Characterization of FPGA-based Arbiter Physical Unclonable Functions

JINGNAN SHAO
Characterization of FPGA-based Arbiter Physical Unclonable Functions

JINGNAN SHAO
Abstract

The security of service, confidential data, and intellectual property are threatened by physical attacks, which usually include reading and tampering the data. In many cases, attackers can have access to the tools and equipment that can be used to read the memory or corrupt it, either by invasive or non-invasive means. The secret keys used by cryptographic algorithms are usually stored in a memory. Physical unclonable functions (PUFs) are promising to deal with such vulnerabilities since, in the case of PUFs, the keys are generated only when required and do not need to be stored on a powered-off chip. PUFs use the inherent variations in the manufacturing process to generate chip-unique output sequences (response) to a query (challenge). These variations are random, device-unique, hard to replicate even by the same manufacturer using identical process, equipment and settings, and supposed to be static, making the PUF an ideal candidate for generation of cryptographic keys.

This thesis work focuses on a delay-based PUF called arbiter PUF. It utilizes the intrinsic propagation delay differences of two symmetrical paths. In this work, an arbiter PUF implemented in Altera FPGA has been evaluated. The implementation includes Verilog HDL coding, placement and routing, and the communication methods between PC and FPGAs to make testing more efficient.

The experimental results were analyzed based on three criteria, reliability, uniqueness, and uniformity. Experimental results show that the arbiter PUF is reliable with respect to temperature variations, although the bit error rate increases as the temperature difference becomes larger. Results also reveal that the uniqueness of the PUFs on each FPGA device is particularly low but on the other hand, the proportions of different response bits are uniform after symmetric routing is performed.

Keywords

Arbiter PUF, FPGA, Altera, Verilog HDL
Sammanfattning

Tjänstens säkerhet, konfidentiella uppgifter och immateriell egendom hotas av fysiska attacker, som vanligtvis inkluderar läsning och manipulering av uppgifterna. I många fall kan angripare ha tillgång till de verktyg och utrustning som kan användas för att läsa minnet eller skada det, antingen med invasiva eller icke-invasiva medel. De hemliga nycklarna som används av kryptografiska algoritmer lagras vanligtvis i ett minne. Fysiska okonabla funktioner (PUF: er) lovar att hantera sådana sårbarheter eftersom, för PUF: er, nycklarna genereras endast när det behövs och inte behöver lagras på ett avstängd chip. PUF: er använder de inneboende variationerna i tillverkningsprocessen för att generera chip-unika utgångssekvenser (svar) på en fråga (utmaning). Dessa variationer är slumpmässiga, enhetsunika, svårt att kopiera till och med av samma tillverkare med identisk process, utrustning och inställningar, och antas vara statisk, vilket gör PUF till en idealisk kandidat för generering av kryptografiska nycklar.

Detta avhandlingsarbete fokuserar på en fördröjningsbaserad PUF som kallas arbiter PUF. Den använder de inneboende utbredningsfördröjningsskillnaderna för två symmetriska vägar. I detta arbete har en arbiter PUF implementerad i Altera FPGA utvärderats. Implementeringen inkluderar Verilog HDL-kodning, placering och routing och kommunikationsmetoderna mellan PC och FPGA för att effektivisera testningen.

De experimentella resultaten analyserades baserat på tre kriterier, tillförlitlighet, unikhet och enhetlighet. Experimentella resultat visar att arbiter PUF är tillförlitlig med avseende på temperaturvariationer, även om bitfelfrekvensen ökar när temperaturdifferensen blir större. Resultaten avslöjar också att unikheten hos PUF: erna på varje FPGA-enhet är särskilt låg men å andra sidan är proportionerna av olika svarbitar enhetliga efter att symmetrisk dirigering har utförts.

Nykkelord

Arbiter PUF, FPGA, Altera, Verilog HDL
Acknowledgements

First, I would like to thank my examiner, Professor Elena Dubrova, and Masoud Asadi at Ericsson for giving me the opportunity to start this thesis project. They gave me guidelines for the project so I can keep moving forward.

Second, I want to express my gratitude to my supervisors Sumedh Dhabu and Sha Tao at Ericsson for their support, kindness, and patience. They gave me considerable suggestions and help during the whole process and really inspired me a lot.

Third, I would like to thank Yang Yu for providing detailed instructions at the beginning of the project. I also want to thank Tommy Lindgren B and Weiping Huang for your help with the software and the temperature chamber.

Fourth, I am much obliged to my line manager, Joakim Plahn. He arranged everything for me in the office so I can have a good study environment. He always encouraged and motivated me when I slacked off.

A special thanks to my friends in Stockholm and back home in China to brightened my days during the last half year. Thanks to my parents who always care about my life.

Stockholm, October 2019
Jingnan Shao
# Contents

1 Introduction 1  
1.1 Background ............................................. 1  
1.2 Goals ................................................... 2  
1.3 Research Methodology ................................. 2  
1.4 Structure of the Thesis ............................... 2  

2 Background 3  
2.1 Physical Unclonable Function ......................... 3  
2.2 Arbiter PUF ............................................. 4  
2.3 Evaluation Metrics for PUF ............................ 5  
  2.3.1 Uniformity ......................................... 5  
  2.3.2 Uniqueness ......................................... 5  
  2.3.3 Reliability ......................................... 6  
2.4 Related Work ............................................ 6  

3 Methods for FPGA Design and Characterization 8  
3.1 Workflow ............................................... 8  
3.2 Function Specification ............................... 9  
3.3 Experimental Design .................................. 9  
  3.3.1 Arbiter Function Verification ..................... 9  
  3.3.2 Test Environment .................................. 10  
  3.3.3 Hardware and Software ........................... 10  

4 Design and Implementation of Arbiter PUF Based on Altera FPGA 13  
4.1 Verilog HDL Design ..................................... 13  
  4.1.1 128-bit LFSR and Arbiter PUF ..................... 14  
  4.1.2 Finite-State Machine ............................... 14  
4.2 Placement ............................................... 15  
4.3 LUT Routing ............................................ 18  
4.4 Communication Between PC and FPGA ............... 18
5 Results and Analysis

5.1 Before Manual Routing

5.1.1 Reliability
5.1.2 Uniqueness
5.1.3 Uniformity

5.2 After Manual Routing

5.2.1 Reliability
5.2.2 Uniqueness
5.2.3 Uniformity

5.3 Close Placement

5.3.1 Uniqueness
5.3.2 Uniformity

6 Conclusions and Future Work

6.1 Conclusions
6.2 Limitations
6.3 Future Work

Bibliography

A Verilog HDL
List of Figures

2.1 Arbiter PUF ........................................ 4
2.2 Switch block structure .............................. 4
2.3 Arbiter operation of D flip-flop ....................... 5

3.1 The flowchart of the characterization project .......... 8
3.2 General structure of the arbiter PUF’s design on FPGA ...... 9
3.3 Altera DE2-115 Development and Education Board ........... 10
3.4 A LAB with 16 LEs of Cyclone IV ...................... 11
3.5 LUT of Cyclone IV ................................... 12
3.6 Climate chamber in Ericsson ........................... 12

4.1 FSM for Verilog HDL design ........................ 13
4.2 Galois LFSRs with taps [128, 126, 101, 99] .......... 14
4.3 The structure of 2-to-1 MUX and D flip-flop ............ 15
4.4 Partial placement of the elements ..................... 16
4.5 Close placement of the elements ...................... 17

5.1 Reliability(%) under the fixed temperature ............... 20
5.2 Bit error rate compared to 0°C ........................ 21
5.3 The distribution of HW for different boards ............. 23
5.4 Bit error rate compared to 0°C after defining LUT routing ... 25
5.5 The distribution of HW for different boards after defining LUT routing ........................................ 26
5.6 The connection and routing of MUXs in Chip Planner .... 27
5.7 The distribution of HW before and after defining LUT routing 28
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Reliability(%) under the fixed temperature</td>
<td>19</td>
</tr>
<tr>
<td>5.2</td>
<td>Bit error rate(%) compared to 0°C</td>
<td>20</td>
</tr>
<tr>
<td>5.3</td>
<td>Uniqueness(%) among 5 boards</td>
<td>21</td>
</tr>
<tr>
<td>5.4</td>
<td>Uniqueness(%) between 2 boards, 20°C</td>
<td>22</td>
</tr>
<tr>
<td>5.5</td>
<td>Uniformity(%) of 5 boards at 20°C</td>
<td>22</td>
</tr>
<tr>
<td>5.6</td>
<td>Reliability(%) under the same temperature after defining LUT routing</td>
<td>24</td>
</tr>
<tr>
<td>5.7</td>
<td>Bit error rate(%) compared to 0°C after defining LUT routing</td>
<td>24</td>
</tr>
<tr>
<td>5.8</td>
<td>Uniqueness(%) among 5 boards after defining LUT routing</td>
<td>24</td>
</tr>
<tr>
<td>5.9</td>
<td>Uniformity(%) of 5 boards at 20°C after defining LUT routing</td>
<td>25</td>
</tr>
</tbody>
</table>
# List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>PUF</td>
<td>Physical Unclonable Function</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read-only Memory</td>
</tr>
<tr>
<td>CRP</td>
<td>Challenge Response Pair</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>RO PUF</td>
<td>Ring Oscillator PUF</td>
</tr>
<tr>
<td>HW</td>
<td>Hamming Weight</td>
</tr>
<tr>
<td>HD</td>
<td>Hamming Distance</td>
</tr>
<tr>
<td>LFSR</td>
<td>Linear Feedback Shift Register</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-up Table</td>
</tr>
<tr>
<td>LAB</td>
<td>Logic Array Block</td>
</tr>
<tr>
<td>LE</td>
<td>Logic Element</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Cryptographic algorithms are used to protect service, confidential data, and intellectual property, whose security is threatened by attackers nowadays. Cryptographic algorithms are a set of mathematical instructions to encrypt or decrypt data and they rely on cryptographic keys used in the encryption and decryption process [1]. The keys are usually stored in a non-volatile memory, e.g. flash or EEPROM. Therefore, the strength of cryptographic protection depends on the difficulty for an attacker to read the memory. Physical attacks like side-channel attacks towards memory can be carried out through physical access by adversaries [2]. Physical tampering attacks like laser cutting, micro-probing, and glitch attacks can be used to extract digitalized cryptographic keys from integrated circuits (ICs) [3].

Physical unclonable functions (PUFs) have been proposed [4] to deal with the limitations of non-volatile memory key storage. The PUF is a function that is based on a certain physical system. PUFs exploit intrinsic manufacturing process variations to generate device-specific keys that cannot be cloned or stolen by attackers even if they have full access to the device. The secret keys are generated only when demanded and do not remain stored on the chip. This makes the chip or IC with better tamper-resistance.

The arbiter PUF proposed in [5] is a delay-based PUF [6]. Delay-based PUFs exploit the delay characteristics of electronic elements and wires caused by the manufacturing process variations from chip to chip. A simple arbiter PUF consists of two delay paths and one arbiter. The paths both end in the arbiter. The input challenges are used to decide the construction of the two paths, but the paths are designed carefully in advance to have the same nom-
inal delay. Due to the subtle silicon variations, two exactly identical paths don’t exist. As a result, the signals on two paths won’t have the same arrival time. The arbiter figures out which signal arrives first and outputs a binary response. Different challenges generate unpredictable responses because of the unknown delay differences. Those challenge-response pairs (CRPs) can be used for encryption.

1.2 Goals

The goal of this master thesis project is to characterise and evaluate an arbiter PUF in terms of technical parameters, uniformity, uniqueness, and reliability.

1.3 Research Methodology

There are two implementation platforms to design cryptosystems like PUF, application-specific integrated circuit (ASIC) and field programmable gate arrays (FPGA). Compared with ASIC, FPGA can be reprogrammed multiple times. The implementation of FPGA-based systems takes a shorter time to market and costs less money. FPGA-based systems also take less time in software implementation. Therefore, in this thesis, an FPGA-based system was chosen to evaluate the arbiter PUF.

1.4 Structure of the Thesis

Chapter 2 gives an overview of PUF technology and an introduction of the structure of an arbiter PUF and performance metrics used in the evaluation.

Chapter 4 shows the detailed implementation of the arbiter PUF on Altera FPGA - Cyclone IV including the Verilog HDL design and the execution on FPGA.

Chapter 5 presents the experimental results and corresponding analysis/discussions.

Chapter 6 gives the conclusions and suggestions for future work.
Chapter 2

Background

This chapter briefly provides background information about PUF technology in the last two decades. Additionally, the chapter describes the structure of an arbiter PUF and what metrics are used to evaluate its performance.

2.1 Physical Unclonable Function

PUFs are proposed as a physical random function by Gassend et al [4]. It can be an approach to decreasing the vulnerabilities of common cryptographic key generation and storage processes against physical attacks. A PUF is based on a certain physical system. It can exploit the manufacturing variations in integrated circuits (ICs) to generate ‘random’ output values which are unpredictable for attackers, even with physical access to the system. In other words, those specific mappings between a set of inputs and outputs, which are also called challenges and responses respectively, are used to generate keys, and also authenticate devices [7]. PUFs are expected to have many properties like unpredictability, uniqueness, and unclonability, which can be quantified with metrics described in Section 2.3.

Many different types of PUFs have been proposed so far. Maes et al. give readers an overall introduction of those PUFs [8]. Some of the PUFs are classified as memory-based PUFs or delay-based PUFs that can be implemented on embedded hardware. Memory-based PUFs are based on the settling state of digital memory cells while delay-based ones compare the propagation delay differences. The arbiter PUF and the ring oscillator (RO) PUF [7] are two popular delay-based PUFs. The arbiter PUF was first proposed by Lee et al [5].
2.2 Arbiter PUF

There are small delay differences between the symmetric paths on a chip. The arbiter PUF takes advantage of the inherent delay differences between two symmetrical paths to generate a single bit at the output end of the circuit [5]. It consists of multiple switch blocks and one arbiter at the end of the chain as shown in Figure 2.1 [6].

![Figure 2.1: Arbiter PUF](image)

The input of the first switch block is connected to one enable signal. Each switch block has two outputs and three inputs. The inputs are one challenge bit $C_i$ of the $n$-bit challenge and two outputs $A_{i-1}$, $B_{i-1}$ of the previous switch block. A switch block is composed of two 2-to-1 multiplexers (MUXs) with the same selection bit $C_i$ shown in Figure 2.2. If the challenge bit is ‘0’, the paths will be parallel. Otherwise, the paths will be crossed.

![Figure 2.2: Switch block structure](image)

The outputs of the last switch block are connected to an arbiter. The arbiter can be a latch or a flip-flop, which generates one-bit response due to the
different arrival times of the outputs from the $n$-th switch block. A D flip-flop is shown in Figure 2.3. When the upper data input D arrives first, the response will be ‘1’. When the lower clock signal arrives first, the response will be ‘0’. The nominal wire delays on symmetric paths are equal ($\delta_{ia} = \delta_{ib}$ and $\delta_{ic} = \delta_{id}$). Different challenges make wire configuration different, which will lead to unpredictable responses. When the number of challenge bits is $n$, there are $2^n$ combinations of wire delays.

![Arbiter operation of D flip-flop](image)

**Figure 2.3: Arbiter operation of D flip-flop**

## 2.3 Evaluation Metrics for PUF

The performance of PUFs can be evaluated with three parameters: uniformity, uniqueness, and reliability [9] [10].

### 2.3.1 Uniformity

Uniformity characterizes how uniform the proportion of ‘0’ and ‘1’ is in the responses of one PUF. For PUF responses to be unpredictable and random, the probability of ‘1’s should be equal to the probability of ‘0’s ideally. Uniformity is computed as the fractional Hamming Weight (HW) of the total responses:

$$Uniformity(i) = \frac{1}{m} \times HW(R_i) \times 100\%$$  \hspace{1cm} (2.1)

where $R_i$ is the $m$-bit responses on chip $i$ and $HW(R_i)$ is the number of ‘1’s in the responses. The value is supposed to be close to 50%.

### 2.3.2 Uniqueness

Uniqueness characterizes how easily one PUF instance can be distinguished from another PUF instance. Uniqueness is a measure of inter-chip differences so each pair of chips should be considered. It can be evaluated with the average
fractional inter-chip Hamming Distance (HD) among responses generated by the same challenges from different chips as:

\[
Uniqueness = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{HD(R_i, R_j)}{m} \times 100\%
\] (2.2)

where \(k\) is the number of chips, \(R_i\) and \(R_j\) represent \(m\)-bit responses generated on different chips \(i\) and \(j\) \((i \neq j)\) respectively. For the uniqueness of the PUF on each device, the value should approach 50%.

### 2.3.3 Reliability

Reliability characterizes the ability of a PUF in reproducing the responses during different measurements. The same quantity of responses are extracted multiple times under various environments, e.g. temperature and supply voltage. The average fractional intra-chip HD among responses generated by the same challenges on the same chip is used to estimate reliability:

\[
HD_{intra}(i) = \frac{1}{x} \times \sum_{y=1}^{x} \frac{HD(R_i, R'_{i,y})}{m} \times 100\%
\] (2.3)

where \(x\) is the number of samples to be collected on chip \(i\) in the same environment, \(R_i\) is the reference \(m\)-bit responses of chip \(i\) in some environmental condition, and \(R'_{i,y}\) is the responses of the \(y\)th sample in a different condition. The reliability of the PUF on chip \(i\) is defined as:

\[
Reliability(i) = 100\% - HD_{intra}(i) = (1 - \frac{1}{x} \times \sum_{y=1}^{x} \frac{HD(R_i, R'_{i,y})}{m}) \times 100\%\)
\] (2.4)

Ideally, \(HD_{intra}\) is expected to be 0 while the reliability is 100%, which means the PUF is totally reliable.

### 2.4 Related Work

Gassend et al. studied the characteristics of arbiter PUFs on 23 Xilinx Spartan 2 FPGAs [11]. In their experiments, the uniqueness, the intra-chip HD under the same temperature and the intra-chip HD with a temperature change over 40°C were 1.05%, 0.1%, and 0.3% approximately, which were described as inter-chip variation, measurement noise, and temperature variation in their
work. The uniqueness is relatively low on Xilinx Spartan 2 FPGAs because of the asymmetry of their implementation.

Morozov et al. analyzed the delay-based PUFs including the arbiter PUF on Xilinx Spartan3E FPGAs [12]. The delay variation due to static routing is higher than the variation due to manufacturing difference on this type of FPGA so the arbiter PUF didn’t work in the identification and authentication of devices.
Chapter 3

Methods for FPGA Design and Characterization

3.1 Workflow

Figure 3.1: The flowchart of the characterization project
The characterization of the arbiter PUF based on Altera FPGA contains the FPGA-based design and implementation of the PUF and also the tests and analysis on different devices under various temperatures. The workflow of the characterization and evaluation project is illustrated in Figure 3.1.

### 3.2 Function Specification

The Altera FPGA-based system design of the arbiter PUF’s characterization shown in Figure 3.2 is based on a previous work [13]. It contains two main blocks, an \( n \)-bit linear feedback shift register (LFSR) and an arbiter PUF. The \( n \)-bit LFSR is used to generate different challenges continuously when testing, while the \( n \)-bit value is constant when generating a fixed number of responses.

There are two test modes. If \( Selection \) is equal to 1, the basic function of the arbiter PUF instance and the feasibility of the design can be tested and verified (see Section 3.3.1). If \( Selection \) is equal to 0, the \( l \)-bit identification keys generated by the arbiter PUF can be saved and analyzed later. The length of keys and challenges are assigned in Section 4.

![General structure of the arbiter PUF’s design on FPGA](image)

**Figure 3.2: General structure of the arbiter PUF’s design on FPGA**

### 3.3 Experimental Design

As previously described in 2.3, tests should be performed on different devices under various temperatures.

#### 3.3.1 Arbiter Function Verification

Given \( Selection = 1 \), the \( n \)-bit challenges are expected to make the bitstream the same value, either all ones or all zeros. Try several random constant values to check if the responses are always the same for one value. If not, calculate
the error bit rate to find out whether the problem is related to HDL design or placement and routing.

### 3.3.2 Test Environment

Given $Selection=0$, for each device (board) at the temperature from $0^\circ$C to $80^\circ$C, the same experiment is repeated 11 times to generate a large amount of data. The number of boards is 5.

### 3.3.3 Hardware and Software

The Altera DE2-115 board (Figure 3.3) with Cyclone IV EP4CE115F29C7N chip is the device type used for physical experiments [14].

![Figure 3.3: Altera DE2-115 Development and Education Board](image)

Quartus II is the design software for Altera FPGAs. The version of Quartus II used is 18.1 Subscription Edition.
FPGA Architecture and Resources

The oscillator clock source of DE2-115 is 50MHz and the power supply is 12V \[15\]. Cyclone IV has logic elements (LEs) as the smallest logic units, each LE with a 4-input look-up table (LUT) and a register. Each logic array block (LAB) contains 16 LEs as shown in Figure 3.4. There are 1,395 LABs in total on Cyclone IV.

The 4-input LUT is a circuit in Cyclone IV FPGA to implement any logic function of 4 inputs, whose architecture is illustrated in Figure 3.5. 4 inputs are connected to the selector pins of the corresponding 2-to-1 MUXs. The expected function of this example is a 2-to-1 MUX and the desired output is $B \bar{D} + CD$, where D is the selector input and the two inputs are B and C. In this case, the LUT mask should be F0CC in hexadecimal or 1100000011001100 in binary.

The JTAG cable is used for communication between PC and FPGA.

Test Equipment

The experiments related to temperature variations are conducted in the climate chamber (Figure 3.6) provided by Ericsson.
CHAPTER 3. METHODS FOR FPGA DESIGN AND CHARACTERIZATION

Figure 3.5: LUT of Cyclone IV

Figure 3.6: Climate chamber in Ericsson
Chapter 4

Design and Implementation of Arbiter PUF Based on Altera FPGA

The simple arbiter PUF with 128-bit challenges \( n=128 \) in Figure 3.2 was designed and implemented. The length of keys extracted from the bitstream is set to be 128 as well \( l=128 \) in Section 3.2.

4.1 Verilog HDL Design

With the approach of HDL design, the 128-bit LFSR and the arbiter PUF are built first. The evaluation process shown in Figure 3.2 should be repeated 11 times to generate a large number of bits for characteristic analysis. Therefore, a finite-state machine (FSM) shown in Figure 4.1 is used to control different parts of the design. The Verilog HDL files are in Appendix A.

Figure 4.1: FSM for Verilog HDL design
4.1.1 128-bit LFSR and Arbiter PUF

The LFSR with the Galois style uses fewer resources compared to the one with the Fibonacci style [16]. The 128-bit LFSR with Galois style shown in Figure 4.2 can be clearly coded with Verilog HDL. Each block is one register, saving a binary value of the 128-bit challenge. The values are shifted in the direction of the arrows when the trigger signal arrives. The bit positions that affect the next state with the exclusive-OR (XOR) linear function are called taps [17]. The taps for maximum-length LFSR 128-bit counters with a primitive polynomial are 128, 126, 101, and 99 [18].

![Figure 4.2: Galois LFSRs with taps [128, 126, 101, 99]](image)

The arbiter PUF with 128-bit challenge inputs contains 256 2-to-1 MUXs as delay elements and switch blocks along with one D flip-flop as the arbiter. For the bottom-up procedure, the MUX can be defined once and instantiated multiple times in a loop so that they are identical. Input \( j \) is the one connected to ‘0’ in Figure 2.2 while \( k \) is what connected to ‘1’, and \( s \) is the selection bit. The D flip-flop is rising-edge triggered like in Figure 2.3. A D flip-flop in Figure 4.3c is made of two D latches in Figure 4.3b.

4.1.2 Finite-State Machine

The FSM is triggered by some ‘start’ signal manually. The start state is idle.

The idle state is followed by the LFSR state. The 128-bit LFSR will be started to generate a challenge the next clock cycle after the LFSR start state. The LFSR wait state follows in the next clock cycle. That state will cost 32 clock cycles to guarantee the complement of challenge generation.

The PUF start state is followed by the LFSR wait state. The arbiter PUF is enabled to get one bit with either the challenge which is automatically generated by LFSR or the challenge which is manually input (128-bit value in Figure 3.2). The PUF wait state acts as the LFSR wait state to make sure the finish of the response generation.

The key is set to be 128 bits wide, thus 128 bits are considered as a ‘word’. Every response bit is stored every inner circle in Figure 4.1, but a ‘word’ with 128 bits is stored every 128 inner circles to the on-chip memory. The inner circle will be repeated a fixed number of times set up in advance in the design.
The wait state will be locked as the present state unless another trigger signal is given manually. During the wait state, the bitstream can be read to a file.

4.2 Placement

Quartus II is the software used to build the project since a Cyclone IV FPGA is the device type chosen for the PUF implementation. The Quartus fitter places the logic elements (LEs) for better timing and power supply. If the placement is automatically executed by the fitter, the delay on one path in the arbiter PUF may be shorter than the delay on another one no matter what challenges are given. Therefore, it’s necessary to place the PUF components symmetrically by hand.

The Quartus Prime Settings File (.qsf) contains all of the project-wide and entity-level assignments and settings [19]. The floorplan of the chip is presented in the Chip Planner. The locations of each LE of an LCELL can be found in the Properties window. The assignments of the components, MUXs and the flip-flop, are added to the .qsf file and the new placement can be checked in the Chip Planner after recompilation. For example, the assignment of the enable signal is

```
set_location_assignment FF_X24_Y72_N27 -to puf_en
```
and the assignments of the first four MUXs are

```plaintext
set_location_assignment LCCOMB_X23_Y71_N0 -to
 "arbiter_puf:puf_ins | mux4x2:loop0|xg"
set_location_assignment LCCOMB_X25_Y71_N0 -to
 "arbiter_puf:puf_ins | mux4x2:loop0|yg"
set_location_assignment LCCOMB_X23_Y71_N24 -to
 "arbiter_puf:puf_ins | mux4x2:loop[1].inst|xg"
set_location_assignment LCCOMB_X25_Y71_N24 -to
 "arbiter_puf:puf_ins | mux4x2:loop[1].inst|yg"
```

Figure 4.4: Partial placement of the elements

The partial placement at the beginning of the circuit is shown in Figure 4.4, which includes the register generating the enable signal, several LUTs
realising MUXs on the delay paths, and the LUTs used for the D flip-flop.

In order to avoid the effects of the other elements’ placement, the Design Partition and the LogicLock Region (LLR) in Chip Planner are applied for locking the area of the PUF instance on the chip. Set “State: Locked” and “Reserved: On”.

The interconnections within a LAB and between LABs are different. During the analysis of the experimental results, a new close placement is performed shown in Figure 4.5. The new assignments of the elements shown above are

```plaintext
set_location_assignment FF_X26_Y71_N1 -to puf_en
set_location_assignment LCCOMB_X25_Y71_N0 -to "arbiter_puf:puf_ins|mux4x2:loop0|xg"
set_location_assignment LCCOMB_X27_Y71_N0 -to "arbiter_puf:puf_ins|mux4x2:loop0|yg"
set_location_assignment LCCOMB_X25_Y71_N2 -to "arbiter_puf:puf_ins|mux4x2:loop[1].inst|xg"
set_location_assignment LCCOMB_X27_Y71_N2 -to "arbiter_puf:puf_ins|mux4x2:loop[1].inst|yg"
```

Figure 4.5: Close placement of the elements
4.3 **LUT Routing**

Because the routing is also determined by the Quartus fitter like placement, it may also lead to the result that the delay of one path is always shorter than the other in one switch block. As described in [20], “Back-Annotate Assignments” feature can be used to generate the Routing Constraints File (.rcf). However, for both Quartus II 15.0 and 18.1 that have been tested, “Pin, cell, routing & device assignments” cannot be found under “Back-Annotate Assignments”. Therefore, defining the LUT routing wasn’t completed successfully first.

Manual changes of routing were then performed in *Chip Planner*. LUT inputs and masks can be modified in the *Quartus Resource Property Editor*. All the changes should be applied by clicking “Check and Save All Netlist Changes” in *Change Manager*. Routing modification and recompilation for this design took a longer time than expected but finally, LUT routing was defined successfully to see if there would be better performance, which is shown in Section 5.2.

For the upper MUX, *xg*, *yg*, and *challenge* bit are connected to B, C, and D while the mask is F0CC so that *xg* is selected when *challenge* bit is 0. For the lower MUX, *xg*, *yg*, and *challenge* bit are connected to C, B, and D while the mask is F0CC.

4.4 **Communication Between PC and FPGA**

The communication between PC and FPGA is through FPGA’s JTAG interface. The following tools are all in the *tools* option in the menu bar. With *Programmer* in Quartus, the configuration bitstream (.sof) file is downloaded to the chip. With *In-System Memory Content Editor*, the data stored on the chip is read and can be saved to a Memory Initialization File (.mif) file containing bitstream. With *In-System Sources and Probes Editor*, input signals are given values continuously from PC such as *Selection* and *128-bit value* in Figure 3.2. Both of the in-system tools need to be instantiated in the design.
Chapter 5

Results and Analysis

The length of the responses is 128 bit. The number of responses is $2^{13} = 8,192$. The number of CPRs for the arbiter PUF is $2^{13} \times 128 = 1,048,576$. MATLAB is used to process the .mif files generated by In-System Memory Content Editor in Quartus II, which contains response results.

5.1 Before Manual Routing

First, the routing is conducted by the fitter in Quartus II automatically.

5.1.1 Reliability

Reliability under the same temperature is the average value among measurements of the same PUF with the same set of challenges 11 times. From Table 5.1, the reliability for PUFs on every board is all above 99.8%.

Table 5.1: Reliability(%) under the fixed temperature

<table>
<thead>
<tr>
<th>Temperature(°C)</th>
<th>0</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
</table>
Figure 5.1 shows the reliability of most boards increases slightly as the temperature goes up.

![Figure 5.1: Reliability(%) under the fixed temperature](image)

Reliability under varying temperatures is evaluated by generating the PUF responses for the same set of input challenges 11 times and comparing it to the expected ‘error-free’ reference responses. The reference responses are obtained by using majority voting to the 11 results. The average bit error rate (BER) shown in Table 5.2 can describe the intra-chip Hamming Distance.

**Table 5.2: Bit error rate(%) compared to 0°C**

<table>
<thead>
<tr>
<th>Temperature(°C)</th>
<th>0</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. 1</td>
<td>0</td>
<td>0.25</td>
<td>0.40</td>
<td>0.56</td>
<td>0.78</td>
</tr>
<tr>
<td>No. 2</td>
<td>0</td>
<td>0.28</td>
<td>0.47</td>
<td>0.66</td>
<td>0.85</td>
</tr>
<tr>
<td>No. 3</td>
<td>0</td>
<td>0.21</td>
<td>0.37</td>
<td>0.54</td>
<td>0.74</td>
</tr>
<tr>
<td>No. 4</td>
<td>0</td>
<td>0.23</td>
<td>0.40</td>
<td>0.60</td>
<td>0.80</td>
</tr>
<tr>
<td>No. 5</td>
<td>0</td>
<td>0.19</td>
<td>0.34</td>
<td>0.50</td>
<td>0.70</td>
</tr>
<tr>
<td>Average</td>
<td>0</td>
<td>0.23</td>
<td>0.40</td>
<td>0.57</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Figure 5.2 shows that the larger the temperature differences are, the larger the BER is, which indicates the less reliable the arbiter PUF is. The BER in-
creases near linearly with the temperature. Temperature influences the wire delay in this design indeed. The average BER among different chips is 0.77% with a temperature change of 80°C. The reliability against various temperatures is all above 99%, which means the PUF-based key is consistent in different temperatures for FPGA in general.

![Figure 5.2: Bit error rate compared to 0°C](image)

### 5.1.2 Uniqueness

The uniqueness represents the inter-chip variation. It is expected to be about 50%, which is not the case from the results in Table 5.3 and 5.4.

<table>
<thead>
<tr>
<th>Temperature(°C)</th>
<th>0</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniqueness(%)</td>
<td>3.0341</td>
<td>2.9433</td>
<td>2.8623</td>
<td>2.7768</td>
<td>2.6828</td>
</tr>
</tbody>
</table>

The PUF-based key isn’t unique device by device in this case. The dominant reason is the asymmetric routing. It is really hard and complicated to define the routing on FPGA. The fitter of Quartus II decides on the routing itself. In the tool *Chip Planner*, the LUT routing of each switch block can be acquired by double-clicking. Most of the LUT routing differs from each
### Table 5.4: Uniqueness(%) between 2 boards, 20°C

<table>
<thead>
<tr>
<th></th>
<th>No. 1</th>
<th>No. 2</th>
<th>No. 3</th>
<th>No. 4</th>
<th>No. 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. 2</td>
<td>3.0133</td>
<td>2.8864</td>
<td>2.9395</td>
<td>2.8839</td>
<td></td>
</tr>
<tr>
<td>No. 3</td>
<td>2.6317</td>
<td>3.1179</td>
<td>3.2966</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. 4</td>
<td>2.7963</td>
<td>2.9972</td>
<td></td>
<td></td>
<td>3.0418</td>
</tr>
<tr>
<td>No. 5</td>
<td></td>
<td>2.8701</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

other. Some of the operations in previous work [20] can’t be realized at first mentioned in Section 4.3.

### 5.1.3 Uniformity

An arbiter PUF is supposed to generate ‘0’s and ‘1’s randomly, thus for 128-bit keys, the average Hemming Weights should be close to 64 and the percentage of Hemming Weights should be 50%. The average uniformity of 5 boards at 20°C is 33.34% shown in Table 5.5.

Table 5.5: Uniformity(%) of 5 boards at 20°C

<table>
<thead>
<tr>
<th>FPGA</th>
<th>No.1</th>
<th>No.2</th>
<th>No.3</th>
<th>No.4</th>
<th>No.5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniformity(%)</td>
<td>33.02</td>
<td>34.15</td>
<td>33.58</td>
<td>33.08</td>
<td>32.89</td>
<td>33.34</td>
</tr>
</tbody>
</table>

A distribution diagram is used to inspect the uniformity of the responses. From the results shown in Figure 5.3, the distribution of Hamming Weights is similar to a Gaussian one but biased towards 40. The responses depend mostly on the placement and routing so this design apparently can’t be the best routing.

### 5.2 After Manual Routing

As described in Section 4.3, the LUT routing is conducted manually. LUT inputs and masks are able to be defined. Manual changes for the routing of 128 switch blocks takes a couple of hours. The application of the changes takes around 15 minutes while the normal compilation only takes 5 minutes. The same tests and analyses are repeated to compare the results.
5.2.1 Reliability

From Table 5.6, reliability for PUFs on every board is all above 99.8% after manual LUT routing. There is not much difference between the results before and after manual modification.

Additionally, the BER compared to $0^\circ C$ increases almost linearly with the temperature shown in Table 5.7 and Figure 5.4 after defining LUT routing.

5.2.2 Uniqueness

However, after making the LUT inputs and masks identical within the switch blocks, the uniqueness in Table 5.8 seemed a little lower than the previous
### Table 5.6: Reliability(%) under the same temperature after defining LUT routing

<table>
<thead>
<tr>
<th>Temperature(°C)</th>
<th>0</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. 4</td>
<td>99.8805</td>
<td>99.8996</td>
<td>99.9029</td>
<td>99.9033</td>
<td>99.9046</td>
</tr>
<tr>
<td>No. 5</td>
<td>99.8806</td>
<td>99.8936</td>
<td>99.9025</td>
<td>99.9008</td>
<td>99.9036</td>
</tr>
</tbody>
</table>

### Table 5.7: Bit error rate(%) compared to 0°C after defining LUT routing

<table>
<thead>
<tr>
<th>Temperature(°C)</th>
<th>0</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. 1</td>
<td>0</td>
<td>0.25</td>
<td>0.37</td>
<td>0.53</td>
<td>0.61</td>
</tr>
<tr>
<td>No. 2</td>
<td>0</td>
<td>0.21</td>
<td>0.35</td>
<td>0.51</td>
<td>0.62</td>
</tr>
<tr>
<td>No. 3</td>
<td>0</td>
<td>0.24</td>
<td>0.38</td>
<td>0.53</td>
<td>0.63</td>
</tr>
<tr>
<td>No. 4</td>
<td>0</td>
<td>0.27</td>
<td>0.45</td>
<td>0.56</td>
<td>0.68</td>
</tr>
<tr>
<td>No. 5</td>
<td>0</td>
<td>0.26</td>
<td>0.41</td>
<td>0.54</td>
<td>0.65</td>
</tr>
<tr>
<td>Average</td>
<td>0</td>
<td>0.25</td>
<td>0.39</td>
<td>0.53</td>
<td>0.64</td>
</tr>
</tbody>
</table>

ones, which is not what was expected. The uniqueness at 20°C is 1.9247% but the previous uniqueness is 2.9433%. Homogeneous LUT routing can’t bring improvement in the uniqueness.

### Table 5.8: Uniqueness(%) among 5 boards after defining LUT routing

<table>
<thead>
<tr>
<th>Temperature(°C)</th>
<th>0</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniqueness(%)</td>
<td>2.0160</td>
<td>1.9247</td>
<td>1.8464</td>
<td>1.7809</td>
<td>1.7203</td>
</tr>
</tbody>
</table>

### 5.2.3 Uniformity

On the other hand, different from the uniqueness, after manually changing the routing of the upper and lower switch blocks to be exactly the same, the dis-
Figure 5.4: Bit error rate compared to 0°C after defining LUT routing

...distribution became closer to a Gaussian one as shown in Figure 5.5. The average HW $\sigma$ is close to 64 which is just the expected result. Uniformity can be improved a lot after manual LUT routing.

Table 5.9: Uniformity(%) of 5 boards at 20°C after defining LUT routing

<table>
<thead>
<tr>
<th>FPGA</th>
<th>No.1</th>
<th>No.2</th>
<th>No.3</th>
<th>No.4</th>
<th>No.5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniformity(%)</td>
<td>46.76</td>
<td>46.87</td>
<td>46.98</td>
<td>47.13</td>
<td>46.44</td>
<td>46.84</td>
</tr>
</tbody>
</table>

5.3 Close Placement

A close placement shown in Figure 4.5 is applied to inspect if the distances between the switch blocks affect the uniqueness. The reliability of the new PUF instance still has no much difference.

5.3.1 Uniqueness

The uniqueness before and after manual LUT routing of 5 boards at 20°C is 3.1686% and 2.1007%, while the previous values are 2.9433% and 1.9247% respectively. Similarly, identical LUT routing doesn’t improve the uniqueness. The distances between switch blocks don’t affect the uniqueness a lot either.
Figure 5.5: The distribution of HW for different boards after defining LUT routing

The connection of four MUXs in Chip Planner is shown in Figure 5.6. Figure 5.6a and 5.6b depict the parallel and crossed connection in a switch block. Even though the LUT routing of each MUX is the same, the expanded connections in blue lines and routing in red lines are not symmetric in Figure 5.6b. The nominal delays of the two paths may not be the same.

5.3.2 Uniformity

The average uniformity of 5 boards at 20°C before and after manual LUT routing is 45.36% and 48.17%, while the previous values are 33.34% and 46.84% respectively. The distribution of HW with the new placement in Figure
5.7 shows obvious improvement towards a normal distribution. LUT routing makes a difference to the uniformity.

Figure 5.6: The connection and routing of MUXs in *Chip Planner*
Figure 5.7: The distribution of HW before and after defining LUT routing
Chapter 6

Conclusions and Future Work

6.1 Conclusions

The structure of the arbiter PUF isn’t complex but the implementation on FPGA took lots of effort. Although temperature differences can increase the bit error rate, the responses of the arbiter PUF on FPGA are quite reliable in general, with reliability all above 99%. The uniqueness among devices is approximately 3%, quite lower than what was expected. The uniformity of the responses can be improved after making the LUT routing symmetric between the two delay paths.

6.2 Limitations

Though the LUT routing can be defined to make the wire symmetric in each switch block, some wire delay can’t be guaranteed the same on two paths, for example, the wire delay between the enable signal to the first switch block and the wire delay between the last switch block and the arbiter. Besides, the structure of D flip-flop is also asymmetric, which may cause delay variation.

6.3 Future Work

Since the arbiter PUF compares the propagation delays, those challenges causing the response bits fluctuate (unreliable) may bring better uniqueness. In this project, random challenges are given so it deserves to discuss about the robustness of those challenges, which can obtain both excellent reliability and uniqueness.
For this design, it seems to be a physical ‘clonable’ function instead of a physical unclonable function. In the future, it is worthy to be discussed if the PUF with low uniqueness can be used in the industry except for device identification. It may be a method to develop type-specific key generator but not device-specific. The arbiter PUF implemented in FPGA may also be used as a random number generator.
Bibliography


Bibliography


module puf_rng
(
    input clk
);

    parameter idle = 3'b000;
    parameter lfsr_start = 3'b001;
    parameter lfsr_wait = 3'b111;
    parameter puf_start = 3'b010;
    parameter puf_wait = 3'b011;
    parameter puf_store_bit = 3'b100;
    parameter puf_store_word = 3'b101;
    parameter start_wait = 3'b110;

    reg [2:0] state;
    reg [6:0] cnt_128;
    reg [4:0] cnt_wait_cycles;
    reg start;
    reg stop;
    reg rst_n;
    reg store_bit;
    reg store_word;

    reg puf_en;
    reg en_lfsr;
    reg rst_lfsr;

    reg [127:0] challenge;
    wire response;
    reg [127:0] word_128;

    always @(posedge clk or negedge rst_n)
    begin
        if(!rst_n)
            begin
                state <= idle;
                puf_en <= 1'b0;
                store_bit <= 1'b0;
                store_word <= 1'b0;
        
    end
end

APPENDIX A. VERILOG HDL 35

```verilog
// Example HDL code
module example(
    // Inputs
    input start,
    // Outputs
    output stop,
    // Variables
    reg cnt_128, cnt_wait_cycles, stop,
    // Control signals
    reg puf_en, store_bit, store_word, en_lfsr

    // Case statements
    case (state)
    idle:
        begin
            if(!start)begin
                stop <= 1'b0;
            end
            if(start && (!stop))begin
                cnt_128 <= 7'b1111111;
                state <= lfsr_start;
            end
        end
    lfsr_start:
        begin
            en_lfsr <= 1'b1;
            cnt_wait_cycles <= 5'b11111;
            state <= lfsr_wait;
        end
    lfsr_wait:
        begin
            cnt_wait_cycles <=
                cnt_wait_cycles - 1;
            if (cnt_wait_cycles == 0)
                state <= puf_start;
        end
    puf_start:
        begin
            puf_en <= 1'b1; // pos edge gen
            cnt_wait_cycles <= 5'b11111;
            state <= puf_wait;
        end
    puf_wait:
        begin
            cnt_wait_cycles <=
                cnt_wait_cycles - 1;
            if (cnt_wait_cycles == 0)
                state <= puf_store_bit;
        end
    puf_store_bit:
        begin
            store_bit <= 1'b1;
        end
endmodule
```
state <= puf_store_word;
end

puf_store_word:
begin
if (cnt_128 == 0) begin
store_word <= 1'b1;
state <= start_wait;
end
else begin
  cnt_128 <= cnt_128 - 1;
  state <= lfsr_start;
end
end

start_wait:
begin
  if (mem_wr == 1 && mem_addr == 13'b1111111111111)
  begin
    stop <= 1'b1;
  end
  if (mem_wr == 1)
  begin
    state <= idle;
  end
end

default:
begin
  state <= idle;
end
endcase
end

always @(posedge clk or negedge rst_n)
begin
  if (!rst_n) begin
    word_128 <= 128'b0;
  end
  else if (store_bit) begin
    word_128 <= (word_128 << 1);
    word_128[0] <= response;
  end
end

reg [127:0] mem_data;
reg [12:0] mem_addr;
reg mem_wr;

always @(posedge clk or negedge rst_n)
begin
  if (!rst_n) begin
    mem_addr <= 13'b1111111111111;
  end
end
mem_wr <= 1'bx0;
mem_data <= 0;
end
else begin
  mem_wr <= 1'bx0;
  if (store_word) begin
    mem_data <= word_128;
    mem_addr <= mem_addr + 1;
    mem_wr <= 1'bx1;
  end
end

reg [2:0] source;
wire [127:0] chal_cst;
wire [127:0] chal_lfsr;
reg sel;

always @(posedge clk or negedge rst_n)
  rst_lfsr <= (!rst_n);

testreg testreg(
  .probe(),         //
    probes.probe
  .source_clk(clk),  //
    source_clk.clk
  .source(source)    //
    sources.source
);

always @(posedge clk)
begin
  start <= source[0];
  rst_n <= source[1];
  sel  <= source[2];
end

lfsr_reg lfsrreg(
  .probe(),         //
    probes.probe
  .source_clk(clk),  //
    source_clk.clk
  .source(chal_cst)  //
    sources.source
);

G_LFSR128 lfsr_ins(.clk(clk),
  .en(en_lfsr),
  .rst(rst_lfsr),
  .stage(chal_lfsr)
);

always @(posedge clk or negedge rst_n)
begin
if(!rst_n)
  challenge <= 128'b0;
else begin
  if (sel==1)
    challenge <= chal_cst;
  else challenge <= chal_lfsr;
end

// assign challenge = (sel==1) ? chal_cst : chal_lfsr;

arbiter_puf puf_ins (.in_X (puf_en),
  .in_Y (puf_en),
  .Chal(challenge),
  .out_Q(response)
);

test_mem testmem (.address (mem_addr),
  .clock(clk),
  .data(mem_data),
  .wren(mem_wr),
  .q()
);

endmodule

module arbiter_puf #
  ( parameter le = 128 )
  ( input in_X,
    input in_Y,
    input [le-1:0] Chal,
    output out_Q
  );
  wire [le-1:0] A;
  wire [le-1:0] B;

  mux4x2 loop0 (.A(in_X),
    .B(in_Y),
    .c(Chal[0]),
    .X(A[0]),
    .Y(B[0])
  );

  genvar i;
  generate
    for (i = 1; i < le; i = i + 1)
      begin : loop
        mux4x2 inst (.A(A[i-1]),
          .B(B[i-1]),
          .c(Chal[i]),
          .X(A[i]),
          .Y(B[i])
        );
      end
  endgenerate
endgenerate
flipflop FF(A[127], B[127], out_Q);
endmodule

module mux4x2 (A, B, c, X, Y);
  input A, B, c;
  output X, Y;
  wire xg, yg/* synthesis keep */;
  mux2x1 M1(A, B, c, xg);
  mux2x1 M2(B, A, c, yg);
  assign X = xg;
  assign Y = yg;
endmodule

module mux2x1 (j, k, s, m);
  input j, k, s;
  output m;
  wire sg;
  wire jg, kg;
  not (sg, s);
  and (jg, j, sg);
  and (kg, k, s);
  or (m, jg, kg);
endmodule

module flipflop(Df, Cf, Qf);
  input Df, Cf;
  output Qf;
  wire Qm;
  d_latch master(Df, ~Cf, Qm);
  d_latch slave(Qm, Cf, Qf);
endmodule

module d_latch
{
  input D,
  input C,
  output Q
};
  wire R, S, Qn;
  wire R_g, S_g /* synthesis keep */;
  assign S = D;
  assign R = ~D;
  and (R_g, R, C);
  and (S_g, S, C);
  nor (Q, R_g, Qn);
  nor (Qn, S_g, Q);
endmodule
module G_LFSR128
(
    input clk,
    input rst,
    input en,
    output reg [127:0] stage
);

always @ (posedge clk or posedge rst)
begin
    if (rst)
    stage <= 128'
        hc68d8f390b46dd048f9eb80572892b7d
    ;
    else if (en)
    begin
        //stage[0] is feedback
        stage[0] <= stage[1];
        stage[1] <= stage[2];
        stage[2] <= stage[3];
        stage[3] <= stage[4];
        stage[4] <= stage[5];
        stage[5] <= stage[6];
        stage[6] <= stage[7];
        stage[7] <= stage[8];
        stage[8] <= stage[9];
        stage[9] <= stage[10];
        stage[10] <= stage[11];
        stage[11] <= stage[12];
        stage[12] <= stage[13];
        stage[13] <= stage[14];
        stage[14] <= stage[15];
        stage[15] <= stage[16];
        stage[16] <= stage[17];
        stage[17] <= stage[18];
        stage[18] <= stage[19];
        stage[19] <= stage[20];
        stage[20] <= stage[21];
        stage[21] <= stage[22];
        stage[22] <= stage[23];
        stage[23] <= stage[24];
        stage[24] <= stage[25];
        stage[25] <= stage[26];
        stage[26] <= stage[27];
        stage[27] <= stage[28];
        stage[28] <= stage[29];
        stage[29] <= stage[30];
        stage[30] <= stage[31];
        stage[31] <= stage[32];
        stage[32] <= stage[33];
        stage[33] <= stage[34];
        stage[34] <= stage[35];
        stage[35] <= stage[36];
        stage[36] <= stage[37];
        stage[37] <= stage[38];
    end
end

stage[38] <= stage[39];
stage[39] <= stage[40];
stage[40] <= stage[41];
stage[41] <= stage[42];
stage[42] <= stage[43];
stage[43] <= stage[44];
stage[44] <= stage[45];
stage[45] <= stage[46];
stage[46] <= stage[47];
stage[47] <= stage[48];
stage[48] <= stage[49];
stage[49] <= stage[50];
stage[50] <= stage[51];
stage[51] <= stage[52];
stage[52] <= stage[53];
stage[53] <= stage[54];
stage[54] <= stage[55];
stage[55] <= stage[56];
stage[56] <= stage[57];
stage[57] <= stage[58];
stage[58] <= stage[59];
stage[59] <= stage[60];
stage[60] <= stage[61];
stage[61] <= stage[62];
stage[62] <= stage[63];
stage[63] <= stage[64];
stage[64] <= stage[65];
stage[65] <= stage[66];
stage[66] <= stage[67];
stage[67] <= stage[68];
stage[68] <= stage[69];
stage[69] <= stage[70];
stage[70] <= stage[71];
stage[71] <= stage[72];
stage[72] <= stage[73];
stage[73] <= stage[74];
stage[74] <= stage[75];
stage[75] <= stage[76];
stage[76] <= stage[77];
stage[77] <= stage[78];
stage[78] <= stage[79];
stage[79] <= stage[80];
stage[80] <= stage[81];
stage[81] <= stage[82];
stage[82] <= stage[83];
stage[83] <= stage[84];
stage[84] <= stage[85];
stage[85] <= stage[86];
stage[86] <= stage[87];
stage[87] <= stage[88];
stage[88] <= stage[89];
stage[89] <= stage[90];
stage[90] <= stage[91];
stage[91] <= stage[92];
stage[92] <= stage[93];
stage[93] <= stage[94];
stage[94] <= stage[95];
stage[95] <= stage[96];
stage[96] <= stage[97];
stage[97] <= stage[98];
stage[98] <= stage[99] ^ stage[0];
stage[99] <= stage[100];
stage[100] <= stage[101] ^ stage[0];
stage[101] <= stage[102];
stage[102] <= stage[103];
stage[103] <= stage[104];
stage[104] <= stage[105];
stage[105] <= stage[106];
stage[106] <= stage[107];
stage[107] <= stage[108];
stage[108] <= stage[109];
stage[109] <= stage[110];
stage[110] <= stage[111];
stage[111] <= stage[112];
stage[112] <= stage[113];
stage[113] <= stage[114];
stage[114] <= stage[115];
stage[115] <= stage[116];
stage[116] <= stage[117];
stage[117] <= stage[118];
stage[118] <= stage[119];
stage[119] <= stage[120];
stage[120] <= stage[121];
stage[121] <= stage[122];
stage[122] <= stage[123];
stage[123] <= stage[124];
stage[124] <= stage[125];
stage[125] <= stage[126] ^ stage[0];
stage[126] <= stage[127];
stage[127] <= stage[0];
endmodule