FAST, SECURE, AND REMOTE MULTIBOOT OF LOW-COST FPGAS

A Thesis

by

Abdullah Yıldız

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Electrical and Electronics Engineering

Özyeğin University
September 2012

Copyright © 2012 by Abdullah Yıldız

FAST, SECURE, AND REMOTE MULTIBOOT OF LOW-COST FPGAS

Approved by:

Asst. Prof. H. Fatih Uğurdağ, Advisor
Department of Electrical and Electronics Engineering
Özyeğin University

Assoc. Prof. Sezer Gören Uğurdağ, Advisor
Department of Computer Engineering
Yeditepe University

Asst. Prof. Barış Aktemur
Department of Computer Science
Özyeğin University

Asst. Prof. Gürhan Küçük
Department of Computer Engineering
Yeditepe University

Asst. Prof. İsmail Arı
Department of Computer Science
Özyeğin University

Date Approved: 5 October 2012

To my family,

whose encouragement and blessing

are

the reason of who I am today

ABSTRACT

The purpose of this thesis is to develop an efficient framework to implement secure FPGA-based (*Field Programmable Gate Array*) systems. An FPGA is a reconfigurable device that has the ability to adapt the hardware during runtime by loading a new circuit on the reconfigurable fabric. However, a circuit design formed as *configuration data* (*bitstream*) can be easily counterfeited and needs to be protected against the risks of cloning, overbuilding, and reverse-engineering. Although many applications could be implemented on low-cost FPGAs, protection schemes and dedicated hardware are mostly available on high-end FPGAs. In addition to this, only high-end FPGAs support *dynamic partial self reconfiguration* (*DPSR*), which is the ability to change a part of a design at runtime. This thesis focuses on developing a security scheme leveraging hardware intrinsic features on low-cost FPGAs by using *physical unclonable functions* (*PUFs*). A PUF provides a way to extract security keys which are unique to each device. This thesis combines PUFs with another security scheme called *obfuscation*. Obfuscation is the act of intentionally modifying the description or structure of a circuit in order to conceal its functionality. Obfuscation is implemented in this thesis at RTL-level and is used to authenticate and control the device by using the keys by exploiting the PUF technique within a finite state machine (FSM). These methods are further used to implement "secure MultiBoot". The MultiBoot feature allows to reconfigure the FPGA fully at runtime as opposed to DPSR for devices which do not support partial reconfiguration. This thesis also establishes a framework that enables secure remote MultiBoot. A bitstream compression technique is applied to reduce the transmission time over the network. A proof-of-concept example is implemented using the proposed framework.

ÖZETÇE

Bu tez çalışmasının amacı, sahada programlanabilir kapı dizisi (*Field Programmable Gate Array* ya da *FPGA*) tabanlı güvenli sistemler meydana getirmek için verimli bir iskelet yapı oluşturmaktır. Bir FPGA, çalışma zamanı sırasında donanıma yeni bir yapılandırma verisi yüklenerek donanımını yeniden uyarlayabilme yeteneğine sahip bir tür yeniden yapılandırılabilir cihazdır. Fakat, yapılandırma verisi (*bitstream*) kolaylıkla taklit edilebilir ve kopyalama, üzerine ekleme, ya da tersine mühendislik gibi tehlikelere karşı korunması gerekmektedir. Birçok uygulama düşük maliyetli FPGA'ler kullanılarak gerçeklenebildiği halde, bahsedilen tehlikelere karşı koruyucu yöntemler ve özel donanımlar çoğunlukla üst seviye FPGA cihazlarında mevcuttur. Ayrıca, sadece üst seviye FPGA'ler *Dinamik Kısmi Kendi Kendine Yeniden Yapılandırma* (*DPSR*) özelliğini desteklemektedir. DPSR, FPGA'in çalışırken donanımının bir kısmını belli durumlarda kendi kendine değiştirmesidir.

Bu tezde, *fiziksel klonlanamaz fonksiyonlar (PUF)* kullanılarak düşük maliyetli FPGA aygıtları için, donanımın doğasında bulunan ve donanımın üretimi esnasında meydana gelen karakteristik varyasyonlardan faydalanarak bir güvenlik şeması geliştirilmesi üzerinde durulmuştur. PUF, her bir cihaza özgü seri numaraları ve güvenlik anahtarları elde etmeyi sağlamaktadır. Bu tezde, PUF ve *bulandırma* adlı bir başka güvenlik yöntemi birleştirilerek sistemin taklit edilebilirliğini önlemek amaçlanmıştır. Bulandırma, bir devrenin tanımını ya da yapısını kasıtlı olarak değiştirerek çalışma zamanı sırasında işlevselliğini gizlemek üzerinedir. Bulandırma tekniği, RTL (Register - Transfer Language) seviyesinde gerçekleştirilmiştir ve PUF yöntemini bir sonlu

durum makinesi içerisinde kullanarak cihazı yetkilendirmek ve kontrol etmek için kullanılmıştır. Daha sonra bu yöntemler güvenli çoklu yükleme (MultiBoot) tekniği ile birleştirilmiştir. MultiBoot, DPSR'dan farklı olarak, FPGA'in çalışırken tamamen yapılandırılmasını sağlamaktadır. Bunun yanı sıra, sisteme uzaktan bağlanabilme özelliği eklenerek sistemin ağ üzerinden denetimi sağlanmıştır. Ağ üzerindeki iletim zamanını azaltmak için bir bitstream sıkıştırma tekniği kullanılmıştır. Sonuç olarak, bu tez düşük maliyetli FPGA'li sistemlerde güvenli uzaktan MultiBoot için bir iskelet yapı oluşturmaktadır. Önerilen iskelet yapı örnek bir uygulama kullanılarak gerçeklenmiştir.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to Professors Sezer Gören Uğurdağ and H. Fatih Uğurdağ for their assistance in the preparation of this manuscript and also for their patience during the thesis work. They brought in a fresh viewpoint both as an engineer and academic to my career. I also would like to thank Özgür Özkurt for his contributions to this thesis. Thanks also to the members of the school council for their valuable input.

TABLE OF CONTENTS

DEDICATION . . . . **iii**

ABSTRACT . . . . **iv**

ÖZETÇE . . . . **v**

ACKNOWLEDGEMENTS . . . . **vii**

LIST OF TABLES . . . . **x**

LIST OF FIGURES . . . . **xi**

I INTRODUCTION **1**

1.1 Problem Statement 1

1.2 Contributions of the thesis 7

1.3 Overview 9

II FPGA DESIGN FLOW **10**

2.1 Xilinx Spartan-6 FPGA Configuration Sequence 14

2.2 Xilinx Spartan-6 FPGA MultiBoot Design Flow 16

2.2.1 IPROG Command and ICAP 19

2.2.2 MultiBoot using ICAP 20

2.2.3 SPI and BPI Configuration Modes 22

III PHYSICAL UNCLONABLE FUNCTIONS **25**

3.1 Silicon-based PUFs 26

3.2 An FPGA-specific PUF Design: Anderson's PUF 30

3.3 Our Modification of Anderson's PUF 36

3.3.1 Multi-bit PUF Signature Extraction 39

IV PUF KEY-BASED ACTIVE HARDWARE OBFUSCATION . . . **42**

V PROPOSED METHODOLOGY **48**

5.1 Building Blocks of the Proposed Framework 51

5.1.1 PUF Key Generator Design 51

5.1.2 Base Design 52
5.1.3 Obfuscated Design 55
5.1.4 Bitstream Compression 56
5.1.5 Server 59
5.2 Implementation of Proposed Framework 59
VI CONCLUSION **63**
APPENDIX A — SLICEM DETAILS XILINX VIRTEX-5 VS. SPARTAN-6 . . . . **65**
APPENDIX B — CUSTOM FSM GENERATORS (AS PERL SCRIPTS) **67**
APPENDIX C — XILINX SPARTAN-6 CONFIGURATION DETAILS **80**
APPENDIX D — GLOSSARY **82**
REFERENCES **85**
VITA **88**

LIST OF TABLES

1 Sync Word sequence for x8 (8–bit) and x16 (16–bit) modes. 15

2 User-selectable cycle of startup events. 16

3 ICAP_SPARTAN6 port descriptions. 20

4 Example sequence of commands for MultiBoot over ICAP interface. . 22

5 Comparison of slice utilization between original and obfuscated FSM. 47

6 Comparison of BitGen-compressed and double-compressed techniques for various application bitstreams. 58

7 Hamming distance of 64-bit PUF keys on five Spartan-6 devices. . . . 61

8 Comparison of resource utilization between original and obfuscated designs. 62

9 Spartan-6 FPGA configuration modes. 80

10 Spartan-6 FPGA family IDCODE values. 81

LIST OF FIGURES

1	Number of incidents reported by semiconductor businesses.	3
2	Most counterfeited semiconductors in 2011.	4
3	Downloading configuration data into FPGA over configuration interface.	6
4	An abstract view of an SRAM-based FPGA.	12
5	Typical FPGA design flow.	13
6	Basic MultiBoot framework.	17
7	MultiBoot cycle.	18
8	Xilinx Spartan-6 ICAP interface signals.	20
9	SPI bus with one master and one slave device.	23
10	FPGA configuration interface for Master Serial/SPI mode.	23
11	FPGA configuration interface for Master SelectMAP/BPI mode.	24
12	Arbiter PUF.	27
13	Ring oscillator PUF.	28
14	Butterfly PUF.	28
15	An example of common logic block and switch block.	30
16	Routing asymmetry of wires.	31
17	Xilinx Virtex-5 logic block architecture.	32
18	Block diagram of Anderson's PUF in Xilinx Virtex-5.	33
19	N2 signal captured by a D flip-flop to generate the PUF signature bit.	34
20	PUF bit generator rearranged.	35
21	Register/Latch configuration in a slice of Virtex-5.	37
22	Register/Latch configuration in a slice of Spartan-6.	38
23	N2 signal captured by a D flip-flop to generate the PUF signature bit.	38
24	Glitch capture circuit by a D flip-flop in a SLICEM of Spartan-6.	39
25	Row and column relationship between CLBs and Slices in Spartan-6.	40
26	Multi-bit PUF signature extraction.	41
27	Basic diagram for an obfuscated FSM.	43

28 Multi-bit PUF signature defines initial power-up state and state transition passkeys. 46
29 Proposed framework. 49
30 Block diagram of PUF key generator design. 51
31 Block diagram of base design. 53
32 Memory organization in Flash memory. 54
33 "Double-compressed" bitstream using proposed technique. 57
34 Reference client system. 60
35 Block diagram of SLICEM in Xilinx Virtex-5. 65
36 Block diagram of SLICEM in Xilinx Spartan-6. 66

CHAPTER I

INTRODUCTION

In this chapter, we intended to give a comprehensive introduction about the purpose of the thesis with the reason and the motivation behind concentrating on this topic.

1.1 Problem Statement

This thesis work aims at implementing an efficient and combined methodology to make applications more secure and powerful which exploit low cost FPGA-based systems.

One of the motivations of this thesis is the increasing importance of reconfigurable devices in global electronics market. Although deciding an optimal hardware platform solution requires to compare criteria concerning costs, tool availability, and effectiveness, time-to-market comes often at the top of the list. Today verification stage spans most of the time in any design cycle and in general, time pressure is the main reason for poor verification effort. As a consequence, poor design verification practices manifest themselves in unexplained system crashes and delayed product releases. Therefore, FPGAs as a reconfigurable device offer the fastest solution among other hardware options most of the time. Also, when FPGA is chosen as a main platform, the cost effect of nonrecurring engineering minimizes. Moreover, modern FPGAs come with phase-locked loops (PLLs), low-voltage differential signal (LVDS), clock data recovery, high speed, hardware multipliers, dedicated memory, programmable I/O, IP (Intellectual Property) blocks and even microprocessor cores. This provides

a better way to implement cost-effective and high-speed applications. For example, a part of an application which requires high speed audio processing could be implemented on dedicated DSP blocks on FPGA die. Then, dedicated Ethernet PHY IP could be used to connect to the network for the sake of power consumption and performance. Finally, the hard microprocessor core could be utilized for performing decision-making control tasks relatively in slower clock frequencies.

The secondary motivation of this thesis is the increasing significance of mobile and smart electronic devices extensively more than ever. Today, energy-saving products control homes and workplaces. Many public transport and traffic systems are controlled via sensors and managed by centralized computers. Cars are equipped with infotainment systems which provide driver-free functionality such as self-parking and connectivity throughout Wi-Fi or cellular network. Recent estimations show that more than 50 billion devices will be connected by 2020. As people become more dependent to these devices, it is inevitable to make these platforms more reliable and secure.

Today's semiconductor business model separates design and manufacturing phases by using offshore manufacturing fabrication plants (foundries) around the world. This makes the design company rather out of control during manufacturing period. From a security perspective, this brings out many issues for manufacturers, design houses, 3rd party vendors, and end-users.

Today one of the biggest problems that semiconductor companies encounter is *counterfeiting*. Counterfeiting refers to the activity of building and marketing of phony semiconductor devices and covers every phase of the semiconductor supply chain from design stage to market. Figure 1 shows graphically the number of counterfeiting incidents between 2006 and 2011.

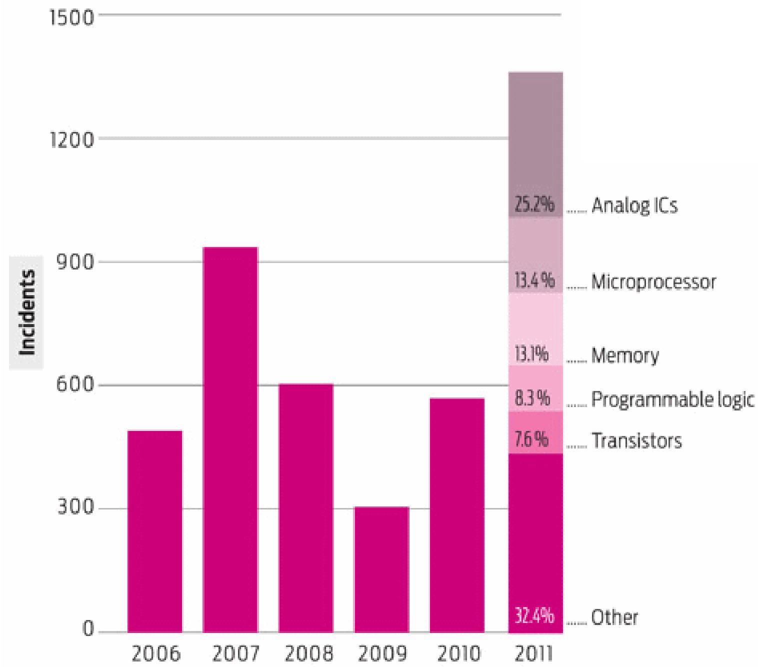


Figure 1: Number of incidents reported by semiconductor businesses [1].

The most encountered counterfeiting activities are achieved by using one of the following methods:

- *Cloning* is the method of directly copying a design, circuit or device without modifying or improving it. This provides the counterfeiters with direct replacement of the product with fast time-to-market. In addition to this, counterfeiters can offer lower cost products than the original ones.

- *Overbuilding* is the method of adding extra functionality into the device or product, in contrast to cloning. However, overbuilding offers better profits and lower product costs than cloning provides. Today, overbuilding is by far the most common counterfeiting method that takes place. With regard to original equipment manufacturer (OEM) or company, overbuilding becomes a big issue since it does not have an idea about what has been added to the original one.

As a consequence, this makes management and support of the product in the market more difficult.

- *Reverse Engineering* is different from both cloning and overbuilding, as it requires engineering investment, technical work and development time and cost. In a common sense, reverse engineering refers to the work that at which a system, a product or a device is deeply analyzed by tampering it using sophisticated instruments and qualified knowledge.

The biggest effect which brings out with these counterfeiting methods is that the result can badly hit the company's market potential leaving them with greatly reduced income and reliability.

Today the five most commonly counterfeited semiconductor types are analog integrated circuits (ICs), microprocessors, memory ICs, programmable logic devices and transistors, all of which are commonly used in commercial and military applications, as shown in Figure 2 [2].

Rank	Commodity Type	% of Reported Incidents
#1	Analog IC	25.2%
#2	Microprocessor IC	13.4%
#3	Memory IC	13.1%
#4	Programmable Logic IC	8.3%
#5	Transistor	7.6%

Figure 2: Most counterfeited semiconductors in 2011 (% of counterfeit reports) [2].

This conclusion is particularly important for designers, manufacturers and users of programmable logic devices such as FPGAs. In many cases, these devices make

it possible to reprogram the functionality of a system in the field and to transfer circuitry and IP from one system to another simply by copying the configuration information. Without adequate protection against such situations, an FPGA cannot provide effective design or data security. As functionality is encapsulated in firmware (bitstream) and reconfigurable circuits that may be unlocked at runtime by providing a key, design security becomes increasingly important to data security.

FPGAs are the most important and preferred type of reconfigurable devices in electronic systems today. Furthermore, the performance and cost disadvantage gap compared previously to ASIC (Application Specific Integrated Circuit) or ASSP (Application Specific Standard Product) devices are negligible or even better in some cases. Hence, it will be not surprising to use electronic products that have at least one programmable SoC (System-on-Chip) or more in the near future.

What puts FPGAs forward with respect to design security is the availability of bitstream stored in an off-chip memory. Figure 3 represents a basic FPGA system which consists of a configuration interface and an external configuration memory. As shown in Figure 3, whenever a new configuration needs to be downloaded into FPGA, the bitstream on the configuration memory is read over the configuration interface and downloaded into the FPGA device. However, during transfer operation, one can easily intercept the communication and capture the bitstream in behalf of copying and implementing the same design on another FPGA. Hence, it is required to use a protection mechanism such that a design should be able to work only on a specific set of devices in order to prevent such attacks.

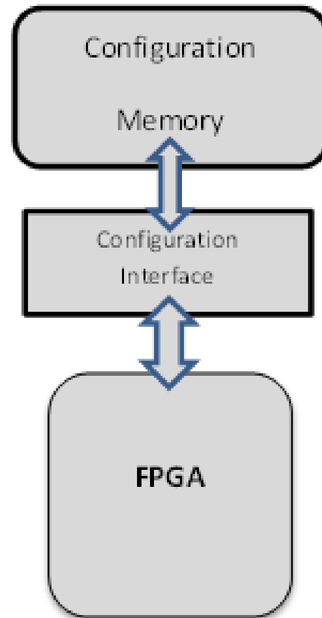


Figure 3: Downloading configuration data into FPGA over configuration interface.

Today, solutions exist on the market targeting FPGA design security. In general, FPGA manufacturers handle this problem by equipping their products with factory-set ID numbers, encryption/decryption logic or advanced cryptographic IP blocks. There are also 3rd party solutions [3, 4] which offer similar solutions.

Although encryption support seems a powerful technique to protect FPGA designs, a recent study [5] showed that it is possible to extract secret keys from an FPGA where the bitstream encryption feature of the device is enabled.

Xilinx [6] offers a feature called "Device DNA" [7], available on Xilinx Spartan-6 FPGAs, which is a per-chip ID code that has a fixed factory-set number which can be extended by supplementary data bits. Hence, one can use a DNA-based authentication to at least prevent cloning (if not reverse engineering). If the design includes a module that checks the device's DNA with the pre-stored DNA value in the design, then the same bitstream would not function on a different device as the DNA value

in the copied bitstream would not match the new device's DNA. Instead of a direct comparison, the key inside the bitstream can be a scrambled version of the DNA of the authorized device. Nevertheless, designs with Device DNA can be cloned using the following methodology:

- Read the DNA from the device that you want to clone.
- Copy the bitstream from the Flash memory.
- Reverse engineer it into a netlist [8].
- Locate where the DNA hooks up to the design.
- Disconnect the DNA port and hard-wire it to the DNA values (apply step 1).
- Convert the new netlist to a bitstream.
- Copy it to as many devices as you like, and it will work although they have unauthorized DNA codes as the modified bitstream bypasses the DNA.

Most of these security techniques are mostly available for high-end family of FPGA devices [9, 10, 11]. However, many applications could be designed and implemented on low-cost FPGAs instead of using these expensive devices equipped with advanced protection schemes.

1.2 Contributions of the thesis

In this thesis, we propose a security framework that enables secure remote reconfiguration of low-cost FPGAs. Our framework combines two important security schemes. The first scheme deploys *hardware intrinsic features* of a device to realize a kind of security shield against mentioned attacks above. More specifically, this technique is

also known as *physical unclonable function (PUF)* [12, 13, 14, 15, 16, 17, 18, 19] and provides a way for utilizing the inherent uniqueness in each silicon device to extract unique ID numbers or more appropriately keys. Today PUF technology attracts attention by several design companies and semiconductor manufacturers [20, 21, 22, 23] in the market.

The second scheme that we utilized in our proposed framework is the *hardware obfuscation technique* [24, 25, 19]. Hardware obfuscation is locking the functionality of the design. An obfuscated design could only be unlocked when a specific sequence of events have occurred. Although obfuscation received skepticism [26] in the past, several works [24, 19, 27] have shown the feasibility of secure key-based obfuscation. Recently, Koushanfar [19] demonstrated proofs for developing secure integrated circuit (IC) control mechanism with the functional description of the design as well as unique and unclonable IC identifiers. In an earlier work [28], we combined PUF key-based obfuscation and MultiBoot feature of Spartan-6 devices to achieve full bitstream protection. In MultiBoot, FPGA has to overwrite its configuration completely and externally from a Flash memory.

Another aspect this thesis focuses on is reconfiguration of low-cost FPGAs. It is common that today's systems have to support many standards. That results in situations some of the features are not needed and new features are required to be up and running. This behavior can be efficiently implemented with *Dynamic Partial Self Reconfiguration (DPSR)*, where an up and running FPGA decides what subdesigns to download from the Flash memory (i.e., new features) and reclaims resources taken up by subdesigns not needed anymore (i.e., old features).

This thesis targets low-cost FPGAs that do not have advanced protection and security schemes. Hence, we propose a security framework by combining PUF and obfuscation techniques. Additionally, we leverage a MultiBoot feature to enable reconfiguration of

FPGAs over the network in a secure way. This thesis uses the following technologies to achieve the equivalent of DPSR with encryption [29]:

- PUF→ in place of DNA –as a security key
- HDL-level obfuscation → in place of encryption
- MultiBoot → in place of DPSR

1.3 Overview

The remainder of this thesis is organized as follows: *Chapter 2* gives an introduction about generic FPGA design flow, configuration details and how a MultiBoot operation takes place within FPGA. *Chapter 3* introduces PUF methodology, explains previously proposed PUF structures and describes our proposed PUF technique. *Chapter 4* gives a brief introduction to obfuscation and then describes our proposed security methodology which combines PUF and hardware obfuscation. *Chapter 5* represents our security framework which enables fast, secure, and remote MultiBoot for low-cost FPGAs. Conclusions and suggestions for future work are given in *Chapter 6*.

CHAPTER II

FPGA DESIGN FLOW

This chapter covers topics which provide a technical background for the material used in this thesis work. At first, most fundamental features of FPGAs will be introduced. Then FPGA MultiBoot embedded design flow for Xilinx Spartan-6 FPGAs is explained in detail.

FPGA is a type of IC designed to configure its logical functionality after manufacturing. An FPGA can be used to implement any logical function that an ASIC, a microprocessor or other types of devices could perform. One of the biggest advantages of FPGAs is that they have the benefits of both hardware and software. From a hardware perspective, FPGAs implement circuits providing power, area, and performance benefits. From a software perspective, FPGAs implement circuits that can be reprogrammed cheaply and can be used to perform a wide range of tasks. In this way, it is possible to design more efficient systems than both ASIC and microprocessor based platforms.

Since early ages of FPGAs, they were fabricated via various manufacturing technologies as antifuse, Flash, SRAM, and Flash-SRAM hybrid. Devices based on antifuse technology are one-time programmable (OTP) and use a dedicated antifuse with each configuration cell. In its unprogrammed state, an antifuse has such a high resistance to make its wire open circuit. Programming of device is done by applying pulses of relatively high voltage and current on particular antifuses, thereby lowering resistance on those wires to implement a particular circuit.

Flash-based devices are nonvolatile, so they can keep their configuration data after power is cut off. Additionally, they can be reprogrammed again. Flash-SRAM hybrids contain both Flash and SRAM configuration cells. One disadvantage of both antifuse and Flash-based FPGAs is that their fabrication requires additional process steps in addition to CMOS process.

Today SRAM-based FPGAs are the dominant type among reconfigurable platforms. They can be fabricated using a pure CMOS process hence they can benefit from the latest manufacturing technology and can offer higher capacities and higher performance. The distinctive feature of SRAM-based FPGAs than others is that they are reprogrammed every time power is up.

Figure 4 illustrates the internal organization of a general SRAM-based FPGA. Herein, reconfigurable area is divided into two main categories as logic and interconnect. Logic blocks are spread through the silicon die and contain processing elements for implementing both combinational and sequential logic. Each logic block generally contains LUTs (look-up table), multiplexers, and FFs (flip-flop). The general routing structure is also spread equally on silicon die and provides the connection and wiring between these logic blocks. By appropriate programming of device by utilizing the particular wires and logic blocks, any boolean function with different complexity could be implemented on an FPGA.

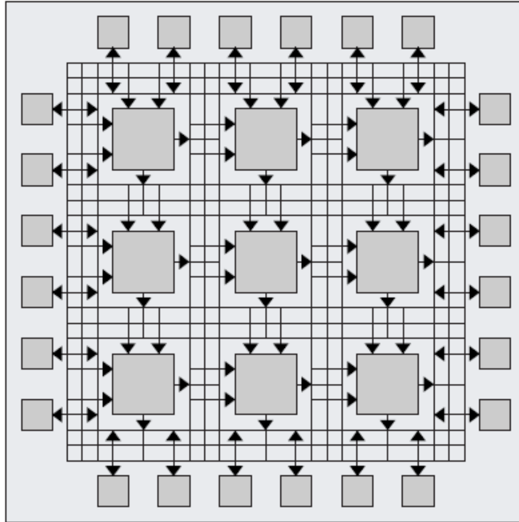


Figure 4: An abstract view of an SRAM-based FPGA [30].

A typical FPGA design flow is organized as shown in Figure 5. *Design Entry* represents an abstract description of circuit design which is usually defined using an HDL (Hardware Description Language) like Verilog, VHDL, etc. *Logic Synthesis* step takes design entry as input and converts to logic gate equivalents as output. This process is followed by *Technology Mapping* which deals with utilizing FPGA's logic resources by using the output from logic synthesis step. Then *Placement* step determines which specific logic blocks are allocated to be used by design and *Routing* step utilizes the specific interconnect resources which provide signal and data transmission between instances of design.

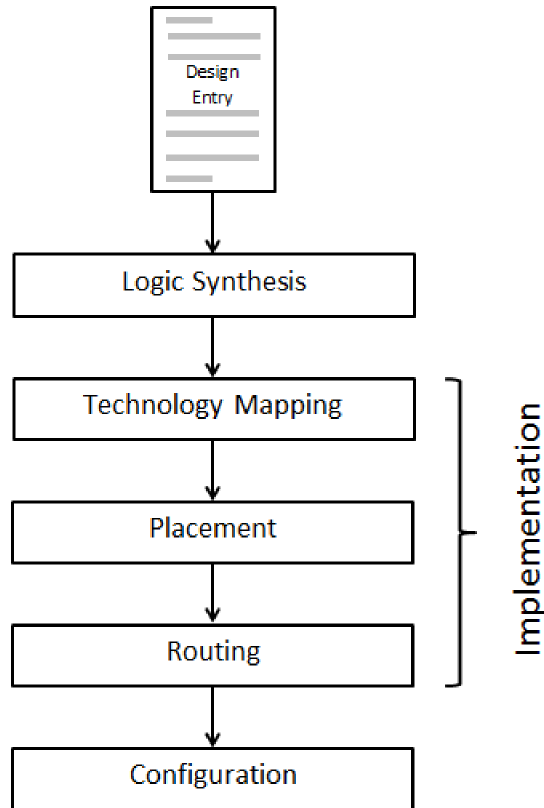


Figure 5: Typical FPGA design flow.

Technology mapping, placement, and routing could be grouped as *Implementation* stage as these steps generate a netlist called NCD (Native Circuit Description). A netlist is a kind of context that contains specific data for a selected device, technology, and all required circuit information as connectivity, instances of design, attributes, etc. *Configuration* step takes this netlist and then translates it to generate a binary file called *Bitstream*. A bitstream contains utilization data for each logic and routing element present on FPGA as well as initialization commands to the FPGA configuration logic. Programming of FPGA is done by configuring memory locations assigned to each of these logic and routing elements.

2.1 *Xilinx Spartan-6 FPGA Configuration Sequence*

Each configuration operation in Xilinx Spartan-6 FPGAs starts with a few steps which initialize the device and perform some operations to make it functional before downloading and running the configuration. Control of the operations are managed by an on-chip state machine and a set of configuration registers. On-chip state machine is a configuration controller which takes bitstream and processes it by utilizing configuration memory and configuration registers. Configuration registers hold some control and status information as well as keep configuration commands and data processed by the state machine.

Configuration sequence starts by supplying the device with appropriate level of voltages at specific device pins. Once the device is powered up, configuration memory is cleared consecutively and mode pins (See Appendix C) are sampled when INIT_B signal transitions to logic-1. Before sampling the configuration data, there is a period of time that FPGA needs to synchronize and align its internal configuration logic with upcoming configuration data. Thus, a synchronization word/bus detection pattern is processed first. Then, device becomes ready to capture the configuration data on the rising edge of the clock signal.

As shown in Table 1, synchronization word consists of 32 bits or 4 bytes: 0xAA, 0x99, 0x55, 0x66. Bus-width detection logic checks these incoming bytes on input pins. For 8–bit configuration, bus–width detection logic first finds 0xAA on the D[0:7], followed by 0x99, 0x55, and 0x66. For 16–bit configuration, bus–width detection logic checks the first byte to find 0x99 on D[0:7], followed by 0x66 the next cycle. The rest of the sync word is received on the upper bits. Then FPGA is ready on which bus width to receive the rest of the data.

Table 1: Sync Word sequence for x8 (8–bit) and x16 (16–bit) modes [7].

Cycle	*First*	*Second*	*Third*	*Fourth*
x8 Mode D[7:0]	0x55	0x99	0xAA	0x66
x16 Mode D[15:0]	0x5599	0xAA66	N/A	N/A

After synchronization of device, a device ID check is performed to prevent the device from loading faulty configurations. A faulty configuration could be a bitstream that is formatted for a different device. If a device ID check error occurs during configuration, the device attempts to do a fallback reconfiguration. For more information about this step, see Appendix C.

The next step after the synchronization word has been loaded and device ID has been checked is loading of configuration data frames. As the configuration data frames are loaded, the device calculates a Cyclic Redundancy Check (CRC) value. After finishing of loading of configuration data frames, the configuration bitstream can issue a "Check CRC" instruction followed by the expected CRC value. If the expected CRC value does not match the value calculated, then the device does not continue to configuration and abort the process.

After configuration frames are loaded, controller state machine initiates a couple of startup events. These are a sequence of tasks each have to be performed before device starts to run actual design. An eight-phase sequential state machine controls these events. Table 2 shows these events which can be customized by user.

Table 2: User-selectable cycle of startup events [7].

Phase	*Event*
1-6	Wait for DCMs and PLLs to lock (optional)
1-6	Assert Global Write Enable (GWE), allowing RAMs and flip-flops to change state
1-6	Negate Global 3-State (GTS), activating I/O
1-6	Release DONE pin
7	Assert End Of Startup (EOS)

2.2 Xilinx Spartan-6 FPGA MultiBoot Design Flow

Today many systems that use FPGAs may have to support many standards, integrate IP (intellectual property) cores into the system or run various applications. Generally, a fully functioning system at a specific time do not need all of these. Hence, one that uses or employs all these functional units at the same time may need a large chip area. In addition to this, unused blocks can increase power consumption and may lead to performance penalty.

Since FPGAs are reprogrammable in the system, it is possible to update configuration information of device during normal operation. In this way, relatively small-size FPGAs become equivalent to larger and more expensive devices such as ASICs or FPGAs just programmed or configured once.

A variety of methods (download modes) exist to reconfigure FPGAs during normal operation. Through an external *intelligent agent*, such as a computer, processor, microcontroller, or a test device, FPGA can be reprogrammed or reconfigured many times. Xilinx Spartan-6 FPGAs have 5 different configuration modes [7] as *Master Serial/SPI*, *Master SelectMAP/BPI*, *JTAG*, *Slave SelectMAP*, and *Slave Serial*.

Besides these, the reconfiguration process could be automated using a modular-based approach. Some FPGA devices provide a capability named "MultiBoot" which allows FPGA to selectively reprogram itself from an external memory device. Xilinx Spartan-6 FPGAs support MultiBoot operation in SPI (x1, x2, and x4) and BPI configuration modes.

Figure 6 represents a system which exploits MultiBoot. There are N different bitstreams which reside in external memory. When existing FPGA configuration triggers a MultiBoot operation, FPGA reconfigures automatically with a new configuration bitstream.

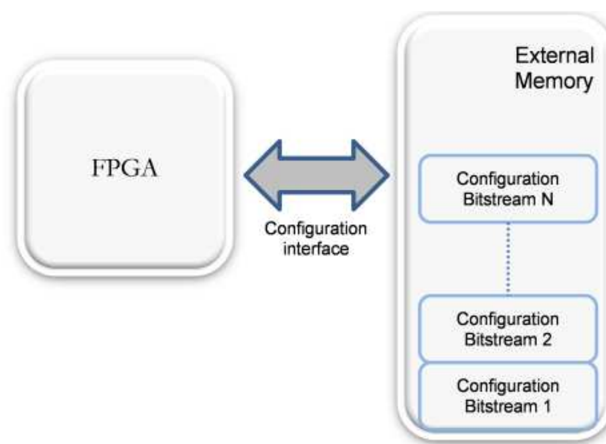


Figure 6: Basic MultiBoot framework.

Xilinx Spartan-6 FPGAs have dedicated MultiBoot logic which is used when a fallback or MultiBoot operation is required. Fallback is a recovery mechanism which could be a critical solution for application updates during MultiBoot operations.

After initiating a MultiBoot reconfiguration, the entire configuration logic is cleared except the dedicated MultiBoot logic and some of the configuration registers. Then, configuration process is restarted by clearing configuration memory and following the similar steps as described in previous section.

Xilinx has a reference MultiBoot method [7] which consists of three images for MultiBoot operation. The first image is called as *Header* and contains the sync word, the commands which set the addresses for the next bitstream and the fallback bitstream. The second image is the configuration that user intends to run first. The third image is the *fallback (golden)* bitstream. The fallback bitstream is a fail safe bitstream in case of an error during configuration. If the configuration fallback occurs and the fallback bitstream is reached, the only way to boot back into the MultiBoot bitstream is to toggle the PROGRAM_B pin or power cycle the device.

Figure 7 shows the fail safe MultiBoot design proposed by Xilinx. Here, MultiBoot logic utilizes a *strike count* which keeps the number of attempts to perform a succesful configuration and is stored in BOOTSTS configuration register. The only way to clear strike count is to perform a hard reboot (pulse the PROGRAM_B pin) or cycle power.

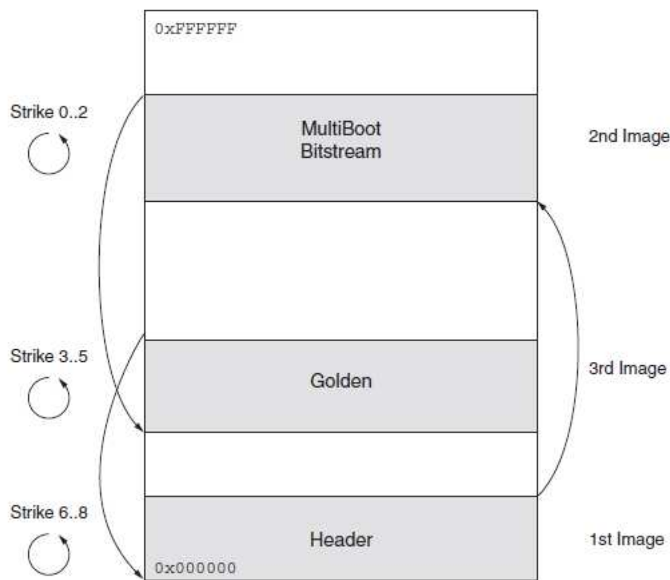


Figure 7: MultiBoot cycle [7].

Accordingly, the header image must start at address 0x000000. If an error is detected or the watchdog timer times out, the strike count increments and configuration restarts if the strike count is less than 3. When the strike count becomes 3,

configuration halts with INIT and DONE signals driven low.

The address location of MultiBoot image is defined by GENERAL1 and GENERAL2 configuration registers. MultiBoot image has also three strikes assigned to it. If an error is detected, the strike count increments and configuration restarts at the address specified in GENERAL1 and GENERAL2 if the count is less than 3. If the count hits 3, configuration moves to the fallback bitstream.

The fallback image can reside in memory locations defined by GENERAL3 and GENERAL4 registers. The fallback image has also 3 strikes allotted to it. If an error is detected, the strike count increments and configuration will restart at the address specified in GENERAL3 and GENERAL4, if the count is less than 6. The value is 6 since it shares the strike counter with the MultiBoot image. If the strike count becomes 6, configuration tries an attempt to boot the header image. Then, MultiBoot logic attempts to access both the MultiBoot image and the fallback image three more times before halting configuration. If fallback reconfiguration fails after three strikes, configuration halts with both INIT_B and DONE driven low.

2.2.1 IPROG Command and ICAP

Every MultiBoot operation requires an *IPROG* command to be issued. IPROG command has similar effect as pulsing PROGRAM_B pin, except that it does not reset the dedicated reconfiguration logic. IPROG command can be issued as embedded in the bitstream or using a dedicated block called *ICAP (Internal Configuration Access Port)* on FPGA. ICAP is an internal configuration interface on FPGA fabric and allows the user to access configuration registers and readback configuration memory after a configuration.

Although not all Xilinx FPGAs contain ICAP primitive, Xilinx Spartan-6 FPGAs

have a dedicated ICAP block called *ICAP_SPARTAN6*. Figure 8 shows block diagram and I/O interface for Xilinx Spartan-6 FPGAs.

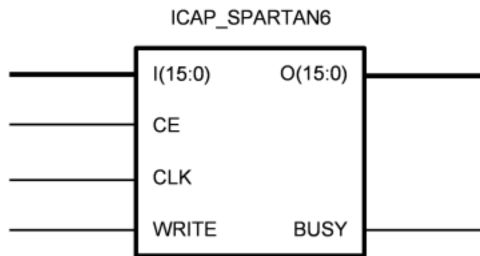


Figure 8: Xilinx Spartan-6 ICAP interface signals.

Table 3 gives definitons for each I/O signal of ICAP_SPARTAN6 primitive.

Table 3: ICAP_SPARTAN6 port descriptions [7].

Port	*Type*	*Width*	*Function*
BUSY	Output	1	Busy/Ready
CE	Input	1	Active–Low ICAP Enable
CLK	Input	1	Clock
I	Input	16	Configuration Data Bus
O	Output	16	Configuration Data Bus
WRITE	Input	1	Read/Write Control

2.2.2 MultiBoot using ICAP

MultiBoot operation in Xilinx Spartan-6 FPGAs is supported in SPI or BPI configuration modes. Therefore, an SPI or BPI supported nonvolatile memory should be employed and configured properly with configuration bitstreams at specific address locations before starting a MultiBoot operation.

MultiBoot operation can be initiated anytime after a successful configuration. It starts with by setting GENERAL1 and GENERAL2 configuration registers. GENERAL3 and GENERAL4 configuration registers are also set if there is a second configuration or fallback bitstream or they keep their default values (0x0000). Here, GENERAL1 and GENERAL3 configuration registers are used to store lower 16-bit part of memory address locations ([15:0]) of MultiBoot and fallback bitstreams, respectively. The upper 8-bit of memory address locations ([23:16]) of MultiBoot and fallback bitstreams are stored respectively in lower byte field of GENERAL2 and GENERAL4 configuration registers. The upper byte fields of GENERAL2 and GENERAL4 are used to store opcode which refers to the read instruction of nonvolatile storage device over SPI or BPI.

After utilizing GENERAL1 and GENERAL2 (optionally GENERAL3 and GENERAL4) configuration registers, ICAP primitive must be synchronized before any command to be sent. Hence, a bus-width detection logic is employed inside ICAP primitive by sending Sync Word (0xAA99 and 0x5566 in subsequent cycles) to synchronize its configuration interface. Following the synchronization step, a command is sent which tells the ICAP state machine to write the next 16-bit data to GENERAL1 configuration register. Similar operations are performed for each GENERAL2, GENERAL3, and GENERAL4 configuration register. After setting all the configuration registers, another command is sent which tells the ICAP state machine to write a special command into CMD register which will be sent the following cycle. The next cycle CMD register is set to IPROG command which starts a sequence of events: first, device is reset and configuration memory is cleared, then configuration bitstream is downloaded from nonvolatile memory over SPI or BPI interface and FPGA is reconfigured. The sequence of commands are given in Table 4. Dummy Word and NO OP (No operation) command have no effect in MultiBoot.

Table 4: Example sequence of commands for MultiBoot over ICAP interface.

$ConfigurationData$	$Definition$
0xFFFF	Dummy Word
0xAA99	Sync Word
0x5566	Sync Word
0x3261	Type 1 Write 1 Words to GENERAL1
0xXXXX	MultiBoot Start Address [15:0]
0x3281	Type 1 Write 1 Words to GENERAL2
0xXXXX	Opcode and MultiBoot Start Address [23:16]
0x32A1	Type 1 Write 1 Word to GENERAL3
0xXXXX	Fallback Start Address [15:0]
0x32C1	Type 1 Write 1 Word to GENERAL4
0xXXXX	Opcode and Fallback Start Address [23:16]
0x30A1	Type 1 Write 1 Word to CMD
0x000E	IPROG Command
0x2000	Type 1 NO OP

2.2.3 SPI and BPI Configuration Modes

The Serial Peripheral Interface Bus or SPI Bus is a synchronous serial data link standard named by Motorola that operates in full duplex mode. There are master and slave operating modes in which devices communicate to each other. Multiple slave devices can be connected at the same time.

As shown in the Figure 9, clock of the slave SPI device is provided by the master. Hence, there is no need for specific handshaking protocols to synchronize the devices. In addition to this, there are two data lines (MOSI and MISO) which allow data

transfer in full duplex mode. An *active-low* slave select signal is used to enable and select the slave device.

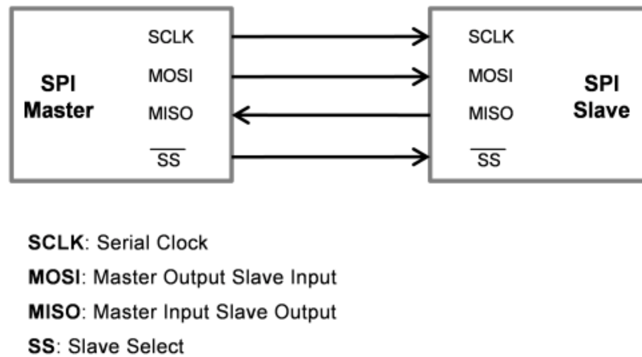


Figure 9: SPI bus with one master and one slave device.

The simple structure of SPI bus without complex data transfer logic provides a convenient way to implement it on an FPGA. A small-sized logic (less than 2-3 % of logic cell number of a generic FPGA with approximately 45K logic cells) could easily handle SPI operations and the rest of the logic could be used for other requirements. Figure 10 shows the configuration interface in SPI mode for MultiBoot.

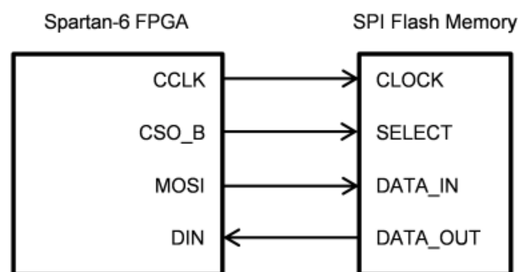


Figure 10: FPGA configuration interface for Master Serial/SPI mode [7].

Similarly, the Byte Peripheral Interface (BPI) Bus could be used to reconfigure an FPGA (See Figure 11). However, the number of configuration interface pins and its

timing specification makes SPI more advantageous over BPI with respect to configuration complexity.

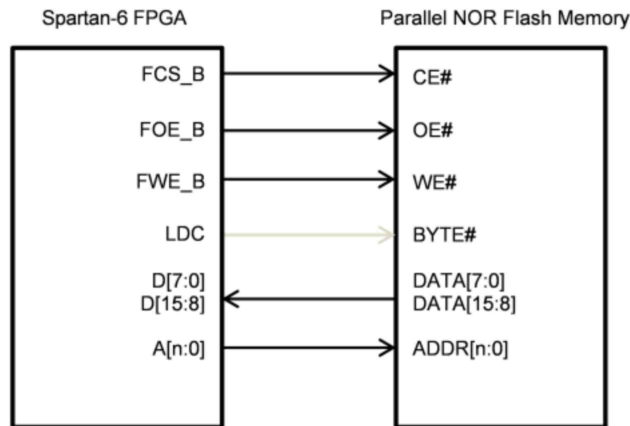


Figure 11: FPGA configuration interface for Master SelectMAP/BPI mode [7].

Put it in a nutshell, MultiBoot is a powerful technique with SPI. A fully functioning system which uses MultiBoot could store configuration bitstreams in an SPI supported memory with only a few megabytes[1].

[1]Our reference system has a 128 Mbit (16 MB) SPI NOR FLASH memory.

CHAPTER III

PHYSICAL UNCLONABLE FUNCTIONS

A physical unclonable function (PUF) is a physical pseudo-random function. It is based on the idea of utilizing intrinsic features of a physical instance. More specifically, a PUF is characterized by unique manufacturing variances that are present within a device. This makes a PUF device practically impossible to duplicate. In this respect, it is the hardware analog of a one-way function. The concept of PUF was first proposed by Pappu et al. [12] as "Physical One-Way Functions". Pappu et al. described the physical one-way function as a function "if there exists a deterministic physical interaction between the probe (challenge) and the system which produces an output in constant time, inverting the function is difficult, simulating the physical interaction is computationally demanding and the physical system is easy to make but difficult to clone". In other words, a PUF is easy to evaluate but hard to predict.

PUFs implement *challenge-response authentication* unlike the cryptographic key-based schemes. When a physical stimulus is applied to the physical structure, it reacts in an unpredictable way due to the complex interaction of the stimulus with the physical microstructure of the device. This exact microstructure depends on physical factors introduced during manufacturing processes which are unpredictable. The applied stimulus is called the *challenge*, and the reaction of the PUF is called the *response*. A specific challenge and its corresponding response together form a *challenge-response pair* or *CRP*.

There are three types of PUFs: optical PUFs [12], coating PUFs [15, 16], and silicon-based PUFs [13]. Optical and coating PUFs exhibit explicitly-introduced randomness while silicon-based PUFs use intrinsic randomness.

3.1 Silicon-based PUFs

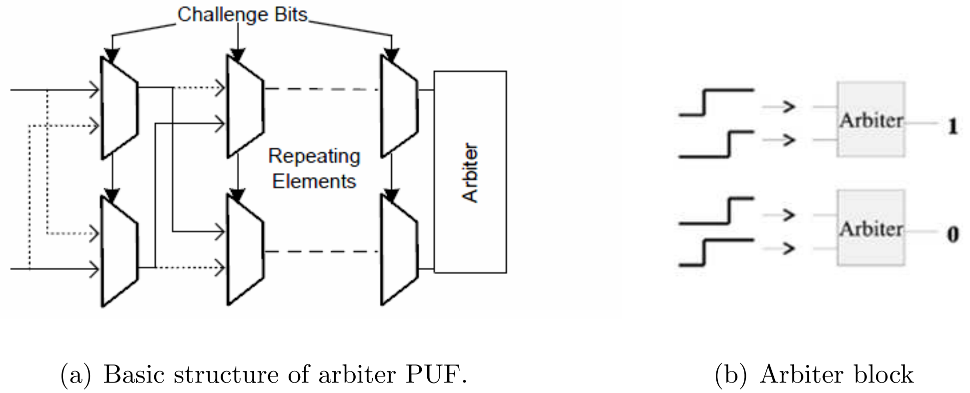
The first integrated PUF on a silicon device was proposed by Gassend et al. [13]. Gassend et al. claimed that a complex integrated circuit could be viewed as a silicon PUF and described a technique to identify and authenticate individual integrated circuits. In addition to this, Tuyls et al.[16] proposed that an implementation of read-proof hardware that is resistant against invasive attacks by employing protecting coating that contains a lot of randomness.

Many PUF implementations proposed by now exploit the intrinsic randomness on the silicon device without modifications to the manufacturing process. As PUFs are implemented by exploiting the small manufacturing variances within a device, for an integrated circuit (IC), PUF could be implemented by utilizing variances of the wire and gate delays even if they are logically identical. Hence, it could be used to represent the *fingerprint* of a particular device.

The most common type of PUFs on integrated circuit devices are *delay based PUFs*. Delay based PUFs consist of at least two logically identical transition paths. Given an input challenge, a race condition is set up in the circuit, and these transitions that propagate along different paths are compared to see which comes first. An arbiter, typically implemented as a latch, produces a 1 or a 0, depending on which transition comes first. When a circuit with the same layout mask is fabricated on different chips, the response is different for each chip due to the random variations of delays.

There are three types of delay based PUFs which are the most common used:

- *Arbiter PUF or APUF* [15] is composed of two identically configured delay paths that are stimulated by an activating signal. The difference in the propagation delay of the signal in the two delay paths is measured by an edge triggered flip-flop known as the *arbiter*. The delay difference is a function of the manufacturing process variation present in the delay paths.



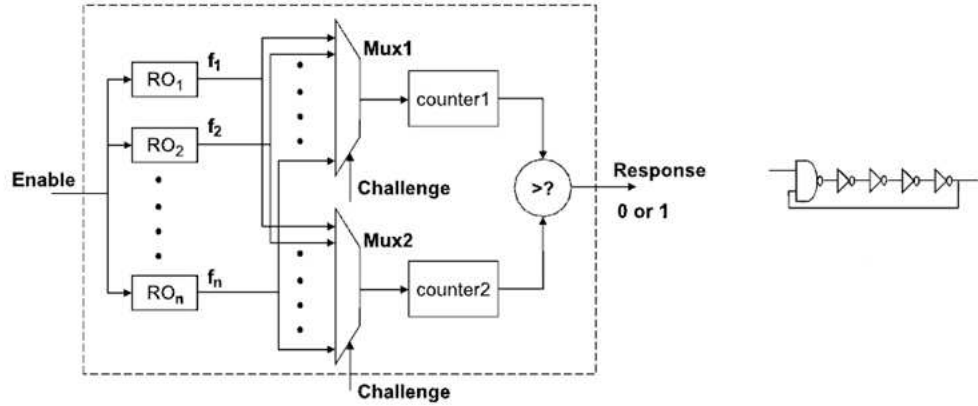
(a) Basic structure of arbiter PUF.

(b) Arbiter block

Figure 12: Arbiter PUF [15].

- *Ring oscillator based PUF or ROPUF* [17, 31] is composed of identically arranged ring oscillators, $\mathrm{RO_1}$ to $\mathrm{RO_n}$, with frequencies $\mathrm{f_1}$ to $\mathrm{f_n}$, respectively and a pair of multiplexers to take inputs from ring oscillator outputs. PUF challenge bits are used as the select inputs to multiplexers. A response bit r is obtained by selecting a pair of frequencies, $\mathrm{f_a}$ and $\mathrm{f_b}$ ($a \neq b$) and using a simple comparison method. Due to process variations, $\mathrm{f_a}$ and $\mathrm{f_b}$ tend to differ from each other and the response bit is defined as follows:

$$r = \begin{cases} 1, & \mathrm{f_a} > \mathrm{f_b}, \\ 0, & \text{otherwise.} \end{cases}$$



(a) Ring oscillator

(b) A basic five-stage ring oscillator loop

Figure 13: Ring oscillator PUF [17, 31].

- *Butterfly PUF or BPUF* [18] circuit defines two cross-coupled latches to generate a response bit. The main idea here is that it is desired to create a race condition temporarily on cross-coupled latches. While circuit goes to stable after some time, the output is either logic 1 or logic 0, depending on the manufacturing process variations.

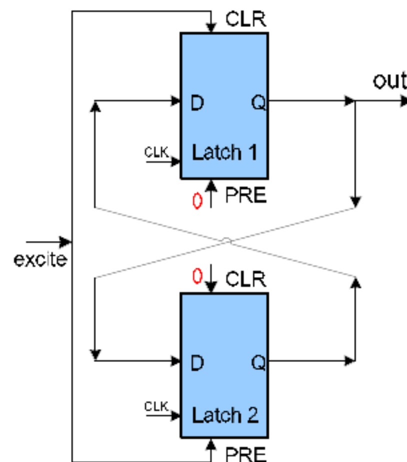


Figure 14: Butterfly PUF [18].

PUF implementations explained above could be used such that several PUF response

bits can be generated by configuring the delay paths in multiple ways using the challenge inputs. So far many security schemes for SRAM-based FPGAs are developed and implemented by using delay-based PUF circuits. Indeed, they are relatively easier to realize with respect to other approaches. However, they have the disadvantage of creating combinational logic on FPGAs since it is not straightforward to create combinatorial paths using available resources (i.e., flip-flops). Another challenge for these PUF approaches is that they assume exact symmetric routing between logical components which is not feasible using current design automation tools. For this reason, it is more convenient to use a PUF structure that specifically targets FPGAs.

As shown in Figure 15, the smallest functional unit in FPGA consists of a logic block which generally includes FFs, LUTs, and multiplexers, etc. and a switch block which provides the signal communication between near and remote logical resources. Although design automation tools allow to place logical resources symmetrically, it is not feasible to create symmetric routing paths between logic elements.

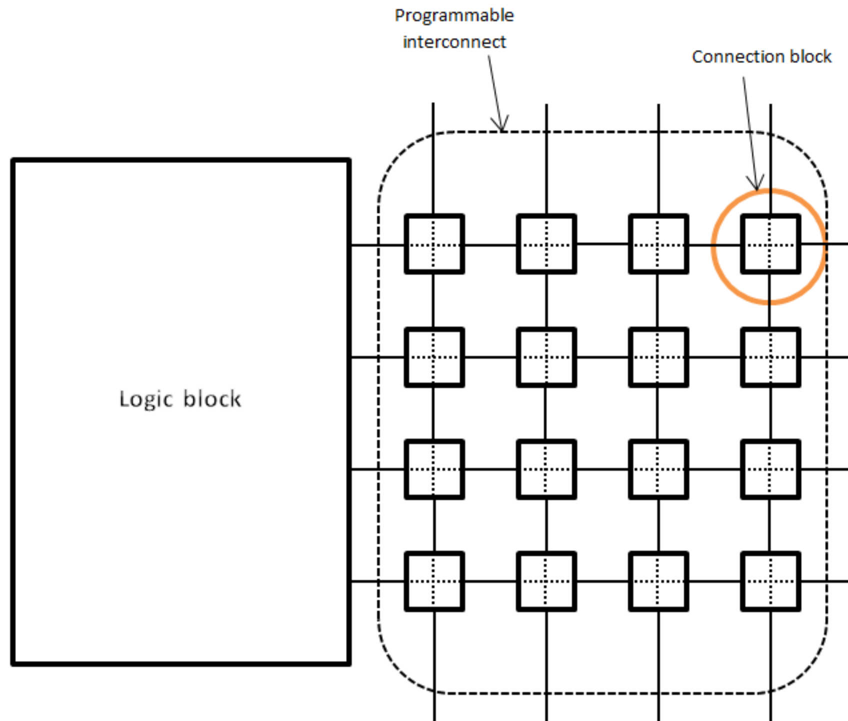


Figure 15: An example of common logic block and switch block.

Another example in Figure 16 represents the top view of a configurable logic block (CLB) on Xilinx Spartan-3 FPGA after a place&route (PR) operation. The thin red lines represent the routing path for a butterfly PUF circuit which are needed to create two cross-coupled latches for a single PUF instance. As can be seen in the picture, apart from the complexity of implementing combinational logic by using sequential logic components (flip-flops), it is not an efficient solution with asymmetric routing paths between FPGA logical resources.

3.2 An FPGA-specific PUF Design: Anderson's PUF

The proposed PUF structures above are designed specifically for silicon devices such as ASICs and other type of integrated circuits. Since FPGAs contain regular arrays of logical elements and interconnect, it is not likely to implement an appropriate

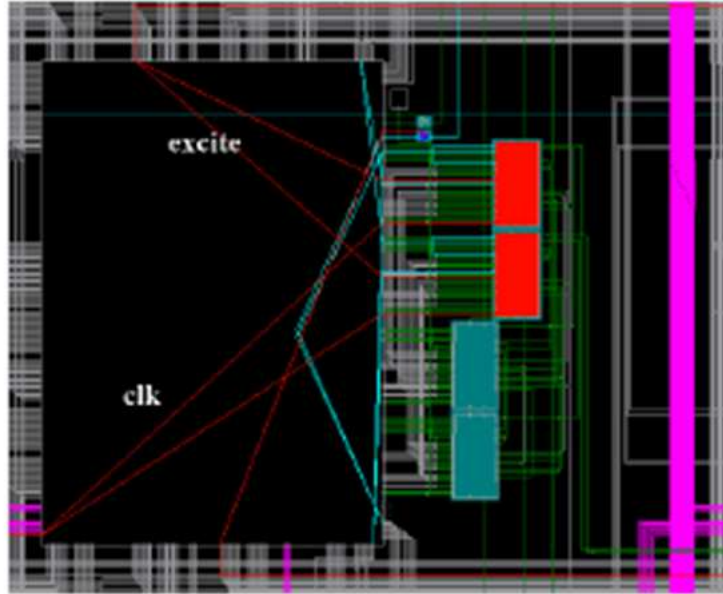


Figure 16: Routing asymmetry of wires in Butterfly PUF.

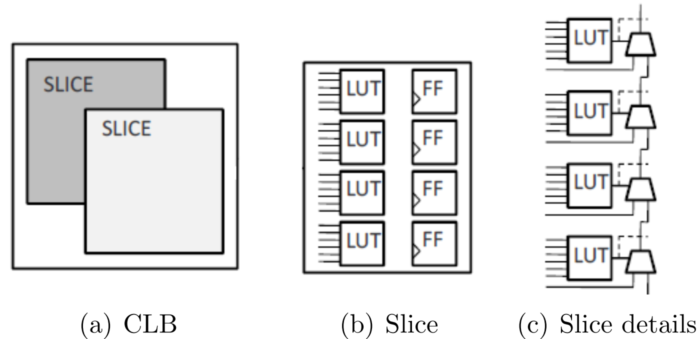
arbiter, ring oscillator or butterfly PUF circuit using logical components along with routing blocks. Instead, a PUF architecture should be employed which benefits from available device resources inside FPGA.

In a recent study, Anderson [14] proposed a PUF design that specifically targets FPGAs. The novelty of this PUF design comes from the fact that it makes use of the underlying FPGA architecture and it could be integrated into a design at RTL level. Anderson [14] claimed that it consumes a very little area and does not require the use of *hard macros* with fixed routing.

Figure 17(a) depicts a configurable logic block or CLB on a Xilinx Virtex-5 FPGA which provides combinatorial and synchronous logic as well as distributed memory and shift register capability. A CLB comprises two slices. Each slice (Figure 17(b)) contains four 6-input LUT (look-up table), four storage elements (FFs), multiplexers, and arithmetic circuitry.

Figure 17(c) represents a slice which is utilized for the proposed PUF design [14]. In

this diagram, each LUT output is connected to the select input of a 2-to-1 multiplexer. Multiplexers are arranged in a "carry chain" style such that each multiplexer receives one of its data inputs from the multiplexer below it.



(a) CLB (b) Slice (c) Slice details

Figure 17: Xilinx Virtex-5 logic block architecture [14].

Although LUTs are mostly used to implement combinational logic functions, it is intended to use them as memory units in the proposed PUF design [14]. Such LUTs reside in SLICEM[1] [32] blocks in Xilinx Virtex-5 where the "M" indicates that LUTs can be used as memories. In this way, LUTs in a slice can be combined to obtain SRAM units with different sizes. Additionally, the SLICEM architecture allows the LUTs to be chained together serially to function as a shift register.

Figure 18 shows block diagram for proposed PUF design [14]. A single PUF instance produces a single bit and consists of 2 LUT blocks and 2 multiplexers. LUTs are configured as shift registers and concatenated with 2-to-1 carry chain multiplexers. Shift register contents are pre-initialized such that

- LUT A: 0101 0101 0101 0101 (0x5555)
- LUT B: 1010 1010 1010 1010 (0xAAAA)

Note that the contents of both LUTs are compliment of each other. Shift register

[1]25% of the LUTs in Xilinx Virtex-5 can be configured to operate as either shift registers or distributed RAM.

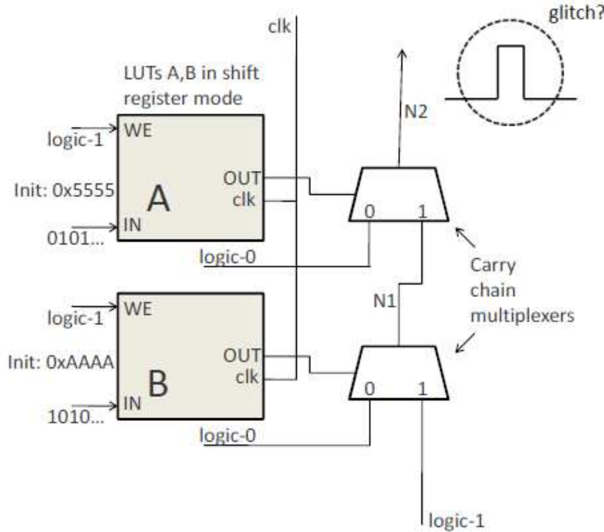


Figure 18: Block diagram of Anderson's PUF [14] in Xilinx Virtex-5.

inputs *IN* continue to supply the same sequences and shift register outputs *OUT* drive the select input pins on carry chain multiplexers. "0" data input on both carry chain multiplexers are tied to logic-0. The bottom carry chain multiplexer has its "1" data input tied to logic-1. The output of bottom carry chain multiplexer drives the "1" data input of the upper carry chain multiplexer.

Considering the ongoing behavior of this block, this PUF circuit will produce either a logic-0 or logic-1 based on random process variations. Initially, the *OUT* pin of *LUT A* is at logic-0, and consequently *N2* signal is at logic-0. Likewise, the *OUT* pin of *LUT B* is at logic-1 which drives *N1* signal at logic-1. At the rising clock edge, the *OUT* pin of *LUT A* will transition from logic-0 to logic-1 and the *OUT* pin of *LUT A* will transition from logic-0 to logic-1 and select *N1* signal.

After this point, there are two available scenarios. In the first case, *LUT B* and the multiplexer it drives are faster than *LUT A* and its multiplexer. Hence, when *LUT B* transitions from logic-1 to logic-0, signal *N1* also transitions from logic-1 to logic-0. After that, slower *LUT A* drives the multiplexer from logic-0 to logic-1 and consequently signal *N2* is held constant at logic-0. In the second case, *LUT A*

and its multiplexer behaves faster than the other ones. Therefore, output of $LUT\ A$ transitions from logic-0 to logic-1 before *N1* signal transitions from logic-1 to logic-0. As a result, a glitch appears on *N2* signal for a while before *N1* transitions to logic-0.

The presence or absence of the glitch and the period of that glitch are all due to complex manufacturing process variations present on physical components. Consequently, the presence or absence of glitch could be used to determine the output that will be either logic-0 or logic-1.

Figure 19 shows the proposed structure for how to capture a PUF signature bit from *N2* signal. *N2* signal is connected to the asynchronous preset input of a D-type flip-flop. The flip-flop is initialized to logic-0 and its output Q is fed back to its input D. Accordingly, when a glitch on *N2* line occurs, flip-flop output Q becomes logic-1 and it is said the PUF signature is 1. Otherwise, flip-flop output Q is logic-0 and it is said the PUF signature is 0.

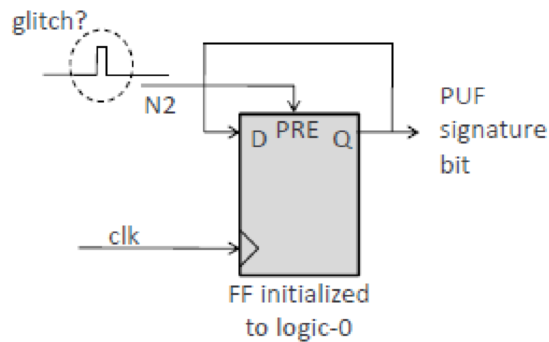


Figure 19: N2 signal captured by a D flip-flop to generate the PUF signature bit [14].

However, one issue that should be handled is that process variations may trigger a short pulse such that it may be filtered out due to electrical characteristics of the wires causing to obtain a logic-0 as the PUF signature with a high probability.

Conversely, a glitch which is too wide makes the PUF signature logic-1 with high probability. Therefore, it is needed to arrange the slice locations in order to maximize the randomness of the PUF signature.

Anderson [14] found that the arrangement like in Figure 20 produced the best results. $LUTA$ is in the top position of the top SLICE while $LUTB$ is in the third position of the bottom SLICE. Select inputs of intermediate multiplexers between $LUTA$ and $LUTB$ are tied to logic-1. The flip-flop receiving the *N2* signal is placed in the top SLICE. The interconnection between SLICEs are provided with a dedicated wire instead of SLICE-to-SLICE general-purpose interconnect.

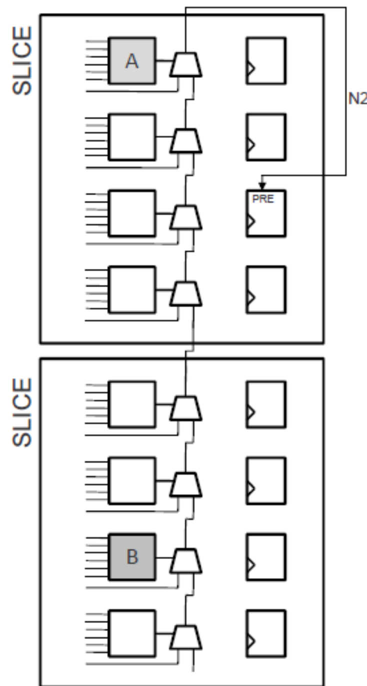


Figure 20: PUF bit generator rearranged [14].

One of the most important aspects of Anderson's PUF is that it is completely described in VHDL and can be handled by synthesis and place&route tools without requiring manual intervention. It could be implemented by using only two SLICEs

without any hard block or routing elements.

3.3 Our Modification of Anderson's PUF

This study on this work targets relatively low-cost FPGAs and implements a security framework which utilizes hardware intrinsic features for these devices. For this reason, a security scheme could be developed by modifying Anderson's PUF structure [14] for low-cost FPGAs.

As a part of the thesis work, Xilinx Spartan-6 FPGAs were used as the reference for low-cost FPGA devices. As opposed to Virtex series, Xilinx Spartan family of FPGAs [11] covers applications that require low-power footprint and have extreme cost sensitivity and high-volume potential. However, Virtex family of FPGAs [9] have more advanced features such as system monitor, tri-mode EMAC (Ethernet MAC), etc. If reconfigurable logic section of both Virtex and Spartan family of devices are compared, it can be seen that both family of devices are based on the same style of architectural arrangement and pretty much accomodates the same features. Besides these, the "SLICEM" type of slice block -which provides the logical functionality in Anderson's PUF design [14]- is found within both family of devices with slight differences (See Appendix A).

The function generators existed in each slice in Spartan-6 FPGAs are implemented as 6-input LUTs as in Virtex-5. Additionally, *SLICEM* block in Spartan-6 can also be configured as 32-bit or 16-bit shift register and a fast carry logic is found along with 8 storage elements (There are 4 storage elements in each slice of Virtex-5). Therefore, the same approach followed in Anderson's PUF [14] could be employed for Spartan-6 with minimal effort.

The main difference between *SLICEMs* in Virtex-5 and Spartan-6 is the initialization

state of storage elements. Figure 21 shows available attributes for storage elements within slices of Virtex-5. As shown in the figure, the initial state after configuration or global initial state is defined by separate *INIT0* and *INIT1* attributes apart from *SRLOW* and *SRHIGH*.

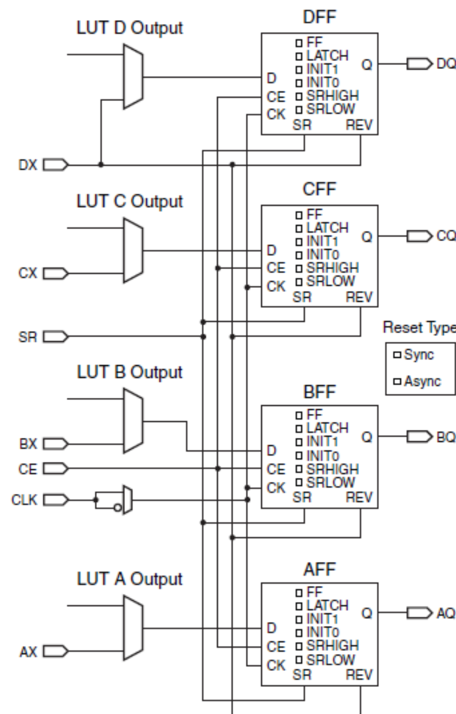


Figure 21: Register/Latch configuration in a slice of Virtex-5 [32].

However, the initial state of storage elements in each slice of Spartan-6 are specified by *SRINIT0* and *SRINIT1* attributes which also provide set and reset functionality, as shown in Figure 22. The problem lies in the fact that the initialization attributes of storage elements in a slice of Spartan-6 do not allow to implement the same circuitry as in Anderson's PUF design [14]. Remember that Anderson's PUF design [14] needed a storage element to capture a potential glitch to define the PUF signature as shown in Figure 23.

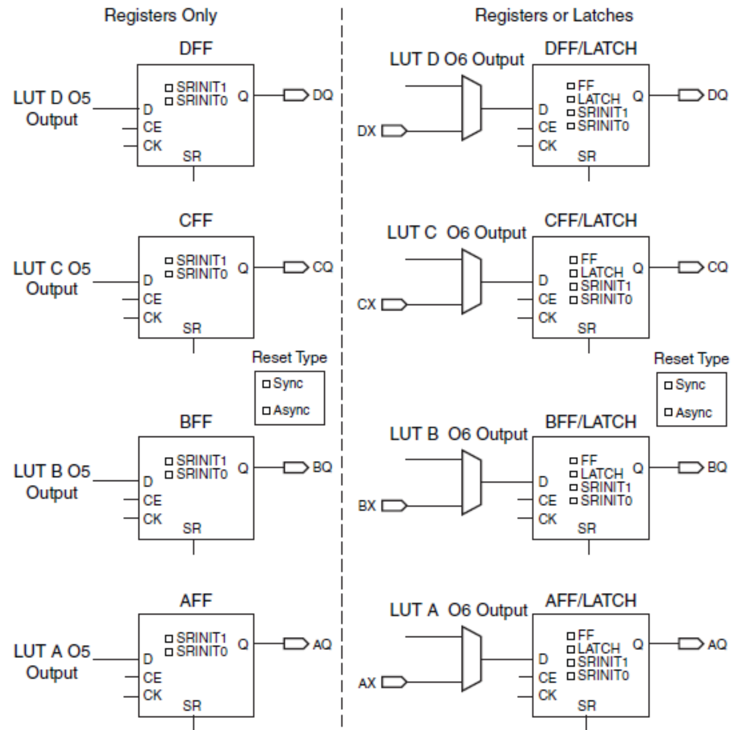


Figure 22: Register/Latch configuration in a slice of Spartan-6 [33].

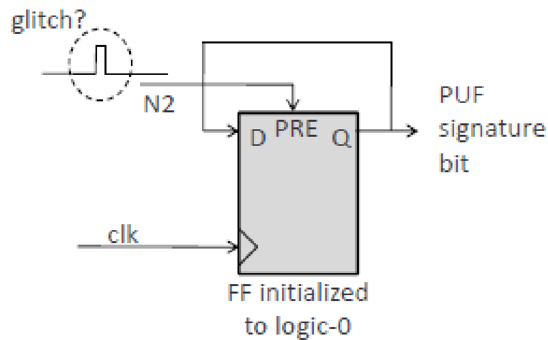


Figure 23: N2 signal captured by a D flip-flop to generate the PUF signature bit [14].

In Spartan-6, there is no separate attributes that one defines the initialization state and one another defines set/reset state. Therefore, to capture a potential glitch, a storage element in a *SLICEM* could be utilized as shown in Figure 24.

The design in Figure 24 rearranges the glitch capture circuit by setting the data input *D* to logic-0 and the initial state of FF by an external signal called RESET

which is connected to CE (Clock Enable) input of FF. Whenever CE is asserted, FF is initialized to logic-0 and hence a glitch could be captured by the SR input which is set by SRINIT1 attribute and asserted by PRESET signal coming from the top carry chain multiplexer, thereby achieving the same functionality as in Anderson's PUF [14].

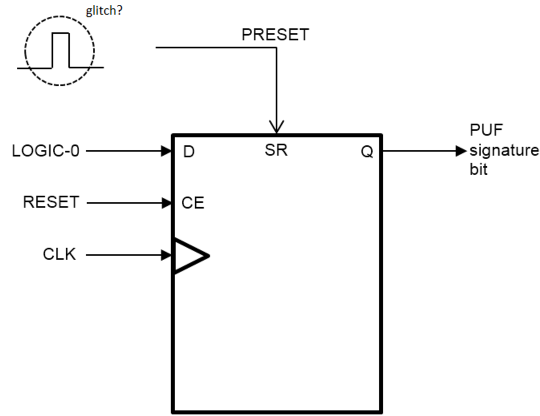


Figure 24: Glitch capture circuit by a D flip-flop in a SLICEM of Spartan-6.

3.3.1 Multi-bit PUF Signature Extraction

Our design methodology could be further generalized to extract multiple PUF signatures [34]. For the experimental validation, we instantiated 64 instances of our proposed design to generate a 64-bit signature. We evaluated the design using five Xilinx Spartan-6 XC6SLX45 FPGAs on Digilent ATLYS design platforms. The Xilinx Spartan-6 XC6SLX45 FPGA has 43.661 logic cells which are comprised of 6.822 slices, 54.576 FFs and maximum of 401Kb distributed RAM. 1602 of these slices are type of SLICEMs which corresponds to circa 25% with respect to total number of slices. If we take into account that it is required 2 SLICEMs to generate a single-bit PUF signature, we can calculate the required logic resource to generate a 64-bit PUF signature as 128 SLICEMs in total which is approximately 8% of total number of SLICEMs.

As Spartan-6 XC6SLX45 FPGA has a large island-style of logical resources, it is reasonable to organize the FPGA die into partitions such that each PUF signature set could be extracted from different portions of the FPGA silicon die.

Figure 25 shows general CLB and slice hierarchy inside Spartan-6 FPGAs. With reference to this, each CLB column contain two slice columns. One column is a *SLICEX* column, and the other column alternates between *SLICEL* and *SLICEM*. Since we need *SLICEM* type of slices to implement our PUF design, we can arrange each PUF instance in FPGA die as shown in Figure 26 [34].

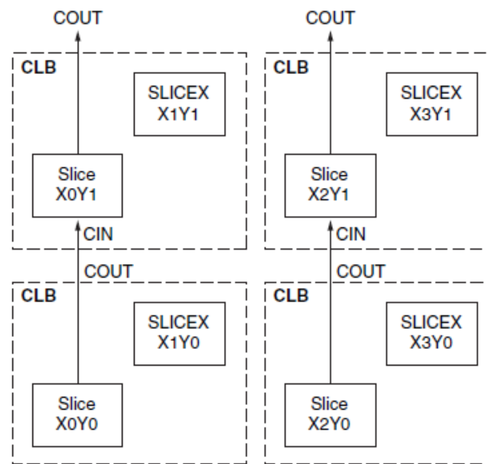


Figure 25: Row and column relationship between CLBs and Slices in Spartan-6 [33].

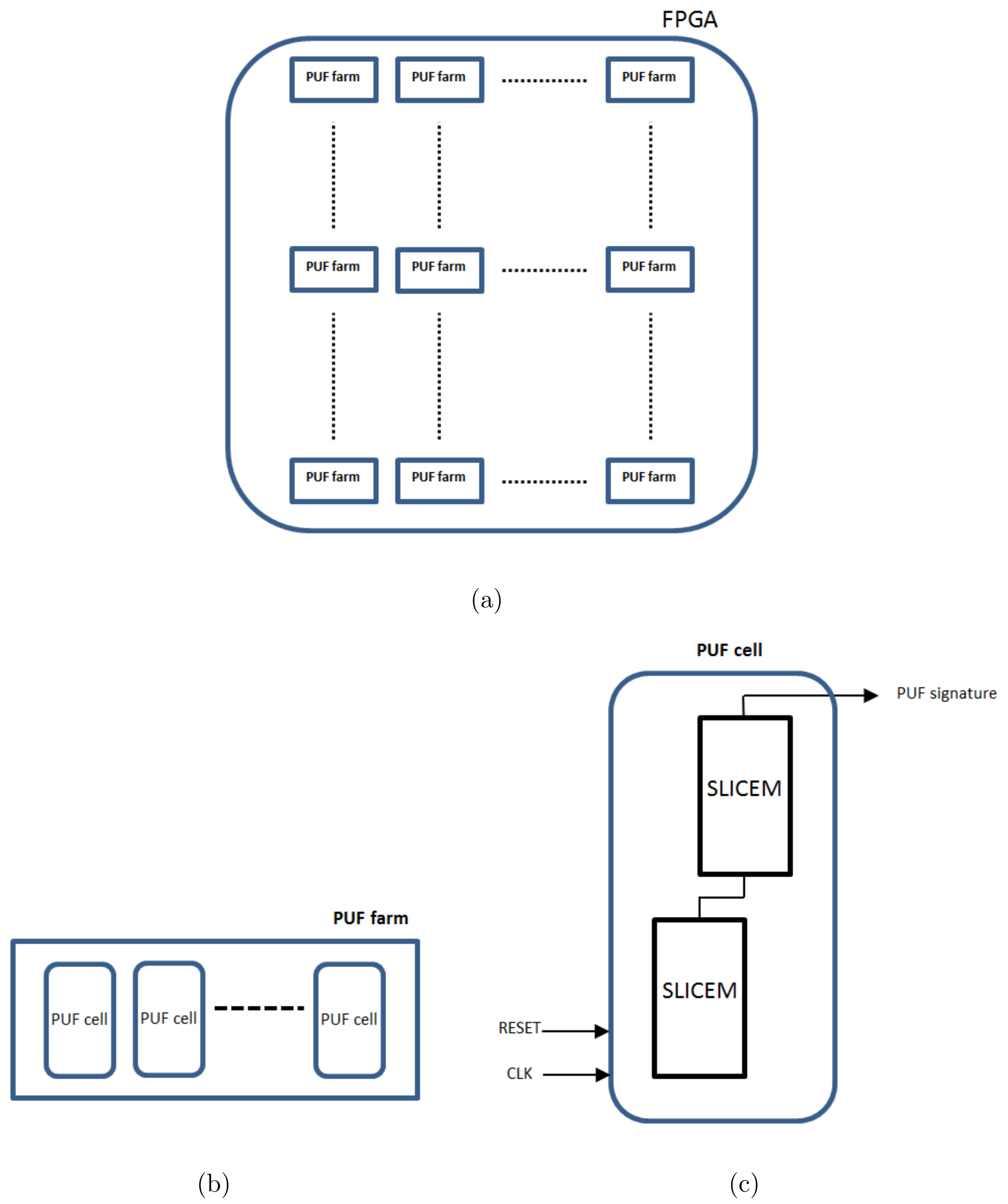


Figure 26: Multi-bit PUF signature extraction [34].

CHAPTER IV

PUF KEY-BASED ACTIVE HARDWARE OBFUSCATION

In this chapter, the protection method called "obfuscation" is introduced and then is presented a robust security approach which leverages hardware security by combining obfuscation and PUF.

Obfuscation is the concept of hiding the intended description or structure of a system in order to conceal its functionality and hence make it secure against reverse-engineering. Apart from encryption, obfuscation does not include ciphering and deciphering phases. Instead, it protects the design or application by covering it and creating transition paths such that one can only access the correct functionality if it has the sequence of the right path.

In terms of FPGA design flow, there are two approaches for hardware obfuscation. The first one is called passive obfuscation in which the comprehensibility of the description of the hardware (Hardware Description Language (HDL)) is concealed without changing the functionality of the circuit. As opposed to passive obfuscation, active obfuscation directly alters the functionality of the circuit. Active obfuscation techniques are often "key-based" in which normal functionality of the obfuscated design can only be unlocked by applying a key or a sequence of keys; otherwise the circuit exhibits incorrect functionality.

The advantage of utilizing hardware intrinsic features for device security and authentication could further be expanded by combining hardware obfuscation technique with PUF method. "PUF key-based active hardware obfuscation" is done by embedding

a well-hidden finite state machine (FSM) or modifying the controller FSM of the circuit such that modified FSM controls the functional modes based on the PUF response of the device. An obfuscated FSM includes the original FSM, and additional initialization and isolation state spaces and transitions as shown in Figure 27.

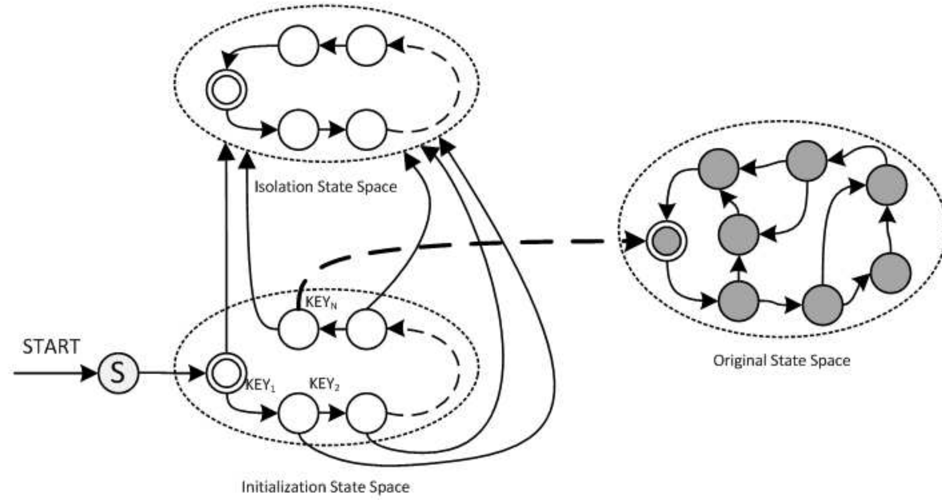


Figure 27: Basic diagram for an obfuscated FSM.

Initialization state space defines a path controlled by a key sequence to authenticate the access to original state space. Depending on the key size, states and paths between those states form a state machine linking with both original and isolation state spaces. However, there is only one path taking to original state space such that state transitions is done only by having the correct keys. Once an erroneous key is detected in a particular state, there is no way of accessing one of the states inside the original state space and then the next state is mapped to isolation state space.

Isolation state space prevents an attacker from accessing the original state space. It is controlled by the modified state machine which continuously map the next transition into one of the states inside the isolation state space.

Designing an obfuscated FSM with PUFs was previously examined by Koushanfar

[19]. Koushanfar demonstrated the BFSM (Boosted Finite State Machine) to enable designers to control their chips after fabrication by adding enough states and generating unique passkeys by using PUFs for some parts of the unlocking sequence of BFSM.

What is proposed by Koushanfar [19] is that upon power-up, the initial values of the design's added FFs are determined by the unique response from the PUF module. The number of added FFs should be large enough so that there is a high probability that PUF response sets the initial power-up state to one of the added states. Then one needs to provide a sequence of keys (PUF_K1, PUF_K2, , PUF_KN) required for traversal from the power-up state to the reset state of the original FSM.

Assuming the original FSM has |S| states, it can be implemented using $K = \log|S|$ FFs. Now if we add |S′| more states to build BFSM, it can be implemented using K'' $= \log\{|S| + |S'|\}$ FFs. Hence, for a linear growth in the number of FFs, the number of states exponentially increases.

In terms of security, it is highly important that there should be large numbers of added states. Therefore, the number of states should be set such that the value $2^{K''}$ $>> 2^{K}$. For this reason, our security framework consists of 32-bit state registers (FFs) so that one must try every time tampering one of the 2^{32} possible state values, thereby reducing the risk of reverse-engineering.

In addition to this, our security framework combines FPGA-based PUF method [14] which is explained in previous chapter with hardware obfuscation. As mentioned in PUF chapter, we obtain a multi-bit PUF signature by utilizing various portions of FPGA die. Then these obtained signatures are used as initial power-up state value and state transition passkeys in obfuscated FSM design.

Figure 28 describes how extracted PUF signature is associated with obfuscated FSM

design. We mentioned in chapter 3 that we instantiated 64 instances of our proposed design to generate a 64-bit signature. Consequently, 32-bit part of 64-bit PUF signature value is used to define the initial power-up state of obfuscated FSM by utilizing 32-bit state registers. Then, the other half of the PUF signature is arranged as four 8-bit transition passkeys such that each 8-bit is required to map succesfully to the next state in initialization state space. Otherwise, FSM controller sets the state registers to one of the states in isolation state space. The number of possible set of PUF keys can be calculated as follows:

$$2^{32} \times 2^{8} \times 2^{8} \times 2^{8} \times 2^{8} = 17.592.186.044.416$$

As can be seen, the probability to find the correct set of PUF keys is very low and an attacker must try every possible combination to find the correct initialization sequence by probing state registers manually at runtime. This result could be further generalized as follows:

- 2^N: Number of states
- M: Number of bits for transition passkeys
- K: Number of transitions

Then the number of possible combinations C could be calculated as

$$C = 2^{N} \times 2^{M \times K}$$

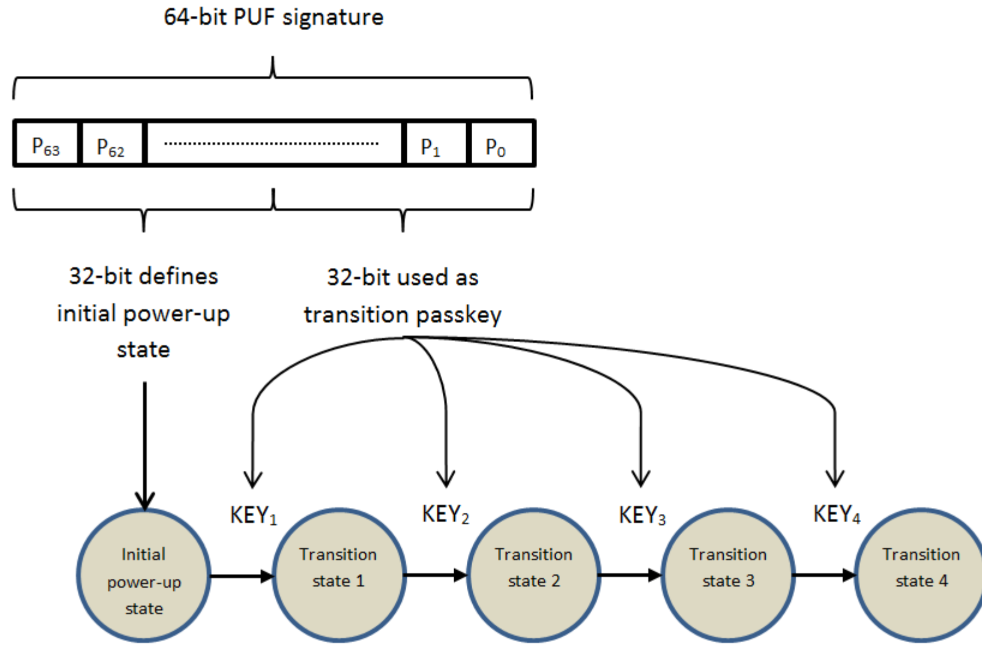


Figure 28: Multi-bit PUF signature defines initial power-up state and state transition passkeys.

The obfuscated FSM design used in our tests utilizes a state machine with 32-bit state registers by adding isolation states to the original design, thereby satisfying $S'' >> S$ condition. The table below gives a brief idea about area overhead in terms of slice utilization caused by 32-bit state registers for a couple of reference designs. The results were obtained for Xilinx Spartan-6 XC6SLX45 FPGA[1] by running Xilinx ISE DS 13.2 design automation tool.

As shown in Table 5, a reference set of circuits were compared with respect to the design with original FSM and the design which is obfuscated the original FSM by employing 32-bit state registers. As the number of states of original FSM increases, area overhead shows a reasonable growth with respect to total reconfigurable region.

[1]Xilinx Spartan-6 XC6SLX45 FPGA has 43.661 logic cells which correspond to 6.822 slices.

Table 5: Comparison of slice utilization between original and obfuscated FSM.[2]

# of States	*# of Inputs*	*# of Outputs*	*Slices Occupied*		*Area Overhead %*
			Original FSM	Obfuscated FSM	
8	8	8	7	81	1.08
8	16	16	14	90	1.11
8	32	32	19	96	1.13
16	8	8	10	114	1.52
16	32	32	24	114	1.32
32	32	32	39	181	2.08
64	32	32	79	324	3.59
128	32	32	144	499	5.20

Reference circuits used in tests were given to design software as HDL entries which were generated by a custom Perl script. An example of these scripts can be found at the end of this thesis at Appendix B.

[2]The results are the estimated values generated by the software after mapping stage.

CHAPTER V

PROPOSED METHODOLOGY

In this chapter, our proposed methodology will be described in detail and then a reference design which exploits our methodology will be introduced.

We propose fast, secure and remote MultiBoot which is a security framework for reconfigurable embedded systems. Our methodology is secure since it leverages hardware security by combining PUF method with hardware obfuscation. Additionally, our methodology is fast since it minimizes bitstream transmission time over network by implementing an efficient compression method. The remote feature of our methodology points to the fact that our proposed system could be controlled as a remote client over network. Figure 29 represents a general view of the proposed framework.

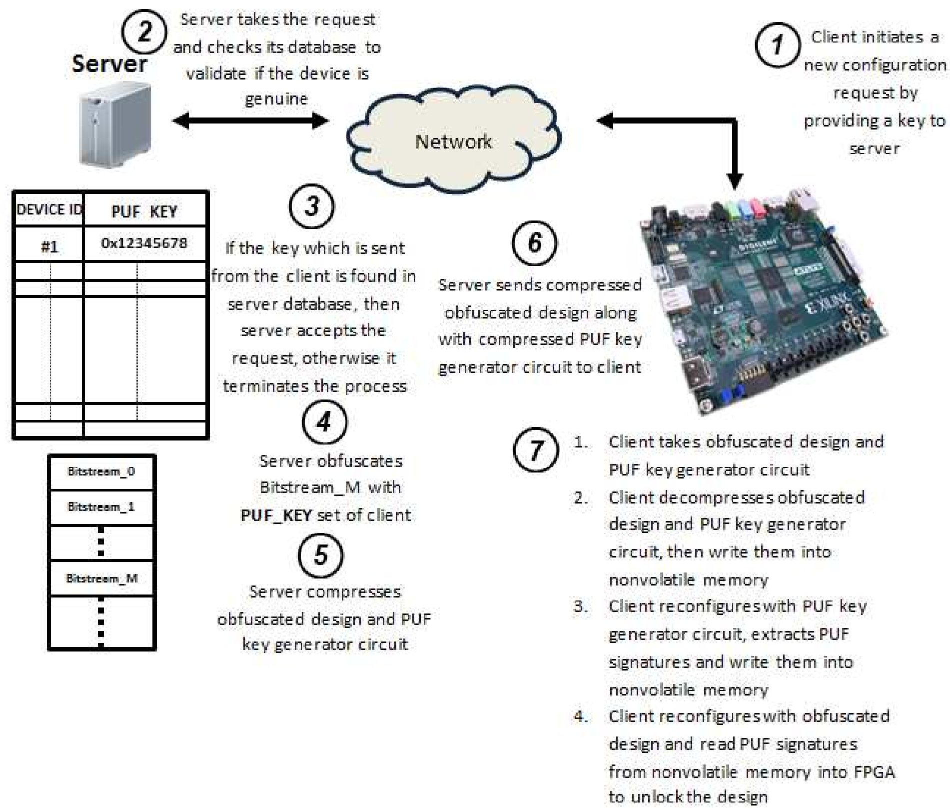


Figure 29: Proposed framework.

Here, *server* represents a design house and *client* represents an FPGA-based system which is maintained and checked by design house. Additionally, server has possible PUF signatures of that particular device before the client system has been dispatched to the user.

Client obtains an application design which is obfuscated with particular PUF keys by design house whenever it needs to change its configuration. Such an example could be an update operation where user needs to update its design with a new version, or a case in which a faulty design has to be changed with a correct one in a secure way.

The proposed framework could be explained step-by-step as follows:

Step 1: Client initiates a new configuration request by sending a particular key

to server over unsecure network. The *key* could be Device-DNA of FPGA, device ID of nonvolatile memory, or a combination of them which is a predefined number and known by the server. We used 56-bit Device-DNA key of FPGA device in our reference platform.

Step 2: Server takes the request of a new configuration with key sent from the client system. Then, server checks its database to control if the client system is genuine by using the key.

Step 3: If the key of client system is found in device database of the design house, server obfuscates the design with specific PUF keys of client system. Otherwise, server terminates the process.

Step 4: Server compresses obfuscated design along with PUF signature generator circuit.

Step 5: Server sends compressed and obfuscated design and compressed PUF signature generator circuit to the client system.

Step 6: Client takes PUF signature generator circuit and obfuscated design, decompresses them and writes to the nonvolatile memory at particular locations. Then, client sets the related configuration registers for a MultiBoot operation and reconfigures with PUF signature generator circuit.

Step 7: PUF generator circuit extracts 64-bit PUF signature of device and write them into nonvolatile memory.

Step 8: PUF generator circuit sets client system for a MultiBoot operation and reconfigures with obfuscated design.

Step 9: Obfuscated design reads 64-bit PUF signature from nonvolatile memory and

uses them to unlock the design.

5.1 Building Blocks of the Proposed Framework

In this section, main blocks of our proposed framework will be described in detail.

5.1.1 PUF Key Generator Design

PUF key generator design is used to extract 64-bit unique keys from client device. As previously explained in Chapter 3.3, we utilize a "PUF farm" on FPGA to obtain particular PUF keys of client. Then extracted keys are written into nonvolatile memory via SPI interface and a MultiBoot operation is performed. Figure 30 shows the block diagram of PUF key generator design.

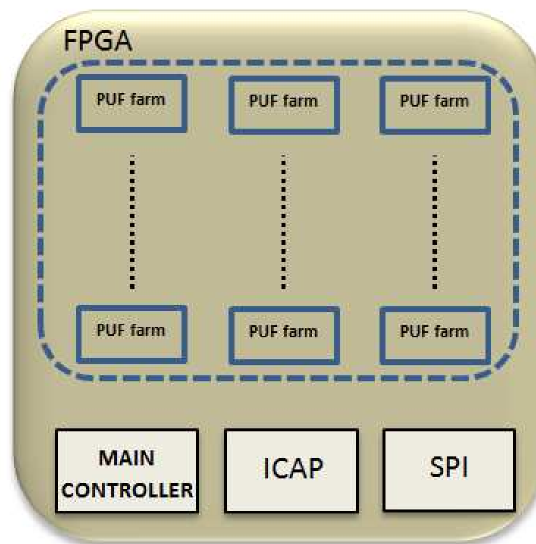


Figure 30: Block diagram of PUF key generator design.

Main controller unit employs a finite state machine which controls PUF block, ICAP interface, and SPI operations. When the design starts running, first it initiates PUF

key extraction process. After that, it takes extracted keys and write them into the nonvolatile memory by using SPI block which incorporates a custom SPI controller. Finally, main controller unit utilizes ICAP interface in order to restart FPGA with obfuscated design by MultiBoot.

5.1.2 Base Design

Our reference platform has a *Base Design* which is the base configuration in our proposed framework and it is used by the client system should it needs a new configuration. Basically, it sends a configuration request along with a key specific to client system. Then it receives compressed obfuscated design and PUF key generator design bitstreams. After that, base design decompresses these bitstreams and write them into nonvolatile memory. Finally, it initiates a MultiBoot operation to reconfigure FPGA with obfuscated design.

The underlying hardware of base design consists of a soft processor based system. It employs a 32-bit soft processor named *MicroBlaze* [35] and a set of IP blocks and peripherals needed during the authentication process between client & server and for other tasks. Figure 31 shows main hardware blocks employed in base design.

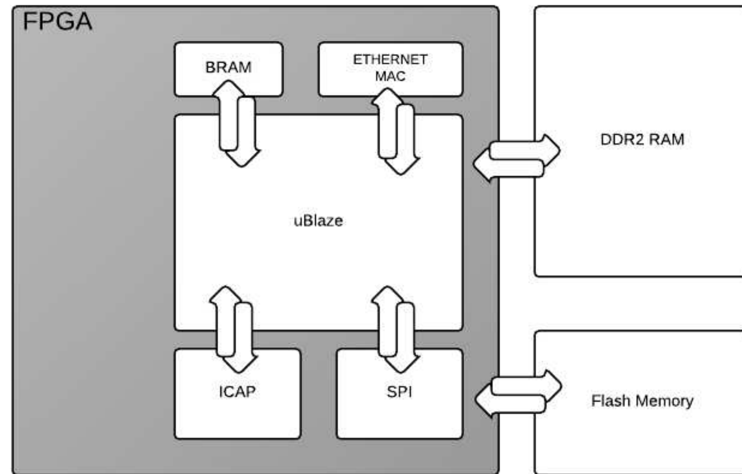


Figure 31: Block diagram of base design.

As shown in Figure 31, MicroBlaze soft-processor core is at the center of our hardware design. All the IP blocks and peripheral interfaces are connected to MicroBlaze and each other by 32-bit PLB (Processor Local Bus) [36] interface. A short description about each hardware block used in base design is given as follows:

- BRAM: XPS Block RAM (BRAM) [37] interface controller is utilized to implement a 64 KB local memory. It is used to start a bootloop program on software which waits for a configuration request.

- DDR2 RAM: Multi-Port Memory Controller (MPMC) [38] is used to interface with 128 MB DDR2 RAM. This memory is used to implement a basic file system.

- ICAP: Basic design utilizes ICAP to enable the access to the configuration logic and to initiate MultiBoot process. For this reason, the XPS HWICAP [39] IP core was added to enable MicroBlaze core to read and write the FPGA configuration memory through the ICAP at run time. Hence, it is possible to control configuration memory and FPGA by a software program.

- ETHERNET MAC: LogiCORE IP XPS Ethernet Lite Media Access Controller (MAC) [40] is used to enable network access of client system.

- SPI: A custom SPI block is added to interface with Flash memory in order to write obfuscated design and PUF key generator design bitstreams into Flash memory which are received from server.

- Flash Memory: 16 MB SPI Flash memory is used to store base design bitstream, PUF key generator design bitstream, obfuscated design bitstream, and PUF keys. Memory organization of these items on Flash memory is given in Figure 32.

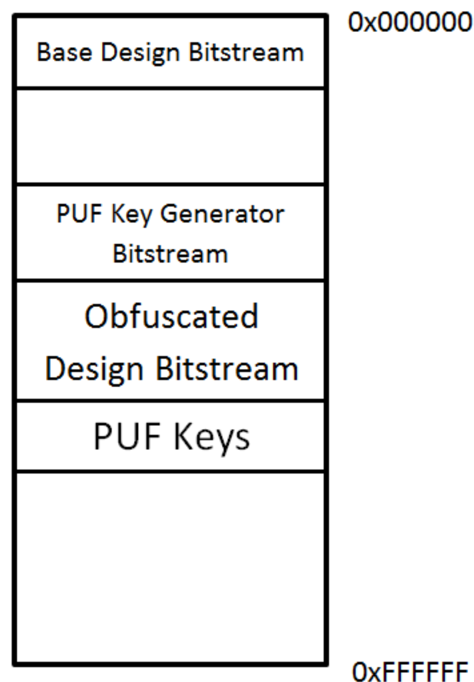


Figure 32: Memory organization on Flash memory.

Base design employs a software application which is built on a low-level software layer (Standalone BSP) to control hardware. Additionally, *Xilinx Memory File System (xilmfs)* is used to implement a basic file system and *lwIP TCP/IP Stack* [41] library

is used to run a network application in order to communicate with server.

The software application on base design starts a TCP/IP session in order to send a configuration request to server and then take compressed obfuscated design and PUF key generator design bitstreams. We used TFTP (Trivial File Transfer Protocol) in our reference platform to send & receive data over a local network, but any TCP/IP protocol could be selected.

The next step of software program is to decompress the received obfuscated design and PUF key generator design bitstreams and then write them into nonvolatile memory by using custom SPI controller. After that, a MultiBoot operation is started by utilizing ICAP to reconfigure FPGA with obfuscated design.

5.1.3 Obfuscated Design

Obfuscated design is the next configuration of client after extracting PUF keys. We described the methodology for obfuscating a design by using PUF keys of a device in Chapter 4. As previously mentioned, a 64-bit PUF key is used in order to "unlock" the obfuscated design. Therefore, PUF keys which are extracted from device and then written into nonvolatile memory by PUF key generator design are read from nonvolatile memory by an SPI controller block in obfuscated design. 32-bit part of PUF key defines the initialization state of obfuscated design and the other four 8-bit PUF keys defines transition keys between states in initialization state space (See Figure 28). In our proposed framework, we applied our proposed PUF-key based active hardware obfuscation method by creating an obfuscated finite state machine with a computer program (See Appendix B). Then we manually modified the example application to integrate it into generated obfuscated finite state machine. Although this process represents a semi-automatic flow, a fully automatic flow is considered as

a future work.

5.1.4 Bitstream Compression

Majority part of a typical bitstream consists of configuration data which utilizes each reconfigurable cell of FPGA during configuration. Therefore, a bitstream generally has a fixed size for each design to be implemented on a particular device. However, if utilization rate of these identical elements are low for any specific implementation, then bitstream could contain many redundant bits which may lead an increase at both transmission and configuration time. Another issue is that one who has access to the bitstream can discover the device model by measuring its size. Then he can analyze or modify and compare the attacked bitstream to extract data. Therefore, bitstream compression could be used to prevent these circumstances.

Xilinx BitGen tool which is built-in its ISE toolkit can generate a compressed bitstream for the netlist (by using *-g option*), however the compressed bitstream data still contains redundancies. For this reason, an additional compression technique would be meaningful in order to increase protection during both transmission and configuration.

Our proposed framework applies a simple bitstream compression algorithm over BitGen compressed bitstream. In this algorithm, repeated bytes or words are represented with a "MW" which stands for "Marker Word". A marker word indicates that the data which comes after the marker word represents the number of consecutive data which is received just before the marker word. After compressing the bitstream, the marker word is appended at the beginning of the compressed bitstream to inform the recipient side. As a result, we obtain a "double-compressed bitstream". Figure 33 shows an example flow for the algorithm.

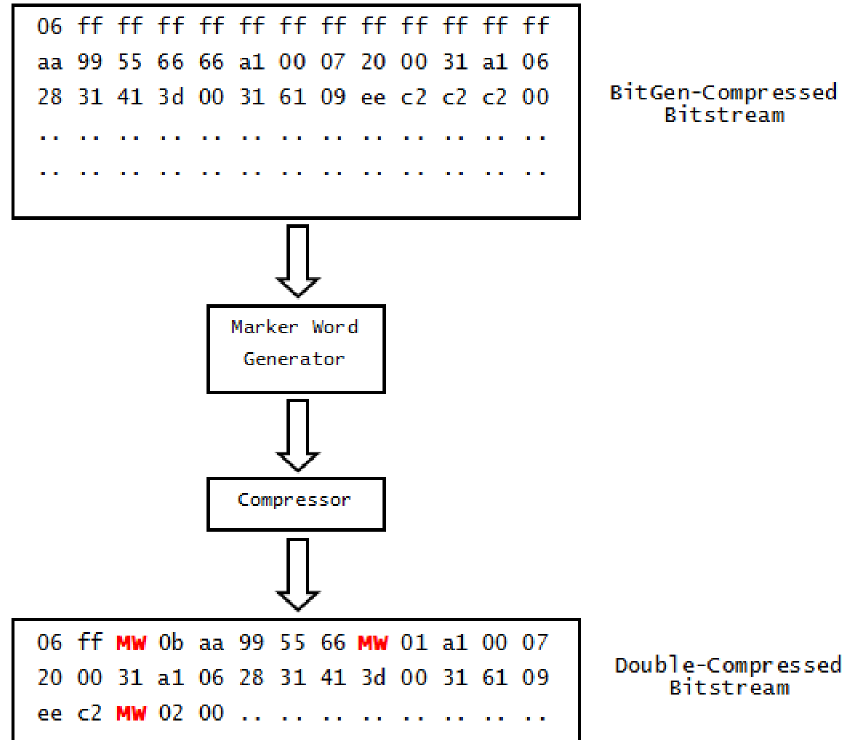


Figure 33: "Double-compressed" bitstream using proposed technique.

In this flow, at first BitGen compressed bitstream is analyzed by "Marker Word Generator". Marker word generator tries to find a unique data which is not included in the bitstream. Once the marker word is found, BitGen compressed bitstream and the obtained marker word is sent to "Compressor". Then compressor generates the "double-compressed" bitstream and appends the marker word with its length at the beginning of the bitstream. Table 6 shows the obtained results by applying double-compression technique and compares them with BitGen-compressed technique for a couple of reference designs.

Table 6: Comparison of BitGen-compressed and double-compressed techniques for various application bitstreams.

Design Content	*Bitstream Size [byte]*		*Overall Reduction*
	BitGen-Compressed	Double-Compressed	
Simple I/O operations ICAP interface	483.818	144.714	70.1%
20-bit counter Simple I/O operations ICAP interface	492.780	146.359	70.3%
RGB to YCrCb Color Space Converter	530.542	157.480	70.3%
1024 word 18-bit FIFO ICAP interface	488.146	145.868	70.1%
32-bit Floating-to-Fixed Point Conversion	489.920	146.115	70.2%

The inverse operation is performed at the recipient side and Xilinx BitGen compressed bitstream is reobtained. Since FPGA can detect and decompress BitGen compressed bitstream, there is no need to perform another decompression process again.

The implementation of compression algorithm is performed on server side by a Perl script which finds the marker word and generates the compressed bitstream. For the implementation of decompression algorithm, client system runs a software which decompresses the received bitstreams.

5.1.5 Server

Server runs a software program which waits for a configuration request from a client system. When a configuration request is received, server program examines whether client system is authentic. If server finds that the client is authentic, then it obfuscates the design by using PUF keys of client device. In our proposed framework, we manually modified the design by applying the flow in Chapter 4. After design is obfuscated, server program compresses both PUF key generator and obfuscated designs as described in previous subsection and sends them over network.

5.2 Implementation of Proposed Framework

As an example implementation of our proposed framework, we demonstrated a reference video application for the representation of client system. This reference application runs on a development board from Digilent Inc. [42] called "Atlys Spartan-6 FPGA Development Board" which includes a Xilinx Spartan–6 LX45 FPGA, a 128Mbyte DDR2 RAM, a 16Mbyte SPI Flash memory, 10/100/1000 Mbps Ethernet PHY, and many more features to accommodate our system requirements. During all the design process of our proposed framework, Xilinx ISE DS 13.2 toolkit was employed. Additionally, a stereo camera module from Digilent Inc. [42] called "Vmod-CAM" which employs two 2-megapixels CMOS digital image sensors is integrated with the development board. Figure 34 shows an image of hardware of our reference client system.

(a) Atlys Development Board

(b) VmodCAM - Stereo Camera Module

Figure 34: Reference client system.

In this reference client system, two development boards were used to represent the genuine system and the counterfeited system respectively. First, we started the genuine system by initiating a configuration request and sending its Device DNA to server. Then server approved that the system is authentic and responded with compressed PUF key generator and obfuscated design bitstreams. After taking these compressed bitstreams and decompressing them, client system generated 64-bit PUF key and wrote them into Flash memory and reconfigured with obfuscated bitstream. Finally, 64-bit PUF signatures were read from Flash into FPGA and state registers

utilized with correct PUF keys. As a result, the video application started to operate normally.

We repeated the same process above for the other client which represents a counterfeited system. First, counterfeited client sent its Device DNA to server and then server terminated the process since it could not find a paired Device DNA in its database. After that we modified counterfeited client system by modifying the value which is sent to server as the same of the genuine device. Then server confirmed the request and sent compressed PUF key generator and obfuscated design bitstreams. Although the counterfeited client was able to decompress and reconfigure with obfuscated design, it couldn't run the video application correctly since it didn't have the correct PUF keys.

Table 7 shows the Hamming distance results for our 64-bit PUF key implementation on five Spartan-6 devices. We used one PUF farm area for PUF key generation. The average value of our data points is 29.2 which is relatively close to the expected 32. This result can further be improved by utilizing more area on FPGA die and comparing larger set of data points.

Table 7: Hamming distance of 64-bit PUF keys on five Spartan-6 devices.

$DEVICE$	#1	#2	#3	#4	#5
#1		28	31	25	24
#2			31	33	28
#3				24	39
#4					29
#5					

Table 8 shows comparison of resource utilization between the original design and the

obfuscated version of original design for the reference video application. The increase in resource utilization is reasonable since an additional SPI block is included into obfuscated design to read the extracted PUF keys from Flash memory.

Table 8: Comparison of resource utilization between original and obfuscated designs.

Resources	*Original Design*	*Obfuscated Design*
Slice Registers	910	1827
Slice LUTs	1452	1882
Occupied Slices	547	697
Bonded IOBs	102	108
Slice LUTs used as Memory	42	59

CHAPTER VI

CONCLUSION

This thesis focused on developing a framework for secure reconfiguration of low-cost FPGAs which have limited design protection options against counterfeiting. For this reason, we adapted PUF methodology for FPGAs by using an efficient PUF implementation technique and then combined it with another powerful design protection scheme called active hardware obfuscation. The combination of these two technique provided an advantage over other solutions since the proposed framework utilizes hardware intrinsic features of the device, thereby eliminating the need for a particular security key. Another advantage which comes out with this methodology is that a design, which is obfuscated with PUF keys of a particular device will only operate correctly if user has that particular device. Additionally, our proposed framework protects designs without a dedicated protection block. Another difference is that high-end FPGA-based systems have dedicated encryption blocks and an encrypted design does not work on a counterfeited product if the attacker has not the correct key. However, our framework deceives the adversary by locking the design and exhibiting incorrect input and output behavior at runtime.

The proposed framework also established a server-client relationship to enable reconfiguration of device over network. Since the transmission time over network could be a bottleneck for timely critical systems, we used a simple compression algorithm to shrink the bitstream size. As a result, our framework forms a strong infrastructure for companies or design houses which need to control their products in the market.

As a proof-of-concept, we demonstrated the effectiveness and feasibility of our framework by implementing a video application. We showed that the application can only function properly on genuine client system for which the application design is obfuscated by using PUF keys dedicated to that system.

As a future work, our framework will be enhanced by automating the process of PUF-key based active hardware obfuscation with FSM extraction techniques.

APPENDIX A

SLICEM DETAILS

XILINX VIRTEX-5 VS. SPARTAN-6

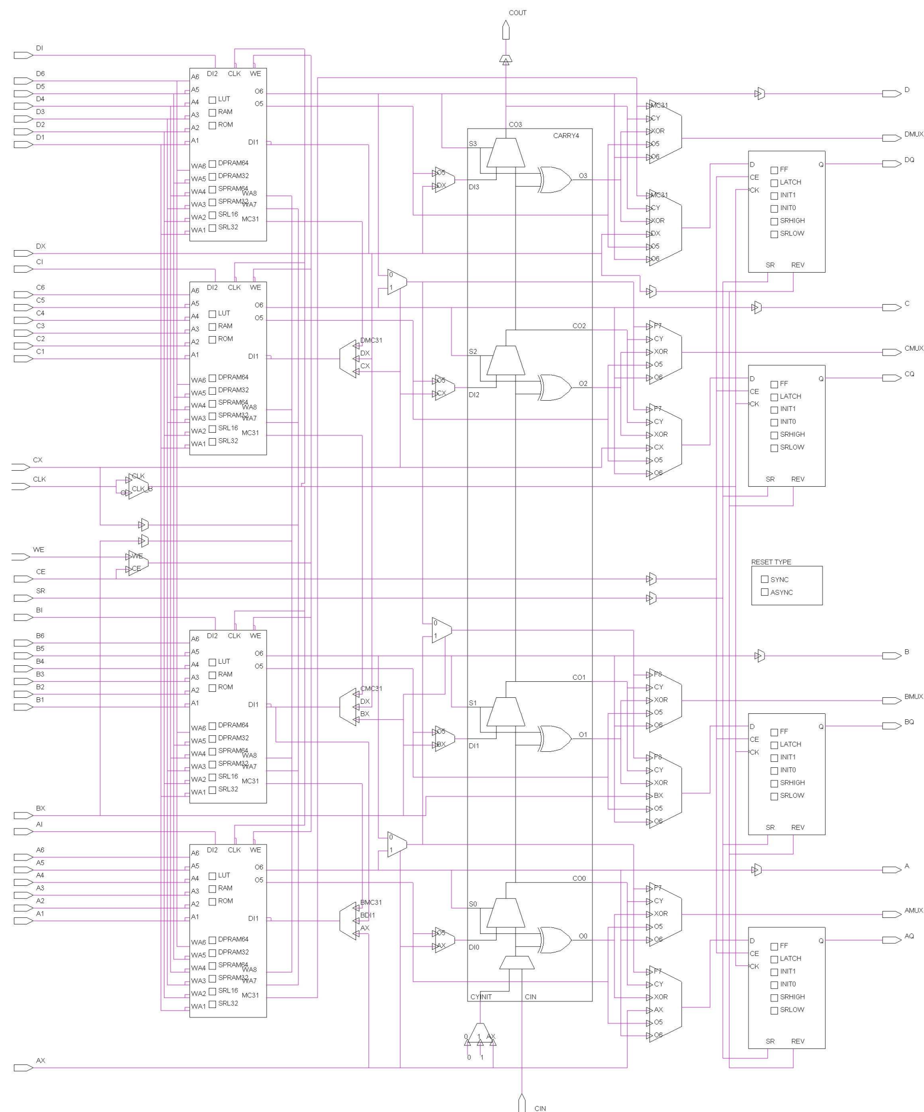


Figure 35: Block diagram of SLICEM in Xilinx Virtex-5 [32].

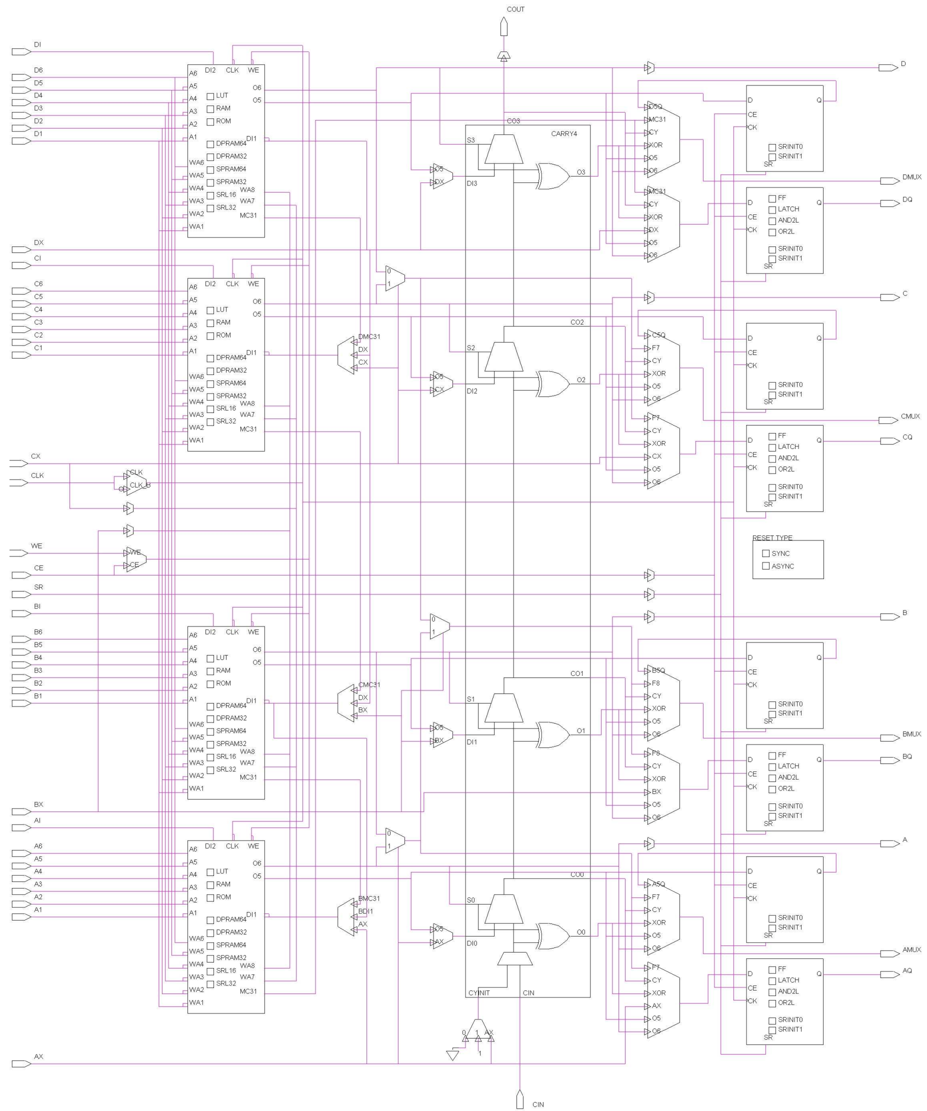


Figure 36: Block diagram of SLICEM in Xilinx Spartan-6 [33].

APPENDIX B

CUSTOM FSM GENERATORS (AS PERL SCRIPTS)

B.1 Original FSM Generator

```
1  #!/usr/local/bin/perl
2  use warnings;
3
4  if( @ARGV != 3 ){
5          print "\nMISSING or EXCESSIVE NUMBER of ARGUMENTS\n";
6          print "\nSYNTAX: #_of_ORIGINAL_SPACE_STATEs #_of_INPUTS #
               _of_OUTPUTS\n\n";
7          exit;
8  }
9
10 print "\nTemplate FSM Generator v0.1\n\n";
11
12 $maincontroller = "template_fsm.v";
13
14 $STATES = $ARGV[0];
15 $FF = log($STATES)/log(2);
16 $INPUTS = $ARGV[1];
17 $OUTPUTS = $ARGV[2];
18
19 open(MAINCONTROLLER, ">", $maincontroller) or die "can't open the
       file!";
20
21 create_top_module($FF);
```

```
22 create_symbolic_states($STATES, $FF);
23 create_state_registers($FF);
24 insert_main_control_path_state_register();
25 insert_main_control_path_next_state_logic($STATES, $FF);
26 close_top_module();
27
28 close(MAINCONTROLLER);
29
30 #SUBROUTINE-1: Create top module
31 sub create_top_module
32 {
33         print MAINCONTROLLER "module main_controller(clk, rst,
               ctrlin, ctrlout);\n";
34         print MAINCONTROLLER "\ninput clk, rst;";
35         print MAINCONTROLLER "\ninput [", $INPUTS - 1, ":0] ctrlin;"
               ;
36         print MAINCONTROLLER "\noutput reg [", $OUTPUTS - 1, ":0]
               ctrlout;\n";
37 }
38
39 #SUBROUTINE-2: Close top module
40 sub close_top_module
41 {
42         print MAINCONTROLLER "\n\nendmodule\n";
43 }
44
45 #SUBROUTINE-3: Insert state registers
46 sub create_state_registers
47 {
48         print MAINCONTROLLER "\n\n//STATE REGISTERS for the MAIN
               CONTROL PATH";
```

```
49          print MAINCONTROLLER "\nreg [", $_[0]-1, ":0] state_reg,
                state_next;";
50  }
51
52  #SUBROUTINE-4: Insert symbolic states
53  sub create_symbolic_states
54  {
55          my $i = 0;
56          for ($i = 0; $i < $STATES; $i++) {
57                  my $i_str = unpack("B32", pack("N", $i));
58                  $i_str =~ s/^0+(?=\d)//;
59                  print MAINCONTROLLER "\nlocalparam [", $_[1]-1, ":0]
                        ", "STATE" . $i, " = ", $_[1], "'b", $i_str, ";"
                      ;
60          }
61
62  }
63
64  #SUBROUTINE-5: Insert main control path state register
65  sub insert_main_control_path_state_register
66  {
67          print MAINCONTROLLER "\n\n//MAIN CONTROL PATH STATE REGISTER
                ";
68          print MAINCONTROLLER "\nalways@(posedge clk) begin";
69          print MAINCONTROLLER "\n\tif(rst) begin";
70          print MAINCONTROLLER "\n\t\tstate_reg <= STATE0;";
71          print MAINCONTROLLER "\n\t";
72          insert_end();
73          print MAINCONTROLLER "\n\telse begin";
74          print MAINCONTROLLER "\n\t\tstate_reg <= state_next;";
75          print MAINCONTROLLER "\n\t";
76          insert_end();
```

```
77          print MAINCONTROLLER "\n";
78          insert_end();
79  }
80
81  #SUBROUTINE-6: Insert main control path next state logic
82  sub insert_main_control_path_next_state_logic
83  {
84          print MAINCONTROLLER "\nalways@(*) begin";
85          print MAINCONTROLLER "\n\tstate_next = state_reg;";
86          $i = 0;
87          my $i_str = unpack("B32", pack("N", $i));
88          $i_str =~ s/^0+(?=\d)//;
89          print MAINCONTROLLER "\n\tctrlout = ", $OUTPUTS, "'b",
              $i_str, ";";
90          print MAINCONTROLLER "\n\tcase(state_reg)";
91          for($i = 0; $i < $STATES; $i++){
92                  print MAINCONTROLLER "\n\t\t", "STATE" . $i, ":";
93                  print MAINCONTROLLER "\n\t\t\tbegin";
94                  my $i_str = unpack("B32", pack("N", $i));
95                  $i_str =~ s/^0+(?=\d)//;
96                  if($i != ($STATES - 1)){
97                          my $randout = int(rand(2 ** ($OUTPUTS)));
98                          my $out_str = unpack("B32", pack("N",
                              $randout));
99                          $out_str =~ s/^0+(?=\d)//;
100                         print MAINCONTROLLER "\n\t\t\t\tctrlout = ",
                               $OUTPUTS, "'b", $out_str, ";";
101                         my $randin = int(rand(2 ** ($INPUTS)));
102                         my $in_str = unpack("B32", pack("N", $randin
                              ));
103                         $in_str =~ s/^0+(?=\d)//;
```

```
104                     print MAINCONTROLLER "\n\t\t\t\tif( ctrlin
                            == ", $INPUTS, "'b", $in_str, " ) begin";
105                     print MAINCONTROLLER "\n\t\t\t\t\tstate_next
                             = ", "STATE" . ($i+1), ";";
106                     print MAINCONTROLLER "\n\t\t\t\tend";
107             }
108             else{
109                     my $randout = int(rand(2 ** ($OUTPUTS)));
110                     my $out_str = unpack("B32", pack("N",
                            $randout));
111                     $out_str =~ s/^0+(?=\d)//;
112                     print MAINCONTROLLER "\n\t\t\t\tctrlout = ",
                             $OUTPUTS, "'b", $out_str, ";";
113                     print MAINCONTROLLER "\n\t\t\t\tstate_next =
                             ", "STATE0;";
114             }
115             print MAINCONTROLLER "\n\t\t\tend";
116     }
117     $i = 0;
118     print MAINCONTROLLER "\n\t\tdefault: ";
119     print MAINCONTROLLER "\n\t\t\tbegin";
120     my $randout = int(rand(2 ** ($OUTPUTS)));
121     my $out_str = unpack("B32", pack("N", $randout));
122     $out_str =~ s/^0+(?=\d)//;
123     print MAINCONTROLLER "\n\t\t\t\tctrlout = ", $OUTPUTS, "'b",
            $out_str, ";";
124     print MAINCONTROLLER "\n\t\t\t\tstate_next = ", "STATE" . $i
            , ";";
125     print MAINCONTROLLER "\n\t\t\tend";
126     print MAINCONTROLLER "\n\tendcase";
127     print MAINCONTROLLER "\nend";
128 }
```

```
129
130 #SUBROUTINE-7 Insert 'end' keyword
131 sub insert_end
132 {
133         print MAINCONTROLLER "end";
134 }
```

B.2 Obfuscated FSM Generator

```
 1 #!/usr/local/bin/perl
 2 use warnings;
 3 use POSIX;
 4
 5 if( @ARGV != 3 ){
 6         print "\nMISSING or EXCESSIVE NUMBER of ARGUMENTS\n";
 7         print "\nSYNTAX: #_of_ORIGINAL_SPACE_STATEs #_of_INPUTS #
               _of_OUTPUTS\n\n";
 8         exit;
 9 }
10
11 print "\nTemplate FSM Obfuscator v0.1\n\n";
12
13 $maincontroller = "template_obfuscated_fsm.v";
14
15 $STATES = $ARGV[0];
16 $INPUTS = $ARGV[1];
17 $OUTPUTS = $ARGV[2];
18
19 my @orig_space_state_array = ();
20 my @init_space_state_array = ();
21 my @tran_space_state_array = ();
22 my @main_state_array = ();
23 my @rand_state_array = ();
```

```
24 my $ff;
25
26 $ff = 32;
27
28 open(MAINCONTROLLER, ">", $maincontroller) or die "can't open the
     file!";
29
30 create_top_module();
31 create_symbolic_states();
32 create_state_registers($ff);
33 insert_main_control_path_state_register();
34 insert_main_control_path_next_state_logic();
35 close_top_module();
36
37 close(MAINCONTROLLER);
38
39 #SUBROUTINE-1: Create top module
40 sub create_top_module
41 {
42         print MAINCONTROLLER "module main_controller(clk, rst,
             ctrlin, ctrlout);\n";
43         print MAINCONTROLLER "\ninput clk, rst;";
44         print MAINCONTROLLER "\ninput [", $INPUTS - 1, ":0] ctrlin;"
             ;
45         print MAINCONTROLLER "\noutput reg [", $OUTPUTS - 1, ":0]
             ctrlout;\n";
46 }
47
48 #SUBROUTINE-2: Close top module
49 sub close_top_module
50 {
51         print MAINCONTROLLER "\n\nendmodule\n";
```

```
52 }
53
54 #SUBROUTINE-3: Insert state registers
55 sub create_state_registers
56 {
57         print MAINCONTROLLER "\n\n//STATE REGISTERS for the MAIN
               CONTROL PATH";
58         print MAINCONTROLLER "\n(* KEEP = \"TRUE\" *) reg [", $_
               [0]-1, ":0] state_reg;";
59         print MAINCONTROLLER "\nreg [", $_[0]-1, ":0] state_next;";
60 }
61
62 #SUBROUTINE-4: Insert symbolic states
63 sub create_symbolic_states
64 {
65         my $i = 0;
66         for ($i = 0; $i < $STATES; $i++){
67                 push(@orig_space_state_array, "ORIGSPACE" . "STATE"
                       . $i);
68         }
69         @init_space_state_array = ("RESET", "INIT");
70         @tran_space_state_array = ("CHECKPOINT1", "CHECKPOINT2", "
               CHECKPOINT3", "CHECKPOINT4");
71         @main_state_array = (@init_space_state_array,
               @tran_space_state_array, @orig_space_state_array);
72
73         print MAINCONTROLLER "\n\n//SYMBOLIC STATE DECLARATIONS";
74         $ff = 32;
75         $i = 0;
76         foreach (@main_state_array) {
77                         my $randval = int(rand(2 ** 32));
78                         while( $randval ~~ @rand_state_array ) {
```

```
79                                my $randval = int(rand(2 ** 32));
80                        }
81                        push(@rand_state_array, $randval);
82                        my $i_str = unpack("B32", pack("N", $randval
                              ));
83                        $i_str =~ s/^0+(?=\d)//;
84                        print MAINCONTROLLER "\nlocalparam [", $ff -
                                1, ":0] ", "GENERICSTATE" , $i, " = ",
                              $ff, "'b", $i_str, "; //", $_, " STATE";
85                        $i++;
86          }
87
88  }
89
90  #SUBROUTINE-5: Insert main control path state register
91  sub insert_main_control_path_state_register
92  {
93          print MAINCONTROLLER "\n\n//MAIN CONTROL PATH STATE REGISTER
                ";
94          print MAINCONTROLLER "\nalways@(posedge clk) begin";
95          print MAINCONTROLLER "\n\tif(rst) begin";
96          print MAINCONTROLLER "\n\t\tstate_reg <= GENERICSTATE0;";
97          print MAINCONTROLLER "\n\t";
98          insert_end();
99          print MAINCONTROLLER "\n\telse begin";
100         print MAINCONTROLLER "\n\t\tstate_reg <= state_next;";
101         print MAINCONTROLLER "\n\t";
102         insert_end();
103         print MAINCONTROLLER "\n";
104         insert_end();
105 }
106
```

```
107 #SUBROUTINE-6: Insert main control path next state logic
108 sub insert_main_control_path_next_state_logic
109 {
110         print MAINCONTROLLER "\nalways@(*) begin";
111         print MAINCONTROLLER "\n\tstate_next = state_reg;";
112         my $i = 0;
113         my $i_str = unpack("B32", pack("N", $i));
114         $i_str =~ s/^0+(?=\d)//;
115         print MAINCONTROLLER "\n\tctrlout = ", $OUTPUTS, "'b",
            $i_str, ";";
116         print MAINCONTROLLER "\n\tcase(state_reg)";
117         $i = 0;
118         foreach(@init_space_state_array){ #INIT SPACE STATES
119                         print MAINCONTROLLER "\n\t\t", "GENERICSTATE
                            " . $i, ":", " //", $_, " STATE";
120                         print MAINCONTROLLER "\n\t\t\tbegin";
121                         print MAINCONTROLLER "\n\t\t\t\tstate_next =
                             ", "GENERICSTATE", ++$i, ";";
122                         print MAINCONTROLLER "\n\t\t\tend";
123         }
124         $temp = @tran_space_state_array + $i;
125         $temp2 = @init_space_state_array + @tran_space_state_array;
126         foreach(@tran_space_state_array){ #TRANSITION SPACE STATES
127                         print MAINCONTROLLER "\n\t\t", "GENERICSTATE
                            " . $i, ":", " //", $_, " STATE";
128                         print MAINCONTROLLER "\n\t\t\tbegin";
129                         my $randout = int(rand(2 ** ($OUTPUTS)));
130                         my $out_str = unpack("B32", pack("N",
                            $randout));
131                         $out_str =~ s/^0+(?=\d)//;
132                         print MAINCONTROLLER "\n\t\t\t\tctrlout = ",
                             $OUTPUTS, "'b", $out_str, ";";
```

```
133                     my $randin = int(rand(2 ** ($INPUTS)));
134                     my $in_str = unpack("B32", pack("N", $randin
                           ));
135                     $in_str =~ s/^0+(?=\d)//;
136                     print MAINCONTROLLER "\n\t\t\t\tif( ctrlin
                           == ", $INPUTS, "'b", $in_str, " ) begin";
137                     $in_str = unpack("B32", pack("N", $temp2));
138                     $in_str =~ s/^0+(?=\d)//;
139                     $ff = 32;
140                     if( $i == ($temp - 1) ){
141                             print MAINCONTROLLER "\n\t\t\t\t\
                                   tstate_next = ", "GENERICSTATE" .
                                    ($temp), ";";
142                             print MAINCONTROLLER "\n\t\t\t\tend"
                                   ;
143                             print MAINCONTROLLER "\n\t\t\t\telse
                                    begin";
144                             print MAINCONTROLLER "\n\t\t\t\t\
                                   tstate_next = ", "state_reg + 1'
                                   b1;";
145                             $i++;
146                     }
147                     else{
148                             print MAINCONTROLLER "\n\t\t\t\t\
                                   tstate_next = ", "GENERICSTATE" .
                                    ++$i, ";";
149                             print MAINCONTROLLER "\n\t\t\t\tend"
                                   ;
150                             print MAINCONTROLLER "\n\t\t\t\telse
                                    begin";
```

```
151                         print MAINCONTROLLER "\n\t\t\t\t\
                                tstate_next = ", "state_reg + 1'
                                b1;";
152                     }
153                     print MAINCONTROLLER "\n\t\t\t\tend";
154                     print MAINCONTROLLER "\n\t\t\tend";
155         }
156         $temp1 = $i;
157         foreach(@orig_space_state_array){
158                 print MAINCONTROLLER "\n\t\t", "GENERICSTATE" . $i,
                        ":", " //", $_, " STATE";
159                 print MAINCONTROLLER "\n\t\t\tbegin";
160                 my $i_str = unpack("B32", pack("N", $i));
161                 $i_str =~ s/^0+(?=\d)//;
162                 if( $i != ($temp1 + $STATES - 1) ){
163                         my $randout = int(rand(2 ** ($OUTPUTS)));
164                         my $out_str = unpack("B32", pack("N",
                                $randout));
165                         $out_str =~ s/^0+(?=\d)//;
166                         print MAINCONTROLLER "\n\t\t\t\tctrlout = ",
                                 $OUTPUTS, "'b", $out_str, ";";
167                         my $rand = int(rand(2 ** ($INPUTS)));
168                         my $i_str = unpack("B32", pack("N", $rand));
169                         $i_str =~ s/^0+(?=\d)//;
170                         print MAINCONTROLLER "\n\t\t\t\tif( ctrlin
                                == ", $INPUTS, "'b", $i_str, " ) begin";
171                         print MAINCONTROLLER "\n\t\t\t\t\tstate_next
                                 = ", "GENERICSTATE" . ++$i, ";";
172                         print MAINCONTROLLER "\n\t\t\t\tend";
173                 }
174                 else{
175                         my $randout = int(rand(2 ** ($OUTPUTS)));
```

```
176                         my $out_str = unpack("B32", pack("N",
                               $randout));
177                         $out_str =~ s/^0+(?=\d)//;
178                         print MAINCONTROLLER "\n\t\t\t\tctrlout = ",
                                $OUTPUTS, "'b", $out_str, ";";
179                         print MAINCONTROLLER "\n\t\t\t\tstate_next =
                                ", "GENERICSTATE", $temp1, ";";
180                 }
181                 print MAINCONTROLLER "\n\t\t\tend";
182         }
183         print MAINCONTROLLER "\n\t\tdefault: ";
184         print MAINCONTROLLER "\n\t\t\tbegin";
185         my $randout = int(rand(2 ** ($OUTPUTS)));
186         my $out_str = unpack("B32", pack("N", $randout));
187         $out_str =~ s/^0+(?=\d)//;
188         print MAINCONTROLLER "\n\t\t\t\tctrlout = ", $OUTPUTS, "'b",
                $out_str, ";";
189         $temp2 = @init_space_state_array + @tran_space_state_array;
190         $in_str = unpack("B32", pack("N", $temp2));
191         $in_str =~ s/^0+(?=\d)//;
192         print MAINCONTROLLER "\n\t\t\t\tstate_next = ", "state_reg +
                1'b1;";
193         print MAINCONTROLLER "\n\t\t\tend";
194         print MAINCONTROLLER "\n\tendcase";
195         print MAINCONTROLLER "\nend"
196 }
197
198 #SUBROUTINE-100 Insert 'end' keyword
199 sub insert_end
200 {
201         print MAINCONTROLLER "end";
202 }
```

APPENDIX C

XILINX SPARTAN-6 CONFIGURATION DETAILS

C.1 Configuration Modes

Xilinx Spartan-6 devices support several different configuration methods. There are five different configuration interface available on Spartan-6 FPGAs as shown in Table 9. The configuration interface by which the configuration operation is performed is determined by 2-bit mode pins (M[1:0]).

Table 9: Spartan-6 FPGA configuration modes [7].

$ConfigurationMode$	$M[1:0]$	$BusWidth$
Master Serial/SPI	01	1, 2, 4
Master SelectMAP/BPI	00	8, 16
JTAG	XX	1
Slave SelectMAP	10	8, 16
Slave Serial	11	1

In serial configuration modes, configuration of FPGA is done by loading one bit per configuration clock cycle. The SelectMAP configuration interface is an 8-bit or 16-bit bidirectional configuration interface and provides access to device configuration logic hence can be used both for configuration and readback. The Master Serial Peripheral Interface (SPI) allows an SPI-based serial Flash memory device to be used to store configuration data. The Master Byte-Wide Peripheral Interface (BPI) supports configuration over a parallel NOR Flash memory device. The JTAG interface

is available as configuration interface anytime the device is powered up.

C.2 Device ID Check

Device ID check step compares device IDCODE register with the IDCODE value built in the configuration bitstream. Xilinx Spartan–6 FPGA JTAG IDCODE register has the following format:

vvvv:fffffff:aaaaaaaaa:ccccccccccc1 where

v represents revision,
f represents 7–bit family code,
a represents 9–bit array code (4–bit subfamily and 5-bit device identifier),
and c represents 11–bit company code.

Table 10 shows ID codes which belong to some of Spartan–6 FPGAs.

Table 10: Spartan-6 FPGA family IDCODE values [7].

Device	*IDCode(Hexadecimal)*
XC6SLX4	0xX4000093
XC6SLX9	0xX4001093
XC6SLX16	0xX4002093
XC6SLX25	0xX4004093
XC6SLX45	0xX4008093
XC6SLX75	0xX400E093
XC6SLX100	0xX4011093
XC6SLX150	0xX401D093

APPENDIX D

GLOSSARY

Antifuse technology - A kind of fusible-link technology. Devices based on this technology are said to be one-time programmable (OTP).

ASIC - Acronym for "Application Specific Integrated Circuit". A semiconductor device customized for a particular use.

ASSP - Acronym for "Application Specific Standard Product". An integrated circuit that implements a specific function.

Bitstream - Device configuration information for reconfigurable semiconductor devices.

Bootloop - A special software application that keeps the processor in a defined state until the actual application can be downloaded and run.

BPI - Acronym for "Byte Peripheral Interface". A parallel configuration interface.

CLB - Acronym for "Configurable Logic Block". The main logic resource for implementing sequential as well as combinatorial circuits.

DCM - Acronym for "Digital Clock Manager". A circuit for manipulating clock signals by multiplying and dividing an incoming clock, changing duty cycle, phase shifting, eliminating clock skew, etc.

FPGA - Acronym for "Field Programmable Gate Array". A semiconductor device designed to be configured in terms of hardware after manufacturing.

HDL - Acronym for "Hardware Description Language". A computer language which is used to describe and design electronic circuits.

ICAP - Acronym for "Internal Configuration Access Port". A configuration interface

that allows to access to the configuration memory of an FPGA.

IP - Acronym for "Intellectual Property". A reusable unit of logic, cell, or chip layout design that is the intellectual property of one party.

LogiCORE - A brand name for Xilinx-based IP units.

LUT - Acronym for "Look-Up Table". A kind of logic resource inside FPGA to implement a combinational logic function.

MAC - Acronym for "Media Access Control". A data communication protocol which provides addressing and channel access control mechanisms to communicate within a network.

MicroBlaze - A soft processor core designed for Xilinx FPGAs from Xilinx. Implemented entirely in the fabric of FPGA.

MultiBoot - A self-reconfiguration mechanism provided for some Xilinx FPGAs to fully reconfigure the device from a nonvolatile memory.

NCD - Acronym for "Native Circuit Description". A kind of description of a circuit design that physically represents the design mapped to the components in FPGA.

Netlist - Describes the connectivity of an electronic design and contain description of the parts or devices utilized.

PLB - Acronym for "Processor Local Bus". A bus infrastructure designed for Xilinx FPGAs.

PLL - Acronym for "Phase-Locked Loop"'. A control system that generates an output signal whose phase is related to the phase of an input reference signal.

PUF - Acronym for "Physical Unclonable Function". A function which utilizes inherent physical manufacturing variances to extract random output.

SelectMAP - A configuration interface that provides bidirectional data bus interface to the FPGA configuration logic.

SLICE - Primary programmable logic block in Xilinx FPGAs.

SoC - Acronym for "System-on-Chip". An integrated circuit that integrates all components of an electronic system into a single chip.

SPI - Acronym for "Serial Peripheral Interface". A synchronous serial data link standard.

TFTP - Acronym for "Trivial File Transfer Protocol". A file transfer protocol that is generally used for automated transfer of configuration or boot files between machines in a local environment.

Bibliography

[1] http://spectrum.ieee.org.

[2] 2012. http://www.ihs.com.

[3] http://www.ipcores.com.

[4] http://www.maximintegrated.com.

[5] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, "On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks,"

[6] http://www.xilinx.com.

[7] *Spartan-6 FPGA Configuration User Guide, UG380, xilinx.com.*

[8] É. Rannaud, "From the bitstream to the netlist," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pp. 264–264, Citeseer, 2008.

[9] *Virtex-5 Family Overview, DS100, xilinx.com.*

[10] *Virtex-6 Family Overview, DS150, xilinx.com.*

[11] *Spartan-6 Family Overview, DS160, xilinx.com.*

[12] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, no. 5589, pp. 2026–2030, 2002.

[13] 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*, pp. 148–160, ACM, 2002.

[14] J. Anderson, "A PUF Design for Secure FPGA-based Embedded Systems," in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pp. 1–6, IEEE, 2010.

[15] D. Lim, J. Lee, B. Gassend, G. Suh, M. Van Dijk, and S. Devadas, "Extracting secret keys from integrated circuits," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, pp. 1200–1205, 2005.

[16] P. Tuyls, G. Schrijen, B. Skoric, J. van Geloven, N. Verhaegh, and R. Wolters, "Read-Proof Hardware from Protective Coatings,"

[17] G. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pp. 9–14, IEEE, 2007.

[18] S. Kumar, J. Guajardo, R. Maes, G. Schrijen, and P. Tuyls, "Extended abstract: The butterfly PUF protecting IP on every FPGA," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pp. 67–70, IEEE, 2008.

[19] F. Koushanfar, "Provably secure active IC metering techniques for piracy avoidance and digital rights management," *Information Forensics and Security, IEEE Transactions on*, no. 99, pp. 1–1, 2011.

[20] http://www.hisinitiative.org.

[21] http://www.intrinsic-id.com.

[22] http://www.verayo.com.

[23] http://www.microsemi.com.

[24] R. Chakraborty and S. Bhunia, "RTL hardware IP protection using key-based control and data flow obfuscation," in *VLSI Design, 2010. VLSID'10. 23rd International Conference on*, pp. 405–410, IEEE, 2010.

[25] R. Chakraborty and S. Bhunia, "Security Against Hardware Trojan Attacks Using Key-Based Design Obfuscation," *Journal of Electronic Testing*, pp. 1–19, 2011.

[26] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Advances in CryptologyCrypto 2001*, pp. 1–18, Springer, 2001.

[27] B. Lynn, M. Prabhakaran, and A. Sahai, "Positive results and techniques for obfuscation," in *Advances in Cryptology-EUROCRYPT 2004*, pp. 20–39, Springer, 2004.

[28] S. Goren, O. Ozkurt, A. Yildiz, and H. Ugurdag, "FPGA bitstream protection with PUFs, obfuscation, and multi-boot," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pp. 1–2, IEEE, 2011.

[29] S. Goren, H. Ugurdag, A. Yildiz, and O. Ozkurt, "FPGA design security with time division multiplexed PUFs," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pp. 608–614, IEEE, 2010.

[30] S. Hauck and A. DeHon, eds., *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* Systems-on-Silicon, Elsevier, 2008.

[31] A. Maiti and P. Schaumont, "Improved ring oscillator PUF: an FPGA-friendly secure primitive," *Journal of cryptology*, vol. 24, no. 2, pp. 375–397, 2011.

[32] *Virtex-5 FPGA User Guide, UG190, xilinx.com.*

[33] *Spartan-6 FPGA Configurable Logic Block User Guide, UG384, xilinx.com.*

[34] Özgür Özkurt", "FPGA Design Security with PUF, Obfuscation, and Partial Reconfiguration," Master's thesis, Bahçeşehir University, Istanbul, 2012.

[35] `http://www.xilinx.com/tools/microblaze.htm`.

[36] *LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a), DS531, xilinx.com.*

[37] *XPS Block RAM (BRAM) Interface Controller (v1.00b), DS596, xilinx.com.*

[38] *LogiCORE IP Multi-Port Memory Controller (MPMC) (v6.04.a), DS643, xilinx.com.*

[39] *LogiCORE IP XPS HWICAP (v5.01a), DS586, xilinx.com.*

[40] *LogiCORE IP XPS Ethernet Lite Media Access Controller, DS580, xilinx.com.*

[41] `http://savannah.nongnu.org/projects/lwip/`.

[42] `http://www.digilentinc.com/`.

[43] S. Drimer, "Volatile FPGA design security–a survey," *IEEE Computer Society Annual Volume*, pp. 292–297, 2008.

[44] 2012. `http://www.erai.com`.

VITA

Abdullah YILDIZ

Education

Sep. 2010 to Sep. 2012

M.Sc. in Electrical & Electronics Engineering

Özyeğin University

Jan. 2009 to Sep. 2010

M.Sc. in Electrical & Electronics Engineering - Embedded Video Systems Program

Bahçeşehir University

Sep. 2004 to Sep. 2008

B.Sc. in Electronics Engineering

Uludağ University

Awards

Turkcell Akademi & TBD Scholarship for part of the thesis work (one-year)

Published Papers

Gören S., Yıldız A., Özkurt Ö., Uğurdağ H.F. FPGA Bitstream Protection with PUFs, Obfuscation, and Multi-boot, 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC2011), Montpellier, France, June

2011.

Gören S., Uğurdağ H.F., Yıldız A., Özkurt Ö, "FPGA Design Security with Time Division Multiplexed PUFs", International Conference on High Performance Computing & Simulation (HPCS'2010) , Caen, France, July 2010.

Gören S., Uğurdağ H.F., Özkurt Ö, Yıldız A. "PUF, DPSR ve Bulandırma Yoluyla Sayısal Yongaların Güvenilir Yapılması", EMO 3. Ağ ve Bilgi Güvenliği Ulusal Sempozyumu, Ankara, Turkey, Feb. 2010. (in Turkish)

Field of Study

- Reconfigurable Computing
- Embedded Systems
- Hardware Security