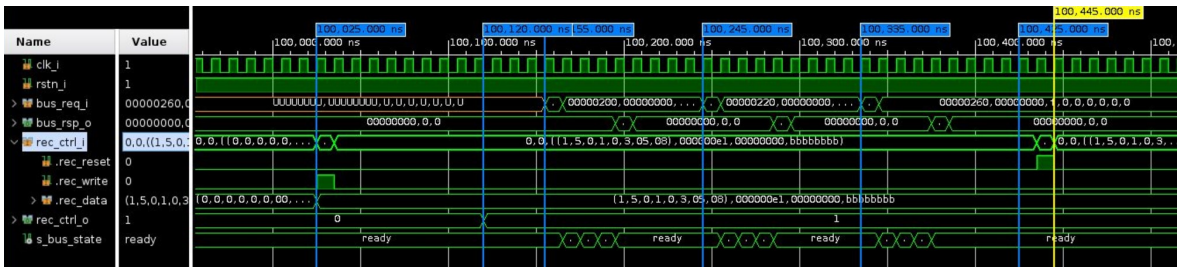

Figure A.5: A screenshot with the clock cycle timing of the record read process.

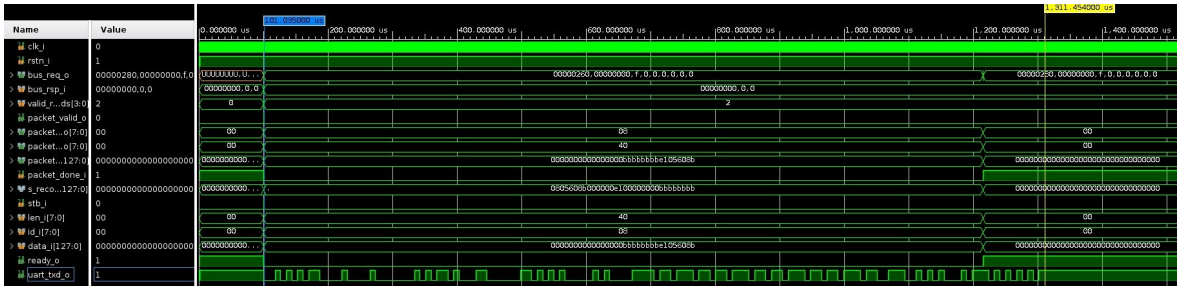

Figure A.6: A screenshot with the clock cycle timing of a UART transmission.

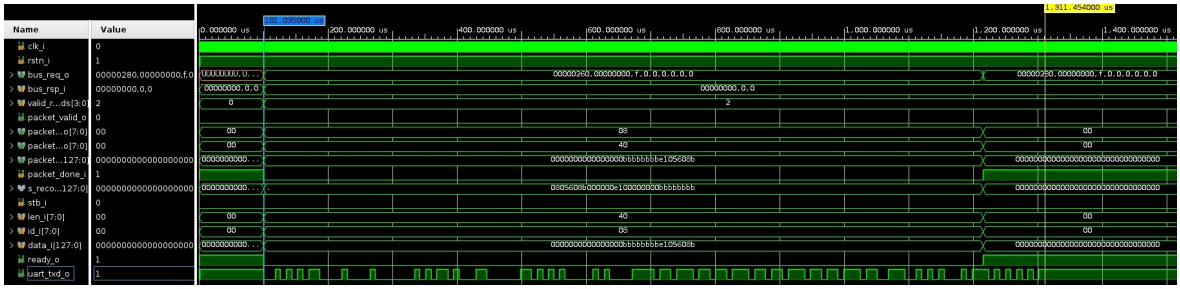

Figure A.7: A screenshot with the clock cycle timing of a UART transmission start.

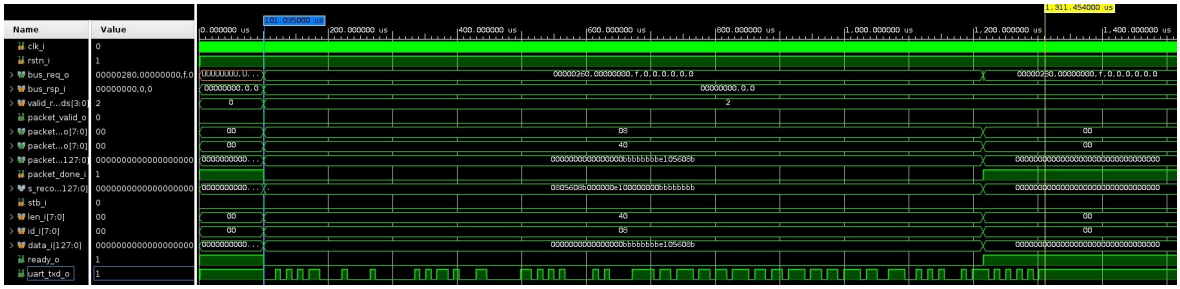

Figure A.8: A screenshot with the clock cycle timing of a UART transmission end.