Institute of Computer Architecture and Computer Engineering

University of Stuttgart
Pfaffenwaldring 47
D–70569 Stuttgart

Masterarbeit

Development of an Infrastructure for Creating a Behavioral Model of Hardware of Measurable Parameters in Dependency of Executed Software

Denis Schwachhofer

Course of Study: M. Sc. Informatik

Examiner: Prof. Dr. rer. nat. habil. Ilia Polian

Supervisor: Prof. Dr.-Ing. Steffen Becker

Commenced: 11 June 2020

Completed: 08 January 2021

Abstract

System-Level Test (SLT) gains traction not only in the industry but as of recently also in academia. It is used to detect manufacturing defects not caught by previous test steps. The idea behind SLT is to embed the Design Under Test (DUT) in an environment and running software on it that corresponds to its end-user application. But even though it is increasingly used in manufacturing since a decade there are still many open challenges to solve. For example, there is no coverage metric for SLT. Also, tests are not automatically generated but manually composed using existing operating systems and programs. This master thesis introduces the foundation for the AutoGen project, that will tackle the aforementioned challenges in the future. This foundation contains a platform for experiments and a workflow to generate Systems-on-Chip (SoCs). A case study is conducted to show an example on how on-chip sensors can be used in SLT applications to replace missing detailed technology-information. For the case study a "power devil" application has been developed that aims to keep the temperature of the Field Programmable Gate Array (FPGA) it runs on in a target range. The study shows an example on how software and parameters influence the extra-functional behavior of hardware.

Kurzfassung

System-Level Test (SLT) nimmt nicht nur in der Industrie Fahrt auf sondern seit kurzem auch in der akademischen Welt. Es wird verwendet als Ergänzung zu existierenden Testschritten, um weitere Defekte zu erkennen, die bisher nicht gefunden worden sind. Die Idee bei SLT ist, dass der zu testende Chip in einer Umgebung eingesetzt und Software ausgeführt wird, die einer möglichen Endanwendung entspricht. Auch wenn es eine immer größere Bedeutung erlangt, existieren noch Herausforderungen, die gelöst werden müssen. Zum Beispiel hat SLT keine Metrik, um die Fehlerabdeckung zu bestimmen. Weiterhin werden Tests von Hand zusammen gestellt aus existierenden Betriebssystemen und Programmen und nicht automatisiert erzeugt. Diese Masterarbeit stellt die Grundlagen für das AutoGen Projekt vor. In diesen Grundlagen enthalten sind eine Plattform für weitere Experimente und ein Arbeitsfluss, um Systems-on-Chip (SoCs) zu erzeugen. Eine Fallstudie wird durchgeführt, um ein Beispiel vorzustellen, wie Sensoren, die sich im Chip befinden genutzt werden können, fehlende detaillierte Technologieinformationen zu ersetzen. Für diese Fallstudie wurde eine “power devil” Applikation entwickelt, die versucht die Temperatur des Field Programmable Gate Arrays (FPGAs) in einem Zielbereich zu halten. Die Studie zeigt ein Beispiel dafür, wie Software und Parameter das nicht-funktionale Verhalten von Hardware beeinflussen kann.

Contents

1 Introduction **13**

2 Basics of Hardware Testing **15**
2.1 Testing in General 15
2.2 Functional Testing 16
2.3 Structural Testing and Design for Test 17
2.4 Design-for-Test 17

3 System-Level Test: What is it and why do we need it? **19**
3.1 Definition and Position of System-Level Test 19
3.2 Advantages and Challenges of System-Level Test 20
3.3 Structural Test vs Functional Test vs SLT 22

4 Related Work **25**

5 Case Study: Self-aware SLT using software-based adaptive Temperature Control **27**
5.1 Requirements on Platform and Workflow 27
5.2 Components of the Platform 28
5.3 Building the Platform and Workflow 33

6 Results: Self-aware SLT using software-based adaptive Temperature Control **39**

7 Conclusion **45**

Bibliography **47**

A Listings **51**

List of Figures

3.1 Testing flow for a generic chip (from [PAB+20]) 20

5.1 Established flow 35
5.2 Block diagram of hardware design 36
5.3 Flow chart of control task 37

6.1 Temperature curve for Mandelbrot with floating point arithmetic 40
6.2 Temperature curve for Mandelbrot with fixed point arithmetic 40
6.3 Temperature curve for SHA256 41
6.4 Temperature curve for floating point Mandelbrot over longer period of time . . . 42

List of Tables

5.1 Comparison of available RISC-V cores 32

6.1 Details about LargeBoom 39

Listings

5.1 Example Core 30

A.1 LargeBoom 51

Acronyms

ALU Arithmetic Logic Unit. 33

AMT Asynchronous Multi-Threading. 33

ASIC Application Specific Integrated Circuit. 15

ATE Automatic Test Equipment. 22

ATPG Automatic Test Pattern Generation. 16

AXI Advanced eXtensible Interface. 28

BIST Built-In Self Test. 15

BOOM Berkeley-Out-Of-Order. 29

CI/CD Continuous Integration/Continuous Deployment. 27

CPU Central Processing Unit. 21

DDR RAM Double Data Rate Random Access Memory. 19

DfT Design for Test. 15

DIMM Dual Inline Memory Module. 19

DMA Direct Memory Access. 30

DPPM Defective Parts Per Million. 15

DUT Design Under Test. 3, 13

DVFS Dynamic Voltage and Frequency Scaling. 20

FPGA Field Programmable Gate Array. 3, 5, 13

FPU Floating Point Unit. 29

FT Final Test. 19

GPU Graphical Processing Unit. 19

Hart Hardware Thread. 29

HDL Hardware Description Language. 25

IC Integrated Circuit. 13

IP Intellectual Property. 26

ISA Instruction Set Architecture. 16

MMIO Memory-Mapped I/O. 30

MMU Memory Management Unit. 31

NIST National Institute of Standards and Technology. 31

NSA National Security Agency. 31

PCI Peripheral Component Interconnect. 19

RISC Reduced Instruction Set Computer. 29

RSN Reconfigurable Scan Network. 22

SBST Software-Based Self Test. 16

SLT System-Level Test. 3, 5, 13

SMT Simultaneous Multi-Threading. 31

SoC System-on-Chip. 3, 5, 13

TLB Translation Lookaside Buffer. 30

UART Universal Asynchronous Receiver Transmitter. 30

UCB-BAR University of California - Berkeley Architecture Research. 28

WS Wafer Sort. 19

XADC Xilinx® Analog-to-Digital Converter. 33

1 Introduction

System-Level Test (SLT) is gaining relevance in the industry over the last decade [BRR+20]. It is used by manufacturers to detect defective Integrated Circuits (ICs) which were not caught by previous test steps. It also finds increasing use in post-silicon validation and characterization. The idea behind SLT is to run functional tests in an environment which both approximate the end-user application of the IC. It is able to detect faults which cannot be detected by conventional testing.

But SLT also has its challenges. One of these is that there is no coverage metric that can be used. Such a coverage metric is necessary to quantify and improve test quality. SLT is unable to use metrics from conventional testing, as will be explained later in this thesis. And there is no sufficient replacement yet.

SLT is currently performed using existing operating systems and software. Chen [Che18] mentions that SLT pattern development is an interesting topic to examine further. It seems that there is currently no method available that is able to generate shorter, more targeted SLT programs. And a long run-time is another challenge.

The goal of this thesis is to demonstrate some aspects of a “greybox” approach for SLT. Currently, SLT uses a black box model of the Design Under Test (DUT), because either no details are provided for it or its complexity is too high and as such it is decided to ignore them. As such, on-board sensors, that nowadays come on many Systems-on-Chip (SoCs), can be used to provide some information to an SLT test program. This approach will be expanded upon in the PhD project “P5: Automated Generation of System-Level Test Programs for Characterization of Parametric Device Properties” (short: “AutoGen”). AutoGen intends to provide a method to generate SLT test programs, that also consider extra-functional properties such as temperature or power, and to find a coverage metric.

To demonstrate these aspects a case study is conducted, where the temperature sensor of a Field Programmable Gate Array (FPGA) is used to control the temperature while a power devil program runs on a SoC that is loaded onto it. Along developing the platform for the case study a workflow for generating SoCs and running programs on these will be established as well. The choices for the components and composition of these are all made in relation to AutoGen. As such, the results of this thesis are used as basic work for AutoGen.

We will show in this thesis, that we are able to use the temperature sensor to control the temperature. We will also answer the question of how short the measurement period, i. e., the time between measurements, can be without loosing control. The effects of algorithm choice, the hardware itself and the length of the measurement period, as a tunable parameter, on temperature are shown and discussed. Additionally, an effect, called “software-based thermal runaway” in this thesis, is discussed and possible causes are elaborated.

The rest of this thesis is structured as follows: Chapter 2 introduces the reader to the necessary basics of hardware testing. It discusses testing in general and presents functional and structural test. Chapter 3 introduces the reader to the manufacturing test flow of an SoC, how SLT fits into this

and why it is necessary. It lists the advantages and challenges of SLT and current solutions for the latter. It also compares SLT to structural and functional testing. Chapter 4 presents some related work on SLT and on power devil programs. Chapter 5 lists the requirements on the workflow and the platform and shows how these were fulfilled. It lists the components and elaborates the reason for their usage in this thesis. It also introduces the structure of the case study. Chapter 6 shows the results from the case study and briefly discusses them. Finally Chapter 7 concludes this thesis, discusses the effects of "software-based thermal runaway" on SLT and test program generation and gives an outlook on AutoGen.

2 Basics of Hardware Testing

This chapter introduces the reader to the necessary basics of hardware testing. It assumes that the reader has basic knowledge about digital components but missing knowledge can for example be looked up in [Mic03]. Otherwise, this chapter is mainly based on the books by Bushnell and Agrawal [BA04] and Wang et al. [WWW06] and as such these can be consulted for more in-depth information about the presented topics.

This chapter starts with presenting the reason for testing and important terminology used throughout this thesis in Section 2.1. It introduces the concept of fault models and coverage metrics. Section 2.2 gives an short overview of functional testing. Section 2.3 introduces structural testing and discusses its advantages and disadvantages compared to functional testing. Finally, the concept of Design for Test (DfT) is explained in Section 2.4 and two popular instances of it, scan and Built-In Self Test (BIST), are introduced.

2.1 Testing in General

Due to Moore's Law [Moo06], which to this day carries an important role in the Application Specific Integrated Circuit (ASIC) industry, current day ICs grew in complexity and shrank in feature size. Because of the small feature size IC become more susceptible to process variations in the manufacturing process. This makes testing an integral part of the manufacturing process to catch defects caused by aforementioned variations as early as possible. Wang et al. [WWW06] mention the "rule of ten", which states that the cost of detecting a faulty IC increases tenfold for each stage in the manufacturing process.

To measure the quality of the testing process a metric called Defective Parts Per Million (DPPM), also known as reject rate, is used. DPPM indicates the defective parts that have been sent to customers and have been returned. The goal of testing is to reduce DPPM as much as possible as this in turn reduces the cost for the manufacturer. Returned defective ICs are analyzed to find the cause and the manufacturing process is improved according to the findings.

Possible defects include broken interconnects, oxide breakdowns or shorts. These defects cause a mismatch in the output of an IC compared to what is expected and this faulty output is called error. But defects cannot be modeled directly. As such, fault models are introduced as an abstraction. An instance of a fault model is called a fault. Several fault models exist, e. g., stuck-at, bridging faults, crosstalk, flip-flop transparency faults and many more. The stuck-at-0/1 fault models state that a wire in the IC is permanently driven by a constant logic 0 respectively a constant logic 1. Bridging faults represent undesired connections between two wires. There exist several types, e. g., 4-way, AND-dominated, and more. Crosstalk faults occur when the distance between two wires is small enough such that activity on one wire can induce current on the other. Flip-flop transparency

faults cause affected flip-flops to degrade to combinatorial buffers thus loosing the ability to act as a memory [UKW17]. Bushnell and Agrawal [BA04] provide a more exhaustive list of fault models with more details.

Test patterns are used to test ICs. They are applied as inputs and the outputs of the DUT, i. e., the tested IC, are observed to look for differences compared to the expected output. If there are any, the DUT is classified as faulty otherwise as passing. Test patterns are nowadays generated with the help of Automatic Test Pattern Generation (ATPG). ATPG takes the DUT as a gate-level netlist and a list of all possible faults of a certain fault model that can occur in the DUT. It then generates test patterns using algorithms, such as for example boolean satisfiability (SAT), with the goal to have as many faults as possible be detected by those.

To measure the quality of the test patterns fault coverage is used. Fault coverage fc is defined as follows:

$$fc = \frac{\text{\#detected faults}}{\text{\#total faults}}$$

The higher the fault coverage the better the test patterns. But a fault coverage of 100% cannot always be achieved, for example due to redundancies in the DUT or because there is no sensitizable path, i. e., the effect of the fault cannot be propagated to an output. Also, sequential circuits are harder to test because gates and wires deep within the design become increasingly harder to control and observe.

Fault coverage is determined by executing fault simulation or fault emulation. Fault simulation is a logic simulation of the DUT where faults are injected into it. Then all test patterns are applied and the outputs are observed to see, if it differs from the fault-free circuit. If that is the case, the fault counts as detected. In the simplest implementation, all faults are sequentially simulated with all test patterns. But, there exist many algorithms and techniques to increase the speed of fault simulation. Still, with increasing size of the DUT fault simulation takes up a lot of time.

2.2 Functional Testing

Functional testing looks at the function of the DUT and checks for a set of functional inputs if the corresponding output is correct, e. g., it checks if an adder returns 6 when applying 2 and 4 as inputs. It targets faults that are not caught by structural testing, e. g., defects that depend on certain orders of instructions. The quality of the functional test patterns, or test programs, can be measured using fault coverage. A representative of functional testing is Software-Based Self Test (SBST). In SBST, test programs are generated on the assembly level to target structural faults using the microprocessor's Instruction Set Architecture (ISA). They are embedded into the DUT. The test program and the test patterns or test pattern seeds are loaded into the cache of the microprocessor and then the test program is executed. Generally, functional tests are kept as short as possible to reduce testing and fault simulation time. Functional tests are often generated by modeling the functionality of a microprocessor as a graph but for SBST an ATPG tool is available [RCS+16].

The reason that IC testing shifted away from functional testing is that it requires a high number of test patterns where generally many of them do not contribute much to the total fault coverage. Also, these patterns are difficult to optimize.

2.3 Structural Testing and Design for Test

In structural testing the DUT is abstracted as a gate-level netlist. Test patterns are generated by “looking” at the netlist and finding input patterns to trigger the fault and propagate its effect to an output. Structural testing is the most used kind of IC testing nowadays. Many ATPG tools were developed for structural testing, from the D-algorithm [Rot66] to PODEM [GR81] to SAT-based algorithms [Lar92], which are still further developed today with one example being PHAETON [SBP16].

Structural testing is able to deploy test compaction strategies. For example, test patterns that contain dont-cares, i. e., signals whose value can be freely chosen, can be combined to a single pattern as long as there are no conflicting assignments. Also, generally test patterns can be applied in any order but there are fault models where the order is important, such as for example for delay faults such as slow-to-rise or slow-to-fall where a rising respectively falling transition has to be triggered to detect the such faults.

Since structural testing does not consider the functionality of the DUT there exists the possibility of overtesting. Overtesting occurs when a fault is detected in a structural context that cannot ever occur in a functional context. This implies that ICs which are perfectly usable in the field are marked as bad and thrown away.

2.4 Design-for-Test

While combinatorial ATPG is able to reliably generate test patterns with as high as possible fault coverage in a reasonable time the same cannot be said about sequential ATPG. The problem with sequential circuits is that some faults may need a lot of clock cycles to be triggered or to be observable. To model multiple clock cycles the DUT is unrolled, i. e., the flip-flops are removed and their input lines are handled as outputs of the circuit and their output lines as input to the circuit. This approach makes sequential ATPG unfeasible already for a small number of clock cycles as the size of the resulting netlist rapidly starts to grow too much to be able to be processed in a reasonable fashion.

This is why DfT is applied. An example for DfT is scan design. In scan designs, all or most of the flip-flops are replaced by scan cells. These scan cells are able to function as normal flip-flops in operating mode but can form a so-called scan chain in test mode. This allows to use combinatorial ATPG since the state of the DUT is fully controllable.

A typical test with a scan design runs as follows: First, the DUT is put into test mode. Then a test pattern is shifted into the scan chain. The inputs are applied and the DUT is set to operation mode. A single clock cycle is applied and afterwards the DUT is again set to test mode, the results of applying the test pattern are shifted out and at the same time a new pattern is shifted in. This is repeated until all patterns were applied and results shifted out.

Another instance of DfT is BIST. In BIST, special circuitry is embedded into the IC to run tests either from memory or pseudo-randomly generated. It can either run concurrently or while the DUT is “offline”. An example for offline BIST is on-board diagnostic software. Also, SBST is another implementation of BIST.

3 System-Level Test: What is it and why do we need it?

This chapter introduces SLT in the context of IC testing. Section 3.1 presents the current day test flow for a SoC. It shows how SLT fits into the flow and why it is becoming more and more important. Its advantages and challenges are listed in Section 3.2 and how the latter are currently solved. Finally, in Section 3.3 SLT is compared to structural and functional test.

3.1 Definition and Position of System-Level Test

Before the 90's, functional test and system-level test were commonplace. But due to Moore's law [Moo06] and the general increase in complexity, it became more and more time consuming and expensive and an alternative was necessary. Thus, structural testing and DfT techniques became popular. But the effectiveness of structural testing recently starts to plateau and there is evidence that functional tests in form of SLT become necessary to augment them [Che18]. As such, around a decade ago manufacturers started to incorporate SLT into their test flow. In 2012 Biswas and Cory [BC12] published a paper about SLT in Graphical Processing Units (GPUs) and in 2014 Tipparthi and Kumar [TK14] published a paper about using SLT for SoCs.

According to Polian et al. [PAB+20], three definitions for SLT are predominant in the context of ICs. As in the aforementioned paper itself, the following definition is used in this thesis: SLT is used in outgoing quality control of the manufacturer to prevent the delivery of defective ICs, i. e., reduce DPPM, and improve its own quality control. Another definition is similar to this one but shifts the burden of SLT to the customer of the manufacturer for incoming quality control. The main difference between the first and second definition is the knowledge about the IC versus the knowledge about the application environment. In the first definition the former is greater while in the second this is true for the latter. The third definition is testing the complete system focusing on interactions between its components.

In SLT, the IC is embedded into an environment which corresponds to its end-user application one e. g., a motherboard with Dual Inline Memory Module (DIMM) and Peripheral Component Interconnect (PCI) slots where further devices such as Double Data Rate Random Access Memories (DDR RAMs) are connected. Then functional tests, such as booting an operating system and executing programs typical for its end-user application, are performed and if the IC does not cause a crash or fails in another way it is counted as passing SLT.

Generally, an IC goes through the following test steps:

Wafer Sort (WS) Test in wafer before dicing and packaging.

Package test or Final Test (FT) Test after packaging but before shipping to the customer.

Burn-In Long running test under high thermal and electrical stress to catch cases of infant mortality (early failures).

SLT Functional test of the IC in an environment approximating its end-user application.

Figure 5.1 shows the above test (or quality assurance) flow (without Burn-In). If Burn-In is performed Almeida et al. [ABC+19] show that it can be combined with SLT.

3.2 Advantages and Challenges of System-Level Test

In the following, the advantages and challenges of SLT are listed and how the latter are tackled currently.

3.2.1 Advantages

Beside hard (physical) defects, which are in most cases detectable under any circumstance, there are so-called marginal defects which manifest themselves only under certain conditions such as executing certain sequences of instructions, temperatures, voltages or operating frequencies. As lithography becomes more challenging by each generation, the number of faults due to process variations is increasing. Such defects happen often within boundaries of logical gates and standard cells at the level of smallest geometries where lithography becomes the most challenging. And these small process variations cause marginal defects [RAH+14]. Marginal defects can occur at random locations [Che16]. SLT is able to catch these marginal defects which cannot be covered by structural tests. Ryan et al. [RAH+14] point to the emerging number of marginal defects which underlines the necessity of SLT further.

Timing failures in low voltage operation are also hard to detect using structural test despite using more and more advanced fault models. This type of failure makes SLT especially important for battery powered SoCs which employ aggressive Dynamic Voltage and Frequency Scaling (DVFS).

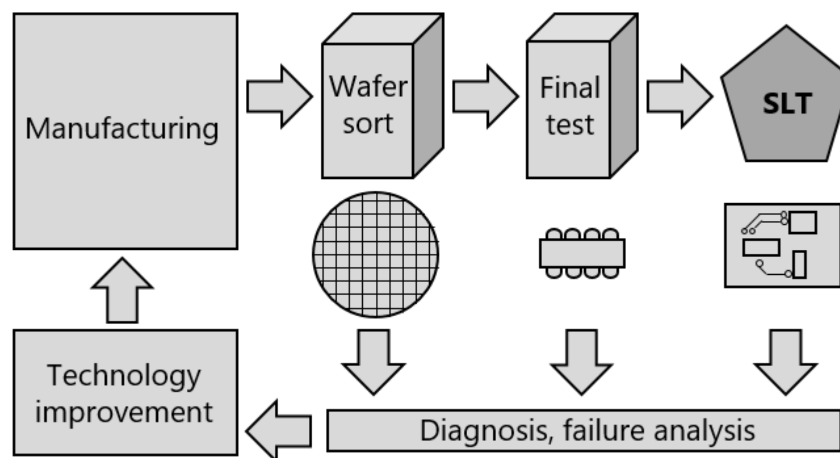

Figure 3.1: Testing flow for a generic chip (from [PAB+20])

With lower supply voltage, a gate's nominal delay increases, but Singh [Sin19] also shows that random gate delays caused by process variations are exacerbated by lower voltage, too, even more so than nominal delays. From this results an increase of DPPM [BRR+20].

Additionally, test mode behavior of ICs diverges more and more from real-world behavior [Che18]. Singh [Sin19] describes an example for this: Scan based structural testing only tries to "mimic" two cycles of functional operation which causes different circuit electrical conditions compared to continuous operation.

Power-aware ICs, for example SoCs used in smartphones, can put themselves into different power states in production. These power states are generally untestable by structural tests because a scan design relies on all scan chains being active but certain power states may disable some of them making testing impossible.

Customer returns are generally analyzed for test escapes. From this data, new test cases can be derived or existing ones modified. The SLT software test suite can be easily adapted to catch these test escapes, when requirements change or when a part has been determined redundant. This can be done by replacing programs or operating systems with different ones or leaving them out entirely. It allows to do these steps faster than Wafer Sort or Final Test since no expensive test recharacterization is necessary.

There are some additional cases which SLT can address compared to structural testing, such as complex clock and power domain interactions [ASH17], functional interactions during operating system boot or exercising the Central Processing Units (CPUs) with extreme work loads and more [BRR+20]. Finally, there are cases where SLT is able to detect faults which ATPG tools consider as "untestable" or which cannot be tested due to time and tester memory constraints [PAB+20].

3.2.2 Challenges and possible Solutions

SLT incurs high costs due to the following reasons [Che18]:

- test equipment acquisition and maintenance
- space for equipment and inventory
- human labor intensive effort for failure analysis and diagnosis

SLT is usually applied in addition to other testing steps rather than replacing them. That means it adds additional time and cost.

Additionally, SLT cannot be run on regular test equipment since periphery such as memories, peripherals and interfaces may be necessary for the intended workload [PAB+20]. The industry is currently working on SLT-oriented testers that can be reused across different products as much as possible.

An important aspect in testing is the coverage metric since it helps determining and improving test quality. Every testing step except for SLT uses fault coverage as a metric which will be further explained in Section 3.3. But fault coverage cannot be used in SLT as fault simulation cannot be applied to SLT workloads. Additional analysis and data mining is necessary to identify redundant tests or tests with low contribution to overall test quality. The currently used metric is the DPPM due to SLT.

Another disadvantage is the long run-time. SLT runs may take up to tens of minutes for complex SoCs which incurs higher cost. This can be mitigated by using more and/or specialized SLT equipment which is cheaper than Automatic Test Equipment (ATE) and even IC-level parallelism may be exploited to reduce testing time as Tipparthi and Kumar [TK14] demonstrated. But especially this long run-time makes fault simulation infeasible.

Another option to reduce testing time and effort would be to use data mining on previous test results to identify possible candidates which had noticeable timing deviations in previous testing steps [Sin19]. Adaptive SLT can not only decide about running or not running the tests for a specific chip but also the amount of tests [LTGW18]. This approach can be combined with machine learning using data from parametric and structural test [LPY+18] or even from other mature boards [LLCG19]. This increases throughput since not every IC has to go through SLT and less SLT testers are necessary [Che16].

Determining the root cause for a failure is more difficult in SLT since the effect of a failure may only be observable many cycles after the failure has occurred. Conventional testing has many efficient diagnosis methods available whereas SLT is lacking them currently. Methods from post-silicon validation, such as Quick Error Detection [LHL+14], can be used to mitigate this problem [Che18; PAB+20].

3.3 Structural Test vs Functional Test vs SLT

When comparing SLT to structural testing as described in Section 2.3, it is evident that both are fundamentally different. But it is also evident, that structural testing enjoyed a lot of research going into it. For example, scan designs are developed further which is evident with Reconfigurable Scan Networks (RSNs). In short, RSNs are networks of scan chains that can be connected almost arbitrarily to allow fast access to specific components on an SoC. This became necessary as scan chains became longer and longer with the increasing complexity of current-day SoCs. SLT does not have the advantage of decades of research going into it. Instead, it is still at the very beginning and research just recently start to put increasing effort into it. Recall, that currently there is no coverage metric despite DPPM nor any method to automatically generate test programs.

In the case of functional testing, the difference is harder to see. It appears that literature sees SLT as a form of functional test while Polian et al. [PAB+20] believe that there is a subtle difference. They argue that the approach of SLT is different enough to "traditional" functional test. For example, functional test vectors can be generated using evolutionary algorithms and they generally target a single component whereas SLT looks at the system as a whole. Functional testing takes significantly less time than SLT which allows to assess the quality of the functional test vectors by using fault simulation. SLT is able to detect different faults than functional tests, for example, functional testing may be able to uncover faults caused by a high temperature gradient between components when run thousands of times since temperature gradients need time to form themselves. An open question is, if and how much of SLT can be shifted to faster functional tests. To answer this question the cause for SLT-unique faults needs more investigation. Another argument for a difference between functional testing and SLT is that SLT can be described as running non-deterministic tests while the former together with structural testing is deterministic [TK14].

We side with the view of Polian et al. [PAB+20] because with the above arguments SLT appears to be distinct enough to not be classified as a kind of functional test. Additionally, another argument speaking for differentiating SLT from functional testing is the fact, that in SLT extra-functional parameters also play a role. There is power-aware test [TBD+15], where efforts are undertaken to avoid excessive power consumption during test to avoid effects such as accelerated aging. The general goal is to reduce power consumption during manufacturing and in-field tests and those tests are designed accordingly. But they are just considering these parameters and not actively controlling or observing them, as will be demonstrated in this thesis. With respect to marginal defects and the current understanding of SLT-unique faults, the ability to monitor and control extra-functional parameters becomes more important to be able to detect them. But future research on SLT-unique faults may invalidate that importance.

4 Related Work

This chapter lists some related work to this thesis. First, papers relevant to SLT in general are listed. They are extensively referenced in Chapter 3. These papers have been found by looking at the references of [PAB+20], which was provided by the examiner. Additionally, papers referencing and referenced by [Che18] were looked up. Finally, searches on Google Scholar [1] were performed with the keywords “System-level Test SoC” and the references of those findings were looked at, too. Then, papers representing applications for power devil programs are presented. These were chosen as related work, since this thesis will apply them, too. And it would be interesting to see, in which context power devil programs are generally used. These papers were found as a reference to [PAB+20] and by searching on Google Scholar for “power virus generation”.

In [Che18], the author points at the importance of SLT in the manufacturer’s testing flow. He describes SLT and lists why there is a need for it. He also mentions some of the challenges coming with SLT. Polian et al. [PAB+20] go into more detail on SLT and SLT-unique faults and analyze the challenges more thoroughly than [Che18]. They also differentiate SLT from conventional testing as introduced in Chapter 2.

The next few papers are handling case studies and practical applications of SLT. Biswas and Cory [BC12] conducted an industrial study for SLT for GPUs and talk about their experience with it. To the best of our knowledge, this is the first industrial study on SLT. Tipparthi and Kumar [TK14] give a first approach to speed up SLT by using concurrency and the fact that some tests can be executed in parallel. Almeida et al. [ABC+19] introduce a method to combine Burn-In testing with SLT. Bernardi et al. [BRR+20] conducted a study, too, where SLT is applied to an automotive SoC. They talk about the implementation and how SLT-unique faults were isolated.

An interesting approach to SLT which may use the results from this thesis is adaptive SLT. Singh [Sin19] and Chen [Che16] present an approach for adaptive SLT where results from parametric and structural testing are used to determine if the DUT should go through SLT or not. Letchumanan et al. [LTGW18] extend this approach to only run a select subset of all tests.

It seems that power virus program generation in general is more intended for power or performance estimation, maximizing CPU temperature or stability testing. Najeeb et al. [NKH+07] present a framework to generate power devil programs based on the behavioral model of hardware, i. e., Hardware Description Language (HDL) descriptions. The intended application of this framework is early power estimation of CMOS circuits. The framework developed by Ganesan et al. [GJB+10] incorporates genetic algorithms to generate power devil programs. They compare their framework with existing commercial software on the market and use three different ISAs. Ganesan and John [GJ11] introduce a framework that is adapted to multi-core applications but the intended application

[1] scholar.google.de

is the same. Huang and Mishra [HM17] extend the usage of power virus programs to reliability analysis of Intellectual Propertys (IPs), i. e., that in no case these IPs is vulnerable to inputs that cause a violation of its extra-functional properties.

Searching for “power virus testing”, “power devil testing”, “power virus generation testing” or “power virus functional test” on Google Scholar did not produce any useful results. As such it seems that, to the best of our knowledge, this thesis is the first to apply power devil programs for testing.

5 Case Study: Self-aware SLT using software-based adaptive Temperature Control

The goal of this thesis is to show that we can leverage on-die sensors to keep the temperature in a given target range. This is achieved by running a power devil program on a soft core and regularly measuring the temperature. This demonstrates an instance of a “greybox” model and how SLT can profit from the information collected from on-board sensors. We also compare three different algorithms used as power devil to show how software influences the behavior of hardware with respect to certain characteristics, which in our case is temperature. Another goal was to find out at which rate we can read the temperature without causing the regulation to fail. Regulation here means that the power devil program is controlled in a way to keep the temperature in a certain range. The regulation is seen as failing when either the target range is missed or the power devil program runs forever or never from a certain point on.

This chapter introduces the platform for the case study. First, the requirements posed at the platform and the workflow are presented in Section 5.1. Then, in Section 5.2 each component is introduced. First, the basics of each component is introduced. Then, features and properties of each are listed and finally it is discussed why they were chosen. And finally, in Section 5.3 the workflow for generating SoCs and running the case study programs on them is introduced and the platform itself. It is explained how the workflow is implemented, how it contributes to this thesis and later to AutoGen and some future improvements are listed. Then the structure of the hardware side of the platform is displayed and its contribution is laid out as well. Finally, an example run through the software side is given from the bootloader to the case study program itself.

5.1 Requirements on Platform and Workflow

In this section the requirements on the platform of the case study are laid out. Additionally, requirements for a workflow that established itself throughout the thesis are presented, too. The requirements are posed not only in relation to this thesis but also to the project AutoGen introduced in Chapter 1.

Fundamentally, the cores used in this thesis must be able to run on an FPGA and must be able to generate enough heat under load to allow for any regulation. The FPGA must provide an on-die temperature sensor. Also, some sort of communication interface besides the programming interface is required since we need to get the temperature measurements as results. The core itself must be easily customizable such that we are able to create many different cores to examine. It must use a relevant ISA so that the results are applicable in the real world. Optimally, it is royalty-free. It is desirable to be able to create a workflow that can be as much as possible automated using Continuous Integration/Continuous Deployment (CI/CD) since AutoGen will most likely generate many cores to create the extra-functional model of hardware for SLT program generation. The core

must also be able to connect external periphery in a simple manner using commonly used interfaces such as Advanced eXtensible Interface (AXI) since other sensors may be connected and interactions with the outside world must be possible.

On the software side it was decided to use an operating system since bare-metal development would have taken too much time. Thus, the chosen operating system must be small, fast to boot and set up and it optimally has an existing port for the chosen ISA. Also, it should not have high requirements to the platform itself. The power devil algorithms should be simple but make intensive use of arithmetic computations to achieve high activity on the core and thus heating up the FPGA.

5.2 Components of the Platform

In this section all non-generic components used in the platform and workflow are listed. Features and properties of these components are introduced and it is discussed why they were chosen.

5.2.1 RISC-V

As the ISA for the cores we chose RISC-V. It is a promising, open standard and royalty-free ISA which has great flexibility due to its modularity. RISC-V is a young ISA compared to x86 and ARM but is gaining traction with SiFive[1] as one of its biggest contributors at the time of writing. The first specification was released in 2011 by Waterman et al. [WLPA11]. It originates from the University of California - Berkeley Architecture Research (UCB-BAR), which also founded SiFive. RISC-V has some similarities to the MIPS ISA (and SPARC). It is a little endian architecture.

A fully extended RISC-V core (ignoring experimental extensions) with a 64-bit architecture consists of these extensions:

I basic Integer, Logic and Jump instructions, required

M Integer multiplication and division, optional

A Atomic instructions, optional

F Single-precision IEEE floating point instructions, optional

D Double-precision IEEE floating point instructions, optional

C Compressed instructions, optional

Instructions in RISC-V are generally 32-bit wide but using the C-extension they can be compressed to 16-bit for common operations, e. g., operations that involve the stack pointer or the zero register. The above specification can be shortened to RV64IMAFDC (or shorter RV64GC, where G is used as an abbreviation for IMAFD).

[1] https://www.sifive.com/

There is also a 32-bit ISA specification, which is called RV32GC analogously, and for the future a 128-bit ISA is intended but not specified yet. Many extensions are being worked on, for example the V extension: a vectoring extension comparable to the SSE or MMX extensions on x86 processors. Furthermore, the ISA can be extended by user defined instructions, e. g., for coprocessors.

As the name implies, RISC-V is a Reduced Instruction Set Computer (RISC). It has 32 registers with either 32, 64 or 128 bit (in the future) width, where register "zero" is hardwired to all zeroes. But the number of available registers can be reduced to 16 using the so-called E extension, intended to replace the I extension for small embedded 32-bit processors. It is, at the time of writing, still in draft status [WA19].

Volume I of the RISC-V specification [WA19] defines cores as components with their own instruction fetching unit. Cores can have multiple so-called Hardware Threads (Harts) but by definition at least contain one. A core with multiple Harts can be compared to Intel processors with hyper-threading, i. e., a single physical core with multiple logical cores. Harts get their instructions from the instruction fetcher of the core they belong to.

RISC-V fulfills all requirements on the ISA: it is relevant, used in the industry and royalty-free. On top of that it is modular, which allows us to test varying designs of SoCs. And finally, the next component further justifies the choice of RISC-V.

5.2.2 Chipyard

Chipyard is a framework to generate RISC-V based SoCs in an agile environment. It is developed by the UCB-BAR and is written in Scala. The idea is to provide a system to generate SoCs depending on one's needs and incorporate it into a CI/CD pipeline. Chipyard does not only generate a Verilog netlist of the generated core but also compiles simulators for it. These simulators can be used to check if the software intended to run on it behaves as expected and also allow for easy debugging.

The Verilog netlist can then be plugged into either an ASIC flow or an FPGA flow. For ASICs, Chipyard offers a tool called Hammer to automate the flow. It supports steps from synthesis over place and route to power simulation including Layout-vs-Schematic verification and simulations between each step. For more details see the paper by Wang et al. [WIS+18]. The FPGA flow has no such tool nor support by Hammer at the time of writing.

Chipyard provides three different RISC-V core designs: Rocket [AAB+16], Berkeley-Out-Of-Order (BOOM) [ZKGA20] and Ariane [ZB19]. Ariane has been renamed to CVA6 but for this thesis we will stay with Ariane since that change has not been reflected in Chipyard yet. The core designs are compared in Section 5.2.4.

Cores are generated based on a configuration. These configurations consist of several configuration fragments. Chipyard itself provides many fragments but the user is able to add more. The user is also able to integrate custom periphery, e. g., a SHA3 accelerator.

Listing 5.1 shows an example for a core configuration. The SoC consists of a single Rocket Core with branch prediction units, a Floating Point Unit (FPU) and support for user mode. It has no ports for external interrupts. An L2 cache is connected. It can be configured for optimal performance but

```
class RocketConfig
  extends Config(
    new chipyard.iobinders.WithTieOffInterrupts ++
      new chipyard.iobinders.WithBlackBoxSimMem ++
      new chipyard.iobinders.WithTiedOffDebug ++
      new iobinders.WithDontTouchPorts ++
      new WithSimAXIMMIO ++
      new WithBootROMFile(s"software/bootrom/bootrom.bin") ++
      new chipyard.iobinders.WithTraceIO ++
      new chipyard.config.WithL2TLBs(1024) ++
      new freechips.rocketchip.subsystem.WithExtMemSize((1 << 30)) ++
      new freechips.rocketchip.subsystem.WithNoSlavePort ++
      new freechips.rocketchip.subsystem.WithInclusiveCache() ++
      new freechips.rocketchip.subsystem.WithNExtTopInterrupts(0) ++
      new freechips.rocketchip.subsystem.WithNBigCores(1) ++
      new freechips.rocketchip.subsystem.WithCoherentBusTopology ++
      new freechips.rocketchip.system.BaseConfig
  )
```

Listing 5.1: Example Core

in this case it is left at its default parameters (16 kiB, 8-way associative) The external memory has a size of 1 GB. It is connected to the Rocket core via AXI. The example SoC does not use the Slave interface. Chipyard SoCs can provide three AXI buses:

- The memory bus—a Master AXI interface connected to a memory controller
- The Memory-Mapped I/O (MMIO) periphery bus—a Master AXI interface connected to any periphery
- The Slave bus—a Slave AXI interface to connect, e. g., a Direct Memory Access (DMA) controller

This SoC has a L2 Translation Lookaside Buffer (TLB) with 1024 entries. Chipyard offers a so-called TraceIO interface to monitor each core and Hart while the system is running. This is especially useful when using an FPGA since an integrated logic analyzer can be used to watch the cores. The SoC uses a custom BootROM. All statements above the BootROM in Listing 5.1 are intended for the test harness. They tell the test harness how to handle all interface ports of the core. The test harness provides the necessary environment for building the simulator as it connects or ties-off every bus and pin and provides simulated memory.

Chipyard allows to mix and match different cores. For example, one could design a SoC with a small Rocket core as controlling core and one or two BOOM cores for high performance tasks. There are additional interfaces available. At the time of writing there are options to equip a GPIO, SPI, I2C, Ethernet and an Universal Asynchronous Receiver Transmitter (UART) interface. For more details see the paper by Amid et al. [ABG+20].

As demonstrated with the small example core, Chipyard allows us to easily generate multiple different cores. And it can be by design used in a CI/CD pipeline. Having this flexibility also underlines the choice of RISC-V since, at the time of writing, it is to the best of our knowledge unknown if there exists a comparable framework for other ISAs.

5.2.3 Operating System and Power Virus Algorithms

FreeRTOS[2] is an open source real-time operating system which is administered by Amazon. FreeRTOS requires a low amount of RAM[3] and can easily run on an RV32I RISC-V core. It requires a CPU that can handle interrupts and a timer for the scheduler. No Memory Management Unit (MMU) is necessary. FreeRTOS is intended to be used on a single core. There are no mechanisms that allow Simultaneous Multi-Threading (SMT). It was chosen over running on bare-metal since it provides all necessary basics (memory management, task switching) without having high requirements. Additionally, a RISC-V port for FreeRTOS already exists, though, it had to be adapted to RV64GC and code handling the FPU had to be added as well as the original port was written for an RV32I core.

The chosen power devil algorithms are:

- Mandelbrot (floating point)
- Mandelbrot (fixed point)
- SHA-256

The Mandelbrot set [Man80] is defined by the set of complex numbers c where the following function is bounded when iterating from $z = 0$:

$$f_c(z) = z^2 + c$$

The Mandelbrot algorithm used in this thesis calculates a 3940x2160 pixel image with maximum 320 iterations per pixel from $-2 - 1i$ to $1 + 1i$. It is an algorithm that is used to assign color values to pixel to produce an image for visualization purposes. To display a Mandelbrot set, the algorithm iterates the above function until it reaches a certain value, generally $4 = |z|^2$ is chosen. If the maximum number of iterations is reached before this bound, the algorithm assumes that for the value c the above iterations diverge and moves on to the next one.

The fixed point Mandelbrot avoids floating point calculations at all costs and uses a fixed point library for this purpose. There are only two operations which involve floating points: the conversion of the input argument to fixed point and vice versa.

SHA-256 is a part of the SHA-2 family of cryptographic hash functions developed by the United States National Security Agency (NSA) and standardized by the National Institute of Standards and Technology (NIST) [NIS02]. The 256 in the name indicates the length of the digest, i. e., the output of the hash function, which is in this case 256 bits long. The SHA-2 family mainly uses bit shifts and rotations, logical operations (and, xor) and add operations.

The algorithm for the power devil can be chosen at compile time. There is no benefit to running multiple power devil algorithms at once due to the task switching overhead. The tick frequency in FreeRTOS, i. e., how often the scheduler runs, is equal to the frequency at which the temperature is

[2] https://www.freertos.org/

[3] https://www.freertos.org/FAQMem.html shows an example how much RAM is used and also provides an idea of context switching time, though that depends strongly on the platform and the quality of the port.

Name	Language	Supported ISAs	Pipeline design	Pipeline Stages	Author
Rocket	Chisel	RV32GC and RV64GC	in-order	5	UCB-BAR
BOOM	Chisel	RV32GC and RV64GC	out-of-order	7 (10)	UCB-BAR
Ariane	SystemVerilog	RV64GC	in-order	6	ETH Zürich

Table 5.1: Comparison of available RISC-V cores

read. This reduces scheduling overhead and maximizes idle time since there will be no unnecessary scheduler runs to check if another task is available. The power devil algorithm runs as a task in FreeRTOS in an infinite loop.

These algorithms were chosen because most of them are integer based and simple to implement. The single instance of the floating point Mandelbrot is included to see how the temperature behaves when using the FPU instead.

5.2.4 Rocket vs BOOM vs Ariane

Here we compare all three RISC-V core designs that are available in Chipyard. Table 5.1 gives an overview.

The three available designs are: Rocket, BOOM and Ariane. Rocket and BOOM are both written in Chisel and developed by the UCB-BAR. Chisel [BVR+12] is an HDL based on Scala. Ariane (or CVA6) is developed by the ETH Zürich and is written in SystemVerilog. Rocket can be seen as the reference implementation of a RISC-V core in the context of Chipyard or the UCB-BAR. Many modules from Rocket are intended to be re-used as was done for BOOM. BOOM's goal is to be used as a design for studying out-of-order architectures. ETH Zürich strives to reduce the critical path length with their Ariane core.

Rocket and Ariane are both designs with an in-order pipeline while BOOM's is out-of-order. All three designs support the RV64GC ISA but only Rocket and BOOM also have support to generate RV32GC cores. All three designs contain a branch prediction unit and a MMU. They are able to run Linux. Each design contains an L1I and L1D cache each which can be configured. Generally, all three designs share most configuration options in Chipyard, such as cache size and associativity, FPU pipeline length or branch table size and many more. Rocket has the least number of pipeline stages with 5 stages, while Ariane is implemented with 6 stages. Also, as can be seen in Table 5.1, BOOM has actually ten stages, but several of them are collapsed to one, which is why it is implemented with seven. It has the following pipeline stages: Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback and Commit. In the current implementation Decode/Rename, Rename/Dispatch and Issue/RegisterRead are combined and Commit is not counted as a pipeline stage by the developers as it occurs asynchronously. In contrast to Rocket and Ariane, BOOM cores contain multiple Harts per core, called execution units. These execution units can execute instructions in parallel but are only fed by a single instruction decoder, as mentioned in Section 5.2.1.

It is important for this thesis that the chosen design generates a high enough amount of heat when under load to allow for any meaningful regulation. Also, the fact that FreeRTOS is only implemented for single-core systems poses an additional constraint to the choice of the design. As such, BOOM has been chosen, because it is able to generate enough heat for meaningful regulation, as will be seen in Chapter 6, without any changes to FreeRTOS.

A single Rocket core was unable to generate a high temperature difference. The biggest measured increase was around 2 °C. This small increase does not allow for any meaningful regulation. Thus it was decided not to test Ariane, since its design isn't as fundamentally different compared to BOOM and even more limiting—the authors discourage the use of multiple Ariane cores. Also, due to the fact that Ariane is written in SystemVerilog, a generated Ariane core contains one additional file—the core itself—which has been found out after the workflow was finished. The workflow would have to be adapted to respect that additional file while exchanging Rocket with BOOM and vice versa is possible without any changes.

As mentioned above, FreeRTOS is not designed for multi-core systems and it would have taken too long to adapt it to such. Asynchronous Multi-Threading (AMT) could be realized but BOOM's execution units make it easier to leverage the increased area and activity for heat generation without making any changes to FreeRTOS. In AMT, every core runs its own instance of FreeRTOS. In contrast, in SMT all cores run a single instance of FreeRTOS and every core knows about every other core. AMT comes with its own set of problems in implementation. The available memory space would have to be partitioned for each core. The bootloader will be run by all cores and as such they have to be orchestrated, to not interfere with UART booting. Additionally, FreeRTOS would have to be compiled and changed in such a way that it is position independent, i. e., it does not care where in memory it is loaded. Furthermore, the access to the UART interface and the Xilinx® Analog-to-Digital Converter (XADC) would have to be coordinated, since they cannot be duplicated. As such, exploiting the super-scalar nature of BOOM is the best solution for this thesis. The availability of multiple Harts in turn influenced the decision on the power devil algorithms since a higher heat generation is possible when all available Arithmetic Logic Units (ALUs) can be used as much as possible at the same time, due to BOOM's super-scalar pipeline.

Given the requirements in Section 5.1 and the constraints from the choice of operating system and core design, the currently selected components for the case study seem to be the best solution.

5.3 Building the Platform and Workflow

In this section the previously listed components are now connected to build the platform for the case study and the workflow. It first starts with the workflow. It is explained how a sample configuration together with some code for a power devil program create the results for the case study. Then the hardware side of the platform is presented and finally the software execution is illustrated.

5.3.1 Workflow

Here we introduce the workflow from the SoC configuration up to receiving the results. Figure 5.1 shows a flow chart of it.

The three inputs to this workflow are a Chipyard configuration, the source code for the BootROM and the source code for the case study program. Since Chipyard is intended to be used in an agile environment we decided to use it in Jenkins[4]. The Jenkins instance is part of a Kubernetes[5] Cluster and uses Pods for each execution of a pipeline.

First the Compiler Pipeline is run, since Chipyard depends on the compiled BootROM, and afterwards the Chipyard pipeline runs. Chipyard is very resource intensive, especially on RAM—it needs around 10 GB of RAM for generating a BOOM core.

When Chipyard has finished generating the core, the netlist is transferred to a server where Vivado is available. There, tcl scripts orchestrate the synthesis steps. In the end Jenkins collects the bitstream and the binary of the case study program.

The next steps are not automated yet. The bitstream is loaded onto the FPGA and afterwards the binary is streamed onto it via UART. The program runs then and prints the temperature over UART, which is collected by a Python script. It also prints if the power devil task is running or suspended. The Python script then stores the measurements into a csv file.

The benefit of this workflow is that it is mostly automated now, as indicated by the asterisk at certain steps in Figure 5.1, and will be fully automated in the future. Especially, since synthesis takes a long time, it is of advantage to automate every step to save time and also to parallelize the generation of multiple configurations with ease. For a full automation at least one additional pipeline is necessary that will run after the other two are done. That third pipeline will then execute scripts to first load the bitstream onto the FPGA, then another script to load the program and finally the Python script to collect the measurements. The biggest challenge here is that only a single FPGA is available, so multiple cores must be applied in sequential order and put in a queue and each run should uniquely identify which design with which program was used. Besides fully automating it, there are other steps that should be tackled in the future, for example, load balancing the synthesis over multiple servers. Currently, a single machine is responsible. Additionally, when the BootROM is stable, it can be integrated into the Chipyard pipeline without the need to always recompile it. This makes the Chipyard and the Compiler pipelines more independent of each other. Finally, more resources must be provided to the Jenkins instance since the resource intensity of Chipyard caused some problems.

5.3.2 Platform: Hardware

The cores were synthesized for a Xilinx® Virtex®-7 FPGA VC707[6] evaluation board equipped with a XC7VX485T. The XC7VX485T provides 485,760 Logic Cells. The cores were connected to the RAM on the board, its UART interface and also to the XADC. The evaluation board contains 1GB of DDR3 RAM. The fan on the evaluation board was disconnected since it was unclear how it behaves when the FPGA chip heated up. It also reduced the maximum achievable temperature gradient of all designs enough to make regulation impossible. The target frequency is 50 MHz.

[4] https://www.jenkins.io/
[5] https://kubernetes.io/
[6] https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

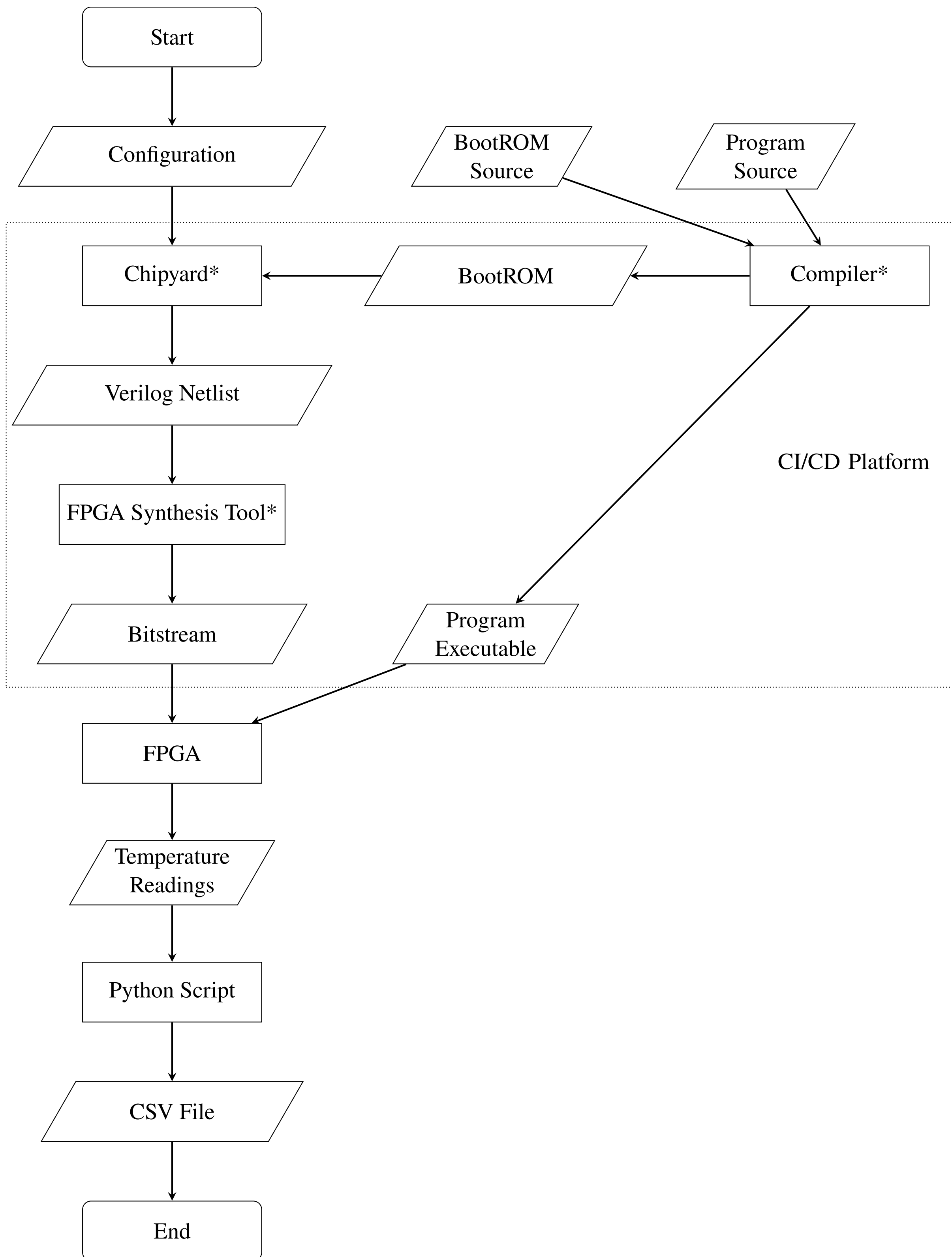

Figure 5.1: Established flow

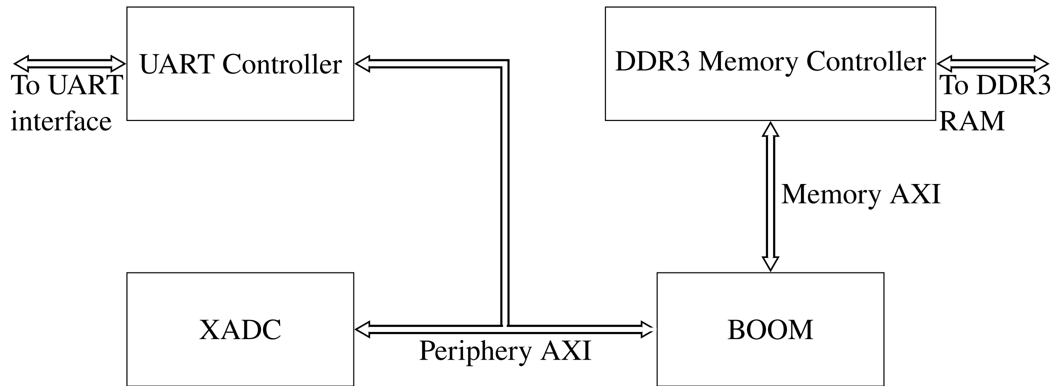

Figure 5.2: Block diagram of hardware design

Higher frequencies were very hard to achieve, especially when including the L2 cache. Thus, it has been removed from all tested cores. Figure 5.2 shows a simplified block diagram of the hardware design loaded onto the FPGA.

The XADC[7] is able to measure on-die temperature, different voltages of the FPGA and externally supplied voltages. It supports multiple sampling modes, e. g., sequentially sample multiple channels or sample them all at once, but for this thesis the single channel sampling mode was chosen. The XADC is able to return an average over multiple samples. We chose to use an average over 128 samples to smoothen the temperature curve. The automatic gain and offset correction has been enabled. It has support for multiple alarms, e. g., over-temperature, under-temperature, over-volt, etc., which can individually be configured on runtime, but they are not used in this thesis.

5.3.3 Platform: Execution and Control Loop

In the following a single run of the program (FreeRTOS with power devil algorithm) is illustrated with every step from loading it onto the FPGA to the control loop and the role of each FreeRTOS task.

The program is transferred onto the FPGA over UART with a speed of 115200 baud. The custom BootROM is used as a UART bootloader. Loading the program like this allows more flexibility in testing different power devil algorithms. A downside is the slow transmission speed. That is the reason why the chosen operating system must be small. Also, the FPGA has to be reset every time before loading a new program. There was also some confusion about how to load the program into RAM.

The bootloader itself does little setup and disables all interrupts. It then receives the program from the UART interface and collects each received byte in a register. Writing 8 bytes at a time is more efficient than writing every single byte alone and avoids alignment issues. When 8 bytes were collected it checks if they are equal to the end signature of the binary. If not, the data is transferred to memory, otherwise it will start the power devil task. As it turned out, the instructions have to be

[7] https://www.xilinx.com/support/documentation/user_guides/ug480_7Series_XADC.pdf

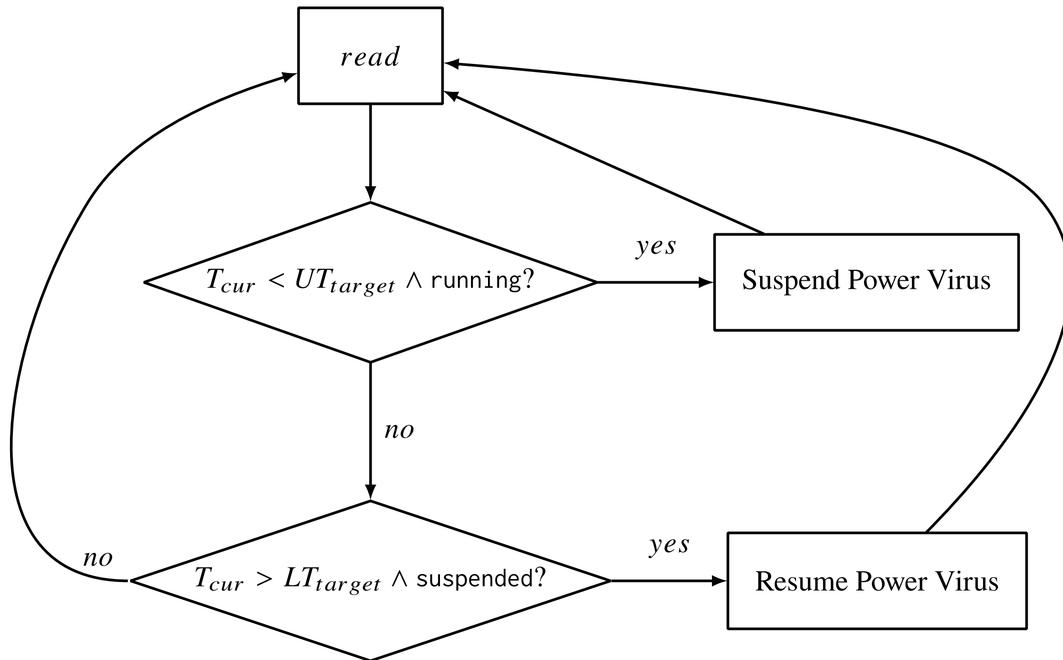

Figure 5.3: Flow chart of control task

transferred in big endian and every instruction is swapped with the next, ƨidɈ ɘkil. The bootloader does not know the size of the power devil binary in advance. Thus, a signature at the end of the binary is used to let the bootloader know when transmission is done.

Besides the power devil task, another task, called the control task, exists that reads the temperature periodically and either suspends the power devil or resumes it depending on the current temperature. As mentioned before, the control task allows the power devil task to run until the upper bound target temperature has been reached. It then suspends the power devil task until the measured temperature falls below the lower bound. Afterwards it will resume the power devil task and it goes on like this in a loop. It prints over UART the measured temperature and if the power devil task is suspended or running. Figure 5.3 shows the state-machine for the control task. UT_{target} is the upper bound target temperature and LT_{target} the lower bound target temperature respectively. T_{cur} is the last measured temperature. `suspended` and `running` indicate the state of the power devil task.

When no task is active, i. e., the power devil task is suspended and the control task waits for the next measurement, then an idle task is run which suspends the processor by issuing an `wfi` (wait for interrupt) command which basically halts the CPU. This is an easy but effective way to minimize activity and thus cool down the FPGA.

There exists the possibility to use interrupts which fire when the temperature becomes too high leveraging the XADC's alarms. But for ease of implementation it was decided against using them. The advantage of the interrupt driven design would be a faster reaction when reaching the target temperature since there is no fixed measurement period. For this thesis though, that does not play a role.

6 Results: Self-aware SLT using software-based adaptive Temperature Control

This chapter illustrates the results from the case study introduced in Chapter 5.

For the case study a core based on a BOOM configuration was generated, called LargeBoom. Large in this case refers to the number of Harts in the core. Details are shown in Table 6.1. The Chipyard configuration can be found in the Appendix (see Listing A.1). Note that the maximum temperature has been determined by letting the fixed point Mandelbrot algorithm run on the FPGA for about 20 hours. Ambient temperature was about 22 °C throughout every run. The target temperature band was between 65.5 °C and 66.5 °C.

Before every run the FPGA was allowed to cool down by resetting and clearing the current programming for 10 minutes, which means there was absolutely no activity on it. It then was programmed and the program for the case study was loaded onto the FPGA and run for 20 minutes.

Each Hart in LargeBoom contains 2 issues (operation units): an ALU and a divisor, an ALU and the FPU and finally an ALU and a multiplier. Issues cannot run in parallel.

LargeBoom takes up 85.67% and the full design takes up 94.3% of all available LUTs in the FPGA. As such, it is as big as possible and thus better suited to generate heat than the smaller design variants Small (1 Hart) and Medium (2 Harts), but is still able to fit into the FPGA compared to the bigger variants Mega (4 Harts) and Giga (5 Harts).

In the following, the results of running each algorithm on the core are presented and analyzed. The following graphs in Figures 6.1 to 6.3 each start from reaching 62 °C. Also, for each run 6 minutes are shown. This choice was made to show a part of the unconstrained temperature increase while also showing enough details about the regulated temperature. Each graph contains the raw data (green), raw data after applying a moving average with window a size of ten (red) and the lower and upper target temperatures (orange and blue respectively).

Looking at the graphs in Figures 6.1 to 6.3 the power devil task and the control task are working indeed as intended and the temperature is kept between 66.5 °C and 65.5 °C.

LUTs (percent of max. available)	Harts	ALUs	Max. Temperature
260084 (85.67%)	3	3	74.5 °C

Table 6.1: Details about LargeBoom

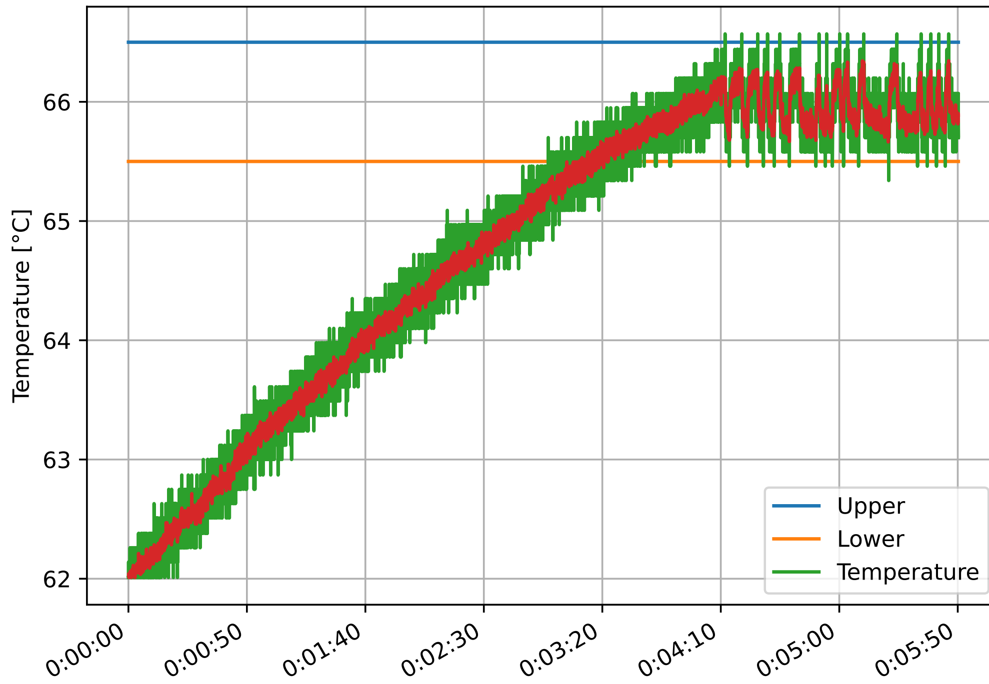

Figure 6.1: Temperature curve for Mandelbrot with floating point arithmetic

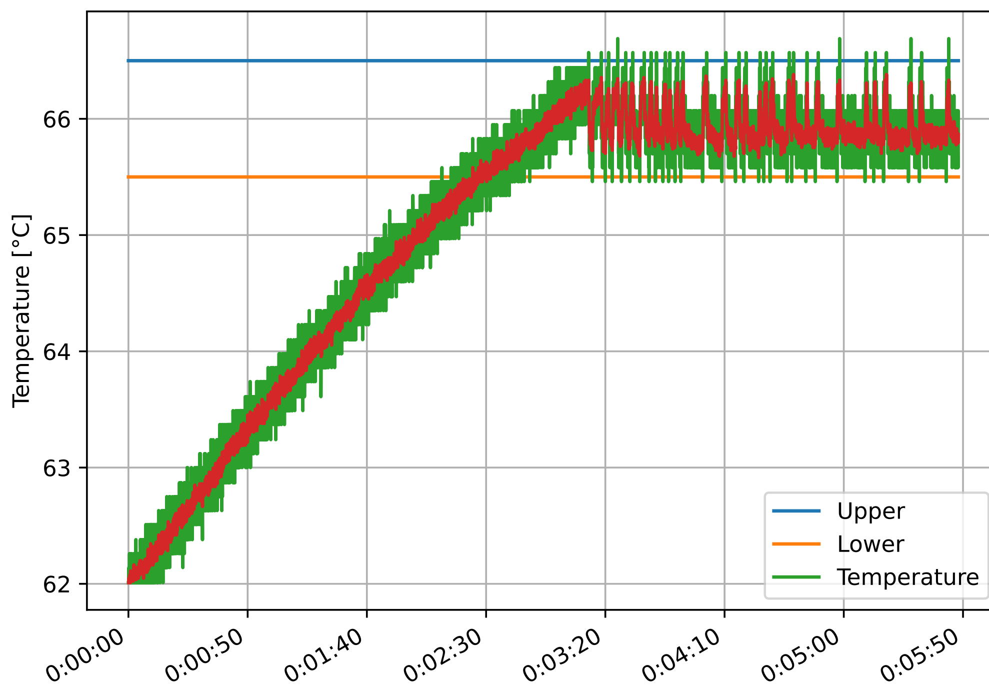

Figure 6.2: Temperature curve for Mandelbrot with fixed point arithmetic

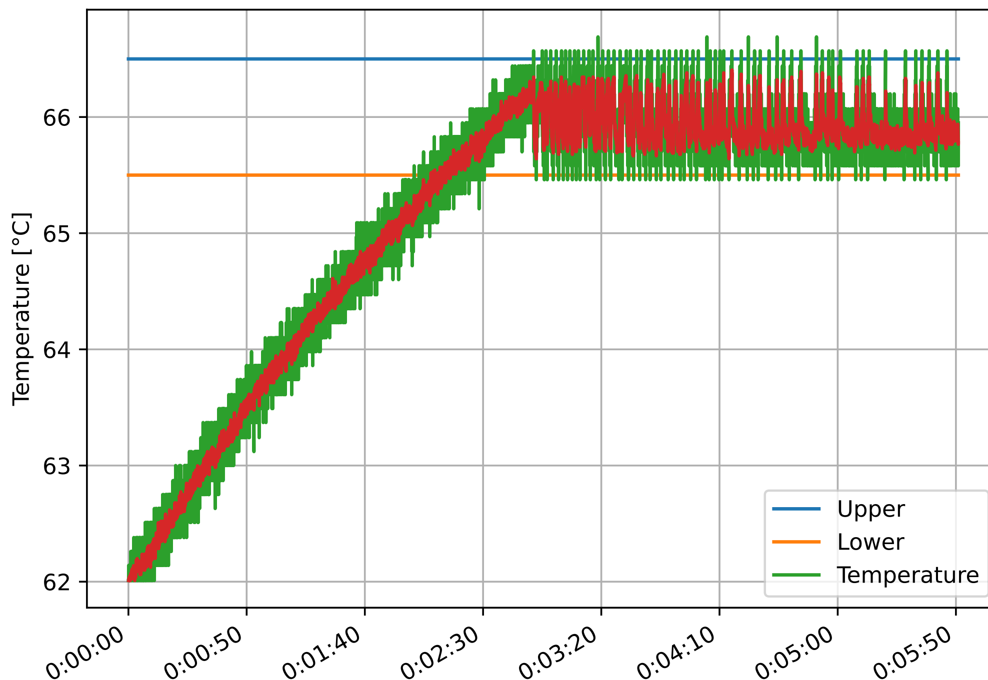

Figure 6.3: Temperature curve for SHA256

When comparing the graphs with each other, it can be seen that the Mandelbrot algorithm using floating point is the slowest to reach the upper bound temperature for the first time and also does not have as low of a suspend-resume cycle time than the other two algorithms. This can be easily explained by the fact that there is only one FPU compared to the multiple ALUs in the core and thus only one Hart is really used.

As for the reason why SHA-256 is increasing the temperature faster than fixed point Mandelbrot, it probably is due to the fact that there are several occurrences of FPU commands in the latter which weren't eliminated while the former is completely free of them. Important to mention here is also that context switching takes longer when the state of the FPU is dirty, i. e., it has been used in the previous task. RISC-V offers the ability to check if the FPU's state has to be preserved when context switching or not and thus reduce time spent in context switching. The FPU itself also has 32 registers which all have to be saved in the general case.

Another reason for SHA256 running the FPGA hotter is that the fixed point library for Mandelbrot is issuing many function calls. Additionally, SHA256 uses only ALU operations while Mandelbrot (fixed) also uses the multiplier which can only run on a single Hart.

It can be observed in most graphs that the rate of suspending and resuming the power devil task is decreasing over time. This can be best seen in Figure 6.2. Especially over a longer period it seems that the average temperature of the FPGA is increasing. This can exacerbate itself so much that it is no longer possible for the power devil to run again. The cause for this is probably that the temperature reading and task switching takes time which is taken away from the idle period. And both actions cause activity on the core which generates heat. As such, over time, the FPGA is not cooling down enough and it starts to reach an average temperature which is higher than the lower

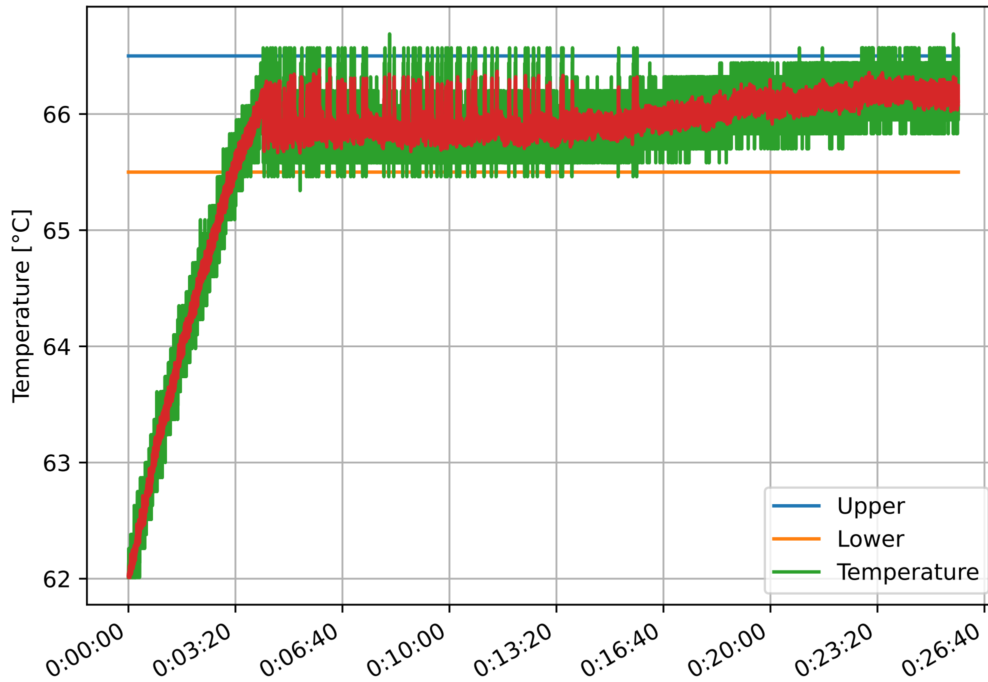

Figure 6.4: Temperature curve for floating point Mandelbrot over longer period of time

bound. This temperature is reached earlier the shorter the reading period. It turned out that for every period lower than 10 ms this prevents nominal regulation and instead causes the temperature to rise constantly but slower than under full load with the power devil running. This effect does occur for 10 ms but is not pronounced enough to prevent regulation.

For this thesis we will call this effect "software-based thermal runaway" (short: "thermal runaway") since the temperature seems to get out of control of the regulation loop with software being the cause. This instance of thermal runaway is comparable to the hardware-based one described by Vassighi and Sachdev [VS06] but not as fatal, as it only prevents regulation. And the primary cause for the hardware-based runaway effect is different, i. e., a physical defect causes higher power consumption which increases the temperature which causes even more power consumption and so on, but the symptoms are similar. In both cases more heat is generated than is removed and the consequence is an uncontrollably rising temperature. The effect is demonstrated in Figure 6.4 and can be observed after around 16 minutes.

The actual idle time in milliseconds can be estimated by using the following formula:

$$t_{idle} = p - 2t_{switch} - t_{read}$$

p is the period of the control task in milliseconds and the scheduler. In this case it equals to 10 ms. But the scheduler also runs when the control task suspends itself and then activates the idle task, which then allows the FPGA to cool down. Due to how scheduling in FreeRTOS works, the control task runs every p ms after it was last suspended. Thus, t_{read}, i. e., the time it takes for the task to run, is missing from the possibly available idle time. In this time period p two context switches are

happening: from any other task to the control task and from the control task to the idle task. A rough estimate, based on the number of instructions for task switching and temperature reading, results at about 1 ms spent not idling.

Another cause for this effect may be the distance of the target temperature to the idle temperature. It might have been too low with around 6 °C above idle temperature. And since the core is not only idling in the period where the power devil task is suspended the actual operating temperature must be higher than the idle temperature. As can be seen in Table 6.1, the maximum temperature that the core reached is about 8 °C higher than the target temperature. But due to time constraints it was not possible to verify the assumption that the aforementioned idle to target temperature gap is the cause.

7 Conclusion

We have shown the feasibility of the "greybox" approach intended for the AutoGen project by conducting a case study. This case study uses an on-chip temperature sensor of an FPGA to regulate the temperature, i. e., keep it inside a target range. And the results of the case study show that we are able to regulate the temperature. We have also shown how software choice can influence temperature. We have also found out how parameters such as the measurement period influence the temperature and the control loop itself. Finally, we also developed a basis for the AutoGen project with the platform and workflow introduced in Chapter 5.

The smallest possible measurement period has been found and some observations were made surrounding it. The optimal period in this case study was found to be 10 ms. Any shorter, and an effect, called "software-based thermal runaway" in this thesis, occurs which makes temperature regulation impossible. Software-based thermal runaway describes the effect, that the "non-power virus" period in the regulation still causes enough activity, such that the average temperature on the IC rises above the lower bound temperature. As a consequence it prevents the power devil program to ever be run again. In the case study, a temperature band was chosen to show more clearly that and how the regulation is working, but such bounds may be used in SLT, albeit with a greater distance between the upper and the lower bound. Also note, that in the case study the SoC is doing nothing while idling as it simply waits for the timer interrupt to occur. In real world applications this time may be used for other tests that generate a small load on the processor itself. For example, this can be used to cause a big temperature gradient to occur, which also is a possible trigger for marginal defects. But even if the upper and lower bound target temperature were to be the same, software-based thermal runaway can become a problem.

Let's consider two scenarios: The first one has a lower bound that is 1 °C lower than the upper bound. In the second scenario, both bounds are equal. In the first scenario, if thermal runaway occurs, the average temperature of the IC can either rise to in between both bounds and stay there or it can rise beyond the upper bound. The first case prevents the IC to hit the upper bound temperature which may be a limit to trigger certain marginal defects. And it also prevents the occurrence of temperature gradients. The second case is similar to the second scenario. A possible SLT program could try to hold the upper bound temperature for a fixed period of time before letting the IC cool down. In the second scenario, there will inevitably be a point in time where regulation starts and stops the power devil program in a very small number of measurement periods (low single-digits). And it can happen at the latest at this point, that thermal runaway will cause the temperature of the IC to go beyond the target temperature.

As discussed in Chapter 6 the difference between idle temperature and the lower bound target temperature may play a role. Since context switching and measuring is not actually letting the SoC idle but cause a bit of activity the average temperature of the IC will inevitably rise. This may play a big role when combining SLT with Burn-In [ABC+19], since in Burn-In all ICs are exposed to a very high ambient temperature with the goal to let them reach high die temperatures,

for example 110 °C, for an extended period of time. That in turn raises the idle temperature. And if idle temperature and target temperature are not sufficiently apart from each other then the activity caused in the idling or “non-power devil” phase may increase the IC’s temperature over time enough for thermal runaway to occur. And what can happen then is that hardware-based thermal runaway as described by Vassighi and Sachdev [VS06] occurs. Many modern SoCs have an emergency shutdown mechanism on reaching a certain temperature. Now, if thermal runaway in the SLT context occurs and the upper bound is near the shutdown limit, then that might cause the SoC to be marked bad while it isn’t. And such cases are very hard to diagnose, because it is not clear if this SoC has been marked bad due to bad test design or due to an actual defect which causes hardware-based thermal runaway.

In the end, further investigations into software-based thermal runaway have to be made. Software-based thermal runaway may become a major problem when generating software as is intended by the AutoGen project. As such, it has to be researched on how to detect its occurrence and how to avoid it in software generation.

An important part of the basis is the workflow. But, as mentioned in Section 5.3.1, further work on it is still necessary. It has to be fully automated and the BootROM should be incorporated into the Chipyard pipeline to reduce the dependency between it and the Compiler pipeline. Improvements such as distributing synthesis tasks between multiple servers are desirable.

Additionally, for AutoGen, more sensors shall be incorporated such as current sensors, voltage sensors, delay sensors and more. An interesting goal would be to include the IEEE 1687 (aka RSN) infrastructure developed for the BASTION project [JLL+17] or similar. The most interesting feature of that project for future work is the incorporation of aging monitors. It has to be tested if and how they can be used or, more generally, RSNs connected to components and sensors can be leveraged for SLT.

The thesis itself concentrated on a single core design, namely a single core BOOM. In the future, work is necessary to either modify FreeRTOS for multi-core use or find another operating system, that can easily replace FreeRTOS and additionally provide multi-core support. It then would be interesting to compare the single core BOOM with a multi-core Rocket to see how they behave. Especially with workloads, that make extensive use of the FPU, a significant difference is to be expected, because a multi-core Rocket also contains multiple FPUs. Also, designs that use coprocessors will be examined in the future.

Another interesting field to investigate are power devil program generators. Techniques mentioned in Chapter 4 could be leveraged to generate SLT programs, especially when looking at the work of Najeeb et al. [NKH+07] and Ganesan et al. [GJB+10]. It would be interesting to run such generated programs on the BOOM core used in this thesis to see, how it will behave, especially in comparison with the algorithms used here. More investigation into current state program generation techniques is necessary and research on if and how we can leverage these to generate SLT programs.

Bibliography

[AAB+16] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, A. Waterman. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[ABC+19] F. Almeida, P. Bernardi, D. Calabrese, M. Restifo, M. S. Reorda, D. Appello, G. Pollaccia, V. Tancorre, R. Ugioli, G. Zoppi. “Effective Screening of Automotive SoCs by Combining Burn-In and System Level Test”. In: *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, Apr. 2019. DOI: 10.1109/ddecs.2019.8724644.

[ABG+20] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanovic, B. Nikolic. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (July 2020), pp. 10–21. DOI: 10.1109/mm.2020.2996616.

[ASH17] H. Amrouch, V. M. van Santen, J. Henkel. “Interdependencies of Degradation Effects and Their Impact on Computing”. In: *IEEE Design & Test* 34.3 (June 2017), pp. 59–67. DOI: 10.1109/mdat.2016.2594180.

[BA04] M. L. Bushnell, V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer US, Dec. 15, 2004. 712 pp. ISBN: 978-0-7923-7991-1. DOI: 10.1007/b117406.

[BC12] S. Biswas, B. Cory. “An Industrial Study of System-Level Test”. In: *IEEE Design & Test of Computers* 29.1 (Feb. 2012), pp. 19–27. DOI: 10.1109/mdt.2011.2178387.

[BRR+20] P. Bernardi, M. Restifo, M. S. Reorda, D. Appello, C. Bertani, D. Petrali. “Applicative System Level Test introduction to Increase Confidence on Screening Quality”. In: *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, Apr. 2020. DOI: 10.1109/ddecs50862.2020.9095569.

[BVR+12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, K. Asanović. “Chisel”. In: *Proceedings of the 49th Annual Design Automation Conference on - DAC ’12*. ACM Press, 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[Che16] H. H. Chen. “Data analytics to aid detection of marginal defects in system-level test”. In: *2016 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, Apr. 2016. DOI: 10.1109/vlsi-dat.2016.7482550.

[Che18] H. H. Chen. “Beyond structural test, the rising need for system-level test”. In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, Apr. 2018. DOI: 10.1109/vlsi-dat.2018.8373238.

[GJ11] K. Ganesan, L. K. John. “MAximum Multicore POwer (MAMPO)”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’11*. ACM Press, 2011. DOI: 10.1145/2063384.2063455.

[GJB+10] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, L. K. John. “System-level max power (SYMPO)”. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT ’10*. ACM Press, 2010. DOI: 10.1145/1854273.1854282.

[GR81] P. Goel, B. Rosales. “PODEM-X: An Automatic Test Generation System for VLSI Logic Structures”. In: *18th Design Automation Conference*. IEEE, 1981. DOI: 10.1109/dac.1981.1585361.

[HM17] Y. Huang, P. Mishra. “Test Generation for Detection of Malicious Parametric Variations”. In: *Hardware IP Security and Trust*. Springer International Publishing, 2017, pp. 325–340. DOI: 10.1007/978-3-319-49025-0_14.

[JLL+17] A. Jutman, C. Lotz, E. Larsson, M. S. Reorda, M. Jenihhin, J. Raik, H. Kerkhoff, R. Krenz-Baath, P. Engelke. “BASTION: Board and SoC test instrumentation for ageing and no failure found”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, Mar. 2017. DOI: 10.23919/date.2017.7926968.

[Lar92] T. Larrabee. “Test pattern generation using Boolean satisfiability”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.1 (1992), pp. 4–15. DOI: 10.1109/43.108614.

[LHL+14] D. Lin, T. Hong, Y. Li, E. S, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, S. Mitra. “Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.10 (Oct. 2014), pp. 1573–1590. DOI: 10.1109/tcad.2014.2334301.

[LLCG19] M. Liu, X. Li, K. Chakrabarty, X. Gu. “Knowledge Transfer in Board-Level Functional Fault Identification using Domain Adaptation”. In: *2019 IEEE International Test Conference (ITC)*. IEEE, Nov. 2019. DOI: 10.1109/itc44170.2019.9000172.

[LPY+18] M. Liu, R. Pan, F. Ye, X. Li, K. Chakrabarty, X. Gu. “Fine-Grained Adaptive Testing Based on Quality Prediction”. In: *2018 IEEE International Test Conference (ITC)*. IEEE, Oct. 2018. DOI: 10.1109/test.2018.8624891.

[LTGW18] S. Letchumanan, T. H. H. Tan, Y. P. Gan, S. L. Wong. “Adaptive test method on production system-level testing (SLT) to optimize test cost, resources and defect parts per million (DPPM)”. In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, Apr. 2018. DOI: 10.1109/vlsi-dat.2018.8373239.

[Man80] B. B. Mandelbrot. “Fractal aspects of the iteration of $z \rightarrow \lambda z(1\text{-}z)$ for complex λ and z”. In: *Fractals and Chaos*. Vol. 357. 1. Blackwell Publishing Ltd Oxford, UK, 1980, pp. 249–259. DOI: 10.1007/978-1-4757-4017-2_3.

[Mic03] A. Miczo. *Digital Logic Testing and Simulation*. John Wiley & Sons, Inc., Aug. 2003. DOI: 10.1002/0471457787.

[Moo06] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. DOI: 10.1109/n-ssc.2006.4785860.

[NIS02] NIST. "Secure Hash Standard (SHS)". In: *FIPS PUB 180-2.* (2002).

[NKH+07] K. Najeeb, V. V. R. Konda, S. K. S. Hari, V. Kamakoti, V. M. Vedula. "Power Virus Generation Using Behavioral Models of Circuits". In: *25th IEEE VLSI Test Symmposium (VTS'07)*. IEEE, May 2007. DOI: 10.1109/vts.2007.49.

[PAB+20] I. Polian, J. Anders, S. Becker, P. Bernardi, K. Chakrabarty, N. ElHamawy, M. Sauer, A. Singh, M. S. Reorda, S. Wagner. "Exploring the Mysteries of System-Level Test". In: *2020 IEEE 29th Asian Test Symposium (ATS)*. IEEE, Nov. 2020. DOI: 10.1109/ats49688.2020.9301557. (Early Access).

[RAH+14] P. G. Ryan, I. Aziz, W. B. Howell, T. K. Janczak, D. J. Lu. "Process defect trends and strategic test gaps". In: *2014 International Test Conference*. IEEE, Oct. 2014. DOI: 10.1109/test.2014.7035276.

[RCS+16] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, B. Becker. "A Flexible Framework for the Automatic Generation of SBST Programs". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.10 (Oct. 2016), pp. 3055–3066. DOI: 10.1109/tvlsi.2016.2538800.

[Rot66] J. P. Roth. "Diagnosis of Automata Failures: A Calculus and a Method". In: *IBM Journal of Research and Development* 10.4 (July 1966), pp. 278–291. DOI: 10.1147/rd.104.0278.

[SBP16] M. Sauer, B. Becker, I. Polian. "PHAETON: A SAT-Based Framework for Timing-Aware Path Sensitization". In: *IEEE Transactions on Computers* 65.6 (June 2016), pp. 1869–1881. DOI: 10.1109/tc.2015.2458869.

[Sin19] A. D. Singh. "An Adaptive Approach to Minimize System Level Tests Targeting Low Voltage DVFS Failures". In: *2019 IEEE International Test Conference (ITC)*. IEEE, Nov. 2019. DOI: 10.1109/itc44170.2019.9000173.

[TBD+15] A. Touati, A. Bosio, L. Dilillo, P. Girard, A. Virazel, P. Bernardi, M. Reorda. "Exploring the Impact of Functional Test Programs Re-Used for Power-Aware Testing". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE Conference Publications, 2015. DOI: 10.7873/date.2015.1031.

[TK14] D. K. R. Tipparthi, K. K. Kumar. "Concurrent system level test (CSLT) methodology for complex system-on-chip". In: *2014 IEEE 16th Electronics Packaging Technology Conference (EPTC)*. IEEE, Dec. 2014. DOI: 10.1109/eptc.2014.7028421.

[UKW17] D. Ull, M. Kochte, H.-J. Wunderlich. "Structure-Oriented Test of Reconfigurable Scan Networks". In: *2017 IEEE 26th Asian Test Symposium (ATS)*. IEEE, Nov. 2017. DOI: 10.1109/ats.2017.34.

[VS06] A. Vassighi, M. Sachdev. "Thermal Runaway in Integrated Circuits". In: *IEEE Transactions on Device and Materials Reliability* 6.2 (June 2006), pp. 300–305. DOI: 10.1109/tdmr.2006.876577.

[WA19] A. Waterman, K. Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 20190608-Priv-MSU-Ratified. RISC-V Foundation. Dec. 2019. URL: https://riscv.org/technical/specifications/https://riscv.org/technical/specifications/ (visited on Nov. 9, 2020).

[WIS+18] E. Wang, A. M. Izraelevitz, C. Schmidt, B. Nikolić, E. Alon, J.Bachrach. "Hammer: Enabling Reusable Physical Design". In: 2018. URL: https://people.eecs.berkeley.edu/~edwardw/pubs/hammer-woset-2018.pdf (visited on Nov. 10, 2020).

[WLPA11] A. Waterman, Y. Lee, D. A. Patterson, K. Asanović. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. Version 1.0. CS Division, EECS Department, University of California, Berkeley, May 13, 2011. URL: https://riscv.org/technical/specifications/https://riscv.org/technical/specifications/ (visited on Nov. 9, 2020).

[WWW06] L.-T. Wang, C.-W. Wu, X. Wen. *VLSI Test Principles and Architectures*. MORGAN KAUFMANN PUBL INC, June 1, 2006. 808 pp. ISBN: 0123705975. DOI: 10.1016/b978-0-12-370597-6.x5000-8.

[ZB19] F. Zaruba, L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. DOI: 10.1109/tvlsi.2019.2926114.

[ZKGA20] J. Zhao, B. Korpan, A. Gonzalez, K. Asanovic. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine". In: *Fourth Workshop on Computer Architecture Research with RISC-V*. May 2020. URL: https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf (visited on Nov. 10, 2020).

A Listings

```
class SynthLargeBoomCachelessConfig
    extends Config(
      new chipyard.iobinders.WithTieOffInterrupts ++ // tie off top-level interrupts
        new chipyard.iobinders.WithBlackBoxSimMem ++ // drive the master AXI4 memory with a
blackbox DRAMSim model
        new chipyard.iobinders.WithTiedOffDebug ++ // tie off debug (since we are using
SimSerial for testing)
        new iobinders.WithDontTouchPorts ++
        new WithSimAXIMMIO ++ // Add the AXI Master MMIO port
        new WithBootROMFile( s"software/bootrom/bootrom.bin" ) ++
        new chipyard.config.WithL2TLBs(1024) ++ // use L2 TLBs
        new WithTraceIO ++ // Include Tracing IO to see what each hart is doing
        new WithoutTLMonitors ++ // Remove TileLink monitors, they should not be synthesized
        new WithPBUSFreq(BigInt(50000000)) ++ // Set the frequency of the periphery bus (
normally 100MHz)
        new freechips.rocketchip.subsystem.WithFastMulDiv ++ // Enfore usage of a fast (
unrolled) multiply divide unit
        new freechips.rocketchip.subsystem.WithExtMemSize((1 << 30)) ++ // 1GB external Memory
        new freechips.rocketchip.subsystem.WithNoSlavePort ++ // no top-level MMIO slave port
(overrides default set in rocketchip)
        new freechips.rocketchip.subsystem.WithNExtTopInterrupts(0) ++ // no external
interrupts
        new boom.common.WithLargeBooms ++ // Large boom config
        new boom.common.WithNBoomCores(1) ++
        new freechips.rocketchip.subsystem.WithCoherentBusTopology ++ // hierarchical buses
including mbus+l2
        new freechips.rocketchip.system.BaseConfig
    )
```

Listing A.1: LargeBoom

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature