Universiteit Leiden
The Netherlands

Computer Science

On the vulnerability of open-source eFPGAs to malicious attacks based on power-hammering designs

Joris Gravesteijn

2573172

Supervisors:
Todor Stefanov & Nele Mentens

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

12/07/2023

Abstract

The reconfigurability and versatility of FPGAs and eFPGAs result in an increase in use and popularity. They are used in high-value sectors like defense and medical applications that require high reliability and security. They are also becoming more popular among hobbyists and commercial entities. This is partly due to the ease of sharing Intellectual Property and designs which lowers the learning curve and accelerates adoption of FPGAs and eFPGAs. However, this also makes it easy for malicious parties to infect and abuse Intellectual Property which poses a risk to the end users. This thesis investigates these risks for eFPGAs in general with a focus on the SymbiFlow open-source design flow and power hammering circuits. The results show that SymbiFlow offers little to no protection. Furthermore, eFPGAs can suffer greatly under the influence of very small and easily embedded power hammering designs, requiring as little as 7.3% of the available logic space to render the eFPGA unpredictable.

Contents

1 Introduction

This thesis focuses on Denial-of-Service and power draining attacks on embedded Field-Programmable Gate Arrays. It analyses the protection capabilities of the used open-source design flow (SymbiFlow) and investigates the effect of multiple types of ring oscillators on the power consumption and core validity of the eFPGA core.

Before the invention of Field Programmable Gate Arrays (FPGAs) in 1984 [Tri15], developers had to order a new chip for every design change in a circuit, resulting in high development costs. They also had to wait for new components since every design change required a physically different Integrated Circuit (ICs). The most common chips that were used before the utilization of FPGAs are Application Specific Integrated Circuits (ASICs). These semiconductor devices are created with a specific circuit implementation and cannot be changed after production [CK00]. Meaning that for every design change, new ICs needed to be ordered [Utm21]. FPGAs provide an alternative which is re-configurable on the go instead of waiting for a new IC. This brings the benefit of reduced development costs and faster development cycles but with a higher cost per unit.

During the initial invention and development of FPGAs, ASICs offered magnitudes of better performance. This performance gap has shrunk significantly since then but ASICs still offer higher performance with regards to FPGAs [Tri15, KR06]. This smaller difference in performance resulted in FPGAs becoming a viable alternative while the maximum attainable performance still lies with ASICs [Int23]. FPGAs are also popular due to the possibility to reconfigure them while they are in the final product. This allows manufacturers to solve issues or upgrade capabilities via a software update instead of having to physically change components.

1.1 Problem Statement

The rise in popularity of FPGAs and eFPGAs will likely result in an increase of using open-source tooling like SymbiFlow [OS23], especially for hobbyists. A large benefit of eFPGAs is that the designs and configurations can be shared as Intellectual Property (IP) which only require adaptation to the specific FPGA architecture used. This allows hobbyists but also companies to share and use IPs, lowering the learning curve for using FPGAs. However, this also means that there is a possibility of malicious IPs which can act as a virus resulting in unexpected or unwanted behaviour. One type of these malicious IPs are designs containing power hammering circuits that consume a lot of power and can potentially crash or damage the (e)FPGA [GOT17]. It is important that the implications of power hammering circuits are known and that the current SymbiFlow design flow is analysed for security leaks. The reconfigurability of the (e)FPGA could be a security risk, not only exposing the FPGA but the entire system [ZQ14]. These facts rise the following research questions:

RQ1: What are the effects of different kinds of power hammering circuits on the correct functioning and power consumption of the eFPGA core on the EOS-S3 SoC [EOS18]?

RQ2: What are the current protection capabilities of the open-source SymbiFlow [OS23] design flow for preventing and warning the user of possibly unwanted code in their designs?

1.2 Thesis contributions and overview

This thesis provides the following:

- A framework to automatically test the effect of different power hammering circuits on eFPGAs.
- An experimental setup to test the EOS-S3 SoC. This setup can generate multiple types of ring oscillators with a variable number of circuits. It can also program the board and perform power measurements.
- An analysis of the protection capabilities of the SymbiFlow design flow against possibly unwanted and malicious designs containing ring oscillators.

This chapter contains the introductions; Section 2 provides the background information needed for this thesis; Section 3 describes work related to this research; Section 4 presents the materials used and the experimental setup; Section 5 presents the results from the experiments; Section 6 discusses the results; Section 7 concludes this thesis.

Bachelor thesis for Computer Science by Joris Gravesteijn at LIACS, University of Leiden under the supervision of Todor Stefanov.

2 Background

This section introduces the various concepts needed to understand this thesis.

2.1 Field Programmable Gate Arrays (FPGAs)

A Field Programmable Gate Array (FPGA) is an integrated circuit that contains a grid of Configurable Logic Blocks (CLBs) with programmable interconnects between the CLBs. An FPGA also contains I/O blocks that facilitate communication with components outside of the FPGA such as other FPGAs, general-purpose processors and sensors. This interoperability allows for FPGAs to be used in complex systems in which the FPGA optimizes highly specific tasks while the general-purpose host takes care of the tasks which the FPGA is not optimized for. Nowadays most FPGAs also contain Block Random Access Memory and Digital Signal Processing slices. The Block Random Access Memory (BRAM) blocks can be used as Random Access Memory in the same manner as with a CPU. Furthermore, Digital Signal Processing Slices (DSPs) can be used to carry out digital signal processing functions such as filtering or multiplying [KR06]. A block diagram of a general FPGA is shown in Figure 1. A grid of CLBs (blue) is shown with the black lines representing the programmable interconnects. The switch matrices are used to connect the different interconnects, allowing for connections to be created. The I/O blocks are shown in yellow and allow for interaction with peripherals outside of the FPGA. BRAM and DSPs are not shown. Also, a simplified version of a CLB is shown in Figure 2. This CLB consists of two LUT3 primitives, a Full Adder, three multiplexers and a D-Flip Flop. This CLB has 5 inputs, a clock line and two outputs.

FPGAs are most commonly programmed by using a Hardware Description Language (HDL) [ES08].

This type of programming language is used to describe the digital circuits that need to be implemented on the FPGA. This includes structure, behaviour and timing of the circuits. HDL is a low-level language which means that each component and its connection is described. A tool called the synthesizer takes the HDL code and creates a binary file that configures the individual CLBs and interconnections based on the architecture of the FPGA. First, the HDL code is converted to primitives. These primitives represent configurations of a singular CLB. For example, a Look-Up Table with 2 inputs (LUT2) primitive represents the required ports to be able to configure a CLB as a LUT table with 2 inputs. After the required primitives are known, the synthesizer places them within the architecture of the FPGA. The final step is routing the necessary connections in between blocks [CoQ20].

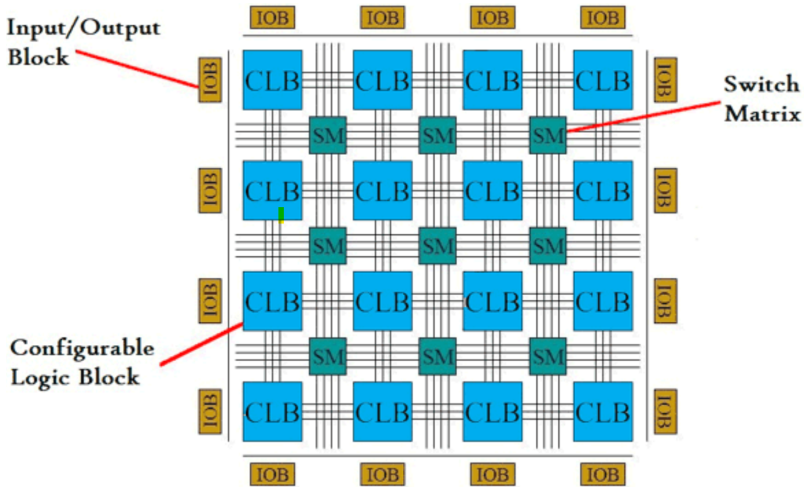


Figure 1: Introduction to FPGA and its Programming Tools, Abhimanyu Pandit, 24-03-2019, From https://circuitdigest.com/tutorial/what-is-fpga-introduction-and-programming-tools

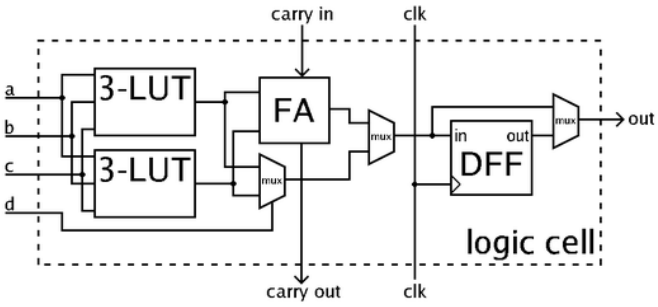


Figure 2: Simplified CLB, Peter Kallstrom, 02-05-2010, From https://en.wikipedia.org/wiki/Logic_block

Generally, HDL designs, also called Intellectual Property (IP), can be exchanged in between FPGAs and only require adaptation to the FPGA architecture. The adaptation can usually be done by the synthesizer without any modification to the actual HDL code. This allows hobbyists but also companies to share and use IPs thereby lowering the learning curve for using FPGAs. However,

this also means that there is a possibility of malicious IPs which can act as a virus resulting in unexpected or unwanted behaviour.

Aside from being re-programmable, FPGAs also offer higher performance with regards to latency and parallelism when compared to Central Processing Units (CPUs). They offer lower latency and can thereby achieve a higher throughput [LGM+12].

FPGAs do have multiple downsides when compared with ASICs. They are generally more expensive than ASICs due to their increased complexity resulting in lower development costs but higher production costs. FPGAs also consume more power [AAE06] and are physically larger due to onboard storage (BRAM) and resources for reprogramming and configuration. ASICs can also be more optimized when compared to FPGAs because they are designed for one highly-specific task [KR06].

2.2 Embedded FPGA (eFPGA)

Embedded FPGAs (eFPGAs) are a programmable logic core (FPGA) directly integrated within the ASIC or System on Chip (SoC) [Fle05], as shown in Figure 3. Figure 3 shows that less space is needed on the PCB and also various improvements with regards to FPGA power consumption, unit cost, latency with regards to the SoC and bandwidth. Generally, almost half of the power consumption of an FPGA is dedicated to re-programmable I/O circuits around a standalone FPGA. However, since the eFPGA has a direct connection to the ASIC or SoC, these circuits are no longer required and a significant chunk of the power consumption is eliminated [Utm22, Fle05]. The ASIC or SoC can also take over the role of a lot of the surrounding peripherals for the FPGA such as clock generation which saves space and routing difficulty for the PCB. Since FPGAs offer significant performance boosts in specific tasks when compared to ARM or RISC-V processors, a combination of the two can be very efficient. Integrating the FPGA core into the SoC means a close proximity between the eFPGA core and the processor, allowing for a connection with high-bandwidth and low-latency [vSNBN06]. Furthermore, the designer of the SoC can determine the size and I/O of the eFPGA core which gives a lot of versatility and allows for optimal configurations where no space is wasted. It also means that current products that have physical FPGA chips and processors can be minimized by integrating it into one SoC, resulting in smaller PCBs and thus devices.

eFPGAs are very popular in the defense, AI, and acceleration sector. They are widely used in automation, automatic driving systems, and medical electronics [Leo08]. All of which are critical sectors where consistency and reliability are key.

2.3 Attacks on FPGAs and eFPGAs

With the rise in popularity of FPGAs and eFPGAs, the risk of attackers and people with malicious intentions also rises. As mentioned in Section 2.1, it is possible to share IPs that contain the configuration of an FPGA. This lowers the learning curve for FPGA design and allows for more easy adaptation and development. However, this also creates an opportunity for IPs with malicious circuits to infect FPGAs. A common example of malicious circuits are power hammering

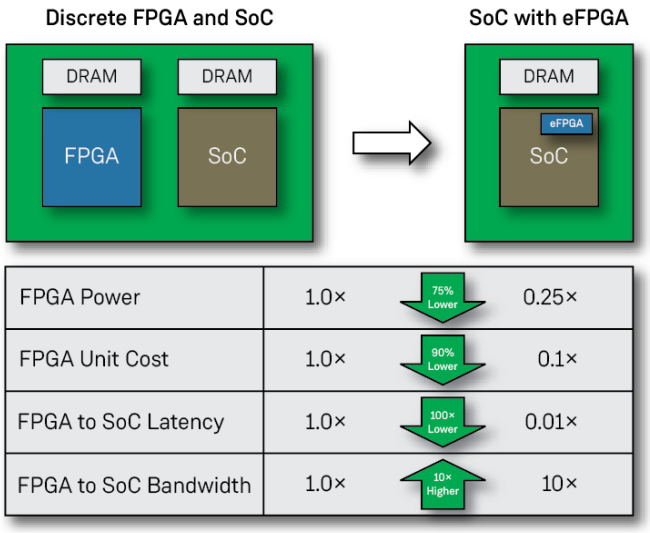


Figure 3: FPGA vs eFPGA configuration. From https://www.edn.com/has-the-time-for-embedded-fpga-efpga-ip-finally-come/

circuits that draw high amounts of power via ring oscillators. This is further explained in Section 2.4.

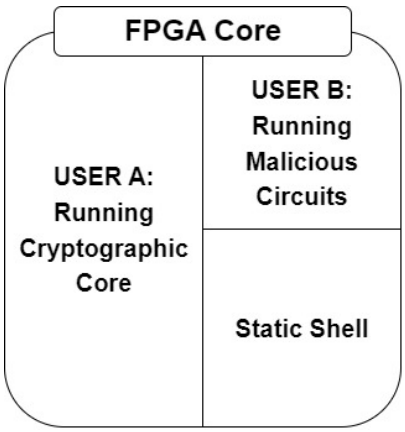


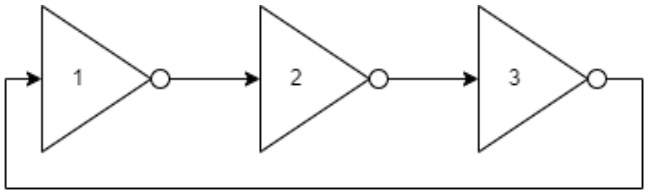
Figure 4: FPGA as a Service (FaaS) with 1 FPGA core being shared amongst 2 users

Another example is in the cloud-based domain. In the current day and age, a lot of resources are moved to the cloud. Big vendors like Amazon Web Services (AWS) and Microsoft's Azure already offer FPGAs in the cloud as FPGA as a Services (FaaS). These cloud-based FPGAs divide a single FPGA core into multiple chunks such that the single large FPGA core can be fully utilized by multiple individuals, as shown in Figure 4. These users share the power budget which creates the possibility of a Denial of Service (DoS) attack. During such an attack, one of the users implements power hammering circuits such as oscillators with very high switching frequencies. These circuits consume the power budget for the FPGA and this can result in the FPGA crashing. This attack results in the FPGA becoming unusable for the other users [LMG+20].

The sharing of IPs with malicious code is especially a risk in eFPGAs. Most of the time, they share the power rail of the SoC with the other peripherals. This means that simple and small ring oscillators can be used to consume the power budget, limiting the functionality of the entire SoC. This can result in a significant enough voltage drop to thereby crash the eFPGA, as with normal FPGAs [LMG+20, GOT17]. These ring oscillators can be very hard to find and hidden in the binary file that is uploaded to the eFPGA. A possible attack is that one injects ring oscillators to a functional bit-stream, resulting in a hybrid binary with malicious and non-malicious code. This can have numerous implications on the reliability of the eFPGA but also on the battery life of mobile systems that use eFPGAs.

2.4 Ring Oscillators

A very common way of power hammering is by embedding multiple instances of ring oscillators on an (e)FPGA. A ring oscillator is a circuit with a very high switching frequency [MS10]. The power consumption of the FPGA is directly linked to the switching frequency of the logic circuit, with a higher switching frequency resulting in higher power consumption. Most ring oscillators are not clocked and they can reach switching frequencies in the GHz range based on the propagation delay of the used primitives [LMG+20]. The most basic form of a ring oscillator is an uneven number of NOT gates, with the output of the last NOT gate being fed into the first NOT gate, as shown in Figure 5a and Table 5b. The amount of power drawn decreases with the number of gates in one loop, since this adds to the total propagation delay. A longer delay means a lower switching frequency and thus less power consumption.



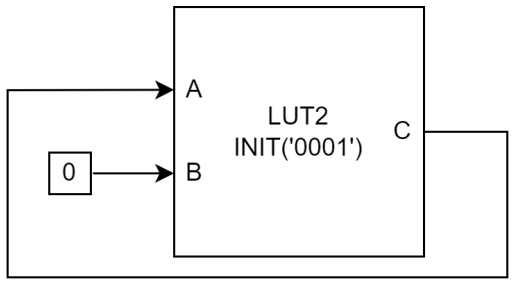
(a) Generic Ring Oscillator based on three NOT gates

Gate #	1	2	3	1	2	3	1
Value	0	1	0	1	0	1	0

(b) NOT Gate values, the first and last cell are the same, forming a loop.

Figure 5: The most basic form of a ring oscillator based on three NOT gates which are linked together to form a loop.

Another common example of a ring oscillator is based on Look-Up Tables (LUT). A lookup table is a table that generates an output based on the inputs and a predefined INIT value. A LUT2 table with two inputs and one output is shown in Table 6b. LUTs can be used to implement any boolean function as long as there are enough inputs available. They can also mimic other gates such as AND gates and OR gates. To create a ring oscillator using a LUT2, one of the outputs is connected back into one of the inputs of the LUT. This can be directly fed into itself, as shown in Figure 6a and Table 6b, or in a sequence of multiple LUTs.



(a) Generic Ring Oscillator based on a LUT2, the truth table is shown in table 6b

Input A	Input B	Output C
0	0	1
1	0	0
0	1	0
1	1	0

(b) LUT2 table with 2 inputs (A, B) and 1 output (C)

Figure 6: LUT2 based ring oscillator with two inputs and one output.

3 Related Work

In related work [LMG+20], fifteen types of ring oscillators are tried. It shows that FPGAs are vulnerable to power hammering designs and that only 3% of the FPGA is required to disrupt the functionality of the FPGA. The related work only implements power hammering designs on an FPGA board. This thesis aims to replicate their results using an eFPGA to see if eFPGAs are vulnerable to the same type of designs. This thesis replicates the combinatorial designs that are based on the LUT and MUX primitives and it expands the related work by comparing variations of these designs that use different types of primitives such as the MUX2 primitive. This will show the vulnerability of eFPGAs to power hammering designs and will examine the effect of power hammering designs based on different primitives.

Related works [GOT17, PHT19, KGT18, MS19] show that FPGAs are vulnerable to voltage drop based attacks and that these attacks can disrupt multi-tenant systems. The voltage is dropped to such a degree that the FPGA experiences timing faults [MS19], crashes [GOT17, PHT19, KGT18] or remains unresponsive [GOT17]. Related work [GOT17] shows that in some cases, the functionality of the FPGA is only restored after a full power-cycle. Related work [GOT17] only examines LUT based power hammering designs. This thesis replicates the voltage drop attack vector and examines the effect on the EOS-S3 SoC. Furthermore, this thesis aims to expand this attack vector by replicating the attack with MUX and NOT primitives instead of LUT primitives. This will give insight into the effect of different primitives on the voltage drop based attack and could result in a design that requires a smaller percentage of the total logic space to crash the eFPGA when compared to the LUT primitive based power hammering design.

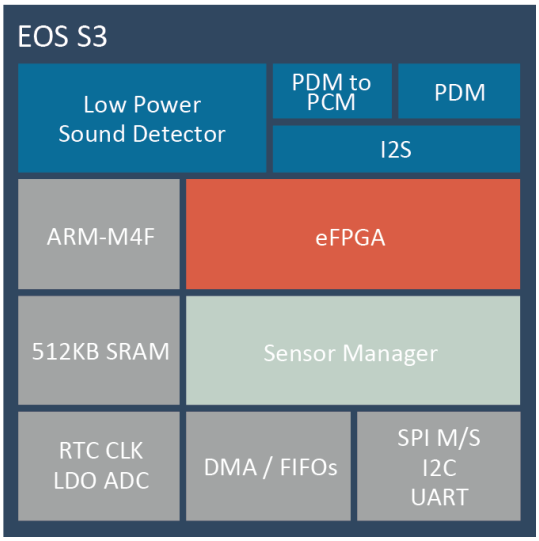
Furthermore, related works [PHT19, KGT18] implement attacks on multi-tenant cloud-based FPGA systems. This thesis will examine the effect of malicious circuits that are embedded in functional designs. This mimics the attack on multi-tenant systems as proposed in related works [PHT19, KGT18]. This will give insight to the vulnerability of the eFPGA when used in multi-tenant configurations or when the eFPGA is split up to execute multiple tasks in parallel.

4 Experimental Methods and Materials

This section explains the used EOS-S3 SoC with an eFPGA core, the SparkFun QuickLogic Thing+ development board and the experimental setup, which we have developed, which automatically creates and tests power hammering circuits. It also explains the used power hammering circuits.

4.1 EOS-S3

The SoC used for the experiments is the EOS-S3 from QuickLogic [EOS18] implemented on the SparkFun QuickLogic Thing+ (QT+) board [Spa21]. The block diagram of the EOS-S3 is shown in Figure 7a. This board was chosen due to its relatively cheap price, its open-source tools for programming the board and it was readily available. The eFPGA contains 891 CLBs, the general layout of a CLB is shown in Appendix A and more statistics can be seen in Table 7b. Each CLB is a multiplexer-based single bit register with 22 simultaneous inputs and four outputs, of which three are combinatorial and is in a register. This allows for a high variety of configurations per CLB which include but are not limited to two independent 3-input functions, 4-input functions and 8-to-1 mux functions [Qui20]. The EOS-S3 is programmable over a UART connection and for this thesis only the eFPGA core is used, not the ARM-M4F processor.



(a) EOS-S3 block diagram, From https://www.quicklogic.com/products/soc/eos-s3-microcontroller/

Feature	EOS-S3
Logic Cells	891
8K RAM Modules (512x18 - 9.216 bits)	8
FIFO Controllers	8
RAM Bits	73.728
Configurable Interface	32
Multiplier	2x32x32 4x16x16

(b) EOS-S3 On-Chip Programmable Logic Major features, From https://www.quicklogic.com/wp-content/uploads/2020/06/QL-EOS-S3-Ultra-Low-Power-multicore-MCU-Datasheet.pdf, table 7

Figure 7: The block diagram of the EOS-S3 core and statistics about the integrated eFPGA core.

One CLB of the EOS-S3 can be split up into four fragments to fit multiple functions into one CLB. Appendix A shows this with the first letter of the input and output pin corresponding to the fragment that it belongs to. These fragments do have different capabilities but it is possible to utilize multiple fragments of one CLB for different functions. For example, the f-fragment can be

used to implement a NOT gate while a LUT2 table is implemented in the t-fragment. This allows for more efficient use of the available CLBs. These fragments can also be grouped to allow for a larger component inside one CLB like a LUT4 table which requires 16 inputs. The synthesizer does have the functionality to merge or split different primitives into combinations of other primitives while retaining the desired behaviour. Examples of this are using two LUT3 primitives and a multiplexer to implement a LUT4 primitive or using a LUT1 primitive to mimic a NOT gate. The latter is used when the f-frag of a CLB is used up while the t-frag is still available. This allows for maximum utilization of the eFPGA.

4.1.1 Power Rail for eFPGA core

An important aspect of the EOS-S3 implementation is the power supply for the eFPGA. On the SparkFun QuickLogic Thing+, the eFPGA is powered by Low-DropOut (LDO) regulators within the EOS-S3 SoC. These convert the external input of 3.3V down to approximately 1.1V. A block diagram is shown in Figure 8. It is important to note that the operation of the eFPGA is only considered valid while the voltage supply is above 0.95v as explained in [Qui17], page 80. This was also further confirmed by the QuickLogic Corporation in a private communication and is used as the threshold for breaking the functionality of the eFPGA during the experiments.

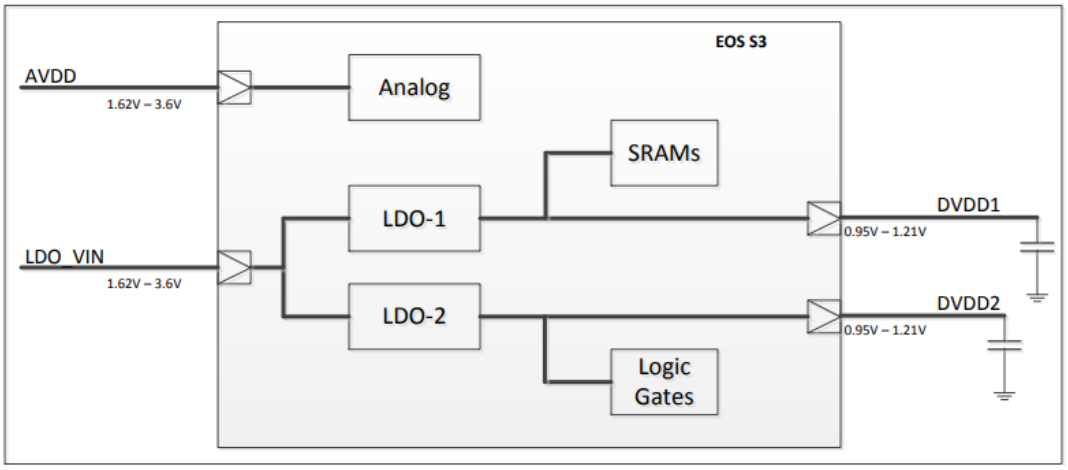


Figure 8: LDO Configuration on the SparkFun QuickLogic Thing+ [Qui17], page 80

4.2 SymbiFlow

SymbiFlow is the open-source design flow which is used to generate EOS-S3 binaries from custom HDL code. It uses Verilog To Routing (VTR) [MPZ+20] to map the custom HDL code to a EOS-S3 binary. First, VTR uses Yosys Open SYnthesis Suite (YOSYS) [Yos23] to map the HDL file to the EOS-S3 primitives. Then, Versatile Place and Route (VPR) is used to route the connections between the I/O blocks and the CLBs. Figure 9 shows a block diagram of SymbiFlow.

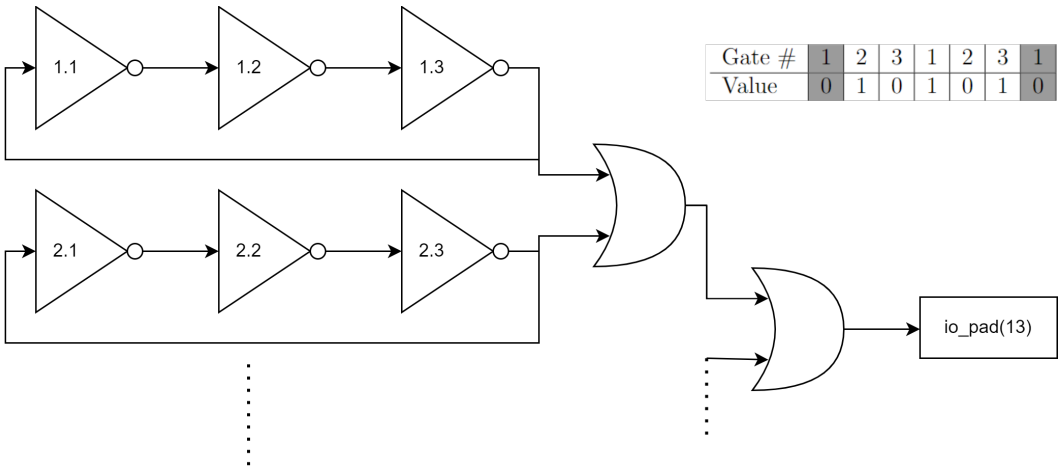
Figure 9: Block diagram of the SymbiFlow design flow submodules

4.3 Power Hammering Circuits

This section explains the different power hammering circuits that we have implemented and tested. All power hammering circuits that are used for testing are based on combinatorial ring oscillators that can generate high frequency switching activities with a minimal of one CLB. They are generated by bash scripts, that we developed, called "Verilog Generators", that are further explained in Section 4.3.5. All the generators and results are available in the projects GitHub repository https://github.com/JsGraaf/Thesis_QT_Testing_Software.

4.3.1 NOT3 Based RO

Figure 10 shows the implemented ring oscillator based on linking three NOT gates in a sequence. An OR tree is implemented to link all the ring oscillators due to the requirement that every circuit must have an output. Otherwise, the entire circuit will be removed from the design during the optimization stages. The OR tree compresses all the NOT3 ring oscillators to one output pin, lowering the I/O requirement for the eFPGA. As explained in Section 2.4, an uneven number of inverters results in a ring oscillator. The value is inverted after the last NOT gate and fed back into the first NOT gate. This results in an endless loop with every iteration changing the value of the NOT gates. This change in value results in a power draw. On the EOS-S3, the NOT gates are implemented in the f-frag section of the CLB [QC21] and one ring oscillator instance requires at least 3 CLBs. Since they can utilize the small f-frag section of a CLB, they are easily embedded into designs.



Gate #	1	2	3	1	2	3	1
Value	0	1	0	1	0	1	0

Figure 10: NOT3 based ring oscillator with 2 instances. The table shows the propagation of the values trough one ring oscillator

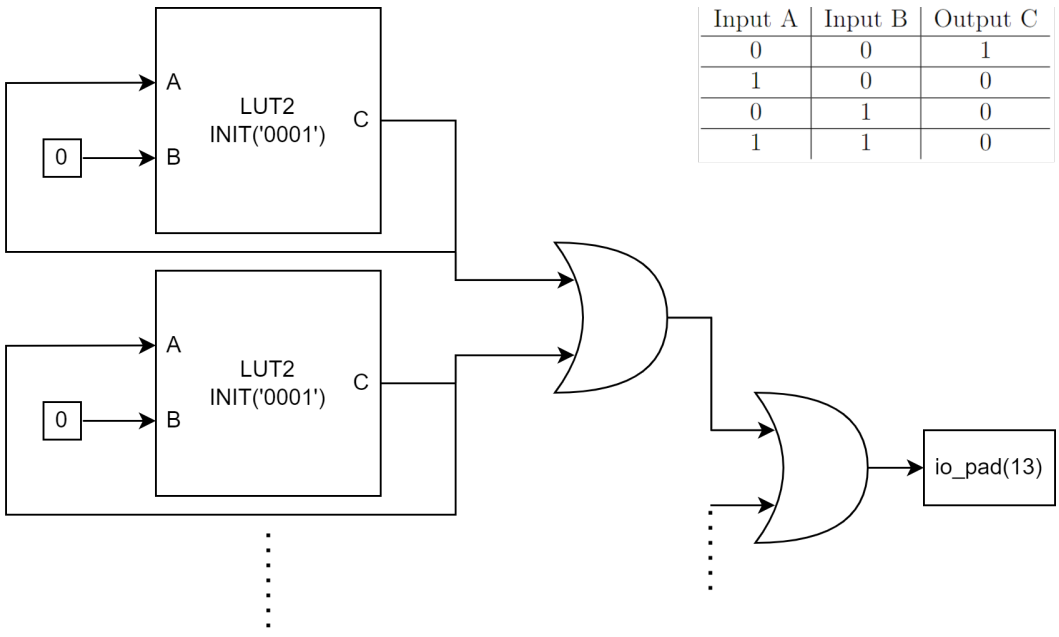
4.3.2 LUT Based ROs

Figure 11 shows the implemented ring oscillator based on a single LUT2 primitive. Like with the NOT3 ring oscillator, an OR tree is used to link the outputs of the instances. The truth table in Figure 11 shows the initialisation of the LUT2 primitive. Input B is always set to zero and the value of the input A is changed every time it propagates through the LUT2 primitive. This results in an inverter which begins to self-oscillate.

In this implementation, no enable signal is used to simulate the real-world implementation where the malicious circuit should start when the eFPGA receives power. This can easily be modified by replacing the constant 0 of output B with an enable signal and changing the INIT value to INIT('0100'). By doing this, the LUT2 will always output 0, unless input B is equal to 1, at which point it will start oscillating. The LUT2 table is implemented in the t-frag section of an EOS-S3 CLB [QC21]. Only one CLB is required to implemented this type of ring oscillator.

The implementation for the LUT1, LUT3 and LUT4 based ring oscillators is the same with the only change being the primitive that is used. The only modification needed is setting the remaining outputs of the primitives to zero and adding leading zeros to the INIT variable. In the case of the LUT1 primitive, input B is removed and the INIT variable is changed to INIT('00'). In the case of LUT3, input D is added and the INIT variable is changed to INIT('00000001'). Finally, in the case of LUT4, input D and E are added and the INIT variable is changed to INIT('0000000000000001').

All LUT primitives are implemented on the t-frag of a EOS-S3 CLB, thereby requiring only one CLB for a functioning RO [QC21].



Input A	Input B	Output C
0	0	1
1	0	0
0	1	0
1	1	0

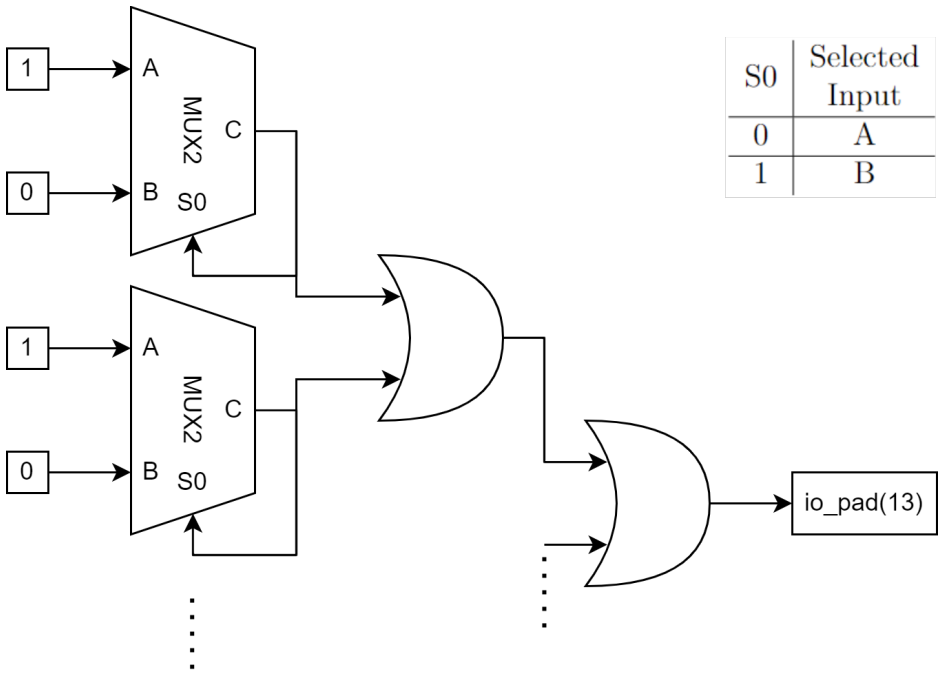
Figure 11: LUT2 based ring oscillator with 2 instances

4.3.3 MUX Based ROs

Figure 12 shows the implemented ring oscillator based on a single MUX2 primitive. Like with the previous ring oscillator implementations, an OR tree is used to link the outputs of the instances. The MUX2 primitive propagates the state of one of the inputs based on the S0 selection input. The table in Figure 12 shows which input is propagated with which value. Due to the values of inputs A and B, input A selects input B, which then selects input A. This results in constant switching of the MUX2 primitive, resulting in power draw. The MUX2 primitive is implemented on the f-frag section of a EOS-S3 CLB and requires only on CLB for a fully functional RO [QC21]. Due to it only utilizing the f-frag portion, it can easily be embedded in a design.

For the MUX4 implementation, two inputs D and E are added together with one selection line S1. All of these new inputs are connected to 0, resulting in the same behaviour. MUX8 adds an additional 6 inputs labelled with D, E, F, G, H and I together with the two selection lines S1 and S2.

The MUX4 primitive is implemented on the t-frag section of a EOS-S3 CLB [QC21]. The MUX8 primitive is implemented on a combination of the t-frag and b-frag section (tb-frag) section of an EOS-S3 CLB [QC21]. Both MUX4 and MUX8 require one CLB to implemented a fully functional ring oscillator.



S0	Selected Input
0	A
1	B

Figure 12: MUX2 based ring oscillator with 2 instances

4.3.4 Note on D-flip flop and Latch based Ring Oscillators

The SymbiFlow design flow currently does not support the D-flip flop and Latch primitives of the EOS-S3. They can be defined by using the *logic_cell_macro* primitive and creating the components from scratch but there was not enough time within this thesis to implement these primitives.

4.3.5 Circuit Generation

An important part to automate the experiments is our Verilog Generator which can create a valid Verilog file with the specified number of RO circuits. This enables the PC host to scale the amount of ring oscillators in the FPGA to see the effect of each increment. The generator is split into four parts: The initialisation of the generator, initialisation of the modules, the instance generation (one circuit) and the end of the module. The generator for the NOT3 experiment is available in appendix B for reference.

4.3.5.1 Verilog Generator: Generator Initialisation

```
#!/bin/bash

### <CIRCUIT NAME HERE> generator for verilog code for QT plus ###

# Usage: ./<Filename> <Amount of circuits> <output file path>
FILE_NAME='MODULE_top.txt'
CIRCUIT_NAME="<CIRCUIT NAME HERE>"
# Check if arguments where given
if [ $# -eq 0 ]
  then
    echo "No arguments supplied"
    exit
fi

# Get amount from arguments
AMOUNT=$1
OUTPUT_FILE_PATH=$2

echo "Generating ${1} $CIRCUIT_NAME"
```

This code block is responsible for parsing the input from the Command Line Interface (CLI) when calling the generator. It checks if all the required arguments are present and informs the caller on the amount and type of circuits that are going to be generated.

4.3.5.2 Verilog Generator: Module Initialisation

```
### INITIALISATION OF SUBMODULES ###
# This can be deleted if there are no submodules
echo """
<ADD SUBMODULE DEFINITIONS HERE>
""" > $FILE_NAME
```

This section is responsible for the pre-module initialisation. The <ADD SUBMODULE DEFINITIONS> string should be replaced with all the code which needs to be defined before the top module generation. This can be global variables, includes from other libraries or submodules.

```
### TOP OF MODULE ###
echo """
module MODULE_top(
   io_pad
);
```

```
// GPIO
inout    wire     [31:0]      io_pad      ;
""" >> $FILE_NAME

echo """
<ADD TOP MODULE DEFINITION HERE>
<BEWARE OF \$AMOUNT == 0>
""" >> $FILE_NAME
```

This section is responsible for the initialisation of the top module. This starts with the required header that exposes the I/O pins of the QuickLogic Thing+. This is required for proper compilation. After this initialisation, the custom code for the module can be defined. This can include testing and debugging modules or other components that need to be available outside a single instance. It should be noted that the generator should account for zero instances being generated. For example: If a register is being created based on the number of instances generated, there should be a verification such that "reg [0:0] registers" is avoided since this will result in a compilation error.

4.3.5.3 Verilog Generator: Instance generation

```
### INSTANCE GENERATION (individual Circuit) ###
if [ "$AMOUNT" -gt "0" ]; then
   for i in $( eval echo {1..$AMOUNT} )
   do
      echo """<ADD INSTANCE DEFINITION HERE>
""" >> $FILE_NAME
   done
else # In case the design is empty (circuits = 0), add 1 inverter
echo """inv invBase (
   .A(io_pad[2]),
   .Q(io_pad[25])
);
""" >> $FILE_NAME
fi
```

This section is responsible for the generation of the circuit instances. The instance design should be placed in the < ADD INSTANCE DEFINITION HERE > field. One instances should contain one fully functional circuit but the instances can be connected to each other. In case of an empty design, one inverter is created inside of the module. This is due to the restrictions of the SymbiFlow design flow which requires the presence of at least one component. The inverter connects one input to one output such that a valid circuit is created. This has a negligible effect on the power measurements on the circuit since the CLB changes state once, directly after powering on.

4.3.5.4 Verilog Generator: End of Module

```
### END OF MODULE ###
echo '<ADD MODULE END HERE>' >> $FILE_NAME
echo 'endmodule' >> $FILE_NAME

mv $FILE_NAME $OUTPUT_FILE_PATH
```

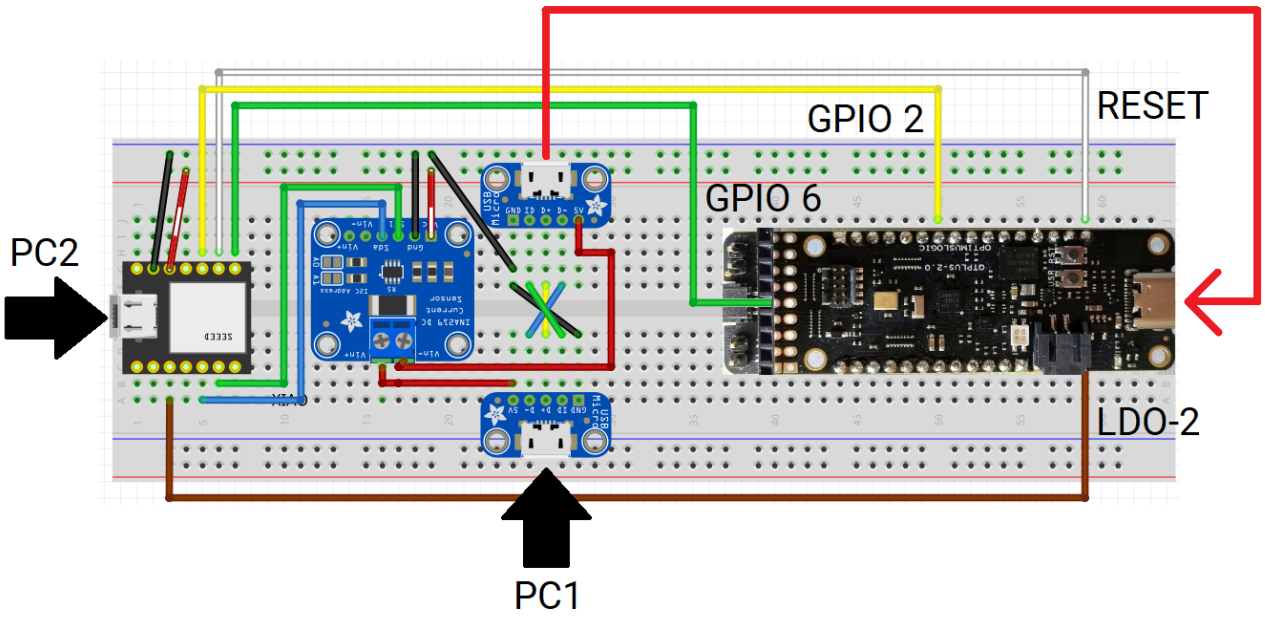


Figure 13: The experimental setup using a Seeeduino XIAO(left) [Hu23], INA219(middle) [Tex15] and QuickLogic Thing+(right) [Spa21].

This section is responsible for generation the end of the module definition and Verilog file. The "endmodule" statement is required but the "endmodule" statement can be replaced with any code.

4.4 Automatic Testing Hardware

We have created an automated automatic testing framework and a hardware experimental setup ,shown in Figure 13, to conduct the experiments in an efficient and consistent manner. A Seeeduino XIAO [Hu23] was used in combination with a PC to automatically flash the QuickLogic Thing+(QT+). The XIAO was connected to a INA219 [Tex15] voltage and current sensor to measure the power draw of the QT+ at the 5V USB input. The QT+ requires two actions to enter programming mode, in which it accepts a new FPGA design. Firstly, the reset button needs to be pressed and within 5 seconds, the USR_BTN needs to be pressed to enter programming mode. These two actions have been diverted to two General-Purpose Input/Output (GPIO) pins of the XIAO. This allows the XIAO to enable programming mode without human intervention. Furthermore, one of the GPIO pins of the XIAO is connected to the GPIO2 of the QT+. This GPIO pin is accessible in the FPGA and can be used to enable the ring oscillators. Finally, an Analog-to-Digital Converter (ADC) capable pin of the XIAO is connected to Test Point 2 (TP2) of the QT+. This allows the XIAO to measure the output voltage of LDO-2, which is responsible for powering the digital logic inside of the eFPGA core [Qui20]. This is used to determine if the output of the core can be considered valid, or in other words higher than 0.95V. The PC1 and PC2 inputs should be connected to the host PC. A cable should also be added between the top USB port and the QT+ USB port. A full wiring diagram is shown in Figure 13:

- The RESET wire is used to reset the QT+, this is required to enable the programming mode.
- The GPIO6 wire is connected to the USR_BTN line, which needs to be pulled low within 5

seconds of resetting the QT+ to enable the programming mode.

- GPIO2 is connected to one of the I/O pins of the EOS-S3. This pin is also available to the FPGA core.
- LDO-2 is connected to TP2 on the QT+. This is a test point which allows the measurement of the voltage which is being supplied to the FPGA digital logic.

4.5 Host PC Software

We have created a python program which can automatically generate Verilog HDL code, flash the board and run power tests on the USB interface and EOS-S3 core. The Host PC software consists of two parts: A python script and a Verilog generator written in bash. The entire testing framework with the used Verilog generators, python script and QT+ project is available at https://github.com/JsGraaf/Thesis_QT_Testing_Software.

4.5.1 QORC SDK

To generate the binary for the EOS-S3, the open-source tools that were suggested by the QuickLogic Corporation were used. V1.10.0 of the QORC_SDK was used, which is available at https://github.com/QuickLogic-Corp/qorc-sdk/releases/tag/v1.10.0. Two modifications were made to make the SDK compatible with the experimental setup:

- **Replacing the default project structure**: The default structure as proposed by the QORC_SDK lacks in several areas. It does not allow for easy compilation and flashing to the board. During the preliminary research, a GitHub repository with experimental and updated projects was found at https://github.com/coolbreeze413/qorc-onion-apps. This repository was created by one of the developers of the QORC_SDK and had a new and fully tested project structure with multiple quality-of-life improvements such as a framework of bash scripts and Makefiles with the ability to flash and debug boards. A blank project directory can be found in the GitHub of this thesis.
- **Flashing via a modified Makefile**: The experiments were compiled under Windows Subsystem for Linux 2 [CM22] as this is only possible on Linux. A downside to this approach is that WSL2 does not have access to the serial port via which the SparkFun QuickLogic Thing+ is flashed. This was solved by moving the programming script to the Windows machine and using a PowerShell script to launch it. This accesses the binary on the Linux partition while running under Windows, giving it access to the serial port. The command can be found inside the *flash_fpga_m4.sh* file in the .scaffolding folder inside of the blank project directory in the GitHub repository for this thesis.

Furthermore, all three of the SparkFun QuickLogic Thing+ boards had broken bootloaders from the manufacturer. None of them were able to enter programming mode and this had to be resolved by using a JTAG debugger and re-flashing the bootloader. This issue has been reported to the manufacturer SparkFun and a detailed guide on how to fix this can be found at https://forum.quicklogic.com/viewtopic.php?t=29. This guide was written for the original QuickLogic QuickFeather development kit but is applicable to the SparkFun board by substituting *"ql_loadflash.bin"* to *"qf_loadflash.bin"*

4.5.2 AutoTester.py Python Script

Our python script that is running on the host is responsible for the following:

- Creating and maintaining a serial connection to control and communicate with the XIAO micro-controller.
- Compiling the new FPGA configuration from the Verilog file and QT+ project. This is done in the background while the previous iteration is running to save time.
- Gathering the results from the XIAO micro-controller via the serial connection and outputting these to a XLSX workbook for further processing.

The command line inputs are shown in Table 1.

Name	Description	CLI Flag	Required	Default
QORC Port	The serial port to which the QuickLogic Thing+ is connected	-q	Yes	-
COM Port	The serial port to which the XIAO micro-controller is connected on the Windows OS	-p	Yes	-
Generator	The desired Verilog generator to use for this iteration. Must be located in the "Verilog_Generators" directory	-g	Yes	-
Circuit Count	The maximum number of circuits to generate using the Verilog generator	-c	No	200
Increment	The number of circuits to increment by. Starts at 0 and runs until the Circuit Count	-i	No	10
Test Length	Specifies the amount of time during which the measurements are taken by the XIAO micro-controller	-l	No	50000ms
Delay	Specifies the amount of time between measurements taken by the XIAO micro-controller	-d	No	500ms

Table 1: Description of input parameters for the PC host software

4.6 XIAO Micro-controller Software

The serial communication with the XIAO is responsible for starting the measurements and enabling the programming mode on the QT+. The commands are available in Table 2 and a typical

Command	Description
r	Resets the QT+ by pulling down the reset pin for 1 second, Waiting for 1 second for the QT+ bootloader and Pulling GPIO6 (USR_BTN) low for 1 second
t	Tests the connection with the XIAO Returns 'c' if successful
p	Run the power tests using the INA219 and 12-bit ADC for the core voltage. Executes them based on the specified parameters and outputs the results to the serial connection Pulls down the GPIO2 pin during the tests
y	Continuous power testing used for verifying the new ROs Measures every 500ms and outputs it to the Serial connection

Table 2: Overview of serial commands for the XIAO

communication flow for one generator iteration is available in Table 3. The serial communication is defined as follows:

Command send by host	XIAO return	Description
-	-	Serial connection is opened by the host
r	c	Starts the reset process for the QT+
-	d	Confirms that the reset process is finished
t	c	Tests the connection before starting the power tests
-	Measurement data xx:xx:xx:xx;...	XIAO outputs the measurement data in the form of 1. USB voltage (V) 2. USB power draw (mW) 3. USB current draw (mA) 4. LDO-2 voltage (V)

Table 3: Overview of the typical serial communication between the PC host and the XIAO for one generator iteration.

5 Experimental Results

Figures 14-21 show the results from the power hammer experiments. They display the highest amount of current drawn at the 5V USB input and the lowest voltage of the eFPGA core at different instance amounts of ring oscillator circuits. The 0.95V threshold for the eFPGA core is given for

reference. Also for reference: An empty design uses around 3.3mA with a core voltage of 1.15V. A 32-input 4-bit clocked adder tree that uses 81 CLBs consumes 8mA with a core voltage of 1.15V. The measurements were performed with 500ms intervals for a total duration of 50 seconds. The lowest core voltage was plotted together with the highest current draw. All results are available in the GitHub repository for this thesis at https://github.com/JsGraaf/Thesis_QT_Testing_Software.git.

5.1 Look-Up Table (LUT) based ROs results

Figure 14-17 show that in all cases the core voltage starts to drop after the current draw raises above 15mA. In all four cases this happens at a different amount of instances with the LUT1 based ROs making the voltage drop after only 25 instances or 2.8% of the entire eFPGA. The 0.95V threshold is reached after 65 instances or 7.3% of the eFPGA. The results for the LUT2, LUT3, and LUT4 primitives are shown in Table 4. Figure 14-17 also show that the current draw never exceeds 16mA. When the core voltage is close to the threshold of 0.95V, the current drops to around 14mA. While the different LUT primitives start to drop core voltage at different times, they all show the same behaviour with regards to the current: The current remains at 15mA when the core voltage starts to drop and drops to 14mA when the core voltage reaches 0.95V.

Primitive	Voltage Drop		Voltage Threshold	
	Instances	% of total	Instances	% of total
LUT1	25	2.8	65	7.3
LUT2	55	6.2	125	14.0
LUT3	55	6.2	125	14.0
LUT4	35	3.9	80	9.0

Table 4: LUT primitive based results.

5.2 MUX based RO results

Figure 18-20 show similar behaviour to the LUT based ring oscillators. Again, the core voltage starts to drop when 16mA is reached, dropping down to 14mA when the core voltage approaches 0.95V. The MUX2 primitive requires the least amount of instances before the core voltage starts to drop at 25 instances or 2.8% of the entire eFPGA. The voltage threshold is reached with 65 instances or 7.3% of the entire eFPGA. The results from the MUX4 and MUX8 primitive based ROs are visible in Table 5.

5.3 NOT3 based RO results

Figure 21 shows that it takes 45 instances of NOT3 ring oscillators or 15.3% of the entire FPGA, to hit 16mA current draw. The percentage is relatively high because one functional NOT3 based ring oscillator requires three NOT primitives, each requiring one CLB. After this point, the voltage

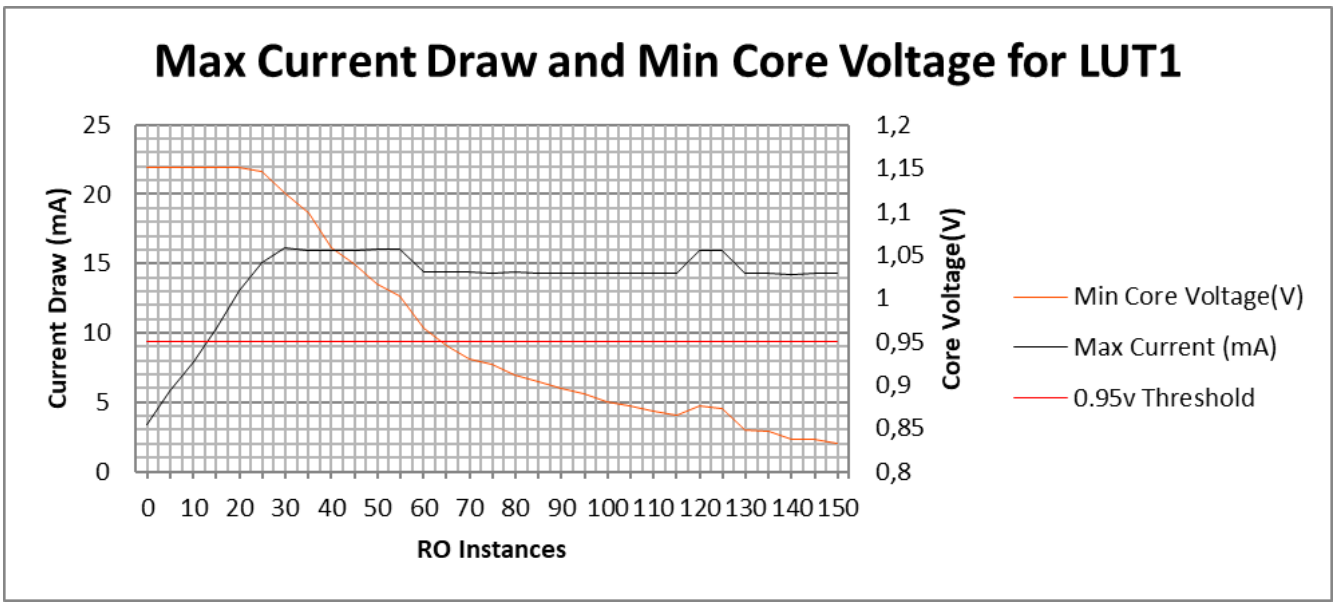


Figure 14: Results of LUT1 ring oscillator per instance amount. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.

Primitive	Voltage Drop		Voltage Threshold	
	Instances	% of total	Instances	% of total
MUX2	25	2.8	65	7.3
MUX4	55	6.2	125	14.0
MUX8	35	3.9	80	9.0

Table 5: MUX primitive based results.

starts to drop and the current drops to around 14mA after the core voltage starts to reach the 0.95V mark at 95 instances or 32.0% of the entire eFPGA. At 5 instances, the core voltage drops to 1.1V and climbs back up to 1.15V at 10 RO instances. This appears to be an artifact.

5.4 LUT2 with Enable results

To confirm that the voltage drop is caused by the RO instances, a LUT2 based implementation was created with a different INIT variable. This allowed for enabling and disabling the circuit, making it possible to see the core voltage level with and without the RO instances active. The implementation is shown in Figure 23 and the results in Figure 22. Figure 22 shows that the eFPGA has a current draw of around 3 mA and a core voltage of around 1.15V when the RO instances are not active. When the RO instances are enabled after 8.5 seconds, the current jumps to around 14 mA which is consistent with the LUT, MUX and NOT3 results. The core voltage also drops below the 0.95V threshold with occasional spikes slightly above 0.95V.

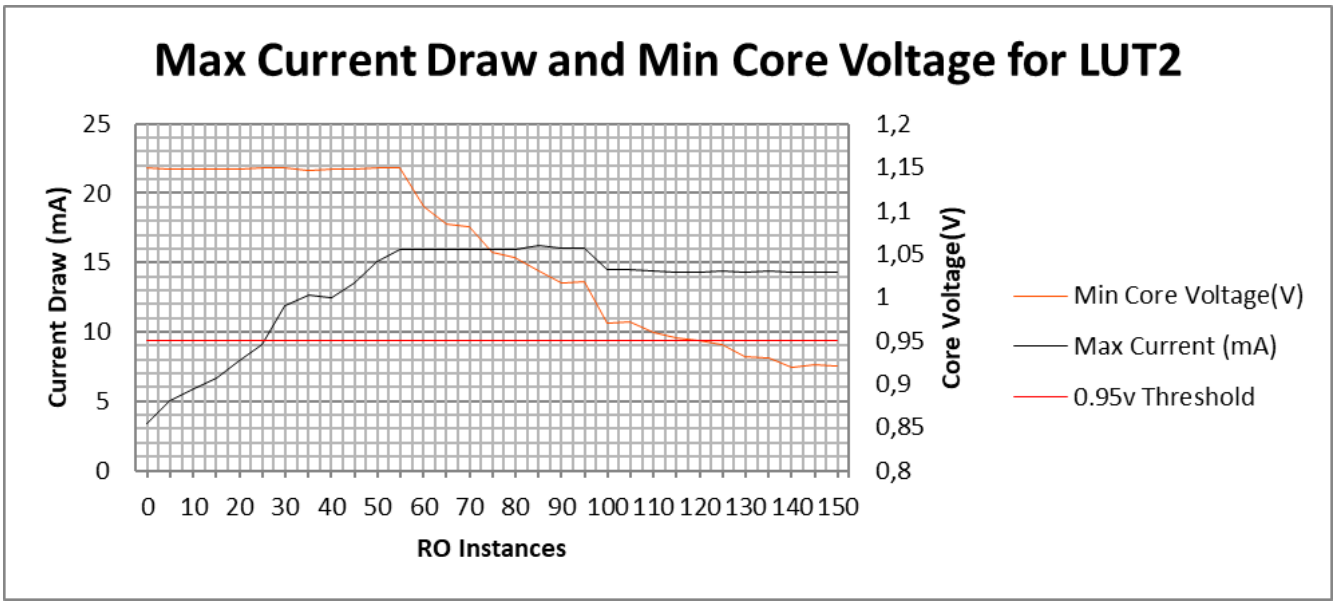


Figure 15: Results of LUT2 ring oscillator per instance amount. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.

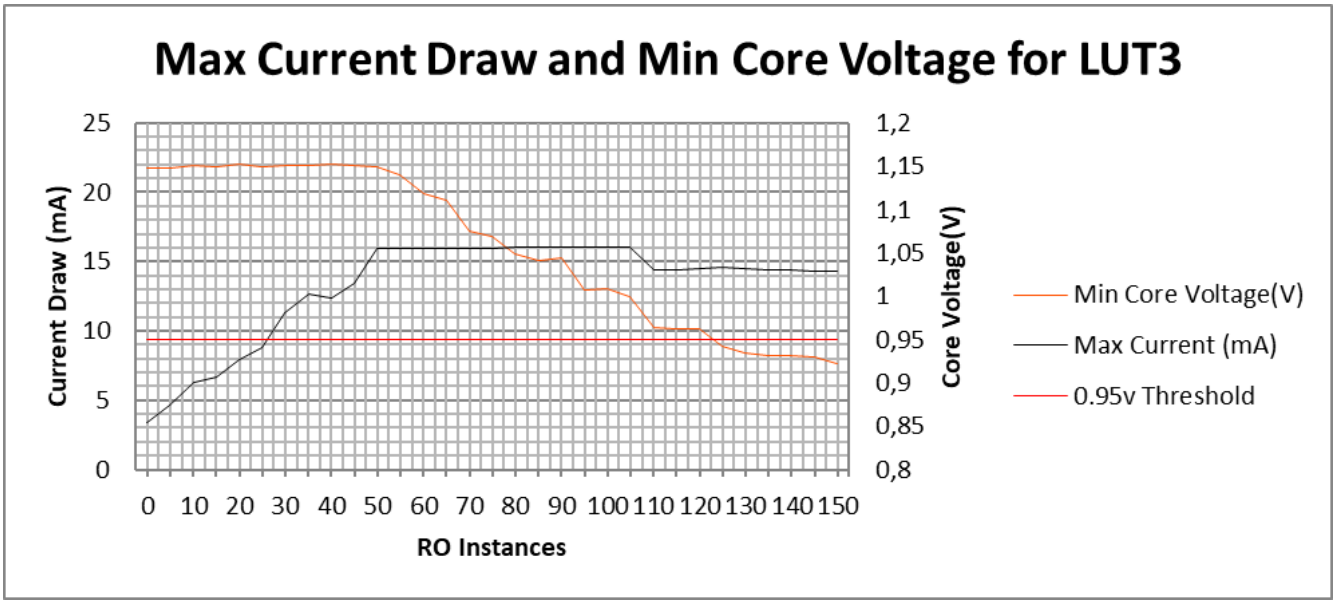


Figure 16: Results of LUT3 ring oscillator per instance amount. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.

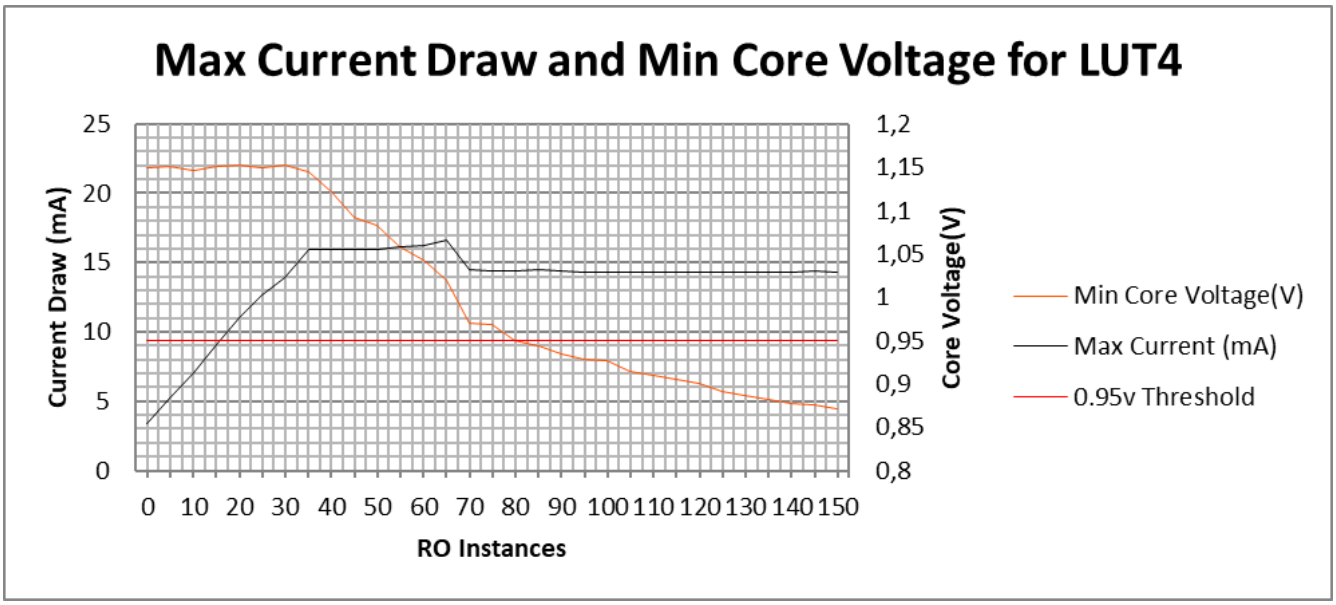


Figure 17: Results of LUT4 ring oscillator per instance amount. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.

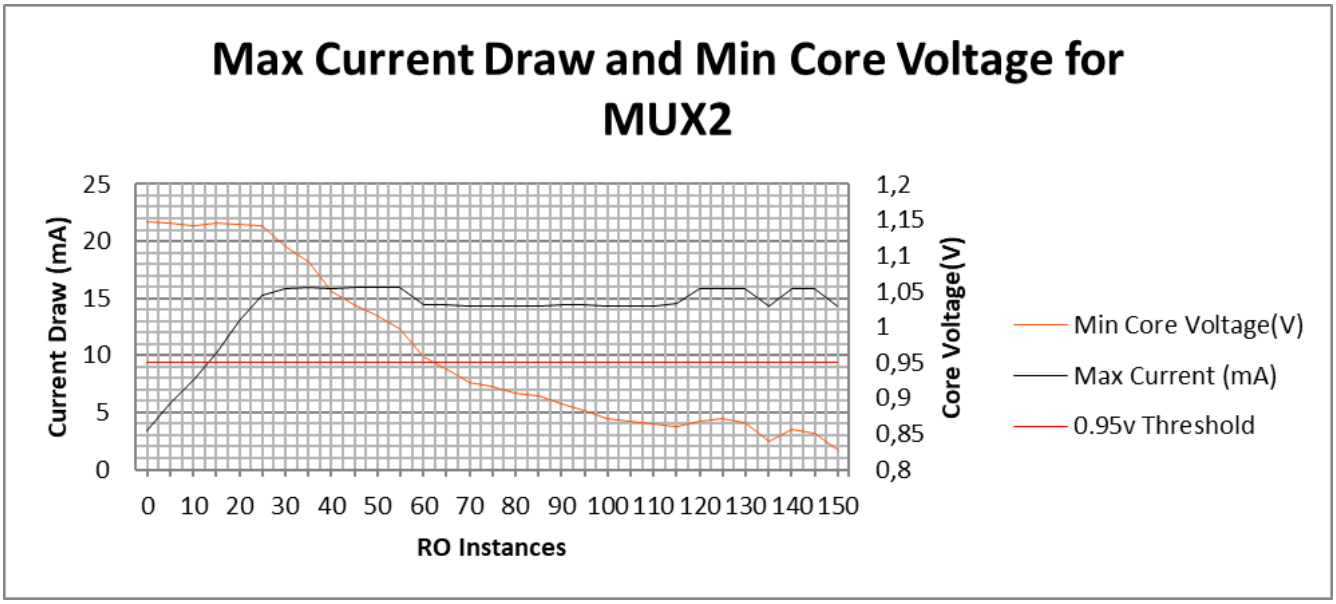


Figure 18: Results of MUX2 ring oscillator per instance amount. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.

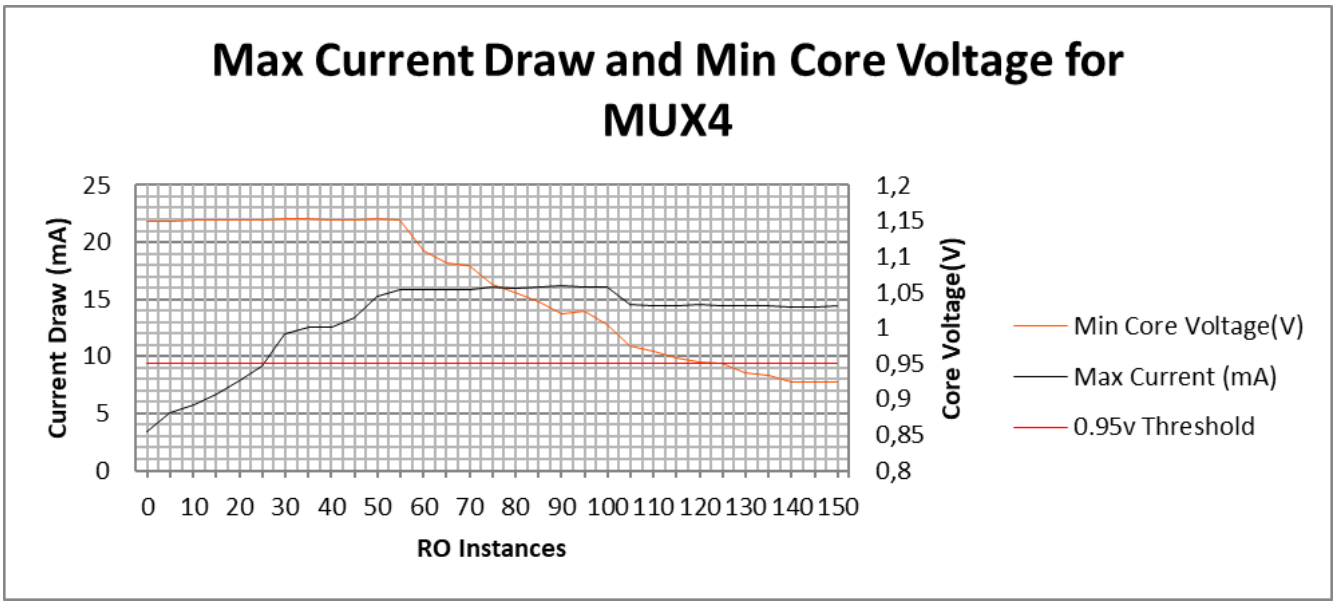


Figure 19: Results of MUX4 ring oscillator per instance amount. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.

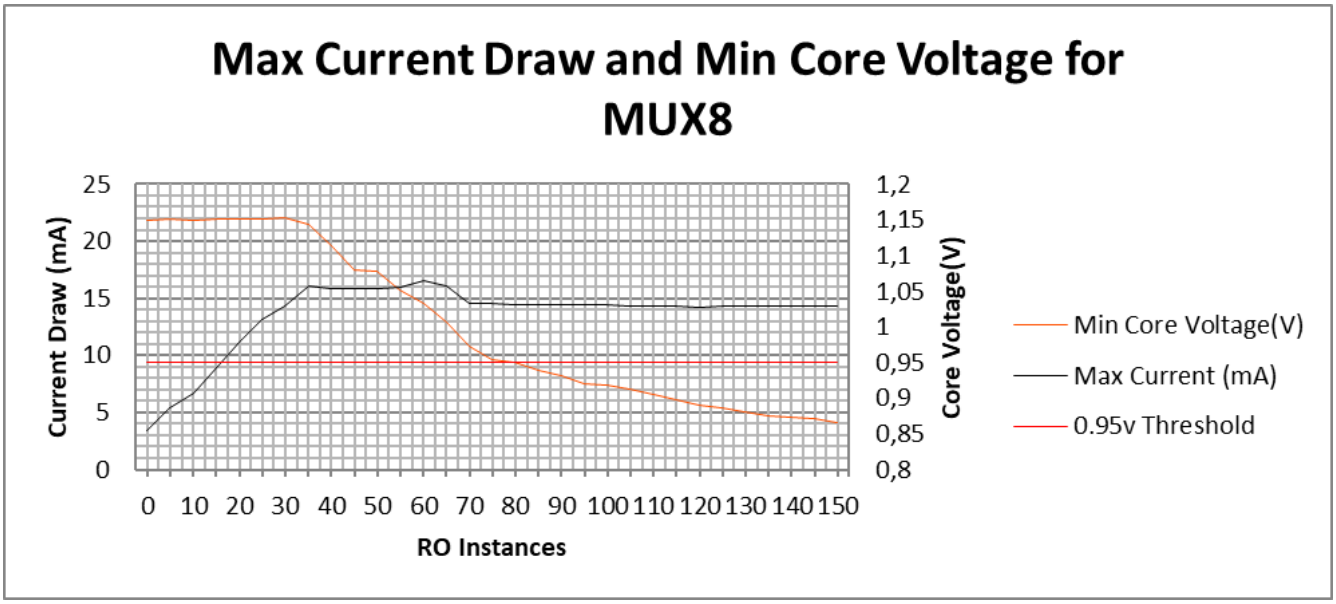


Figure 20: Results of MUX8 ring oscillator per instance amount. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.

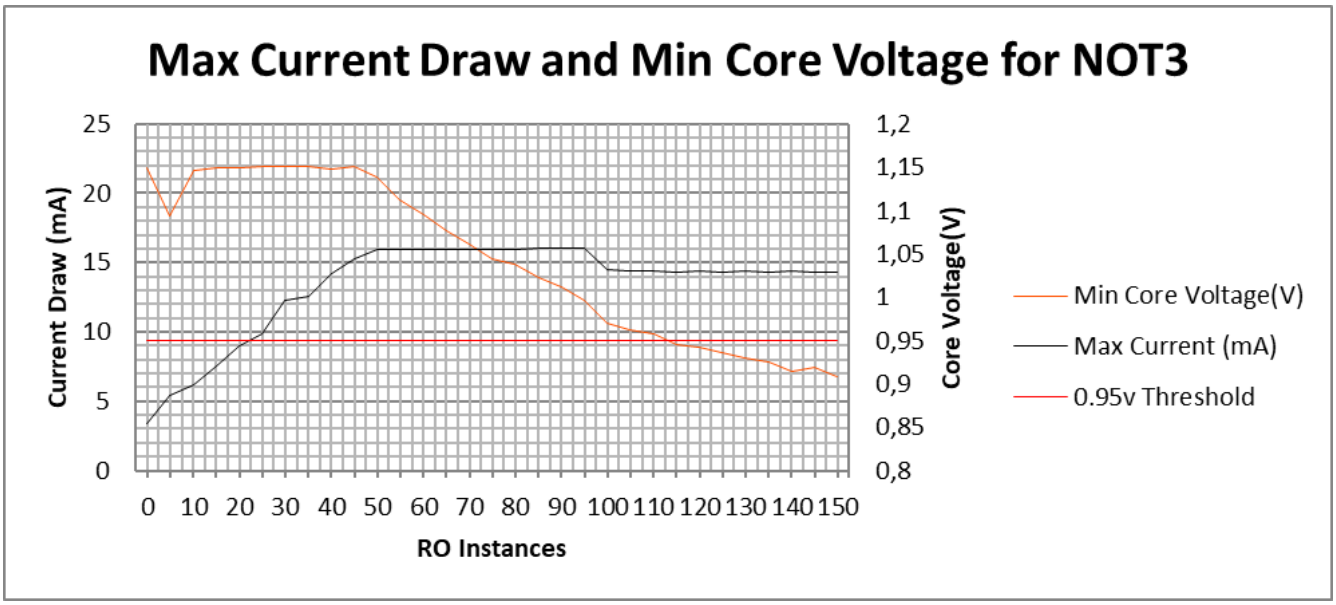


Figure 21: Results of NOT3 ring oscillator per instance amount. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.

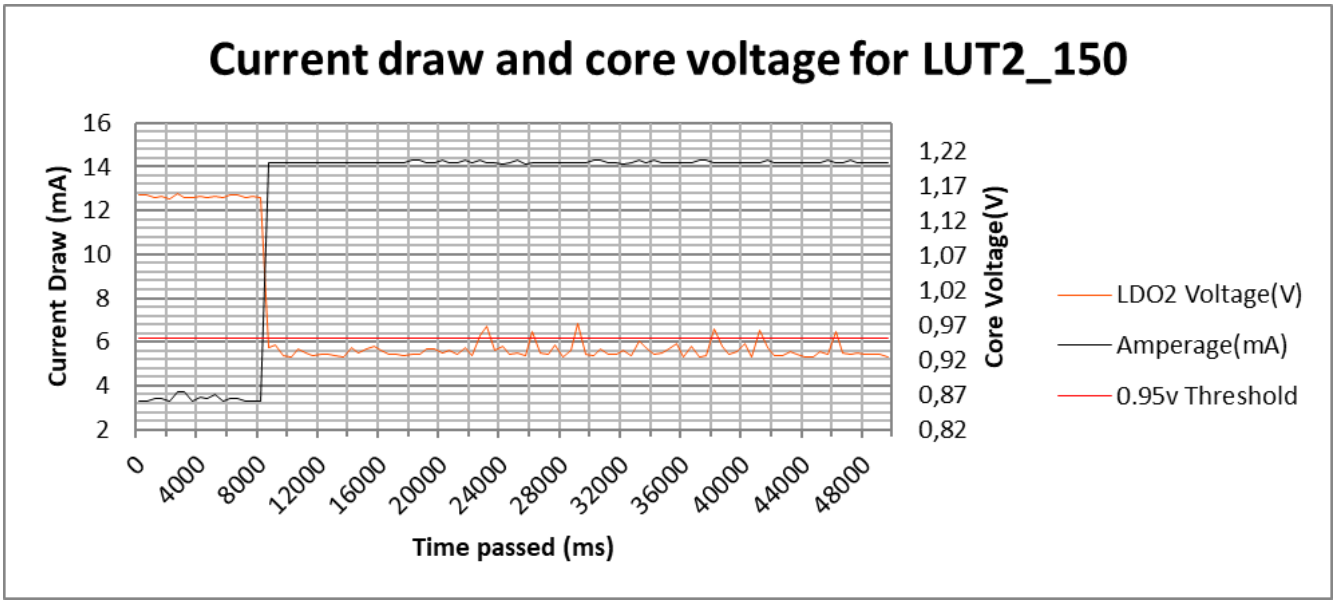
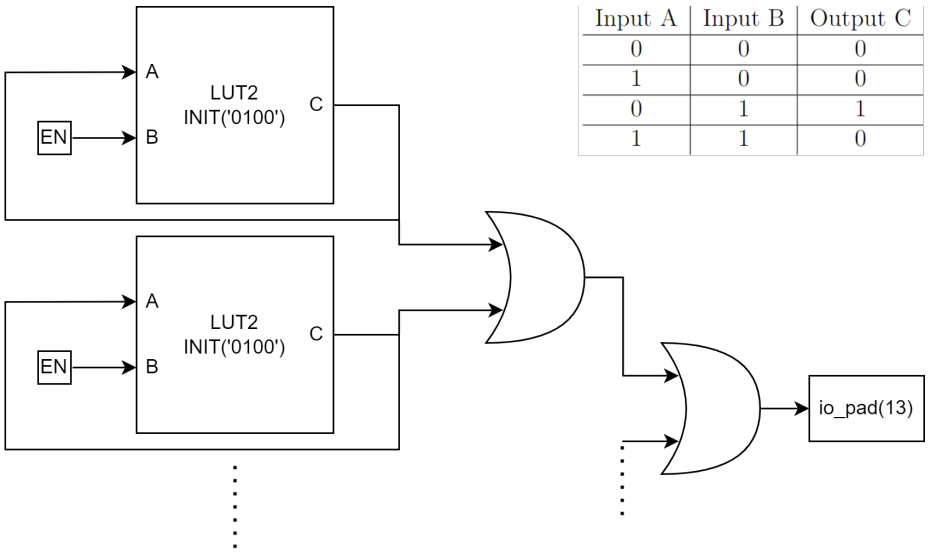


Figure 22: Results of LUT2 with Enable ring oscillators with 150 instances. All ROs have an enable delay of 8.5 seconds. The current draw at the 5V USB port and the voltage at the eFPGA core are shown. The 0.95V threshold for validity is also given for reference.



Input A	Input B	Output C
0	0	0
1	0	0
0	1	1
1	1	0

Figure 23: LUT2 based Ring Oscillator with enable signal.

5.5 Adder with RO instances

The behaviour of the EOS-S3 while having a core voltage lower than 0.95V has also been investigated. 200 LUT2 RO instances with an 8.5 second delay were added to the 32-input 4-bit adder tree which was mentioned in Section 5. The eFPGA core checks the output of the adder with predefined constants to the correct output. When this output is correct, the green LED lights up. When the USR_BTN of the SparkFun QT+ is pressed, one of the inputs is changed. This results in the green LED turning off, since the output is no longer correct. The adder uses 81 logic CLBs or 10% of the EOS-S3 FPGA core. Figure 24 shows that the adder consumes around 7mA until the 8.5 delay. At this point the RO instances are enabled and the current jumps up to 18mA This is higher than the previous 16mA but the green LED was turned on, unlike in the previous experiments.

The USR_BTN was pressed for 10 seconds at the 20 second mark and 40 second mark. The green LED instantly turned off and after the button was released, instantly turned back on. The QT+ consumed 2mA less while the green LED was turned off.

Note: This experiment was also conducted with 150 RO instances, but this did not cause the core voltage to drop below 0.95V.

6 Discussions

All eight different ROs show consistent behaviour as explained in Section 5. In all cases, a certain amount of RO instances use up the power budget of the eFPGA at which point the core voltage starts to drop. The LUT1 and MUX2 based ROs required only 25 instances, or 2.8% of the entire eFPGA, before the current limit is reached. In both cases, the voltage threshold is passed with 65 instances (7.3%). The eFPGA core seems to be current limited when the 5V USB current is at 16mA. This means that the difference between an empty design, which consumes 3.3mA and

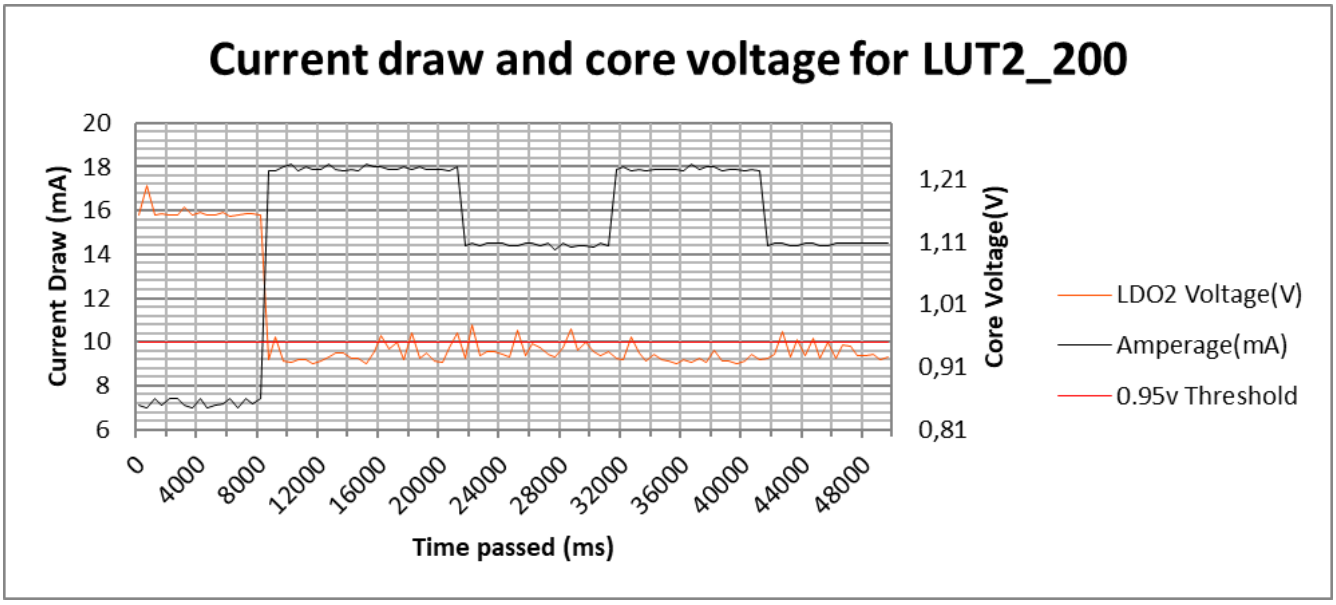


Figure 24: 200 instances of LUT2 based Ring Oscillators with a 32-input 4-bit adder

the peak draw is $16 - 3.3 = 12.7mA$ at the 5V input bus. Due to the layout of the SparkFun QT+, it is not possible to measure the input current and compare it with the output current to definitively determine the current draw at the eFPGA core. Future research would benefit from placing the EOS-S3 on a different PCB with access to both the LDO_VIN and LDO_VOUT ports to determine this current draw. However, Section 5.4 shows that the increase in current is due to the RO instances.

6.1 Different voltage drops for different primitives

Section 5 shows that the voltage drop differs per primitive. For example, the MUX2 primitive based ROs drop the current much quicker than the MUX8 primitive based ROs. This is due to the different fragments that are used inside of a single CLB. As mentioned in Section 2.4, the power draw of a circuit is based on the switching frequency of the ROs. This switching frequency is determined by the number of gates that the signal has to pass through before reaching the output. The MUX2 primitive uses the f-fragment of a CLB [QC21] and the signal only has to pass through a single multiplexer, as shown in Appendix A. This means that the propagation delay is mostly determined by the propagation delay of that one single multiplexer. Meanwhile, the MUX8 primitive uses the tb-fragment of the CLB [QC21], this is a combination of the t and b-fragment of the CLB. Appendix A shows that this signal has to go through 8 multiplexers and a D-flip flop. This results in a higher propagation delay that further results in a lower switching frequency and thereby a lower power draw. This lower power draw means that the voltage drop is less significant with 60 MUX8 primitive based ROs when compared to the same number of MUX2 primitive based ROs.

Another important note is that the power draw is not only dependent on the propagation delay but also on the number of gates that change state with every cycle. Table 5 shows that the MUX4

primitive requires more instances to reach the voltage threshold than the MUX8 primitive. The MUX4 primitive uses the same tb-fragment of the CLB but multiple input ports are tied to ground [QC21]. This means that a smaller number of gates is changing state with each clock cycle when compared to the MUX8 primitive, resulting in a smaller power draw. This in turn results in a smaller voltage drop.

6.2 Bootloader current limit

Section 5.5 shows that the eFPGA seems to be working fine when 200 RO instances are enabled together with our 4-bit adder tree, as the adder tree seems to produce a correct result. Figure 8 and the text in Section 4.1.1 show that the proper operation of the eFPGA core is no longer guaranteed after breaking the 0.95V threshold. This was also confirmed by the QuickLogic Corporation in a private communication. This fact combined with the hard current limit prompted a further investigation into the EOS-S3 data sheet and technical manual. Figure 25 shows that the current for the LDO-2, here noted by LDO_30_IMAX, can be configured in the bootloader. with the default being 8mA. This is inconsistent with the results from the experiments, which showed a limit of 16mA. We have performed a separate experiment, at which the LDO_30_IMAX was set to the minimal 1mA and the maximum 30mA configuration. We found that the 1mA configuration resulted in a maximum draw of 57mA at the 5V USB at 900 RO instances with a core voltage lower than 0.95V. The 30mA configuration resulted in a maximum draw of 14mA at the 5v USB with 50 RO instances and a core voltage lower than 0.95V. The latter is consistent with the current limit that was hit by the experiments. This shows that the EOS-S3 has power monitoring circuits which prevent the LDOs from burning out. Further investigation with the 1mA configuration could result in even more current draw. This would require the modification of the bootloader and thus physical access which is outside of the scope of the proposed attack and thereby this thesis. This could however be another attack vector which is specific to the EOS-S3 SoC.

	LDO_30_IMAX	4:2	RW	0x3	Configures the control for maximum expected current imax current (mA) 000 1 001 2 010 4 **011 8** 100 12 101 16 110 22 111 30

Figure 25: LDO_30_IMAX bootloader setting. Page 150 from the EOS-S3 technical reference manual [Qui17]

6.3 Core voltage

As soon as the current hits its peak, the voltage starts to drop. This shows that the EOS-S3 core cannot supply more power in its current configuration. During the experiments, the pin depicted by GPIO2 in Figure 13, that is connected between the XIAO and the QT+ also experienced a voltage drop. When the pin is enabled high, the voltage stopped at 0.623V instead of reaching the 3.3v of

```
Warning 1: Detected 400 strongly connected component(s) forming combinational
           loop(s) in timing graph
Warning 2: Arbitrarily disabling timing graph
           edge 5 (lut1.t_frag.XAB[0] -> lut1.t_frag.XZ[0])
           to break combinational loop
```

Figure 26: Warning displayed by VPR in the MODULE_Top.log file when 400 LUT1 RO instances are implemented

the desired high signal. This is not the case when the current limit is not reached. This further shows that the power budget of the EOS-S3 is saturated and that power hammering circuits can have an effect on not only the SoC but the entire system. Since the EOS-S3 eFPGA performance is not guaranteed at a core voltage of 0.95V or below, our experiments show that the eFPGA core can be disrupted by using as little as 2.8% of the eFPGA. However, the EOS-S3 SoC does not require a full power cycle to become usable again as is the case with other eFPGAs [GOT17]. It is important to note that the EOS-S3 never crashed or stopped functioning correctly during the experiments.

6.4 Analysis of the security of SymbiFlow

After compilation, the SymbiFlow design flow offers no warning or errors in the command terminal for any of the tried malicious circuits. In the build directory of the FPGA section of the project, multiple log files are created. One is generated by the Yosys synthesis suite and three by VPR. Only the VPR log files contain warnings about a combinatorial circuit and these warnings are displayed near the beginning of the log file. One of these warnings is shown in Figure 26. These are a result of the timing analysis, which calculates the maximum clock frequency of the virtual clocks inside of the eFPGA. The timing analysis is performed on a local copy of the design and the combinatorial loops are only broken inside the copy. This is required such that there is no loop inside of the design which would prevent a correct timing analysis. This breaking of combinatorial loops has no effect on the final design implementation. The user is not warned that the combinatorial loop can be malicious and no further precautions are taken to prevent malicious intent.

Commercial synthesizers like Xilinx Vivado [Fei12] offer a Design Rule Checker (DRC). The designer can create rules which the HDL design has to follow and the DRC can give warnings when they are broken. The DRC does not only check during the timing analysis but during every step of the synthesis. The warnings are categorized in three categories: Advisory, Warning and Critical Warning. The critical warnings prevent the user from implementing the design and require manual override in case the circuit is as desired [Xil23]. Table 2 in [LMG+20] shows that the Vivado DRC recognizes many of the ring oscillators and generates the appropriate warnings. SymbiFlow would benefit from this DRC system and could inform the user of the potential hazards within the design.

Furthermore, nothing was found which prevented the compilation and implementation of a malicious design or presented a warning in the command prompt.

7 Conclusions and Further Research

This thesis presented an automated testing framework for evaluating the EOS-S3 subsystem and performed an analysis of the open-source SymbiFlow design flow which is used to program the EOS-S3. It showed that eFPGAs are under threat from power hammering designs and that little to no warning is given to the command prompt after compilation. The only warnings given are buried within multiple directory layers in between 20+ files and don't inform the user of possible malicious intent.

The first research question is answered by performing and presenting experiments and results indicating that the power hammering resulted in lower voltages inside of the EOS-S3 core but also lowered the voltage of connected peripherals. For some power hammering designs like the LUT2 primitive based design, only one CLB is required for a power draining effect, which can easily be embedded into an IP. This malicious circuit could result in a decrease in battery life of mobile FPGAs and potentially damage or break the eFPGA. The MUX2 primitive based ring oscillator is the most effective power hammering circuit available on the EOS-S3. It only utilizes the f-fragment of a CLB and only needs 65 circuits or 7.3% of the FPGA to saturate the power budget. The LUT1 primitive based ring oscillator also requires 65 instances but is implemented on the t-fragment of the CLB. This section is more frequently used in designs then the f-fragment and it would therefore be harder to embed malicious circuits in the fragment. This could be different on other FPGAs with different CLB configurations. It is important to note that the EOS-S3 never showed signs of incorrect functioning during the experiments.

The second research question is answered as follows: The SymbiFlow design flow should be improved by scanning the log files for errors and reporting them back to the command prompt. This would warn potentially inexperienced users about combinatorial loops and prevent malicious design implementation. It should not prevent the user from compiling certain circuits since this limits the versatility of the software. The EOS-S3 is well protected in such a way that the SoC performs power monitoring such that the electrical components cannot be crashed. This could however change with different bootloader settings but this requires further research.

In further research, the LDO_30_IMAX bootloader setting should be investigated. By changing this setting, the total power draw of the EOS-S3 SoC could be further increased, resulting in an even higher waste of power and could have currently unknown consequences on the rest of the system. The EOS-S3 should be placed on a PCB where the LDO_VIN, LDO1 and LDO2 ports are exposed. This would allow for an accurate measurement at the EOS-S3 core.

8 Acknowledgments

I would like to thank the Leiden Embedded Research Center (LERC) for providing the SparkFun QuickLogic Thing+ boards and other materials required to carry out this research.

References

[AAE06] Amara Amara, Frédéric Amiel, and Thomas Ea. Fpga vs. asic for low power applications. *Microelectronics Journal*, 37(8):669–677, 2006. https://www.sciencedirect.com/science/article/pii/S0026269205003927.

[CK00] David G Chinnery and Kurt Keutzer. Closing the gap between asic and custom: An asic perspective. In *dac*, volume 10, pages 337292–337602. Citeseer, 2000.

[CM22] Craigloewen-Msft. Windows subsystem for linux documentation, Jun 2022. https://learn.microsoft.com/en-us/windows/wsl/.

[CoQ20] CoQube. Fpga design flow, Feb 2020. Visited 26-06-2023, https://coqube.com/fpga-design-flow/.

[EOS18] Quicklogic EOSS3. Quicklogic eos-s3 + efpga soc, August 2018. Visited 18-06-2023, https://www.quicklogic.com/products/soc/eos-s3-microcontroller/.

[ES08] Michael Engel and Olaf Spinczyk. Aspects in hardware: What do they look like? ACP4IS '08, New York, NY, USA, 2008. Association for Computing Machinery.

[Fei12] Tom Feist. Vivado design suite. *White Paper*, 5:30, 2012.

[Fle05] Bryan H Fletcher. Fpga embedded processors. In *Embedded Systems Conference*, page 18, 2005.

[GOT17] Dennis R. E. Gnad, Fabian Oboril, and Mehdi B. Tahoori. Voltage drop-based fault attacks on fpgas using valid bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, 2017.

[Hu23] Shuxu Hu. Seeeduino xiao, Jan 2023. Visited 21-06-2023, https://wiki.seeedstudio.com/Seeeduino-XIAO/.

[Int23] Mordor Intelligence. Field programmable gate array (fpga) market size, 2023. Visited 23-06-2023, https://www.mordorintelligence.com/industry-reports/field-programmable-gate-array-fpga-market/market-size.

[KGT18] Jonas Krautter, Dennis R. E. Gnad, and Mehdi B. Tahoori. Fpgahammer: Remote voltage fault attacks on shared fpgas, suitable for dfa on aes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):44–68, Aug. 2018.

[KR06] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, page 21–30, New York, NY, USA, 2006. Association for Computing Machinery.

[Leo08] Philip H. W. Leong. Recent trends in fpga architectures and applications. In *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*, pages 137–141, 2008.

[LGM+12] John W. Lockwood, Adwait Gupte, Nishit Mehta, Michaela Blott, Tom English, and Kees Vissers. A low-latency library in fpga hardware for high-frequency trading (hft). In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, pages 9–16, 2012.

[LMG+20] Tuan Minh La, Kaspar Matas, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. Fpgadefender: Malicious self-oscillator scanning for xilinx ultrascale + fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 13(3), sep 2020.

[MPZ+20] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jai Min Wang, Mohamed ElDafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. Vtr 8: High performance cad and customizable fpga architecture modelling. *ACM Trans. Reconfigurable Technol. Syst.*, 2020.

[MS10] Mrinal Mandal and Bishnu Charan Sarkar. Ring oscillators: Characteristics and applications. *Indian Journal of Pure and Applied Physics*, 48:136–145, 02 2010.

[MS19] Dina Mahmoud and Mirjana Stojilović. Timing violation induced faults in multi-tenant fpgas. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1745–1750, 2019.

[OS23] Open-Source. Symbiflow, 2023. Visited 29-06-2023, https://github.com/SymbiFlow.

[PHT19] George Provelengios, Daniel Holcomb, and Russell Tessier. Characterizing power distribution attacks in multi-user fpga environments. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 194–201, 2019.

[QC21] QuickLogic-Corp. Quicklogic-corp/eos-s3: Device description files (architecture, timing, configuration bitstream, and general documentation) for eos s3 mcu+efpga soc, Sep 2021. Visited 24-06-2023, https://github.com/QuickLogic-Corp/EOS-S3/blob/master/Device%20Architecture%20Files/QLAL4S3B_lib.xml.

[Qui17] QuickLogic Corporation. *QuickLogic EOS S3 Ultra Low Power multicore MCU technical reference*, 2017. Version 1.0.0, https://www.quicklogic.com/wp-content/uploads/2020/06/QL-S3-Technical-Reference-Manual-Vol-1.pdf.

[Qui20] QuickLogic Corporation. *QuickLogic EOS S3 Ultra Low Power multicore MCU*, 2020. Version 3.3d, https://www.quicklogic.com/wp-content/uploads/2020/06/QL-EOS-S3-Ultra-Low-Power-multicore-MCU-Datasheet.pdf.

[Spa21] Sparkfun. Sparkfun quicklogic thing plus - eos s3, 2021. Visited 18-06-2023, https://www.sparkfun.com/products/17273.

[Tex15] Texas Instruments. *INA219 Zerø-Drift, Bidirectional Current/Power Monitor With I 2C Interface*, 2015. Revision December 2015, https://www.ti.com/lit/ds/symlink/ina219.pdf.

[Tri15] Stephen M. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331, 2015.

[Utm21] Utmel. What is an asic chip?, Nov 2021. Visited 22-06-2023, https://www.utmel.com/blog/categories/integrated%20circuit/what-is-an-asic-chip.

[Utm22] Utmel. What is efpga?, Jan 2022. Visited 15-06-2023, https://www.utmel.com/blog/categories/integrated%20circuit/what-is-efpga.

[vSNBN06] T. von Sydow, B. Neumann, H. Blume, and T. G. Noll. Quantitative analysis of embedded fpga-architectures for arithmetic. In *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 125–131, 2006.

[Xil23] Xilinx. Xilinx documentation, Jan 2023. Visited 05-07-2023, https://docs.xilinx.com/r/en-US/ug899-vivado-io-clock-planning/Viewing-DRC-Violations.

[Yos23] YosysHQ. Yosyshq/yosys: Yosys open synthesis suite, 2023. Version 0.30, https://github.com/YosysHQ/yosys.

[ZQ14] Jiliang Zhang and Gang Qu. A survey on security and trust of fpga-based systems. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 147–152, 2014.

A Appendix: EOS-S3 CLB

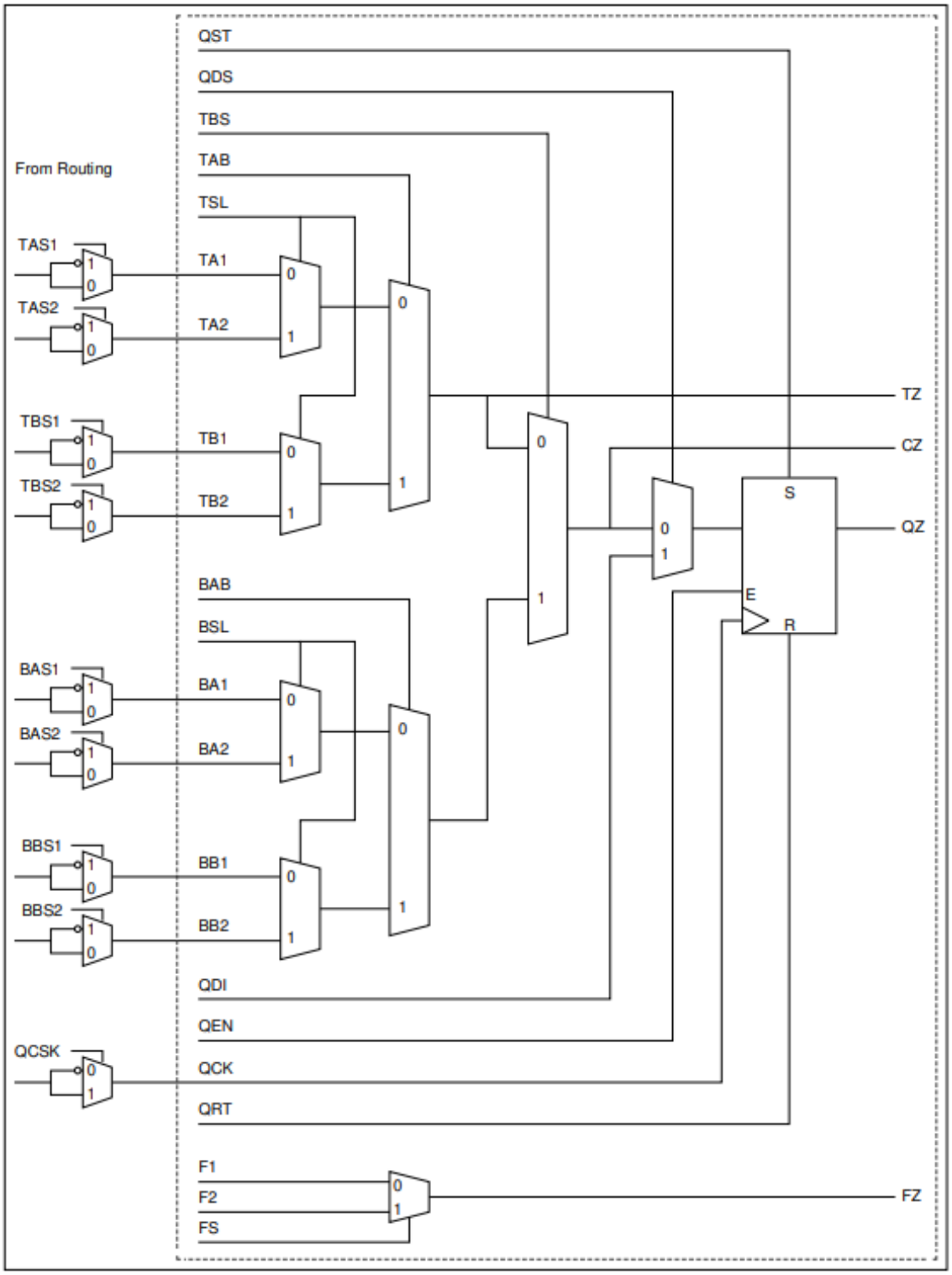


Figure 27: EOS-S3 CLB, From https://www.quicklogic.com/wp-content/uploads/2020/06/QL-EOS-S3-Ultra-Low-Power-multicore-MCU-Datasheet.pdf, figure 36

B Appendix: NOT3 Verilog Generator

```
#!/bin/bash

### NOT3 Seperate generator for verilog code for QT plus ###

# Usage: ./<Filename> <Amount of circuits> <output file path>
FILE_NAME='MODULE_top.txt'
CIRCUIT_NAME="NOT3 Seperate"
# Check if arguments where given
if [ $# -eq 0 ]
  then
    echo "No arguments supplied"
    exit
fi

# Get amount from arguments
AMOUNT=$1
OUTPUT_FILE_PATH=$2

echo "Generating ${1} $CIRCUIT_NAME"

### INITIALISATION OF SUBMODULES ###
echo """module not_3_ro(
   input en,
   output out,
);

assign connect[0] = en ? connect[3] : 0;
assign out = en ? connect[3] : 0;

(*keep*)wire [3:0] connect;

inv inv1 (
   .A(connect[0]),
   .Q(connect[1])
);

inv inv2 (
   .A(connect[1]),
   .Q(connect[2]),
);

inv inv3 (
   .A(connect[2]),
   .Q(connect[3])
);
endmodule
""" > $FILE_NAME

### TOP OF MODULE ###
echo """
module MODULE_top(
```

```
   io_pad
);

// GPIO
inout    wire     [31:0]      io_pad      ;
""" >> $FILE_NAME

if [ "$AMOUNT" -lt "2" ]; then
   if ["$AMOUNT" -eq "1"]; then
      echo "wire out;" >> $FILE_NAME
   else
      echo "wire [2:0] out;" >> $FILE_NAME
   fi
else
   echo "wire [$(( AMOUNT-1 )):0] out;" >> $FILE_NAME
fi
echo """
reg en;
initial begin
   en <= 1;
end
""" >> $FILE_NAME

### INSTANCE GENERATION (individual Circuit) ###
if [ "$AMOUNT" -gt "0" ]; then
   for i in $( eval echo {1..$AMOUNT} )
   do
      echo """not_3_ro ro$((i - 1)) (
   .en(en),
   .out(out[$((i-1))]),
);
""" >> $FILE_NAME
   done
else # In case the design is empty (circuits = 0), add 1 inverter
echo """inv invBase (
   .A(io_pad[2]),
   .Q(io_pad[4])
);
""" >> $FILE_NAME
fi

### END OF MODULE ###
echo 'assign io_pad[2] = |out;' >> $FILE_NAME
echo 'endmodule' >> $FILE_NAME

mv $FILE_NAME $OUTPUT_FILE_PATH
```