UNIVERSITY OF SALERNO

Department of Computer Science

Bachelor's degree in Computer Science

THESIS

Arbiter PUF implementation on FPGA

SUPERVISOR
Prof. Christiancarmine Esposito
University of Salerno

CANDIDATE
Salvatore Ruocco
Matr.: 0512115080

Academic Year 2023-2024

Persistence,

nothing in the world can take the place of persistence.

Abstract

The security of services, confidential data, and intellectual property is at risk from physical attacks, which often involve reading or tampering with data. Attackers can frequently access tools and equipment capable of reading or corrupting memory, either through invasive or non-invasive techniques. Cryptographic algorithms typically store their secret keys in memory, making them a potential target. Physical Unclonable Functions (PUFs) offer a promising solution to these vulnerabilities, as they generate keys only when needed, eliminating the need for storage on a powered-off chip. PUFs leverage inherent manufacturing variations to produce chip-specific output sequences (responses) in reaction to specific queries (challenges). These variations are random, unique to each device, difficult to replicate—even by the same manufacturer with identical processes, equipment, and settings—and are expected to remain static, making PUFs ideal for cryptographic key generation.

This thesis focuses on a delay-based PUF known as the Arbiter PUF, which exploits intrinsic differences in the propagation delay of two symmetrical paths. The research involves implementing the PUF on an Nexys A7-100 FPGA using Verilog for the design coding.

Experimental results were analyzed based on three main criteria: uniformity, stability, and randomness.

Keywords: Arbiter PUF, FPGA, HDL, Verilog

Abstract

La sicurezza dei servizi, dei dati riservati e della proprietà intellettuale è messa a rischio da attacchi fisici, che spesso comportano la lettura o la manomissione dei dati. Gli aggressori possono spesso accedere a strumenti e apparecchiature in grado di leggere o corrompere la memoria, con tecniche invasive o non. Gli algoritmi crittografici tipicamente memorizzano le loro chiavi segrete in memoria, il che li rende un bersaglio vulnerabile. Le Physical Unclonable Functions (PUFs) offrono una soluzione promettente a queste vulnerabilità, poiché generano le chiavi solo quando sono necessarie, eliminando la necessità di memorizzarle su un dispositivo. Le PUF sfruttano le variazioni hardware intrinseche per produrre sequenze di output (responses) specifiche per il chip in risposta a input specifici (challenges). Queste variazioni sono casuali, uniche per ogni dispositivo, difficilmente replicabili - anche dallo stesso produttore con processi, attrezzature e impostazioni identiche - e si prevede che rimangano statiche, rendendo le PUF ideali per la generazione di chiavi crittografiche.

Questa tesi si concentra su una PUF basata sul ritardo, nota come Arbiter PUF, che sfrutta le differenze intrinseche nel ritardo di propagazione di due percorsi simmetrici. La ricerca prevede l'implementazione della PUF su un FPGA Nexys A7-100 utilizzando Verilog per l'implementazione del design dell'hardware hardware.

I risultati sono stati analizzati in base a tree metriche: uniformità, stabilità e casualità.

Parole chiave: Arbiter PUF, FPGA, HDL, Verilog

Contents

List of Figures

List of Tables

List of Source Codes

CHAPTER 1

Introduction

1.1 Motivation

In recent years, the rapid evolution of technology has underscored the need for robust security solutions in hardware design, particularly in fields like IoT, telecommunications, and embedded systems. Driven by a curiosity to deepen my understanding of secure hardware design, I chose to focus on implementing an Arbiter PUF on an FPGA for my thesis project. This endeavor not only allows me to investigate the fascinating principles of PUFs but also enables me to engage in hands-on experimentation with FPGA technology, thus bridging theoretical concepts with practical applications. The Arbiter PUF, in particular, presents a unique opportunity to explore a secure identification method that leverages the natural variations in silicon manufacturing, making it an ideal starting point for anyone interested in hardware-based security.

1.2 Thesis objectives

The goal of this bachelor's thesis project is to implement and evaluate an arbiter PUF in terms of uniformity, stability, and randomness.

1.3 Thesis structure

Chapter 2 gives a general explanation of the various topics discussed in this thesis: Hardware security, PUF, FPGA, and Verilog.
Chapter 3 talks about the tools used for this work and how the translation from Verilog to electrical circuit occurs.
Chapter 4 shows the detailed implementation of the arbiter PUF.
Chapter 5 discusses how the results were extrapolated and explains the evaluation metrics used.
Chapter 6 shows the analysis of the obtained results and compares them with other similar works.

CHAPTER 2

Background

2.1 Hardware Hacking

Hardware hacking refers to exploiting vulnerabilities in physical devices to assess and improve their security. This specialized area of security testing focuses on the physical components of a system, including embedded systems, IoT devices, and industrial control systems.
Key activities in hardware hacking include:

- **Reverse Engineering**
  Analyzing the hardware design and firmware (the software that runs on the hardware) to understand how the system works, identify weak points, and potentially uncover vulnerabilities such as undocumented features or insecure communication protocols.
- **Exploiting debug interfaces (JTAG, UART, SPI)**
  Many devices have debug ports that can be used to extract sensitive information, alter firmware, or gain unauthorized access. Pentesters often use these interfaces to interact with the hardware at a low level.
- **Firmware analysis**

Extracting and analyzing the firmware for vulnerabilities, such as hardcoded credentials, backdoors, or outdated libraries. This might involve dumping the firmware via debug interfaces or directly from flash memory.

- **Side-Channel Attacks**
  These involve extracting information by analyzing unintended emissions from device, such as power consumption, electromagnetic leaks, or even acoustic emissions. A famous example is power analysis, where power usage patterns can reveal encryption keys.
- And many other activities...

2.2 PUF - Phisical Unclonable Function

The concept of PUFs is vital in the fields of hardware security and cryptography due to their ability to generate unique, device-specific identifiers that are nearly impossible to clone or replicate. Unlike traditional methods that rely on stored keys or software-based algorithms, PUFs harness the inherent physical variations present in each hardware component variations introduced naturally during the manufacturing process. This unique fingerprint can serve as a robust foundation for security, offering advantages in both device authentication and secure key generation.
In hardware security, PUFs are particularly valuable because they do not store sensitive data in memory; instead, they derive it directly from the physical characteristics of the hardware. This makes PUFs resilient to certain types of physical and side-channel attacks, which are increasingly concerning in the context of embedded systems and IoT devices. Since each PUF is inherently unique, even two devices built from the same design will produce different outputs, making counterfeiting and unauthorized cloning extremely difficult.
In cryptography, PUFs serve as a source of high-quality randomness and as a means for generating cryptographic keys that are difficult to reproduce. This is essential for secure key management and can be used to enhance protocols for secure communication, data protection, and identity verification. The input to a PUF is called **challenge** and the output is called **response**. A challenge issued and its response is

called challenge-response pair (**CRP**), and the relationship between the challenge and response of a particular PUF is called CRP behavior.

2.2.1 Main types of PUF

There are some types of PUF, we list only the electronic PUFs (This article [1] describes many types of PUFs, even those that were not created to build a PUF, but have properties that make them considered PUFs.) which are divide in:

- Delay-based PUFs
  - Arbiter PUF
  - Ring oscillator PUF
- Memory-based PUFs
  - SRAM PUF
  - Butterfly PUF
  - Latch PUF
  - Flip-flop PUF

Arbiter PUF utilize the timing differences in signal propagation through two parallel delay paths. A challenge determines the path configuration, and the arbiter outputs a 0 or 1 based on which signal arrives first. The sensitivity to manufacturing variations makes each Arbiter PUF unique. While efficient and compact, they can be vulnerable to modeling attacks if not carefully designed. Enhancements, like XORing multiple responses, improve security by increasing resistance to such attacks.

Ring oscillator PUF [2] exploit frequency differences in ring oscillators, which are small circuits that generate oscillating signals. Each oscillator's frequency depends on process variations, creating unique signatures. Challenges often involve selecting specific pairs of oscillators for comparison. RO PUFs are popular due to their simplicity and robustness against environmental variations, but they typically consume more power and area compared to other PUFs.

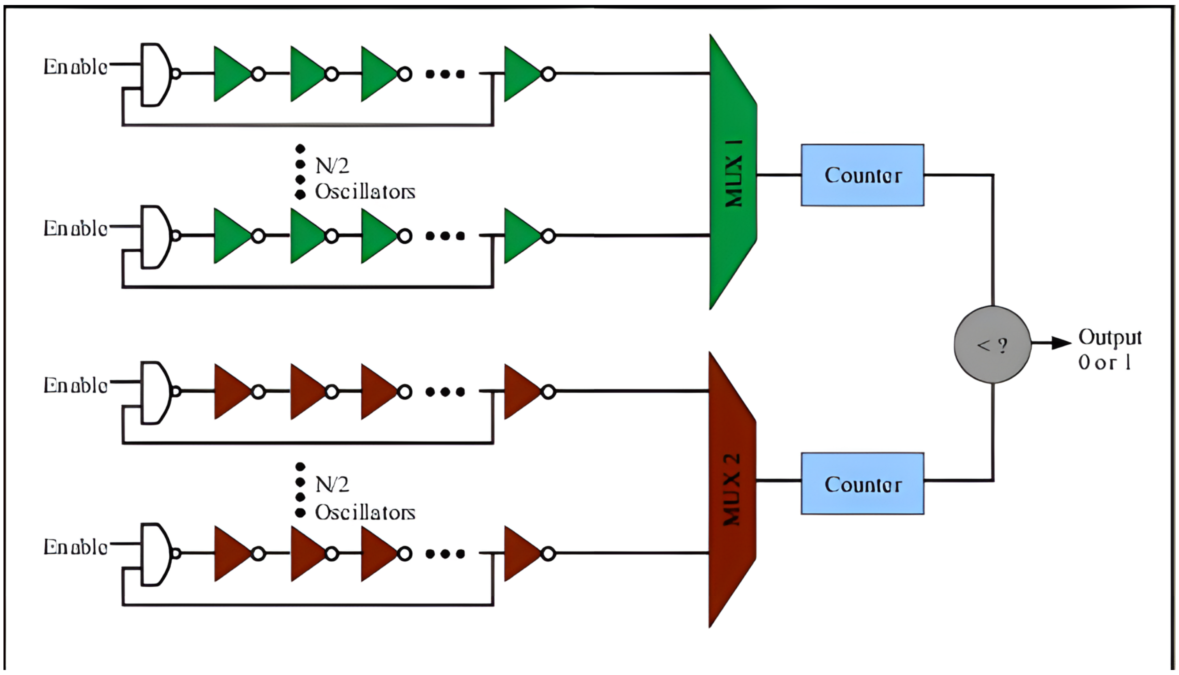

Figure 2.1: Ring Oscillator PUF

SRAM PUF leverage the power-up state of static random-access memory cells. Upon powering up, each cell unpredictably settles to a 0 or 1 due to variations in transistor behavior. These PUFs are attractive because they require no additional hardware on devices already using SRAM. They are highly stable across environmental conditions, making them suitable for secure key storage.

Butterfly PUF mimics the behavior of cross-coupled inverters found in SRAM cells but uses two latches or flip-flops instead. It operates independently of the power-up process, with the random state determined by slight mismatches in the latches. Its design makes it ideal for platforms where standard SRAM is unavailable, though achieving high reliability requires precise control of circuit parameters.

Latch PUF function similarly to SRAM PUFs, relying on the metastable state resolution of latches. Upon a reset, the latch settles randomly into one of two stable states due to process variations. They are simpler to implement on certain hardware platforms but can be less robust than SRAM PUFs due to environmental sensitivity.

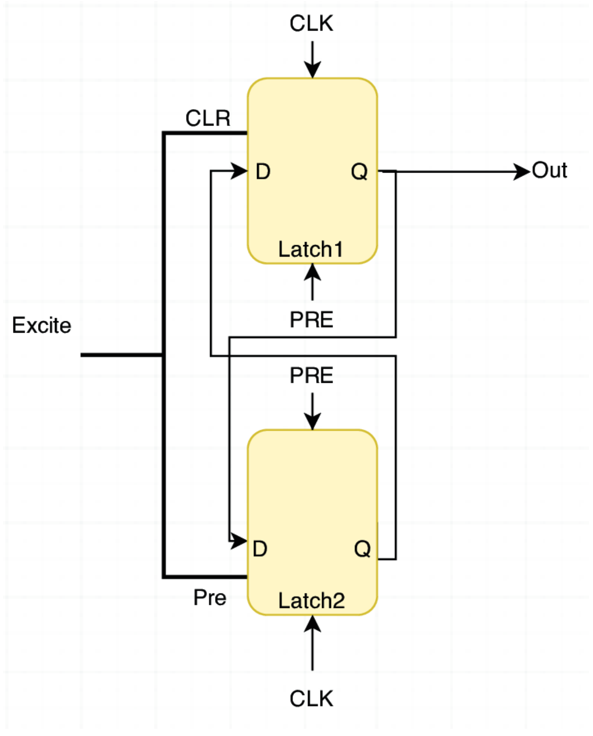

Figure 2.2: Butterfly PUF

Flip-flop PUF generate responses from the power-up state of uninitialized flip-flops. Like SRAM PUFs, these rely on manufacturing variations that influence startup states. They are straightforward to implement on FPGA platforms, as flip-flops are common in FPGA architectures. However, the need for specific initialization sequences can complicate their deployment.

This article [1] describes many types of PUFs, even those that were not created to build a PUF, but have properties that make them considered PUFs.
In the initial phase of this work, among the various PUFs that I could implement, I decided to immediately discard the memory-based PUFs because they are usually weak PUF, and it is hardly implemented on FPGA device because many modern FPGA chips initialize memory units at power-up.

2.2.2 CRP behaviour

In the realm of PUFs, the concept of CRP behaviour can vary significantly depending on the specific implementation. For some PUFs, this functionality is inherently embedded in their design and operation, making the relationship between the challenge and response straightforward and implicit. For example, the behavior of certain PUFs, like SRAM PUFs, is directly tied to their physical properties upon power-up, and no explicit challenge needs to be provided. The unique characteristics of the SRAM cells naturally determine the response.

In contrast, other types of PUFs, such as Arbiter PUFs, require an explicit challenge to function. For an Arbiter PUF, the challenge typically takes the form of a binary string that controls multiplexers in the circuit, dictating the signal propagation paths. These paths generate unique time delays due to manufacturing variations, and the resulting delays are compared to produce the response. In such cases, the challenge must be carefully defined, as it directly determines the behavior of the PUF and the uniqueness of the response.

Moreover, in some PUF implementations, identifying what constitutes the challenge is less evident. It might be necessary to specify specific parameters or settings that act as the challenge. For instance, in Ring Oscillator PUFs, the challenge may involve selecting pairs of oscillators to compare. This variability highlights how the definition of a challenge can range from straightforward digital inputs to physical or parametric inputs, depending on the PUF's architecture.

Another crucial aspect to consider is the **interpretation of the response**, particularly in light of the extensive post-processing that is often applied to PUF outputs. The raw output of a PUF is rarely used directly; instead, it undergoes several steps, including quantization and error correction. Quantization converts the inherently analog measurements of the PUF—such as time delays, frequencies, or voltages—into discrete binary values. This process is essential for aligning PUF responses with the binary format required by digital systems. However, it can introduce potential challenges, such as instability or error, especially under varying environmental conditions like temperature or supply voltage.

Error correction is another common step to ensure reliability. PUFs are inherently sensitive to environmental changes, and error correction mechanisms help produce consistent responses for a given challenge. In some systems, additional transformations, such as cryptographic hashing, may be applied to convert the processed response into a secure key or output suitable for authentication purposes.

These intermediate steps complicate the clear identification of where the "response" should be defined. Depending on the context, the response may refer to the raw physical output of the PUF, the quantized binary result, or the fully processed output ready for use in a cryptographic protocol. This ambiguity underscores the importance of clearly defining the response when analyzing or describing a PUF's behavior.

Finally, while it is generally preferable to represent both the challenges and responses as binary strings for compatibility with digital systems, this often necessitates additional decoding and quantization. For example, in SRAM PUFs, the power-up states of individual memory cells—although already binary—may still require further error correction to ensure consistency across different operating conditions.

Therefore the challenge-response behavior in PUFs is a nuanced concept that varies between implementations. While some PUFs have this functionality inherently embedded, others require explicit definitions of challenges and the parameters they involve. Furthermore, the interpretation of responses is often influenced by the necessary post-processing steps, such as quantization and error correction. These factors must be carefully considered in the design and evaluation of PUF systems to ensure their robustness and usability in practical applications.

2.3 FPGA

A **FPGA** is a type of configurable integrated circuit that can be repeatedly programmed after manufacturing. FPGAs are a subset of logic devices referred to as programmable logic devices. They consist of an array of programmable logic blocks with a connecting grid, that can be configured "in the field" to interconnect with other logic blocks to perform various digital functions. To simplify, I like to think of an FPGA as a box of colorful, non-trademarked building blocks for creating digital circuits. I can connect the blocks together any way that I want to fit my application. If you can describe how a digital circuit behaves, you can create it inside an FPGA with enough logic elements. Need 25 PWM outputs? No problem. Need a special communication protocol? Write some hardware to handle it. FPGAs are often used in limited quantity production of custom-made products, and in research and development, where the higher cost of individual FPGAs is not as important, and where creating and manufacturing a custom circuit wouldn't be feasible. Other applications for FPGAs include the telecommunications, automotive, aerospace, and industrial sectors, which benefit from their flexibility, high signal processing speed, and parallel processing abilities. Field-Programmable Gate Arrays are widely used for implementing digital systems due to their configurability and high performance. However, in many cases, simpler microcontrollers can achieve similar outcomes, offering significant advantages in cost and ease of integration. Microcontrollers are inexpensive and straightforward to implement on a printed circuit board (PCB), making them an appealing choice for many applications. In contrast, FPGAs often come with higher costs, more complex layout requirements, and additional demands for external circuitry. These factors can render them impractical for certain use cases. Despite these challenges, FPGAs excel in scenarios where flexibility and performance are critical. The first major advantage of FPGAs is their **flexibility**. Unlike fixed-function microcontrollers, FPGAs feature configurable logic blocks, enabling hardware to be reprogrammed as requirements evolve. This adaptability allows designers to implement additional peripherals, such as timers or UARTs, as long as sufficient logic resources are available. Furthermore, this reconfigurability can extend the lifecycle of a product by allowing updates to accommodate new technological advancements or shifting application needs. The second key advantage of FPGAs lies in their inherent **parallelism**, which enables superior speed for certain applications. Unlike microcontrollers, which execute instructions sequentially, FPGAs can perform multiple operations simultaneously due to their hardware architecture. This capability is particularly advantageous for tasks such as Fast Fourier Transforms (FFT)[3] or graphics processing, which are computationally intensive on sequential processors. Additionally, FPGAs often provide high-speed I/O capabilities, including Low-Voltage Differential Signaling (LVDS) and transceivers capable of data rates exceeding 10 Gbps, enabling support for high-bandwidth protocols such as HDMI. In summary, while microcontrollers remain a practical and economical choice for many designs, FPGAs offer unparalleled flexibility and performance advantages in applications demanding reconfigurability, parallel processing, or high-speed data communication.

2.3.1 Steps to program an FPGA

Programming an FPGA involves several key steps, each contributing to the design, testing, and deployment of a digital circuit on the programmable fabric. The process ensures that the desired functionality is correctly implemented and optimized for the target device. Below is a detailed description of these steps.

1. **Design creation**
   The first step in programming an FPGA is creating the design that specifies the circuit's functionality. There are two primary approaches:

   - **HDL-based design**:
     Using a hardware description language (HDL) like Verilog or VHDL, designers write code to describe the circuit at a high or low level of abstraction. This method provides flexibility and control over the hardware implementation, making it suitable for complex designs or those requiring precise optimizations.
   - **Graphical design**:
     Alternatively, tools like the block design feature in Vivado allow designers to create the circuit visually by connecting pre-built blocks. This method is intuitive and ideal for users who prefer working with graphical interfaces, especially when integrating complex IP cores such as memory controllers or communication modules.

2. **Assign ports**
   Once the design is created, the next step is to **map the inputs and outputs** of the circuit to the **physical pins** of the FPGA. This process is critical as it defines how the FPGA interfaces with external components, such as sensors, actuators, or other digital systems. The port assignments must align with the constraints of the development board or custom PCB being used.

   In Vivado, this step often involves defining the **constraints file** (e.g., an XDC file) that specifies the physical pin mappings and other parameters, such as clock frequencies or signal standards.

3. **Simulation**

   Simulation is a vital phase that ensures the design functions as intended **before it is implemented in hardware**. Using the built-in simulator in Vivado or an external tool, designers test the circuit under various input scenarios to verify its logical behavior. Simulation helps identify and correct errors early in the design process, saving time and resources. Two types of simulations are typically performed: **functional simulation** and **timing simulation**.

   Functional simulation is the initial stage of testing and focuses purely on the logic of the design without considering physical or timing constraints. It ensures that the circuit performs the correct operations based on the written code or design logic.

   Timing simulation is performed after the design has been synthesized and implemented. It accounts for real-world physical constraints, such as signal delays, interconnect routing, and device-specific timing properties.

4. **Synthesis**

   Synthesis translates the HDL or graphical design into a **gate-level netlist** that the FPGA can understand. During this step, the tool optimizes the design by mapping it to the specific resources available on the FPGA, such as look-up tables (LUTs) and flip-flops. The synthesis report provides insights into resource usage, potential bottlenecks, and areas for optimization, such as reducing area or improving timing performance.

5. **Implementation**

   The implementation phase in FPGA design is a critical step where the synthesized design is mapped onto the physical resources of the FPGA. This process ensures that the logical description of the circuit is translated into a hardware configuration that meets timing, area, and performance requirements.

   During implementation, the first task is **placement**, where the design's logical elements are assigned to specific physical locations on the FPGA. The placement process is guided by constraints and optimization goals, such as minimizing the distance between interconnected elements to reduce signal delay.

   Once the placement is completed, the design moves to the **routing** stage. In

this step, the connections between the placed elements are established using the FPGA's programmable interconnect network. Routing is a complex task, as it must balance the competing demands of timing performance, signal integrity, and resource utilization. The routing process ensures that signals travel efficiently from one element to another without introducing excessive delays or conflicts.
After placement and routing, there is **timing analysis** to verify that the design meets all the specified timing constraints. This includes ensuring that signals propagate through the circuit within the required time for proper operation, particularly for clocked elements like flip-flops. The timing analysis checks for issues such as setup and hold violations, which can compromise the reliability of the design.

6. **Bitstream Generation**
   The final step is generating the bitstream file, which is the **binary configuration file** that programs the FPGA. This file contains all the information needed to configure the FPGA's logic and routing to match the design. The bitstream is loaded onto the FPGA using tools like Vivado's Hardware Manager. Once the programming is complete, the FPGA begins operating as specified by the design, ready for testing or deployment.

2.4 HDL

To program a FPGA we need to use an Hardware Description Language (**HDL**), it enables a precise, formal description of an electronic circuit that allows for the automated analysis and simulation of the circuit. It also allows for the synthesis of an HDL description into a netlist (a specification of physical electronic components and how they are connected together), which can then be **placed and routed** to produce integrated circuit (IC).
Place and route is a stage in the design of PCB, IC, and FPGAs, it is composed of two steps. The first step, **placement**, involves deciding where to place all electronic components, circuitry, and logic elements in a generally limited amount of space.

This is followed by **routing**, which decides the exact design of all the wires needed to connect the placed components. This step must implement all the desired connections while following the rules and limitations of the manufacturing process.

2.5 State of Art

Early references about systems that exploit the physical properties of disordered systems for authentication purposes date back to Bauder in 1983[4] and Simmons in 1984[5]. Naccache and Frémanteau provided an authentication scheme in 1992 for memory cards[6]. PUFs were first formally proposed in a general fashion by Pappu in 2001, under the name Physical One-Way Function (POWF)[7], with the term PUF being coined in 2002[8].
Since their introduction in [9], Arbiter PUFs have been extensively researched for applications in device authentication and cryptographic key generation.

2.5.1 Improvements in Arbiter PUF

Although Arbiter PUFs offer significant advantages, they are susceptible to modeling attacks that exploit their linear delay behavior. Over the years, researchers have proposed various enhancements to improve their resilience:

- **Feed-Forward**: by introducing intermediate arbiters within the PUF design, as proposed in [9], the challenge-response relationship becomes nonlinear, increasing resistance to machine learning attacks.
- **Lightweight PUFs**: studies such as[10] have explored lightweight PUF implementations to reduce resource usage while maintaining robustness against attacks.
- **Multi-Challenge PUFs**: Approaches like those in[11] use multiple challenges simultaneously to improve entropy and randomness.

2.5.2 Security and vulnerabilities

While Arbiter PUFs are attractive for their simplicity, their linear delay model has been extensively exploited in machine learning-based modeling attacks, as demonstrated in[10]. These attacks can recover the PUF's internal parameters with high accuracy if a sufficient number of challenge-response pairs (CRPs) are available. To counter these vulnerabilities, challenge obfuscation techniques (e.g., XOR Arbiter PUFs) have been explored to make modeling attacks computationally expensive.

2.5.3 Applications of Arbiter PUFs

Arbiter PUFs have found use in diverse applications, such as:

- **Authentication**: Generating unique device identifiers[9].
- **Key generation**: Cryptographic key extraction, as discussed in[12].
- **Anti-counterfeiting**: Securely identifying hardware to prevent cloning[10].

CHAPTER 3

Working tools

3.1 Nexys A7

The Nexys A7 FPGA board, developed by Digilent, is a versatile and accessible platform designed around the **Xilinx Artix-7 FPGA family**. This board is particularly suited for educational purposes and projects that require high flexibility in digital design. It offers a rich set of features that make it a powerful tool for prototyping and implementing complex systems. At its core, the Nexys A7 uses the Xilinx XC7A100T-1CSG324C FPGA, which includes 15,850 logic slices, 4,860 Kbits of block RAM, and 240 DSP slices. These specifications support designs requiring high computational power, memory capacity, and digital signal processing capabilities. The FPGA operates with internal clock speeds exceeding 450 MHz and includes advanced functionalities such as an on-chip analog-to-digital converter (**XADC**). The board supports a variety of interfaces, including 10/100 Ethernet, USB, UART, JTAG, and VGA, making it suitable for different applications. Its additional features include a 16-switch input interface, 16 LEDs for visual feedback, two tri-color LEDs, and two four-digit seven-segment displays. These elements facilitate interaction and debugging during project development. Expansion is supported through four Pmod connectors and a dedicated Pmod for XADC signals. Memory capabilities

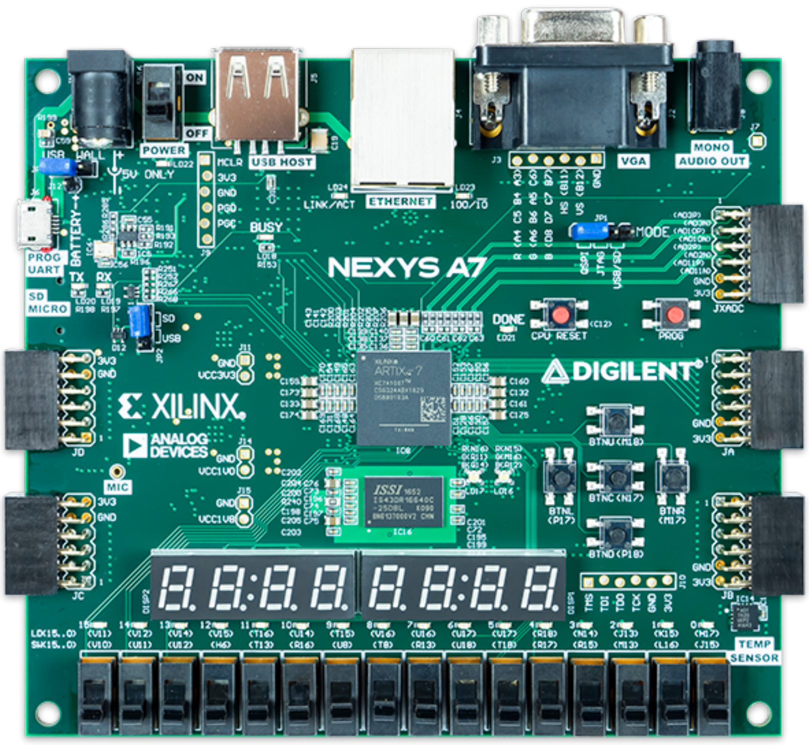

Figure 3.1: Nexys-A7

are robust, featuring 16 MB of QSPI Flash and 128 MB of DDR2 RAM, enabling the implementation of data-intensive applications. The board can be powered via USB or an external 7V-15V source.

3.2 Vivado

Vivado Design Suite, developed by Xilinx (now part of AMD), is a software tool designed for the development and programming of FPGAs and System-on-Chip (SoC) devices. It offers a suite of tools for all stages of FPGA design, from specification to implementation, simulation, and deployment.

There are other software tools for programming FPGAs, even open source ones like F4PGA, unfortunately I discovered it at an advanced stage of this work.

3.3 Verilog

Verilog is a HDL that is used to describe digital systems and circuits in the form of code. It was developed by Gateway Design Automation in the mid-1980s and later acquired by Cadence Design Systems. Verilog is widely used for design and verification of digital and mixed-signal systems, including both application-specific integrated circuits (ASICs) and FPGAs. It supports a range of levels of abstraction, from structural to behavioral, and is used for both simulation-based design and synthesis-based design. The language is used to describe digital circuits hierarchically, starting with the most basic elements such as logic gates and flip-flops and building up to more complex functional blocks and systems. It also supports a range of modeling techniques, including gate-level, RTL-level, and behavioral-level modeling.

3.3.1 Before Verilog

Before the development of Verilog, the primary HDL used for digital circuit design and verification was VHDL (VHSIC Hardware Description Language). VHDL was developed in the 1980s by the U.S. Department of Defense as part of the Very High-Speed Integrated Circuit (VHSIC) program to design and test high-speed digital circuits. VHDL is a complex language that enables designers to describe digital systems using a range of abstraction levels, from the low-level transistor and gate levels up to complex hierarchical systems. It was designed to be more descriptive and flexible than earlier HDLs, such as ABEL (Advanced Boolean Expression Language), ISP (Integrated System Synthesis Procedure), and CUPL (Compiler for Universal Programmable Logic). Despite the development of Verilog and its increasing popularity since the 1980s, VHDL remains a widely used HDL, particularly in Europe and in the military and aerospace industries. Today, both Verilog and VHDL are widely used in digital circuit design and verification, with many companies and organizations using a combination of the two languages.

3.3.2 Verilog is better than its predecessor languages

Verilog introduced several important improvements over its predecessor languages, which helped make it a more popular and effective HDL for digital circuit design and verification. Here are a few reasons why Verilog is considered better than its predecessor HDLs:

- **Simpler syntax**:
  Verilog has a simpler syntax compared to VHDL, which allows designers to write code more quickly and with fewer errors.
- **Better support for behavioral modeling**:
  Verilog provides better support for describing the behavior and functionality of digital designs. It supports a range of modeling techniques, from gate-level to behavioral-level modeling, which makes it easier to describe the behavior of complex digital circuits.
- **Higher level of abstraction**:
  Verilog provides a higher level of abstraction than its predecessor languages. It enables designers to describe digital circuits using concepts such as modules and ports, which makes the design process more efficient.
- **Better tool support**:
  Due to its increasing popularity, Verilog has better tool support than its predecessor languages. Verilog has a range of integrated development environments (IDEs) and simulation tools available, which makes it easier to design and verify digital circuits.

3.3.3 Some differences from other languages

Verilog has a syntax very similar to C language, but it has many differences from the most common high-level languages.

Sequence of instructions In most common languages the instructions are executed one at a time, line by line, from top to bottom, in a procedural manner. Instead, in

HDL we describe logic circuits, such as logic gates (AND, OR, NOT), multiplexers, Flip-Flop and circuits to execute arithmetic operations (half adder, full adder), while the paradigm is not sequential but concurrent: that multiple parts of circuits can operate simultaneously.

There aren't loops In Verilog, there are **for** and **while** statements, but they do not function the same way as they do in other procedural programming languages like C or C++. Instead of being used to control program flow dynamically during execution, as in software programming, these constructs are primarily intended to describe and replicate hardware structures.
Here an example is shown:

```c
int sum = 0;
for (int i = 0; i < 3; i++) {
    sum += array[i];
}
```

Listing 1: Sum elements of an array in C

This is a simple loop in C that sums the first 3 elements of an array sequentially.

```verilog
sum = 0;
for(i = 0; i < 3; i = i + 1) begin
    sum <= sum + array[i];
end;
```

Listing 2: Sum elements of an array in Verilog

This doesn't sum values one at a time, but describes a **combinatorial circuit** that sums all elements of array simultaneously, using parallel logic; thus each iteration of the loop constitutes a part of the circuit that contributes to the sum. These loops are processed only during synthesis; they do not physically exist as a run-time loop.
This is a Verilog loop to sum the first three elements of an array in sequential way. But this code is not synthesizable, so it cannot be run in hardware, but only on a software simulator since there is the **initial block** used mainly to *mock up* variables for use in testing phase.

```
module example_module;
    reg [7:0] array [0:3];
    integer i, sum;

    initial begin
        for (i = 0; i < 3; i = i + 1) begin
                sum = sum + array[i];
        end
    end
endmodule
```

Listing 3: Verilog loop not synthesizable

```
always @(posedge clk) begin
    y = a & b;
end
```

Listing 4: Always block

Unlike a loop, an **always block** doesn't repeat in time sequentially, but runs in parallel with everything else in the circuit whenever an event occurs that is specified among the block's input parameters. In this case, the block is executed at the positive edge of clock.

From these differences you can understand that the first approach to FPGA programming is not trivial.

3.3.4 Nets and variables

Nets and variables are the two main groups of data type in Verilog.

Nets are used to connect between hardware entities like logic gates and so don't store any value on its own. There are different types of nets each with different characteristics, but the most popular and widely used net in Verilog is of type **wire**. The wire is similar to the electrical wire that is used to connect two components on a breadboard.

Variable is an abstraction of a data storage element and can hold values. **reg** in Verilog is a variables and it is usually represented in hardware by a flip-flop.

3.3.5 Scalar and vector

A **wire** or **reg** declaration without a range specification is considered 1-bit wide and is a scalar. If a **range** is specified, then the net or variable becomes a multi-bit entity known as a vector.

```
wire        single_bit;       // scalar
wire [7:0]  wire_8_bit;       // 8-bit vector net
reg         parity;           // single bit scalar variable
reg  [31:0] addr;             // 32 bit vector variable
```

Listing 5: Scalar and vector examples

3.3.6 Ports

Ports are a set of signals that act as inputs and outputs to a particular module and are the **primary way of communicating with it**. Think of a module as a fabricated chip placed on a PCB and it becomes quite obvious that the only way to communicate with the chip is through its pins. Ports are like pins and are used by the design to send and receive signals from the outside world. There are three types of port:

- input
- output
- inout

The syntax to declare a port is:

```
input [netType] [range] portName
```

if the *netType* is omitted, defaults to wire.

3.3.7 Number format

We are most familiar with numbers being represented as decimals. However, numbers can also be represented in binary, octal and hexadecimal. By default, Verilog simulators treat numbers as decimals. In order to represent them in a different radix, certain rules have to be followed.

```
[size]'[baseFormat][number]
```

- **size** is written only in decimal to specify the number of bits in the number.
- **baseFormat** can be:
  - decimal (d or D)
  - binary (b or B)
  - octal (o or O)
  - hexadecimal (h or H)
- **number** is simply the number written in the chosen base format.

```
3'b010;
// size is 3, base format is binary ('b), and the number is 010
3'd2;
// size is 3, base format is decimal ('d) and the number is 2
8'h70;
// size is 8, base format is hexadecimal ('h) and the number is 0x70
```

Listing 6: Number format some examples

Numbers without base format are decimal number by default.

Numbers without size have a default number of bits depending on the type of simulator and machine. For example, in the statement:

```
integer a = 'h1AD7
```

size isn't specified, because *a* is int (32 bits) value stored in *a* is *32'h00001AD7*.

3.3.8 Future of Verilog

It's difficult to predict exactly what may replace Verilog in the future, but there are several emerging technologies and languages that may have an impact on the future of digital system design and verification. One technology that may affect the future of digital system design is High-Level Synthesis (**HLS**), which is a technique for automatically generating hardware designs from high-level descriptions in languages like C and C++. HLS allows designers to express their design intents and functionality at a higher level of abstraction, rather than specifying the details of logic gates and register transfers in Verilog or VHDL. This could enable more efficient and rapid design of digital systems, and allow designers to explore more design space in a shorter period of time. There are also emerging HDLs that are trying to address some of the limitations of Verilog and VHDL, such as **Chisel** and **MyHDL**, which are based on more modern programming concepts and provide higher-level abstractions.

CHAPTER 4

Description and design

4.1 Arbiter PUF structure

The fundamental concept involves creating a digital race condition between two signal paths on a chip. An arbiter circuit is then used to determine which path completes the race first. When the two paths are designed to be symmetrical, meaning they are intended to have identical delays, the outcome of the race cannot be predetermined. However, due to manufacturing variations inherent in the chip production process, physical differences arise between the paths. These variations introduce a small, random offset in the delays, ultimately influencing which path wins the race.

The input of the two first multiplexers is the same and it is a simple enable signal. Each bit of the challenge input is the value for selection bit of each MUX blocks in the chain, every MUX has three inputs(selection bit and the outputs of the two previous MUXes) and one output. In this case, each challenge size is 128 bits, so there are two symmetric chains formed by 128 blocks, each block has 2 multiplexers (one above and one below). At the end of chains, there is a **D latch**, where the value on the D input is latched by the other input. If the upper signal arrives first before the latch, a '1' will be outputted, otherwise a '0' is output. The race will all depend on the delays

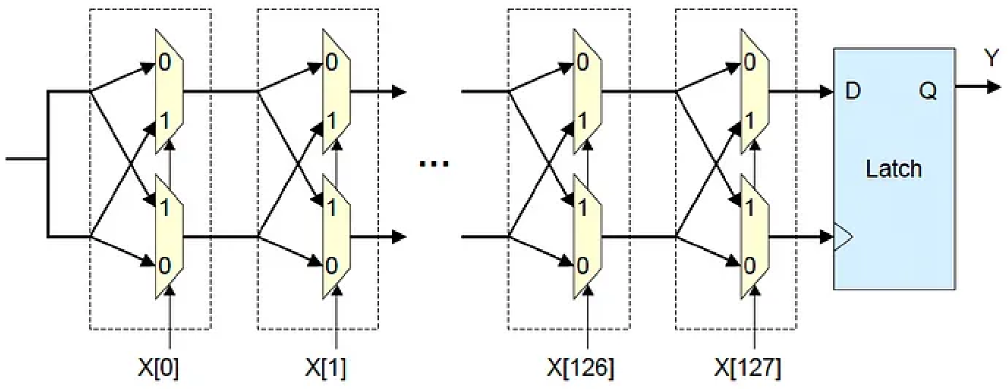

Figure 4.1: Arbiter PUF

in the lines and in the MUXes. There will be 2^{128} different delay paths involved.

4.2 Verilog implementation

This section describes how I implemented the various modules that form the Arbiter PUF in Verilog.

4.2.1 UART communication between PC and FPGA

This project implements a UART (Universal Asynchronous Receiver-Transmitter)[13] communication interface to enable data exchange between an FPGA and a PC via a terminal application. The FPGA acts as the data transmitter, while the PC serves as the receiver.

I implemented a Verilog module to handle serial communications, these are the parameters:

- **Baud rate**: Set to 9600 bps for compatibility with common serial interfaces.
- **Clock frequency**: Defined as 100 MHz, the operating clock frequency of the FPGA.
- **Clocks for bit**: Calculated as $\frac{clock_frequency}{baud_rate}$, representing the number of FPGA clock cycles required to transmit a single bit.

A finite state machine controls the operation on the transmitter, it includes four states: *idle, start, data, stop*.
On the PC, the terminal application Minicom is used to interact with the UART interface.

4.2.2 LFSR

To generate the challenges for the Arbiter PUF, I implemented a Linear Feedback Shift Register (LFSR) in Verilog. The LFSR is a pseudo-random number generator, which produces **deterministic sequences** that appear random but are fully repeatable given the same initial seed. I chose LFSR because it is simple to implement directly in hardware.

The role of the LFSR in this context is to produce 128-bit challenge vectors efficiently and consistently, meeting the requirements for input diversity in the PUF's challenge-response authentication process.

An LFSR is essentially a shift register where the bits are shifted sequentially on every clock cycle. However, unlike a basic shift register, the LFSR includes feedback logic. Certain positions (or "taps") in the shift register are XORed together, and the resulting value is fed back into the least significant bit (LSB) of the register. This feedback mechanism creates a sequence that cycles through many states before repeating, effectively producing a pseudo-random output.

In my implementation, the LFSR consists of a 128-bit register, which holds the current state of the sequence. The feedback taps are positioned at specific positions 98, 100, and 125. These bits are XORed to generate the new LSB value at every clock cycle, ensuring the pseudo-random nature of the sequence. Obviously there is a **static initial seed**.

I chose to implement LFSR because it is a lightweight and hardware-friendly approach to generate long sequences, requiring minimal logic compared to other random number generation methods[14].

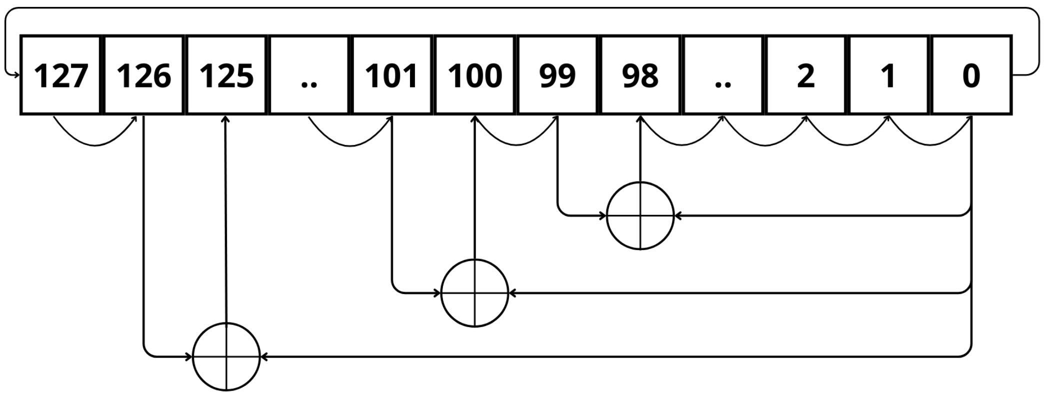

Figure 4.2: LFSR 128 bit

4.2.3 Multiplexer blocks

The design of the Arbiter PUF relies on the structured and scalable use of multiplexer blocks to create two parallel chains of interconnected logic. These chains, each consisting of 128 multiplexer blocks. Each block contains two MUX 2x1 (two bit input, one bit selection and one bit output). This structure has been implemented in verilog in a modular way, each module does only one specific thing, in fact there is a mux2x1 module that takes care of the structure of each single multiplexer, then there is the mux_block module that contains two *MUX2x1* and creates the structure to be connected to the previous and next block. While in the top module there is the generation of the 128 *mux_block*.

4.2.4 D Latch

The D-latch is a fundamental component used as the arbiter in the Arbiter PUF architecture. Its role is to resolve the race condition between the two parallel signal paths. The D-latch operates on two inputs: a data input (D) and a clock signal (clk). Its output, Q, depends on the state of the clock:

- When the clock is high (clk = 1): The latch becomes transparent, and Q directly follows the value of D. This allows the signal from one of the paths to determine

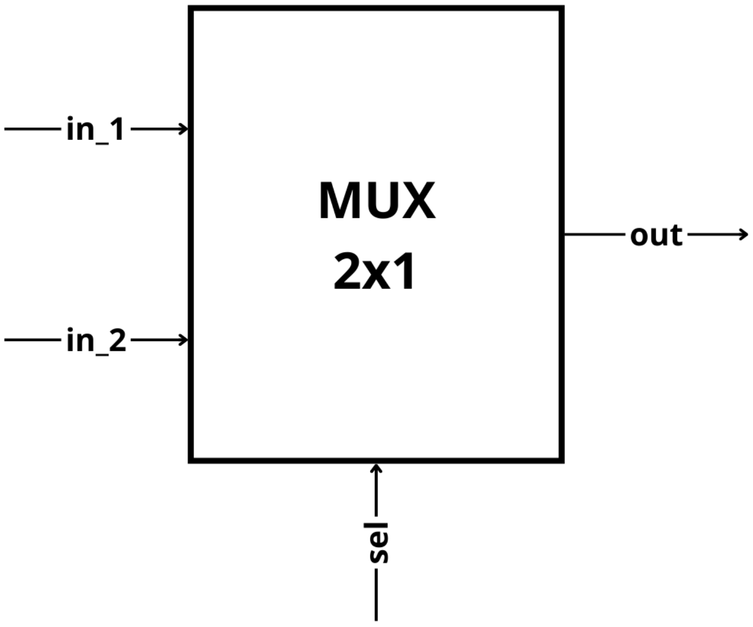

Figure 4.3: MUX 2x1

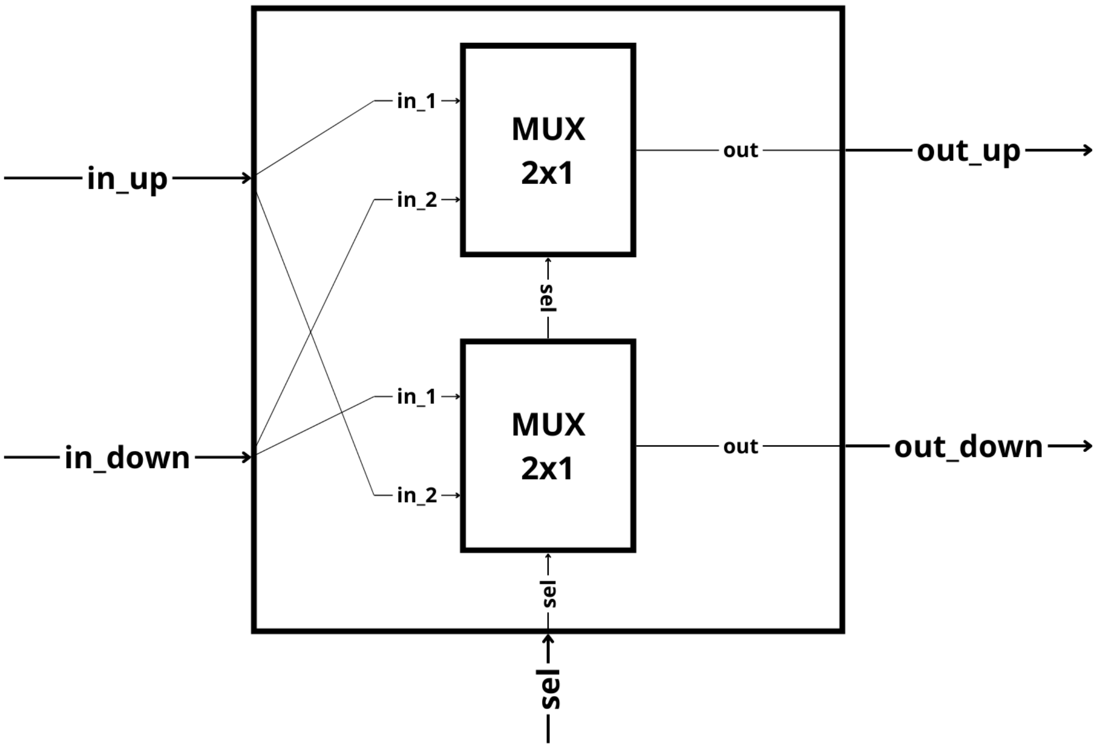

Figure 4.4: MUX block

the output.

- When the clock is low (clk = 0): The output reflects the inverted value of D.

CHAPTER 5

Simulation and verify

5.1 Simulation of each separate module

To ensure the reliability and correctness of the final Arbiter PUF design, each individual module was tested separately through simulation before integration. This modular approach allowed for identifying and resolving potential issues in isolation, simplifying debugging and improving the overall design process.

MUX Block This test ensured that the MUX block behaved as expected under a variety of input combinations, providing confidence in its functionality before replicating it across the chain.

D-Latch The D-latch module, used as the arbiter in the PUF design, was simulated to confirm its capability to produce a stable output (*0* or *1*) based on the relative arrival times of its input signals. Testing this module was crucial, as its functionality directly impacts the correctness of the PUF's response generation. The simulation confirmed that the D-latch output correctly reflected the comparison of the input signal delays.

LFSR To generate pseudo-random input challenges for the PUF, a 128-bit LFSR was implemented and simulated. This module was tested to ensure that it produced a deterministic yet sufficiently diverse sequence of challenges when given a specific seed. The test confirmed that the LFSR behaved predictably and generated the necessary patterns to drive the PUF during testing and evaluation.

UART communication The UART communication module, essential for transferring data between the FPGA and an external terminal, was tested in a stand-alone configuration before being integrated into the Arbiter PUF system. The UART was simulated to verify its ability to transmit and receive data correctly under different baud rates and conditions. Stand-alone testing ensured that the UART worked as expected, facilitating the collection of response data from the PUF in later stages.

5.2 Simulation of PUF

After verifying the functionality of the individual modules, the complete Arbiter PUF system was simulated using the software simulator in Vivado. This simulation provided an opportunity to test the full design before implementing it on hardware, identifying potential issues early in the design process.However, the simulation results revealed a consistent output response of *0* for all challenges provided. This unexpected behavior highlighted a fundamental limitation of software simulation: the lack of **physical delay variability**. The randomness of a PUF's output relies heavily on subtle differences in signal propagation delays, which are inherently present in hardware due to manufacturing variations and physical conditions such as temperature or voltage fluctuations. These characteristics cannot be accurately replicated in a purely software-based simulation environment.This limitation reinforced the necessity of testing the design on actual FPGA hardware to observe the true behavior of the Arbiter PUF. Despite the lack of randomness in the simulation, the process was valuable for verifying the logical correctness of the design and ensuring proper signal routing. Once the simulation phase was complete, the design was prepared for hardware implementation, where the physical properties of the FPGA could be leveraged to achieve the desired randomness in the PUF responses.

5.3 Data gathering

The process of collecting data from the PUF was essential to evaluate its behavior and effectiveness. For this purpose, a quite number of responses were gathered by executing the PUF more times. Each response consisted of a single bit, representing the outcome of the signal propagation and arbitration within the PUF's architecture. To retrieve these responses, the FPGA was connected to a computer via a UART interface. Through this setup, each bit generated by the PUF was transmitted to the terminal, where it was displayed in real-time. These output bits were then saved into a text file for further analysis, ensuring a reliable record of the PUF's behavior.

5.4 Evaluation metrics

The collected data served as the foundation for evaluating the PUF using three key metrics: **uniformity**[15], **stability**, and **randomness**.

5.4.1 Uniformity

Uniformity of a PUF estimates how uniform the proportion of '0's and '1's is in the response bits of a PUF. For truly random PUF responses, this proportion must be 50%. Uniformity is computed as the fractional Hamming Weight (HW) of the total responses:

$$Uniformity(c) = \frac{1}{m} * HW(R_c) * 100\%$$

where m is the length of response on chip c and $HW(R_c)$ is the number of '1's in the responses.

5.4.2 Stability

The function for calculating stability measures the consistency of PUF responses when run multiple times. Stability is an important metric for assessing whether the PUF produces the same output under invariant conditions, such as the same challenge and physical environment. Ideally, the hamming distance should be as close to 0 and the stability close to 100.

$$Stability(c) = (1 - \frac{\sum_{i=0}^{m} HD(R_{c,i}, R_{c,i+1})}{m}) * 100$$

m is the length of response on chip c and $HD(R_{c,i}, R_{c,i+1})$ is the Hamming Distance between the *i*-bit and *i+1*-bit in the response.

5.4.3 Randomness

Randomness is a measure that assesses the degree of unpredictability of PUF-generated responses. This characteristic is critical to ensure that the behavior of the PUF cannot be predicted or modeled by an adversary. The function calculates randomness by analyzing all possible pairs of collected responses. The Hamming Distance between each pair is calculated and used to determine how different the responses are from each other. Ideally, a PUF with high randomness should produce responses that differ significantly even for similar inputs.

$$Randomness(c) = (1 - \frac{1}{\binom{m}{2}} * \sum_{i=0}^{n} (R_{c,i}, R_{c,i+1})) * 100$$

$\binom{m}{2}$ is used to normalize the sum for the number of couples

CHAPTER 6

Results and analysis

6.1 Results presentation

6.1.1 Uniformity

To evaluate the uniformity of the Arbiter PUF, I extracted 100 responses from the circuit by executing the PUF multiple times under identical conditions. Uniformity measures the balance between '0's and '1's in the output responses, ideally aiming for 50%. This ensures that the PUF outputs are unbiased and evenly distributed, which is a crucial property for secure and reliable PUF operation.
From the collected data, the uniformity was calculated to be **47%**. While this result is close to the ideal value, it indicates a slight bias towards the output state of *0*.

6.1.2 Stability

To evaluate the stability, I tested its response consistency by applying the same challenge 200 times under identical conditions. Stability measures the ability of the PUF to consistently produce the same response for a given challenge, even in the presence of environmental noise or intrinsic hardware variations. This is a critical metric for ensuring reliability in practical applications.

So, the stability is **97.98994974874373%**. This high level of stability indicates that the PUF consistently generates repeatable outputs with minimal deviation.

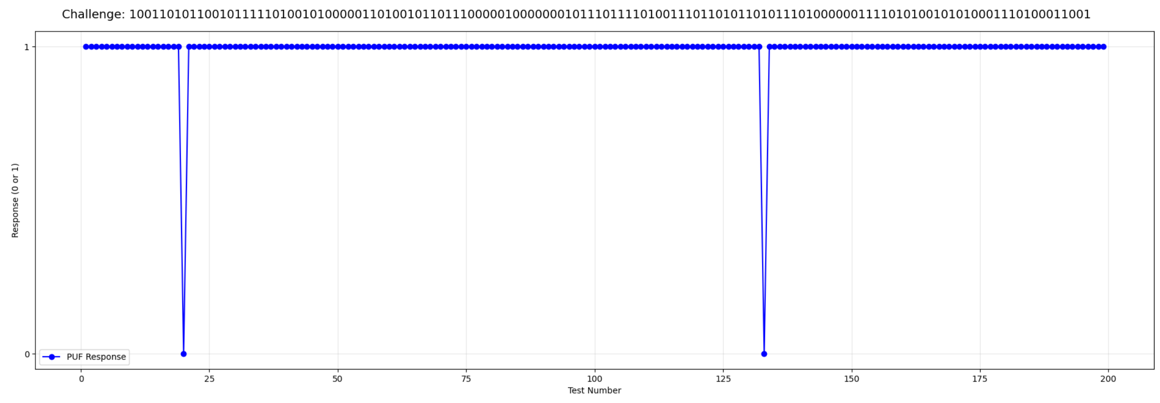

Figure 6.1: Plot of stability response

6.1.3 Randomness

To assess the randomness of the Arbiter PUF, I tested it using 100 different input challenges. Randomness measures how unpredictable the responses are across a wide range of challenges. An ideal PUF should produce outputs that appear completely random, with no discernible patterns, making it resistant to reverse engineering or modeling attacks.
From the collected data, the randomness was calculated to be **99.01%**, which is an exceptional result. This indicates that the responses are highly unpredictable and exhibit near-perfect randomness.

Uniformity	Stability	Randomness
47%	97.99%	99.01%

Table 6.1: Results of evaluation metrics

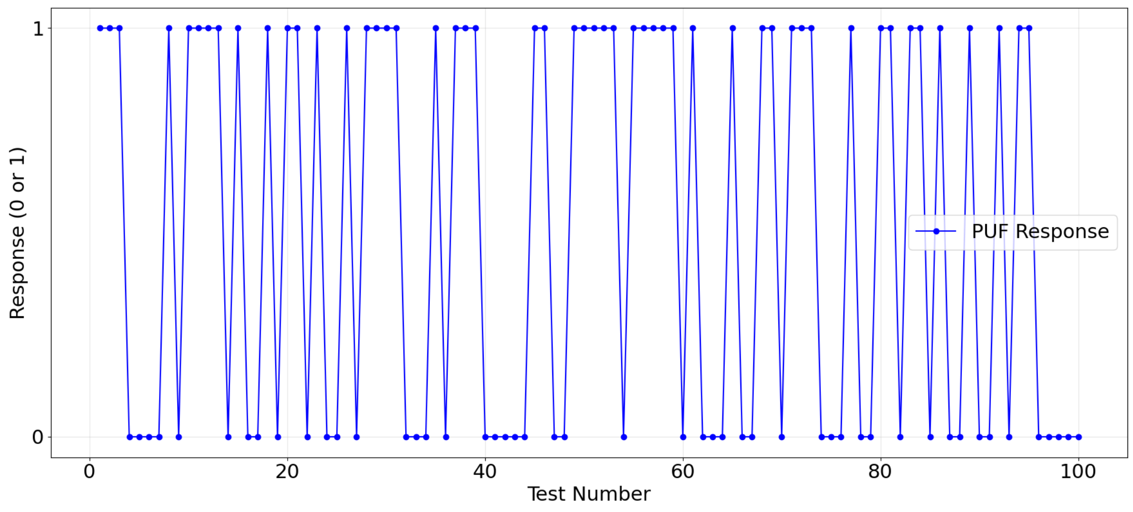

Figure 6.2: Plot of randomness response

6.2 Results analysis

6.2.1 Uniformity

Achieving a uniformity of 47% demonstrates that the PUF design is robust and capable of generating responses that are sufficiently balanced for practical use. This result suggests that no significant post-processing techniques, such as output conditioning or XORing, are required to improve uniformity.

6.2.2 Stability

The excellent stability result suggests that the PUF implementation is highly resilient to small variations in hardware and environmental conditions. The approximately 2.01% inconsistency observed is within the acceptable range for PUFs and could stem from minor fluctuations in operating conditions, such as noise in the power supply or thermal drif

6.2.3 Randomness

A randomness value of 99.01% shows that the PUF outputs are extremely well-distributed across the input space, with minimal bias or correlation between responses. This result aligns closely with the ideal randomness level of 100%, suggesting that the PUF design is highly effective in generating different outputs.

CHAPTER 7

Conclusion

As you can see, the structure of Arbiter PUF is enough simple, but the first approach with FPGA and Verilog is not. I have to admit that I had difficulties with what were the first exercises to familiarize myself with the development environment, but I think this is normal. However, I am satisfied with my work, as I practiced with completely new technologies that were little covered in the course of study I just completed.

7.1 Future works

This section outlines potential future developments to enhance and expand the work carried out in this thesis. While the current implementation provides a basic understanding of PUFs and their evaluation, there are several areas where further exploration could lead to valuable insights and practical applications.

Test PUF with the reliability metric One significant aspect of PUF behavior that warrants further investigation is its reliability across different **environmental conditions**, particularly temperature. Temperature fluctuations can impact the propagation delay of signals within the PUF, potentially causing instability in responses. Future

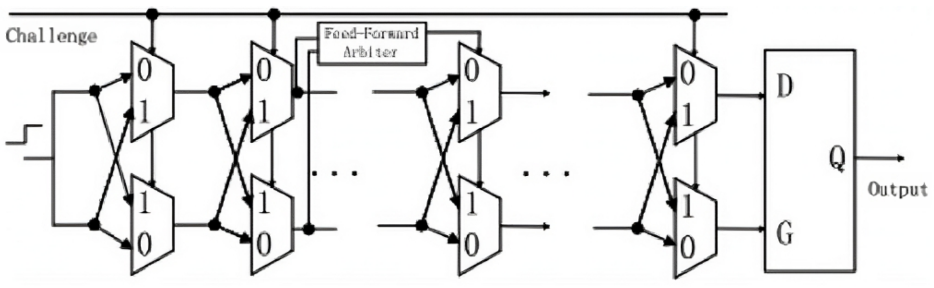

Figure 7.1: Feed-forward architecture

work could involve conducting experiments to measure the stability of the PUF's output at different temperature levels.

Defense against mathematical modeling attacks Another promising direction is the implementation of **feed-forward architectures**[16] in the PUF design. Arbiter PUF can sometimes be vulnerable to attacks based on mathematical modeling, where an adversary attempts to predict responses using machine learning. Feed-forward structures introduce additional layers of complexity by feeding the outputs of intermediate stages back into earlier stages. This added complexity can make the PUF significantly more resistant to predictive attacks, enhancing its security.

Test PUF in real-world applications Finally, an important step forward would be to integrate the PUF into practical use cases. PUFs hold great potential for various applications, such as secure key generation, authentication, and hardware security. Future work could focus on embedding the PUF into a system where it plays a critical role, such as an IoT device requiring lightweight and robust security.

Bibliography

[1] R. Maes, "Physically unclonable functions: A study on the state of the art and future research directions," pp. 6–21, 2010. (Citato alle pagine 5 e 7)

[2] K. Lata and L. R. Cenkeramaddi, "Fpga-based puf designs: A comprehensive review and comparative analysis," *Cryptography*, vol. 7, no. 4, 2023. [Online]. Available: https://www.mdpi.com/2410-387X/7/4/55 (Citato a pagina 5)

[3] G. Slade, "The fast fourier transform in hardware: A tutorial based on an fpga implementation," 03 2013. (Citato a pagina 10)

[4] D. Bauder, "An anti-counterfeiting concept for currency systems," 1983. (Citato a pagina 14)

[5] G. Simmons, "A system for verifying user identity and authorization at the point-of-sale or access," pp. 1–21, 1984. (Citato a pagina 14)

[6] P. F. David Naccache, "Unforgeable identification device, identification device reader and method of identification," 1992. (Citato a pagina 14)

[7] B. T. J. G. Pappu, R.; Recht, "Physical one-way functions," 2002. (Citato a pagina 14)

[8] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM Conference on Computer and*

Communications Security, ser. CCS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 148–160. [Online]. Available: https://doi.org/10.1145/586110.586132 (Citato a pagina 14)

[9] J. Lee, D. Lim, B. Gassend, G. Suh, M. van Dijk, and S. Devadas, "A technique to build a secret key in integrated circuits for identification and authentication applications," vol. 176-179, 07 2004, pp. 176 – 179. (Citato alle pagine 14 e 15)

[10] U. Rührmair, H. Busch, and S. Katzenbeisser, *Strong PUFs: Models, Constructions, and Security Proofs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 79–96. [Online]. Available: https://doi.org/10.1007/978-3-642-14452-3_4 (Citato alle pagine 14 e 15)

[11] J. Delvaux, D. Gu, D. Schellekens, and I. M. R. Verbauwhede, "Helper data algorithms for puf-based key generation: Overview and analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, pp. 889–902, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:3815882 (Citato a pagina 14)

[12] R. Maes, A. Van Herrewege, and I. Verbauwhede, "Pufky: A fully functional puf-based cryptographic key generator," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, E. Prouff and P. Schaumont, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 302–319. (Citato a pagina 15)

[13] W. Huang and G. Sheng, "Analysis and research on uart communication protocol," in *2024 4th Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*, 2024, pp. 768–771. (Citato a pagina 26)

[14] S. Akter, K. Khalil, and M. Bayoumi, "Efficient pseudo random number generator (prng) design on fpga," in *2024 IEEE 17th Dallas Circuits and Systems Conference (DCAS)*, 2024, pp. 1–5. (Citato a pagina 27)

[15] A. Maiti, V. Gunreddy, and P. Schaumont, "A systematic method to evaluate and compare the performance of physical unclonable functions," Cryptology ePrint Archive, Paper 2011/657, 2011. [Online]. Available: https://eprint.iacr.org/2011/657 (Citato a pagina 33)

[16] K. Hatti and P. C, “Design and implementation of enhanced puf architecture on fpga,” *International Journal of Electronics Letters*, vol. 10, 12 2020. (Citato a pagina 40)

Acronyms

PUF	Physical Unclonable Function
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
HSL	High-Level Synthesis
RTL	Register Transfer Level
LUT	Look-Up Table
MUX	Multiplexer
CRP	Challenge-Response Pair
POWF	Physical One-Way Function
UART	Universal Asynchronous Receiver-Transmitter
JTAG	Joint Test Action Group
SPI	Serial Peripheral Interface
LFSR	Linear-Feedback Shift Register
RO PUF	Ring Oscillator PUF
IoT	Internet of Things
IC	Intregrated Circuit

ASIC	Application Specific Integrated Circuit
PCB	Printed Circuit Board
LVDS	Low-Voltage Differential Signaling
FFT	Fast Fourier Transforms
SoC	System on Chip
ABEL	Advanced Boolean Expression Language
CUPL	Compiler for Universal Programmable Logic
IDE	Integrated Development Environments
HD	Hamming Distance
HW	Hamming Weight

GitHub Repository

https://github.com/tatore02/ArbiterPUF_on_FPGA

Ringraziamenti

In primis, voglio ringraziare i miei genitori per il sostegno e per non avermi mai messo alcuna pressione durante il percorso universitario (basta già la mia).

Un ringraziamento speciale va ai miei compagni di viaggio: Antonio, Francesco, Gianmarco e Simone, con cui ho condiviso e condivido ancora pensieri, idee e progetti.

Un pensiero va anche ai ragazzi del team CTF ByteTheCookies, con cui ho il privilegio di poter collaborare e 'catturare bandiere' il fine settimana (non tutti, ahimè purtroppo), col tempo sto capendo l'importanza del lavoro di squadra.

Non posso non citare i professori, alcuni di loro si sono distinti per la passione con cui trasmettevano le loro conoscenze, a questi docenti va il mio ringraziamento per avermi dato maggiori stimoli.

Infine, ringrazio il Professore Esposito per avermi dato la possibilità di lavorare a questa tesi e per i suoi preziosi consigli durante lo sviluppo di essa.

Un grazie va anche ai ragazzi del laboratorio Habes - Franco, Marco e Biagio - che mi hanno accolto e dato consigli utili sullo svolgimento di questo lavoro.

Grazie.