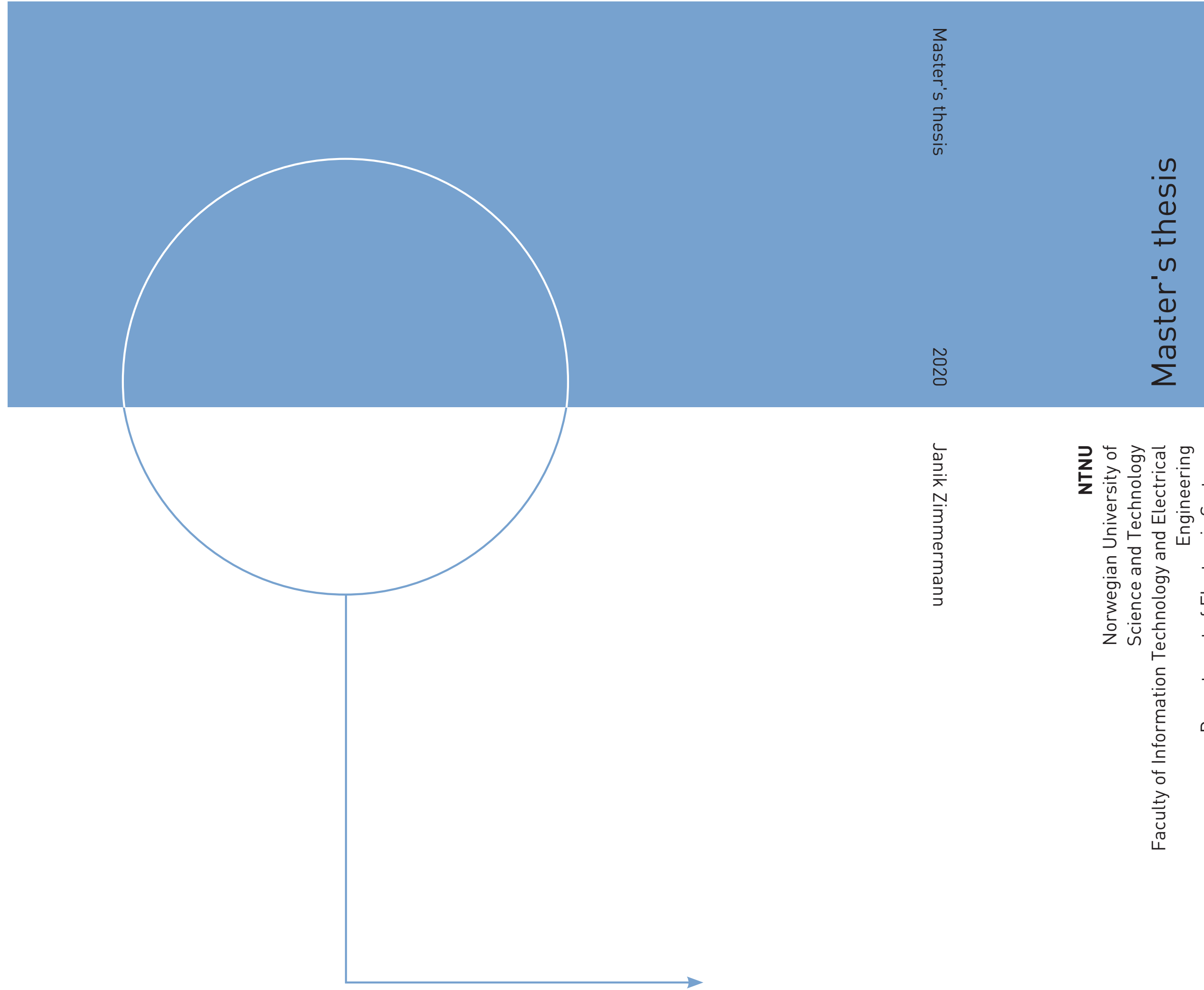

Janik Zimmermann

An exploration of options to increase functional safety in the AVR core

July 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

An exploration of options to increase functional safety in the AVR core

Janik Zimmermann

Electronics System Design and Innovation
Submission date: July 2020
Supervisor: Bjørn B. Larsen
Co-supervisor: Vitaly Marchuk

Norwegian University of Science and Technology
Department of Electronic Systems

Abstract

This thesis aims to select cost effective measures to modify a Microchip micro-controller, in order to make it comply with functional safety requirements set by the automobile industry.

As micro-controllers are used for increasingly safety critical tasks in road vehicles, mitigating the consequences of these components breaking down becomes a major focus in their design. Detecting and addressing internal faults has the potential to save multiple lives.

The main objective of this thesis is getting the AVR processor, a component on many Microchip micro-controllers, to comply with the ASIL-B safety integrity level, as defined by the ISO 26262 international standard for functional safety in road vehicles.

Well known approaches for achieving functional safety in electronic components were compared, and one was selected based on an analysis of the costs and benefits associated with each. The selected approach, duplicating the CPU, was implemented and tested to estimate its impact on performance, production and operating cost as well as verify the level of functional safety provided by the approach.

Duplicating the CPU lead to the detection of 47.3% of all single bit stuck-at-faults injected into the CPU. While this number seems low, the argument of this thesis is that it does not actually indicate poor functional safety. It was determined that this figure is a function of the program running on the duplicated CPUs, as the program determines which faults propagate out of the CPU to be detected.

While this thesis covered fault detection in the CPU, this is only one component of many contained in the micro-controller. These remaining components must be addressed before the system as a whole can comply with the functional safety standard. Further, this thesis does not address the question of what to do if a fault is detected. The system must transition to a safe state, the nature of which should be defined in collaboration with customers.

Sammendrag

Målet med denne oppgaven er å velge kostnadseffektive metoder for å endre en Microchip-mikrokontroller slik at den følger standardene for *functional safety* som har blitt satt av bilindustrien.

Mikrokontrollere blir brukt i stadig mer sikkerhetskritiske oppgaver i kjøretøy. Derfor har minsking av konsekvensene av at disse komponentene bryter sammen blitt et hovedfokus i deres design. Man kan potensielt redde flere liv ved å detektere og ta hånd om interne feil.

Denne oppgavens hovedmål er å sørge for at AVR-prosessoren, en komponent som brukes i mange av Microchips mikrokontrollere, oppnår sikkerhetsnivået ASIL-B, som definert i ISO-26262, den internasjonale standarden for *functional safety* i kjøretøy.

Velkjente metoder for å oppnå *functional safety* i elektroniske komponenter ble sammenliknet, og én av disse ble valgt basert på an analyse av kostnad og nytte assosiert med hver av metodene. Den valgte metoden, duplisering av CPU-en, ble implementert og testet for å estimere dens innvirkning på ytelse, produksjons- og operasjonskostnader. I tillegg ble metodens oppnådde nivå av *functional safety* verifisert.

Duplisering av CPU-en førte til en deteksjon av 47.3% av alle enbits *stuck-at-faults* som ble injisert i CPU-en. Selv om dette tallet virker lavt, viser denne oppgaven at det ikke er en indikasjon på dårlig *functional safety*. Det ble argumentert at dette tallet er en funksjon av programmet som kjører på de dupliserte CPU-ene, siden programmet dikterer hvilke feil som sendes ut av CPU-en og kan bli detektert.

Denne oppgaven handler om feildeteksjon på en CPU, som er én av mange komponenter i en mikrokontroller. De resterene komponentene må bli tatt hånd om før hele systemet kan følge standarden for *functional safety*. Oppgaven har ikke tatt opp spørsmålet om hva som skal gjøres når en feil har blitt detektert. Systemet må skifte til en trygg tilstand. Denne tilstanden må defineres i samarbeid med kunden.

Preface

This master thesis was written between January and July 2020, as a conclusion to the degree Master of Science in Electrical System Design and Innovation at NTNU Trondheim. The thesis is a collaboration with the private company Microchip, as it is common practice for engineering studets at NTNU to collaborate with local companies during academic projects.

Please note that this thesis presents a preliminary design by Microchip, and does not reflect the performance of any Microchip products.

The assignment was proposed by Vitaly Marchuk from Microchip and professor Kjetil Svarstad from NTNU. Sadly, Kjetil Svarstad passed shortly before work on this thesis was scheduled to start. Professor Bjørn Larsen replaced Kjetil Svarstad as the supervising professor before the start of this project.

Please note that the Corona virus had a significant impact on the progress of this thesis. It prevented me from accessing the Microchip offices, designs and equipment for a significant part of the process, forcing me to reconsider the scope of the project. It also meant that all meetings with my supervisors and colleagues had to happen online.

I would like to thank Vitaly Marchuk, Bjørn Larsen and Kjetil Svarstad for their contributions to this thesis. I also thank the Microchip employees Einar Fredriksen and Johannes Wågen who provided me with help and guidance, both in how to approach the thesis itself and in the use of the development tools.

Janik Zimmermann

July 2020, Trondheim

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 2

1.3 Outline 2

2 Background **4**

2.1 Functional safety and fault detection 4

2.2 The nature of faults 6

2.2.1 Systematic and random faults 6

2.2.2 Permanent, intermittent and transient faults 6

2.2.3 Fault models 7

2.2.4 Single-point and multi-point faults 7

2.3 The AVR core 7

2.4 The digital design flow 9

2.4.1 Implementation 9

2.4.2 Dynamic analysis 9

2.4.3 Static analysis 10
2.4.4 Synthesis 10
2.4.5 Place, clock tree synthesis and route 11
2.5 Estimating fault detection rates 11

3 Fault detection mechanisms 13

3.1 Self-testing routines 13
3.1.1 Software self-testing 16
3.1.2 Hardware aided self-testing 18
3.1.3 Hardware self-testing 19
3.2 Hardware redundancy 22
3.2.1 Full hardware duplication 22
3.2.2 Checksums 23
3.2.3 Combining full duplication and parity bits 25

4 Test and Results 30

4.1 Selecting approaches for implementation 30
4.2 Test method 31
4.2.1 Building the hardware duplication 31
4.2.2 Fault injection simulation 31
4.2.3 Synthesis and cost estimate 33
4.3 Results 33

5 Discussion 35

5.1 Objectives 35
5.2 Limitations 38

6 Conclusion 40

6.1 Main findings 40
6.2 Future work 41
6.2.1 Expanding to a system level solution 41
6.2.2 Functional safety concerns when placing, routing and generating a clock tree 41
6.2.3 Reducing the cost by defining a set of safe instructions 42
6.3 Closing words 42

A Synthesis area report 45

B Synthesis power consumption report 47

C Fault injection log 49

Chapter 1

Introduction

1.1 Motivation

In modern vehicles, safety critical systems are increasingly controlled digitally. A failure of these digital systems may in some cases have catastrophic consequences, such as multiple loss of life. Achieving functional safety means bringing the probability of such an event down to an acceptable level. Thus digital components, like those designed and manufactured by Microchip, must be designed to minimise the risk of unexpected behaviour.

The safety standard ISO 26262 Road vehicles - Functional Safety [1] was published in 2011, outlining safety standards for electrical/electronic systems for automotive applications. As a result, automotive manufacturers require their suppliers, including Microchip, to provide products which comply with this standard.

The safety standards, including ISO 26262, divide failures into two categories; systematic and random failures. Systematic failures are caused by mistakes in the definition and implementation by hardware engineers, or mistakes in the configuration of the system by the software engineer. Random failures are not consistently reproducible across multiple copies of the chip, as they result from damage, production process variations and environmental conditions with regards to one individual chip.

The design and production process is already designed to minimise the likelihood of systematic failure due to definition or implementation mistakes, or due to production process variations. Additionally, a large part of Microchip's resources are devoted to helping customers configure Microchip products correctly. However, hardware faults introduced into the system after it has been deployed, while unlikely, must be addressed before the system can meet the functional safety standard.

This thesis is a step towards a higher functional safety level in Microchip products, by detecting any random faults that may occur within the AVR CPU.

1.2 Objectives

An existing internal Microchip design will be modified to comply with functional safety standards. The following functional safety objectives were defined based on the functional safety standard ISO 26262 [1] certification ASIL B (automotive safety integrity level B) and requests from automotive customers.

1. Detect at least 90% of single point faults
2. Achieve a fault tolerant time interval of at most 10 milliseconds
3. Minimise the chance of introducing new systematic hardware failures
4. Minimise the chance of introducing new systematic failures in software, by maintaining the system's ease of use
5. Minimise both development and production cost of the system

1.3 Outline

Chapter 2 provides the necessary theoretical background to follow the rest of the thesis. Functional safety and fault detection, which are the main topics

of this thesis, are discussed in section 2.1. The nature of faults is covered in section 2.2, in which different types of faults and fault models are presented. Microchip's AVR core, the architecture used in this thesis, is presented in section 2.3. Then, a brief overview of the digital design flow is covered in section 2.4. Finally, different methods for evaluating fault detection solutions are presented in section 2.5.

Chapter 3 presents different fault detection approaches. The approaches are compared with respect to the safety related goals outlined in section 1.2, as well as their impact on production cost, performance and memory.

In chapter 4, one of the approaches presented in chapter 3 is tested on the CPU. The results are discussed in chapter 5. Each of the 5 objectives defined in section 1.2 will be assessed. Then the limitations of the chosen approach are discussed.

Chapter 2

Background

2.1 Functional safety and fault detection

Functional safety is defined as "the absence of unreasonable risk due to hazards caused by malfunctioning behaviour of electronic systems" by the ISO 26262 [1]. Absence of unreasonable risk means that the combination of the probability of the occurrence of a failure and the severity of the failures consequences is acceptable. There is always some probability for the occurrence of a critical failure, but by being aware of the potential failures and taking action to mitigate them, this risk can be brought to an acceptable level.

When a part of the system acts abnormally, that is a fault. After some time this fault may lead to a hazardous event, especially in safety critical systems like road vehicles. This is illustrated in figure 2.1. A fault that may lead to a hazardous event is a failure.

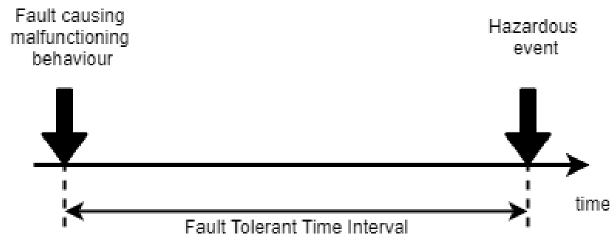

Figure 2.1: Showing a fault leading to a hazardous event or failure

The time interval between a fault and a possible hazardous event is the fault tolerant time interval. The goal of fault detection mechanisms, like a self-testing routine, is to detect and address the fault within the fault tolerant time interval. Addressing the fault happens in the form of entering a safe state, where the vehicle's safety no longer relies on the component. This process is shown in figure 2.2.

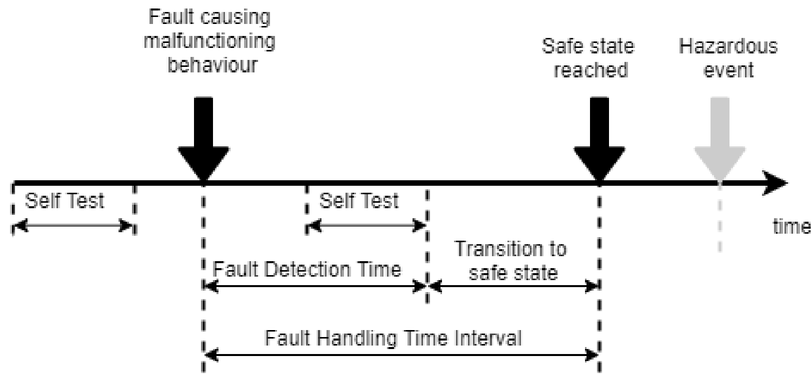

Figure 2.2: Showing a fault being detected by a self-testing routine before a hazardous event can occur.

How the transition to a safe state happens depends on the particular application. As an example, if the automatic steering system in a vehicle fails, it may inform the driver through an auditory signal, asking the driver to control the vehicle. How this is implemented ultimately depends on the engineer designing the vehicle, our customer.

2.2 The nature of faults

2.2.1 Systematic and random faults

Faults are split into systematic faults and random hardware faults. Systematic faults or "bugs" are reproducible on every copy of the chip and must be addressed through changes to the hardware or the documentation. Random hardware faults may happen, but are not consistently reproducible on other copies of the chip.

The hardware design process is carefully designed to minimise the probability of systematic faults and failures. By favouring a modular and simple design, the probability of introducing systematic hardware failures is minimised. Re-use of well-trusted hardware elements, interfaces and mechanisms for the detection and control of failures can further reduce the probability of systematic failures.

2.2.2 Permanent, intermittent and transient faults

Random hardware faults may happen during the lifetime of a hardware element and follow a probabilistic distribution. They may be permanent, intermittent or transient.

Permanent faults are permanent, even through resetting the system. They are the result of various types of physical damage within the chip. Oxide breakdown, electromigration, stress voiding and package damage are the main causes of permanent faults [2].

Intermittent faults occur from time to time and then disappear again. They are commonly caused by permanent damage to the chip or systematic errors like setup and hold violations or production process inaccuracies. They may depend on crosstalk, environmental conditions like temperature or supply voltage variations.

Transient faults are faults that occur once and subsequently disappear. They can be the result of unusual environmental conditions, electromagnetic radi-

ation or crosstalk.

2.2.3 Fault models

Permanent faults most commonly manifest themselves as a stuck-at-fault. In this fault model, the value of a wire is permanently set to 0 or 1 regardless of what it should be. It could be caused by transistor breakdown in the driving logic or short-circuiting of the data wire to either ground or the supply voltage.

Permanent faults may also be caused by the wrongful connection of two data wires within the system. This may be modelled in different ways, depending on the driving logic of the wires. Either one of the wires dominates the other and controls the value of both wires, or the connection results in a wire-and or wire-or behaviour.

Transient faults are defined as only occurring once - either changing a 0 to a 1 or vice versa. This may occur within combinatorial logic or at storage.

2.2.4 Single-point and multi-point faults

A single point failure is a failure resulting from a single hardware fault, while a multi-point failure results from multiple independent hardware faults.

If multiple faults are correlated, they have a common root cause. These faults are called common mode faults.

2.3 The AVR core

The AVR core is a 8 bit CPU using the AVR specific instruction set [3]. The main focus of the AVR core is to be cost effective, energy efficient and highly predictable. In line with these values the core uses no caches and a short instruction pipeline.

The main components of the CPU and the system as a whole are shown in figure 2.3. This figure and the explanation is based on the one given by Mazidi, Naimi and Naimi [4], and is amended by examining the source code of the design itself.

The main CPU components are a register file and an ALU for computation, a program counter including branch logic, and an instruction register. The program counter points at a location in the program memory. The instruction is loaded into the instruction register and broadcast throughout the CPU. Each CPU component receives the instruction, decodes it and acts accordingly.

Most of the system-on-chip's operation involves interfacing with its peripherals and the IO pins. This happens through control and status registers in each of the peripherals, which are mapped into the address space. The addressing is either direct, where the address is written into the instruction, or indirect, where the address is read from a set of predetermined CPU registers.

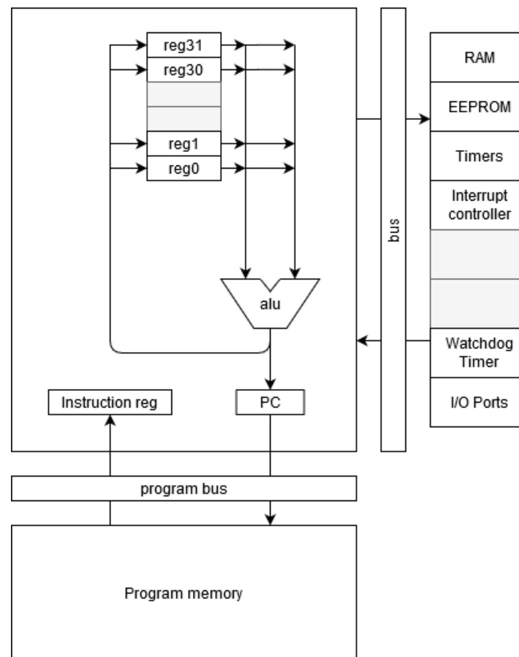

Figure 2.3: A simplified overview of the AVR CPU architecture and how it interfaces with the rest of the system [4].

Most instructions are single cycle, but there are many multi cycle instructions. Multi cycle instructions include the more complex arithmetic operations and program flow instructions like jumps, calls and branches. Any conditional branches use the ALU flags which were set during the previous arithmetic instruction. Most instructions are 16 bit instructions, while some multi cycle instructions use 32 bit.

2.4 The digital design flow

This chapter contains an overview of the digital design process at Microchip. It is based on the explanation given in my 2019 project [5].

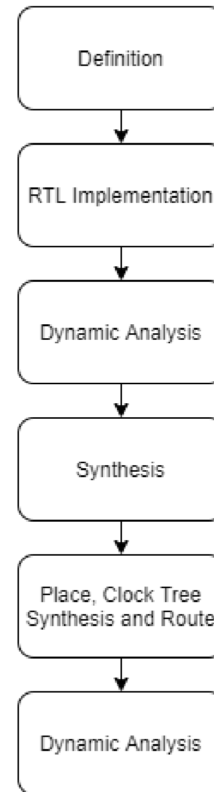

Figure 2.4: An overview of the standard digital design flow with an emphasis on the points that are relevant to the thesis.

2.4.1 Implementation

First, the desired circuit is specified in detail. Then it is implemented by engineers writing RTL code. This code is usually written in SystemVerilog. This is the stage at which hardware changes will be made in this thesis.

2.4.2 Dynamic analysis

During dynamic analysis, the circuit is simulated to verify its functionality. The words simulation and dynamic analysis will be used interchangeably. Stimulus is applied to the circuit and the effect on the rest of the circuit is captured. The stimulus is predefined, and the set of predefined stimuli is called a test. The test may be created by an engineer or randomised. The circuits used during this project contain a CPU. In such cases, the test often includes a program for the CPU to run.

These simulations are executed at various stages in the design process, as seen in figure 2.4. During and after implementation, the RTL code is simulated to verify that its behaviour is according to the specifications.

2.4.3 Static analysis

The alternative to dynamic analysis is static analysis. Here, statistics are propagated through the system in order to estimate the probability of a signal being 0 or 1. Take the example of a simple AND-gate. If both inputs are uncorrelated, and their values are 1 50% of the time, the output of the AND-gate is 1 25% of the time. This 25% value can then be used to estimate the probabilities for the next gate.

Static analysis is not very useful as a verification tool, and is rarely examined directly by an engineer. It is often used by tools to extract statistics about the system in order to estimate factors like power consumption and fault detection rates.

2.4.4 Synthesis

When the circuit seems to meet the specifications, the circuit is synthesised. The synthesis translates RTL code into logic cells. These logic cells are given by a standard cell library, and describe the building blocks that can be fabricated when creating the actual circuit. The output from this process is a netlist, which is written in Verilog. The tool used for synthesis is Synopsys Design Compiler [6].

During synthesis, optimisation is carried out to meet timing constraints, minimise area and power consumption. This involves removing redundant elements and simplifying logic. As functional safety solutions often involve deliberately introducing redundancy, some measures must be taken to ensure that these redundant elements are not optimised away.

Synopsis Formality is used after the synthesis process to verify that the netlist

is logically equivalent to the RTL code.

2.4.5 Place, clock tree synthesis and route

Next, the logic cells are placed and connected. This process is largely automated, and the result is a complete plan of the circuit. When this is done, wire lengths and the location of different elements are known. With this, the circuit can be simulated in higher detail, taking delays and capacitances into account. Timing is usually a major concern at this stage.

In order to spread the clock signal throughout the chip, a clock tree is constructed. The clock tree is constructed such that the distance the clock signal has to travel from its source to the various components is about the same. This ensures that all components receive the clock at approximately the same time, which is necessary to avoid setup and hold violations. The clock tree also contains gates, allowing the clock signal to be blocked. Blocking the clock signal from propagating into regions of the chip which do not currently need it can save significant amounts of power.

These tools are used to insert spare gates. These can be used to fix small mistakes made in the implementation, while only editing the metal layer. Any modification which can keep within this small budget is significantly cheaper to retrofit into existing devices than a modification requiring a full redesign of the chip layout and production.

2.5 Estimating fault detection rates

There are different methods for evaluating the fault detection solution. In this project, this will be done by using Z01X [7], a tool by Synopsys. This tool is based on dynamic analysis as described in chapter 2.4.2. It allows the engineer to run a large number of simulations, each with different faults injected, and evaluate their effect on the system. The goal is to quantify how many faults are detected, and how many go undetected.

An alternative tool would be TestMAX FuSa [8] by Synopsys. This tool is

based on static analysis, as described in chapter 2.4.3. The advantage of such an approach like this is that it is independent of the base simulation, and achieves faster runtime. The main disadvantages are that the statistics are not necessarily representative of how the system acts during typical operation. This tool was not available for this project, but may have been better suited.

The fault injection simulations could be run on RTL code or on a netlist. The netlist would yield more accurate results. However, it increases the complexity of the simulation considerably. When the RTL code is translated to a netlist, a large number of intermediate signals are created as the larger logic operations are broken down to simple logic gates. For example, when creating an adder ($A + B = C$) in RTL, there is little reason to instantiate more signals than the inputs (A and B) and output (C) of the adder. When breaking the adder down into smaller logic gates for a netlist, internal signals such as carry bits connecting single bit full adders are introduced. Losing these potential fault injection points reduces the accuracy of the result.

For these simulations, the engineer must specify where and how faults should be injected. Different fault models are available, as described in chapter 2.2.

When evaluating a hardware solution for detecting faults, the system must be simulated as described in chapter 2.4.2. To do this, a test needs to be selected. The test ultimately dictates the quality of the analysis, as a simple or repetitive test could result in many faults never propagating though the system. Faults that don't propagate through the system are not detectable.

When evaluating a software solution, the software would become the test. Here we wish for the software to find as many faults as possible, and we have no way of knowing if a undetected fault is harmless or not.

During the simulation, faults must be classified. First, a fault free version of the test is run. In subsequent runs, single faults are introduced to the system. Z10X allows the engineer to easily compare signals in a faulty system to the signals in the fault free system. Based on this, simple functions can be called to classify the fault. At the end of all fault injection simulations, the tool will provide statistics about how the faults were classified.

Chapter 3

Fault detection mechanisms

In this chapter, different potential approaches to detecting faults in the AVR core will be covered. The fault detection mechanisms will be compared in terms of how well they achieve the safety related objectives outlined in section 1.2.

The approaches can be divided into two main groups: self-testing routines and redundant hardware. Section 3.1 outlines self-testing routines, where the main program is paused periodically in order to test whether the chip is faulty. Section 3.2 outlines the use of hardware redundancy to detect internal faults without disrupting the main program.

3.1 Self-testing routines

During a self-test, the main program running on the chip halts temporarily to run a self-testing routine. The goal of the self-testing routine is to check for at least 90% of the potential faults in the system, in order to comply with objective 1. These tests need to be run often enough to achieve a fault tolerant time interval of at most 10ms, to comply with objective 2. How often exactly depends on how long a transition to a safe state would take.

Self-testing routines have a set of major disadvantages. The main program is

paused and it can only detect permanent faults. The nature of self-testing is such that it can only consistently find permanent faults. If a transient fault affects the main program, by its very definition it is gone by the time the self-testing routine is started. Interrupt priority during self-tests should also be considered. Leaving the main program at a time which is unpredictable to the software engineer also introduces a decent chance for bugs. This chance for bugs might be mitigated by having the customer activate the self-test, instead of starting it automatically.

The increase in power consumption due to self-testing routines is independent of the activity of the system while executing the main program. This means that in the context of a relatively low activity application, activity introduced by the self-test can lead to a significant increase in power consumption. On the other hand, if the circuit is relatively active, the increase due to the self-test becomes less significant. In high activity circuits, the performance consumed by the periodic self-test poses a challenge to the customer.

Because transient faults cannot be detected consistently through self-testing, they will not be included in this discussion. To simplify this discussion further, only stuck-at-faults will be considered. Other fault models, like bridging faults, are less common and are often caught by the same measures that identify stuck-at-faults.

In a self-test, we must ensure that no registers are stuck and that no wires in combinatorial logic are stuck. Registers are relatively easy: one simply must write a 0 and a 1 to them, to see if the registers hold values as expected. However, depending on the nature of the self-test, some registers like those holding the state of a state machine might be hard to target. Combinatorial logic is a more complex problem.

When testing combinatorial logic, one has to ensure that that every single bit stuck-at-fault has the chance to propagate to a point where it can be caught. Take the example of an AND gate. To test its functionality, we have the ability to apply signals to its two inputs, and observe signals at its output. To test it, one might start by applying 0 to both inputs, expecting a 0 at the output. However, this only tells us is that the output is not stuck at 1. Either input might be stuck at 1, but that would not be observed at the output while the other input is 0. If any signal is stuck at 0, this will not be detected, as a stuck-at-0 fault has no effect if the signal is already supposed

to be 0. Table 3.1 shows the three test patterns required to exhaustively test an AND gate for stuck-at-faults.

A	B	C	rules out
0	1	0	A-sa1, C-sa1
1	0	0	B-sa1, C-sa1
1	1	1	A-sa0, B-sa0, C-sa0

Table 3.1: The testing patterns required to rule out a stuck-at-fault in the and gate A*B=C

The combination of patterns, like the set shown in 3.1, is commonly generated by a tool. In practice, these patterns are applied to the system as shown in figure 3.1, they are loaded into the input register, the system is clocked once so the values propagate to the output register, and then they can be read and compared to the expected output. Testing by this principle is carried out in the factory, which is explained further in section 3.1.3. In such a factory test, over 3000 different patterns are used - but this catches all potential stuck-at-faults, whereas the goal for this project is to catch at least 90%.

Figure 3.1: Illustrations of how combinatorial blocks are tested. A pattern is applied to the input register, and the output register is read. The values from the output register are compared to the expected value.

Self-testing routines can be implemented purely in software, which is outlined in section 3.1.1. Adding hardware to improve the coverage provided by these tests is discussed in section 3.1.2.

Lastly, hardware self-testing routines are outlined in section 3.1.3. Here some or all of the self-testing routine is done through dedicated hardware, reducing or eliminating the need for a software based self-test.

3.1.1 Software self-testing

In software self-testing, we attempt to run tests like the one illustrated in figure 3.1 purely by using software. The main advantage of this approach is obvious: no hardware changes. This means that it can be included in existing products without any changes to the production line. However, designing a software self-test poses some challenges. As we are testing the system while it is running the self-test, many registers cannot be written to freely without breaking the test. Many registers are also not readable directly from software. This set of challenges means that a self-test has to be handcrafted to cover as much functionality as possible.

Handcrafting a self-test means abandoning the pattern based approach outlined earlier, and rather focusing on verifying the correct functionality of individual instructions. The AVR instruction set manual contains about 120 instructions. Each of these must be tested separately to ensure their proper operation.

Testing arithmetic instructions is straight forward. Test patterns are loaded into the register, the instruction is applied, a pattern containing the correct output is loaded and the actual output is compared to the correct output. This routine is illustrated in the pseudo-code below.

```
Load test_pattern_A
Load test_pattern_B
instruction actual_output, test_pattern_A, test_pattern_B
Load correct_output
compare correct_output, actual_output
branch_if_not_equal error_state
```

This pseudo-code, combined with the right set of test patterns, could verify the combinatorial logic used to compute the output of the arithmetic instruction, similar to the AND-gate described earlier. However, arithmetic instructions set flags based on the value of their output. These flags are used by a subsequent conditional branch instruction. Each branch instruction uses a specific flag.

To test the register file, one might cycle through the registers in subsequent

tests. This means that any data used by the main program needs to be moved to memory before running the software self-test, and loaded back after it is finished.

To test the program counter, the self-test should be spread throughout the memory space, ensuring that no program counter bit is stuck. This can be combined with tests targeting program flow operators like jump, branch, call and return.

This paragraph aims to make a rough estimate of the size of a self-testing program. Arithmetic instructions are tested through the pseudo-code above, which takes 6 clock cycles. Considering the bitwise AND instruction, the code has to be run 3 times, leading to a total of 18 clock cycles per instruction. If this is representative for all instructions, the self-test would take around 2150 clock cycles. If it is stored sequentially in memory, the total memory cost would be about 4.3 kilobytes, while the total execution would take about 2150 cycles. Given a fault tolerant time interval of about 10ms, the self-test needs to run more than once every 10ms. Running 2150 instructions once every 10ms means running 215000 instructions per second, consuming 215kHz of CPU performance.

The assumption that all 120 instructions will take an average of 3 repetitions to test might be slightly low. A majority of instructions should only need two repetitions, while some need many more.

The code can be optimised of course. The estimate above assumes that tests are stored back to back in program memory. However, through utilising loops, the memory cost can be reduced at the expense of speed. See the dragon book [9] for a comprehensive overview of optimisation techniques. Further, chaining multiple instructions, using the output of one as input to the next may be used to check the same amount of instructions, using significantly less memory and time. However, one instruction may mask out the error from the previous instruction, further increasing the number of repetitions needed. Alternatively, reusing test patterns for different instructions could save storage and improve speed.

The only practical way to find out how many faults are actually detected, is simulation with fault injection. Achieving 90% fault coverage with software alone might prove to be challenging or impossible. It would definitely be an

iterative process, repeatedly simulating to see how many faults were caught. As such iteration takes about a day to simulate, the idea was abandoned. Academically, this method also is of little interest, as its implementation and success are highly dependent features, specific to the AVR architecture. It is unlikely that this approach would lead to a portable result.

3.1.2 Hardware aided self-testing

Hardware aided self-testing involves introducing hardware structures to achieve faster or better self-test procedures than simple software self-testing. If improvements to coverage are the objective, hardware changes can be used as part of the iterative process discussed in section 3.1.1.

Watchdog timer

Watchdog timers are already present on today's Microchip products. This is a peripheral that needs to be written to from software periodically to verify that the program is not stuck. It throws an error if it is written to outside of specified time windows or if a time window is missed. This ensures that program flow is somewhat correct. If the program counter were stuck, either on a single instruction or in a small loop, the watchdog timer would detect this.

Another benefit is that it can be used to detect systematic software failures which cause the program to enter a deadlock without a hardware fault being the cause.

Safe hardware

To execute any kind of software self-testing safely, we must assume that a basic program with branching and comparisons can run safely. This may not be a valid assumption. This approach focuses on creating a minimal set of safe hardware that is verified through other means, and use it as a base for software self-testing. One approach could be to duplicate any components

needed for program flow, that is, the program counter, the instruction register and the decoding hardware.

More accessible hardware

Another way to improve the results is to make all internal registers readable/writable directly from software. Additionally, more observation and injection points can be introduced specifically to allow for more focused testing. The problem with these injection points is that they may be very intrusive, potentially introducing new systematic hardware failure. Additionally, Microchip and their customers have to protect its intellectual property. Adding too much access could make reverse engineering the software or hardware easier.

3.1.3 Hardware self-testing

The solutions in this section aim to replace parts or all of the software self-test with self-testing routines carried out purely though hardware.

At synthesis, a scan-chain is generated. This allows the use of external testing and debugging. All registers are connected into a long chain leading to input/output pins. This allows a debugger to shift any value into any internal register, and to read any internal register. The approaches presented here utilise these scan-chains to test the hardware using a on-chip controller.

Basic register stuck-at test using scan-chains

One simple model could be to shift a test pattern through all registers in the system, just to observe how it emerges on the other side. For example, this could be one pattern containing only zeros and one containing only ones. If any register along the chain is stuck at either 0 or 1, this will show up in one of the test patterns emerging on the other side.

Figure 3.2 shows a simplified overview of how the scan-chain controller could be implemented. A multiplexer is inserted to allow values to be shifted in a

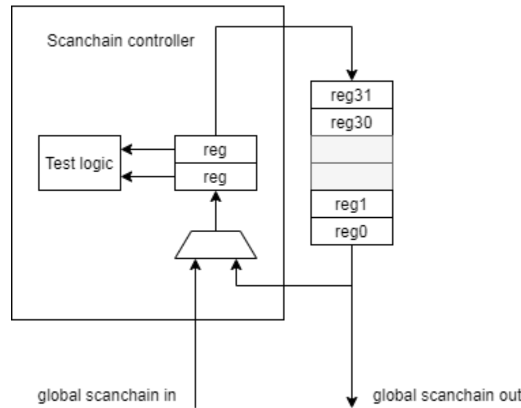

Figure 3.2: An example of a simple self scan-chain controller used for the basic register stuck-at test.

loop including the CPU registers and two test registers, instead of along the global scan-chain. Before a self-test, the test registers are loaded with the test pattern. During the self-test, these patterns are shifted through until they return to their original position. Simple logic can then be used to see if the patterns are unchanged. In addition to what is shown in the figure, some control logic is required: a counter to keep track of how far the pattern has to be shifted for example. This amounts to a relatively small amount of hardware, so it could probably be implemented using the spare logic existent on the chip.

As discussed in the section on software self-tests 3.1.1, a test has to read and write to all registers to verify their functionality. Before this can be done, any data stored there must be pushed to memory. Once the test is over, the data can be loaded back. If a scan-chain was used to verify the register functionality, the register data would not have to be stored in memory, it would simply be shifted around the loop. This allows us to minimise the number of registers used in the software self-test, maybe going as low as two registers used.

Testing the registers using scan-chains removes the need to push and pop 30 of the 32 registers to stack, saving 90-120 clock cycles depending on the AVR model (1 or 2 cycles per push, 2 cycles per pop). The scan itself also takes some time, depending on the length of the scan chain. The downside

is that this almost certainly reduces the quality of the test, compared to reading/writing to the registers through CPU instructions, losing coverage of the circuitry used to select registers for reading and writing. This loss in coverage is likely one we cannot afford if the functional safety of the CPU is based on a software self-test.

Scan-chain self-test

During a factory scan chain test, a pattern is shifted into the chip, the chip is clocked one or more times, and then the data is shifted back out. By checking that the data is as expected, the combinatorial logic between the registers is tested. Running similar scan-chain tests of the CPU using an on-chip scan-chain controller, a software test could be replaced all together.

All scan patterns, as well as correct output patterns, have to be stored somewhere on chip, probably in EEPROM. Factory tests use over 3000 patterns to test for stuck-at-faults alone. The CPU has about 450 bits of register space. Storing 3000 input patterns and 3000 output patterns for 450 bits of registers amounts to about 337 kilobytes of information. This is significantly more than the estimate for the software self-test, but it is guaranteed to detect all stuck-at-faults. It is likely that the number of patterns can be reduced significantly.

In terms of time, the memory interface becomes the major bottleneck. Reading 337 kilobytes of information at two bytes per cycle leads to a total execution time of 168500 cycles, leading to a total loss of performance of 1.685 MHz - quite significant.

There is also some concern about IP protection. Allowing too easy access to the chips internals would make reverse engineering possible.

3.2 Hardware redundancy

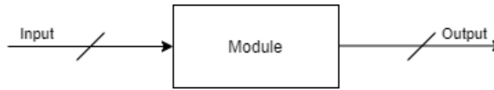

Figure 3.3: A module without any redundancy added to it. This acts as a reference point for any hardware redundancy solutions that are explored.

Hardware redundancy introduces additional hardware, in order to make room for invalid states in the logic table. When a fault occurs, the hardware should fall into a invalid state, which is detected and reported. The main advantage of using redundancy is that it verifies the system's functionality continuously while it is running. This lets it detect transient faults, and allows faults to be detected near instantly. Further, this method does not incur a performance penalty, as the system does not have to pause its normal operation to run a check.

The downside of hardware redundancy is that it can mean a quite significant amount of additional logic, leading to higher power consumption and area. The power consumption scales with the amount of activity in the circuit, meaning that it tends to consume less power than self-tests in low activity applications, and more power in high activity applications.

3.2.1 Full hardware duplication

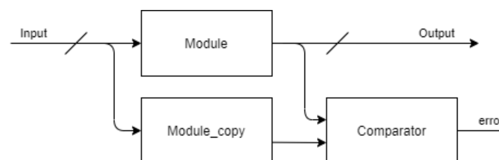

Figure 3.4: A illustration of full hardware redundancy.

The simplest form of hardware redundancy is full duplication of the hardware. Both instances of the hardware get fed the same input data, and so their output is expected to be the same. If the outputs are not equal, one of the two

instances is faulty. A duplicated module, with its outputs being compared is shown in figure 3.4. While being cheap on man hours and simple to verify, it more than doubles the area and power consumption of the logic in question.

The main advantage of full duplication is its versatility. As will become clear in section 3.2.2, fully duplicating the hardware is often the only practical way to introduce hardware redundancy into complex combinatorial blocks. Additionally, the separation between combinatorial blocks and registers is not necessary here, since whole modules can be duplicated with their registers included.

This method is what is referred to as symmetric redundancy in the literature [1]. Symmetric redundancy is considered to be weaker than asymmetric redundancy, where both instances are implemented differently. Asymmetric redundancy reduces the chance for common mode faults, that is a fault that affects both instances in the same way, like a failing supply voltage. It also protects against systematic faults, when one of the instances contains a mistake that is not present in the other. This of course adds considerable development and verification cost.

Alternatively, the redundancy can be made more robust by offsetting the instances from each other in time. Some common mode faults may then affect the two copies at the same time. Because the copies are at different points in the program at this point in time, the resulting malfunctioning behaviour will differ. This difference can be detected. However, this method only improves the detection of non permanent faults, which are not the focus of this thesis.

3.2.2 Checksums

The use of checksums or parity bits is an attempt to achieve redundancy without the cost of full duplication. The most common approach is to add a checksum bit to each signal or memory location. To introduce a checksum bit, we add just one additional bit to a signal. This additional bit is set so that the number of logic ones in the multi bit signal is always odd. If the signal arrives and this is no longer true, one or more bits in the signal must have changed value. If two bits change value, the number of ones is odd

again, meaning that more complex faults may go undetected. This may be improved further by introducing more checksum bits.

Checksum bits are well suited for data transport and storage, but they may be hard to handle in combinatorial logic. Some logic operators are easily extended to include the checksum bit, like XOR, where the checksum bits of the two input signals are simply XOR-ed together to generate the checksum bit for the output signal. In this case, any faults within the XOR logic are also detectable when verifying the output signal checksum. See equation (3.1) through (3.4) for a mathematical proof.

The ˆ operator in the equations (3.1) is the symbol for XOR. If ˆ is between two variables like $F = A\hat{}B$, that means bitwise XOR, i.e. $F_0 = A_0\hat{}B_0$, $F_1 = A_1\hat{}B_1$ etc. If it is in front of a variable like $C_A = \hat{}A$, it means a XOR reduction, $C_A = A_0\hat{}A_1\hat{}A_2$ etc. C_A is the checksum bit for A. F is the variable we wish to evaluate, defined as seen in equation (3.1). C_A is the checksum bit for A. F is the variable we wish to evaluate, defined as seen in equation (3.1)

$$C_A = \hat{}A \qquad C_B = \hat{}B \qquad F = A\hat{}B \qquad C_F = \hat{}F \tag{3.1}$$

We wish to evaluate the bitwise XOR of signal A and B, F. Additionally we wish to evaluate the checksum bit C_f for F, not using F itself. The value for the checksum bit is defined as stated here. However, we wish to evaluate it without using F directly, by instead using the checksum bits of A and B. This way if either C_a or C_b is wrong, C_f will be wrong. If either input signal A or B is wrong, F will be wrong. This means that assuming that a total of one bit is faulty in A and B, this fault can be detected by evaluating the checksum bit for F. Equation (3.2) substitutes A xor B for F.

$$C_F = \hat{}(A\hat{}B) \tag{3.2}$$

Equation (3.3) is a rearrangement of equation (3.2).

$$C_F = (\hat{}A)\hat{}(\hat{}B) \tag{3.3}$$

Equation (3.4) substitutes C_A for A and C_B for B.

$$C_F = C_A\hat{}C_B \tag{3.4}$$

As is shown in equation (3.4), the carry bit for the output can be generated using the carry bit from the input. However, this is only possible because of the nature of the XOR operator allowing the step from equation (3.2) to equation (3.3). For other logic operators, like AND, propagating a checksum is more challenging. This brings us to approaches combining duplication and checksums, which is discussed in chapter 3.2.3.

3.2.3 Combining full duplication and parity bits

As discussed in 3.2.2, propagating checksum bits through a combinatorial module poses a challenge. In fact, the only universal approach that comes to mind is to verify and discard the checksum bit of the input signals, and then generate a new checksum bit for the output. However, if any faults are introduced within the module, after the input signals are verified and before output checksum is generated, this will go undetected. To remedy this, the module can be duplicated, making it a combined approach, as seen in figure 3.5.

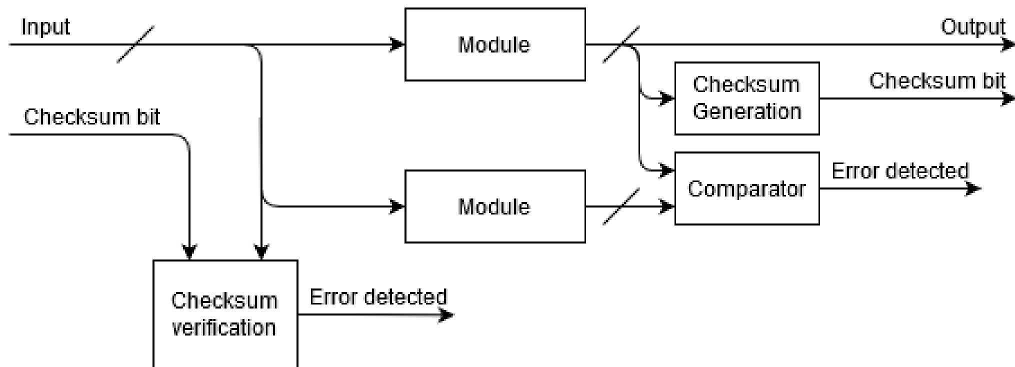

Figure 3.5: The suggested way of interfacing between signals with checksums and duplicated modules.

Why can the circuit in figure 3.5 be considered to be safe even though the safety mechanisms, i.e. the comparator, the checksum generation or the checksum verification, do not contain any redundancy? This is based on the objective that we only have to detect single point faults. Any single point fault in the safety mechanisms can only result in a error detected or have no effect on the system. If it results in a error detected that is valid, as there is indeed a fault within the safety mechanism. If it does not result in a error detected, for example because the "error detected" signal is stuck-at-0, that is harmless as it does not affect the rest of the system. If an actual fault occurs within the area covered by the safety mechanism while it is faulty, that would be the second fault, and the ASIL B standard is quite relaxed when it comes a multi point / latent faults.

One could argue that the comparator block in figure 3.5 is unnecessary, by instead using the output from one module to generate the output signal and the output from the other module to generate the checksum bit, as seen in figure 3.6. The reasoning being that if a fault is introduced into one of the modules, this is likely to result in a invalid combination of checksum bit and output signal, being detected by the next checksum verification block. However, depending on the nature of the module, a single bit error within a module can manifest itself as a multi bit error on its output. As stated in chapter 3.2.2, if there is an even number of bits that are flipped due to the error, a single checksum bit will not detect it. If a single bit fault propagates through a module, it might result in a multi bit fault at the output. In many cases this might be acceptable, as a permanent fault is likely to be detected before the fault-tolerant time interval, given that the inputs keep changing. However, as shown by the example of figure 3.7, a one bit error on the input wires can manifest itself as a two bit error on the outputs regardless of inputs. There are also many modules whose inputs do not typically change very much. Thus, it is safer to follow the structure in figure 3.5.

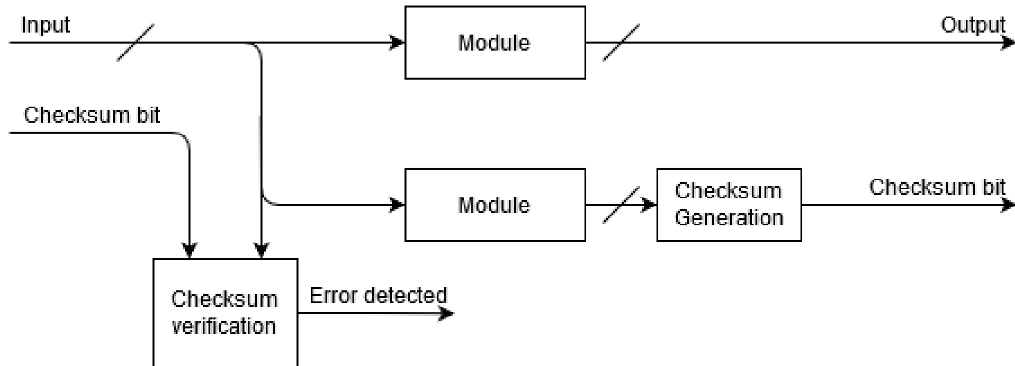

Figure 3.6: A insufficient way of interfacing between duplication and checksums.

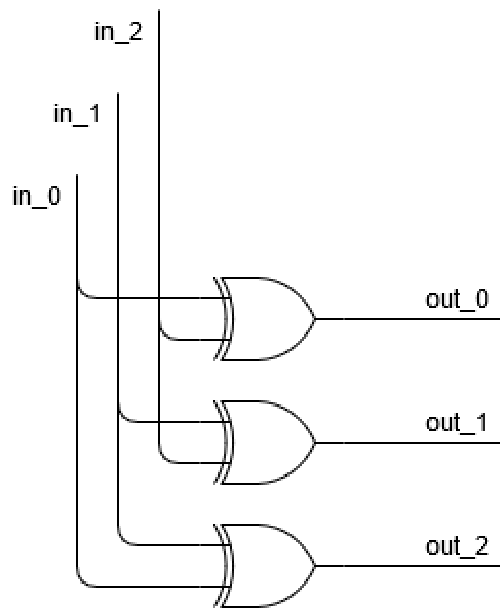

Figure 3.7: An example of a circuit that consistently produces a two bit fault at the output given a one bit fault at the input. Used as an example to explain why the circuit in figure 3.6 is insufficient.

As is evident from figure 3.5, there is a quite significant overhead to transitioning from a checksum signal to a duplicated module, and back. If two copies of figure 3.5 are put in a chain, perhaps containing different modules, the cost of transitioning from duplication to checksum and back likely

outweighs the benefit of only transferring one checksum bit instead of a duplicate signal. At that point it is likely cheaper to just duplicate the data wires between the modules, removing the need for a comparator, checksum generation and verification, as seen in figure 3.8.

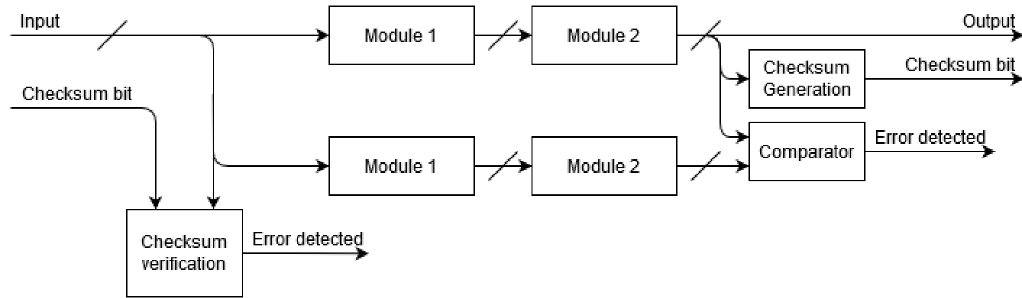

Figure 3.8: An illustration of a set of two modules being duplicated together. This can often be cheaper than connecting the modules using checksum verified signals.

On the other hand, there are situations where the transition is clearly worth it. If a large hardware block can operate on checksums, and has narrow input and output signals compared to its size, it becomes worth it. Addressable data storage is a prime example. Using checksums here allows us to only add one bit per storage location, istead of duplicating the storage. This includes memory, but also the register file within the CPU. Significant savings over a full duplication can likely be made by using checksums in the register file.

This comes down to a partitioning problem. Each part of the system must contain redundancy from either checksums or full duplication, each part of the circuit is more suited for one or the other and there are costs associated with transitioning between the two approaches. Partitioning problems like these are known to be NP hard, and are usually approximated. This is outside the scope of this project.

In addition to optimising the amount of hardware required, the amount of work is also a concern. Implementing something like this manually could be feasible if very large regions are chosen. There is also a risk for introducing new bugs when old modules well tested in the field apart like this.

In the case of this project, implementing the register file using checksums,

while the rest of the CPU is duplicated shows the most promise.

Duplicating the CPU in a system that is largely verified through checksums could also be considered as a combined approach.

Chapter 4

Test and Results

4.1 Selecting approaches for implementation

It was decided that the full hardware duplication as described in section 3.2.1 is the most promising approach for the CPU,

The full duplication solution is likely the best in terms of the number of faults covered. Any faults that propagate out of the CPU instances and into the comparator would be detected. It is the most versatile, being usable on any module. It also requires little implementation and verification work.

A combined approach, like the one described in chapter 3.2.3 also shows great promise. It could potentially yield the same single point fault detection rate as a full duplication, at lower cost.

The self scan chaining approach also shows great promise in terms of coverage, but based on the estimates made in that chapter, the storage cost and execution time would be far greater than what is acceptable. This section overestimates the amount of storage and execution time greatly, but even after adjusting the estimates this would be the slowest self-test. There is also some concern about giving customers too great access to the hardware, allowing for reverse engineering.

The software self-tests show little promise. Achieving the necessary coverage

to reach the certification seems challenging. While testing the hardware duplication implementation, a software self-test will be tested as a by product.

4.2 Test method

4.2.1 Building the hardware duplication

As described in chapter 3.2.1, a second instance of the CPU module was created. The original CPU will be referred to as the primary CPU, while the second instance will be referred to as the secondary CPU. The secondary CPU receives the same input signals as the primary CPU, but drives a newly instantiated set of output signals.

The signals are then compared by the Verilog code shown below. Note output signals are not explicitly listed in the in this document, instead output_signals and output_signals_copy hold their place.

```
always_ff @(posedge clk)
    if ( reset )
        error_detected <= 1'b0;
    else if(output_signals == output_signals_copy)
        error_detected <= 1'b0;
    else
        error_detected <= 1'b1;
```

The circuit is then simulated without fault injection, using Microchip's test library for the CPU. If the functionality of the system changes due to the changes introduced as part of this thesis, this would be caught here.

4.2.2 Fault injection simulation

Next, the circuit is simulated using Z01X, as described in chapter 2.5. As mentioned there, the following questions need to be addressed when setting up the fault injection simulation:

1. What test should be used as a base for the simulations?
2. Which faults should be injected where?
3. How should the results be classified?

Addressing question 1, a pre-existing test used internally for verification is selected. It is quite similar to the self-test program described in section 3.1.1. This test tries all CPU instructions at least once, giving decent test coverage of the CPU in a relatively short time. It was not designed to detect all stuck-at-faults, but rather to find implementation errors. Still, using this test lets us draw some conclusions about the potential of a software self-test.

For question 2, stuck-at-0 and stuck-at-1 faults are injected at all signals and registers in the primary CPU, apart from sections not used by the customer. This results in a total of 7781 fault injections, resulting in 7123 separate simulations. The difference in the number of faults injected and the amount of simulations is due to some faults being equivalent to each other. The resulting simulation time is about 18 hours. No more complex fault models are simulated for this proof of concept.

In a separate set of simulations, a few faults were injected into the secondary CPU. These faults are not a part of the statistics presented later, and were only simulated to verify that the faults injected into the secondary CPU have equivalent results to the faults injected in the primary CPU.

Regarding question 3, the fault classification is based on the automatic classification by Z10X. Unwanted classifications are disabled, like hyperactive faults or the one generated when accessing a file, which is illegal during Z10X simulations. Faults which set off the *error_detected* signal in the Verilog block in section 4.2.1 are classified as detected signals. Signals which propagate out of the CPU and do not set off the *error_detected* signal are classified as dangerous faults. This last category is expected to be empty while only injecting single point faults in one CPU. However, if the simulations are expanded to include multi point faults or alternative solutions are tested, some faults may fall in this category.

Faults not classified by the conditions outlined above are classified automatically by the tool. This includes the faults that are not propagated out of

the CPU. These faults are classified as "not controlled" when the simulation never attempts to change the signal away from its stuck-at value. Alternatively they are classified as "not detected" or "not observed" if the fault does have an effect, but does not propagate out of the CPU.

Note that the "detected error" fault classification was not intended. This will be mentioned in section 5.2.

4.2.3 Synthesis and cost estimate

The system is then synthesised as outlined in section 2.4.4. Additionally, a reference system without the duplicated CPU is synthesised using the exact same method. The synthesis tool is then used to generate both estimates of logic area and power consumption. Both estimates could be more accurate by placing and routing the design, but should be good enough to compare the two systems.

4.3 Results

A part of the fault injection simulation log is shown in appendix C, and summarised in table 4.1. The fault injection simulation was carried out as described in section 4.2.2. The results show that 47.29% of all possible stuck-at-faults were detected by the fault detection mechanism during the general CPU test.

Table 4.1: An overview of the fault injection simulation result. These results are based on the report in appendix C.

Fault classification	Percentage	Explanation
Detected	47.29%	Faults that were detected
Detected error	0.06%	Faults that lead to a $error or $fatal
Not Detected	40.83%	Faults that had no effect outside the cpu
Not Observed	0.13%	Faults that had no effect outside the cpu
Not Controlled	11.68%	Faults that had no effect
Dangerous	0.0%	Faults that left the CPU undetected

To estimate the cost of the full duplication, the system was synthesised to get a rough estimate of the area and power consumption. The system was also synthesised without the duplicated CPU as a baseline. The area and power reports from both syntheses are shown in appendix A and B, and are summarised in table 4.2. The table shows that the power consumption increased by 13.595% and the area increased by 8.973% due to the introduction of a second CPU.

Table 4.2: The approximate increase in power consumption and logic area resulting from the CPU duplication. The cell area is given in a arbitrary area unit, but is sufficient for comparison. The values in the table are based on appendix A and B.

	Result	Baseline	% change
Total cell area	1061365	973972	8.973%
Estimated power consumption	13570 μW	11946 μW	13.595%

These synthesis reports only contain the digital logic, meaning that flash, ram, I/O pin drivers, analog components etc. are missing. These components represent about half of the chip's total area and a significant amount of the chip's total power consumption. This means that the relative increase in power consumption and area in table 4.2 are significantly higher than when considering the system as a whole.

Chapter 5

Discussion

Five objectives were defined in section 1.2. In the following, the results corresponding to each of these objectives will be discussed. The extent to which each objective has been achieved will be assessed and possible explanations for unexpected results will be given. In the case of any shortcomings, ways to improve the results in that regard will be presented.

5.1 Objectives

Objective 1: Detect at least 90% of single point faults

No faults propagated out of the CPU without being detected. This means that seen from the outside, the processor never acted abnormally without a error being reported. One could argue that this should be sufficient. However, only 47% of all faults injected in the CPU were detected, meaning that the 90% goal was missed by a wide margin. However, this metric is not really appropriate for evaluating the quality of the solution, as it only describes how many percent of the faults propagated out of the CPU - everything that did propagate out was detected.

The 47% statistic is really a function of the test program used and not the potential of the hardware duplication. As the test program used was very

similar to the software self-test proposed in section 3.1.3, this figure could instead be used as a optimistic figure of how many faults would have been caught by the software self-test. Optimistic because the self-test jumps to a specific location in the program when it detects faulty behaviour, causing the program counter to act differently than in a fault-free simulation. As the program counter is visible in the outputs of the CPU, this is one of the causes which could set off the "error detected" signal in the comparator.

The 90% of single point faults detected objective was really defined with respect to self-testing routines, the solution proposed by the customer. However, catching 100% of faults that leave the CPU is the real metric that we should be looking at, and from this perspective the results were excellent. The ISO 26262 documentation agrees with this conclusion, stating that full duplication of hardware typically leads to the highest level of diagnostic coverage, considerably better than what is required for ASIL B.

Are the faults that go undetected dangerous? Leaving undetected single point faults in the system could lead to more complex multi point faults, when multiple single point faults accumulate. However, it is hard to imagine how this could happen in practice - both CPUs randomly failing the same way.

If improving the 47% figure is deemed necessary, more probing points can be added inside the CPU. By not just comparing its outputs but also some points within the CPUs, like the output of the ALU, more faults could be caught and faults could be caught earlier.

Alternatively this figure can be combining the method with self-testing routines. The presence of two copies of the CPU removes the need to store the expected output patterns, as the other copy could be used for comparison. This allows the design of extremely compact tests where the tests are generated through a pseudo random process, and the correctness is verified by comparing the two CPUs. The comparisons would also be faster, using the comparator that was implemented as part of this thesis, instantly setting off the hardware signal that indicates that a fault was detected.

In conclusion, while the 47% result seems to miss the 90% objective, this result does not really have any significance. The safety provided by duplicated hardware is still greater than that provided by a self-test with 90% coverage.

Objective 2: Achieve a fault tolerant time interval of at most 10 ms

One of the greatest strengths of the hardware duplication approach is that it detects faults as soon as they manifest themselves. This leaves the maximum possible time tolerance for the transition to a safe state. Thus the second objective is achieved.

The CPU also is not the only component that needs to be tested, and the selected implementation leaves a large amount of time for potential testing routines for other components.

Objective 3: Minimise the chance of introducing new systematic hardware failures

The simplicity of the hardware duplication approach minimises the chance of introducing additional bugs. All the other hardware based approaches involved larger changes to the hardware, making room for more potential faults. The hardware was also verified to be equivalent to the starting point through the use of Microchip's library of tests.

For true minimisation, an equivalence checker which mathematically proves that the solution is logically equivalent to the initial design could be used. This may be relevant for more complex solutions which take apart the CPU module, but not in the case of simple duplication.

Objective 4: Minimise the change of introducing new systematic failures in software, by maintaining the system's ease of use

This objective is solved optimally through the use of any of the hardware duplication methods. As this system is functionally equivalent to the AVR cores that are on the market, the chip is backwards compatible, acting exactly like the customer expects. The self-testing routines had the potential to

introduce some issues in regards to this objective. If the customer does not take the self-test into account when writing their application, new bugs may appear.

Objective 5: Minimise both development and production cost of the system

Development costs were definitely minimised by picking hardware duplication. The implementation and verification of duplicate hardware was done in a day, while the other solutions would have taken significantly longer.

With regards to production costs however, the solution of duplicating the hardware might have been the most expensive. While the increases in logic area in table 4.2 do not seem very significant, we must keep in mind that only the CPU has been addressed. Duplicating the CPU naturally takes about twice the area of a single CPU.

It is likely that much of the remaining chip can be tested through cheaper means than the CPU. The combinatorial logic outside the CPU is a lot simpler than within the CPU. This means that testing routines and checksum based approaches are much more promising here. The CPU also spends no time in self-testing routines, leaving more time to test other components.

5.2 Limitations

As noted earlier, table 4.1 includes 0.06% of faults which were classified as "detected error". This was due to a mistake in the configuration of the tool. Faults are classified as "detected error" if a $error or $fatal is encountered in the Verilog code. As $error and $fatal are ignored when synthesising the final design, they do not impact the functionality of the physical chip. A better representation of how the physical chip's behaviour is to simply jump over any $error and $fatal calls, and continuing the simulation until it is otherwise terminated. This can be achieved by disabling this error classification, similar to what was done with illegal file access errors in section 4.2.2. However, this

misconfiguration was not spotted early, and rerunning an 18 hour simulation to address an insignificant number of faults was deemed unnecessary.

In retrospect, there is a possibility that some simulations ended prematurely by encountering a $finish. This would cause the fault to be misclassified as "not detected", "not observed" or "not controlled". However, there is only one $finish in the simulation, and it in the test bench used for correct termination of the simulation. It is triggered through a software to test bench interface commonly used by Microchip, which is highly unlikely to trigger unintentionally without unusual behaviour in the program counter. While this may affect some of the simulations, it should have no significant impact on the final statistics.

The major weakness of duplicated hardware is common mode faults. These are faults where one root cause affects both copies of the CPU in the same way. Many techniques can be deployed to avoid this, by extending the duplication to as many components that could be such a root cause as possible. Sharing clock gates would be a prime example.

Chapter 6

Conclusion

6.1 Main findings

This thesis aimed to select cost effective modifications for the AVR processor to make it comply with the functional safety requirements set by the automobile industry. It was determined that the best approach would be to duplicate the CPU and compare the outputs of the two CPU instances. The presence of any differences between their outputs indicates that one has failed.

The major advantage of using a simple hardware duplication approach is that it does not change the behaviour of the system. It is also easy to implement and arguably provides enough fault detection to fulfil the functional safety requirements.

The fault injection simulations did not yield a high enough fault detection rate to fulfil the objectives set out in the introduction. I would argue that objective regarding fault detection rates was misdefined, as it was written with a self-testing routine in mind, as opposed to hardware duplication. The solution that was implemented provides much better functional safety than a self-test ever could. It can detect all single point faults, permanent or non-permanent, before they can ever influence the outside behaviour of the CPU, and by extension the system as a whole.

6.2 Future work

6.2.1 Expanding to a system level solution

This thesis focused mainly on the AVR CPU, which is only a small part of the Microchip System-on-Chip products. For the product to comply with functional safety standards, all critical parts of the system must meet functional safety requirements.

Ideally, all memory locations and buses on the chip should be using checksum bits to verify their functionality. The interfaces to modules containing combinatorial logic, like the CPU, should follow the practices outlined in section 3.2.3.

Some peripheral modules may be best tested through a testing routine. When the CPU is duplicated and known to be working as expected, constructing software tests becomes less challenging. Typically the user has a lot more control over peripherals than the CPU itself, allowing for more targeted testing.

Many peripheral modules contain analog components, and this thesis has mostly focused on digital logic. Some analog components can be used to check each other, like an analog-to-digital converter verifying the output of a digital-to-analog component. However, for safety-critical components, duplication might be the safest option.

6.2.2 Functional safety concerns when placing, routing and generating a clock tree

A careful placement, routing and clock tree synthesis process is key when it comes to avoiding common mode faults. Due to their shared connections, copies of a module are often placed on the same area of the chip. The similarity in operation also leads to them being on the same branch of the clock tree. Constraints should be set to counteract this, as this causes common mode faults.

6.2.3 Reducing the cost by defining a set of safe instructions

The cost of all potential fault detection mechanisms can be brought down by defining a subset of functionality which is suitable for functional safety. Picking some registers and instructions, and duplicating the hardware needed to make these safe could save a significant amount of resources. Unsafe, more complex instructions can often be emulated by using a series of safe instructions, while many programs only have a few values which really need to be safe.

When picking a software based solution, it is also possible to tailor the test to fit the customer's application. By only verifying the instructions and registers the customer actually wants to use, the same level of safety and customer freedom can be achieved at a much lower self-testing time.

6.3 Closing words

This thesis was a important first step towards making AVR based system-on-chip products comply with the strict standards set by the automobile industry in recent years. With increasing amounts of automation in modern vehicles, using solid and fault resistant logic to control units has the potential to save lives.

When a mistake in the design of your product could lead to loss of life, choosing simple and robust architecture for your design yields the safest results.

Bibliography

[1] "Road vehicles - functional safety," standard, International Organization for Standardization, Geneva, CH, 2018.

[2] J. Gracia, D. Gil, L. Lemus, and P. Gil, "Studying hardware fault representativeness with vhdl models," in *Proc. of the XVII International Conference on Design of Circuits and Integrated Systems (DCIS), Santander (Spain)*, pp. 33–39, 2002.

[3] "AVR instruction set manual." https://onlinedocs.microchip.com/pr/GUID-0B644D8F-67E7-49E6-82C9-1B2B9ABE6A0D-en-US-1/index.html. Accessed: june 2020.

[4] S. N. Muhammad Ali Mazidi and S. Naimi, *The AVR microcontroller and embedded systems, using assembly and c*. Pearson, 2011.

[5] J. Zimmermann, "Power optimal synthesis with dynamic analysis of gated clocks," 2019. Project at NTNU in cooperation with Microchip.

[6] Synopsis Silicon Design & Verification, *PrimeTime Datasheet - Golden Timing Signoff Solution and Environment.*

[7] "Z01X simulator safety verification user guide." Version Q-2020.03, March 2020.

[8] "TestMAX FuSa fast static analysis to improve functional safety datasheet." https://www.synopsys.com/content/dam/synopsys/implementationsignoff/datasheets/testmax-fusa-ds.pdf. Accessed: june 2020.

[9] A. L. S. Ullman, *Compliers: Principles, techniques, and tools.* Pearson, 2nd ed., 2014.

Appendix A

Synthesis area report

Dual CPU synthesis area report:

```
Number of ports:                          82859
Number of nets:                          156965
Number of cells:                          85329
Number of combinational cells:            62407
Number of sequential cells:                9776
Number of macros/black boxes:               554
Number of buf/inv:                        15008
Number of references:                        11

Combinational area:              548328.048517
Buf/Inv area:                     75932.170971
Noncombinational area:           507477.750080
Macro/Black Box area:              5560.108371
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                1061365.906968
```

Reference design synthesis area report:

```
Number of ports:                          81384
Number of nets:                          146502
Number of cells:                          76332
Number of combinational cells:            54009
Number of sequential cells:                9258
Number of macros/black boxes:               547
Number of buf/inv:                        12359
Number of references:                        12

Combinational area:              487802.358095
Buf/Inv area:                     61406.011292
Noncombinational area:           480859.646900
Macro/Black Box area:              5310.276395
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                 973972.281391
```

Appendix B

Synthesis power consumption report

Dual CPU power estimate

Power Group	Internal Power	Switching Power	Leakage Power	Total Power	(%)	Attrs
io_pad	0.0000	0.0000	0.0000	0.0000	(0.00%)	
memory	0.0000	0.0000	0.0000	0.0000	(0.00%)	
black_box	0.0000	651.3143	0.0000	651.3143	(4.80%)	
clock_network	2.3863e+03	1.2975e+03	1.6891e+05	3.6839e+03	(27.15%)	
register	9.0960e+03	1.3137	9.0701e+05	9.0982e+03	(67.05%)	
sequential	1.5329	2.8667e-02	1.8375e+03	1.5634	(0.01%)	
combinational	74.2849	60.0684	8.1341e+05	135.1666	(1.00%)	
Total	1.1558e+04 uW	2.0102e+03 uW	1.8912e+06 pW	1.3570e+04 uW		

Reference design power estimate

Power Group	Internal Power	Switching Power	Leakage Power	Total Power	(%)	Attrs
io_pad	0.0000	0.0000	0.0000	0.0000	(0.00%)	
memory	0.0000	0.0000	0.0000	0.0000	(0.00%)	
black_box	31.9275	651.9062	6.1799e+03	683.8412	(5.72%)	
clock_network	2.0010e+03	1.2331e+03	2.5995e+05	3.2344e+03	(27.08%)	
register	7.9106e+03	1.2822	1.4339e+06	7.9133e+03	(66.24%)	
sequential	1.3000	2.9072e-02	3.0962e+03	1.3322	(0.01%)	
combinational	56.3653	55.5758	1.0700e+06	113.0096	(0.95%)	
Total	1.0001e+04 uW	1.9419e+03 uW	2.7731e+06 pW	1.1946e+04 uW		

Appendix C

Fault injection log

```
# Fault Coverage Summary
#
#                                        Prime              Total
#-------------------------------------------------------------------------------
# Total Faults:                          7123               7781
#
# Dropped Detected                 DD    3217  45.16%       3680  47.29%
# Detected Error                   DE       4   0.06%          5   0.06%
# Dropped Potential                PD       0   0.00%          0   0.00%
# Not Detected                     ND    3084  43.30%       3177  40.83%
# Not Observed                     NO      10   0.14%         10   0.13%
# Not Controlled                   NC     808  11.34%        909  11.68%
#
# Untestable Tied                  UT       6                  7
```