CPE 426 - Introduction to Hardware Security

Lab 1 - PUF

Presented to:
Dr. Stephen R. Beard
California Polytechnic State University, San Luis Obispo

Written by: Hardware InsecuritEE
Qingyu Han, Weston Keitz, Nathan Jaggers

Fall 2023

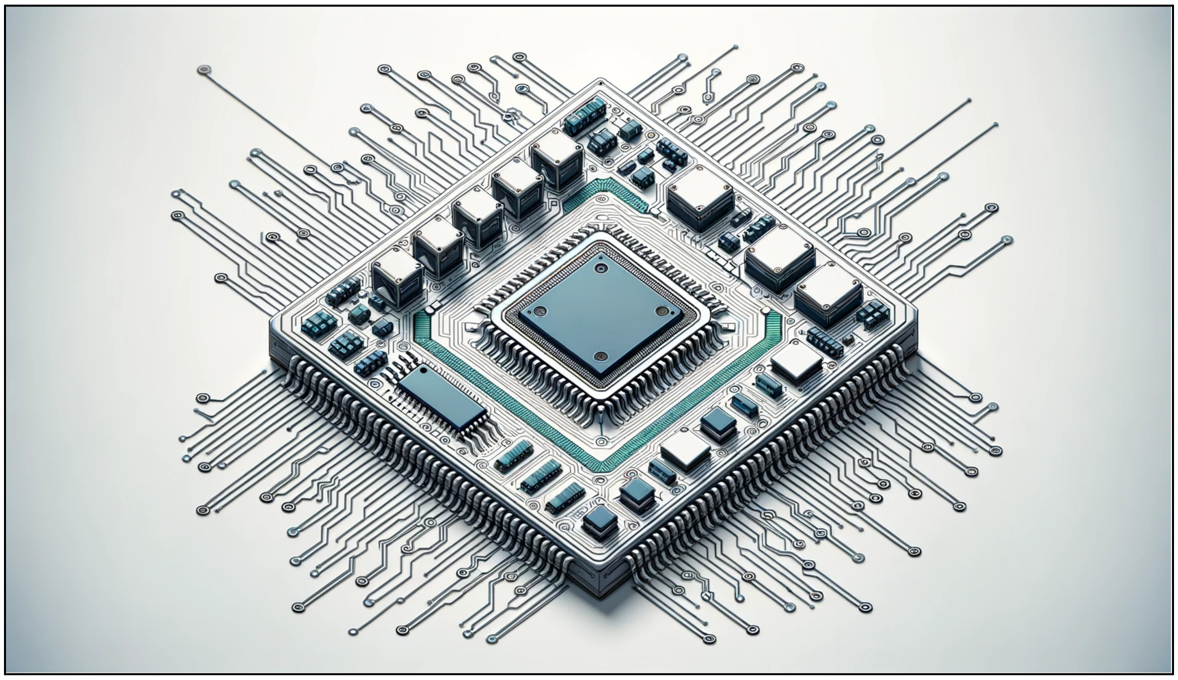
Table of Contents

I. Abstract

This lab exercise explores the design and implementation of a Ring-Oscillator based Physically Unclonable Function (PUF) on FPGA, aimed at imparting a practical understanding of hardware security. By adapting a framework similar to a well-documented design, students engage in developing a variant of the PUF. The lab encompasses various sub-tasks, from creating a configurable ring-oscillator to establishing a working PUF and incorporating a SHA128 hashing algorithm. The design is evaluated across different FPGA locations to analyze the consistency and reliability of the challenge-response pairs.

II. Introduction

Physically Unclonable Functions (PUFs) are central to hardware security, providing a mechanism for secure key storage/generation and device authentication. Ring -Oscillator based PUFs are particularly notable in FPGA environments due to their simplicity. They leverage the inherent process variations in silicon manufacturing to generate unique fingerprints, forming the basis for secure challenge -response pairs.


(Image Source: DALL-E 3)

A series of progressive studies have outlined the evolution of RO-PUF designs, leading to a configurable Ring-Oscillator design tailored for Xilinx FPGAs. This lab aims to transpose the theoretical constructs into a tangible hardware design.

III. Methodologies

The primary objective of this lab is to create a hands-on implementation of Ring-Oscillator based Physical Unclonable Functions (PUFs) on FPGAs, as per the insights garnered from the referenced papers. The lab is structured to closely follow the design elucidated by Xin et al. in their paper, steering towards the creation of a secure hardware framework. Below are the steps we took during this lab.

Initial Setup and LED Demonstration:

- We initiated the lab by implementing a configurable ring oscillator to drive a blinking LED on the Basys board.
- The task involved connecting the Ring Oscillator to the counter and max compare circuit, aiming for a blink rate of once per second.
- This step served as a preliminary test to ensure the correct functionality and understanding of the configurable ring-oscillator design.

PUF Design and Implementation:

- Building upon the initial setup, we proceeded to design the PUF as depicted in the referenced paper (Figure 7, Xin).
- Ensuring external visibility when the response is complete was a crucial part of this sub-task.
- This sub-task aimed at correlating theoretical knowledge with practical implementation, focusing on replicating the described PUF design with our unique modifications.

Comprehensive PUF on BASYS 3:

- Extending the PUF design, we implemented a whole PUF on the BASYS 3, following the described input/output scheme.
- We rigorously tested to ensure that a challenge Ci always yields the same response CRi across many separate calculations.
- This step aimed to verify the reliability and consistency of the PUF design.

SHA128 Algorithm Integration:

- With a working PUF, we integrated the provided SHA128 algorithm onto the board.
- The focus was to ensure the correct hashing of the challenge concatenated with the response.
- This sub-task intended to enhance the security infrastructure of our design.

IV. Questions

Question 1

What did you change about the provided configurable ring-oscillator?

The configurable ring oscillator described in the paper by Xin, Kaps, and Gaj has 8 challenge bits evenly split between sel and bx. In this lab, we are tasked to modify the design to work with 6 challenge bits. To accomplish this we kept the buffer block and tied the sel/bx to constant high, which meant we only needed 6 bits for 3 configurable ring oscillator blocks.

Question 2

How many ROs are necessary?

We need to output the response for our 6 challenge bits to 8 LEDs on the Basys board. For 8 LEDs, we need 8 response bits. From the paper, they describe how response bits are generated from the difference between counter values generated by adjacent pairs of ROs. This means for whatever amount of response bits we desire (n), we will need an additional RO (n+1) for 8 comparator operations.

Question 3

Why will a 50 MHz clock work for this design?

The 50 MHz clock is a sensible choice given the board's oscillator can handle up to 100 MHz. This ensures all components on the board, rated for this maximum speed, operate comfortably and reliably. The main goal is to have a steady clock to increment a counter, deciding when to halt the Ring Oscillator (RO) counting. The 50 MHz clock does the job well, providing a controlled pace for the PUF design to function accurately without pushing the components to their limits.

Question 4

Would any arbitrary clock speed work for this design?

In theory, any clock speed could be used, but it's crucial to consider the Nyquist rate, which states that the sampling rate must be at least twice the highest frequency component in the signal. This ensures that the Ring Oscillator (RO) frequencies are adequately sampled without aliasing. The clock speed primarily dictates how quickly the counter reaches its maximum value. As long as the counting duration allows for a sufficient number of RO counts to register—consistent with the Nyquist rate—the subsequent comparison of values harvested from all the ROs should remain effective.

However, practical considerations also come into play. We are constrained by the resources available and the inherent physics of our device. For instance, we operate under the assumption that the components within the device are rated up to 100 MHz, setting an upper limit on how high the counter max can be configured.

Question 5

What did you decide to use for the max value of std_counter?

We opted for a counter size of 5 million, which translates to an interval of approximately 50 milliseconds (ms) given a system clock speed of 100 MHz.

Question 6

How does this value impact the PUF?

This value affects the stability of the PUF. Initially, we experimented with a counter size of 50,000; however, this yielded an unstable challenge response output. Specifically, for a majority of the challenge bit settings, the response bits exhibited variations across every iteration. A larger counter size, while conducive to stabilizing the response, concurrently elongates the delay between the initiation of the challenge and the receipt of the response, a trade-off we found acceptable for achieving the desired level of stability in our system.

Question 7

Why would we want to know if the challenge response is complete?

Acknowledging the completion of the challenge response is essential for two main reasons. First, it allows for an immediate evaluation against set criteria to ascertain a pass or fail status. Second, it signifies the readiness to transition to subsequent tasks, enhancing workflow coordination. Moreover, with a substantial counter size, there could be notable delays before a response is generated. Knowing when the response process is complete is crucial to manage these delays effectively, ensuring the timely execution of other tasks within the system.

Question 8

What is the purpose of this (why might we want to hash the challenge concatenated with the response)?

This creates a unique fingerprint or identifier for that particular challenge-response pair, which can be used for verification purposes. It also enhances security by obfuscating the original data, making it difficult for malicious entities to reverse-engineer the challenge and response in terms of number of bits for each. Moreover, hashing ensures data integrity, ensuring that the data has not been tampered with during transit or storage. Lastly, it provides a fixed-size output, which can be beneficial for managing data storage and handling, especially in systems with limited resources.

<u>Question 9</u>

Are the challenge-response pairs the same for each location?

No, for a given configuration, the responses vary across different portions of the chip and across different members' Basys boards.

<u>Question 10</u>

Should they be (include details)?

The challenge-response pairs may vary across different locations on the FPGA due to the inherent physical discrepancies in the silicon. This aligns with the core characteristic of Physical Unclonable Functions (PUFs), where unique responses are generated based on the hardware's intrinsic properties. Therefore, we should not expect for the challenge response pairs to be the same for different locations on the chip.

<u>Question 11</u>

What are the intra-board and inter-board hamming distances?

A hamming distance is a measure of the difference between two binary strings. The distance value represents the amount of bits that are different between the two values. You can find the hamming distance by xoring two binary strings together, and counting the amount of 1's in the resulting string (Also in the resulting string you can see what bit positions differ between the two). The Xin paper explores two metrics for measuring the effectiveness of PUFs, inter-chip and intra-chip variation.

Intra-board Hamming Distance is a metric that quantifies the dissimilarity in the responses generated by a PUF when challenged within the same FPGA board, but under different conditions or locations. It evaluates the consistency of the PUF responses under varying internal conditions.

Inter-board Hamming Distance is a metric that quantifies the dissimilarity in the responses generated by a PUF when challenged across different FPGA boards. It evaluates the uniqueness of the PUF responses across different hardware instances. This can also be evaluated at different locations on the same chip. (In vivado, we would do it with the pblock function.)

We found our intra-board hamming distance to be 8.33% on average across the configurations and our inter-board hamming distance to be 50.39% on average across the configurations and locations.

<u>Question 12</u>

What are the ideal hamming distances?

Based on what we discussed previously, we want 50% for inter-chip hamming distance because that means half of our bits are different in our bit string when compared to other PUFs. For intra-chip hamming distance, we want 0% because that means we are able to reproduce the exact same output every single time.

<u>Question 13</u>

Discuss anything of note about your particular implementation or results.

PUF Challenge-Response FSM Diagram

1. **IDLE:** This is the initial state where the FSM waits to start the process. In this state, the upper 4 bits of the challenge (`index`) are reset to zero, and the `comparison_done` flag is also reset.
2. **START:** In this state, the FSM sets the challenge bits for the PUF (Physically Unclonable Function). The challenge is formed by concatenating the `index` with the lower 6 bits of the challenge (`challenge_lower_bits`).
3. **WAIT:** The FSM waits in this state until the PUF operation is complete. The completion is indicated by the `count_complete` signal.
4. **STORE:** Once the PUF operation is complete, the FSM moves to this state to store the output count (`ro_count_out`) into an array (`ro_count_array`) at the position indicated by `index`.
5. **INCREMENT:** In this state, the FSM increments the `index` by 1. This will be used to form the next challenge in the `START` state.
6. **COMPARE:** This is the final state where the FSM compares adjacent `ro_count_out` values stored in the array. The comparison results are stored in the `response` array. Once all comparisons are done, the `comparison_done` flag is set to indicate that the FSM can return to the `IDLE` state.

Each state transition is triggered by the rising edge of the clock signal (`CLK`). The state diagram is shown in the next page.

Other details will be discussed in the approach section below.

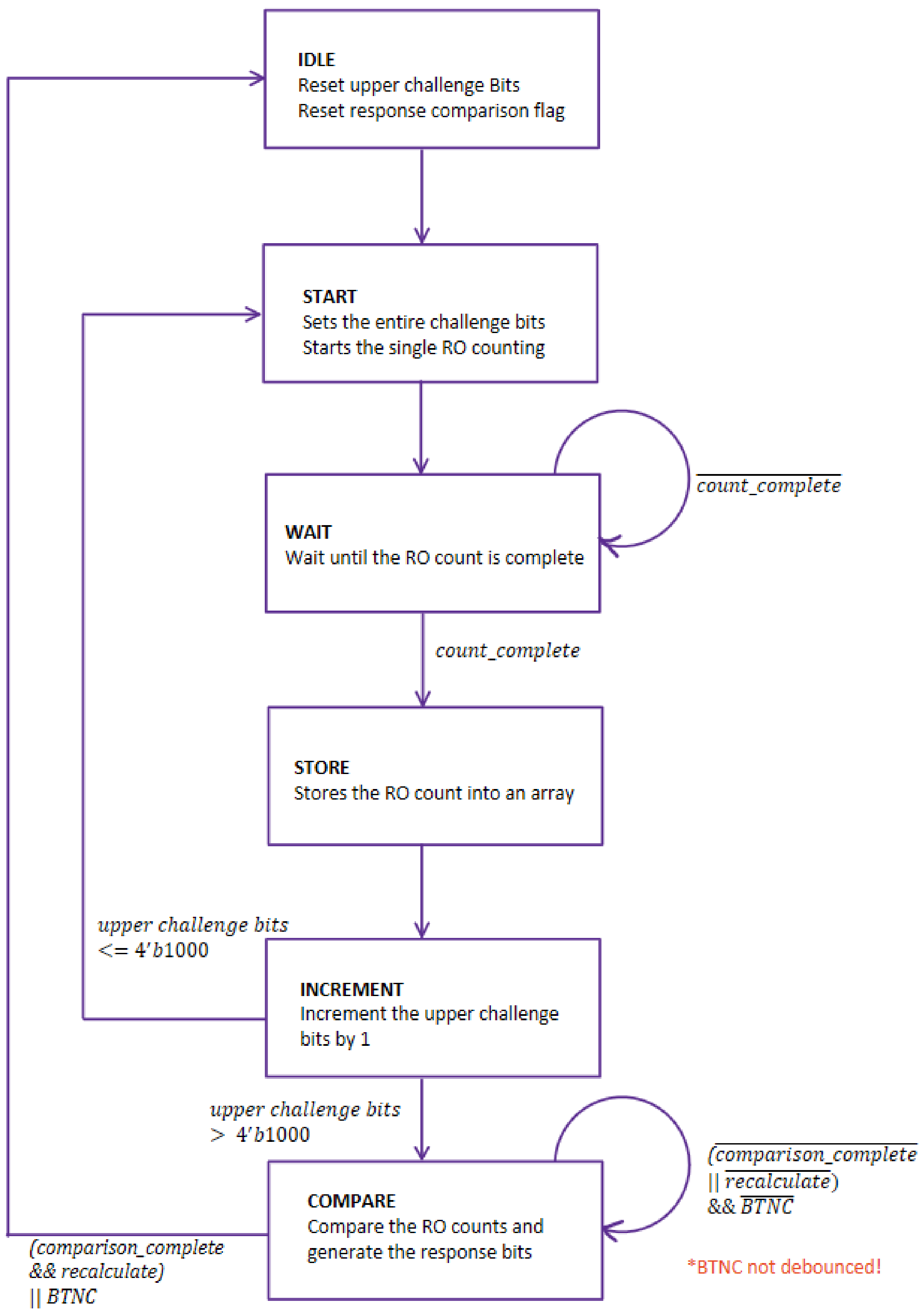

IDLE
Reset upper challenge Bits
Reset response comparison flag
START
Sets the entire challenge bits
Starts the single RO counting
WAIT
Wait until the RO count is complete
$\overline{count\_complete}$
$count\_complete$
STORE
Stores the RO count into an array
upper challenge bits <= 4'b1000
INCREMENT
Increment the upper challenge bits by 1
upper challenge bits > 4'b1000
COMPARE
Compare the RO counts and generate the response bits
$(\overline{comparison\_complete} \,||\, \overline{recalculate}) \,\&\&\, \overline{BTNC}$
*BTNC not debounced!
$(comparison\_complete \,\&\&\, recalculate) \,||\, BTNC$

Figure 4.13.1 Challenge-Response FSM Flowchart

Final Lab Questions

Question 1

How difficult would it be to expand the number of challenge bits and the bits of the response? Describe the process and any considerations needed to ensure the PUF functions correctly.

To increase the number of challenge bits, we can simply add more inverter slices in series with the current ring oscillator design. However, note we must keep the number of inverters ODD in the oscillating loop in order to sustain the oscillation.

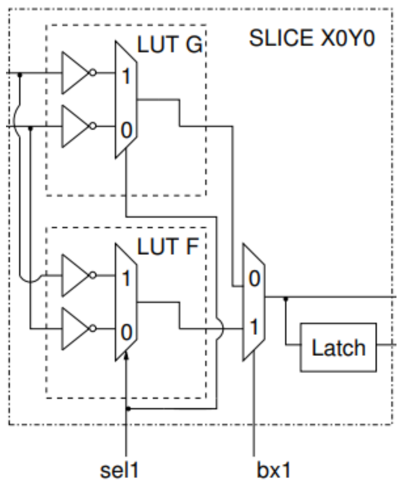


Figure 4.14.1 Single Slice Configuration

To increase the number of response bits, we must hook up more Ring oscillators in parallel in order to get more comparisons. To get n more bits, we need to add n more ring oscillators. This means we need to increase the number of selection bits for the RO Mux, which means we also need to add more challenge bits when the number of total RO reaches a log(#) threshold.

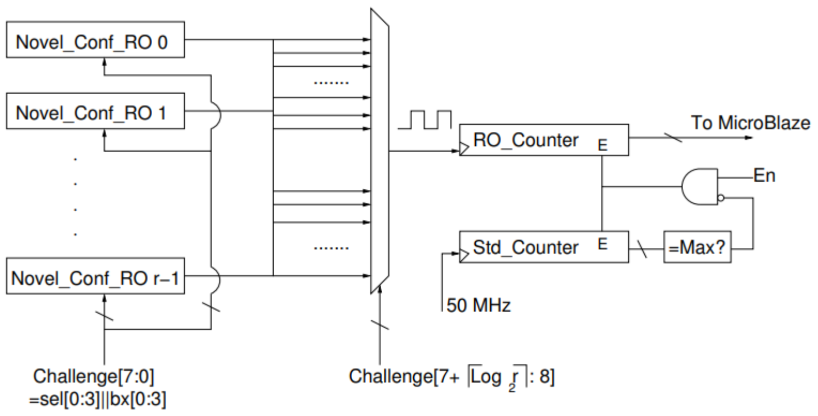


Figure 4.14.2 Configurable RO PUF

If more slices are added to each ring oscillator to increase the input challenge bits, it will increase the time for one complete oscillation to propagate the ring. There must be a proportional increase in the standard counter max to ensure the same number of loops complete. There must be a sufficient number of loops for consistent comparison between ring oscillator variance in counting speed.

Question 2

How might variations in temperature and voltage of the chip affect PUF? Think of both raw entropy and the result of the hash. What are some methods which could mitigate the effect?

Variations in temperature and voltage can impact a PUF by altering the silicon properties, affecting both the raw entropy and the resultant hash. These variations may lead to inconsistent identifiers or keys over time. Mitigation methods include maintaining a stable operating environment, employing error correction codes to handle minor variations in PUF responses, and applying cryptographic processing to raw PUF responses to reduce their sensitivity to environmental changes.

The first method to increase the reliability of the PUF for varying temperature and voltage is to increase the standard counter max. There will be larger count differences in RO counter values which will increase the reliability of the result. The drawback is that the time to output the PUF response will increase.

The second method is to attach the PUF to a thermally consistent area (an example would be a fan or water-cooled CPU) on an FPGA or IC via a heatsink or thermally semi-conductive. This will ensure consistent temperature values during regular operation of the device.

Question 3

You might have noticed in the paper and in our implementation the hamming distance between adjacent challenges is fairly minimal (15% in Xin et. Al). What might cause this? Can you think of a way of changing how we use the counter output to increase the hamming distance? Hint: This might mean reducing the number of usable response bits in the counter.

The observed minimal Hamming distance between adjacent challenges could be indicative of a certain level of dependency among the ring counters. As highlighted in the paper, a single bit alteration in sel[3:0] yields a higher average Hamming distance compared to a similar alteration in bx[3:0]. This is conjectured to be an outcome of the additional latch integrated in their design (Figure 6), as opposed to Maiti's design (Figure 5), making the signal more prone to the effects of manufacturing variations.

A potential strategy to augment the Hamming distance in response bits entails assessing the sensitivity of response bits relative to alterations in challenge bits. Essentially, this involves determining the propensity of a particular response bit to change as the challenge bits are varied. This sensitivity evaluation could be performed by documenting all challenge-response pairings and employing statistical analysis to discern patterns. Although this approach might curtail the number of usable response bits, it could facilitate a higher Hamming distance between adjacent challenges, thereby potentially enhancing the robustness and the unpredictability of the response bits.

Alternatively, we could try applying a non-linear transformation/hashing function to the counter output, where a small difference in the original value could result in a huge difference.

Test each device individually with random variations in p-block placement on the FPGA until the hamming distance is maximized. Certain p-block placements may have trace lengths conducive to biased results regardless of the select and challenge bit choice on the MUX.

Question 4

How could this implementation be modified on a board so that it could be verified from an external source and ensure freshness? (Don't think about the BASYS 3 board, instead consider the fundamental structure of the device, including generic inputs and outputs)

To enable verification from an external source and ensure freshness, you could implement a challenge-response protocol that incorporates a timestamp, nonce (Number Used Once[1]) or device identifier/address. The device then executes the PUF algorithm based on this challenge and generates a response. This response is combined with the timestamp, nonce or identifier, possibly through cryptographic hashing, and sent back to the external source. The external source verifies the combined data to confirm the response's freshness, ensuring it's not a replayed interaction.

[1] A **nonce**, which stands for "Number Used Once," is a unique value that is generated for each transaction or session. It is used to ensure that old communications cannot be reused in replay attacks. In the context of the challenge-response protocol, the nonce serves as a one-time code that is sent along with the challenge from the external source.

IV. Approach

One of the pronounced challenges faced in this lab was the incapability to simulate the Ring Oscillator (RO) in software. However, a silver lining to this was that the RO was the only component that posed such a simulation challenge. To work around this, one of the strategies we employed was to convert the ring oscillator module into a simple clock. This alteration proved instrumental as it enabled the simulation and testing of higher-level modules, making a significant stride towards our lab objectives. Yet, this adaptation didn't spell the end of our challenges, as there were other significant hurdles that we had to navigate through.

Transparent Buffer

A significant challenge we encountered was the verification of the transparent buffer's existence, depicted as the Latch component in the following figure.

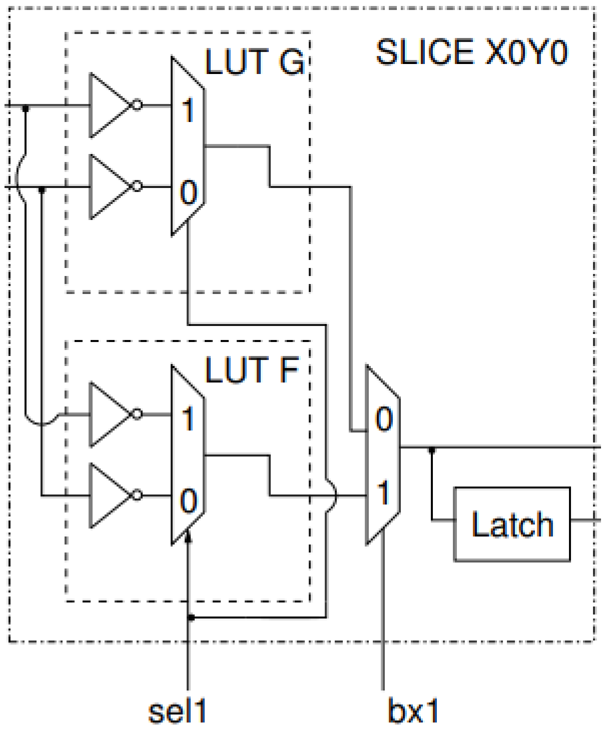


Figure 5.1.1 Single Slice Configuration

At the outset, the straightforward approach seemed to be the utilization of an always_ff block, updating the value with each clock cycle. However, the idea of extending a clock signal solely for this purpose didn't resonate with us, propelling us towards exploring the always_latch block instead.

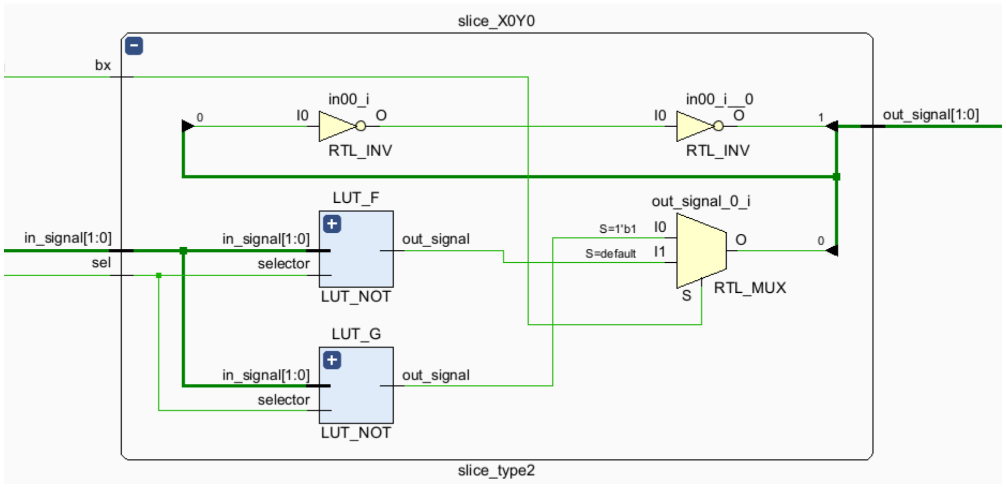


Figure 5.1.2 Vivado RTL Analysis - Elaborated Design

Our venture into Vivado's RTL Analysis - Elaborated Design, which captures the underlying logic but not the timing, presented a hurdle as we couldn't get any transparent buffer to manifest, barring the use of two consecutive inverters. Upon reflection, the circuit's behavior exhibited a stark resemblance whether we employed two inverters or opted for an always_latch block, suggesting the original latch was likely implemented as intended. Nevertheless, having it visually represented on the schematic instilled a higher degree of confidence in our understanding and the implementation.

RO PUF Module

The task of implementing the RO PUF Challenge Response module was fairly straightforward, albeit with a minor stumbling block encountered along the way. As depicted in Figure 4.2.1 below, the configuration involves all Ring Oscillators (ROs) connected in parallel, with the precise selection of the one to be counted being determined by the upper challenge bits.

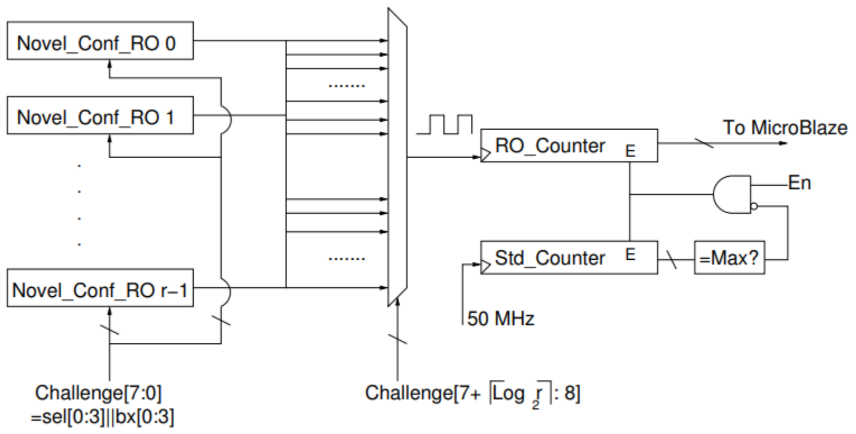
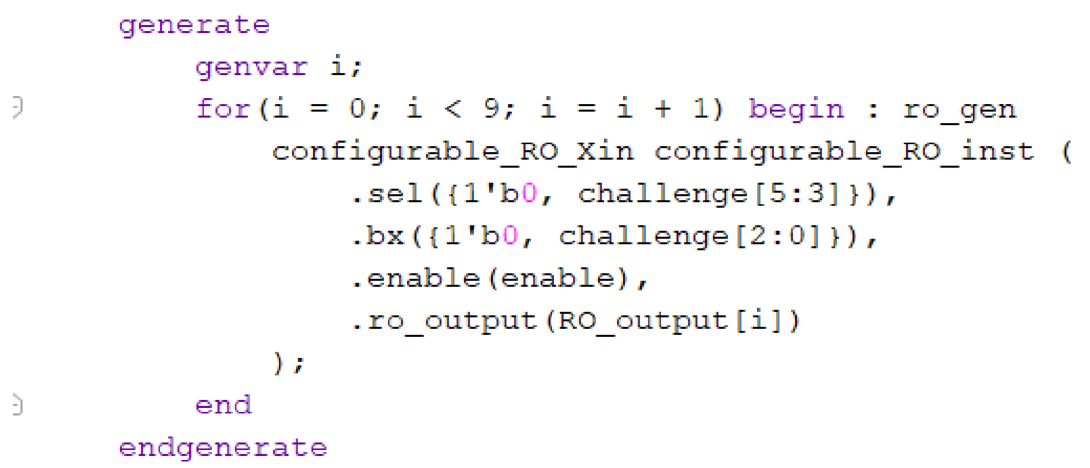


Figure 5.2.1 Configurable RO PUF

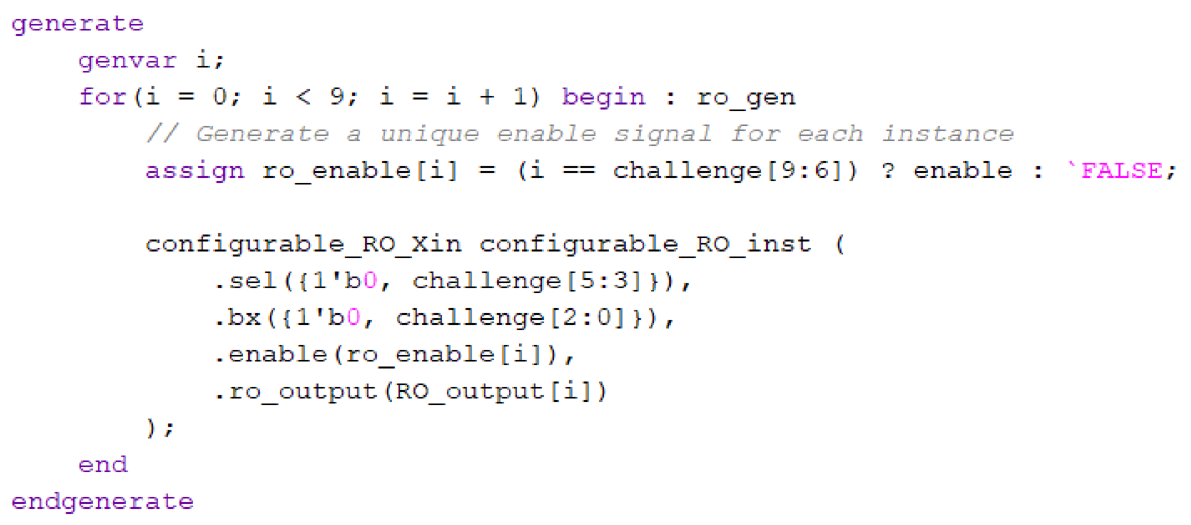
In a bid to streamline the declaration of 9 ROs, we leveraged the iterative generate block, enabling the simultaneous declaration of all 9 instances, as opposed to the cumbersome process of copy-pasting. This approach, illustrated in Figure 4.2.2a, facilitated a seamless pass down of the enable signal from the upper module, and with the code clearing all our tests, it was presumed to be functioning as intended.



```
generate
    genvar i;
    for(i = 0; i < 9; i = i + 1) begin : ro_gen
        configurable_RO_Xin configurable_RO_inst (
            .sel({1'b0, challenge[5:3]}),
            .bx({1'b0, challenge[2:0]}),
            .enable(enable),
            .ro_output(RO_output[i])
        );
    end
endgenerate
```

Figure 5.2.2a SystemVerilog code for RO instance generation (Incorrect)

However, a subtlety that eluded us initially was that, with the above implementation, all the ROs would be activated concurrently when enabled. This concurrency posed a risk where the oscillation from one could potentially influence the outcomes of the others. Although this issue came to light during our endeavor to stabilize the challenge-response bits, the extent of its impact remains uncertain. Nevertheless, we rectified the situation by integrating additional logic, ensuring that only the RO aligned with the mux selection would be enabled, as showcased in Figure 4.2.2b.



```
generate
    genvar i;
    for(i = 0; i < 9; i = i + 1) begin : ro_gen
        // Generate a unique enable signal for each instance
        assign ro_enable[i] = (i == challenge[9:6]) ? enable : `FALSE;

        configurable_RO_Xin configurable_RO_inst (
            .sel({1'b0, challenge[5:3]}),
            .bx({1'b0, challenge[2:0]}),
            .enable(ro_enable[i]),
            .ro_output(RO_output[i])
        );
    end
endgenerate
```

Figure 5.2.2b SystemVerilog code for RO instance generation

Challenge Response FSM

The development of the Challenge Response state machine was straightforward, particularly facilitated by our ability to simulate it in software. The intricacies emerged when transitioning from simulation to hardware. Despite the flawless operation in the simulation environment, the state machine ceased to function as anticipated upon being deployed to the board.

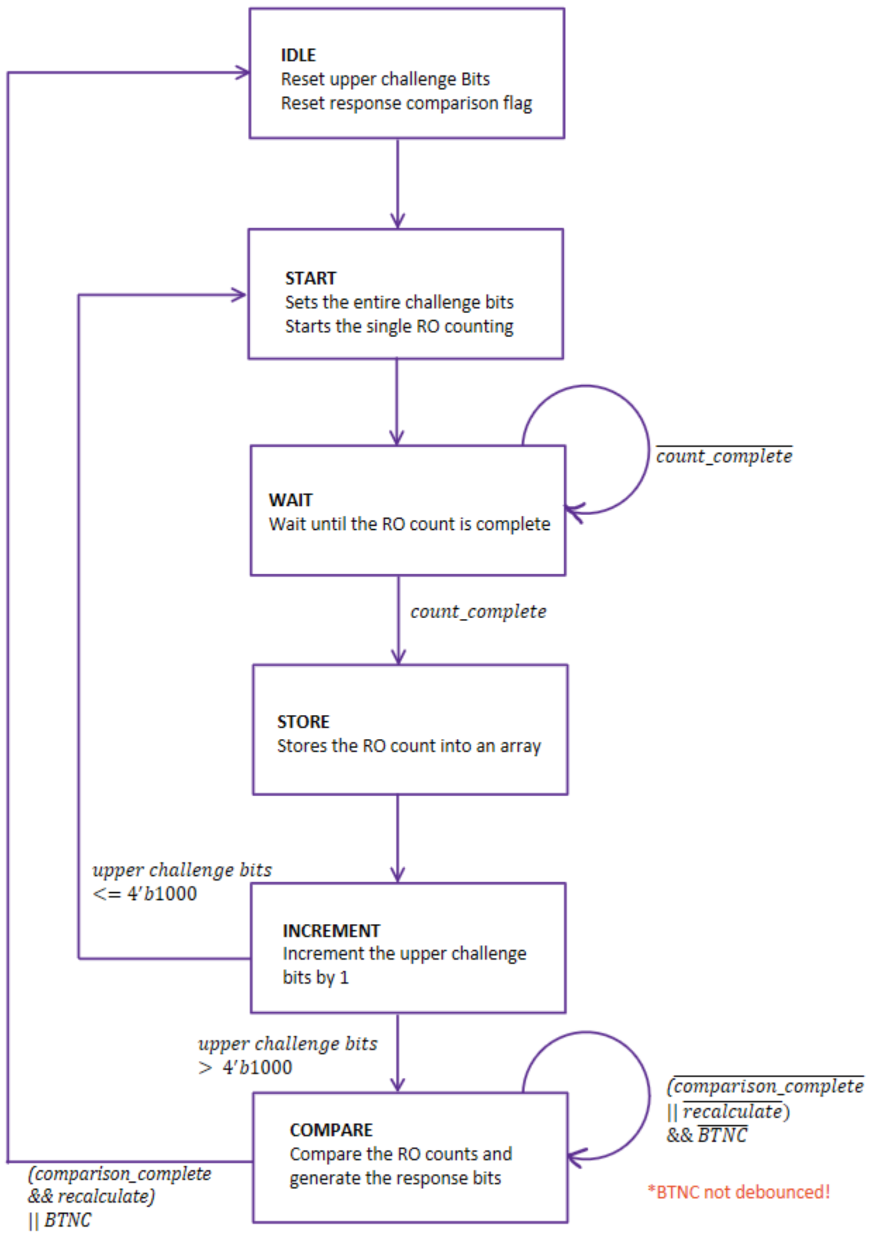

IDLE
Reset upper challenge Bits
Reset response comparison flag
START
Sets the entire challenge bits
Starts the single RO counting
WAIT
Wait until the RO count is complete
$\overline{count\_complete}$
$count\_complete$
STORE
Stores the RO count into an array
INCREMENT
Increment the upper challenge bits by 1
upper challenge bits <= 4'b1000
upper challenge bits > 4'b1000
COMPARE
Compare the RO counts and generate the response bits
$(\overline{comparison\_complete} \,||\, \overline{recalculate}) \,\&\&\, \overline{BTNC}$
*BTNC not debounced!
$(comparison\_complete \,\&\&\, recalculate) \,||\, BTNC$

Figure 5.3.1 Challenge-Response FSM Flowchart

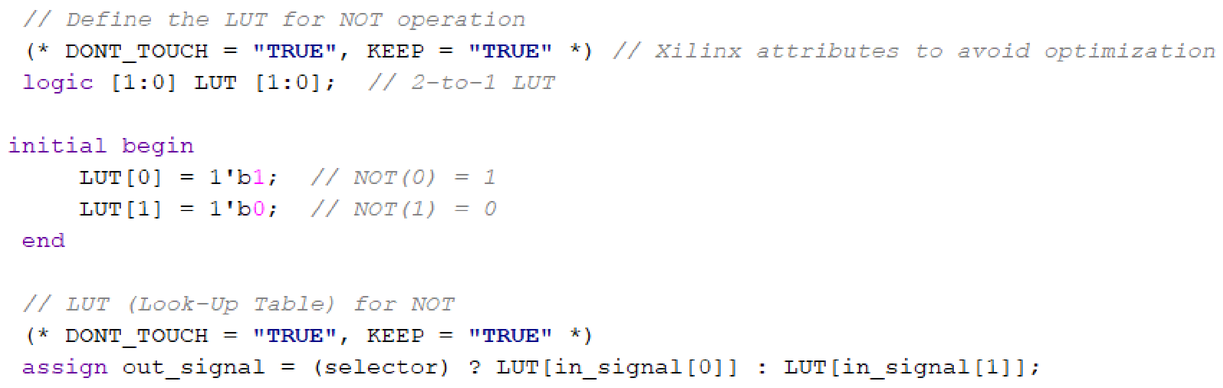
Our subsequent investigation revealed that the malfunction was rooted in our oversight to declare the KEEP and S attributes, pivotal for the FSM's correct operation on the hardware. In

SystemVerilog, the KEEP and S attributes are utilized to instruct the synthesis tool to preserve certain elements in the design during the optimization process. Specifically, the KEEP attribute prevents the optimizer from removing specified nets or cells, ensuring they remain in the synthesized netlist. On the other hand, the S attribute is used to control the synthesis elaboration, helping maintain the structural integrity of the design as intended.

The absence of these crucial attributes led to the synthesis tool possibly optimizing away vital parts of our FSM logic, consequently disrupting its functionality on the hardware. Although this nuance diverted a considerable chunk of our time towards debugging in the simulation environment, it served a silver lining. The rigorous debugging not only aided us in identifying the oversight but also fortified our confidence in the integrity of the FSM. It provided a solid reassurance that the state machine was impeccably designed and functional, streamlining the troubleshooting process in the later stages.

KEEP and S attributes

To preserve the integrity of our ring oscillator structure within Vivado and prevent the software from eliminating or optimizing it during the synthesis phase, we employed the KEEP and DONT_TOUCH attributes on certain signal traces, as depicted in Figure 4.3.1.

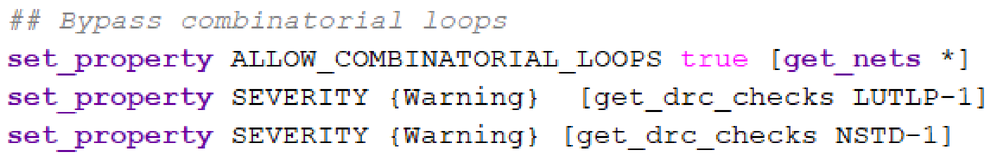


```
 // Define the LUT for NOT operation
 (* DONT_TOUCH = "TRUE", KEEP = "TRUE" *) // Xilinx attributes to avoid optimization
 logic [1:0] LUT [1:0];  // 2-to-1 LUT

initial begin
     LUT[0] = 1'b1;  // NOT(0) = 1
     LUT[1] = 1'b0;  // NOT(1) = 0
 end

 // LUT (Look-Up Table) for NOT
 (* DONT_TOUCH = "TRUE", KEEP = "TRUE" *)
 assign out_signal = (selector) ? LUT[in_signal[0]] : LUT[in_signal[1]];
```

Figure 5.4.1 DONT_TOUCH and KEEP attribute code

Nevertheless, given the inherent looped nature of our Ring Oscillator (RO), Vivado flagged a "Combinatorial Circuit detected" error, hindering the synthesis and implementation of the circuit. Upon delving into some research, we unearthed an attribute, ALLOW _COMBINATORIAL_LOOPS = “TRUE”, which seemed to provide a pathway around this hurdle.

Although this adjustment facilitated the implementation of the design, Vivado persisted in issuing the ERROR concerning the combinatorial loop during the 'Generate Bit Stream' stage, halting further progress. The resolution came after several hours of meticulous internet searches, leading us to a post[2] that provided a snippet of code to override all combinatorial loop checks, among other design checks, as shown in Figure 4.3.2.



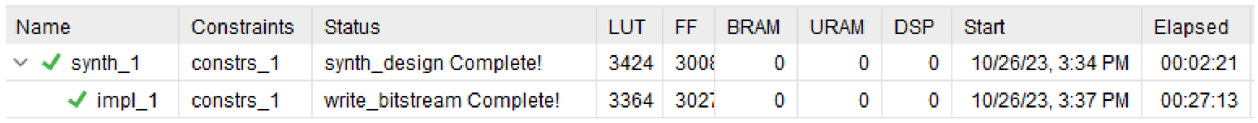
```
## Bypass combinatorial loops
set_property ALLOW_COMBINATORIAL_LOOPS true [get_nets *]
set_property SEVERITY {Warning}  [get_drc_checks LUTLP-1]
set_property SEVERITY {Warning} [get_drc_checks NSTD-1]
```

Figure 5.4.2 Combinatorial Loop Bypass code

We deduced that this code snippet effectively diminishes the warning levels of ALL Design Rule Checks (DRC) by one tier; transforming errors into critical warnings, and critical warnings into mere warnings. While this workaround enabled the generation of the bit stream file within Vivado, we recognize that it is likely not the optimal solution. The downgrading of warning levels could mask other significant issues, potentially spawning emergent problems within the design. However, under the circumstances, it was the only avenue available to us for advancing past the error and achieving the bit stream generation.

SHA and Switch Position Decoding

The incorporation of the provided SHA algorithm into our project was seamless and operated flawlessly upon its initial integration. However, it brought along a substantial increase in the synthesis and implementation time which was not a trivial matter, especially given the iterative nature of the development and debugging process. As illustrated in Figure 4.5.1, the total time surged to over 30 minutes when executed on a relatively outdated laptop, contrasting starkly with the customary total time of about 9 minutes sans the SHA module.

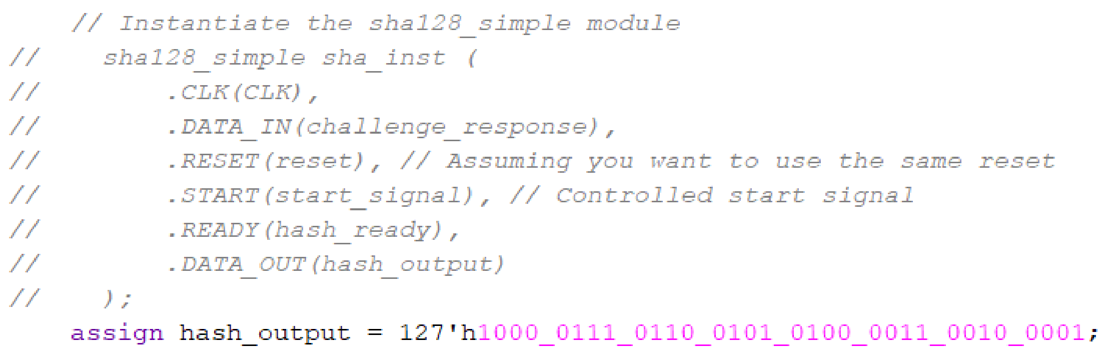


Name	Constraints	Status	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
✓ synth_1	constrs_1	synth_design Complete!	3424	3008	0	0	0	10/26/23, 3:34 PM	00:02:21
✓ impl_1	constrs_1	write_bitstream Complete!	3364	3027	0	0	0	10/26/23, 3:37 PM	00:27:13

Figure 5.5.1 Synthesis and Implementation Time with SHA

Given that the focal point of our project did not revolve around the SHA module, it was imperative to devise a strategy that could ease the debugging process without being encumbered by the extended synthesis and implementation times. Our objective was to maintain a streamlined debugging workflow while temporarily sidelining the SHA module.

[2] Source of the Figure 4.4.2 code: https://support.xilinx.com/s/question/0D52E00006hpdq2SAA/how-to-apply-setproperty-allowcombinatorialloops-true-to-pass-combinatorial-loops?language=en_US



```
    // Instantiate the sha128_simple module
//    sha128_simple sha_inst (
//        .CLK(CLK),
//        .DATA_IN(challenge_response),
//        .RESET(reset), // Assuming you want to use the same reset
//        .START(start_signal), // Controlled start signal
//        .READY(hash_ready),
//        .DATA_OUT(hash_output)
//    );
    assign hash_output = 127'h1000_0111_0110_0101_0100_0011_0010_0001;
```

Figure 5.5.2 Testing without the SHA module

Instead of using the SHA module's hash_output directly, which would have given us random numbers that are hard to understand, we created a custom 128-bit number. (The number in Figure 4.5.2 is in HEX, not BINARY.) After running the switch decoding logic, the selected bits from this number matched the switch positions, confirming that our switch decoding logic was working correctly.

V. PUF Analysis

To evaluate the performance of our PUF, we computed the Intra and Inter Hamming distances. For the Intra-PUF data, we collected three trials of a given configuration, calculated the cumulative hamming distance and then turned it into a ratio by dividing out the number of trials and bit-length (8 bits for the response). This left us with a percentage that tells us how much of the bits of a given configuration change. The data gathered and calculations can be found in the Appendix. The summary table below shows how our PUFs are reliable and repeatable.

Table 6.1 Intra-PUF Hamming Distance Summary

Intra-PUF Summary	
Min	0.00
Max	58.33
Median	0.00
Mean	8.33

Continuing this analysis, we conduct a similar process for the Inter-PUF Hamming distance. We determine this by observing the response bits of our PUF module when placed on different parts of our FPGA using PBlocks to select the location they will be placed and routed. The default and two selected PBlock locations on the FPGA can be seen in the figures below. As before, data and calculations can be found in the Appendix. Here we also found that the PUF performed satisfactory with its hamming bits on average having a ~50% variance. This shows how the PUF generates unique response values for different areas of the chip.

Table 6.2 Inter-PUF Hamming Distance Summary

Inter-PUF Summary	
Min	25.00
Max	66.67
Median	50.00
Mean	50.39

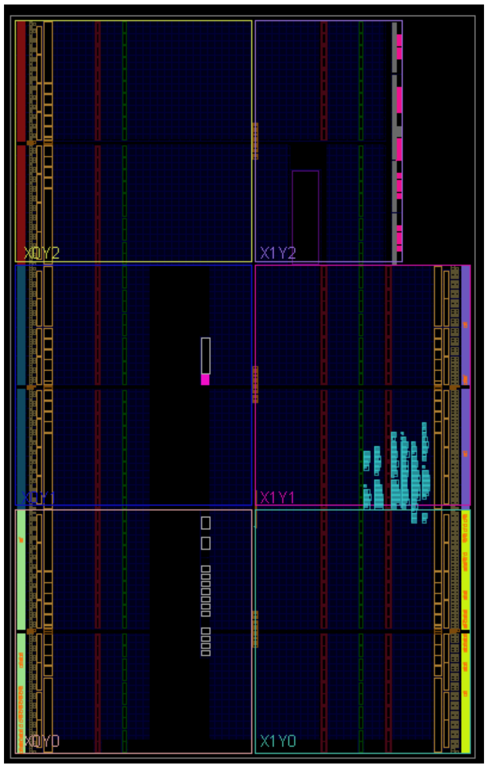


Figure 6.1 PBlock 0 - Default Configuration and Placement

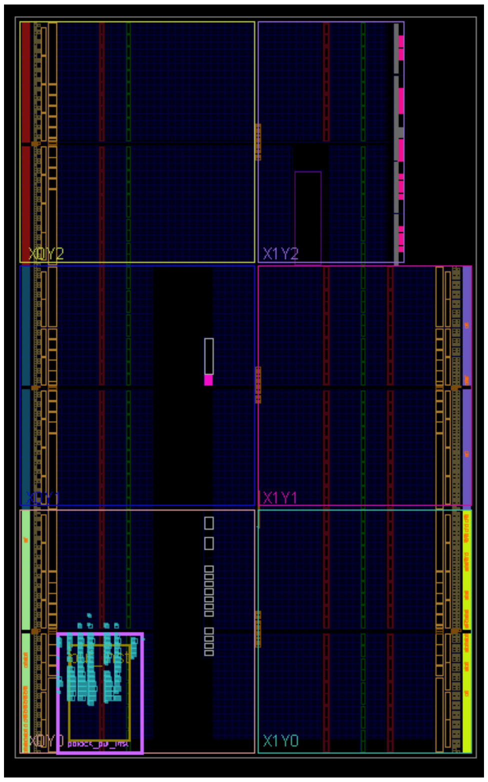


Figure 6.2 PBlock 1 - First Selected Placement

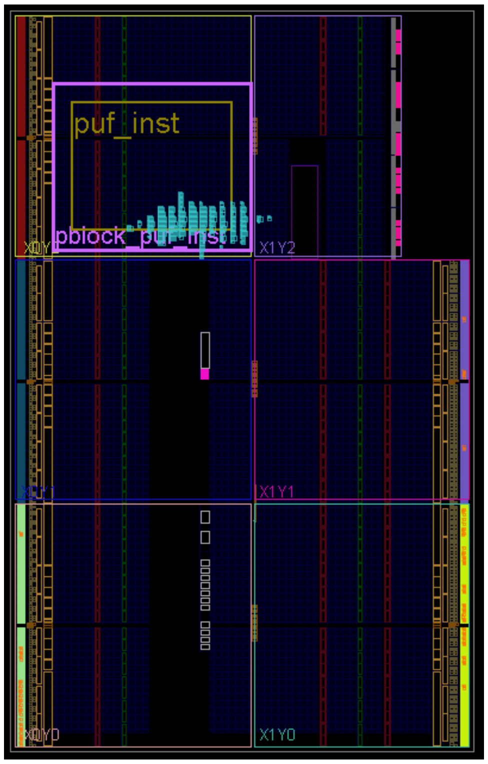


Figure 6.3 PBlock 2 - Second Selected Placement

VI. Appendix

Table 7.1 Intra-PUF Data and Hamming Distance Calculation

Intra-PUF	Button Reset Trials			Cumulative Hamming Distance	Variation (%)
{sel [5:3], bx [2:0]}	Trial 1 (Hex)	Trial 2 (Hex)	Trial 3 (Hex)	xor(1,2)+xor(1,3)+xor(2,3)	(HDist/Trials)/Bit-length
000000	4C	4C	4C	0	0.00
000001	55	56	57	4	16.67
000010	4d	4d	5d	2	8.33
000011	55	56	57	4	16.67
000100	4E	4E	4E	0	0.00
000101	65	66	67	4	16.67
000110	ab	ab	ab	0	0.00
000111	a9	b1	b2	8	33.33
001000	4c	9b	bb	14	58.33
001001	b4	b5	b6	4	16.67
001010	b4	b2	b3	6	25.00
001011	aa	aa	aa	0	0.00
001100	52	53	54	6	25.00
001101	1a	1a	1a	0	0.00
001110	57	57	57	0	0.00
001111	3a	3a	3a	0	0.00
010000	cc	e7	e8	10	41.67
010001	b5	b5	b5	0	0.00
010010	aa	aa	aa	0	0.00
010011	aa	aa	aa	0	0.00
010100	b3	b3	b3	0	0.00
010101	ba	ba	ba	0	0.00
010110	3b	3b	3b	0	0.00
010111	2a	2a	2a	0	0.00
011000	ba	ba	ba	0	0.00
011001	aa	aa	aa	0	0.00
011010	ba	ba	ba	0	0.00
011011	2a	2a	2a	0	0.00
011100	12	13	1b	4	16.67

011101	1a	1a	1a	0	0.00
011110	6b	6b	6b	0	0.00
011111	12	12	12	0	0.00
100000	56	56	56	0	0.00
100001	54	54	54	0	0.00
100010	56	5d	5d	6	25.00
100011	55	55	55	0	0.00
100100	2e	29	29	6	25.00
100101	35	35	35	0	0.00
100110	0d	6d	49	6	25.00
100111	1d	1d	1d	0	0.00
101000	d5	b8	b9	10	41.67
101001	ab	a9	ab	2	8.33
101010	a1	a6	a6	6	25.00
101011	a9	a8	a8	2	8.33
101100	b5	b1	b1	2	8.33
101101	aa	a6	aa	4	16.67
101110	b5	b5	b5	0	0.00
101111	a6	a2	a6	2	8.33
110000	ab	ab	ab	0	0.00
110001	ad	af	ad	2	8.33
110010	aa	aa	aa	0	0.00
110011	aa	aa	aa	0	0.00
110100	a3	a3	a3	0	0.00
110101	ab	ab	ab	0	0.00
110110	a2	a2	a3	2	8.33
110111	ae	aa	aa	2	8.33
111000	ab	ab	ab	0	0.00
111001	a9	a9	a9	0	0.00
111010	aa	aa	aa	0	0.00
111011	aa	2a	aa	2	8.33
111100	b3	b3	b3	0	0.00
111101	37	b6	a6	6	25.00
111110	b6	36	b6	2	8.33
111111	3e	3e	3e	0	0.00

Table 7.2 Inter-PUF Data and Hamming Distance Calculation

Inter-PUF	Board Locations			Cumulative Hamming Distance	Variation (%)
{sel [5:3], bx [2:0]}	Location 1 (Hex)	Location 2 (Hex)	Location 3 (Hex)	xor(1,2)+xor(1,3)+xor(2,3)	(HDist/Trials)/Bit-length
000000	4d	a8	4a	12	50.00
000001	55	28	62	14	58.33
000010	4d	19	4a	10	41.67
000011	55	39	62	14	58.33
000100	6e	94	4a	14	58.33
000101	25	3c	6a	12	50.00
000110	ab	94	ab	12	50.00
000111	29	1c	c9	12	50.00
001000	9b	15	26	14	58.33
001001	b6	b9	57	14	58.33
001010	b2	55	b5	12	50.00
001011	aa	d5	25	16	66.67
001100	53	4e	b3	14	58.33
001101	1a	cb	29	12	50.00
001110	57	56	33	8	33.33
001111	3a	cd	29	14	58.33
010000	e3	4a	ca	8	33.33
010001	b7	1c	d2	14	58.33
010010	aa	15	d1	16	66.67
010011	aa	d7	19	16	66.67
010100	bb	15	95	10	41.67
010101	ba	d5	b5	12	50.00
010110	3a	15	55	12	50.00
010111	2a	55	15	14	58.33
011000	ba	b5	ad	10	41.67
011001	aa	d4	a9	14	58.33
011010	ba	35	19	12	50.00
011011	2a	57	19	14	58.33
011100	13	35	a7	10	41.67
011101	1a	55	2b	14	58.33

011110	6b	4d	33	12	50.00
011111	12	95	1b	10	41.67
100000	54	d4	47	8	33.33
100001	55	74	66	8	33.33
100010	5d	69	53	10	41.67
100011	55	79	52	10	41.67
100100	29	d4	4e	16	66.67
100101	35	58	4e	14	58.33
100110	2b	19	5a	10	41.67
100111	3d	39	52	12	50.00
101000	b9	0b	b6	14	58.33
101001	aa	8a	76	12	50.00
101010	a6	d5	a6	10	41.67
101011	aa	d9	06	16	66.67
101100	b1	4a	23	14	58.33
101101	aa	8a	a3	6	25.00
101110	b6	d4	28	14	58.33
101111	a6	d6	ad	12	50.00
110000	ab	14	e3	16	66.67
110001	ab	12	b6	12	50.00
110010	aa	95	9a	12	50.00
110011	aa	97	b8	12	50.00
110100	a3	95	ec	14	58.33
110101	ab	97	31	12	50.00
110110	a2	95	75	14	58.33
110111	aa	97	3d	14	58.33
111000	ab	b9	aa	6	25.00
111001	a9	bb	aa	6	25.00
111010	aa	3d	2a	10	41.67
111011	aa	9b	28	10	41.67
111100	b3	a5	2a	12	50.00
111101	2b	a6	2b	8	33.33
111110	b6	34	2b	12	50.00
111111	6e	d4	2b	16	66.67