THE REPUBLIC OF TURKEY
BAHÇEŞEHİR UNIVERSITY

# FPGA DESIGN SECURITY WITH PUF, OBFUSCATION, AND PARTIAL RECONFIGURATION

**Master's Thesis**

**ÖZGÜR ÖZKURT**

**İSTANBUL, 2012**

THE REPUBLIC OF TURKEY
BAHÇEŞEHİR UNIVERSITY

THE GRADUATE SCHOOL OF
NATURAL AND APPLIED SCIENCES
ELECTRICAL AND ELECTRONICS ENGINEERING

# FPGA DESIGN SECURITY WITH PUF, OBFUSCATION, AND PARTIAL RECONFIGURATION

Master's Thesis

ÖZGÜR ÖZKURT

Supervisor: Assoc. Prof. Sezer GÖREN UĞURDAĞ

İSTANBUL, 2012

**THE REPUBLIC OF TURKEY**
**BAHÇEŞEHİR UNIVERSITY**

**THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES**
**ELECTRICAL AND ELECTRONICS ENGINEERING**

| | |
|---|---|
| Name of the Master's Thesis | : FPGA Design Security with PUF, Obfuscation, and Partial Reconfiguration |
| Name/Last Name of the Student | : Özgür Özkurt |
| Date of Thesis Defense | : April 26, 2012 |

The thesis has been approved by the Graduate School of Natural and Applied Sciences.

Assoc. Prof. F. Tunç BOZBURA
Acting Director
Signature

---------------------------------

I certify that this thesis meets all the requirements as a thesis for the degree of Master of Science.

Assoc. Prof. M. S. Ufuk TÜRELİ
Program Coordinator
Signature

-----------------------------------

This is to certify that we have read this thesis and that we find it fully adequate in scope, quality, and content, as a thesis for the degree of Master of Science.

| Examining Committee Members | Signature |
|---|---|
| Thesis Supervisor<br>Assoc. Prof. Sezer GÖREN UĞURDAĞ | --------------------------------- |
| Member<br>Asst. Prof. H. Fatih UĞURDAĞ | --------------------------------- |
| Member<br>Assoc. Prof. Levent EREN | --------------------------------- |

## ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor, Assoc. Professor Sezer Gören Uğurdağ, whose knowledge, patience, and understanding greatly enriched my graduate experience. She gave me this opportunity to explore new grounds in the design of programmable electronic systems. It’s her encouragement and guidance that brought this study to a successful completion.

I would also like to express my appreciation to my lifelong mentor Asst. Professor H. Fatih Uğurdağ, for his contribution to this work. Fatih is someone you will suddenly love and never forget once you meet him. He is by far the funniest professor and one of the smartest people I know. I hope that I could be as active and enthusiastic as he could and to someday be able to communicate with people as well as he can.

I would like to thank Assoc. Professor Levent EREN, for taking the time to share his professional experiences to improve the quality of this thesis and for being there as a member of the committee.

My gratitude to my parents İhsan and Nuran Özkurt, who have constantly encouraged my desire for learning, even when it meant us living separate lives for many years.
This thesis is dedicated to them.

Finally, I express my profound appreciation to my girlfriend Özge for her constant encouragement, understanding, and patience that guided me through the most difficult times of this study.

Özgür Özkurt

## ABSTRACT

FPGA DESIGN SECURITY
WITH PUF, OBFUSCATION,
AND PARTIAL RECONFIGURATION

Özgür Özkurt

Electrical and Electronics Engineering

Thesis Supervisor: Assoc. Prof. Sezer Gören Uğurdağ

April, 2012, 50 Pages

FPGAs contain reconfigurable digital circuits and offer parallel computing for a very broad range of applications. With Dynamic Partial Self Reconfiguration (DPSR), FPGAs offer even more computing power per unit chip area. DPSR allows an FPGA to reprogram itself partly during run-time and hence lets different hardware modules use the same chip area through time multiplexing. Unfortunately, DPSR is only offered on high-end FPGAs (problem 1). Another problem is that FPGA configuration bitstreams can be cloned like any other firmware. Some FPGAs support encrypted bitstreams. However, that is again available only on high-end FPGAs (problem 2). On top of that, encrypted partial bitstreams to be used with DPSR is a very useful feature, but then it is currently not offered on any FPGA (problem 3). With this thesis, we address all these problems through a methodology that implements DPSR and protected partial bitstreams on low-end Xilinx Spartan-6 FPGAs. This methodology can also be used with high-end Xilinx FPGAs thus letting users avoid expensive license fees of associated high-end tools. Our methodology supports modular partial reconfiguration and hence scales to cases where there are large differences between subsequent configurations. We offer bitstream protection through a Physical Unclonable Function (PUF) and HDL-level obfuscation. Obfuscation makes reverse engineering quite difficult, and our DPSR approach has only one percent area overhead.

**Keywords**: FPGA, Dynamic Partial Self Reconfiguration, Obfuscation, PUF.

# ÖZET

FKF, BULANDIRMA VE KISMİ YAPILANDIRMA İLE
SPKD'LARIN GÜVENİLİR YAPILMASI

Özgür Özkurt

Elektrik-Elektronik Mühendisliği

Tez Danışmanı: Doç. Dr. Sezer Gören Uğurdağ

Nisan, 2012, 50 Sayfa

Sahada Programlanabilir Kapı Dizinleri (SPKD) yeniden yapılandırılabilen sayısal devreler içeren, çok farklı uygulama alanlarında paralel programlama imkânı sunan yongalardır. Dinamik Kısmi Kendi kendini yeniden Yapılandırma (DKKY) özelliği ile SPKD yongaları birim alanda daha fazla işlem gücü sunabilmektedir. DKKY, SPKD'ye çalışma esnasında kendi kendisini yeniden yapılandırma olanağı sunarak birbirinden farklı donanım modüllerinin aynı kısmi yonga alanını farklı zaman aralıklarında kullanabilmesine imkân sağlar. Fakat bu uygulama sadece üst sınıf SPKD'ler için sağlanmıştır (problem 1). Bir diğer problem ise SPKD'nin programlama bit dizinlerinin kolaylıkla kopyalanabilmesidir. Bazı SPKD yongaları bu problemin üstesinden şifrelenmiş bit dizinleri kullanarak gelebilmektedir. Ancak bu özellikte yalnızca üst sınıf SPKD'lerde mevcuttur (problem 2). Bu özelliklere ek olarak, DKKY uygulamalarında şifrelenmiş kısmi bit dizinleri kullanılarak daha üstün güvenlik önlemleri sağlanabilir; ancak halihazırdaki herhangi bir SPKD yongasının böyle bir özelliği bulunmamaktadır (problem 3). Bu tez çalışması ile bu problemlerin hepsine bir çözüm olabilecek, alt sınıf Xilinx Spartan-6 SPKD yongaları üzerinde DKKY uygulamaları ve güvenli kısmi bit dizinleri oluşturulabilen bir yöntem sunuyoruz. Sunduğumuz DKKY yöntemi, modüler DKKY olanağı sunmaktadır. Bu özellik sayesinde, her yeni yapılandırmada SPKD tasarımında büyük değişiklikler yapılabilir. Ayrıca, sunduğumuz bu yöntem üst sınıf SPKD'ler için de uygulanabilir ve bu sayede kullanıcılar ilgili uygulamalara pahalı lisans ücretleri ödemekten kurtulabilirler. DKKY yöntemi ile birlikte, Fiziksel olarak Klonlanamaz Fonksiyonlar (FKF) ve donanım bulandırma yöntemlerini kullanan bir bit dizini güvenlik tekniği sunmaktayız. Kullandığımız yöntemler, SPKD bit dizinleri üzerinde tersine-mühendislik uygulamaları yapılmasını oldukça zorlaştırmaktadır. Oluşturduğumuz DKKY kontrol modülü, SPKD üzerinde sadece yüzde birlik bir alanı kullanmaktadır.

**Anahtar Kelimeler**: Sahada Programlanabilir Kapı Dizinleri (SPKD), Dinamik Kısmi Kendi kendini yeniden Yapılandırma (DKKY), Donanım Bulandırma, Fiziksel olarak Klonlanamaz Fonksiyon (FKF).

## TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| ASIC: | Application Specific Integrated Circuit |
| CDMA: | Code Division Multiple Access |
| DPSR: | Dynamic Partial Self Reconfiguration |
| DRC: | Design Rule Check |
| DSP: | Digital Signal Processing |
| FF: | Flip-Flop |
| FPGA: | Field Programmable Gate Array |
| GSM: | Global System for Mobile Communications |
| ICAP: | Internal Configuration Access Port |
| IP: | Intellectual Property |
| JTAG: | Joint Test Action Group |
| LUT: | Look-Up Table |
| LVDS: | Low-Voltage Differential Signaling |
| MUX: | Multiplexer |
| NCD: | Native Circuit Description |
| NRE: | Non-Recurring Expenses |
| PAR: | Place and Route |
| PR: | Partial Reconfiguration |
| PUF: | Physical Unclonable Function |
| SPI: | Serial Peripheral Interface |
| Tcl: | Tool command language |
| UMTS: | Universal Mobile Telecommunications System |

# 1. INTRODUCTION

The focus of this thesis is directed towards examining, evaluating, and improving upon specific areas relating to FPGA design flow and its security. Our contributions in this thesis would greatly extend the application range on low cost FPGAs.

## 1.1 OVERVIEW

Due to the rapid advancement in the semiconductor industry, the performance of low-cost FPGAs has dramatically improved. Modern FPGAs offer great features. Today's FPGAs usually come with PLLs, these can transceive LVDS signals, either serialize or de-serialize parallel IOs to, or from multiple GHz signal speeds. They have DSP blocks, large memory blocks, soft IP cores, and microprocessor cores. Furthermore, FPGAs are reconfigurable in their nature. This enables to make instantaneous modifications in the design, as opposed to their counter-part ASICs where design modifications are almost impossible without a complete redesign of the entire system.

In an effort to keep up with the ever-increasing demands of today’s electronics consumer from public to military sector, the emphasis is mostly on the rapid continuous releasing of upgrades of the same products. Choosing FPGAs over ASICs can significantly decrease the NRE and the time to market. Thus, this enables semi-conductor companies to meet the demands of the highly competitive market.

Modern electronic systems have to incorporate a multitude of different protocols in order to meet required standards. For example, a cell phone should be able to work on multi-band GSM, UMTS, and CDMA networks in different regions worldwide. These networks are built for the same purposes but they have specific different protocol requirements with different circuitry. Whilst not all these protocols are used at the same time, modern cell phones have the necessary components for each to enable global roaming. In another example, a modern TV-set is able to process SD and HD video streams and display 3D movies. Whilst an SD stream needs image enhancements on-the fly, an HD stream can be in various compression formats, and a 3D video may need post processing with special equipment (3D glasses, LCD panels) control. These requirements, whilst not used simultaneously, are all necessary. With an FPGA, a cell phone is able to reconfigure itself entirely or partially to work on different communication networks or an FPGA on a TV-set could reconfigure internal sub designs to process different video streams. This ability is called dynamic partial self-reconfiguration (DPSR). Up until now, Xilinx FPGA tools supported partial reconfiguration (PR) flow for only high-end FPGA families. There is an alternative method for all FPGA families (Difference based PR) but it is for making minor modifications only. Instead of PR, Xilinx offers a method called MultiBoot for the low-end FPGA families. This method enables only the complete reconfiguration of the FPGA; thereby its reconfiguration time and memory requirements are vastly greater than PR method. Moreover, the FPGA would become inaccessible during reconfiguration with a resulting loss of clock and data synchronization, which would then require additional total system resetting sequences.

In the FPGA design flow, the definition of the circuit design is generated as a configuration file for the target FPGA platform. The circuit definition is ciphered into a particular file format (i.e. Xilinx bitstream), which is kept as classified information by the FPGA manufacturers. This configuration file can be programmed to the FPGA from a non-volatile storage (i.e. Flash-ROM), or from a controller, through various protocols. The onboard storage is generally the preferred method. However, this flexibility gives rise to security issues as the transmission and storage of bitstream might be compromised. It can be copied from the storage device or can be captured on the fly from its configuration path. To address bitstream protection, encryption methods are available but only for the high-end FPGA families. Xilinx offers a feature called "Device DNA", available on low-end Spartan-6 FPGAs. This factory-set individual chip ID enables authentication at least to prevent cloning (but not reverse engineering).

The fast design flow, reconfiguration ability, new and improved components like internal ARM cores (Xilinx Inc. 2011a) turns modern FPGAs into a powerhouse for IC developers. However, a vital feature (partial reconfiguration) is missing and bitstream protection is not completely available for the low-cost FPGAs nor are the necessary PR tools for the high-end FPGAs free of charge.

## 1.2 MOTIVATION AND OBJECTIVES

Recent version of Xilinx design tools provides no modular PR support for the low-end FPGAs such as the Spartan-6, despite the fact that these FPGAs do not have any physical constraint preventing PR applications. In addition, the last free version of Xilinx modular PR tools (Xilinx Inc. 2006) (Xilinx Partial Reconfiguration Early Access Software Tools), which only provided support for the high-end FPGA families such as the Virtex-4 and 5 respectively is based on an outdated Xilinx ISE version 9.2i. Developing a PR toolkit (and a design flow, if required), which is not only compatible with recent Xilinx tools, but also supports the low-end FPGA families would allow advances in design productivity to be closely followed, and give rise to the provision of low-cost FPGA solutions.

The most significant security problem of the FPGA is the loading of the bitstream, during boot-up time, from an off-chip source. Whilst this configuration method offers many advantages, it also exposes the FPGA and its bitstream to various attacks. Some high-end Xilinx FPGAs do possess the ability to process encrypted bitstreams thus enabling preservation of the confidential information held within. The low-end FPGAs in the Spartan-6 Family, however, have no such system of protection. There is an alternative method for encryption called Hardware Obfuscation. It modifies the description of the hardware to conceal its functionality. Obfuscation makes reverse engineering significantly more difficult and so it can be a real alternative of encryption for low-end FPGAs.

Authentication information is required for the application of obfuscation techniques within the FPGA designs. The “Device DNA” feature of Spartan-6 FPGAs is offered for this purpose, which is a factory-set serial number. Using Device DNA a protection system can be designed to detect over-building or to prevent cloning. Whilst this information is unique to every FPGA, it can be retrieved from the device very easily. This authentication information should only be accessible within the secured circuit. Using Physically Unclonable Functions (PUFs) for this purpose can offer an intrinsic fingerprint for any FPGA. This is accomplished by exploiting the uniqueness of slight manufacturing variations between chips. Using a PUF circuit suitable for FPGA fabric, a volatile fingerprint to be used as an authentication key, can be generated.

The objective of this thesis is twofold: (1) To create a novel DPSR design methodology with the tools necessary to enable usage of reconfiguration on the low-end Xilinx Spartan-6 FPGAs. (2) To propose an FPGA bitstream protection method based on PUF keys with HDL-level active obfuscation. We are targeting Xilinx Spartan-6 FPGAs but our techniques can also be used with high-end Xilinx FPGAs. The steps required to achieve these objectives will be explained next.

In order to add DPSR ability to the Spartan-6 platform, we need to develop a novel DPSR design flow using available Xilinx tools and create the missing parts. Our design methodology should be based on Xilinx Difference Based PR (Eto, 2007) method since it can be used with low-end FPGAs. However, it was created for only making very small modifications on FPGA designs. We have to modify this design approach to enable making large and modular modifications on FPGA designs.

We have to develop a PR controller hardware module that reconfigures the PR modules to the FPGA internally by use of internal configuration access port (ICAP) on the Spartan-6. In addition, we need a compression software, decompression hardware pair to minimize the memory requirements of the PR applications.

In order to offer an IP protection technique in place of encryption techniques, we need to implement provably secure obfuscation (Koushanfar 2012). For its realization, we need unclonable authentication keys. These keys can be generated with PUF circuits. We have to evaluate and experiment several different PUF circuits to select the most appropriate PUF design for the Spartan-6.

Finally, we need to evaluate our PUF key-based obfuscation with DPSR-LD technique in terms of timing, performance, and resource utilization.

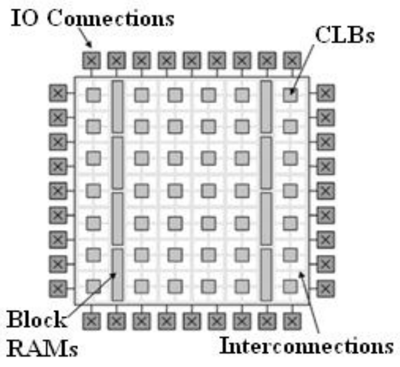
# 2. BACKGROUND AND RELATED WORK

In this chapter, all relevant background information and related work pertaining to this thesis will be covered. We will provide some basic information about FPGAs and its development flow, DPSR, hardware obfuscation, and PUFs.

## 2.1 FPGA

FPGAs are semi-conductor devices that can be re-programmed following manufacture. These devices contain logic components, inter-connections, and input/output blocks (IOBs) that are available for re-programming. Due to this ability, the FPGA design can be changed as many times as is required, rather than being restricted to one pre-determined function. This ensures that an FPGA's function is unknown during production time unlike its counter-part, the ASIC (Application Specific Integrated Circuit). When comparing the FPGA device with that of their ASIC equivalent within the same price bracket it becomes apparent that (due to their re-configuration ability) the FPGA is the slower device. However, they do possess numerous advantages such as a shorter time to market, re-usability, rapid prototyping, and de-bugging on the target hardware in-field updates and lower NRE costs.

**Figure 2.1: An example of an FPGA layout**

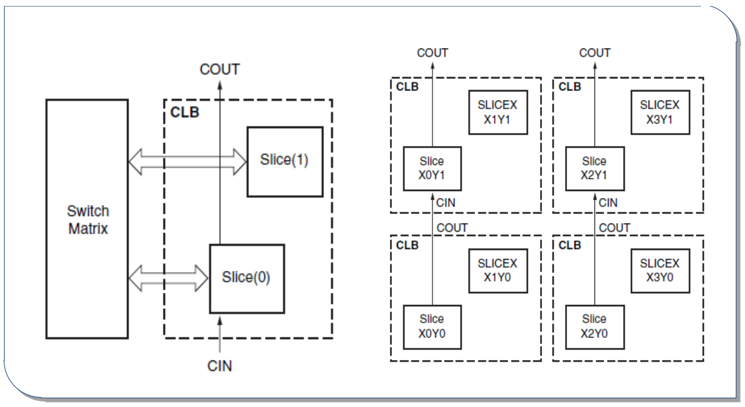


The programmable logic components of FPGAs (configurable logic blocks or CLB) can perform any logical functionality ranging from basic logic gates (AND, OR, XOR, NOT) to the more complex combinational or sequential functions. In order to implement complex functionalities most of the FPGAs have internal memory components (Flip Flops (FFs), Block RAMs). The desired logical function is achieved by joining many CLBs with programmable interconnects. A simple overview of an FPGA layout is illustrated in Figure 2.1.

### 2.1.1 FPGA Configurable Logic Block

In Xilinx Spartan-6 FPGAs, a CLB element contains a pair of slices shown as in Figure 2.2 (Xilinx Inc. 2010). These two slices are not directly connected to each other but some have connections with their vertical neighbors through carry-chain paths. The Xilinx tools label slices starting from bottom-left corner of the die, with an “X” followed by a number that defines the column position of a slice following with an “Y” and another number that identifies the row position of the CLB(the row number is equal for the slices in the same CLB). The labeling counts to top-right corner with this sequence.

**Figure 2.2: Xilinx FPGA CLB arrangement**



*Source: UG384 Spartan-6 Configurable Logic User Guide, 2010*

Every slice contains four look-up tables LUTs and eight Flip-Flops. SLICEX is the base slice with only these elements. SLICELs also contain an arithmetic carry structure that column, and multiplexers. The SLICEMs hold the carry structure and multiplexers in addition have the ability to use the LUTs as 64-bit distributed RAM and shift registers.

Each column of CLBs has two slice columns. One column is a SLICEX column, the other column substitutes between SLICEL and SLICEMs. Thus, approximately half of the available slices are of type SLICEX, while one-fourth of them are type SLICEL and the rest is type SLICEM.

### 2.1.2 FPGA Design Flow

We will now explain how the desired circuits are brought to life using the FPGA design flow. Whilst this flow explanation is based upon the FPGA working in conjunction with Xilinx development tools (Xilinx Inc. 2012), it should give a general perspective for other FPGA development platforms too.

#### 2.1.2.1 Design entry

The most important stage in the FPGA design flow is the design entry stage. This is where the functionality of the FPGA is designed, as well as the selection of both the target platform and communication standards. The Hardware Description Language (HDL) code is the most popular form of design entry, although many other alternatives do exist, such as, schematic, block, and finite state machine (FSM) diagram creation, as well as, the incorporation of a third party IP, and automatically generated cores. There are two different approaches during this stage: the first being the designing of the desired functionality, this is followed by the selection of an FPGA platform with enough resources for the realization of the system. This method is mainly used in applications where the realization of the project is more important than the cost of the FPGA platform (such as a design for test DFT applications). In the second approach, however, the design of the system is dependent upon the limitations of the target FPGA platform; all subsequent success concerning the flow being dependent upon correct planning during this stage.

#### 2.1.2.2 Simulation

The next step is the verification of the design functionality. The design is compared to its mathematical model or the results of the desired functionality in a simulation platform within the help of different complex test bench applications. There are several popular options for FPGA design simulation including tools from Mentor Graphics, Cadence, and Xilinx themselves.

#### 2.1.2.3 Synthesis

During the synthesis stage of hardware implementation, the description of the design from the previous stages then converts this definition in terms of standard digital logic elements such as look-up tables (LUT), logic gates(i.e. and, or, …etc.), and memory elements (i.e. flip-flops). The tools used in this stage (for example Xilinx Synthesis Technology (XST)), can optimize the given design in terms of area, power, or performance. The result is then recorded onto a net-list file format such as Native Generic Circuit Description (NGC) or industry standard EDIF.

#### 2.1.2.4 Translation

This is usually the final stage in which, one can include any further information relevant to the FPGA design. It is also when the synthesized design is combined with User Constraints (UCF) files to a database for the creation of the following stages of implementation called Native Circuit Description (NCD). The UCF files contain all relevant timing, placement, and IO assignment-related information required for realization.

#### 2.1.2.5 Technology mapping

Technology mapping is when the existing native netlist is adapted to incorporate and contain only logic cells supported by the targeted FPGA architecture. This is achieved by matching the native logical expression with the logical resources available within the FPGA platform such as, LUTs and FFs. In Xilinx FPGAs these logic resources are packed into groups called configurable logic blocks (CLB). Xilinx MAP utility is also responsible for matching the suitable logic components together.

#### 2.1.2.6 Placement and routing

The logical resources, along with, their interconnections are arranged in a specific layout with-in the FPGA. The stage of placement and routing (PAR) is when the definition of the design is located on the target FPGA platform, and their internal and IO connections are formed. The locations of these resources should be carefully selected in order to ensure that the design connections are routable, user timing and placement constraints are met, and certain physical problems are eliminated so as to protect the FPGA from damage (design rule checking DRC). Following this, the initial design in the NCD library is converted into a technology-mapped PAR’ed design suitable for the targeted FPGA architecture. The generated library is the last accessible file where modifications of the design can be made with the use of a specific tool such as the Xilinx FPGA Editor.

#### 2.1.2.7 Bitstream generation

Finally, the design cooked in the previous stages should be converted into a file format that can be downloaded onto the FPGA; this is known as bitstream in the Xilinx design flow. The BitGen utility of Xilinx is responsible for the extraction of all necessary configuration data for the FPGA platform. This tool also performs additional DRC tasks on the design ensuring that the FPGA platform will not be damaged by its configuration.

### 2.1.3 FPGA Configuration

The process of downloading the bitstream to the FPGA platform is called configuration. Due to the volatile nature of the FPGA configuration memory, re-programming is required with every system power up. This can be done from either a non-volatile external storage (ex. a Flash-ROM), or from another controller (Xilinx Inc. 2011b). In addition, the circuit design of an FPGA can be changed by using a different bitstream. Xilinx offers a method called MultiBoot (Hussein and Patel 2008), which enables users to configure the FPGA with different bitstreams, stored in different memory locations or storage devices. By using this method, the FPGA can be triggered to reconfigure another entire design to its memory. There are several configuration mediums for Xilinx FPGAs including the most common Boundary-Scan (JTAG), SPI, and ICAP that are explained below.

**Boundary-Scan (JTAG):** This is an industry standard (IEEE 1149.1, and 1532) serial programming mode. External logic from a cable, microprocessor, or other device is used to drive the JTAG specific pins, Test Data In (TDI), Test Mode Select (TMS), and Test Clock (TCK) and sense device response on Test Data Out (TDO). This is the most popular mode thus many other logic devices (PROMs, Microprocessors, and PLDs) can also be configured through this medium.

**Serial Master-Slave Mode (SPI):** This is the simplest method of all FPGA configuration modes and is compatible with all of the Xilinx FPGA families. Using this mode, the FPGA connects to a serial communication medium through as few ports as possible (typically data is transferred through a single port), and it is this mode that is selected for systems where the FPGA has fewer IO pin packages.

**ICAP:** This XILINX specific medium is used to access the FPGA configuration memory internally by the configured design. It is in this mode that the FPGA can be partially re-configured and its internal configuration controller can be managed.

## 2.2 DYNAMIC PARTIAL RECONFIGURATION

Reconfiguration is the ability to change the functionality of the physical circuitry of an IC, which can then be altered to meet the new requirements of the system. In this way, the reconfigurable ICs can be used for many applications that require different functionalities for varying purposes. A system containing a reconfigurable IC can change its functionality at a rapid rate without disrupting the process of unchanged logic. It is due to this ability that the reconfigurable ICs appear in a wide and varied range of applications such as Data Compression, Audio Video Enhancement, and High

Performance Computing.

The most common type of reconfigurable hardware device is an FPGA. Whilst the FPGA configuration method offers reprogramming flexibility, PR takes it one-step further. PR gives users the ability to modify a portion of an FPGA design whilst leaving the remainder of the system unchanged. After the full configuration of an FPGA, partial bitstreams of desired modification can be downloaded to the FPGA without corrupting the integrity of the unmodified part of the design. In a PR application, the modifications applied to the initial design use partial and smaller bitstreams. By using smaller bitstreams, the reconfiguration period is shortened and the amount of additional storage is reduced.

Looking at the PR from an operational perspective, it can be divided into two different groups; Dynamic PR (DPR) and Static PR (SPR). In SPR, the FPGA is not active during reconfiguration (whilst the partial bitstream is loading to the FPGA), the device is stopped and restarted after configuration is completed. SPR is a technique that is now considered obsolete and has since been replaced by DPR. This DPR technique allows the device to not only reconfigure during runtime, but also allows the unmodified parts of the design to continue to function as normal whilst doing so. DPR also allows the creation of efficient systems with FPGAs where devices operate in a mission critical environment that cannot be disrupted while some subsystems are being redefined.

Another key advantage is that DPR can be controlled from inside the FPGA with the help of a processor or dedicated core (DPSR). If the partial bitstreams are small enough to fit into a reasonable amount of memory units inside the FPGA, the system can then be designed to operate completely inside. Moreover, there are application areas where the DPR is the undisputed method of choice, such as evolvable hardware (Upegui and Sanchez 2005, pp. 56-65) and fault-tolerance (Emmert, Stroud, Skaggs and Abramovici 2000, pp. 165-174).

While DPR offers considerable advantages, it does have its drawbacks; to develop a

DPR application, there are several extra processes that need to be incorporated in both design and implementation stages. First, the design should be partitioned into static and time-varying (dynamic) parts and then be developed with the requirements of these partitions being taken into consideration. For instance, the designer should eliminate the dependencies between the dynamic modules since they do not exist with-in the system simultaneously. There are also several physical limitations due to the FPGA platform and project requirements. The physical placements of the static and dynamic parts are separated to different parts of the FPGA layout during implementation. The designer should floor plan this layout providing resources for both parts adequately.

There are different approaches to create a hardware design including DPR. The two of them that are offered with recent Xilinx tools are explained below.

### 2.2.1 Difference based Partial Reconfiguration

Difference based PR (Eto 2007) is method for making small changes to the FPGA functionality. It is particularly useful in cases where a simple low-level modification is required after implementation. This modification can be done with Xilinx FPGA Editor Tool, which allows users to modify the LUT contents, I/O standards, and block RAM (Random Access Memory) contents, and routes between the logic components of an implemented design. The designer generates the original NCD library and bitstream first. After that, the designer modifies this NCD library using the FPGA Editor, thus the modified design is created. The designer can create a partial bitstream that reflects the difference between the original and the modified bitstream design using BitGen (with "-r" switch). BitGen by way of comparison is able to generate a partial bitstream containing only the required configuration information necessary for conversion from the original to the modified design. The size of this partial bitstream depends on the type and number of modifications applied and normally it is orders of a magnitude smaller than the original bitstream. Since the length of the reconfiguration period is parallel to the size of the bitstream, it takes fractions of full bitstream configuration time to reconfigure the FPGA with this partial bitstream.

Difference based PR is useful when making small changes after implementation but proves to be inadequate where complex modifications that include routing changes are required. The theory behind this method being that the new design is derived from the original. The BitGen tool does not compare the functionality of these designs; it only compares the configuration of the logic resources on the FPGA. The only way to verify the integrity of the resultant circuit following PR is by emulation. In (Eto 2007), Emi ETO suggests that while it is possible to use difference based PR to make routing modifications on the design, it is not recommended due to the risk of internal contention during reconfiguration. The risk being that, one of the newly reconfigured circuit's interconnections may form a link with the previous designs active components, before their removal. There is a great possibility that the driving logic of two different designs may be connected to the same net, resulting in two drivers for a single input pin of a component causing electrical failure or permanent damage to the device. Therefore, the limits of use for the use of this method are very narrow indeed.

### 2.2.2 Module based Partial Reconfiguration

This (Xilinx Inc. 2011c) is a more complex and general-purpose methodology. It allows the designer to create modular reconfigurable blocks in the design. The design is separated into its static and dynamic parts. For any different configuration, the dynamic part of the design should be reconfigured while the static part is supposed to remain the same. Each reconfigurable (dynamic) module for a dynamic region must have the same set of module ports (Bus Macros in EAPR (Xilinx Inc. 2006)). These ports ensure that the interface of the dynamic part remains steady during reconfiguration. The static part and the dynamic modules can be developed and implemented in parallel. A region on the FPGA layout should be specified to the dynamic part (region) so that dynamic modules do not overlap with static part in any reconfiguration attempt, as this can corrupt the design. Of course, it is possible to have more than one dynamic region on the FPGA layout.

The floor planning and the implementation of the design can be done with Xilinx PlanAhead tool. Alternatively, the designer can manually utilize individual flow tools with a batch script and a user constraints file. In this flow, NCD libraries of static part and dynamic modules are created separately within specific guidelines. Using these libraries, the BitGen tool is capable of generating both entire and partial bitstreams.

The last free Xilinx toolkit that supported this flow was the Early Access Partial Reconfiguration (Xilinx Inc. 2006). This was deprecated with the release of the paid license version of the current tools (Xilinx Inc. 2011c). In both versions, DPR was only supported for high-end FPGAs such as Virtex-5.

### 2.2.3 Bitstream Compression

The configuration overheads limit the practicality of PR applications. The size of the partial bitstream combined with the speed of the reconfiguration method outlines the performance of these applications. To improve this performance, we can either increase reconfiguration speed or reduce the size of bitstreams. Bitstream compression techniques offer both timing and memory reductions for PR applications. Regarding bitstream compression, Li et al. (Li and Hauck 2001, pp. 147-159) investigated the redundancy in various bitstream files and applied compression algorithms including Huffman coding, Arithmetic coding, and LZ compression. Their simulation results indicated that a compression ratio of 4:1 can be achieved. Dandalis et al. (2001) proposed a dictionary-based compression approach. They showed 11 to 41 percent savings in memory for configuration bitstreams of several applications. Pan et al. (2004) proposed intra-bitstream compression technique for Xilinx Virtex family as opposed to inter-bitstream compression techniques. They reported that their approach achieved 27 to 76 percent improvement over the DV and LZSS algorithms.

Most proposed bitstream file compression techniques are based on complicated compression algorithms in order to achieve high compression ratios. Koch et al. (2009) investigated several compression algorithms with respect to the achievable compression ratio, throughput, and hardware overhead. They reported that Huffman encoding enhances the compression ratio to almost 40 percent and a combined LZSS and Huffman encoding to 34 percent. However, in order to achieve this better compression ratio a much larger resource overhead must be allocated for implementing the hardware decompressor. In (Koch, et.al. 2009) it is also reported that about 5 thousand LUTs are required for the combined LZSS-Huffman accelerator whereas less than 100 LUTs are needed for run-length encoding and LZSS hardware decompressors with 50 percent compression ratios. Liu et al. (2010) proposed a simple compression/decompression technique in which a code word including the count of the repeated words is entered after the repeated words for Xilinx Virtex-4 FPGA.

## 2.3 HARDWARE OBFUSCATION

This is a method for hiding the real functionality of a design by intentionally modifying the definition of the hardware. It is an authentication method as opposed to the digital watermarking methods for the reason that the design can be effectively protected from modifications or from IP thefts. Similar to the other security protocols, a key of authentication is required for obfuscation applications. The Hardware Obfuscation techniques can be divided into two categories; Passive and Active.

### 2.3.1 Passive Obfuscation

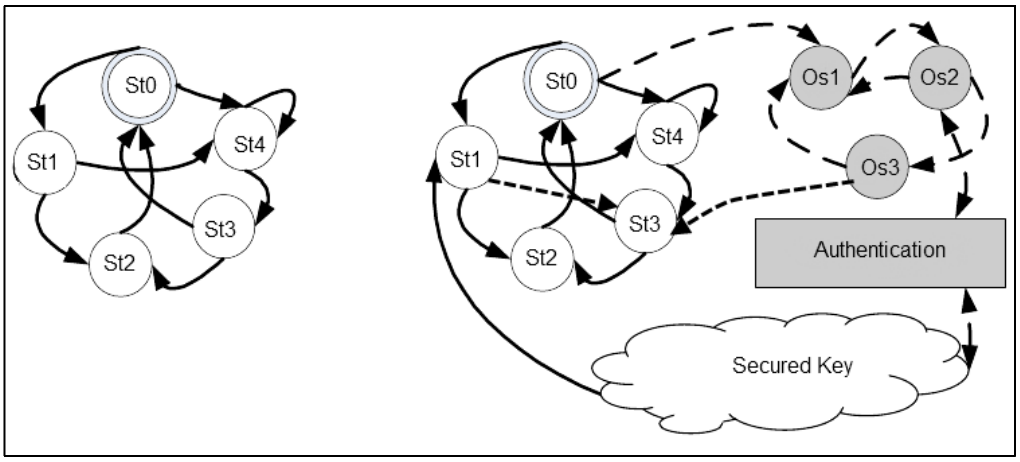
Passive Obfuscation methods are based on the reading skills of humans regarding the digital circuits. In this work (Brzozowski and Yarmolik 2007), the definition of the circuit is altered using the structural variations in its HDL description. In addition to this process, the names of the logic components are changed with randomly generated tags that bare no relation to their actual functionality This method modifies the description of the circuit, during the design entry stage, making it difficult for readers to understand the functionality of the circuit.

The most important problem of Passive Obfuscation is that applying these techniques does not prevent the usage of these modules since the functionality of the design is not modified. The design could be used as a black-box module in a project if the designer does not intend to change the functionality of the IP. Even if there are modifications necessary, the general structure of the design could be reversed from the obfuscated description.

### 2.3.2 Active Obfuscation

Active Obfuscation methods are directly applied with-in the definition of the design. The protected design is active (but functioning falsely) before the design is authenticated thus making tampering and/or brute-force attacks significantly more difficult for a perpetrator to instigate as the correct functionality of the circuit is not easy to determine. This can be accomplished by introducing additional Obfuscation modules to the system (e.g. finite state machines FSM). These modules can be introduced to the circuit design during the design entry stage or following the synthesis stage in the FPGA design flow, such as this application (Chakraborty and Bhunia 2010, pp. 405-410) or this (Chakraborty and Bhunia 2008, pp. 674-677).

**Figure 2.3: Normal data flow vs. Obfuscated data flow**



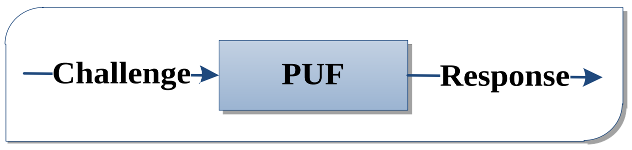
Obfuscation received skepticism (Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan and Yang 2001) in the past. Barak et al. (2001) proved the existence of certain classes of functions that cannot be obfuscated. However since then, several works

(Gören et al. 2010), (Lynn, Prabhakaran and Sahai 2004), (Chakraborty et al. 2010) have shown the feasibility of secure “key-based” obfuscation. Recently, Koushanfar (2012) 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 (Gören et al. 2011), we combined PUF key-based obfuscation and multi-boot feature of Spartan-6 devices to achieve full bitstream protection. In multi-boot, FPGA has to overwrite its configuration completely and externally from a Flash memory.

## 2.4 PHYSICAL UNCLONABLE FUNCTION

PUFs consist of inherently unclonable physical systems. This attribute derives from the fact that they consist of many random uncontrollable components that are present during the manufacturing process of electronic circuits. In particular, a PUF is considered as a function, one that maps challenges to responses (Figure 2.2). This function can only operate inside the physical system, and the responses from every instance are unique.

**Figure 2.4: PUF Black-Box Model**



PUFs have the ability to generate and securely store highly secret data without the requirement for non-volatile storage. PUFs can be created from many physical properties i.e. capacitance, resistance, and timing delay information. Pappu et al. (2001), (Pappu, Recht, Taylor, and Gershenfeld 2002) first proposed the concept of PUFs based on the scattering obtained when shining a laser on a bubble-filled transparent epoxy wafer. Silicon Physical Random Functions were proposed by Gassend et al. (Gassend, Clarke, van Dijk, and Devadas 2002a), which use manufacturing process variations in ICs with identical masks to uniquely characterize each chip. A parameterized self-oscillating circuit is developed to measure the frequency, which characterizes each IC.

Lim et al. (Lim, Lee, Gassend, Suh, van Dijk, and Devadas 2005) proposed Arbiter based PUFs, which use a differential structure and an arbiter to distinguish the difference between the path delays. Gassend et al. (2002b) later introduced a Controlled Physical Random Function, which can only be accessed via an algorithm that is physically bound to the randomness source in an inseparable way to protect a weak PUF from external attacks. Su et al. (Su, Holleman and Otis 2007) proposed a custom-built array of cross-coupled NOR gate latches to uniquely identify an IC. Suh and Devadas (Suh and Devadas 2007) proposed a PUF based on ring oscillators, which can also be implemented on an FPGA. Kumar et al. (2008) proposed Butterfly PUF based on cross-coupled latches, which can also be implemented on FPGAs. Recently, Anderson (Anderson 2010) proposed a PUF design based on shift-registers and specifically for FPGA implementations.

The focus of this chapter will be directed towards silicon based PUFs that are compatible with the FPGA platforms. These PUF designs are based on active circuits meaning that their implementation and their responses are volatile with regards to the rest of the FPGA design. These PUFs are based on timing and delay variations of internal circuitry components inside the FPGAs. The variations during the manufacturing process can cause significant delay differences among identical components and, as a direct result, also between identical FPGAs. This variance is the source of the authentication information extracted from PUF circuits. These circuits do not require any special manufacturing process, programming, or testing steps. The designer can design these circuits with simple RTL coding and using of specific placement constraints. This design can be used as functional modules that generate authentication information. The usage of these modules provides invaluable information regarding security factors, in doing so they consume a certain amount of the logic resources from with-in the FPGA platform.

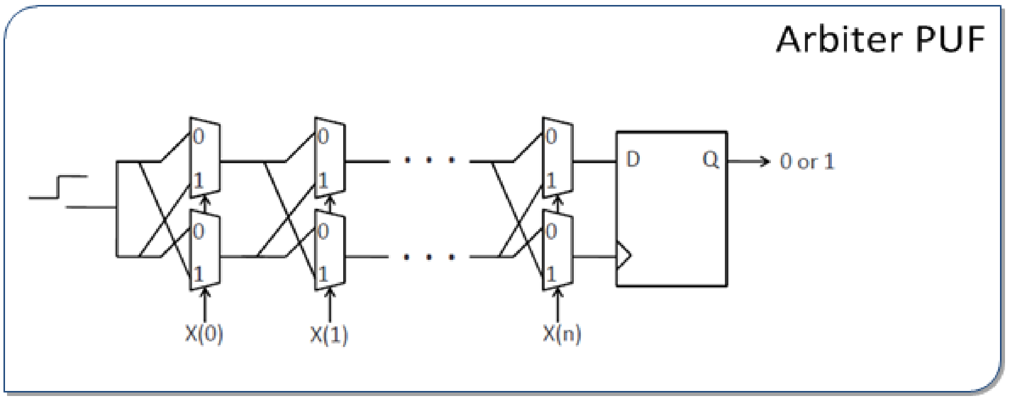
### 2.4.1 Silicon PUF Constructions

The following is a review of some of the most important types of silicon PUFs that are compatible with the FPGA platforms.

#### 2.4.1.1 Arbiter PUF

This is a type of PUF circuit, which extracts the random variance regarding the delay information of digital multiplexers (Lim, et. al. 2005). In this PUF construction, two symmetrical digital delay lines are implemented (Figure 2.5) within the FPGA layout and both are triggered simultaneously. Due to the random variations on the paths, the propagation delay of one will be slightly shorter than the other line. The end of these delay lines are connected to an FF, which samples the digital signal value in the D-input on the rising edge of the clock signal. The top delay line is connected to the D-input of the FF, while the bottom delay line is connected to the clock port. Therefore, if the propagation delay of the top line is shorter than the bottom line, there will be a digital high on the D-input before the clock signal rises, and the FF will capture this value indicating that the top line has a shorter propagation delay. On the other hand, if the bottom line has a shorter propagation delay, the result will be the opposite since there will not be a digital high on the D-input when the clock signal arises.

**Figure 2.5: Arbiter PUF**

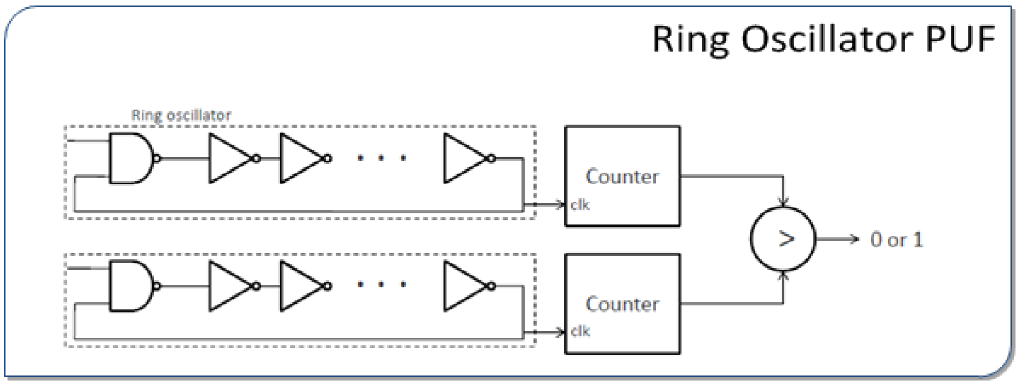


As shown in Figure 2.5, creating this delay path with the serial concatenation of multiplexers (MUX), one arbiter PUF instance can generate $2^N$ different values where N is the number of MUX pairs. This particular PUF design requires identical routing of the delay lines, which is practically impossible using standard Xilinx tools since these routes, between the logic resources of this PUF, can only be routed using programmable interconnections on the FPGA layout. The routing process of PAR tools cannot be limited to just the selection of specific interconnection paths. There are applications where the designer can manually route the interconnections of the PUF instances using the FPGA Editor, but this method has proven to be highly impractical. Even if the routing is performed correctly, there would be a significant difference concerning the lengths of parallel lines in some of the challenge configurations, and the generated result would reflect the length difference, and not the delay variance of these paths.

#### 2.4.1.2 Ring oscillator PUF

Just like the Arbiter PUF, the Ring Oscillator PUF extracts the delay variations of logical components. However, with this implementation (Suh and Devadas 2007), instead of directly making comparisons of two different pairs, the delay path is transformed into a Ring Oscillator (RO). This is achieved by feeding back its inverted output to its input (Figure 2.6).

**Figure 2.6: Ring oscillator PUF**



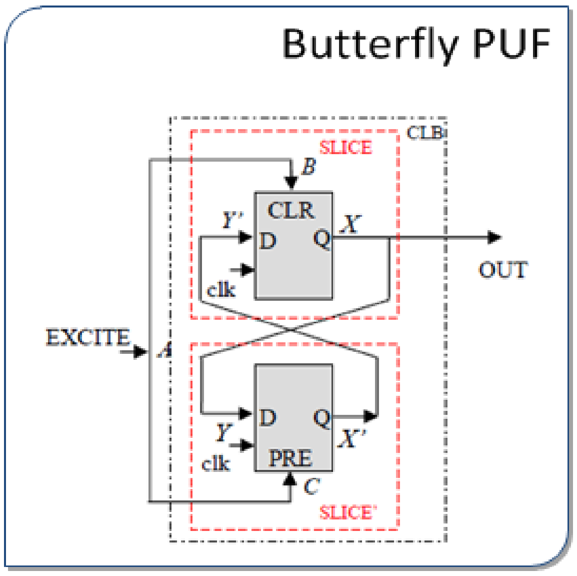
The output of these ROs is connected to the clock port of a counter implementation, which counts for every clock pulse. The variance is extracted by comparing the values of the counters after a certain time limit. If the propagation delay of one of these ROs is shorter than the other, it will oscillate faster, and as a result, following a certain time period, the value on its counter will be significantly more than that of its counterpart. Unlike with the Arbiter PUF, the propagation delay of the path between RO and its counter is not relevant to the resultant values, so that, an RO can be matched with many other ROs.

Similar to the Arbiter PUF Implementation, the routing between logical components cannot be directly controlled in the FPGA implementation flow. Therefore, RO PUF implementations are hard to design and verify. These two PUF constructions are capable of generating massive amounts of data using very few resources, but conversely, the realization of their design within the FPGA platforms is difficult to achieve.

### 2.4.1.3 Butterfly PUF

Similar to the other silicon based PUF constructions, the Butterfly PUF uses the physical variances of logical components caused by the manufacturing processes. However, unlike Arbiter and RO PUFs, the Butterfly PUF (Kumar, et. al. 2008) is not based upon a delay measurement; instead, it uses the mismatch of two cross-coupled latches.

**Figure 2.7: Butterfly PUF**

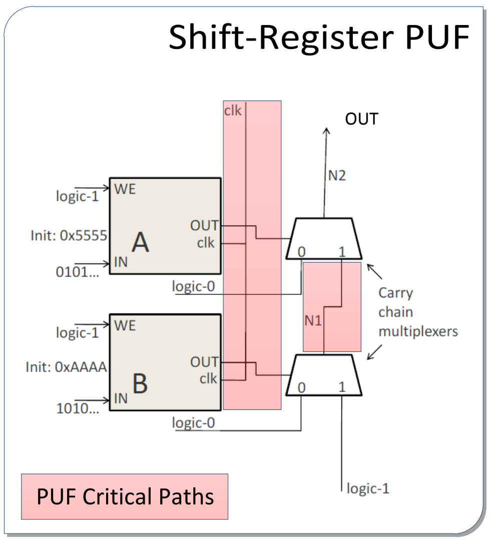


As illustrated in Figure 2.7 this circuit has two logically stable stages, but, with the rising of the excite signal, the top latch is cleared to digital low, and the bottom latch is set to digital high so the system becomes unstable. It will then begin to oscillate for a certain amount of time before stabilizing onto one of the digital stages. The latch that holds its current value for the longest is the one that will determine the stable value. This design is suitable for FPGA applications since the routing resources used for one PUF instance is extremely low. However, the critical paths that are used for extracting the variance, such as the excite signal or the data signals between latches are implemented using programmable interconnections.

#### 2.4.1.4 Shift-Register PUF for FPGA

In 2010, a new type of PUF (Anderson 2010) construction designed specifically for FPGA implementations is published. Just like Arbiter and RO PUFs, this PUF construction is based upon the delay measurement and comparison of logical paths. We call this PUF; the Shift-Register PUF since its creator, J. Anderson, did not give it a name. With this PUF design, the delay path is created using a shift-register and a multiplexer. The shift registers are constantly filled with pulses simultaneously, but with inverse signals. The output of the bottom pair is connected to the "1" input of the top MUX and logical "0" is connected to the "0" port. The bottom MUX has logical "0" for port "0" and logical "1" for port "1" shown as in Figure 2.8.

**Figure 2.8: Shift-Register PUF**



The variance occurs between the transition stages when the delay propagation of the top path is shorter than that of the bottom path. A pulse is activated for a short period of time on the N2 output when the top shift-register changes its output to logical ‘1’ while the bottom shift-registers output transitions to logical ‘0’. This PUF implementation is optimized for FPGA applications since it uses fixed routed paths (inter and intra slice paths in the CLBs.) for timing critical paths. The lengths of which are certain to be identical in every instance.

### 2.4.2 Silicon Based PUFs Comparison

In this section, we have evaluated these PUF constructions specifically from the aspect of our FPGA solution. There are several research works relating to PUF performance evaluation such as, this (Maiti, Gunreddy and Schaumont, 2012) and (Morozov, Maiti and Schaumont 2010, pp.382-387). In these works, several security properties of different PUF Constructions are evaluated such as, randomness, steadiness, correctness, uniqueness, and reliability. However, we have aimed the focus of our evaluation of these PUF designs from a different and less acknowledged perspective; the actual FPGA realization. Most of the available PUF constructions can generate practical authentic data for security applications, but only a very few of them can be easily implemented on an FPGA platform. We have selected four properties that are imperative for FPGA implementation.

We have evaluated the availability of these PUF constructions concerning FPGA implementation, by comparing the logic and wiring resources, (including their constructions), to our Spartan-6 CLBs and routing components. Whilst all of these constructions can be implemented on our platform, Arbiter PUF has a similar architecture to the Carry Chain Multiplexers in the FPGA CLB, but one of every Multiplexers output is always routed from outside the CLB so that the resemblance of two parallel delay paths is not an option. However, this PUF can be implemented using LUTs as multiplexers, but the routing between every component is very hard to constraint so we have selected the Arbiter PUF as the least compatible construction for

FPGA implementation. Similar to Arbiter PUF, RO PUF experiences the same routing problems but the designer could use multiple instances of a pre-generated implementation of (Hard Macro (HM)) RO PUF in place of regeneration. These HMs can be generated by manually placing and routing logic resources using the FPGA Editor. However, the usage of these HMs in the FPGA design flow is problematic, so implementing a design with multiple HMs generates several errors during mapping and PAR stages, although it is possible for the designer to eliminate these errors with the use of placement limitations.

The Butterfly PUF has the same routing problems as the RO PUF and can only be implemented using HMs. Nevertheless, this PUF construction uses only two latches that are readily available in the FPGA CLBs, and, as a result, this construction has a fewer resource usage and lesser the routing complexity than the other constructions. Unlike the other PUF constructions, the Shift-Register PUF is specifically designed for FPGA implementation. In this construction, the paths that should be identical between two compared pairs are routed using fixed carry-chain paths. Therefore, the routing between pairs is identical in every instance. The designer only uses placement constraints for its implementation and manual routing is not necessary.

**Table 2.1: Silicon based PUFs**

| | **Arbiter PUF** | **Ring Oscillator PUF** | **Butterfly PUF** | **Shift-Register PUF** |
|---|---|---|---|---|
| **PAR Complexity** | High | Medium | High | Low |
| **Manual Routing /Hard Macro requirement** | Yes (Not feasible) | Yes | Yes | No |
| **FPGA Implementation Convenience** | Low | Medium | Medium | High |
| **FPGA Hardware Cost** | Low | High | Low | Medium |

The comparison in the Table 2.1 shows the grades we have given to these constructions following in-depth evaluation and analysis. Our evaluation is based upon our Spartan-6 Platform and the recent Xilinx Development tools.

# 3. OUR DESIGN METHODOLOGY

In this section, we propose the design methodology behind our secure DPSR for large differences (DPSR-LD) flow and the various components that are developed for its realization. DPSR-LD flow is offered to augment the tools provided with the recent free Xilinx ISE WebPACK release. In Flow Guideline, first, we have covered the system and the directory structures we use and then we described the important processes that are required in design entry, and physical implementation stages to create modular PR applications in our flow. We explained the implementation of ICAP+ module and PR design extraction script in Specific Components. Finally, in Bitstream Security, we have presented a detailed description of how obfuscation and PUF implementations are deployed within our DPSR-LD flow.

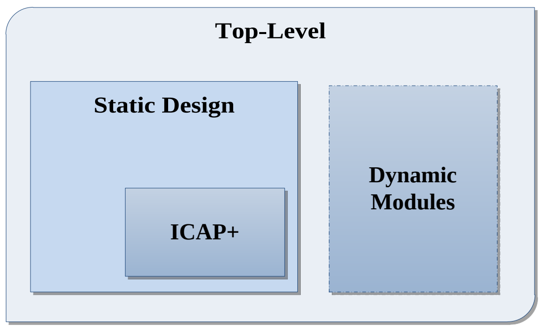
## 3.1 DPSR FLOW GUIDELINE

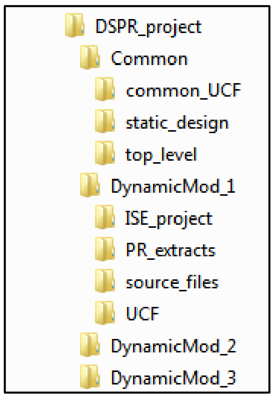
We have developed a design flow similar to other modular PR approaches for dealing with the problems occurs while using Difference based PR for large differences. The projects are developed in modular style and the physical layout is partitioned with similar approaches. However, the implementation and bitstream generation processes are customized for creating large differences using Difference based PR. We propose a system and a directory structure that are essential in DPSR-LD flow.

### 3.1.1 System and Directory Structure

The system structure that we use in our flow is similar to the one used in (Xilinx Inc. 2006) Xilinx EAPR applications. In these applications, the usage of a thin wrapper that contains both static and dynamic modules is recommended. The structure of this wrapper should be an overall top-level design where each functional module is instantiated as components of the wrapper. The common parts of the design (top-level and the static modules) are implemented without dynamic modules. The dynamic region is instantiated at the top-level as a black-box module and its physical site is restricted for PAR tool. The dynamic modules are implemented in separate directories. These projects share limited information with the top-level project; which are the location and the size of the dynamic region and location of its input output connections. In this flow, Xilinx recommends to keep these projects in separate directories. However, in our flow, we implement the dynamic modules within the rest of the system. This means that the common parts of the design are required in every configuration. Therefore, we use a directory system where components such as static design or top-level UCF file are shared while the rest of the project files including ISE projects and dynamic module source files are kept separately, see (Figure 3.1).

**Figure 3.1: DPSR-LD System and Directory Structure**





In our DPSR-LD flow, the reconfiguration process is controlled from inside the FPGA. We designed a hardware module for that purpose, which we call ICAP+. It is responsible for reconfiguration of the dynamic modules. It also blocks the communication of dynamic region during that reconfiguration period. This block provided with the top-level wrapper template is the baseline structure for the DPSR-LD projects.

### 3.1.2 Design Entry and Synthesis

In our design flow, the designer can design a PR circuit as an ordinary circuit following simple rules. First of all, the system architecture should be designed in a way that coexistence of interchangeable functions (modules) is not required in the same period, since it is not possible. The designer should keep logic definitions at top-level limited to IO connections, internal connections, and clock generation. This is because the components at this level do not belong to any part of the design; as a result, the designer cannot easily regulate their implementation. In contrast, the components under a sub module can be separated from the rest of the design since they all have the modules name as a prefix to theirs. Furthermore, the static part should be able to access dynamic modules using a common interface. We use a wrapper module to convert the connections of dynamic modules to a common interface for pairing dynamic modules with the instance of dynamic region at the top-level.

Bus macros were required in the module-level PR schemes such as EAPR flow (Xilinx Inc. 2006). Bus macros provide a means of locking the routing between PR modules and the static design, making the PR modules pin-compatible with the static design. In the EAPR flow, all connections between the dynamic parts of the design and static design must pass through a bus macro with the exception of global signals, BUFG global clocks, GND and VCC, which are handled automatically by the tools in a way that is transparent to the user. Bus macros are provided with the EAPR software tools in the form of pre-placed, pre-routed hard macros. However, bus macro instantiations are no longer required in the latest Xilinx PR tool (Xilinx Inc. 2011c). Instead of hard bus macros, we place a simple block that serves like a bridge between the static and dynamic areas. These blocks are simple FFs in FPGA slices with their enable controlled by static modules. This bridge can both capture the output of a PR module and drive its inputs as well. FPGA slices have multiple FFs, and hence, a single FPGA slice can connect many wires between static and dynamic regions depending on an FPGA family's slice structure. These blocks are generated through simple RTL coding.

In our flow, the synthesis of the dynamic modules is done together with the static design. Using standard settings, Xilinx XST synthesizes and optimizes all of the system as a single circuit. It removes and/or renames the definitions of modules, signal and component names from the synthesized file. However, the module definitions are necessary for extraction of dynamic modules after physical implementation stages. We have used “Keep Hierarchy” synthesis option for disabling optimization between module boundaries. Setting this option helped us to preserve logical description of dynamic modules apart from static design. Additionally, “Keep” attribute is set for the signals in the common bridge interface. This attribute is used for preventing these signals to be absorbed by either static or dynamic regions.

The size of the dynamic region is determined with the resource requirements of dynamic modules. For further processes in the flow, we collect the number of resources used for the dynamic module implementations from XST tools synthesis reports (Table 3.1). For instance, the dynamic region of this circuit (Figure 4.1) should contain at least 64 SLICEMs, 100 LUTs, and 64 FFs for implementation of any of these modules.

**Table 3.1: An example of resource utilization report.**

| | PUF_farm_1 | PUF_farm_2 | ASCII_Parser |
|---|---|---|---|
| Number of FFs | 40 | 40 | 64 |
| Number of LUTs | 92 | 92 | 100 |
| PUF Specific | SLICEMs 64 | SLICEMs 64 | - |

Apart from standard logic components, we have collected the number of the SLICEMs used for our PUF_farm implementations. The Shift-Register PUF design is based on Shift-Registers. The only type of slice that has built-in Shift-Register is SLICEMs. This information is necessary for choosing a suitable region in the physical design.

### 3.1.3 Physical Design

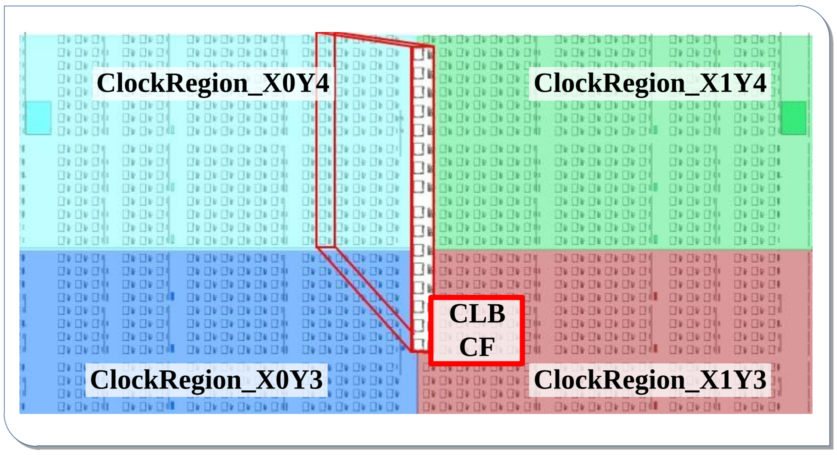
In FPGA development platforms such as Xilinx ISE, most of the physical flow stages are automated. The designer is only concerned about IO placement and timing performance because these procedures are very complex and time-consuming for humans. However, manual placement of specific logic components may be required in some high-speed applications. In the same way, directed physical implementation is necessary in PR applications. We have analyzed this process in the following sections.

#### 3.1.3.1 Dynamic region identification

The first stage of the physical design for modular PR applications is to floor plan the design on the FPGA layout. In the Xilinx PR flow, the dynamic partial region is defined by AREA_GROUP constraints. The designer can define this AREA_GROUP as a rectangular range of slices that reside within its region. Any modules implementation, either static or dynamic can be constrained to a predetermined area within this setting. The PAR tools only use the logic resources that are available in the determined region. That is why the designer has to select a dynamic region, which contains sufficient resources for the target module. These constraints can be generated by floor planning with Xilinx PlanAhead Tool, or the designer can manually add them to the User Constraints File (UCF) file. In Xilinx PR flow, this area group's mode can be set as reconfigurable so that tools treat this geometric region as a reconfigurable area. In our DPSR-LD flow, this constraint is used for the definition of the dynamic region that is used for dynamic module extraction.

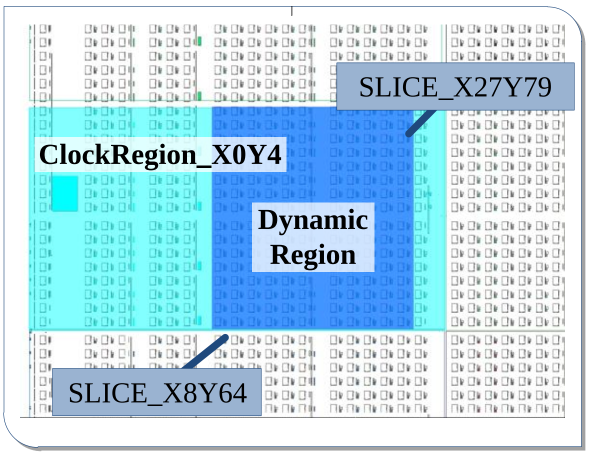
For instance, we have used the resource requirement information of the dynamic modules of example obfuscated-GPIO design (Figure 4.1). Although the ASCII_parser module is the most resource rich application, the resource requirements of PUF_farm modules defined the size of the dynamic region. The PUF construction we have used in our application is created with shift-registers, which are only available in SLICEMs. One PUF construction requires two vertically neighboring SLICEMs and both PUF farms have 32 of them. Since only 25 percent of the available slices are SLICEMs, the dynamic region must occupy at least 256 slices.

**Figure 3.2: An example of Spartan-6 clock regions and CLB CFs**



The size of this region also determines the size of the partial bitstreams. These bitstreams contain Configuration Frames (CF) of their dynamic module. The CF is the minimum addressable unit of the FPGA configuration memory. Spartan-6 has three types of configuration frames; one type for IO definitions, another type for BRAM, PLL, DCM, configurations, and another type for CLB definitions. Mostly, the dynamic modules utilize only CLBs. A CLB CF is a group of 16 vertically aligned CLBs in Spartan-6. On its layout CLB CFs are vertically separated from each other with clock regions.

**Figure 3.3: Dynamic region layout**



In a CLB CF column, even if one slice configuration is changed, the whole CF must be reconfigured. That is why the location and the size of the dynamic region should be selected so that minimum amount of CF is used for dynamic region. For instance, if we define a dynamic region that has 32 CLBs, its partial bitstream can be generated with only 2 CLB CFs. On the other hand, if it can also be generated with 32 CLB CFs by selecting a row of CLBs in the FPGA layout thus using more CLB CFs can result larger partial bitstreams. Therefore, we have selected a dynamic region with minimum CFs by including every CLB of the covered CFs. There are 320 CLBs in the region that is 5120 FFs and 2560 LUTs theoretically (some of these resources cannot be used because of PAR limitations.). This region (Figure 3.3) contains 5 out of 20 CLB CFs that contain SLICEMs. As a result, 40 PUF constructions can be generated on this area per module, which is sufficient for PUF farm implementations. These resources are more than satisfactory for ASCII_Parser implementation.

#### 3.1.3.2 Placement and Routing constraints for partitioning

In order to use Xilinx standard flow placer tool for PR applications, usage of placement constraints is necessary. The dynamic region’s AREA_GROUP should be set with “PLACE=CLOSED” constraint, which prevents the placement of the rest of the design to this region. Within the use of this constraint, static components are placed outside the dynamic region although their nets can be routed through it. For example, a net that has its source and destination in static region can be routed through the locked dynamic region. The PAR tool doesn’t have a constraint for prohibiting this type of operation. Since the configuration of the dynamic region is changed during reconfiguration period, the crossover net can be disconnected. In addition, the nets in the dynamic region can be routed through the static region. The nets of the dynamic modules could corrupt the static design and since more CFs are modified; partial bitstreams generated from these modules will be larger. Routing of these nets could be controlled through complex techniques like “Directed Routing” in which the designer manually generates the routing paths of these signals. Using fixed routing paths is not suitable for our flow since the implemented design changes with different dynamic modules. For our application, we used “CONFIG PROHIBIT” constraint to limit the possibility of this kind of routing problems. This constraint prohibits placement of the design components to the specified FPGA resource. We have created a no place region around the dynamic region in which placement of any logic component is restricted. The routing of the nets can cross this region but the size of it is sufficient to prevent nets of both regions cross over each other.

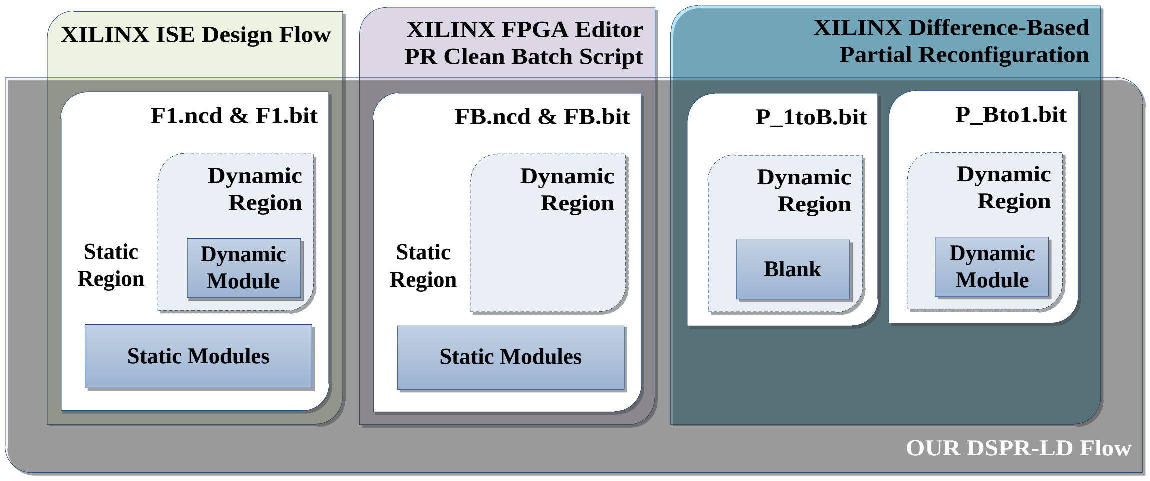
The crossing point of the signals to and from the dynamic region should be placed to a predetermined location. In this way, the designer ensures that the top-level connections are identical for every dynamic module. We fix the position of these modules by LOC and BEL constraints in the UCF. We guarantee the placement of the FFs to the same slice with LOC and arranging them in the same order with BEL constraint.

#### 3.1.3.3 Partial bitstream generation

The final stage of our flow is to extract partial bitstreams of dynamic modules. Although our flow makes use of difference based PR, it works for large differences as well as modular PR. After constraints file creation, the entire design is implemented for every dynamic module. These designs have the same logic definition on the static region but have different modules in their dynamic region.

Static parts, which are processed alongside with different dynamic modules, may be implemented differently. Even two identical designs become different after implementation stages since the placement or the routing of the components varies for every instance. Therefore, the difference of two entire designs with the same static part but with different dynamic modules may contain static region components and nets. In addition, the dynamic modules cannot be absolutely extracted using this approach since the resulting bitstream is the transition from one design to another. Thus, the reconfiguration sequence of dynamic modules is limited to the followed pattern during the extraction of partial bitstreams.

**Figure 3.4: Partial bitstream extraction in our DPSR-LD flow**



Our method addresses these problems with a smart approach. The designs are processed separately so any configuration sequence can be created with this approach. The partial bitstreams of dynamic modules are extracted on the FPGA Editor within help of a batch script. At first, a copy of the first complete design configuration's NCD (for instance F1.ncd) is opened and all the components that belong to dynamic region are removed in the FPGA Editor by the script (Figure 3.4). After this process, the dynamic region becomes truly blank. The modified design is saved in another NCD file and another full bitstream is generated from this design (for instance FB.ncd and FB.bit). Afterwards, the partial bitstream that blanks out the dynamic region (for instance P_1toB.bit) is generated by BitGen with the –r switch with FB.ncd and F1.bit. Furthermore, the partial bitstream that configures the dynamic regions from blank to a complete design configuration (for instance P_Bto1.bit) is generated by BitGen again with the –r switch with F1.ncd and FB.bit.

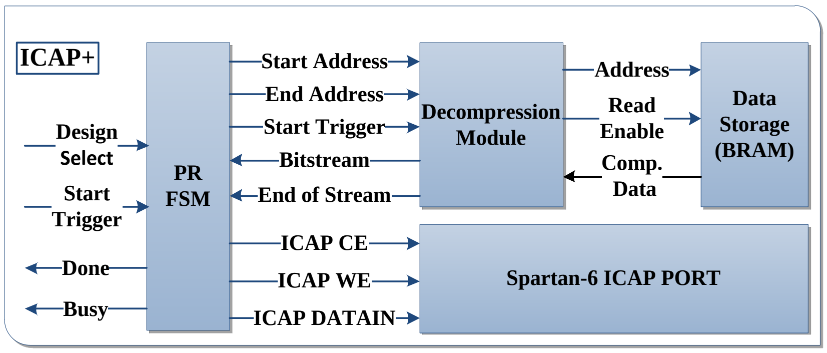
### 3.1.4 Specific Components

In some stages of our DPSR-LD design flow, we have needed extra tools. As a result, we have designed a bitstream compression software, a hardware PR controller module (ICAP+), a dynamic module extraction script, and a partial bitstream generation script which automates the whole PR process. In this section, we are going to describe how we have engineered them. First, we are going to describe how ICAP+ and its compression software pair work, and then describe how dynamic modules are extracted from the entire design.

#### *3.1.4.1 ICAP+*

As stated before, we designed the ICAP+ module to control the PR process from inside the FPGA. This module (Figure 3.5) consists of a BRAM module to store the compressed partial bitstreams, a decompression module to decompress the bitstreams, and a PR FSM module to manage the reconfiguration process. The ICAP+ is controlled with its top-level interface; the partial bitstream is selected with Design Select signal and the reconfiguration is started with the rising edge of the Start Trigger signal. It has a Busy signal to indicate the ongoing reconfiguration process and a Done signal to confirm the successful conclusion of a reconfiguration attempt. Apart from control signals, the partial bitstreams are provided to the storage submodule in a VHDL package as a data array as well as their memory addresses are provided to the PR FSM.

**Figure 3.5: ICAP+ Block Diagram**

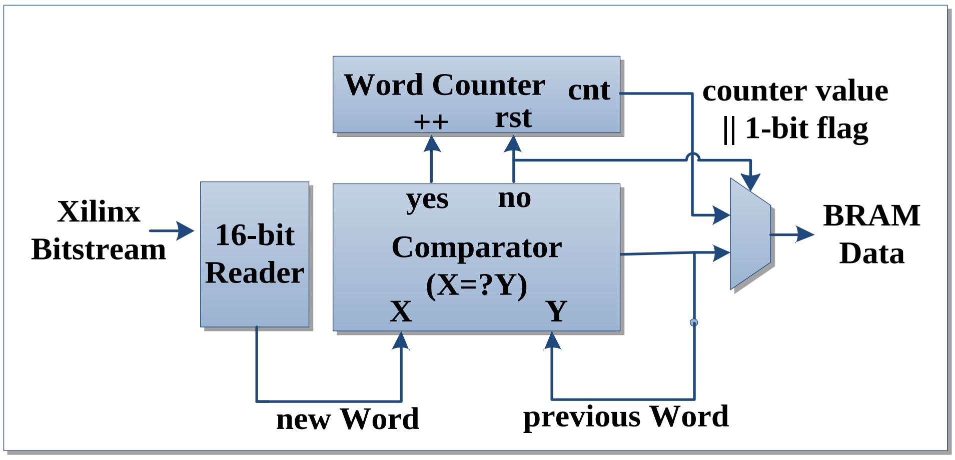


Following the assertion of the top-level Start Trigger signal, PR FSM sends this Start Trigger to the decompression module along with the selected partial bitstreams start and end addresses. At this state, the top-level Busy signal is asserted. Afterwards, the decompression module starts to stream the decompressed partial bitstream through Bitstream port. The PR FSM captures and writes this stream to the FPGA configuration thru ICAP. The ICAP has a simple protocol similar to other SRAM interfaces (i.e.

BRAM). To write a stream, a data word is provided to the ICAP DATAIN for every rising edge of the clock while the active-low ICAP CE and the ICAP WE signals are asserted to digital low states. The configuration words of the partial bitstreams must be delivered to ICAP in a byte swapped format (Xilinx Inc. 2011b). Once the decompression module reaches the provided End Address, it generates a pulse on its End of Stream output. With the assertion of this signal, first, the ICAP is closed, and then a pulse is generated for the top-level Done signal, after that the BUSY signal is de-asserted, and finally the PR FSM is reset to its initial state.

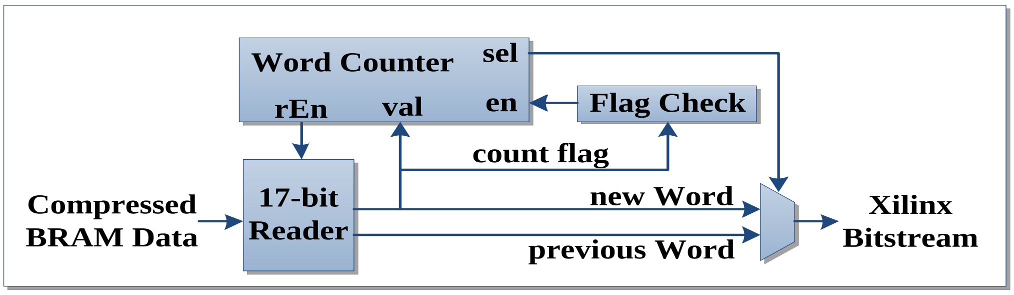
For our secure DSPR-LD application, the partial bitstreams are stored inside the FPGA in a data storage submodule under ICAP+. Therefore, the resource utilization overhead of ICAP+ module is directly proportional with the total size of the partial bitstreams. To reduce this overhead, the bitstream files are stored in a compressed format and are decompressed on the fly during reconfiguration. Since we mostly target the latest low-cost Xilinx Spartan-6 FPGAs, we tailored compression to it. Its configuration word size is 16 bits and its bitstreams have redundancy in the form of many NULL (0000) or NOOP (FFFF) signals (usually in a repeated fashion). This redundancy could be exploited through RLC (Pan, et.al. 2004 pp. 766-773). For that purpose, we have designed a software tool in C++, which uses a simple RLC coding technique to compress bitstreams. Our bitstream compressor (Figure 3.6) always compares the incoming word with the previous one. If the two consecutive configuration words are not identical, the word is passed to the output with its flag set to logic-1 on its MSB. Conversely, if the words are identical, the output of the software is suspended for that word while the word counter is incremented. This continues for all subsequent words until an irregular word appears on the input or the software reaches the end of the bitstream. Eventually, to represent all of the repeated words, the counter value of is passed to the output with a flag set to logic-0.

**Figure 3.6: Compression Software Functional Diagram**



The decompression module on the other hand, reverts this process and regenerates decompressed bitstreams. Our compressed bitstream has 17-bit words. The first bit is a flag. If the flag is 1, the next 16 bits is a value passed as is from the input bitstream. If the flag is 0, then the next 16 bits is the number of times the previous 16-bit word is repeated. The block diagram of the decompression module is shown in Figure 3.7.

**Figure 3.7: ICAP+ Decompression Submodule**



For example, the following sequence of 14 16-bit Xilinx configuration words (in hex) in words “ffff ffff ffff ffff ffff ffff ffff ffff aa99 5566 30a1 0007 2000 31a1” compresses to a sequence of 7 17-bit words “0_ffff 1_0007 0_aa99 0_5566 0_30a1 0_2000 0_31a1”. In Table 3.2, it is shown that the partial bitstream file of the PUF_farm1 module is reduced to 42 percent of its original size using this technique. We have observed that depending on the bitstream content, we usually get between 30 to 50 percent reduction (compared to the original size) using this simple procedure.

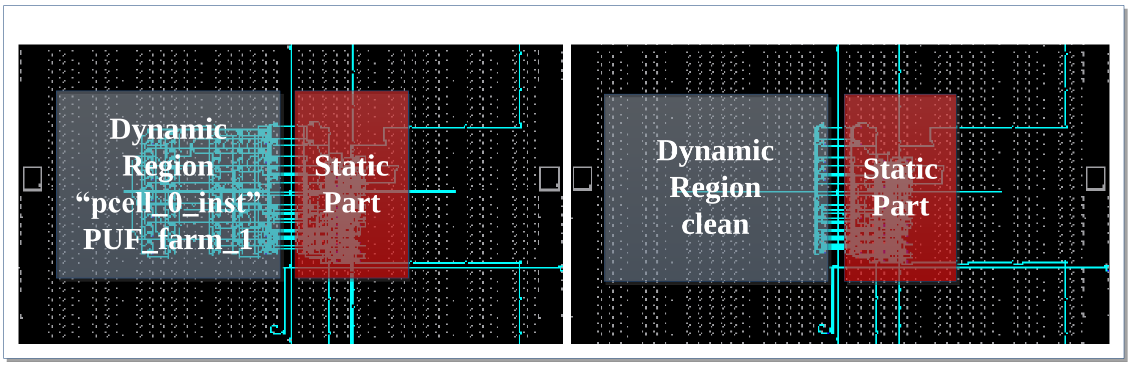
**Table 3.2: Compression performance**

| **Original Bitfile** | **Uncompressed BRAM Data** | **Compressed BRAM Data** |
|---|---|---|
| 26350 * 8 bits | 13113*16 bits | 5197*17 bits |
| 210,800 bits | 209,808 bits | 88,349 bits |
| 100 percent | 99 percent | 42 percent |

### *3.1.4.2 Partial Design Extraction Script*

In our DPSR-LD flow, the dynamic modules are extracted from the entire design after implementation. The logic components and the interconnections under the dynamic module instance should be removed from the rest of the design to generate a bitstream separately for static part and dynamic modules. This could be achieved by editing the NCD library of the design using a dedicated tool, which in our case FPGA Editor. This tool has the ability to record your actions while you are editing a design in the GUI. This is a very powerful feature for creating automated "in flow" procedures when it is not possible to use high-level tools (i.e. PlanAhead) for design partitioning.

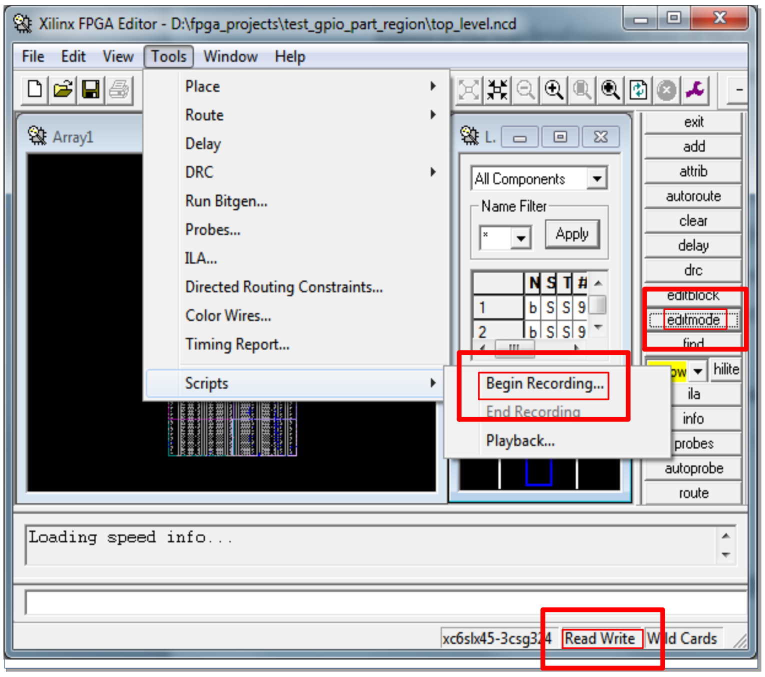
**Figure 3.8: Dynamic Module Extraction**



For instance, we have implemented a circuit design, which consist of a static top-level and two dynamic PUF-Farm submodules that has 32 PUF cells inside with regards to our DPSR-LD flow guideline. Therefore, it is possible to extract the components of the submodules from the entire design since the modular hierarchy is protected and the names of the top-level connections of the dynamic region are preserved. Moreover, the logic components are placed to a predetermined dynamic region with the help of placement constraints.

After the implementation, we have opened one of the projects NCD library in the FPGA Editor. First, the recording tool is activated from the “Scripts” tab in the “Tools” dropdown menu. The next step is to enable the read/write privileges for this file, which is done by clicking the “editmode” button on the right column shown in Figure 3.9.

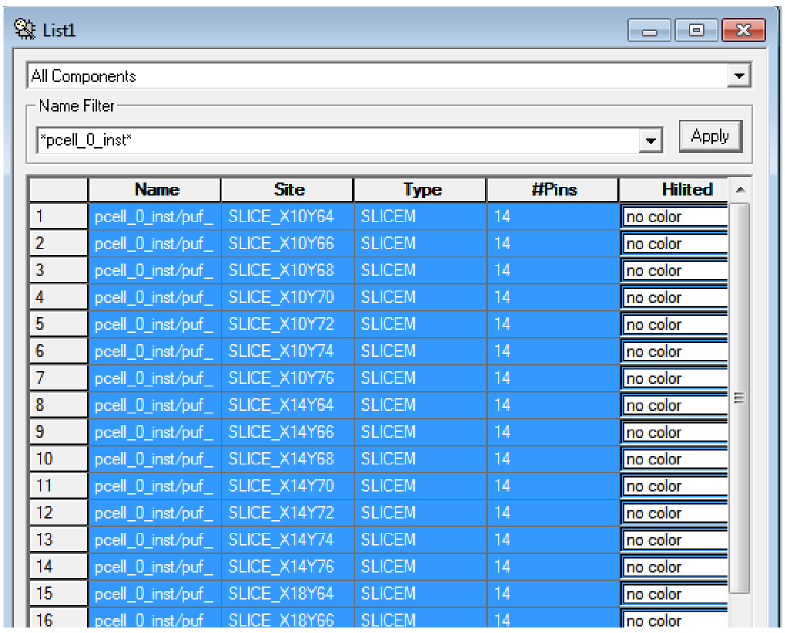
**Figure 3.9: FPGA Editor Script Recording**



Loading speed info...

Next, we searched for the components and interconnections which has "pcell_0_inst/" string in its name because all of the components under the dynamic module have this prefix (Figure 3.10). Then we have removed these components by simply pressing Delete button on keyboard. FPGA Editor keeps the selected items in memory. This helps users to edit multiple items with fewer commands. Deleting components and/or nets result DRC errors since there are several input output pins left unconnected after the procedure, as expected. In order to save the modifications on the NCD file we have disabled the DRC tool, because FPGA Editor prevents users to save files with DRC errors. Finally, we have stopped the script recording to make sure no other processes are recorded into the script file.

**Figure 3.10: FPGA Editor smart selection of logic components**



List1

All Components

Name Filter: *pcell_0_inst*

Apply

| | Name | Site | Type | #Pins | Hilited |
|---|---|---|---|---|---|
| 1 | pcell_0_inst/puf_ | SLICE_X10Y64 | SLICEM | 14 | no color |
| 2 | pcell_0_inst/puf_ | SLICE_X10Y66 | SLICEM | 14 | no color |
| 3 | pcell_0_inst/puf_ | SLICE_X10Y68 | SLICEM | 14 | no color |
| 4 | pcell_0_inst/puf_ | SLICE_X10Y70 | SLICEM | 14 | no color |
| 5 | pcell_0_inst/puf_ | SLICE_X10Y72 | SLICEM | 14 | no color |
| 6 | pcell_0_inst/puf_ | SLICE_X10Y74 | SLICEM | 14 | no color |
| 7 | pcell_0_inst/puf_ | SLICE_X10Y76 | SLICEM | 14 | no color |
| 8 | pcell_0_inst/puf_ | SLICE_X14Y64 | SLICEM | 14 | no color |
| 9 | pcell_0_inst/puf_ | SLICE_X14Y66 | SLICEM | 14 | no color |
| 10 | pcell_0_inst/puf_ | SLICE_X14Y68 | SLICEM | 14 | no color |
| 11 | pcell_0_inst/puf_ | SLICE_X14Y70 | SLICEM | 14 | no color |
| 12 | pcell_0_inst/puf_ | SLICE_X14Y72 | SLICEM | 14 | no color |
| 13 | pcell_0_inst/puf_ | SLICE_X14Y74 | SLICEM | 14 | no color |
| 14 | pcell_0_inst/puf_ | SLICE_X14Y76 | SLICEM | 14 | no color |
| 15 | pcell_0_inst/puf_ | SLICE_X18Y64 | SLICEM | 14 | no color |
| 16 | pcell_0_inst/puf | SLICE_X18Y66 | SLICEM | 14 | no color |

The resulting script is composed of component selection (select -k comp) and removal (delete) commands. The tool also recorded the commands of enabling the edit-mode privileges (setattr main edit-mode Read-Write), disabling the DRC procedure (setattr main auto_run_drc off), and file saving (save). The extracted script had more than 100 commands because the scripting tool has recorded the selection process of components individually. Therefore, the component specific names were recorded in script commands although we needed a script that has the flexibility to delete any submodule targeted. To address this problem we have rewritten a script, which targets the common information between individual dynamic submodules. The FPGA Editor accepts wild characters in commands, like the ones in the regular expression. For instance, the "*" character when added as a suffix to a word "FPGA" it matches every word starting with FPGA like "FPGA_PLL" or "FPGAbusy". The command (select -k comp ' pcell_0_inst/*') selects every single component under the dynamic submodule. In addition, this command will work for any extraction procedure of a dynamic module with the same instance name. With this fix, we have reduced the length of the script to 14 commands while adding flexibility to the work with any modules.

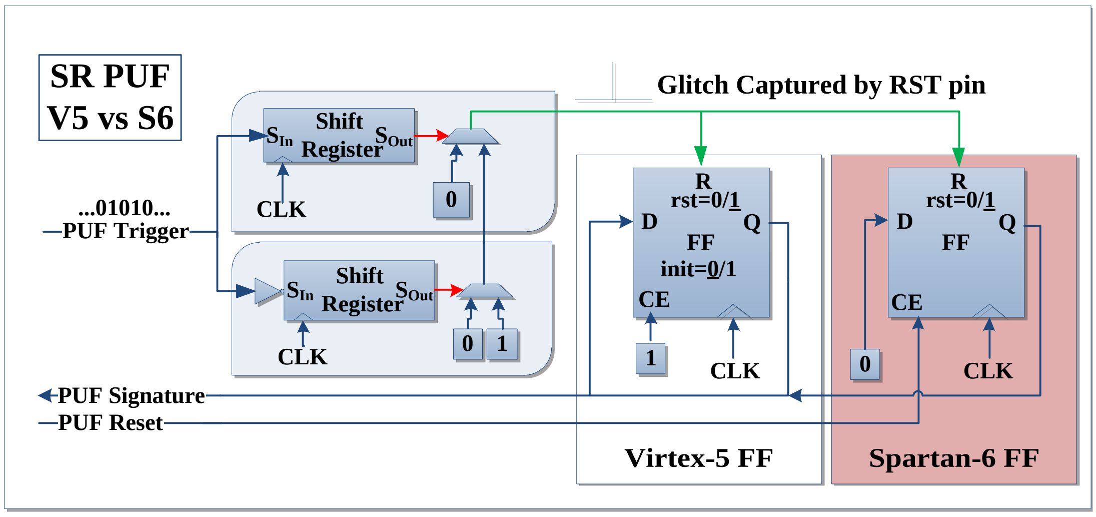
## 3.2 BITSTREAM SECURITY

We propose a FPGA bitstream security method by incorporating active obfuscation methods with PUF circuits. PUF key-based active obfuscation is the process of blending the chip's signature obtained from a PUF module with the IP's functionality. When several chips from the same family and type are configured with the same obfuscated bitstream, only the dedicated chip runs correctly, while the rest of the chips malfunction. We accomplish obfuscation at Register Transfer Level (RTL), where we do not only make the design hard to trace but also the design does not work as expected unless the correct PUF key is applied. The design is injected extra gates that turn into wires with the correct signature. A stable PUF requires quite a few bits of PUF and may not fit in the FPGA together with the design. To reduce resource overhead of this method, the desired circuit is designed by benefiting from our DSPR-LD technique. We use a Time-Division-Multiplexed-PUF (TDM-PUF) (Gören et al. 2010) that divides a single and long PUF into smaller PUFs that run at different time segments. These PUFs are placed in a dynamic region, which is then reclaimed for the actual (protected) design. Regarding obfuscation overhead, even doubling the states would not have a significant impact on the overall design area.

### 3.2.1 PUF Implementations

As stated before, we have chosen to utilize Shift-Register PUF construction in our application. This PUF construction (Figure 3.11) was originally designed for Xilinx Virtex-5 FPGAs. In fact, it exploits a special feature of Virtex-5 FFs. In its construction, every shift-register (SR) pair generates 1-bit signature indicating that either the top SR or the bottom SR has a shorter propagation delay. If the bottom SR is faster than the top SR, the output signal of the pair is held constant at logic-0. However, if the top SR is faster than the bottom one, a short positive spike (a glitch) will appear on the PUF cells output. This glitch can only be captured with an asynchronous logic component, which is the preset pin of a FF in this case. This FF is initialized to logic-0 and has its output Q fed back to its D input. If the glitch reaches to the preset port, the FFs output becomes logic-1 otherwise it is kept at logic-0. The FFs of Virtex-5 can be configured to have different initial and (p)reset values. However, Spartan-6's FFs can only be initialized to their reset value. As a direct result, the original shift-register PUF construction cannot capture the signatures extracted from the shift-registers.

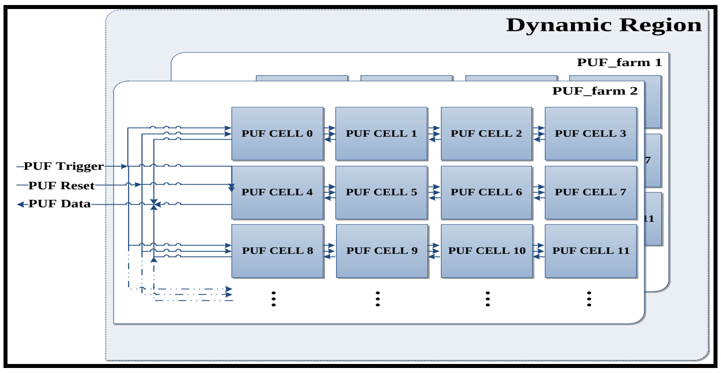
**Figure 3.11: Shift-Register PUF for Virtex-5 and Spartan-6**



We have adapted the design of Shift-Register PUF so that the glitch can be captured with Spartan-6 FFs. Similar to the original design, the glitch is connected to the preset port of FF. In Spartan-6 however, the FF is initialized with the preset value logic-1. We have designed a resetting logic by using CE and D input of the logic. In our application, the FFs belong to the PUF constructions works with their clock-enable disabled. In other words, these FFs are not going to be able to capture the signal values on the D-input. We use this CE signal to reset their values to logic-0 instead. After FPGAs initialization, the PUF Reset signal is set to logic-1 for a few clock cycles. Hence, the FFs of PUF constructions capture the logic-0 value at the D-input so that they are prepared to capture glitches from their preset pin. Unlike the original shift-register PUF, our design initializes with a certain value (logic-1), only after the PUF reset sequence it can generate the signature value.

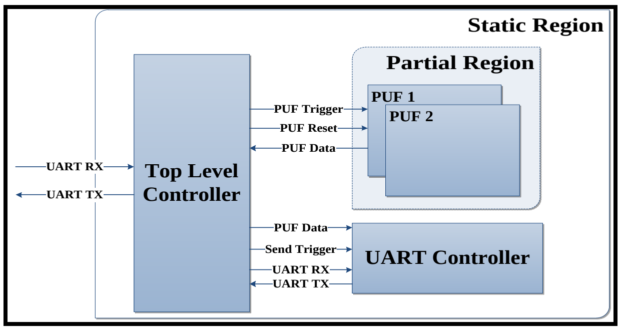
As far as the PUF construction is adapted to Spartan-6, we started using multiple instances of these PUF cells to generate authentication keys. These cells are instantiated under wrapper modules that we call PUF_farm. The instances are produced by using the simple but effective VHDL generate statement. On the other hand, this PUF construction utilizes two vertically neighboring SLICEMs for the implementation of the shift-register pairs. The placement constraints for these pairs must be provided to the PAR tool. We wrote a Tcl script that generates placement constraints for multiple PUF cells. This script can be configured to generate placement constraints for a predetermined area so this way we can place our PUF_farm (Figure 3.12) modules into dynamic regions to create dynamic PUF modules.

**Figure 3.12: PUF farm implementations**



A region on the FPGA layout can be reused to double the amount of extracted signatures. This could be achieved by rearranging the placement of PUF cells. Since any slice has two vertical neighbors, the PUF cells can be placed to pair the slices with the other neighbor. In our application, we used this method to extract 64-bit PUF signature from an area of 40 SLICEMs.

**Figure 3.13: PUF signature extraction circuit**



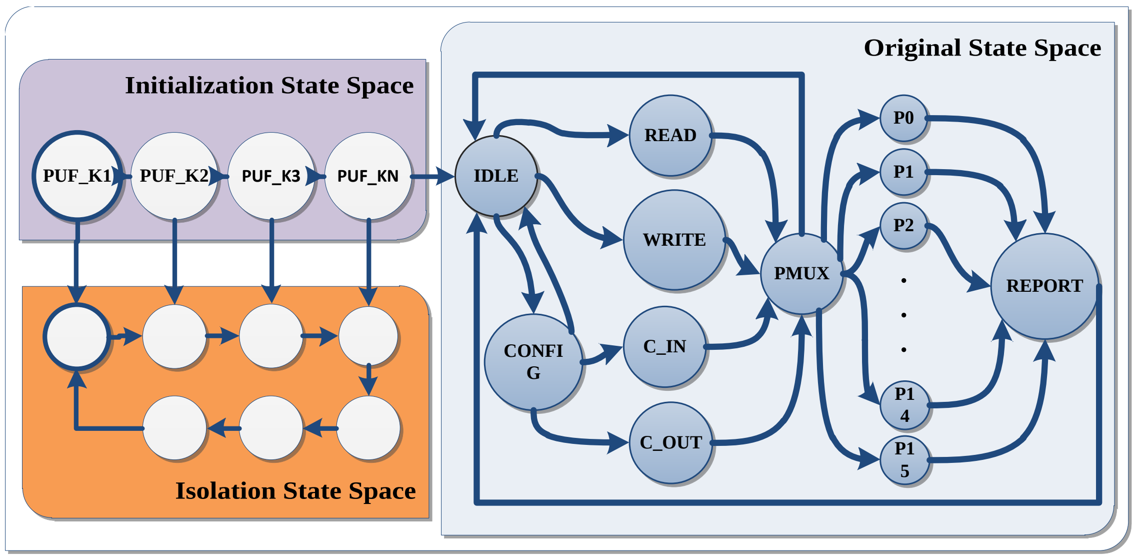
We have designed a circuit shown as in Figure 3.13 to extract the PUF signatures of the

FPGAs we use, which incorporates two PUF_farm modules in a dynamic region. This circuit is captures and sends the PUF signatures of PUF_farm modules to a PC through a UART controller.

### 3.2.2 PUF Key Based Obfuscation

PUF key-based active obfuscation is done by embedding a well-hidden finite state machine (FSM) or modifying the controller FSM of the circuit, which controls the functional modes based on application of the PUF response. Koushanfar (2012) showed how to devise an obfuscated FSM that is provably secure. The obfuscated FSM includes the original FSM, along with a number of added states and transitions. If the original FSM has $|S|$ states, it can be implemented using $\log|S|$ FFs. When we add a large number of new states$|S'|$, $|S| + |S'|$ states can be implemented by a linear growth in the number of FFs that is $\log(|S| + |S'|)$. 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 (shown as (PUF_K1, PUF_K2, …, PUF_KN) in Figure 3.14 required for traversal from the power-up state to the reset state of the original FSM. Figure 3.14 depicts an obfuscated FSM in which three state spaces are shown: (i) original state space, (ii) initialization state space, (iii) isolation state space. Isolation and initialization state spaces consist of new states. Depending on the PUF response, the power-up state can be in either isolation or initialization state space. In Figure 3.14, the power-up state is placed in the initialization state space. After application of a sequence of keys (that is also made up from PUF response), an initialization process sets the next state to the reset state of the original state space. During initialization, application of even a single wrong key sets the next state to a state in isolation state space.

**Figure 3.14: Obfuscated FSM**

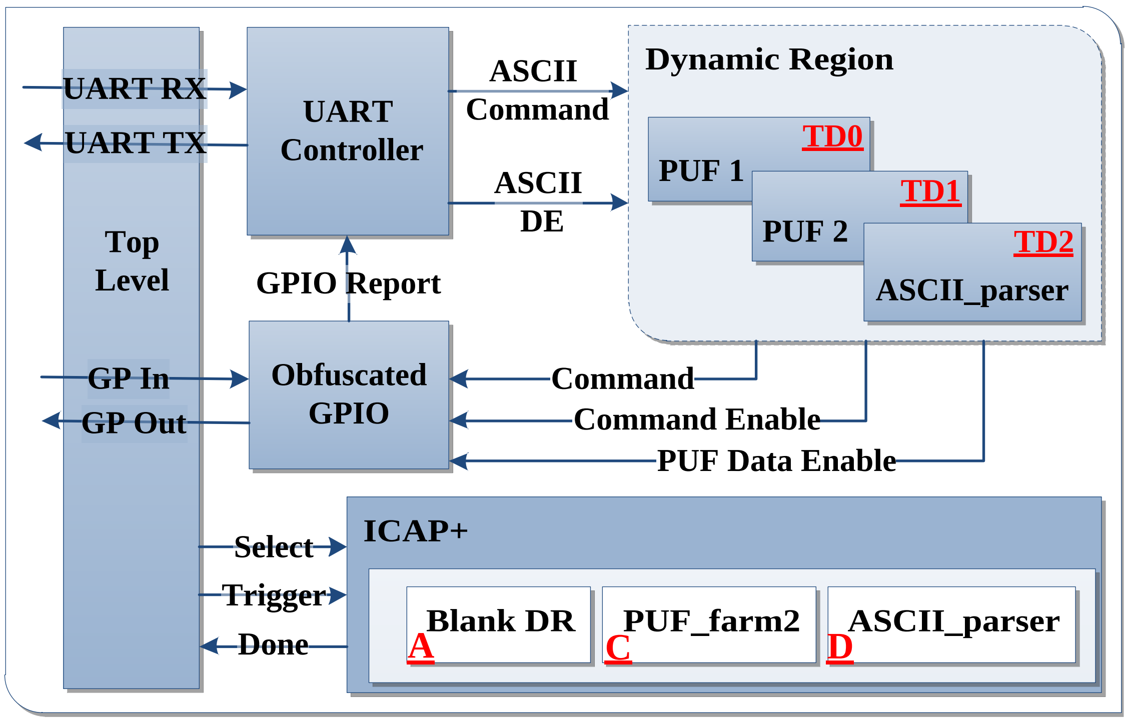


Provably secure obfuscation has an overhead in resource utilization and extensive experimental results were reported in (Koushanfar 2012) on the ISCAS sequential benchmark suite. Their results indicated large fluctuations among the circuits on area overhead, for large circuits the overhead on the average was given as 13 percent. In addition to area overhead due to obfuscation, PUFs also require considerable amount of area. In (Koushanfar 2012), the area overhead for the PUF modules were not included in the results and no solution to this problem was given in (Koushanfar 2012). However, we use DPSR to solve this problem where PUFs are placed in dynamic regions, which are then reclaimed for the actual (protected) design. To the best of our knowledge, our earlier paper (Gören et al. 2010) is the first paper in the literature which propose DPSR to remove the area overhead of PUF modules in PUF key-based active obfuscation using FPGA design flow.

# 4. RESULTS

We have applied our proposed bitstream protection technique on a General Purpose Input Output (GPIO) controller design shown as in Figure 4.1. The design has 16 GPIO ports (with configurable directions). Operations on the ports are controlled by an FSM. We communicate with this FSM through commands (read, write, toggle, configure direction) over an RS-232 interface.

**Figure 4.1: Example Obfuscated Design**



We have implemented two PUF farms to generate a 64-bit key (32-bit from each) as well as a PUF data extractor with UART controller. Original GPIO FSM has 23 states and at least 5 FFs are required to implement it (|S| =23 and K = log(|S|) FFs). We applied obfuscation on the HDL of the GPIO design using the 64-bit PUF key (PUF_K$_{63..0}$) based on the previously stated obfuscation method. In Table 4.1, we give the resource utilization overhead after obfuscation process. As seen in Table 4.1, we added 32 more FFs to the GPIO design and implemented an obfuscated FSM, where we encode the original, initialization, and isolation state spaces as 32-bits. Obfuscated FSM has 23 states in the original, 4 states in the initialization, and ($2^{32}$ – 27) in the isolation state spaces. We initialize the FSM using the first 32 bits of the 64-bit PUF key (PUF_K$_{63..32}$) by setting it to the power-up state. Note that the state encodings are tailored in such a way that the power-up state can only be either the dedicated start state of the initialization state space, or one of the isolation states. We use the rest of the PUF bits (PUF_K$_{31..0}$) applied as inputs in an initialization sequence of 4 cycles: (1) PUF_K$_{31..24}$, (2) PUF_K$_{23..16}$, (3) PUF_K$_{15..8}$, and (4) PUF_K$_{7..0}$. The resource utilization and obfuscation overhead are given in Table 4.1.

**Table 4.1: Obfuscation overhead in resource utilization.**

| **Logic Utilization** | **GPIO Original (#)** | **GPIO Obfuscated (#)** | **Spartan-6LX45 (Resource Overhead over Total) x100 ( percent)** |
|---|---|---|---|
| **FSM states** | 23 | (23 (Orig.) + 4 (Init.) + $2^{32}$–27 (Iso.)) | |
| **LUTs** | 127 | 210 | 0.30 |
| **FFs** | 54 | 86 | 0.05 |

After we complete the PUF key generation and obfuscation, next we prepare the bitstreams using the DPSR-LD flow. We had one static and one dynamic region allocated in the target Spartan-6. The static part consisted of the top-level controller, UART controller, and ICAP+. Three designs (1) PUF1, (2) PUF2, and (3) ASCII parsing submodule of GPIO are targeted for the dynamic region. We generated one complete and three partial configuration bitstreams using DPSR-LD flow. In Fig. 4.2, configuration B configures the whole FPGA with static module and PUF1, configuration A blanks dynamic region, configuration C reconfigures dynamic region with PUF2, and configuration D reconfigures it with the ASCII_parser design. Note that configurations A, C, D are partial, only B is full. They are compressed (occupy 13 percent of BRAMs) and stored in BRAM for ICAP+ to decompress them and reconfigure on-the-fly. The boot-up sequence is as follows (Figure 4.2):

1. B: 32-bit PUF1 signature is generated and sent to obfuscated GPIO module.
2. A: dynamic region is cleared.
3. C: 32-bit PUF2 signature is generated and sent to GPIO module, GPIO unlocked.

4. A: dynamic region is cleared.

5. D: ASCII parser submodule of GPIO design which is responsible for UART decoding is programmed.

**Figure 4.2: Boot-up sequence**

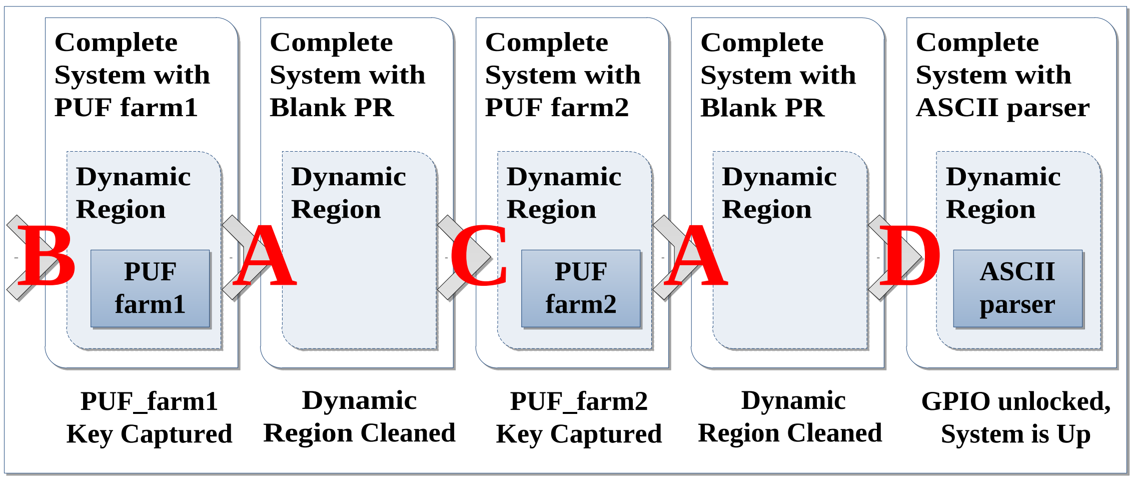


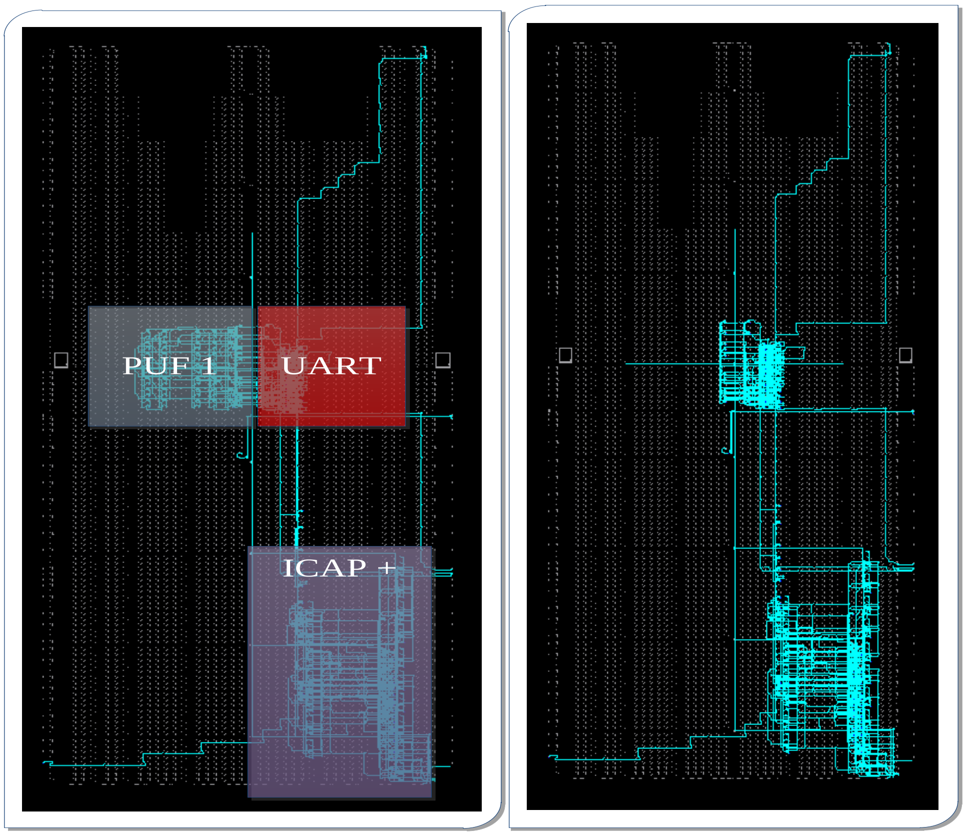
**Table 4.2: Measured Spartan-6 configuration times and bitstream file sizes.**

| Configuration | B | A | | C | | D | |
|---|---|---|---|---|---|---|---|
| Configuration Interface | 1-bit SPI @ 22 MHz | 16-bit ICAP @ 20MHz | | 16-bit ICAP @ 20MHz | | 16-bit ICAP @ 20MHz | |
| Bitstream File Size (KB) | 1450 | Original | Compressed | Original | Compressed | Original | Compressed |
| | | 27 | 1 | 24 | 10 | 27 | 11 |
| FFs (#) | 452 | 0 | | 40 | | 64 | |
| LUTs (#) | 605 | 0 | | 92 | | 100 | |
| Configuration Time (ms) | 700 | 0.7 | | 0.6 | | 0.7 | |

In Table 4.2, we present our measured configuration times and bitstream file sizes (both uncompressed (B, A, C, D) and compressed (A, C, and D)) for four configurations used in the boot-up sequence. Note that the configuration B is a full bitstream and we used 1-bit SPI at 22MHz to configure the chip, whereas we use 16-bit ICAP at 20MHz to configure the chip with partial configurations (A, C, and D). Partial bitstream configuration times are noticeably smaller than the full bitstream configuration times. This is mainly because of the differences in file sizes and configuration interfaces (1-bit

SPI versus 16-bit ICAP). In Table 4.2, the compressed partial bitstream file sizes are also given. A total of 22KB BRAM is required to store them which is about 13 percent of the total BRAM resources of Spartan-6 (XC6SLX45) and is totally reusable after the boot-up sequence. In Figure 4.3, two FPGA Editor screenshots are presented. The left image in Figure 4.3 shows configuration B including PUF1, UART, and ICAP+ modules, whereas the right image shows the same area including UART and ICAP+ modules after configuration A is loaded.

**Figure 4.3: FPGA Editor Screenshots**



# 5. CONCLUSIONS AND FUTURE WORK

Xilinx does not offer any partial bitstream encryption or PR support for low-cost FPGAs. In this study, we have proposed a partial bitstream protection technique for low-cost Xilinx FPGAs using PUFs, obfuscation, and our novel DPSR-LD flow. To the best of our knowledge, we developed the first partial bitstream protection technique, which combines PUF key-based active obfuscation and DPSR.

We demonstrated that DPSR-LD can be successfully applied to cases where there are significant design changes between successive configurations. We showed that DPSR-LD can be used as a DPSR solution for low-cost FPGAs with no Xilinx PR support.

We have implemented a bitstream compressor (software) and the decompressor (embedded in our ICAP+) in order to reduce PR time. The efficiency and practicality of the methods were demonstrated by proof-of-concept implementation of a GPIO controller design on Spartan-6 (XC6SLX45) on a Digilent Atlys Board.

While developing the methods and tools in this thesis, we have excluded several ideas and improvements due to time constraints and desire to keep the thesis focused. The effectiveness of the DSPR-LD flow can be improved by using alternative solutions to the routing problems. Third party PAR tools can be used to overcome the routing problems. The PUF cells can be utilized with multiple challenge signals, which could generate additional signature information. The obfuscation application can be automated with scripting. The tools in the design flow can be packed into a free, publicly accessible toolkit.

# REFERENCES

***Books***

Guajardo, J., Kumar, S.S., Schrijen, G.-J. and Tuyls, P., 2007. FPGA Intrinsic PUFs and Their Use for IP Protection, Cryptographic Hardware and Embedded Systems CHES 9th International Workshop, Vienna, Austria, September 10-13, 2007. *Proceedings*. *Lecture Notes in Computer Science Series* vol. 4727, pp. 63-80, Paillier, P., Verbauwhede, I., (Eds.), Heidelberg: Springer-Berlin.

Maiti, A., Gunreddy, V. and Schaumont, P., 2012. A Systematic Method to Evaluate and Compare the Performance of Physical Unclonable Functions**,** *Embedded System Design with FPGAs*, Athanas, P., Pnevmatikatos, D., Sklavos, N., (Eds.), Springer.

Morozov, S., Maiti, A. and Schaumont, P., 2010. An Analysis of Delay Based Puf Implementations on FPGA, *Reconfigurable Computing: Architectures*, *Tools and Applications*, *Lecture Notes in Computer Science*, vol. 5992, pp. 382-387, Sirisuk, P., Morgan, F., El-Ghazawi, T. and Amano, H., (Eds.), Heidelberg: Springer-Berlin.

Upegui, A. & Sanchez, E., 2005. Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs, *Evolvable Systems: From Biology to Hardware. Lecture Notes in Computer Science Series,* vol. 3637, pp. 56–65, Moreno, J. M., Madrenas, J., and Cosp, J., (Eds.), Heidelberg: Springer-Berlin.

***Periodicals***

Koch, D., Beckhoff, C. and Teich, J., 2009. Hardware Decompression Techniques for FPGA-Based Embedded Systems, *ACM Transactions on Reconfigurable Technology*, **2**(2).

Koushanfar, F., 2012. Provably Secure Active IC Metering Techniques For Piracy Avoidance And Digital Rights Management, *IEEE Trans. on Information Forensics and Security*, **7**(1), pp. 51-63.

Lim, D., Lee, J.W., Gassend, B., Suh, G.E., van Dijk, M. and Devadas, S., 2005. Extracting secret keys from integrated circuits. *IEEE Trans. on Very Large Scale Integration Systems*, **13**(10), pp. 1200-1205.

Pappu, R.S., Recht, B., Taylor, J., and Gershenfeld, N., 2002. Physical one-way functions, *Science*. **297** (5589), pp. 2026-2030.

## *Other Publications*

Anderson, J.F., 2010. A PUF design for secure FPGA-based embedded systems, *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC),* Proceedings, Taipei, Taiwan, pp. 1-6.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P. and Yang, K., 2001. On the (im)possibility of obfuscating programs, *Cryptology Conference on Advances in Cryptology (CRYPTO)*, 19-23 August 2001 Santa Barbara, CA, USA, London, UK: Springer-Verlag, pp.1-18.

Brzozowski, M. and Yarmolik, V. N., 2007. Obfuscation as Intellectual Rights Protection in VHDL Language, *6th International Conference on Computer Information Systems and Industrial Management Applications,* 28-30 June 2007 Elk, Poland, pp.337-340.

Chakraborty, R.S. and Bhunia, S., 2010. RTL hardware IP protection using key-based control and data flow obfuscation, *23rd International Conference on VLSI Design*, 3-7 January, Bangalore, India: pp.405-410.

Chakraborty, R.S. and Bhunia, S., 2008. Hardware Protection and Authentication Through C7Netlist Level Obfuscation, *ICCAD '08 Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, 10-13 November, San Jose, USA: IEEE Press, pp.674-677.

Dandalis, A. and Prasanna, V.K, 2001. Configuration Compression for FPGA-based Embedded Systems, *FPGA '01 Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, 11-13 February, 2011 Monterey, CA, NY, USA: ACM, pp.173-182.

Emmert, J., Stroud, C., Skaggs, B., and Abramovici, M., 2000. Dynamic fault tolerance in FPGAs via partial reconfiguration, *FCCM '00 Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, 17-19 April 2000 Napa Valley, CA, USA: IEEE Computer Society, pp.165-174.

Eto, E., 2007. Difference-Based Partial Reconfiguration [online], Xilinx Inc., http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf [accessed 5 March 2012].

Gassend, B., Clarke, D., van Dijk, M. and Devadas, S., 2002. Silicon Physical Random Functions, *CCS '02 Proceedings of the 9th ACM Conference on Computer and Communications Security*, 17-21 November 2002, Washington, USA: ACM, pp. 148-160.

Gassend, B., Clarke, D., van Dijk, M. and Devadas, S., 2002. Controlled Physical Random Functions, *ACSAC '02 Proceedings of the 18th Annual Computer Security Applications Conference*, 9-13 December 2002, Las Vegas, USA: IEEE Computer Society, pp. 149.

Goren, S., Ugurdag, H.F., Yildiz, A., Ozkurt, O., 2010. FPGA design security with time division multiplexed PUFs, *Proc. Int. Conf. on High Performance Computing and Simulation*, June 28 - July 2, 2010, Caen, France, pp. 608-614.

Goren, S., Ozkurt, O., Yildiz, A., Ugurdag, H.F., 2011. FPGA bitstream protection with PUFs, obfuscation, and multi-boot, *Proc. International Workshop on Reconfigurable Communication-centric Systems-on-Chip*.

Hussein, J. and Patel, R., 2008. MultiBoot with Virtex-5 FPGAs and Platform Flash XL [online], Xilinx Inc., http://www.xilinx.com/support/documentation/application_notes/xapp1100.pdf [accessed 8 March 2012].

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

Li, Z. and Hauck, S., 2001. Configuration Compression for Virtex FPGAs, *Proceedings of The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM '01*, 29 April - 2 May, 2001 Rohnert Park, CA,IEEE Conference Publications, pp. 147-159.

Liu, S., Pittman, R.N. and Forin, A., 2010. Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller, *FPGA '10 Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 21-23 February, 2010, Monterey, CA, USA: ACM, p. 292.

Lynn, B., Prabhakaran, M. and Sahai, A., 2004. Positive Results and Techniques for Obfuscation, *Proc. Int. Conf. on the Theory and Applications of Cryptographic Techniques*, 2-6 May, 2004, Interlaken, Switzerland,pp.20-39.

Note, J.-B. & Rannaud, E., 2008. From the Bitstream to the Netlist. *FPGA '08 Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, 24-26 February 2008, Monterey, CA, USA:ACM, p. 264.

Pan, J.H., Mitra, T. and Wong, W-F., 2004. Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs, *ICCAD '04 Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, 7-11 November, 2004, San Jose, CA, USA: IEEE Computer Society / ACM, pp. 766-773.

Pappu, R.S., (2001). Physical one-way functions. *PhD thesis*, Cambridge, MA: Massachusetts Institute of Technology.

Su, Y., Holleman, J. and Otis, B., 2007. A 1.6pJ/bit 96 percent Stable Chip-ID Generating Circuit using Process Variations, *IEEE International Solid-State Circuits Conference*, pp. 406-408.

Suh, G.E. and Devadas, S., 2007. Physical unclonable functions for device authentication and secret key generation. *Proc. Design Automation Conf.*,4-8 June 2007, San Diego, USA: ACM, pp. 9-14.

Xilinx Inc., Synthesis and Simulation Design Guide UG626 (v13.4), 2012, http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/sim.pdf [accessed 20 March 2012].

Xilinx Inc., Virtual Platforms For Zynq-7000 Extensible Processing Platform (EPP), 2011, http://www.xilinx.com/publications/prod_mktg/zynq7000/Zynq-7000-Virtual-Platforms.pdf [accessed 15 March 2012].

Xilinx Inc., Spartan-6 FPGA Configuration User Guide UG380 (v2.3), 2011, http://www.xilinx.com/support/documentation/user_guides/ug380.pdf [accessed 20 March 2012].

Xilinx Inc., Partial Reconfiguration Guide UG702 (v13.2), 2011, http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/ug702.pdf [accessed 20 March 2012].

Xilinx Inc., Spartan-6 FPGA Configurable Logic Block User Guide UG384 (v1.1), 2010, http://www.xilinx.com/support/documentation/user_guides/ug384.pdf [accessed 20 March 2012].

Xilinx Inc., Design Security for High Volume Applications, 2008, http://www.xilinx.com/publications/prod_mktg/DesignSecurity_ssht.pdf [accessed 1 March 2012].

Xilinx Inc., Early Access Partial Reconfiguration User Guide, 2006, http://www12.informatik.uni-erlangen.de/esmwiki/images/f/f3/Pr_flow.pdf [accessed 25 February 2012].

Xilinx Inc., Device DNA security, http://www.xilinx.com/products/design_resources/security/devicedna.htm. [accessed 20 March 2012].

## CURRICULUM VITAE

| | |
|---|---|
| **Full Name:** | Özgür ÖZKURT |
| **Address:** | Fulya Mahallesi, Ayşecik Sokak, No: 19 D: 9 Şişli, İstanbul |
| **Birth Place/Year:** | Kaman/1986 |
| **Foreign Language:** | English (advanced) |
| **Elementary School:** | Zafer İlköğretim Okulu, Aydın (1992-1997) |
| **High School:** | Adnan Menderes Anadolu Lisesi, Aydın (1997-2004) |
| **BS:** | Bahçesehir University (2004-2008) |
| **MS:** | Bahçeşehir University (2008-2012) |
| **Institute:** | The Graduate School of Natural and Applied Sciences |
| **Programme:** | Embedded Video Systems – Chip Track |
| **Work Experience:** | Design Engineer (October 2010 – ongoing)<br>Vestek Electronic Research & Development Corp.<br>Research Assistant (September 2008 – September 2010)<br>Computer Engineering Department, Bahçeşehir University |