Fault tolerant techniques for finite state machines in hardware designs

te Slaa, Pim S.

Award date:
2020

Department of Computer Science
Electronic Systems Research Group

Fault tolerant techniques for finite state machines in hardware designs

Master Thesis

Pim Sebastiaan te Slaa

Committee:
prof. dr. C. H. V. van Berkel
dr. ir. R. Jordans
dr. ir. A. T. Nelson

Version 1.2

Eindhoven, March 2020

Abstract

Over the last decades our world has changed significantly. Digital systems have emerged in our daily lives and businesses, due to the rapid innovations in technology. Important parts of society, such as hospitals, banks or governmental institutions do their business with the use of these systems. Our world is therefor increasingly dependant on these systems.

Almost every digital system contains some form of integrated circuit. Due to the dependability on these systems, it is important that the occurrences of errors are minimised as much as possible. Counter measures are taken to minimise errors in the design and manufacturing phase of integrated circuits. Counter measures consist of simulation and formal verification.

Although counter measures are taken during the design phase, it is still possible that faults occur in the final design. Fault tolerant techniques are implemented into the designs of integrated circuit, to mitigate faults turning into errors. Nowadays, two techniques are used: Hamming encoding and Triple Modular Redundancy.

Hamming encoding is an error correcting code that adds redundancy to the code word it encodes. By adding this redundancy, possible faults can be detected and corrected. Triple Modular Redundancy triplicates all components of the design. Each triplicated component is connected to a majority voter that gives the output.

This thesis presents an analysis of the two fault tolerant techniques. Additionally, it investigates BCH encoding for multi bit error correction. A set of benchmarks design is used, to which each technique is applied. Afterwards, an analysis is done on the area, timing and error correcting capability of the technique.

The analysis showed that Hamming encoding has an increased area of a factor 3.8 of the baseline. TMR had an increased area of a factor of 1.6 of the baseline. In terms of timing, Hamming increased the clock period of the baseline by 441 pico seconds on average. TMR increased the clock period by 86 pico seconds on average.

Acknowledgements

Before I start, I would like to thank a couple of people. First of all, I would like to thank dr. ir. Roel Jordans for his excellent guidance during the project. In addition, I would like to thank ir. Jos Huisken for all his help on the practical side of the project. Without their knowledge I simply would have been lost. Thanks to prof. dr. Kees van Berkel for join my committee, and thanks to Vincent van de Schaft and Michiel Visser for their initial work on Yosys.

Furthermore, I would like to my parents for making this study possible and always supporting me. Without their support it would have been impossible. Thanks Wiert, Flor, Thomas, Jordi, Huub and Alexander for all the times I could nag about my studies and thesis. You always told me things would be alright, and I can now finally agree with you.

Finally, I would like to thank my girlfriend Anna. Who has always been there even in difficult times. Your care and love has been the fuel I needed to finally finish this work.

To whomever I did not mention, thank you too. I'm very grateful for all the knowledge I've gained and friends I've made over the last 3 years.

List of Abbreviations

ASIC Application Specific Integrated Circuit
BCH Bose, Chaudhuri and Hocquenghem encoding
CPU Central Processing Unit
CERN Conseil Européen pour la Recherche Nucléaire
ECC Error Correcting Code
FEC Forward Error Correction
FPGA Field Programmable Gate Array
FPU Floating-point unit
FXU Fixed-point unit
FSM Finite State Machine
HDL Hardware Description Language
IC Integrated Circuit
IL Intermediate Language
IU Instruction Unit
RAM Random Access Memory
RU Register Unit
SEB Single-Event Burnout
SEE Single-Event Effect
SEFI Single-Event Function Interrupt
SEGR Single-Event Gate Rupture
SEHE Single-Event Hard Error
SEL Single-Event Lactch-Up
SESB Single-Event Snap-Back
SET Single-Event Transient
SEU Single-Event Upset
TMR Triple Modular Redundancy
TMRG Triple Modular Redundancy Generator
VHDL Very High Speed Integrated Circuit Hardware Description Language
YOSYS Yosys Open Synthesis Suite

Contents

Chapter 1

Introduction

Over the last decades our world has changed significantly. Digital systems have emerged in our daily lives and businesses, due to the rapid innovations in technology. Important parts of society, such as hospitals, banks or governmental institutions do their business with the use of these systems. Digital systems come in many forms, two of which are the Application Specific Integrated Circuit (ASIC) and the Field Programmable Gate Array (FPGA).

ASICs are integrated circuits (IC) used to carry out specific tasks, hence the name application specific. These ICs are optimized towards a single application, and carry out their tasks with greater speed and fewer resources than general purpose ICs. The downside of this is that ASICs can not be used for anything different than what they are made for.

Closely related to the ASIC is the FPGA. These devices have emerged since the late 80s and are used in many different domains. FPGAs are programmable logic devices, and help companies and universities do research and build new products. Having a lot of computing power and the ability to reprogram the device, FPGAs are an excellent choice for a lot of research and development.

Just as software might contain bugs, hardware designs can contain errors. As mentioned, today's world is dependant on digital systems. Therefor research in done to increase the robustness of digital systems. For example, server CPU's contain features to increase reliability and availability [1]. Additionally, international standards, such as IEC 61508, ISO-26262 and DO-254, require systems in the automotive, avionics and electrical system industry to contain fault tolerance for safety reasons[2, 3].

During the development process of a hardware design for an ASIC or

FPGA countermeasures are taken to minimize these errors. Simulations are used to verify the output of a design against a known set of values. However, as designs are growing in complexity it becomes more difficult to simulate every possible input combination. Formal verification is used to mathematically analyse the space of possible behaviours of a design, rather than computing results for all values.

Although counter measures are taken to minimise errors in the design phase, devices are still susceptible to faults. Faults may occur due to manufacturing problems, damage, fatigue or other deterioration. Additionally, external disturbances such as: harsh environmental conditions, electromagnetic interference, ionising radiation, unanticipated inputs, or system misuse may cause faults[4].

Programming an ASIC or FPGA is done in a hardware description language such as VHDL or Verilog. The final design often contains two parts: the data path and the control path. The data path consists of the parts that perform arithmetic operations or which hold data. The control path are parts that command the data path, memory, I/O devices according to the instructions of the memory. The control path contains some form of finite state machine. Based on the input of the machine and the current state of the memories, the machine can transition into a next state. A state encoding is used to differentiate between the states of the machine.

A subset of the earlier mentioned faults in a system, may result in actual system errors. This means the actual logical value of an element differs from its intended value. Thus, for a given input and state of the system, the wrong next state and/or output is computed. The produced system error may result in a system failure. A system failure, is defined as the inability to deliver the service as described by the systems specification. As digital systems are part of critical parts of our businesses, societies and lives, it is becoming increasingly important to mitigate any form of fault, error or system failure.

1.1 Problem Statement

Nowadays, fault tolerant techniques are used in order to harness the hardware designs against faults. By doing so, the design might be able to detect a fault and correct it, before it would result in an actual error. Therefore, lowering the amount of system failures that occur in the system.

Techniques used to harness the hardware against faults typically adds redundancy to the hardware. Through redundancy possible faults are de-

tected and corrected. The problem with this approach is the addition of overhead. Adding redundancy results in more hardware, thus more area. Because of the extra area, chances of possible errors through charged particles increases, which might add more faults than actually correct them.

The most commonly used techniques for fault tolerance are Triple Modular Redundancy (TMR) and Hamming encoding. These two techniques are already present in today's synthesis tools, however both techniques are only capable of detecting and correcting individual errors.

This master thesis aims to analyse the current two techniques, TMR and Hamming encoding, in terms of area, timing and correction rate. Additionally, this work aims to investigate an extension of the Hamming encoding technique, such that it is able to correct and detect multiple errors. Also, an analysis of this extended technique is done to compare it with the currently used techniques in terms of area, timing and correction rate.

This work only aims to research the fault tolerant techniques when used for the control paths of the system; the state encoded words of the finite state machine. Fault tolerance on data paths is outside of the scope for this research. The used and proposed technique of Hamming encoding do not lend itself to be used for the data path, as the data bits are not predictable, whereas the state bits of the finite state machine are known beforehand. Therefor the data path would need an application specific solution.

1.2 Motivation

We live in a digitalised world; electronic devices and systems are used everywhere around us. People use smartphones, laptops and tv's for comfort and assistance. Businesses and governments, use digital systems to conduct business and automate certain parts of business. Humanity has gotten very dependent on electronic devices and systems. Not only because of comfort or automation, but also because of safety.

Some systems might be very expensive and mission critical, other systems guarantee safety of the user or the systems itself. Having failures within these kind of systems, might lead to catastrophic failures. It is therefor of utmost importance to prevent system failures and errors.

Already fault tolerant techniques are used in the design of electronic hardware. These techniques are there to prevent the manifestation of errors in systems. Today's techniques mostly detect and correct single bit errors, we aim to analyse an extension to detect and correct multi-bit errors.

Additionally, the fault tolerant techniques are present only in specific

synthesis tools. These tools tend to be expensive or only work for specific brands of hardware. Additionally, some of these tools are under export restrictions, thus limiting it usability. We aim to implement our solution into an open source synthesis tool, such that anyone can use the fault tolerant techniques introduced in this thesis.

1.3 Thesis overview

This thesis is organized as follows: Chapter 2 introduces the reader to the necessary background knowledge of the project's topic. The notion of dependability is discussed. In addition, a fault model is presented. This model defines what a fault, error and system failure is. Finally, an introduction to finite state machines is given.

Chapter 3 discusses related work. Recent work on two fault tolerant techniques is presented. As well as modern synthesis tools and analysis techniques for error correction rates.

Chapter 4 presents the methodology used in during this project. Per fault tolerant technique a strategy is presented. This strategy shows how the technique is implemented and what the overall flow of work is.

Chapter 5 gives an introduction to the synthesis tool Yosys. This tool is used during this project and two fault tolerant encoding techniques are implemented in the tool. One of the Yosys' synthesis passes, the FSM pass, is discussed more in depth. This pass is vital for the recoding of the interal state machine and changes were made to implement new encoding techniques. Additionally, the implementation of the encoding techniques in Yosys is presented.

Finally, Chapter 6 presents the results of the analyses of the different fault tolerant techniques used. Chapter 7 gives a conclusion to this thesis and presents possible future work.

Chapter 2

Background

The aim of this chapter is to provide background knowledge to establish a basis of information used within this thesis. First the notion of dependability in the context of this project is discussed. Afterwards a fault model which is used in this project is presented. Finally, the basics of finite state machines are discussed.

2.1 Dependability

These days people, businesses and governments are more dependent on computer systems than ever. This implicitly means that the dependability of these systems is increased. The definition of dependability according to the Cambridge online dictionary is:

"The quality of being able to be trusted and being very likely to do what people expect"[5]

Another definition was given by [6], which focuses more on the domain of computer systems:

"Dependability is that property of a computer system that allows reliance to be justifiably placed on the service it delivers."

Together with this definition, [6] presented a taxonomy of what dependable computing is. Dependability is built up from three main classes:

- Impairments: Condition which cause the system to be undependable.

An impairment may also result from undependability. This means, the system cannot reliably deliver the service it is specified to.

- Means: Methods to ensure the system can reliably deliver the service it is specified to. Additionally, the means provide tools to the user to validate the ability of the system to reliably deliver its service.
- Measures: Metrics to express the quality of the service delivered by the system.

The final taxonomy tree as presented by [6] is shown in figure 2.1. The class of impairments consists of: faults, errors and failures. Within this research a fault model is presented, such that it is clear what we define as a fault, an error or a system failure.

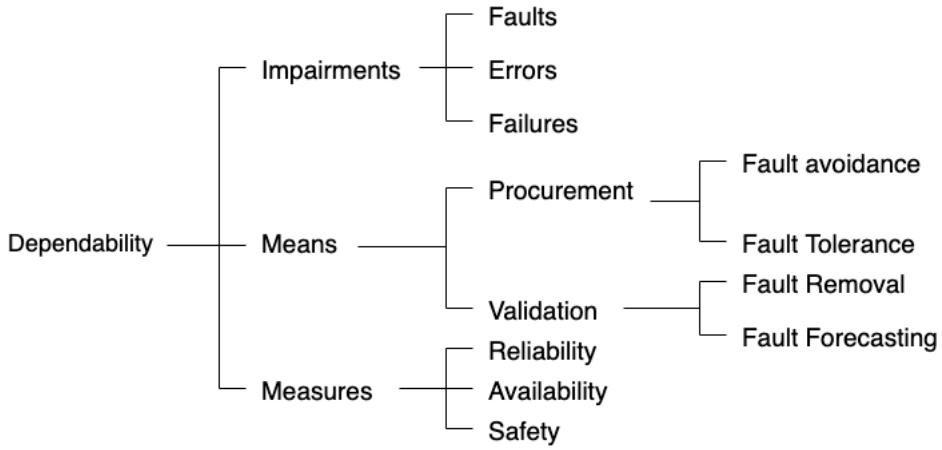

Figure 2.1: *The taxonomy tree of dependability as presented by [6]*

The class of means, is subdivided into procurements and validations. The procurements consist of fault avoidance and fault tolerance. These methods should provide the system to deliver its specified service. Fault avoidance is the prevention of faults by means of construction. For example, while designing a system that operates in space, use shielding to mitigate interference of charged particles. Fault tolerance adds redundancy to the system such that it can deliver the specified service while faults occur or have occurred. Redundancy can be added, for example, through replication of specific components.

The validation class consists of fault removal and fault forecasting. These methods are in place to reach confidence in the system's ability to deliver the

specified service. Fault removal minimises the presence of faults during the design phase by means of verification. Formal verification and simulation are methods which are used to find errors during the design phase. Once found, the system designer can remove the error from the design. The estimation of the presence, the occurrence, and the consequences of faults by means of evaluation is called fault forecasting.

Finally the measures class consists of metrics to express the quality of the service delivered by the system. Reliability is a measure of the continuous delivery of proper service (or, equivalently, of the time to failure) from a reference initial instant. Availability is a measure of uptime with respect to downtime. Lastly, safety is a measure of time to catastrophic failure. Due to their context depended nature, this thesis does not focus on the any of the metrics from the measures class.

Within this research we will mainly focus on the fault impairment and the fault tolerance procurement parts of the taxonomy tree presented by [6]. A fault model is presented and different fault tolerant techniques are discussed. The rest of the means and measures class are out of scope for this research.

2.2 Fault model

In the previous sections we introduced the notion of dependability in a general view. In order to use these notions it needs to be clarified what we perceive as a fault, an error or a system failure.

This section provides information on what faults, errors and system failures are. Different types of faults are presented and finally the application of the fault model is explained.

2.2.1 Faults, errors and system failures

Faults, errors and system failures are closely related to one another; one might be the cause of the other. Over time different classifications and definitions have been given to these phenomena[6, 4]. These studies give a more general view of what faults, errors and system failures are. By using their definitions, our own definitions are formed and applied to our domain. First, faults are defined, after which errors and system failures are defined.

A fault is defined as an anomalous physical condition within the system [4]. Causes of this condition vary from design errors, manufacturing problems or external disturbances. A fault on its own is not harmful to the system. It is only harmful when it results in an error.

An error is a part of the system which causes the systems output to differ from its intended value. [4] states that an error is the manifestation of a fault. Additionally, [6] states that a fault is the cause of an error.

Finally an error may lead to a system failure. This means the system cannot deliver its service as defined by its specification.

Within this project we agree upon the definition that: A fault is physical anomalous condition, which may lead to an error. An error is a manifestation of a fault which may lead to different system output than expected. Because of the error, the complete system may fail to deliver its specified service, thus resulting in a system failure.

2.2.2 Applying the fault model

As [6] mentioned, it might be difficult to find the cause of a fault because of recursion. A fault may lead to an error, which may lead to a system failure. Because of this system failure a fault may be introduced, and the cycle repeats itself:

$$... \rightarrow failure \rightarrow fault \rightarrow error \rightarrow failure \rightarrow ...$$

This cycle is mostly the result of hierarchical system design. System design is often done using a hierarchical approach. The final system is build up from smaller subsystems, which also can be build up from smaller systems. Therefore the final system consists of layers of subsystems. This hierarchical system design, is part of why this failure cycle exists; a failure in a lower level of the system, might introduce an error in a higher level of the system.

Thus, depending on what part of the system you view, each may lead to different views on what faults, errors and failures are. For example, an ionising particle hits a part of an integrated circuit. This particle collision causes a single bit to flip in a memory element. Because of this erroneous bit, a certain LED is not correctly toggled. A light sensor, keeps track of the LED and based on its state provides a service.

Within this example we might claim the ionising particle collision to be the fault, the bit flip in the memory element to be the error, and the system failure is the LED not correctly being toggled. However, it might also be possible to view the erroneous bit as the fault, the incorrectly toggled LED to be the error, and the failing light sensor to be the system failure.

As shown, it is important to have a notice of what we perceive as a fault, an error or system failure. Within this project an erroneous bit flip

in a state signal is considered a fault. The error is the finite state machine moving to an incorrect next state because of the wrong state signal. Finally, the system failure is the hardware failing to provide its specified service.

2.2.3 Types of faults

As previously mentioned a fault is defined as an anomalous physical condition within the system [4], which may be caused by design errors, manufacturing problems or external disturbances. This thesis focuses on faults which are caused by external disturbances, mainly radiation effects. This section aims to present the different types of faults caused by radiation.

Electronic devices and integrated circuits are susceptible to radiation effects. The charged particles from the radiation interact with the circuitry and might change state of memories or influence signals on the circuit. The change in state or signals may result in system errors. Different types of fault have been found and studied.

The effects of radiation on circuitry can be split into two main classes: Cumulative Effects and Single Event Effects (SEE). Cumulative Effects are the result of exposure to radiation for a longer time. Over time the radiation leads to component degradation. Once the components are sufficiently damaged, faults may arise. The Cumulative Effects are not further discussed as they are not within the scope of the problem presented in this report.

SEEs are caused by a single charged particle interaction with the circuitry. The interaction may lead to different effects: Destructive or Non-Destructive. The Destructive SEEs lead to permanent damage and are more difficult to recover from. The Destructive SEEs are the following[7]:

- Single-Event Latch-Up (SEL) - The result of the triggering of a parasitic thyristor mainly existing in CMOS circuits. When it occurs, a high current flows and if the power supply is maintained, the device can be destroyed by thermal effects.

- Single-Event Snap-Back (SESB) - The result of the triggering of a parasitic bipolar structure. For example when each transistor is dielectrically isolated from its neighbours, SOI MOS is not sensitive to SEL, but it can be sensitive to SESB because of floating body effects when body contacts are insufficient.

- Single-Event Hard Error (SEHE) - An unalterable change of state because of damage done to a cell by an ion strike.

- Single-Event Burnout (SEB) - The triggering of the parasitic bipolar structure in a power transistor, accompanied by regenerative feedback, avalanche and high current condition.

- Single-Event Gate Rupture or Single-Event Dielectric Rupture (SEGR) - The destructive rupture of a gate oxide or any dielectric layer by a single ion strike. This leads to leakage currents in the IC.

The Non-Destructive SEEs are temporary errors and are easier to recover from. The following SEEs are Non-Destructive[7]:

- Single-Event Transient (SET) - A temporary voltage spike at a node in the integrated circuit. This is caused by a ionising particle hitting the semiconductor.

- Single-Event Upset (SEU) - A change in a single bit in the state of memory components on the integrated circuit.

- Multiple-Cell Upset / Multiple-Bit Upset - A change in multiple bits or logical cells of the integrated circuit.

- Single-Event Functional Interrupt (SEFI) - Soft error that causes the component to reset, lock-up or otherwise malfunction.

While Destructive SEEs do permanent damage and may lead to permanent error in the state signal, we still mention them as our fault tolerant encodings are able to recover from these errors. This research focuses on the SEUs and the Multiple-Cell Upset types.

2.3 Fault tolerance

Within this project different fault tolerant strategies are implemented and analysed. Each fault tolerant strategy consist of certain elements. This section aims to discuss these elements and present what elements are part of the different strategies used within this project.

2.3.1 Elements of fault tolerant strategies

According to [4], each fault tolerant strategy contains one or more of the following elements:

- Masking: Dynamic correction of generated errors.
- Detection: Detection of an error - a symptom of a fault.
- Containment: Prevention of error propagation across defined boundaries.
- Diagnosis: Identification of the faulty module responsible for a detected error.
- Repair/Reconfiguration: Elimination or replacement of a faulty component, or a mechanism for bypassing it.
- Recovery: Correction of the system to a state acceptable for continued operation.

The masking and detection elements are often used together. Both coding theory and module replication are fault tolerant strategies that combine masking and detection. Using coding theory, the word is encoded and redundancy is added to the word in the form of parity bits. Depending on the amount of parity bits, errors can be detected and corrected.

Module replication uses a different approach to apply masking and detection. A module in a system may produce some form of output. To increase the likeliness of the output being error free, the module is replicated. Each of the replicated modules output is connected to a voter/comparator. The voter chooses all the outputs on majority votes. This way by replication of modules, a possible error can be detected and corrected.

The containment element is closely related to the masking and detection element. Containment tries to minimise error propagation throughout the system. This is achieved by defining containment boundaries in the system. At each boundary, i.e. an interface between modules, a detection or

correction module is placed. Once an error is detected on a containment boundary, the system might correct it or disregard the modules output.

It might happen that a (sub-)module breaks or stops working. This module could behave differently than expected. If the faulty module works together with other modules to output some function, it might not be clear what module is causing the faulty output. To diagnose which module is causing the faulty output, the modules could be supplied with self checking logic. The self checking logic simply checks if the output is reasonable; if not, an error will be forwarded. This error can then be used to ignore the output of the module.

If the system contains a faulty module it might need to be repaired or replaced. One way is to physically replace the faulty module with a new working one. Another way is to switch off the faulty module and offload its work to other resources of the system; this is logical replacement of the module.

Recovery is the act of recovering from an error which was propagated through the system. Often, the longer the time between the occurrence of the error and the detection of the error, the more damage the error does. Therefore recovery scheme does a system rollback to a previous known system state before the occurrence of the error.

2.4 Finite State Machines

Within this project we apply the introduced fault tolerant strategies to finite state machines (FSM) of hardware designs. Therefore an introduction to finite state machines is needed.

This section gives a short introduction to two types of FSMs, namely: The Mealy machine and the Moore machine. These FSMs are often used in the design and implementation of FPGA or ASIC designs. The machines are used as an abstract model of the actual circuitry. Because of this abstraction we can define relations and characteristics of the machines. These relations and characteristics are useful when, for example, comparing machines or minimising machines.

2.4.1 Mealy machine

The Mealy machine is a sequential machine whose input and current state determine the output. The machine has the following formal definition:

Definition 2.4.1. *A Mealy machine is a quintuple*

$$M = (S, I, O, \delta, \lambda)$$

where:

(i) S *is a nonempty set of states;*

(ii) I *is a nonempty set of inputs;*

(iii) O *is a nonempty set of outputs;*

(iv) $\delta : S \times I \to S$ *is a called the transition (or next state) function;*

(v) $\lambda : S \times I \to O$ *is called the output function.*

As an example let us define a Mealy machine M to be a quintuple, such that

$$M = (S, I, O, \delta, \lambda)$$

with $S = \{r, s, t\}$, $I = \{a, b\}$, $O = \{0, 1\}$. Table 2.1 shows the flow (or transition) table of the defined machine M. From the definition of M together with its flow table, the state machine graph can be build up.

Take transition $\delta(r, a)$, the flow table in Table 2.1 shows that the next state is s, thus $\delta(r, a) \to s$. Additionally, $\lambda(r, a)$ is defined as 0 in the flow table, thus $\lambda(r, a) \to 0$. Because of the aforementioned transition and output function, an edge can be drawn from state r to state s with label $\frac{a}{0}$.

If every combination of state and input from the flow table is processed this way, a state machine graph as shown in figure 2.2 is constructed. This is the state machine corresponding to the defined Mealy machine M.

Table 2.1: *The flow table of Mealy machine M*

State	Input		Output	
	a	b	a	b
r	s	r	0	0
s	t	t	0	1
t	r	s	1	0

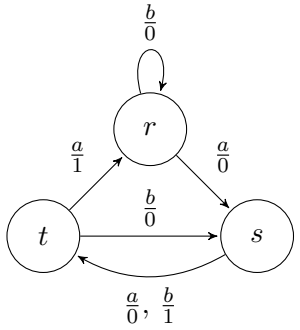

Figure 2.2: *The state machine graph of the Mealy machine M*

2.4.2 Moore machine

The Moore machine's definition is in many ways the same as the Mealy machine's definition. However, the two machines do have a significant difference in the definition of their output function. On the contrary to the output function of the Mealy machine, the Moore machine's output function only depends on its current state. The Moore machine has the following formal definition:

Definition 2.4.2. *A Moore machine is a quintuple*

$$M = (S, I, O, \delta, \lambda)$$

where:

(i) S *is a nonempty set of states;*

(ii) I *is a nonempty set of inputs;*

(iii) O *is a nonempty set of outputs;*

(iv) $\delta : S \times I \to S$ *is a called the transition function;*

(v) $\lambda : S \to O$ *is called the output function.*

As an example let us define a Moore machine N to be a quintuple, such that

$$N = (S, I, O, \delta, \lambda)$$

with $S = \{r, s, t\}$, $I = \{a, b\}$, $O = \{0, 1\}$. Table 2.2 represents the flow table of the defined machine N. Because the transition function δ is not different from the Mealy machine, it is trivial that $\delta(r, a) \rightarrow s$ holds for machine N.

The output function only depends on the current state of the machine therefore $\lambda(r)$ gives the output for state r. As shown in the flow table, the output corresponding to state r is 0. Because only the state determines the output, the output is not written on the edges, but rather in the states.

Again, if the complete flow table is processed this way, the state machine graph can be constructed. Figure 2.3 shows the state machine graph for Moore machine N.

Table 2.2: *The flow table of Moore machine N*

State	Input		Output
	a	b	
r	s	r	0
s	t	t	0
t	r	s	1

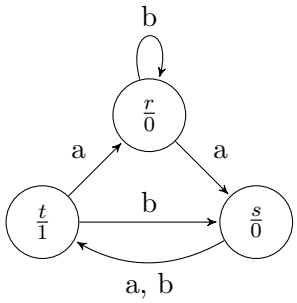

Figure 2.3: *The state machine graph of the Moore machine N*

2.4.3 State encoding

State machines are used as an abstract model of actual circuitry. In the end, the goal is a real implementation of the state machine into circuitry. But before that is possible, the states of the state machine need to be assigned a code. This code will represent the states when the state machine is realised into the circuit.

Choosing the correct encoding is a difficult problem as each choice of encoding has its own implications. For example, different encodings use different amounts of bits to encode a state. Additionally, some encodings result is more hardware. This makes choosing the right encoding an important topic.

A good example of the state assignment problem is given below. This example was introduced in [8] and does a wonderful job of showing different results of different encodings.

Table 2.3: *The flow table of a Moore machine M*

State	Input		Output
	0	1	
1	4	3	0
2	6	3	0
3	5	2	0
4	2	5	1
5	1	4	0
6	3	4	0

Table 2.4: *State assignment method 1*

State	Binary variables		
	y_0	y_1	y_2
$1 \rightarrow$	0	0	0
$2 \rightarrow$	0	0	1
$3 \rightarrow$	0	1	0
$4 \rightarrow$	0	1	1
$5 \rightarrow$	1	0	0
$6 \rightarrow$	1	0	1

Table 2.5: *State assignment method 2*

State	Binary variables		
	y_0	y_1	y_2
$1 \rightarrow$	1	1	0
$2 \rightarrow$	1	0	1
$3 \rightarrow$	1	0	0
$4 \rightarrow$	0	0	0
$5 \rightarrow$	0	0	1
$6 \rightarrow$	0	1	0

Table 2.3 presents the flow table of a machine M. Beneath it, two different state encodings are presented in table 2.4 and 2.5. Transition functions Y_0, Y_1, Y_2 and output function z can be defined for both encodings.

Let us construct Y_0 for encoding method 1, to show the process of constructing such an output function. We are looking for the set of transitions which result in a state where y_0 is 1. Table 2.4 shows that this only holds for state 5 and 6. The next step is looking for the transitions in table 2.3 ending in these states. As can be seen from table 2.3, the transitions $\delta(2,0)$, $\delta(3,0)$ and $\delta(4,1)$ are the transitions which result in either state 5 or 6.

Because the states are encoded, we can now write these transitions as a Boolean function to construct Y_0. For example, $\delta(2,0)$ consists of state 2 and input 0. As state 2 is encoded to be 001, this can be expressed as

$\bar{y}_0\bar{y}_1y_2$. The input is 0, therefor we can express is as $\bar{x}$. Putting these two expressions together, results in: $\bar{y}_0\bar{y}_1y_2\bar{x}$.

If we now repeat this process for every transition in Y_0, Y_1 and Y_2, we end up with the following result for Y_0, Y_1 and Y_2:

$$Y_0 = \bar{y}_0\bar{y}_1y_2\bar{x} + y_1\bar{y}_2\bar{x} + y_1y_2x$$
$$Y_1 = \bar{y}_1x + \bar{y}_0\bar{y}_1\bar{y}_2 + y_0y_2$$
$$Y_2 = y_0x + y_1\bar{y}_2x + y_1y_2\bar{x} + \bar{y}_0\bar{y}_1\bar{x}$$

The output function z can be constructed the same way as the transition functions. We look at the flow table of the machine and determine what states have the output 1. We then use the encoding of these states as a way to write the Boolean function. In this example only state 4 has an output of 1, therefor the output function z is:

$$z = y_1y_2$$

If the same process is used with the second encoding from table 2.5, the output function and transition functions are as follows:

$$Y_0 = y_0x + \bar{y}_0\bar{x}$$
$$Y_1 = y_2\bar{x}$$
$$Y_2 = \bar{y}_1\bar{y}_2$$
$$z = \bar{y}_0\bar{y}_1\bar{y}_2$$

As can be observed, the Boolean functions for the second state encoding method are shorter and simpler in terms of logic. Because of this less logic gates(thus hardware) is needed when realising the state machine with encoding method 2. The difference in the Boolean functions of both encoding methods shows that choosing the right encoding is important as it has a direct effect on the resulting hardware.

Chapter 3

Related work

This chapter provides insight in recent work and studies done related to this project. First, work related to a technique called Triple Modular Redundancy(TMR) is discussed. Afterwards, work related to a technique based on Hamming encoding is presented. Finally, two modern synthesis tools are presented which already offer the use of the discussed techniques. Also analysis techniques for error correcting capabilities are discussed.

3.1 Triple Modular Redundancy

There has been a need for dependable and reliable computing since the first digital computers were built. Already in the 1940-1950's there was a demand for dependable and reliable computers. Computers at that time were built out of components such as: Relays, vacuum tubes and electronic cathode raytubes. These components were known to have a high failure rate, and with that came the demand of reliability.

The first fault tolerant computer, the SAPO, was built in 1950-1954 in Czechoslovakia. The SAPO was also built using cathode raytubes and therefore periodically encountered failures. Some fault tolerant techniques were used to prevent these failures. The memory element of the computer contained parity bits for memory read operations. Additionally, a recurrence of a fault would halt the machine in its current state and all data on the detected errors would be printed to the operator's console. Finally, the computer had a parallel CPU which was triplicated. A majority voter was connected to the outputs of the three identical CPU's [9].

Even in the last two decades research has been done in fault tolerant computing. From server CPU's to microprocessors, most of it contains

some form of fault tolerance. In 1999, [10] presented a the S/390 G5 microprocessor. This processor did not implement TMR, but rather duplicated components of its execution pipeline, such as: the instruction unit (IU), fixed-point unit (FXU), floating-point unit (FPU) and cache and register unit (RU). Additionally, it uses error correcting codes and parity to protect data in the processor.

Later in 2002, [11] presented the architecture and implementation of the LEON-FT processor. This design used TMR on all flip-flops, parity on the caches and BCH encoding on the register file. The total area overhead was 100% on the core itself, and 39% with RAM cells. The timing penalty introduced by TMR was solely introduced through the TMR voter. The penalty was about two gate delays or 8% of the cycle time.

Recently, two researchers at CERN gave a talk a the Chaos Communication Congress [12]. During this talk the researchers discussed their findings on how to design highly reliable digital electronics using fault tolerant techniques against SEEs. They presented techniques for different application levels, such as: Technology level, cell level, block level and system level.

To mitigate SEEs on technology level, the researchers used technology scaling. By scaling down the size of the transistor, the overall design size would shrink. Therefor the chances of the die being hit would lower, and thus chances on faults because of SEEs would lower. However, technology scaling has a disadvantage. A smaller transistor has a lower critical charge than a bigger one; meaning that a smaller charge is needed to change the output of the transistor. Therefor, by technology scaling you might make your design susceptible to particles with a lower charge.

On cell level two techniques could be applied. First, the critical charge of the cell could be increased. This is done by increasing the node capacitance by using bigger transistors (this also increases the collection electrode) or by extra capacitance on metal layers (slows down your design). Second, the information of a cell could be stored in multiple nodes. The downside of applying mitigation on cell level is that changes have to be made to your standard cell library.

Mitigation of single event effects on block level involves triplication (TMR) of the design or parts of the design. This is based on the same principles as the SAPO computer from 1956. The CERN researchers showed three possible implemenations of TMR: Simple, clock skewed or full triplication.

The simple implementation consists of replicating three state registers and connecting these to a voter. The outcome of the voter is fed back as input into the logic, such that errors in the registers are removed one clock cycle later. Figure 3.1 shows the design of a simple TMR implementation.

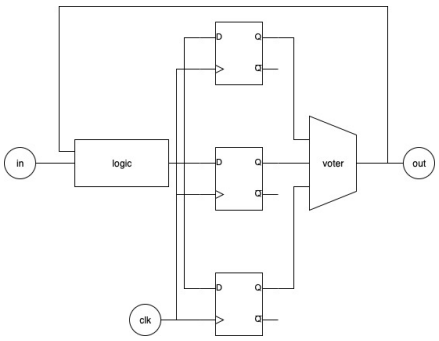

Figure 3.1: *The simple TMR implementation.*

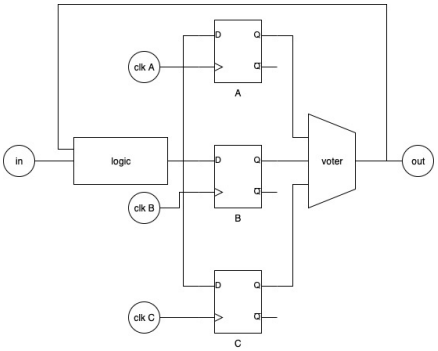

Figure 3.2: *The clock skewed TMR implementation.*

Although the simple TMR implementation already provides some form of fault tolerance, it is not optimal. In the simple implementation the registers are triplicated, however the voter is not protected. If an upset would occur in the voter at the moment of latching the data into the three registers, all three of the registers would become corrupt.

To overcome this problem the researchers used clock skewing on the three registers. Figure 3.2 shows the clock skewed TMR design. Each of the registers has its own clock and each clock is slightly skewed relative to the others. This way each register will latch the output data fed by the voter on a slightly different time. This lowers the probability of latching the corrupted the data in all three registers at once.

Clock skewing might solve the initial problem of corrupting three registers at once, but it also introduces possible indeterministic behaviour in the design. Because of the clock skewing one of the registers might already be in next state, while the others are not. This leads to race conditions and indeterministic behaviour.

The full TMR implementation presented by the researchers triplicates all the elements of the design: logic, voters and registers. This implementation removes errors in registers after one clock cycle. Additionally, if an upset occurs in a voter, the error is only propagated to one register. Although the full TMR implementation seems the most robust and optimal protection, it comes at a cost. Figure 3.3 shows the full TMR implementation as presented by the CERN researchers.

Finally the researchers presented an overview of metrics of each implementation. The results are shown in table 3.1. As can be observed from the table the full TMR provides the best protection; however it comes at

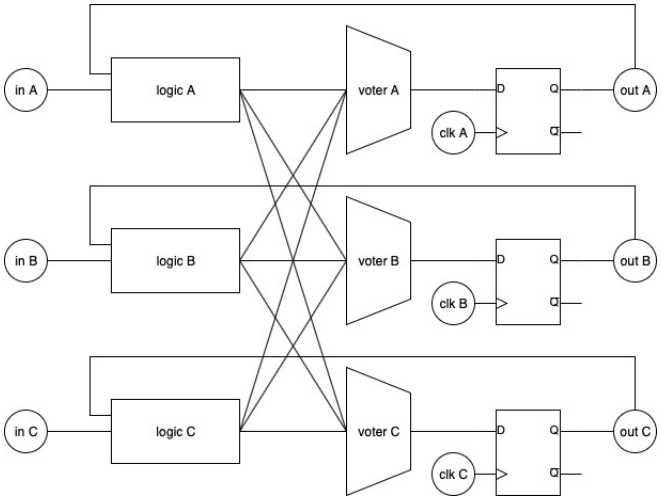

Figure 3.3: *The full TMR implementation.*

the cost of area overhead as all logic, voters and clocks are triplicated. The clock skewed version still provides a good protection but at a much lower speed due to the clock skew and voter delay. Finally the simple TMR has the worst protection, but is the cheapest in terms of area and power.

Table 3.1: *Resource, speed and protection table for the three presented TMR implementations.*

		Simple TMR	Clock skewed TMR	Full TMR
Resources	FF	x3	x3	x3
	logic	x1	x1	x3
	voters	x1	x1	x3
	clocks	x1	x3	x3
Speed		+ (voter delay)	- (voter delay and clock skew)	+ (voter delay)
Protection		-	+	++

3.2 Hamming encoding

Another way of SEE mitigation is the use of forward error correction (FEC) codes on the state signals of the state machine. This way the state encoding of the state machine is redundant and therefor its tolerance to faults is

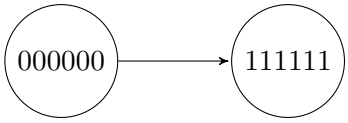

Figure 3.4: *The original FSM with only base states*

increased. Coding techniques such as Hamming or Reed-Solomon are often used for this. Melanie Berg at the Single Event Effects Symposium [13] in 2014 presented a method that uses Hamming encoding to encode the state machine state signals.

The method used in [13] is based on altering the finite state machine of the design by adding redundant states, so-called companion states. These companion states stem from the added redundancy of the Hamming code. Per base state (the original state) of the state machine the following procedure is applied: The base state's bits are encoded using Hamming encoding. Now per possible fault in the state bits, a companion state is added. This companion state has a transition to its base state's next state.

For example, figure 3.4 shows a non encoded FSM with two states. As each state has 5 bits, there are 5 possible faults which can occur. Therefor 5 companion states are added, each covers one fault. The resulting FSM is shown in 3.5. To better understand how the companion states are encoded, a zoomed in version of encoded state 00000 is depicted in 3.6.

If a fault occurs in the encoded version of the FSM, there is a state which covers that fault. By adding transitions from the companion state to the next state of the related base state the error is corrected.

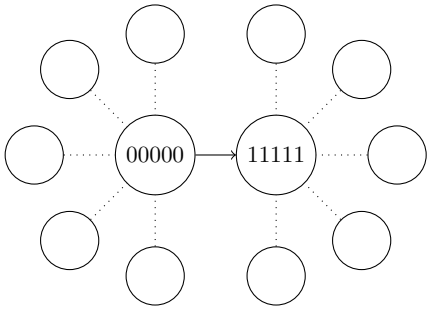

Figure 3.5: *The original FSM with added companion states*

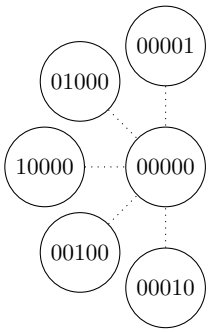

Figure 3.6: *A zoomed in view on the encoded state 00000 of figure 3.5*

3.3 Modern synthesis tools

This section presents two tools which already exist and offer the use of fault tolerant techniques.

- Xilinx TMRTool: Developed by Xilinx, this tool offers automatic generation of TMR for space-grade reprogrammable FPGAs. It is only available for the FGPA device families Virtex-5QV and Virtex-4QV. The tool is specifically designed for space applications and provides full immunity to SEU and SET for any of the mentioned device families.

- Synopsys Synplify: Developed by Synopsys, this tool supports the FPGA device families from: Altera, Lattice, Microsemi and Xilinx. Synplify provides both TMR and Hamming-3 encoding as fault tolerant techniques. Using Synplify, fault tolerant designs can be implemented, such that these designs adhere to industry standards, like: DO-254, IEC 61508 and ISO 26262.

Although both of the presented tools are of professional standards, they come with expensive licenses and export restrictions. This limits the use of the tools by researchers or individuals interested in fault tolerant designs.

To overcome the problem of license costs and export restrictions, the CERN researchers released their tool under an open source license. The Triple Modular Redundancy Generator (TMRG) tool was created 4 years ago by two CERN researchers. This tool is not a synthesis tool, but rather modifies the design at source code level, triplicating all inputs, outputs, registers and logic. According to the tool's creators, implementing TMR at source code level makes debugging it easier.

In line with the open source philosophy of the CERN researchers, the open source synthesis tool Yosys is used during this project. During the course of this project, two encoding techniques are implemented into Yosys. This way, there will be an open source synthesis tool with two fault tolerant encoding techniques, which is free of license costs or export restrictions.

3.4 Analysis of Error Correction Rate

This section presents the notion of cross-section, which is used to define how sensitive a hardware design is to ionizing radiation. Afterwards, different techniques are presented to test and measure the cross-section of hardware designs.

3.4.1 The cross-section

The cross-section(σ) of a device is defined as the SEE response of the device to ionizing radiation. The units for cross-section is cm^2 per device or bit, and it is calculated as follows:

$$\sigma = \frac{number\_of\_errors}{ion\_fluence}$$

3.4.2 Tests and measurements

To determine the cross-section of a device, two types of tests can be done: tests with heavy ions, and tests with lasers. The test with heavy ions, involves placing the device in a vacuum chamber and shooting accelerated heavy ions at the device, while measuring the number of errors and the total particle fluence to determine the cross section.

The tests using lasers are done to more selectively test your device. Using lasers you can better 'steer' the beam of energy, thus target specific parts of your design. This makes it possible to research individual parts of the device. The downside of these approaches is that testing setups are rather expensive to use. We therefor do not use these approaches within this project.

Another way of determining the error correcting capability of a design is done using error injection at netlist level. Erinan is a tool developed at the University of Technology of Eindhoven which injects errors and analyses the propagation of these errors.[14] The tool analyses a design, by making two identical copies of it. Both these copies take the same input; however, errors are injected into the flip-flops of one of the copies. Finally, the output of both design is analysed. This way the tool gives insight on what the effects of the injected errors are and how these errors propagate.

Although Erinan is a cheaper alternative to test and measure error correcting capabilities, it does not take into account the placement of the flip-flops in the design. Additionally, analyzing all possible combinations of errors in a design, might become time consuming as the number of flip-flops increase. Within this project, we therefor stick to a theoretical analysis of the error correcting capabilities.

Chapter 4

Methodology

This chapter provides insight on the methodology used in this project. It discusses what kind of research is done, how it is set up and how evaluation of the gathered data can be done.

4.1 Strategy

Different metrics are used to analyse and compare the fault tolerant techniques used in this project. Within this project we focus on: area, timing and correction rate of errors. This section provides information on how the different metrics are measured and how they are analysed and compared to each other.

A benchmark set of Verilog designs is used to analyse the different fault tolerant techniques in this project. This set of benchmarks consists of different Verilog designs. Each design describes hardware containing some form of FSM. First, the designs are synthesised to netlists by Yosys using the FreePDK45 cell library[15]. Using Yosys, the designs are synthesised and the area metric is measured. The area metric is the chip area in nm^2 as reported by Yosys after synthesis.

Once the designs have been synthesised, the timing metric is measured using Cadence Genus. Cadence Genus reports critical paths of the designs according to some set clock period. The final timing metric can be found in the generated timing reports from Cadence Genus. The timing metric is the clock period for the design in pico-seconds.

In order to compare the different techniques to each other, a baseline implementation is synthesised and analysed. This baseline implementation is the benchmark set without any fault tolerant technique applied to it. The

baseline implementation is used to analyse each fault tolerant technique's overhead.

Once the baseline has been set, the fault tolerant techniques are applied to the benchmark set. These altered benchmarks are also measured in area and timing as already mentioned. After all the techniques are measured, a comparison can be made in terms of all metrics. This comparison is discussed at the end of this thesis.

To apply each of the fault tolerant techniques to the baseline, some form of preparation is needed. The preparation per technique is discussed in the sections below. Additionally, the method which these techniques use are discussed.

4.1.1 Baseline

The baseline implementation does not need a lot of preparation. To get the baseline results, the Verilog benchmark set has to be synthesised by Yosys. Afterwards the timing analysis is done by Cadence Genus. A Yosys script file is used to automate the synthesis flow within Yosys. The script used to synthesise the designs is shown in Listing 4.1. This script reads the Verilog designs, elaborates the design hierarchy and does the technology mapping. Once those steps are completed it tries to map the flip-flops from the specified cell library(in our case FreePDK45) and does the logic mapping. Finally, it writes the synthesised design to an output Verilog file. An overview of this process is depicted in Figure 4.1.

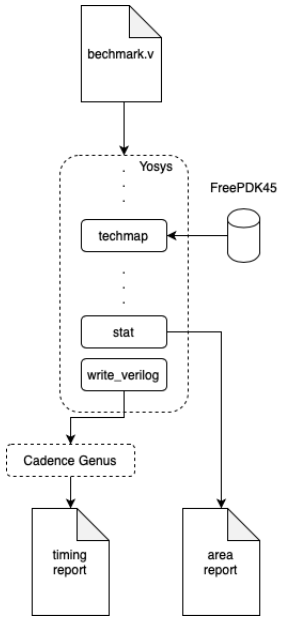

Figure 4.1: *The synthesis flow of the baseline implementation*

The ouput file from Yosys is then used as input file for Cadence Genus. The tcl script shown in listing 4.2 is used to get the timing reports from Cadence Genus. The final results generated by Cadence Genus are stored in a report file named: *modulename_timing.rpt*.

Listing 4.1: *Example synthesis script for Yosys [16]*

```
# read design
read_verilog mydesign.v

# elaborate design hierarchy
hierarchy -check -top mytop

# the high-level stuff
proc; opt; fsm; opt; memory; opt

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Listing 4.2: *Genus tcl script to create timing report of a design*

```
set_db / .library path/to/library.lib
read_netlist synth/synth.v
set top m_module
check_design -unresolved $top
define_clock -period 10000 -name CLK {clk}
report timing -max_paths 10 > timing_rpts/${top}_timing.rpt
exit
```

4.1.2 Triple Modular Redundancy

Triple Modular Redundancy (TMR) is an error correction technique which is based on spacial redundancy. Spacial redundancy is a form of fault tolerant techniques where vital modules of a system are replicated [7]. Each of the modules are connected to the same input. The outputs of the modules are connected to a voter. This voter compares all of its inputs based on majority and determines the correct output base on that comparison.

Figure 4.2 shows two implementations of TMR. Both of them are used to detect and correct different kind of faults. For example, the circuit depicted in figure 4.2(a) is used to detect single-event transient (SET). Here the combinatorial logic is replicated three times, whereas in figure 4.2(b) the flip-flops are replicated. The replication of flip-flops is used to mitigate single-event upsets (SEU).

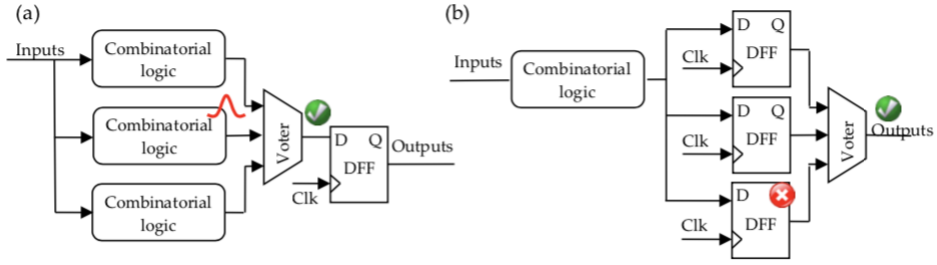

Figure 4.2: *SET(a) and SEU(b) detection with a TMR architecture[7]*

To prepare the benchmark set with the TMR technique applied to it all flip-flop cells in the benchmark design have to be replaced by a TMR flip-flop cell. Unfortunately the FreePDK45 cell library used in this project does not contain a TMR cell design by default. Therefore our own TMR cell implementation is used. The source code is shown in listing 4.3 and the final netlist is depicted in figure 4.4.

Once our TMR cell has been synthesised it is used in the synthesis flow of the TMR implementation. This flow is shown in figure 4.3 and is an extension of the baseline synthesis flow. First, all flip-flops in the baseline output are replaced by our TMR cell. Afterwards, the altered baseline output is fed to Yosys. The output is the same design, except that every flip-flop is now a TMR cell.

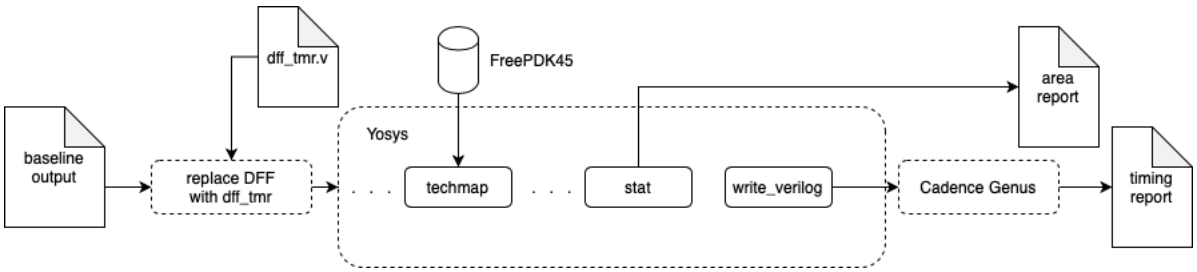

Figure 4.3: *The synthesis flow of the TMR implementation.*

Listing 4.3: *Verilog source code for the TMR cell implementation.*

```
1  module dff_tmr(D, Q, clk);
2          input wire D;
3          input wire clk;
4          output wire Q;
5
6          reg Qa, Qb, Qc;
7
8          always @(posedge clk) begin
9                  {Qa, Qb, Qc} <= {D, D, D};
10         end
11
12         assign Q = (Qa & Qb) | (Qb & Qc) | (Qa & Qc);
13 endmodule
```

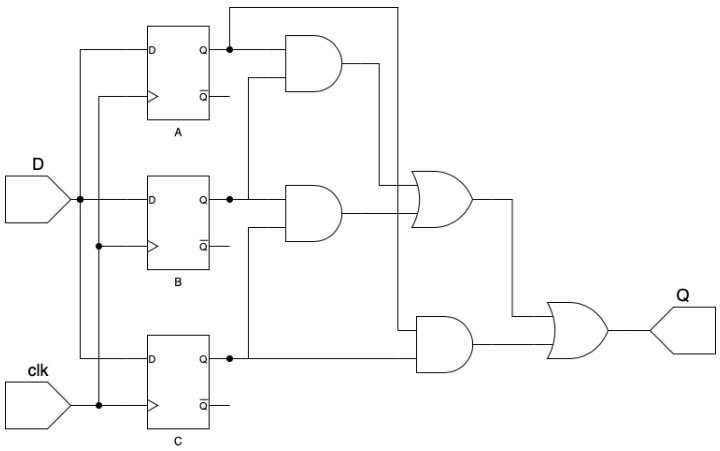

Figure 4.4: *The TMR cell design after synthesis done by Yosys.*

4.1.3 Hamming encoding

Hamming code is a linear error correcting code which adds parity bits to the word it encodes. By adding in more than one parity bit, it is not only possible to detect an error but also which bit causes the error. For example, with a seven bit word there are 7 different errors possible. By adding three parity bits, it is possible to detect the error and specify where the error occurred.

The algorithm works as follows[17]:

All bits which are positioned on a slot which is a power of two are parity bits, all other slots are data bits. Thus, bit 1, 2, 4, 8 etc. parity bits. The parity bits will cover all data bits in positions which have the parity bit set. This means, if we have a 4 bit word covered by 3 parity bits, that parity bit P1 covers bits {3, 5, 7}, because 3, 5 and 7 have their least significant bit (LSB) set. The other parity bits cover the bits that have their second LSB, third LSB etc set. Resulting in: bit P2 covers bits {3,6,7} and bit P3 covers bits {5, 6, 7}. Table 4.1 shows us the positions of data bits D and parity bits P.

D4	D3	D2	P3	D1	P2	P1
1	0	1		1		
0111	0110	0101	0100	0011	0010	0001

Table 4.1: *Hamming code table for 4 data bits(D) and 3 parity bits (P)*

Finally the parity bits have to be set. This can be done by using even or uneven parity. For now, let's assume even parity. As mentioned P1 covers bits {3, 5, 7}. The values of these bits are {1, 1, 1}, therefor to keep an even parity P1 has to be set to 1. P2 covers bits {3, 6, 7} which have the values of {1, 0, 1}. To adhere to even parity, P2 has to be set to 0. Finally, P3 covers bits {5, 6, 7}, which have values of {1, 0, 1}. This means that P3 also has to be set to 0. Table 4.2 shows us the final encoded word 1011.

D4	D3	D2	P3	D1	P2	P1
1	0	1	0	1	0	1
0111	0110	0101	0100	0011	0010	0001

Table 4.2: *Data bits 1011 encoded with Hamming code*

Using Hamming encoding it is possible to detect up to two errors and correct one error. The encoding might seem expensive when used on smaller

word sizes as it adds a lot of overhead to the code word. The example given in table 4.2 shows that for four data bits, three parity bits are needed. However the more data bits the code word has, the less expensive the encoding technique is; parity bits are only added on positions of powers of two.

In general, m number of parity bits can protect the bits from 1 up to $2^m - 1$. However, the $2^m - 1$ number of bits include the m number of parity bits. Therefor, if we subtract the m bits from the $2^m - 1$ bits, we are left with $2^m - 1 - m$ number of data bits protected by the m parity bits. Table 4.3 shows the number of data bits in relation to the number of parity bits. As can be observed, the more data bits a word has, the smaller the overhead of the added parity bits is.

Using Hamming encoding on the state signal of a finite state machine might introduce overhead. This overhead stems from the extra bits added to the state state signal. Due to the extra bits, the next state logic of the finite state machine might become more complex. Thus increasing the area of the final design. However, the more states the original finite state machine has, the fewer parity bits it needs. Therefor the increase in overhead is likely to slow down as the size of the finite state machine increases.

Table 4.3: *Number of parity bits and the maximum number of data bits it can cover*

Parity bits	Data bits
2	1
3	4
4	11
5	26
6	57
7	120
8	247
9	502
10	1013

The method presented by [13] is used as a guideline on how to apply Hamming encoding to the benchmarks in this project. Chapter 6 aims to provide detailed information on how this method is implemented into Yosys.

4.1.4 BCH encoding

BCH encoding is a class of error correcting codes. Its name is an abbreviation of the inventors of the codes, namely: Bose, Chaudhuri and Hoc-

quenghem. The codes in the class of BCH codes can be configured in many ways. Using different configurations, the user of the code can control the amount of data bits, the amount of parity bits and the error correcting capability.

Within this thesis we use a BCH codes with configuration (15,7,5). This configuration has a total length of 15 bits; 7 data bits and 8 redundant bits. Its error correcting capability is 2 bit errors in the code word.

Chapter 5

Implementation in Yosys

This chapter aims to present the implementation of the Hamming and BCH encoding in Yosys. First an introduction to Yosys is given. Afterwards, a general overview of the implementation of the two encoding techniques is presented.

5.1 Yosys

To implement new encoding techniques into the state encoding of FSMs used in synthesis, a synthesis tool is needed. Rather than using the synthesis tools from well known vendors such as Xilinx, we choose to use an open source tool named Yosys. The software being open source makes it easier to add new functionality to it and it comes with no costs.

Yosys is developed by Clifford Wolf as part of his bachelor thesis in 2013. It was chosen that the tool would only work on Verilog source code. As of today, the Yosys can target CE40 FPGAs, Xilinx 7-Series FPGAs, Silego GreenPAK4 devices, and Gowinsemi GW1N/GW2A FPGAs.

The tool is written in C++ and is build up in different passes. It uses UC-Berkeley's ABC for the Sequential Logic Synthesis [18]. Each pass handles one part of the synthesis flow. It is possible to add new passes to Yosys or change existing ones in order to add more functionality. The user can input different commands to go through the synthesis flow step-by-step. Each step has its own output which can be logged into a log file. An example synthesis flow is shown in listing 4.1.

This thesis will focus mostly on the `fsm` pass in the script presented in listing 4.1. With the `fsm` pass all procedures related to finite state machines are called. These procedures consist of [19]:

- `fsm_detect` - This pass detects finite state machines by identifying the state signal.
- `fsm_extract` - This pass operates on all signals marked as FSM state signals. It consumes the logic that creates the state signal and uses the state signal to generate a control signal and replaces it with an FSM cell.
- `fsm_opt` - This pass optimizes FSM cells. It detects which output signals are actually not used and removes them from the FSM.
- `fsm_expand` - This pass is conservative about the cells that belong to a finite state machine. This pass can be used to merge additional auxiliary gates into the finite state machine.
- `fsm_recode` - This pass reassigns the state encodings for the FSM cells. At the moment only binary encoding and one-hot encoding are supported.
- `fsm_info` - This pass dumps all internal information on FSM cells.
- `fsm_export` - This pass creates a KISS2 file for every selected FSM.
- `fsm_map` - This pass translates FSM cells to flip-flops and logic.

Recently two bachelor students from the Technische Universiteit Eindhoven have been able to add FSM optimisations to the Yosys tool[20, 21]. State-space minimisation, state-space partitioning and the grey encoding have been implemented. These accomplishments show that Yosys is highly customisable and that the knowledge is there to further update or extent Yosys' capabilities.

5.2 Implementation

This section provides detailed information on the implementation of Hamming encoding and BCH encoding into Yosys. Both of the encodings use the same general principle. This general principle is discussed in this section.

5.2.1 The FSM pass

The Yosys source code can be split up into different parts: The frontend, the kernel, the passes and the backend. The frontend is the first step of the synthesis flow. It takes the design source code in a hardware description language (HDL), such as Verilog, as input and compiles this to an intermediate language. This intermediate language (IL) is part of Yosys' kernel and describes designs, cells and signals in code. Once a design has been compiled to the IL, the user can call different passes to further synthesise the design. Finally, the backend of Yosys can be used to write output in forms of netlists, intermediate language or JSON.

The `fsm` pass takes care of all FSM related operations. First it tries to detect and extract a FSM from the input design. If instructed Yosys can recode the FSM's states using different types of encoding techniques. Finally, the FSM can be mapped to internal cells or written to KISS2 for analysis.

During the detection and extraction of the FSM, Yosys constructs a data structure called `FsmData`. This data structure holds all the relevant data of the found state machine, such as: the number of states, the number of state bits, the state table and the transition table. Transitions in the transition table are a data structure on its own. This structure consists of: a state_in, state_out, ctrl_in and ctrl_out.

With these core structures the FSM can be changed and recoded. Once the FSM has been recoded, it will be copied to internal cells to proceed with the rest of the synthesis flow.

5.2.2 Recoding the FSM

To implement Hamming and BCH encoding, the method presented by [13] is used. As previously described, this method aims to add companion states to the original FSM. These companion states are copies of the original states, but with a bit error in the encoding. Each of the companion states has a transition to the next state of the original state.

Two extra C++ classes have been added to the Yosys source code. These are: `HammingEncoder` and `BchEncoder`. Both of these classes implement the functions:

- `encode(RTLIL::Const)`: Function which takes a state signal in the form of a `RTLIL::Const`, and encodes this state signal with the appropriate coding technique; either Hamming or BCH.
- `updateTransitionTableForState(RTLIL::Const)`: Function that updates the transition table in the `FsmData` structure. After encoding the state signals, the transition structures in the transition table still have the old state signals. Therefore the transitions need to be updated with the newly encoded state signals.
- `addCompanionStates(RTLIL::Const)`: This function, takes a state from the state table of the `FsmData` structure. It then loops over each bit and flips it, each time creating a companion state. Each new companion state is added to the state table.
- `findOutgoingTransitionsForState(FsmData)`: Function that finds all outgoing transitions from a state to any other state, except itself. This function is called by `addCompanionStates(RTLIL::Const)`, because the added companion states need transitions to the next state of their original state.

The two described encoder classes are used within the `fsm_recode` pass of Yosys. To add the two encoding techniques, we added the options to the encoding option list of Yosys. Finally, one can use the techniques by calling `fsm_recode -encoding hamming` or `fsm_recode -encoding bch`.

5.3 Validation

The changes made to the FSM during the `fsm_recode` pass have to be validated, as bugs in the `fsm_recode` pass might alter the functional behaviour of the designs. Unfortunately, we were not able to run simulations of the benchmark designs. Therefor validation was done through checking the changes to the FSM. A small example was used to analyze and verify these changes.

Listing 5.1: *Verilog source code for the example FSM implementation.*

```
1  module example(input clk, rst, ctrl, output [3:0] O);
2      reg [1:0] state;
3      always @(posedge clk) begin
4          O <= 0;
5          if (rst) begin
6              state <= 0;
7          end else case (state)
8              0: begin
9                  if(ctrl == 1'b1) begin
10                     state <= 1;
11                     O <= 1;
12                 end else
13                     O <= 0;
14             end
15             1: begin
16                 if(ctrl == 1'b1) begin
17                    O <= 1;
18                    state <= 0;
19                 end
20                 else
21                     O <= 0;
22             end
23         endcase
24     end
25 endmodule
```

The example used was a two state FSM of which the source code is presented in listing 5.1. First, the example design was synthesised with auto-encoding by Yosys. Afterwards, the design was synthesised using the Hamming encoding. During both of these runs the output of `fsm_info` was saved and the FSM was exported using `fsm_export`. Finally, the output of both runs was analysed to verify the changes made by `fsm_recode`.

The first run resulted in the FSM shown in figure 5.1. As can be observed, the FSM consists of two states. A transition to another state is made when a 1 is on the input, else the machines stays in the current state.

The second run used hamming encoding on the state bits. The two states of the example FSM have representations 0 and 1. Applying hamming encoding to both of these representations results in 000 and 111 respectively. Additionally, companion states for each state are added. This results in the addition of the following states: 001, 010, 100 for state 000 and 110, 101, 011 for state 111. Finally, each of the companion states have the same outgoing transitions as its base state. In conclusion,

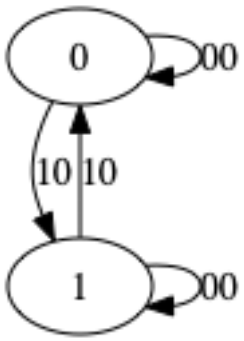

Figure 5.1: *FSM of the example design with auto encoding*

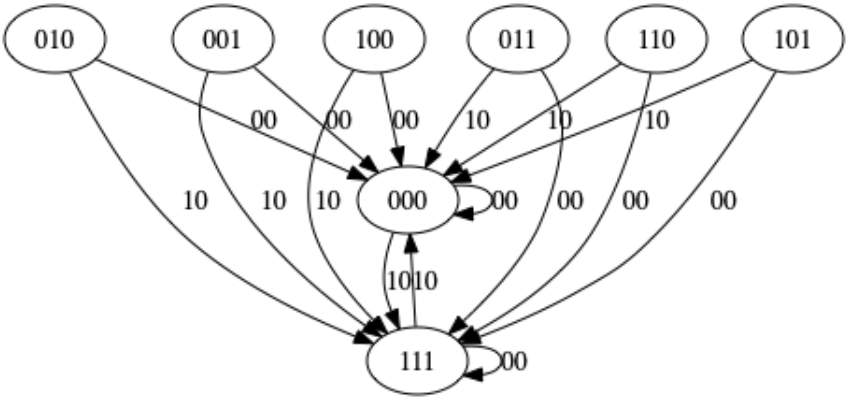

Figure 5.2: *FSM of the example design with Hamming encoding*

the hamming encoded example FSM should contain 8 states: 000, 111, 001, 010, 100, 110, 101, 011. It should also have 16 transitions as each state has two out going transitions.

The final FSM of the example design encoded with Hamming is shown in figure 5.2. As can be observed, the FSM contains everything which was anticipated. We therefor conclude that the implementation contains no bugs.

Chapter 6

Results

This chapter presents the results of the research done during this project. In the first section the area metric is discussed, after which the timing metric is discussed. Finally, a theoretical analysis of the error correction rate is presented.

6.1 Area

This section presents the results of the area measurements for each fault tolerant technique. Each fault tolerant techniques has been applied to all benchmarks designs. The area measurements are done using Yosys' `stat` command, after each design was synthesised.

Figure 6.1 shows the results of the area measurements. As shown in the figure, the area results of Hamming, TMR and Full TMR area are presented. All of the area measurements are presented relative to the baseline area; this factor is represented by the y-axis. The x-axis represents each design of the benchmark set. The number in parentheses following the name of the benchmark, represents the number of states of the designs FSM.

The area measurements for Hamming are obtained by synthesising the benchmark designs using Yosys' `fsm_recode -encoding hamming` command. The TMR area measurements are obtained by replacing each D-flip-flop cell for the TMR cell described in this thesis. Finally, the Full TMR results are done by calculating the total area when Full TMR would have been used. As explained in this thesis, Full TMR triplicates every part of the design and adds voters to handle majority voting. We therefore, multiplied the baseline area by three and added the total voter area to that number.

The BCH area measurements are not presented as part of the benchmark set was not able to be synthesised using this technique. BCH adds so many redundant states to the FSM, that Yosys would fail to do synthesis. Only the design with smaller state machines (up to 15 states) could be synthesised. These designs had on average an increased total area of 38 times the baseline area.

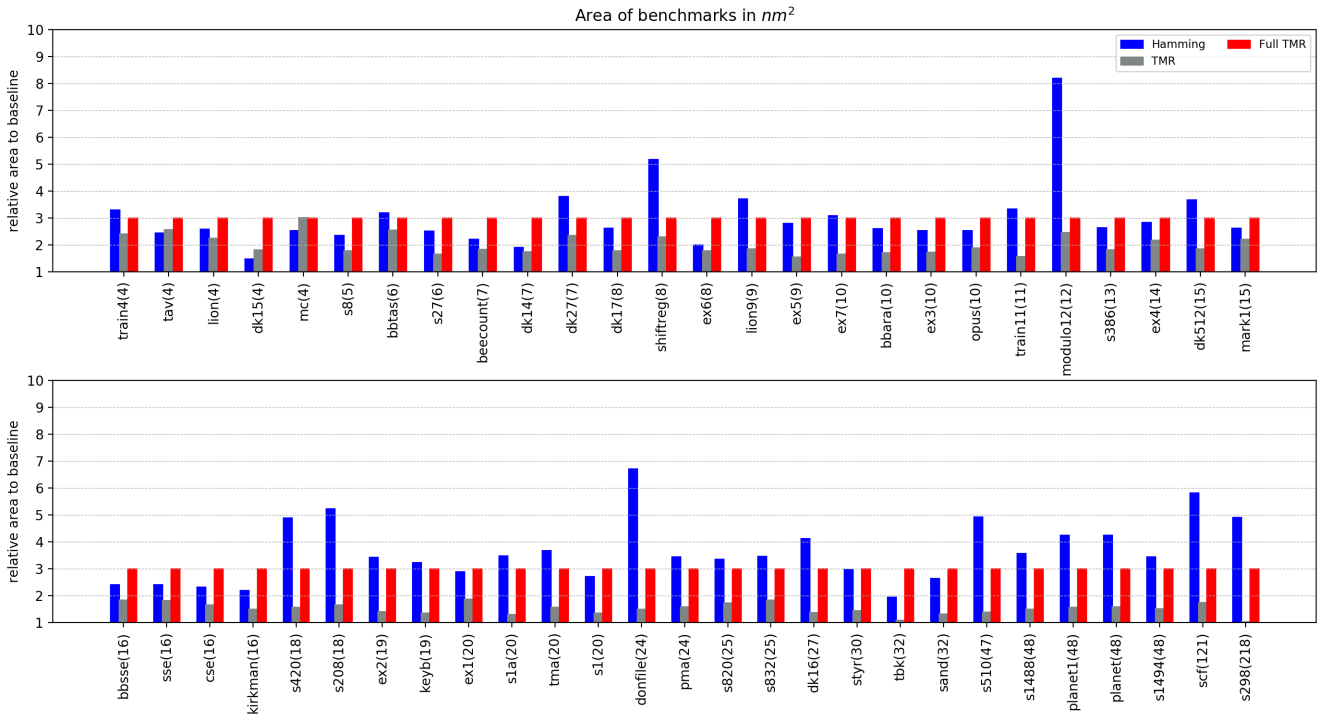

Figure 6.1: *Relative circuit area of the benchmarks with different fault tolerant techniques: Hamming, TMR and Full TMR. The number in parentheses following the benchmark name, indicates the number of states for that state machine.*

6.2 Timing

This section presents the results of the timing analyses of the synthesised benchmarks. These results are obtained by running Cadence Genus on the synthesised designs. The timing analysis is done for the baseline implementation, Hamming and TMR. It was not possible to do a timing analysis on the Full TMR technique. The reason being, that this technique was not synthesised, the area measurement was purely an on-paper approach.

Figure 6.2 shows the results of the timing analyses done by Cadence Genus. The bar graph shows the baseline implementation next to Hamming and TMR. The y-axis represents the clock period of the design in pico-seconds. The x-axis represents each of the design of the benchmark set.

The results are obtained by using the synthesised designs as input for Cadence

Genus. A clock with a period of 10000 pico-seconds was defined. Cadence Genus resolved a maximum of 10 critical paths per design. The first critical path that met the timing constraint was taken from the report. The tool reports the slack of the clock signal. To obtain the clock period as presented in Figure 6.2, the slack was subtracted of initial clock period of 10000 pico-seconds:

$$clock\_period = 10000 - slack$$

Note that these clock periods could potentially be even lower. Cadence Genus does a static timing analysis. This means the tool does not move any cells around to try and lower the clock period. It simply looks at the given cells in the design, and finds the critical paths.

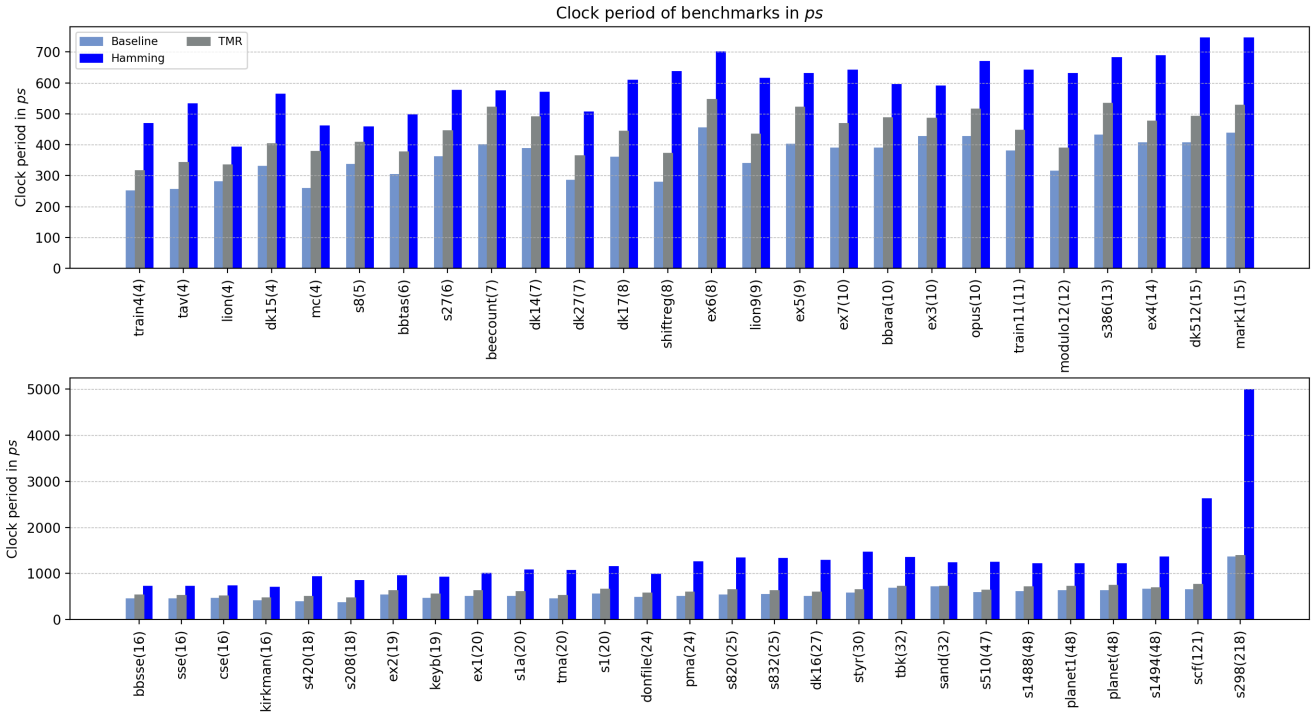

Figure 6.2: *Clock period of the benchmarks with different fault tolerant techniques: Hamming, TMR and Full TMR. The number in parentheses following the benchmark name, indicates the number of states for that state machine.*

6.3 Error correction capability

This section presents a theoretical analysis of the error correction rate of the three fault tolerant techniques: Hamming, BCH and TMR. Per technique the potential error correction rate is discussed. First Hamming and BCH are discussed, after which the TMR technique is analysed.

6.3.1 Hamming and BCH

The potential error correction rates of the Hamming encoding is bound by the Hamming bound. This bound tells us that, for a given Hamming distance there is a maximum number of errors it can correct. The Hamming distance is the number of positions at which two code words of a same length differ. The Hamming bound is defined as follows:

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor \tag{6.1}$$

Where t is the maximum number of errors that can be corrected and d is the Hamming distance.

The Hamming encoding used in this thesis, has a Hamming distance of 3. Therefor its error correcting capability is a single bit error in the code word.

The BCH encoding used in this thesis had a configuration of (15,7). This means the encoded code word has a length of 15 bits, and contains 7 information bits. The error correcting capability of this code is $t = 2$. Therefor, it can correct up to 2 errors in the code word.

6.3.2 TMR

The error correcting capability of TMR is not as 'static' as the error correcting capabilities of Hamming and BCH. With TMR, each flip-flip of the design is triplicated. The three flip-flops are connected to a majority voter. This way a TMR cell can correct up to a single bit error. However, this holds for only one TMR cell. As the number of TMR cells increases the number of potential corrected errors also increases.

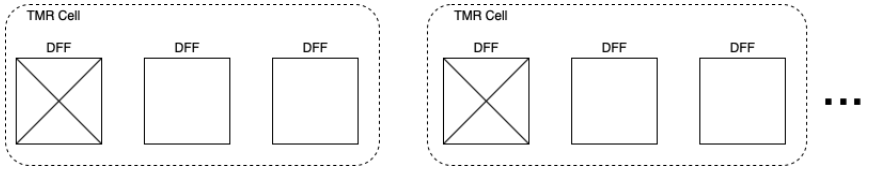

Figure 6.3: *TMR Cell with the best possible distribution of errors.*

Figure 6.3 shows two TMR cells, each with three D-flip-flops inside. As long as only one error occurs per TMR cell, each error is corrected by the majority

voter. However, there is an upper limit to this, bound by the pigeon hole principle. This principle states that if there is n amount of items to be distributed over m containers, and it holds that $n > m$. Then there has to be at least one container that holds two items of n.

The pigeon hole principle also applies to the TMR cells in our context. A TMR cell implementation of n cells, can at most protect the design against m errors, where it always holds that $m \leq n$. For any case of $m > n$, it must hold that there is at least two errors in one TMR cell, thus the cell fails to correct error. Therefor any design that implements n TMR cells on the flip-flops holding state signals, can correct at most up to n errors.

Up to the maximum number of errors, there is a probability that the next error occurs in a flip-flop of a TMR cell that does not have an error yet. This probability can be calculated as follows:

$$\prod_{i=1}^{x} p(i) \tag{6.2}$$

with $p(i)$ defined as:

$$p(i) = 1 - \frac{3(i-1)}{3n} \tag{6.3}$$

Where i is the number of errors that occur and n is the number of flip-flops in the design.

For example, let's assume there is a design with 3 flip-flops and we want to know what the probability is of the second error being corrected. We thus have $i = 2$ and $n = 3$. The probability of the first error being corrected is:

$$p(1) = 1 - \frac{3(1-1)}{3*3} = 1 - \frac{0}{9} = 1 \tag{6.4}$$

Thus the probability of the first error being corrected 1. This is shown in 6.4. As none of the TMR cells contain an error yet, it does not matter in which flip-flop an error occurs.

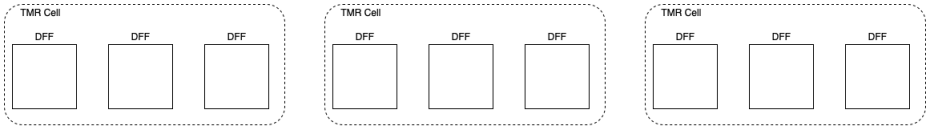

Figure 6.4: *The chance of the first error being corrected is 1, as no TMR cell contains an error yet.*

The probability of the second error being corrected is equal to:

$$p(2) = 1 - \frac{3(2-1)}{3*3} = 1 - \frac{3}{9} = \frac{6}{9} \tag{6.5}$$

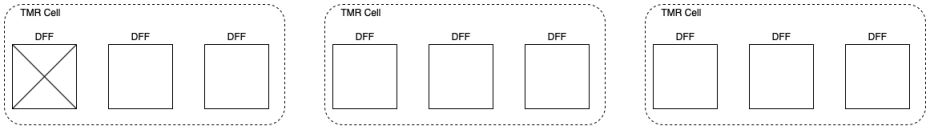

Figure 6.5: *The chance of the second error being corrected is $\frac{6}{9}$ as it can only be corrected if it falls into 6 of the 9 possible DFFs*

A visual representation is shown in figure 6.5. As shown in the figure, the first error can be placed in any of the TMR cells. The second error can only occur, in either two of the other TMR cells. Resulting in a probability of $\frac{6}{9}$. This is also the outcome of equation 6.2.

Finally, the overall behaviour of the error correcting capability of different the different fault tolerant techniques is presented in figures: 6.6, 6.7 and 6.8. The error correcting capabilities of Hamming and BCH are static. These fault tolerant techniques can only correct a static number of faults, independent of the number of state bits. In our case, Hamming can correct 1 fault, and BCH can correct up to 2 faults. This behaviour is clearly visible in figures 6.6 and 6.7.

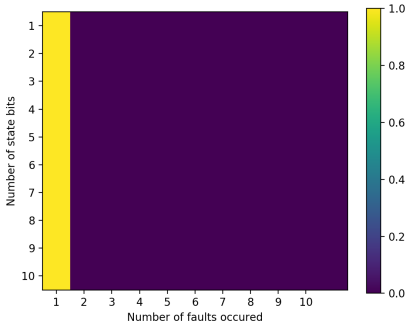

Figure 6.6: *Heat map of the fault correction probability of Hamming encoding.*

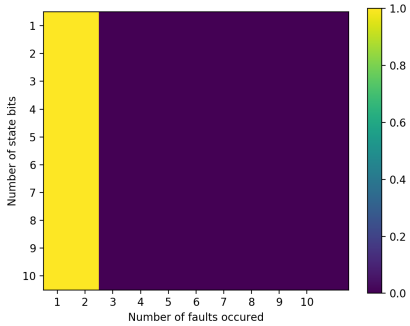

Figure 6.7: *Heat map of the fault correction probability of BCH encoding.*

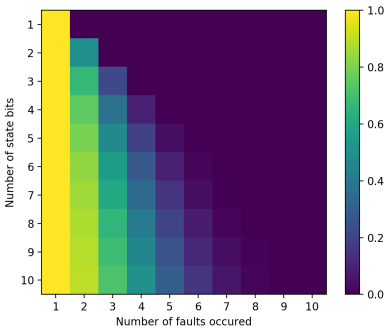

Figure 6.8: *Heat map of the fault correction probability of TMR.*

As previously mentioned, TMR could correct as many faults as it has TMR cells; however the error correcting probability decreases per consecutive fault. This behaviour is shown in 6.8, a design with 1 state bit, can correct only 1 fault. The design with 10 state bits, can correct 10 error, but each consecutive error has a lower probability of being corrected.

6.4 Discussion

This section aims to discuss the results of this project and compare them to a similar research[13].

The results of [13] are shown in figure 6.9. The graph shows three fault tolerant techniques: Binary, Hamming-3 and TMR. The first two techniques are encoding techniques on the state signals, whereas TMR is a form of spacial redundancy. Note that binary encoding has no form of fault tolerance, and can thus be seen as the baseline implementation. The graph presents the cross-section of the reference design on the vertical axis. The horizontal axis represents the clock frequency at which the reference design has been run.

The graph clearly shows that binary encoding has to highest cross-section of the three techniques. This is followed by the Hamming-3 encoding. Finally, TMR has the lowest cross-section and therefore provides the highest fault tolerance. These results are similar to the ones presented in our theoretical analysis of the error correcting capability.

Both the cross-section of Hamming-3 and TMR change as the clock frequency increases. The binary encoding cross-section remains almost the same. This seems trivial, as this encoding technique provides no fault tolerance. TMR's cross-section seems to change linearly as the clock frequency increases; however it can be observed that only two measurements were done. We can therefor not conclude anything about the behaviour of TMR's cross-section as the clock frequency increases.

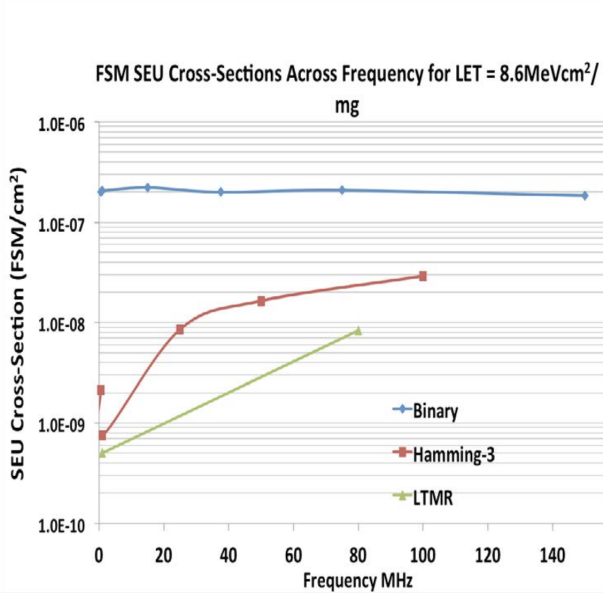

Figure 6.9: *Results of cross-section measurements of three fault tolerant techniques: Binary, Hamming-3 and TMR.[13]*

From our experiments we know the maximum clock frequency of the designs. As observed from the graph, it seems that Hamming can have a higher maximum clock frequency than TMR. This differs from the our results; however as mentioned, the TMR design was only measured by [13] at two frequencies. It might be that it was not tested on a higher frequency.

Chapter 7

Conclusion

This thesis presented an analysis of different fault tolerant techniques applied to FSMs in hardware designs. The fault tolerant techniques used, are: (Full) TMR, Hamming encoding and BCH encoding. The fault tolerant techniques have been applied to different benchmark designs. After implementation the designs have been synthesised and a timing analysis has been done. The results of these experiments have been shown in the previous chapter. This chapter aims to summarize the results, to draw conclusions and discusses future work.

In terms of area the different fault tolerant techniques were analysed against the baseline implementation. The results of the area analysis were shown in figure 6.1. On average the Hamming encoding resulted in an increase in total area by a factor 3.8 relative to the baseline. The TMR implementation had an increase in total area by a factor of 1.6 relative to the baseline. Finally, the Full TMR had an increase in total area by a factor of 3, as the complete design was triplicated. Although on average Hamming encoding resulted in bigger increase than Full TMR, there are cases in which Hamming resulted in a smaller total area. A possible explanation for this, is that these designs might consist of fewer flip-flops and simple next-state logic. These designs therefor do not increase as much, as Hamming only increases the area in terms of next-state logic.

The timing analysis was done for the baseline implementation, Hamming encoding and the simple TMR technique. The results of this analysis were presented in figure 6.2. The timing results were presented as the length of clock periods of the designs. On average the clock periods of the Hamming encoding were $441.6ps$ longer than the baseline clock periods. The average baseline clock period is $437.9ps$, thus designs encoded with Hamming tend to be twice as slow as the baseline designs. The average increase in the length of clock period for TMR is $86.9ps$. This increase is purely due to the voter being added, as TMR did not introduce anything else than the voter.

The error correction capabilities have been theoretically analysed with an on-paper approach. For Hamming it holds that it can only correct single bit errors, whereas the BCH encoding can correct up to 2 bit errors. It has been shown that

TMR can correct as many errors as there are TMR cells, if each subsequent error occurs in a separate cell. However the TMR analysis does not take the placement of the flip-flops into account. As technology size decreases, it is possible that two flip-flops are hit by the same SEU. If these two flip-flops are part of the same TMR cell, an immediate error occurs.

7.1 Future work

There is still much that can be done on the analysis of fault tolerant techniques for FSMs in hardware designs. This thesis analysed three techniques, but other interesting question or methods arrised during the time of this research. A list of possible future work is presented in this section:

1. **Timing of Full TMR**: During this research the area analysis for Full TMR has been done by multiplying the area of the baseline by 3. Because of this simplification the Full TMR design has never been implemented and synthesised. Therefor its timing analysis was not possible. Recently [12] released an open source tool, named TMRG. This tool takes an Verilog design as input and triplicates the complete design using Full TMR. Future work might use this tool to apply Full TMR on the benchmark set and synthesise it. Finally the timing analysis can be done to investigate the added overhead on the clock period.
2. **Investigate different ECCs**: The ECC used in this research for multibit error correction, is BCH. As shown in this thesis, the overhead introduced by this ECC was so significant that most of the benchmarks could not be synthesised. The research of different ECCs was out of scope for this project, but might be interested for further investigation.
3. **Placement of flip-flops**: As mentioned, the error correcting capability of TMR in this thesis is presented. However this approach does not factor in the placement of the flip-flops. As technology size decreases, one SEU might introduce errors in multiple flip-flops at once. Therefor it could be interesting to investigate the effect of the placement of flip-flops on the error correcting capability of TMR.
4. **Analysis of Hamming + One-hot encoding**: One-hot encoding makes sure that only one bit of a word is active at the time. If one would apply Hamming encoding to that word, only the parity bits and the active state bit are needed for decoding and error correction. This could possibly lower the amount of state bits needed.

Bibliography

[1] Nhon Quach, "High availability and reliability in the itanium processor," *IEEE Micro*, vol. 20, pp. 61–69, Sep. 2000.

[2] L. Ciani and M. Catelani, "A fault tolerant architecture to avoid the effects of single event upset (seu) in avionics applications," *Measurement*, vol. 54, pp. 256 – 263, 2014.

[3] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, "Fault-tolerant platforms for automotive safety-critical applications," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pp. 170–177, 2003.

[4] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[5] "The Cambridge English Dictionary." `https://dictionary.cambridge.org/dictionary/english/dependability`. Accessed: 22-09-2019.

[6] A. Avizienis and J. Laprie, "Dependable computing: From concepts to design diversity," *Proc. IEEE, Vol. 74, No. 5, pp. 629-638*, May 1986.

[7] ECSS, *Space product assurance*. ESA Requirements and Standards Division, 2016.

[8] J. Hartmanis, *The Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Inc., 1966.

[9] A. Avizienis, "Fault-tolerance: The survival attribute of digital systems," *Proceedings of the IEEE, vol. 66, no. 10*, 1978.

[10] M. A. Check and T. J. Slegel, "Custom s/390 g5 and g6 microprocessors," *IBM Journal of Research and Development*, vol. 43, pp. 671–680, Sep. 1999.

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

[12] "How to design highly reliable digital electronics." `https://media.ccc.de/v/36c3-10575-how_to_design_highly_reliable_digital_electronics`. Accessed: 14-01-2019.

[13] M. Berg, "An analysis of heavy-ion single event effects for a variety of finite state-machine mitigation strategies," Single Event Effects (SEE) Symposium and the Military and Aerospace Programmable Logic Devices (MAPLD) Workshop, La Jolla, CA, May 19-22, 2014, 2014.

[14] "ERNINAN: Error injection and error propagation analysis in (standard cell) netlists.." `https://git.ics.ele.tue.nl/jhuisken/erinan`. Accessed on: 15-03-2020.

[15] "Freepdk45 website." `https://research.ece.ncsu.edu/eda/freepdk/freepdk45/`. Accessed: 09-01-2019.

[16] "Yosys open synthesis suite." `http://www.clifford.at/yosys/about.html`. Accessed: 02-06-2019.

[17] L. van Moergestel, *Computersystemen en Embedded Systemen*. Sdu Uitgevers bv, 2012.

[18] "ABC: A system for sequential synthesis and verification." `https://people.eecs.berkeley.edu/~alanmi/abc/`. Accessed on: 02-06-2019.

[19] "Yosys manual." `http://www.clifford.at/yosys/files/yosys_manual.pdf`. Accessed: 02-06-2019.

[20] M. S. Visser, "Automatic optimization of finite state machine encodings in yosys," 2019.

[21] V. van de Schaft, "Finite state machine state space reduction algorithms for hardware synthesis in yosys," 2019.