



## Assignment of bachelor's thesis

|                                 |  |
|---------------------------------|--|
| <b>Title:</b>                   | Analysis of TPM Communication Using FPGA     |
| <b>Student:</b>                 | Martin Mandík                                |
| <b>Supervisor:</b>              | Ing. Martin Daňhel, Ph.D.                    |
| <b>Study program:</b>           | Informatics                                  |
| <b>Branch / specialization:</b> | Computer Security and Information technology |
| <b>Department:</b>              | Department of Computer Systems               |
| <b>Validity:</b>                | until the end of summer semester 2022/2023   |

### Instructions

Trusted Platform Module (TPM) is a HW solution that allows to increase the security level of a computing system. Analyze TPM behavior on LPC bus (Intel Low Pin Count). Design and implement an FPGA solution that will return values of LPC-TPM transactions over serial line. Create sufficient test cases of the implementation. Capture the LPC communication using the solution and interpret it according to the analysis.





**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

Bachelor's thesis

# **Analysis of TPM Communication Using FPGA**

*Martin Mandík*

Department of Computer Systems  
Supervisor: Ing. Martin Daňhel, Ph.D.

May 10, 2022



---

# Acknowledgements

I would first like to thank my supervisor, Ing. Martin Daňhel, Ph.D., for providing guidance during the process of writing the thesis. His expertise was truly helpful to me and his clear answers to my questions saved me a lot of time.

I would like to acknowledge Ing. Filip Štěpánek for bringing up a topic this interesting and encouraging me to work on it in my thesis.

I would also like to thank my family that has supported me throughout the time of my studies so far and provided me with a comfortable environment for work.



---

## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No.121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 10, 2022

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2022 Martin Mandík. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### **Citation of this thesis**

Mandík, Martin. *Analysis of TPM Communication Using FPGA*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.



---

# Abstract

The thesis focuses on the communication of the TPM security chip on the LPC bus. The aim is also to develop an FPGA design to capture this communication. In the theoretical part of the thesis, the structure of the TPM chip is described, and its most notable use cases are presented, along with basic information about the LPC bus. In the practical part, an FPGA design which filters only TPM related data from the LPC bus is developed. The data captured by tapping the LPC bus are then sent via the serial line and saved to a text file. Subsequently, the data are analyzed, and it is discovered that, when certain conditions are met, BitLocker Volume Master key can be found between the fetched data. This key can be used to decrypt the drive of the targeted machine. This way, an evil maid type attack is carried out. The attacker who got hold of a target machine can thus read previously encrypted data from the drive. The FPGA design is loaded into a physical board Basys3. Before connecting the FPGA to a real LPC bus, the design and implementation are tested, at first using simulation and after that, Arduino is utilized to mimic the behavior of an LPC bus.

**Keywords** trusted platform module, low pin count bus, FPGA, volume master key, BitLocker, Verilog, Arduino, Basys3

---

# Abstrakt

Práce se zaměřuje na komunikaci bezpečnostního čipu TPM po sběrnici LPC a vývojem FPGA designu pro její odposlech. Ve své teoretické části shrnuje strukturu a nejdůležitější případy užití bezpečnostního čipu TPM a základní informace o sběrnici LPC. Praktická část je věnována návrhu hardwaru v FPGA pro zachycení komunikace na LPC sběrnici a filtraci dat týkajících se pouze TPM. Data zachycená odposlechem sběrnice se následně pošlou přes sériovou linku a uloží do textového souboru v čitelné podobě. Poté proběhne jejich analýza a je zjištěno, že se zde za určitých podmínek nachází i klíč Volume Master Key nástroje Bitlocker. Tento klíč se dá použít dešifrování disku zařízení, na kterém je odposlech prováděn. Tímto způsobem je prakticky proveden útok typu evil maid, kdy útočník, který se zmocnil cílového zařízení, dokáže přečíst z disku zašifrovaná data. Navržený FPGA design je nahrán a spuštěn na fyzické desce Basys3. Design a implementace jsou před použitím na reálné LPC sběrnici otestovány jak simulací, tak pomocí platformy Arduino, která zde napodobuje chování sběrnice.

**Klíčová slova** trusted platform module, low pin count bus, FPGA, volume master key, BitLocker, Verilog, Arduino, Basys3

---

# Contents

|  |           |
|--|-----------|
| <b>Introduction</b>  | <b>1</b>  |
| <b>1 Theoretical Background of Trusted Platform Module</b> | <b>3</b>  |
| 1.1 Structure of TPM . . . . .                             | 3         |
| 1.2 Cryptographic Processor Capabilities . . . . .         | 4         |
| 1.3 Key Hierarchy in TPM . . . . .                         | 5         |
| 1.4 Platform Configuration Registers . . . . .             | 6         |
| 1.5 TPM Use Case Scenarios . . . . .                       | 6         |
| 1.5.1 Device Identification . . . . .                      | 6         |
| 1.5.2 Data Encryption . . . . .                            | 7         |
| 1.5.3 Key Management . . . . .                             | 7         |
| 1.5.4 Anti-Hammering . . . . .                             | 7         |
| 1.5.5 Measured Boot . . . . .                              | 8         |
| 1.5.6 Windows BitLocker . . . . .                          | 8         |
| 1.6 TPM Software Stack . . . . .                           | 10        |
| 1.7 Low Pin Count Bus . . . . .                            | 11        |
| <b>2 Analysis &amp; Design</b>                             | <b>13</b> |
| 2.1 Overview of Target Functionality . . . . .             | 13        |
| 2.2 FPGA Introduction . . . . .                            | 14        |
| 2.2.1 Reason for Choosing FPGA . . . . .                   | 14        |
| 2.2.2 FPGA Development Environment . . . . .               | 15        |
| 2.3 Modules of the FPGA Design . . . . .                   | 16        |
| 2.3.1 Filtration Finite-State Machine Module . . . . .     | 16        |
| 2.3.1.1 TPM on LPC Protocol . . . . .                      | 17        |
| 2.3.1.2 Transaction Abort Mechanism . . . . .              | 19        |
| 2.3.2 First In First Out Module . . . . .                  | 19        |
| 2.3.3 UART Transmitter Module . . . . .                    | 20        |
| <b>3 Implementation</b>                                    | <b>23</b> |

|          |  |           |
|----------|--|-----------|
| 3.1      | Implementing the Design Modules . . . . .                | 23        |
| 3.1.1    | Filtration Finite State Machine Module . . . . .         | 23        |
| 3.1.2    | First In First Out Module . . . . .                      | 25        |
| 3.1.3    | UART Transmitter Module . . . . .                        | 26        |
| 3.1.3.1  | Logic and Synchronization . . . . .                      | 26        |
| 3.1.3.2  | Transmission Finite State Machine . . . . .              | 27        |
| 3.2      | Connecting the Modules Together . . . . .                | 27        |
| 3.2.1    | Connecting the Transmitter and the FIFO . . . . .        | 28        |
| 3.2.2    | Completing the Final Module . . . . .                    | 30        |
| 3.3      | Processing the Serial Data . . . . .                     | 31        |
| <b>4</b> | <b>Simulation &amp; Testing</b>                          | <b>33</b> |
| 4.1      | Behavioral Simulation Using Vivado . . . . .             | 33        |
| 4.1.1    | Simulating the Filtration Finite State Machine . . . . . | 34        |
| 4.1.2    | Simulation of FIFO . . . . .                             | 36        |
| 4.1.3    | Simulation and Testing of UART Transmitter . . . . .     | 36        |
| 4.2      | Functional Testing on Basys3 Board . . . . .             | 36        |
| 4.2.1    | Testing the Serial Line Communication . . . . .          | 37        |
| 4.2.2    | Creating a Slow Clock Module . . . . .                   | 38        |
| 4.2.3    | Testing the FIFO and Transmitter . . . . .               | 39        |
| 4.2.4    | Testing the Final Design . . . . .                       | 40        |
| 4.3      | Testing the Design with Arduino . . . . .                | 41        |
| 4.3.1    | Wiring Arduino and Basys3 Together . . . . .             | 41        |
| 4.3.2    | Programming the Arduino . . . . .                        | 43        |
| <b>5</b> | <b>Forensics of Captured Data</b>                        | <b>45</b> |
| 5.1      | Configuration of FPGA Design . . . . .                   | 45        |
| 5.2      | Searching for the VMK . . . . .                          | 46        |
| 5.2.1    | Structure of the VMK on the LPC Bus . . . . .            | 46        |
| 5.2.2    | Finding the VMK Between Captured Data . . . . .          | 47        |
| 5.3      | Conclusion of the Forensic Analysis . . . . .            | 48        |
| 5.4      | Mitigation . . . . .                                     | 49        |
|          | <b>Conclusion</b>  | <b>51</b> |
|          | <b>Bibliography</b>                                      | <b>53</b> |
|          | <b>A Source Code Fragments</b>                           | <b>57</b> |
|          | <b>B Acronyms</b>  | <b>61</b> |
|          | <b>C Contents of Enclosed SD Card</b>                    | <b>63</b> |

---

# List of Figures

|     |  |    |
|-----|--|----|
| 1.1 | Structure of TPM chip . . . . .  | 4  |
| 1.2 | Diagram of measured boot processes . . . . .   | 8  |
| 1.3 | Layers of the TPM software stack (TSS) hierarchy . . . . .   | 10 |
| 2.1 | Scheme of the desired purpose of the FPGA design. . . . .  | 13 |
| 2.2 | Picture of Basys3 FPGA board . . . . .   | 14 |
| 2.3 | FPGA design . . . . .  | 16 |
| 2.4 | Inputs and outputs of the filtration finite-state machine . . . . .  | 17 |
| 2.5 | Typical timing of LFRAME . . . . .   | 17 |
| 2.6 | Inputs and outputs of the FIFO module . . . . .  | 20 |
| 2.7 | Inputs and outputs of the UART transmitter module . . . . .  | 20 |
| 3.1 | Diagram of finite state machine handling LPC transactions . . . . .  | 24 |
| 2.7 | UART Transmitter inputs and outputs . . . . .  | 26 |
| 3.2 | Inputs and outputs of final module . . . . .   | 28 |
| 3.3 | Connection of UART transmitter and FIFO modules . . . . .  | 29 |
| 3.4 | Connection of UART transmitter + FIFO module and filtration<br>FSM module to complete the final module . . . . . | 30 |
| 4.1 | Waveform window of the filtration FSM simulation . . . . .   | 35 |
| 4.2 | Waveform of one LPC transaction in the filtration FSM simulation   | 35 |
| 4.3 | Basys3 Board Features . . . . .  | 37 |
| 4.4 | Output of Python script during internal test of final module . . . . .   | 40 |
| 3.2 | Inputs and outputs of final module . . . . .   | 41 |
| 4.5 | Basys3 Pmod Head Scheme . . . . .  | 42 |
| 4.6 | Basys3 and Arduino Connection Picture . . . . .  | 43 |
| 5.1 | Discovery of VMK between captured data . . . . .   | 48 |



---

## List of Code Snippets

|     |  |    |
|-----|--|----|
| 3.1 | Read signal in the UART transmitter module . . . . .                       | 29 |
| 3.2 | <i>data_valid</i> signal in the filtration FSM module . . . . .            | 31 |
| 4.1 | Generation of clock signal in testbench files . . . . .                    | 34 |
| 4.2 | Test step of filtration FSM simulation . . . . .                           | 35 |
| 4.3 | <i>Write</i> and <i>read</i> operations in FIFO module testbench . . . . . | 36 |
| 4.4 | Example of setting constraints for ports . . . . .                         | 38 |
| 4.5 | Module generating a slower clock signal . . . . .                          | 38 |
| 4.6 | Input generator for internal testing of FIFO . . . . .                     | 39 |
| 4.7 | Input generator for internal testing of the final module . . . . .         | 40 |
| 4.8 | Constraint for assignment of Basys3 pmod pins . . . . .                    | 43 |
| A.1 | Logic section of the UART transmitter module. . . . .                      | 57 |
| A.2 | Finite state machine in the UART transmitter module . . . . .              | 58 |
| A.3 | Code of Python script handling serial line data. . . . .                   | 59 |
| A.4 | Generation of clock with a frequency of 1 Hz in Arduino UNO. . . . .       | 59 |
| A.5 | Generation of clock with a frequency of 8 MHz in Arduino UNO. . . . .      | 59 |
| A.6 | Arduino UNO function handling clock changes . . . . .                      | 60 |





---

## List of Tables

|     |   |    |
|-----|---|----|
| 2.1 | TPM Write cycle on the LPC bus . . . . .        | 18 |
| 2.2 | TPM Read cycle on the LPC bus . . . . .         | 18 |
| 3.1 | Configuration of FIFO IP module . . . . .       | 25 |
| 5.1 | BitLocker volume master key structure . . . . . | 47 |



---

# Introduction

Nowadays, most people and businesses use electronic devices to complete their daily tasks. They use various programs and digital platforms for this purpose, accessing them either locally or remotely. In both of these cases, the software is used by a particular user working on a concrete device.

It would be desirable to prove that the operations performed by these applications are trusted. In other words, the software is not malicious, the user is trustworthy, and, most importantly, the configuration state of the entire device is genuine.

This is the moment when a hardware chip is suitable. It might not have been well known to a typical user that there is a piece of hardware dedicated strictly to security purposes inside their computer.

That has probably changed, as Microsoft recently released a new version of their operating system, Windows 11. To upgrade, the device is now required to contain the Trusted Platform Module 2.0 (TPM 2.0) chip.

Although the focus is mainly on personal computers in the implementation part of this thesis, the use of the TPM is not limited to them. It can be used in smartphones, tablets, gaming consoles, televisions, or in-car computers with various operating systems.

TPM helps these devices by performing cryptographic operations and adds a factor of physical security. During some types of attacks, an attacker could try to obtain data from the memory of a computer, which may contain secret information, such as passwords or encryption keys. The TPM adds protection against these attacks by being separate from the memory and taking care of these secrets itself. It also helps protect against the injection of malicious code into the memory. Ransomware, which is a term that has recently been heard a lot in the news, can be mentioned as an example.

The TPM is a separate hardware component, so it must be connected to the rest of the system somehow, that is via the LPC bus. What are the data that are sent from and to the TPM? Since it is a security module, is it possible to obtain some interesting or even private information after capturing these

data? In this thesis, the answer is shown to be *yes*, when certain conditions are met.

With the requirement mentioned above, the TPM becomes more relevant, and its existence becomes even more important in the present day. It is interesting to look into its features deeper and explore its implementation in greater detail. It is also an opportunity to present this information to a professional audience without an expertise in hardware security in digestible form, since there are not many technical sources with reasonable depth of information describing the TPM.

The main goal of the thesis is to develop a solution that reads the TPM traffic from a low pin count bus and presents it in a human-readable form on the user's computer. Subsequently, the obtained data are analyzed. The desired output is a properly tested FPGA design capable of filtering the TPM related data and outputting them via a serial line.

---

# Theoretical Background of Trusted Platform Module

The main concepts of trusted platform module (TPM) chip are introduced in this chapter, along with its most significant use cases. It also involves a simple overview of the LPC bus and Windows Bitlocker, which is one of the use cases of TPM. As will be shown in the following chapters, it deserves a more detailed description in context with the practical part of the thesis.

TPM is a security chip introduced by the Trusted Computing Group, who is also the author of its specification. The idea behind creating such a chip is to enhance security at the hardware level.

The TPM can perform various cryptographic operations, it provides multiple use cases where software solution is not sufficient or trusted. It can be used to generate and store cryptographic keys and certificates. One of the main features is ensuring the platform's integrity by measuring the boot code during the boot phase of the system. More detail on use cases of TPM will be provided later in this chapter. In addition, it serves as a technology for device authentication using a unique RSA key that is burned into the TPM chip. [4]

TPM works as a passive device which receives commands, processes them, and returns responses. The discrete TPM is typically connected to the motherboard via the SPI or LPC bus. This thesis addresses the LPC connection and focuses at the data that are flowing through the bus into the TPM. Later in the thesis, it is revealed that some confidential data of Windows Bitlocker can be retrieved from the LPC-TPM communication.

## 1.1 Structure of TPM

The Trusted Platform Module chip consists of three main sections, cryptographic processor, non-volatile, and volatile memory. All computation takes place in the cryptographic processor, which performs essential cryptographic

## 1. THEORETICAL BACKGROUND OF TRUSTED PLATFORM MODULE

---

operations, such as encrypting, decrypting, and data signing. It is also capable of producing data hashes and generating RSA key pairs. A random number generator is also present. [17]

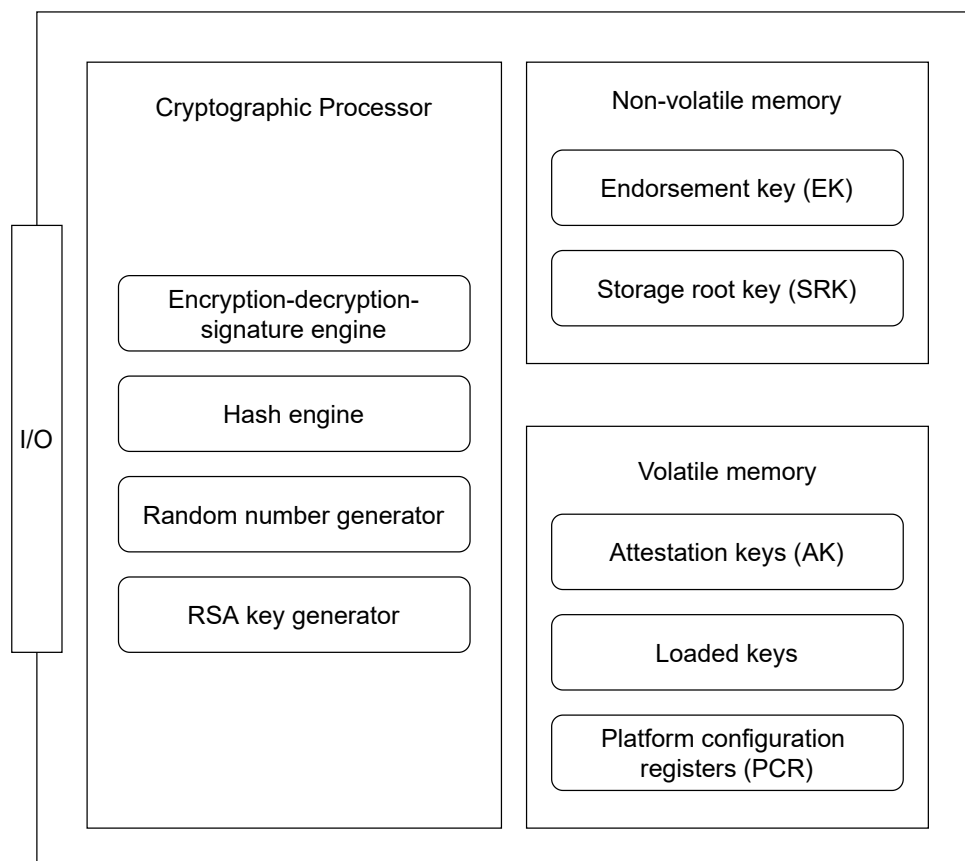


Figure 1.1: The TPM chip structure consists of three pieces. The cryptographic processor performs essential cryptographic computation, the non-volatile and volatile memory are used for storing keys and loading measured values into PCRs. SPI or LPC interface is present to communicate. [1]

The non-volatile memory contains the endorsement and storage root keys, their significance is discussed in a section about the key hierarchy (1.3).

The attestation keys are also discussed in this section. They reside in volatile memory together with the platform configuration registers, which also have a dedicated section (1.4).

### 1.2 Cryptographic Processor Capabilities

There are multiple algorithms that the TPM 2.0 cryptographic processor can use. The encryption-decryption engine is capable of utilizing both symmetric

algorithms, including AES with key sizes of 128 or 256 bits, and asymmetric ciphers, including elliptic curve cryptography (ECC) along with the RSA algorithm.

The hash engine was able to compute only the SHA-1 hash in TPM 1.2. Since the algorithm has been cryptographically broken, it has been deprecated in the TPM 2.0 specification, and multiple newer hash functions are now supported, including SHA-3 or SHA-512.

The random number generator is also present in the cryptographic processor. It serves as a source of randomness for TPM, it is utilized to generate nonces<sup>1</sup> and keys, as well as to provide randomness in cryptographic signing. Software can also make use of a secure RNG present in the TPM by interacting with the TPM's API (more information on TPM API is provided in section 1.6). [1], [5]

### 1.3 Key Hierarchy in TPM

Three types of encryption keys can be found inside a TPM chip. The RSA key pair that is fundamental to the TPM is *Endorsement key (EK)*. It is permanently present in the TPM from the time of manufacture, and its private part never leaves the chip. *EK* practically serves as a TPM identifier. Additionally, a TPM typically provides an endorsement certificate, which is stored in the TPM's internal memory. This certificate is signed by the manufacturer, which ensures that the TPM chip is genuine. [7], [5]

However, the *Endorsement key* pair is used only in a limited number of procedures, as there would be a possibility of identifying the device with its continuous usage. [7] Therefore, for routine transaction, there are *Attestation keys (AK)*. Their purpose is to sign data to prove that they come from a true TPM. That means there has to be also a method to prove that a particular *AK* originates from a genuine *EK*. This can be done by using an Attestation Certificate Authority or encrypting *AK* with *EK* (or another *AK*) and, therefore, associating it with the endorsement hierarchy. Simply put, *AK* can be treated as an alias for *EK*. Typically, a service or application that needs to use *EK* can use *AK* dedicated to this service in particular. [5], [6]

Another important feature is protecting TPM keys created by applications. The *Storage Root Key (SRK)* participates in that. It is the primary key of the owner hierarchy, which means that all the generated keys used for signing and encrypting are derived from *SRK*. It is then possible to verify that a key truly originates from a particular TPM owned by a concrete entity. When a new user takes ownership of the TPM, new *SRK* is created. [8], [7]

---

<sup>1</sup>“A nonce is a random or semi-random number that is generated for a specific use. It is related to cryptographic communication and information technology (IT). The term stands for “number used once” or “number once” and is commonly referred to as a cryptographic nonce.” [34]

## 1.4 Platform Configuration Registers

Platform configuration registers (PCR) act as a way to measure the state of software. Both the software itself and its configuration data are included in the measurement. The recorded values can be then read or a signed report of their state, called an *attestation*, can be generated. It is possible to use the measured data, for example, to detect an unpermitted change of the platform state during the boot phase.

What does it mean to perform a measurement? The values recorded into the registers are obtained as hashes of the data that form the current platform state, e.g. the Master Boot Record. In TPM 1.2, the SHA-1 hash function was the only supported. Now, with version 2.0, the hash algorithm is not strictly defined and can be changed.

An important attribute of PCRs is the method of updating their records. The calculation is called *extend*. Instead of a simple hash of the currently measured state, it is extended to the previously saved value and subsequently hashed.

*Calculation of n-th PCR value:  $PCR[n] = Hash(PCR[n] || Argument)$*

By the use of this operation, one of the main PCR tasks is fulfilled. The history of the measurements is recorded, and due to the non-reversibility of hash functions, there is no way to set a measurement back to a desired state by an adversary. [1], [3]

## 1.5 TPM Use Case Scenarios

A selection of the most common applications of a TPM will be listed in this section.

### 1.5.1 Device Identification

The existence of a unique private key for each TPM chip can be utilized to identify a particular device. There is a possibility to use the machine equipped with TPM as an authentication factor of ownership. This can be useful, for example, in larger organizations. It is often desired to only allow access to their system to certain machines. [1]

For authentication purposes, a factor of knowledge represented by a PIN can also be useful. Thus, the machine can be used as a smart card. Therefore, anyone who knows the PIN and has access to this machine can be authorized to perform certain operations. Specific use cases can be, for example:

- VPN identifying a user or machine before granting access to a network
- User signing or decrypting e-mail
- User authorizing a payment [1]



### 1.5.2 Data Encryption

The TPM has an encryption-decryption engine included in its structure. It is very useful for encrypting or decrypting cryptographic keys used by various pieces of software. This way, they can be safely stored in memory and then decrypted by the TPM. The engine is also capable of cryptographic signing<sup>2</sup>, which is an essential feature of the TPM used in its many applications, such as PCR reports, to name one.

The presence of the engine enables a lot of important features, including file and folder encryption on a device, full disk encryption (used for example by Windows BitLocker), encryption of passwords for a password manager, or encryption of files stored remotely. [1]

### 1.5.3 Key Management

A very large portion of TPM usage involves dealing with cryptographic keys. A key can come to reside in a TPM in three ways: generating it on its own using a seed, using an RNG, or importing a key into the TPM. [1] Letting the TPM generate a key can be advantageous because of its isolation from the rest of the system, thus preventing tampering attempts.

When a user or process already possesses a key, *wrapping* a key by the TPM might be desirable. It is a process of encrypting a key so that it can only be decrypted by the TPM, and therefore not exposing it to other components, software, processes, or users. [2]

Another feature that can be utilized is *sealing* the key to a TPM. When a key is *sealed*, it is wrapped and tied to a certain platform state, that is, measured values stored in PCRs. When the state of the platform changes, the key is no longer accessible and this action is irreversible. That is why, for example, updating BIOS can be tricky as it changes the platform state. Windows BitLocker, for example, uses the technique of *sealing*. [1], [7]

It would be desirable to store the generated keys securely. This can be done either in non-volatile memory inside the TPM or by storing them on a hard drive and *wrapping* them by encrypting the keys with self-generated public key and later decrypting them with the private key that never leaves the TPM. [1]

### 1.5.4 Anti-Hammering

Another topic worth mentioning is the anti-hammering lockout policy defined in the TPM 2.0 standard. Naturally, it is desired to prevent dictionary attacks against TPM authorization.

---

<sup>2</sup>“Cryptographic digital signatures use public key algorithms to provide data integrity. When you sign data with a digital signature, someone else can verify the signature, and can prove that the data originated from you and was not altered after you signed it.” [35]

The lockout activates after 32 authorization failures. Each 10 minutes, one of these failed attempts is restored and added to a list of available attempts. That means, after 320 minutes without a failed attempt, the list goes back to the same state as in the beginning, that is 32 attempts. If a failure occurs in between, the failure also counts and an attempt is deducted for 10 minutes.[2]

### 1.5.5 Measured Boot

The process of measured boot provides a platform integrity check. Each part of the boot process chain performs a *measurement* of the code of the following process. The PCR registers are reset to zero with each power-on and each measured value. After the boot, it is possible to verify whether the platform state requirements are met. Also, as mentioned earlier, the measured PCR values can be used for *sealing* to the platform state. Specifically, after a boot, certain values are present in the PCR registers. When some data are sealed to these values, the PCR registers have to hold the same values for the data to be later unsealed successfully. [5], [7]

The first part of the chain has to be a trusted piece of code, as it cannot be measured by any previous process. This code can reside in the TPM, but it can often be found in the BIOS boot block. [7]

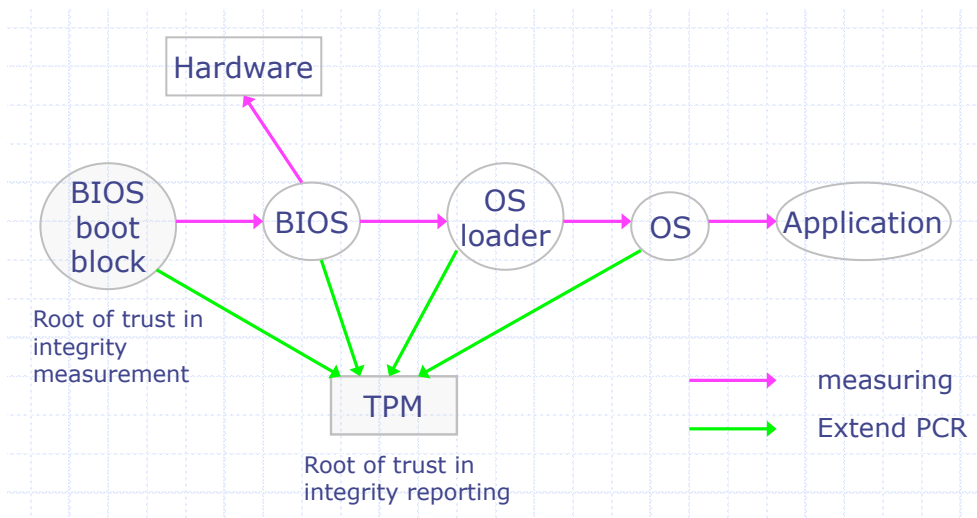


Figure 1.2: Each process of the boot phase measures the following process by extending PCR registers in the TPM. [9]

### 1.5.6 Windows BitLocker

A perfect example of how the TPM chip can be utilized is Windows BitLocker. It is a feature integrated in Windows operating systems first introduced in

Windows Vista. It is used to encrypt the data stored on a hard drive. It adds protection against unauthorized access, for example, when the device or the sole hard drive gets stolen or lost. [19]

BitLocker uses the TPM to check the platform state during the boot phase. If there is no problem, the disk data are decrypted and the Windows operating system boots up.

There are multiple authentication modes in which Bitlocker operates:

- TPM only - Disk data are decrypted immediately after the platform state check is successful. The user does not interact with Bitlocker in any way. This is the default configuration most users employ.
- TPM + PIN - Aside from the successful verification of the state of the platform, the user has to enter a special BitLocker PIN for the drive to be decrypted.
- TPM + startup key - Instead of a PIN, the user inserts a flash drive with the correct startup key and the drive is decrypted.
- TPM + PIN + startup key - This configuration combines the three methods above. [20]

Regarding decryption, the key that is used for it has to be stored somewhere. There are two keys that serve this purpose, *full volume encryption key (FVEK)* and *volume master key (VMK)*. Both are stored on the drive itself, and both are encrypted.

*FVEK* is the key that is used directly to encrypt the raw disk data. It is encrypted by *volume master key*, which is *sealed* to the TPM. Therefore, as described earlier, the TPM decrypts the key only if the PCR measurements are correct. After that, it sends the **unencrypted** key to the OS boot manager, which decrypts *FVEK* and then the data.

To sum it up, the following phases need to occur during boot so that the disk data can be successfully decrypted:

1. The platform state is successfully validated by the TPM.
2. If enabled, PIN or startup key are correctly inserted.
3. *VMK* is unsealed from the TPM which sends it to the OS boot manager, there it is used to decrypt the *FVEK*.
4. Disk data can be decrypted using the *FVEK*. [21], [23]

It is important to note that unsealed *volume master key* has to **go through the LPC bus**.

## 1.6 TPM Software Stack

To make use of the TPM as a software developer, it is much more convenient to use an API than raw communication to interact with the TPM. A program can call the TPM to utilize its features, such as *key wrapping* or generating a random number. There is a specification defined by the TCG defining *TPM Software Stack* and the hierarchy of APIs.

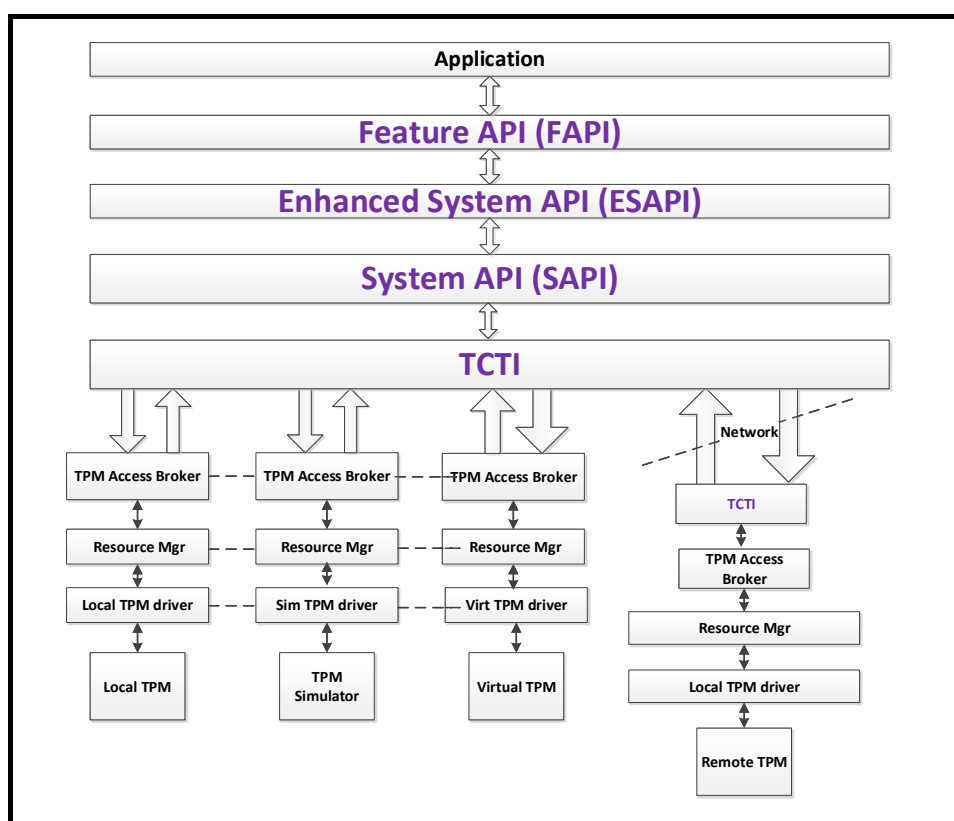


Figure 1.3: Layers of the TPM software stack (TSS) from the highest to the lowest level of abstraction. [11]

### TPM API Layers

The highest layer, *Feature API (FAPI)*, is designed to provide TPM features at the highest level of abstraction. It is meant to be mapped one-on-one with TPM commands, however, not all corner cases are included. Some default selections of algorithms and values implemented. [1], [10]

The *System API (SAPI)* enables the use of all the TPM features, but it also comes with more complexity than the *FAPI*. Above the *SAPI* sits

the *Enhanced system API (ESAPI)*. This layer extends *SAPI* and provides simplification to some features, such as session management. [1]

Below these layers, there resides the *TPM Command Transmission Interface (TCTI)*. It is the layer used to transmit TPM commands and receive responses. Binary streams can be sent and received via *TCTI*. [1], [10]

The *TPM Access Broker (TAB)* is responsible for synchronization of processes that access the TPM. It ensures that multiple processes accessing the TPM do not interfere with each other. [10]

Another daemon, *Resource manager*, takes care of the TPM context. Due to the small capacity of the TPM memory, there is a limitation in resource loading. The RM works similarly to a virtual memory manager and swaps TPM objects and sessions in and out of memory. [1], [10]

## 1.7 Low Pin Count Bus

The Low Pin Count (LPC) is a protocol for an interface between low-bandwidth devices and their connection to the CPU developed by Intel. It replaced their Industry Standard Architecture (ISA)<sup>3</sup>. The LPC operates using a minimum of 7 signals, with 4 bits for the data. That is noticeably less compared to ISA, which contains approximately 40 pins. Also, the data transfer frequency is higher (33 MHz) compared to 8 MHz. The LPC bus provides CPU connection, for example, to TPM or redundant BIOS. [12], [13]

LPC specification requires seven signals:

- Four serial LAD signals for carrying multiplexed data including cycle type, cycle direction, chip selection, address, data, and wait times
- One LCLK clock signal of 33 MHz provided by the host
- One LFRAME to indicate the start or stop of a transaction
- One LRESET to perform bus resets [12]

The LPC bus defines two terms - *host*, which is the part of the interface connected to the CPU, and *peripheral*, which is a separate device, such as a chip or an embedded controller. For example, *TPM* is a peripheral in terms of the LPC bus. LPC enables *Direct Memory Access (DMA)*, memory reads and writes, or standard input and output reads or writes. There are multiple types of transaction cycles for these types of operations. [18] In section 2.3.1.1, the TPM transaction, which is similar to the standard input-output transaction, is described in great detail.

---

<sup>3</sup>“An earlier hardware interface for connecting peripheral devices in PCs. ISA accepted cards for sound, display, hard drives, and other devices. Originally called the “AT bus” and introduced with the IBM PC AT in 1984, the AT/ISA bus extended the PC bus from 8 to 16 bits.”



## Analysis & Design

### 2.1 Overview of Target Functionality

The goal of the implementation part is to create an FPGA design that will filter the data coming from the LPC bus. The FPGA will be connected to the bus and all LPC data will be flowing into it. These data will be processed by the FPGA, so that the output includes only communication with the TPM. The output will flow to a computer via serial line. A Python script will be running on this computer and listening on the serial port. It will save the captured data to a file in a human-readable format, so that it can be used for further analysis.

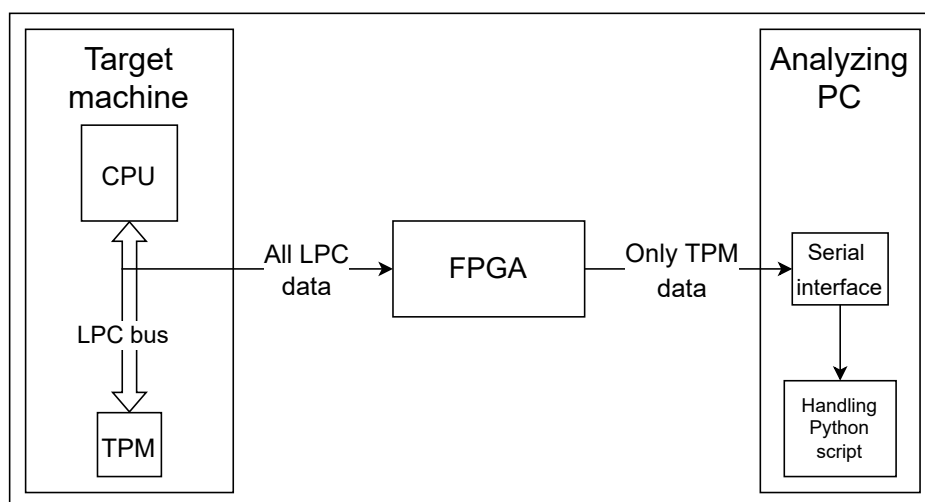


Figure 2.1: The FPGA board connects to the LPC bus and receives all the LPC data on the input. It acts as a filter and outputs only TPM related data via serial interface.

## 2.2 FPGA Introduction

The key component used in this work is the FPGA (field-programmable gate array). It is a set of integrated circuits consisting of programmable logic gates, memory, and other parts. [15] The board can be programmed and optimized for a particular workload, which is an advantage compared to a piece of software running on a CPU, as it is noticeably faster.

In this work, I will be using a *Basys3* design kit. It is based on an FPGA Artix 7 (xc7a35tcbg236-1). It is a kit that is used in one of the courses at our faculty, so it was convenient to borrow it for this thesis. Also, its parameters are completely sufficient for the implementation part.

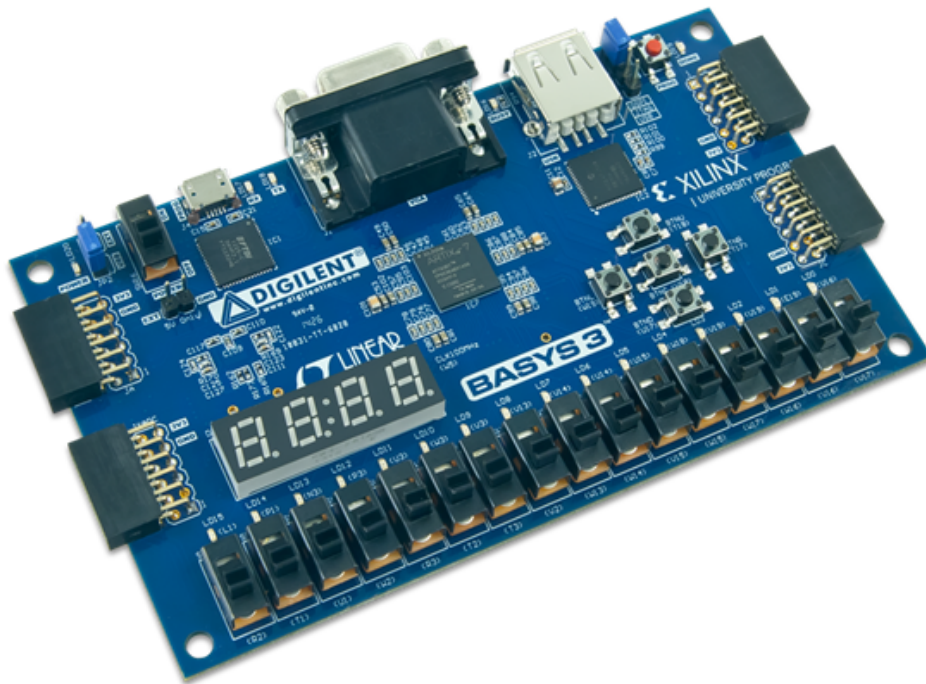


Figure 2.2: The final design of the implementation part will be loaded onto the Basys3 FPGA board, which is shown in this picture. [16]

### 2.2.1 Reason for Choosing FPGA

In the assignment, it is explicitly stated to use an FPGA to capture the TPM data. There is a reason for that, as other methods of sniffing the data exist. To mention one, it would be possible to obtain all data (not only related to



TPM) from the bus using a logic analyzer<sup>4</sup>. Analysis and filtration could then be performed using a visualization of the signals or an analyzing script on a computer.

However, there could be problems with the use of the analyzer. For example, not every visualizing and interpreting software supports the LPC protocol, therefore, it could be necessary to modify the software. Moreover, the analyzer may have a slow sample rate<sup>5</sup>, which results in some of the data not being captured correctly.

Using the FPGA, it is possible to filter the relevant data on the fly by only designing a state machine that defines which data to send to the serial line and which to ignore. It has another advantage, being that there is no need for any analyzing script that would analyze and interpret all the LPC data after capturing. This responsibility is transferred to the FPGA.

### 2.2.2 FPGA Development Environment

For developing the FPGA design, it was decided to use the Vivado design suite. It is a program for the synthesis<sup>6</sup> of FPGA designs that are written in an HDL (hardware description language). Aside from synthesis, it also allows highlighting of source code, provides an interface to generate a bitstream<sup>7</sup> specifically for any selected FPGA board, and also helps to program the target board with the generated bitstream. Finally, it can be useful to test the design with the simulation feature, where the developer can view the behavior of the signals. This way, it is possible to debug the code before loading it into the FPGA. Vivado has been chosen because it is used during one of the university courses at CTU.

The gist of programming the FPGA is a source code written in a HDL. Two of the most known HDLs are *VHDL* and *Verilog*. For the thesis, I chose *Verilog* as I have at least a small experience with this language. I cannot say the same thing about *VHDL*. Vivado supports both languages, so this is not an issue.

---

<sup>4</sup>“A logic analyzer verifies that the digital circuit is working and helps you troubleshoot problems that arise. The logic analyzer captures and displays many signals at once, and analyzes their timing relationships.”[14]

<sup>5</sup>“The sample rate is how often a logic analyzer samples all of its channels.” [38]

<sup>6</sup>“Logic synthesis is the process of automatic production of logic components, in particular digital circuits.” [39]

<sup>7</sup>A binary file defining behavior of a specific FPGA board.

## 2.3 Modules of the FPGA Design

There are three main components that have to be developed and assembled together for the final design. The idea is that the components are designed and developed separately before being connected together via their input and output interface.

- *Filtration finite state machine (filtration FSM)* - It is the main logic of the design that decides which data flowing from the LPC bus are TPM related, therefore sent to the output, and which are ignored.
- *FIFO* - It takes care of asynchronous clock frequencies of the FPGA and the LPC bus. Also, it stores the data received from *filtration FSM* before sending them to output via *UART transmitter*.
- *UART transmitter* - This component reads the filtered data from *FIFO* and transmits them from the FPGA to the computer via serial line.

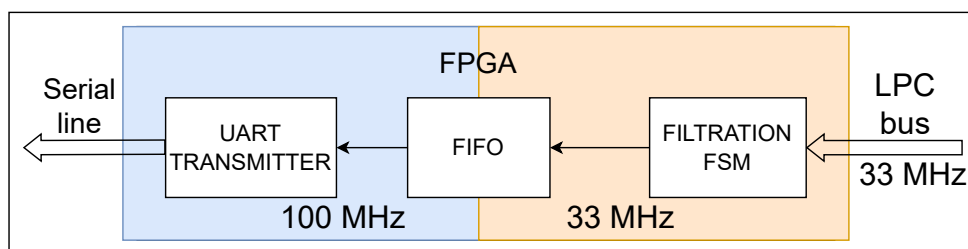


Figure 2.3: The input clock frequency from the LPC bus is around 33 MHz. The filtration FSM works on this frequency and the FIFO expects it to write data with 33 MHz. FIFO's read frequency is 100 MHz, which is coming from the FPGA internal clock. It expects the transmitter to read using this frequency.

### 2.3.1 Filtration Finite-State Machine Module

As mentioned above, the *filtration FSM* module acts as the main logic to detect TPM communication on the LPC bus. The input signals come directly from the LPC bus, it is enough to process 6 of them - 4 *LAD* bits, 1 *LFRAME* signal, which indicates the start or stop of a transaction, and finally 1 *LCLK* clock signal with frequency around 33 MHz.

To create a correct state machine, it is necessary to look into the LPC communication protocol and, most importantly, determine how this protocol is used by the TPM transactions. The purpose of the state machine is to handle the transactions and send the data bytes to the output.

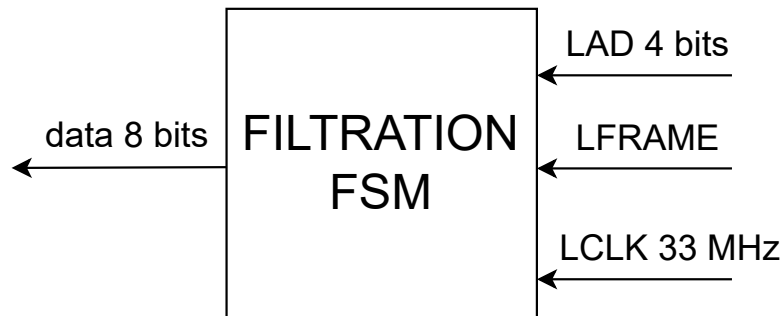


Figure 2.4: The filtration finite-state machine takes three inputs directly from the LPC bus. It also works at LPC clock frequency of 33 MHz.

### 2.3.1.1 TPM on LPC Protocol

Each LPC transaction cycle starts in the same way. The *LFRAME* signal is set to low, and at the same time, a start value is visible on the *LAD* input. By the start value, a peripheral device that uses the LPC bus can determine whether the upcoming cycle applies to it. [18]

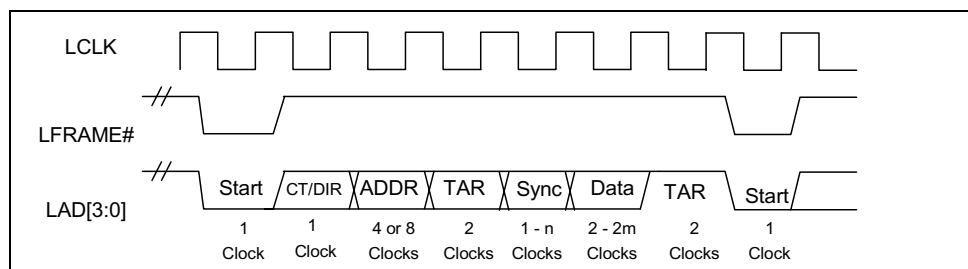


Figure 2.5: Typical timing of an LPC transaction is shown here. The *LFRAME* signal is set to low in the start cycle of a transaction.

[18, Figure 2]

Looking into the Client Platform TPM Specification, the protocol used by the TPM can be seen. There are two possible operations with the TPM on the bus, read and write. Also, as the specification states, both TPM read and write operations resemble the standard LPC I/O operations. The start value for TPM transactions is *0101*, which will be important to reflect in the implementation of the finite-state machine. [17]

There are several more fields to be read after the start signal, which together form an LPC transaction. Their description is as follows:

- *CYCTYPE + DIR* - indicates the type of cycle and its direction. In this particular case, it is only necessary to remember that *0000* means TPM

## 2. ANALYSIS & DESIGN

---

| Field         | Value for Bits [3:0] | Description   |
|---------------|----------------------|---|
| START         | 0101                 | Previously this was a reserved value. It is now allocated for TPM-Write and TPM-Read locality cycles. |
| CYCTYPE + DIR | 0010                 | Same as used for standard LPC I/O Write   |
| ADDR          | See Description      | Four nibbles. Same as the standard LPC I/O Write.   |
| DATA-Low      | DIGEST low nibble    |   |
| DATA-High     | DIGEST high nibble   |   |
| TAR           |                      | Standard LPC TAR  |
| SYNC          |                      | Standard SYNC field for an I/O Write  |
| TAR           |                      | Standard LPC TAR  |

Table 2.1: TPM Write cycle on the LPC bus.

[17, Table 43]

| Field         | Value for Bits [3:0] | Description   |
|---------------|----------------------|---|
| START         | 0101                 | Previously this was a reserved value. It is now allocated for TPM-Write and TPM-Read. |
| CYCTYPE + DIR | 0000                 | Same as used for standard LPC I/O Read  |
| ADDR          | See Description      | Same as for TPM-Write   |
| TAR           |                      | Standard LPC TAR  |
| SYNC          | Standard             | Standard SYNC field for an I/O Read   |
| DATA-Low      | DIGEST low nibble    |   |
| DATA-High     | DIGEST high nibble   |   |
| TAR           |                      | Standard LPC TAR  |

Table 2.2: TPM Read cycle on the LPC bus.

[17, Table 44]

read and *0010* means TPM write.

- ADDR - indicates which address of the peripheral device is being accessed. For the implementation, it is necessary to know that this field is 4 clocks wide (so the address is 16 bits long).
- TAR - turn-around field is 2 clocks wide. Control is transferred to the peripheral during these 2 clock cycles.
- SYNC - this field is used to wait for the peripheral device. The width of this field can differ depending on the peripheral. What is important, the *LAD* signal contains *0000* when the operation is completed.
- DATA-Low and DATA-High - 4 bits of data (1 nibble) are sent through the *LAD* input. Both fields are one cycle wide. Together, these two fields combine one data byte related to one LPC transaction. In the finite state machine, both nibbles are assembled, sent to the output, and written to the FIFO at the end of each transaction. [17], [18]

### 2.3.1.2 Transaction Abort Mechanism

It is also necessary to consider the fact that an LPC transaction can be aborted. This situation typically occurs when a peripheral drives the LPC bus and the transaction is waiting for a SYNC signal. This might take a long time, and in that case, the transaction has to be aborted. However, the abort is not limited to this situation, it can happen during any transaction field.

The *LFRAME* signal indicates the abort of a transaction. It is driven high and stays high for at least 4 clock cycles. Also, the value of *LAD* switches to *1111* during the 4 clock cycles. After that, the *LFRAME* signal stays high for one clock cycle and a new transaction can begin. [18]

Any data that were read or written during an aborted transaction should not be taken into account. Therefore, in the filtration finite-state machine module that handles LPC transactions, it is important to write the data after ensuring that the transaction is completed successfully.

### 2.3.2 First In First Out Module

This module is very important in the design as it solves two major problems. As can be seen in the FPGA design (figure 2.3), two different clock domains meet here. The LPC bus runs on a clock with a frequency of around 33 MHz, whereas the Basys3 clock frequency is 100 MHz.

For crossing the two clock domains, FIFO (first in, first out buffer) is used. The data are written there from the finite-state machine with clock frequency of 33 MHz obtained from the LPC bus. The data are then read by the UART transmitter that runs on the 100 MHz clock.

Moreover, the FIFO stores the data after receiving it from the filtering finite-state machine. This feature is needed because the UART transmitter takes a while to transmit a byte, and more data can be received from the FSM during the transmission of only one byte.

It is also necessary to consider possible overflow of the FIFO. In this case, there is an advantage that the write clock domain is slower than the read clock domain. Therefore, the overflow does not become an issue that quickly. However, because of the speed of the UART transmitter, it can still occur. In that case, the data will not be complete. To delay the point of possible overflow, transmitting a byte over the UART should be as fast as possible. As will be seen in the testing part, if the focus is on the right data, there is no reason to worry about overflow.

As can be seen in the FIFO diagram (2.6), there are also signals that indicate full and empty status. They will help maintain the basic rule when dealing with FIFOs: Never read from an empty FIFO and never write to a full FIFO.

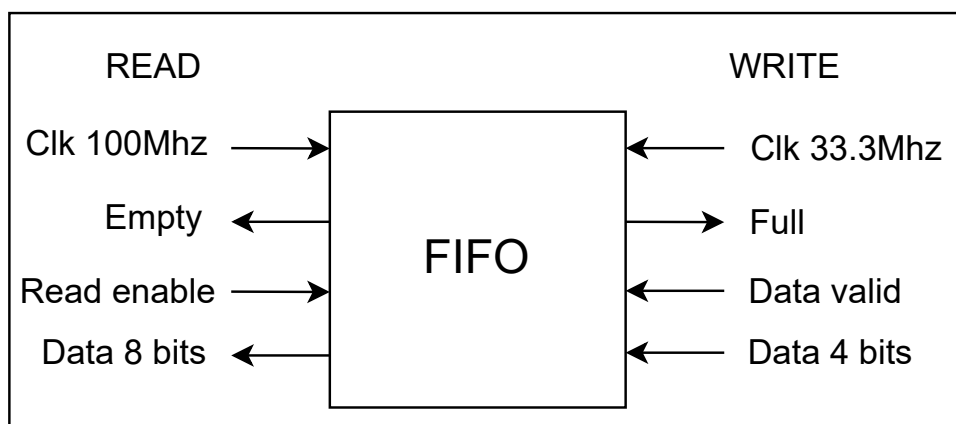


Figure 2.6: The FIFO has three inputs for data write. The filtration FSM performs the data write at the LPC bus clock frequency. The UART transmitter is responsible for reading the data at the frequency of the Basys3 board.

### 2.3.3 UART Transmitter Module

The universal asynchronous receiver-transmitter (UART) serves for sending data to other devices, in this case a computer with a Python script. Since it is only necessary to send data from the FPGA to a computer, the only part that needs to be implemented is the transmitter.

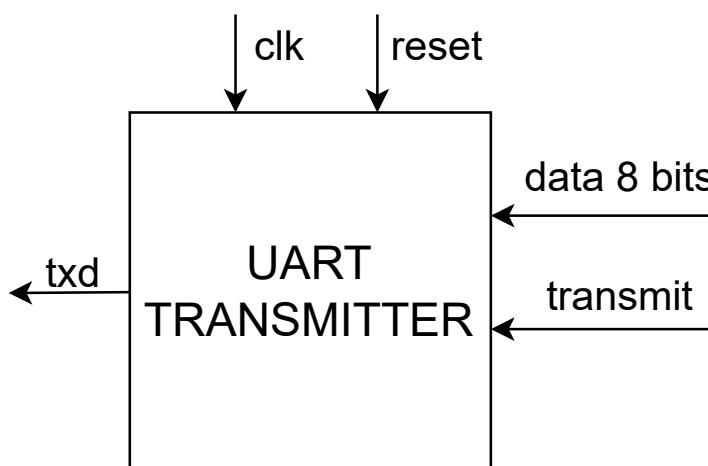


Figure 2.7: The UART transmitter outputs the data via the txd signal. It receives a byte of data as an input and a transmit signal to activate the transmission.

The UART sends individual bytes separately by sending bits one by one. At first, it sends a start bit (0), then 8 bits to form a byte, starting from the least significant bit, and stop bit (1) at the end of a byte.

An important thing to consider is *baud rate*. It determines the rate at which the data are transmitted. It is measured in *bits per second*, and both parties (i.e. receiver and transmitter) have to set the same baud rate.

On Windows operating systems, where the Python script that processes the data will be running, the UART interface is called the COM port.





---

# Implementation

As mentioned in the analysis, the intended approach is to implement each module separately and connect them together later via their input and output interface. As described later in this chapter, this method is only partially successful. Slight adjustments were necessary to allow the modules to work together properly.

It is worth reminding that the source code is written in the *Verilog* hardware description language.

## 3.1 Implementing the Design Modules

### 3.1.1 Filtration Finite State Machine Module

Implementing the finite-state machine is fairly straightforward. It is only necessary to turn this state machine diagram (3.1) into a source file written in Verilog by defining all the states and using a *case* statement. Each case behaves as demonstrated in the diagram which is designed according to the LPC protocol described in section 2.3.1.

By definition, it is a Mealy finite-state machine. The output signals are *data* and *data\_valid*. The *data\_valid* signal is driven high only in the *FTAR2* state, which is the final state, therefore it is safe to say that the transaction completed successfully and was not aborted, as described in section 2.3.1.2.

As seen in the diagram, the *type* register saves the value read in the *TYPE* field, and when the FSM comes to the *ADDR4* state, it is decided what the next state is according to this register. If the value is *0000*, it will move to the branch for reading, if *0010*, it will continue with the write sequence. If none of these values were read, the FSM goes back to the *IDLE* state.

The finite-state machine can be modified to read a given address by changing the parameter *tpm\_address*. By default, it is set as 24hex value, the reason why this address is chosen as default is described in section 5.1. In each of the *ADDR* states, the corresponding parts of the *tpm\_address* parameter and

### 3. IMPLEMENTATION

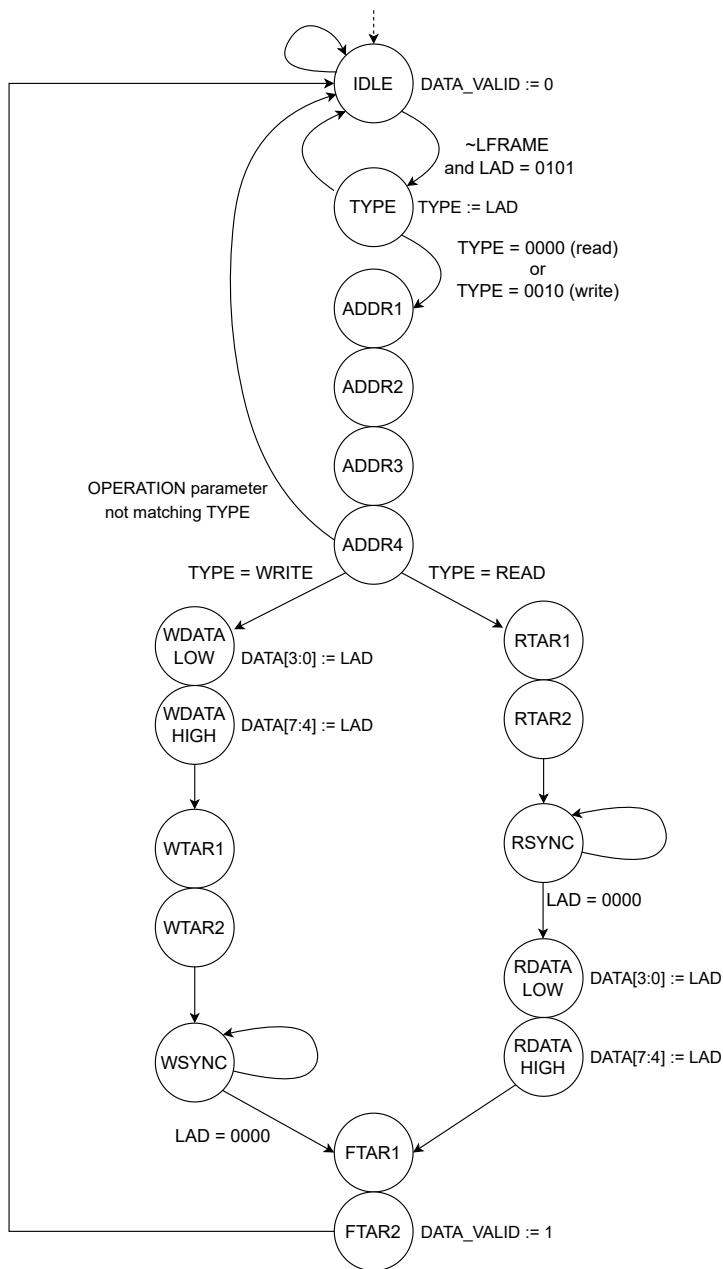


Figure 3.1: Diagram of filtration FSM.  $LFRAME$  and  $LAD$  are the inputs. The FSM behaves according to the TPM on LPC protocol (section 2.3.1.1). It reacts only to TPM transactions and returns the data written or read to the TPM during the transaction. In the last state, it sets the *data\_valid* output high to indicate that the transaction completed successfully with correct data.

the address in the LPC transaction are compared, and if the address does not match, the FSM returns to the *IDLE* state.

The FSM changes its states on the negative edge of the clock. According to the LPC specification, the host and the peripheral device operate on the positive clock edge. This results in a small delay when the *LFRAME* and *LAD* signals change. At the time the negative clock edge is triggered, the data are stable and the FSM can operate with the values (figure 2.5).

If it is necessary to focus on one operation only, the read or write branch can be ignored similarly to the address. It is only necessary to set the *operation* parameter accordingly.

Moreover, another condition that must be taken into account is the *abort* mechanism. In the source code, there is a condition that detects the *abort* mechanism according to the analysis (2.3.1.2). If the *LFRAME* signal is driven low and the FSM is not in the *IDLE* state, the transaction abort is detected and the FSM is reset to the *IDLE* state. Note that the data are not transmitted, as the *data\_valid* signal is high only in the final *FTAR2* state.

### 3.1.2 First In First Out Module

For the purpose of this thesis, it is expected that FIFO behaves as a standard first in, first out. No additional logic is needed inside the FIFO and its functionality is similar to a lot of projects. Therefore, using an already implemented standard FIFO is a reasonable choice. The fact that the design is developed in *Vivado* is advantageous. The design suite provides a library of intellectual property (IP) modules that are ready to use. No coding is needed, it is only necessary to configure the module to the behavior that is expected. In the IP library, a customizable FIFO is available. The configuration is set to match the needs mentioned in the analysis.

|   |                              |
|---|------------------------------|
| Clocking Scheme                               | Independent Clocks           |
| Memory Type                                   | Distributed RAM              |
| Model Generated                               | Behavioral Model             |
| Write Width                                   | 8                            |
| Write Depth                                   | 8191                         |
| Read Width                                    | 8                            |
| Read Depth                                    | 8191                         |
| Almost Full/Empty Flags                       | Not Selected/Not Selected    |
| Programmable Full/Empty Flags                 | Not Selected/Not Selected    |
| Data Count Outputs                            | Not Selected                 |
| Handshaking                                   | Not Selected                 |
| Read Mode / Reset                             | Standard FIFO / Not Selected |
| Read Latency (From Rising Edge of Read Clock) | 1                            |

Table 3.1: Configuration settings of the FIFO IP module.

The configuration properties in figure 3.1 show all the parameters of the FIFO. The key aspect there is *clocking scheme*, where it can be seen that the clocks are independent, which satisfies the need for *33 MHz* on the input

and  $100\text{ MHz}$  on the output. Therefore, this configuration solves the issue of two clock domains. As mentioned in the analysis, it is a symmetric FIFO, therefore the *write width* and *read width* are the same, both being 8 bits.

### 3.1.3 UART Transmitter Module

Aside from the *clock* and *reset* inputs, the transmitter component has a *data* input, which is exactly 1 byte wide, and a *transmit* input. When the *transmit* input indicates high, the *data* are loaded into the component and transmitted through the *txd* output bit after bit.

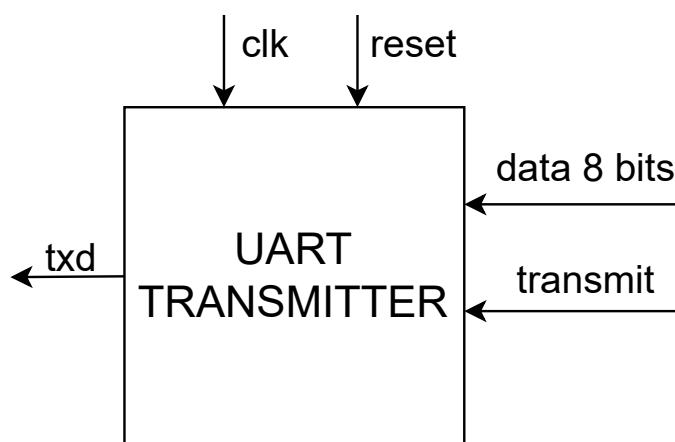


Figure 2.7: The UART transmitter outputs the data via the *txd* signal. It receives a byte of data as an input and a transmit signal to activate the transmission

The implementation of the transmitter consists of 2 parallel operation sections, one being a finite state machine managing the logic of the transmitter. The other part takes care of the transmission logic and synchronization with the serial receiver.

The FSM sets management signals for the logic section:

- *load* - the *data* input is to be loaded and transmitted
- *clear* - transmission of one byte is completed
- *shift* - transmission of one bit has been finished, next one is up

#### 3.1.3.1 Logic and Synchronization

The main purpose of the logic section is to take care of synchronization with the serial receiver. As described in the analysis, the baud rate has to be the same between the receiver and the transmitter. When implementing the

transmitter, it is necessary to know how many clock cycles are needed to transmit one bit. In other words, the *txd* output has to maintain the same value for a certain number of clock cycles to transmit the bit successfully. The amount has to be determined using this formula<sup>8</sup>:

$$ncycles = frequency_{Hz}/baudrate$$

The signals received from the FSM are processed only when the transmitter is in synchronization with the receiver (code snippet A.1). For every clock cycle, *counter* increases by one. When the value reaches the *NCYCLES* threshold, the values of other signals and registers can be changed.

The goal is to transmit a byte as quickly as possible, so the baud rate should be as high as possible. In this particular case, the baud rate is set to *115200* bits per second. It is a value supported by the Windows COM port, and when we use the formula above, the number of cycles needed to transfer one bit is *869*. To use a different baud rate, it is only needed to change the *NCYCLES* parameter.

The *shift\_reg* register stores the *data*, *transmit\_counter* stores the amount of bits already transmitted. According to the signals described above, which are indicated by the FSM, the corresponding operations are carried out. The source code of the logic and synchronization part of the *UART transmitter* module can be viewed in the appendix (A.1).

### 3.1.3.2 Transmission Finite State Machine

The state machine is fairly simple, it operates with two states - *idle* and *transmit*. When the machine is in the state *idle*, it waits for the *transmit* input to be triggered. When that happens, it sets the *load* signal up, so that the *data* input is loaded into the *shift\_reg*. After that, the state changes to *transmit*.

While in the *transmit* state, the *shift* signal is indicated. This way, each time the transmission of one bit is completed, the right-shift operation is carried out with the *shift\_reg* register. The next bit then moves to the offset 0 and is sent to the output via *txd*.

After the transmission of all bits is complete, it is desired to clear the *transmit\_counter* register and change the state back to *idle*. The source code of the FSM can be found in the appendix (A.2).

## 3.2 Connecting the Modules Together

After developing the three modules one by one separately, it is necessary to connect them together and complete the whole design. The final top module looks as shown in the diagram (3.2).

---

<sup>8</sup>The *basys3* clock frequency is 100MHz. If the baudrate is 9600 bits per second, the number of cycles is 10417.

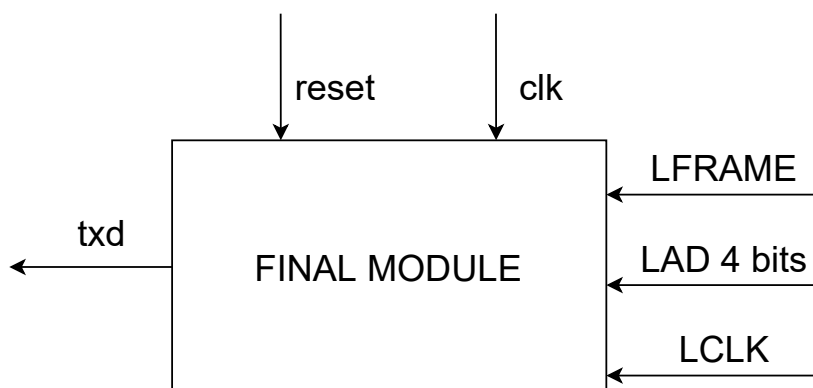


Figure 3.2: The final module has three inputs from the LPC bus, alongside with the clock signal from the FPGA board and a reset signal. It processes the input values and outputs the correct TPM data via the `txd` UART output.

For testing purposes, it was reasonable to first connect the UART transmitter with the FIFO. After making sure that this component works, it could be treated as one module and connected to the filtration FSM.

In both of these cases, some slight adjustments had to be made for the modules to function properly together to fulfill the goal functionality of the design.

### 3.2.1 Connecting the Transmitter and the FIFO

To connect these two components, it is first necessary to know how the FIFO works, more specifically, how the data are read from it, so that they are loaded into the transmitter properly.

As mentioned in the analysis section (figure 2.6), the FIFO provides a *read\_enable* input signal. To read the data, this signal must be held high for one clock cycle. In the next clock cycle, the desired byte appears on the *data* output. What is important to keep in mind is the fact that it should not be read from an empty FIFO.

Taking into account the separate UART transmitter module, there is only the *txd* output. There is no other output signal that could be used as the *read\_enable* input of the FIFO. Therefore, it is necessary to add one. The signal should be driven high when a transmission of one byte is completed. Because of the FIFO interface, this signal must be held high only for one clock cycle, otherwise it would read more than one byte from the FIFO. Moreover, it has to be synchronized in such a way that the data coming from the FIFO are already present on the input at the moment when the transmitter is in sync with the receiver. During this clock cycle, the *data* input is loaded into *shift\_reg*.

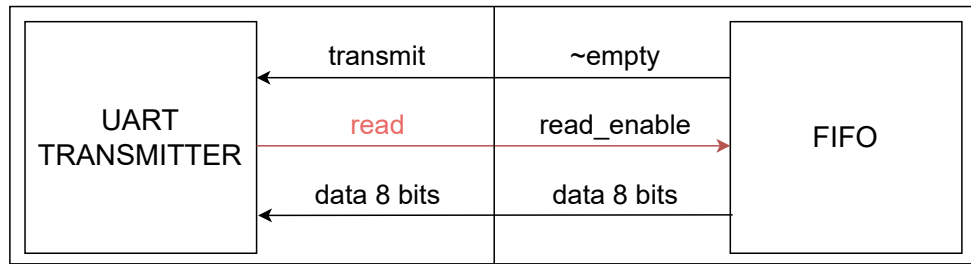


Figure 3.3: A new *read* signal is introduced in the connection of transmitter and FIFO. It is driven high one cycle before synchronization between serial receiver and transmitter occurs.

To achieve this, a *read* signal is introduced in the UART transmitter. It is driven high exactly one cycle before the synchronization occurs. It can only be done when the *load* signal is up, otherwise a transmission is still in progress.

```

counter <= counter + 1;
done <= 0;
if (counter == NCYCLES - 2) //One cycle before we indicate
begin //sync, we want to read data from FIFO
    if(load)
    begin
        read <= 1;
    end
end
end

```

Code snippet 3.1: The *read* signal is only driven high when the *load* signal is active and one cycle before synchronization occurs. This is a code snippet from the *transmitter.v* file

### 3. IMPLEMENTATION

---

Adding the *read* signal is the only adjustment that has to be made. The *transmit* input corresponds to the negation of the *empty* output of the FIFO. When the *transmit* signal is turned low, the FSM in the transmitter remains in the *idle* state and does not load any data. Therefore, if the FIFO is empty and the *empty* signal is held high, the transmitter will not read from the FIFO.

#### 3.2.2 Completing the Final Module

The final step of the FPGA design is to connect the two completed modules. The FIFO and the transmitter are already connected, so they can be treated as one module. After its connection with the Filtration FSM, the FPGA design is complete and a final module is created.

In this case, it is not important that, aside from FIFO, there is also a transmitter present in the previously created module. The input signals of the module correspond to the FIFO input signals and the Filtration FSM only needs to interact with the FIFO.

Regarding the FIFO, this time it is necessary to understand the writing process. As described earlier, the FIFO has a *data\_valid* input. When it is asserted, the value of the *data* input is loaded into the FIFO. The *data\_valid* signal has to be driven high for exactly one clock cycle to write one byte of data.

Filtration FSM already satisfies this condition. As described earlier, the data obtained during an LPC transaction are asserted at the *data* output, and the *data\_valid* signal is driven high only in the *FTAR2* state, which is one clock cycle long.

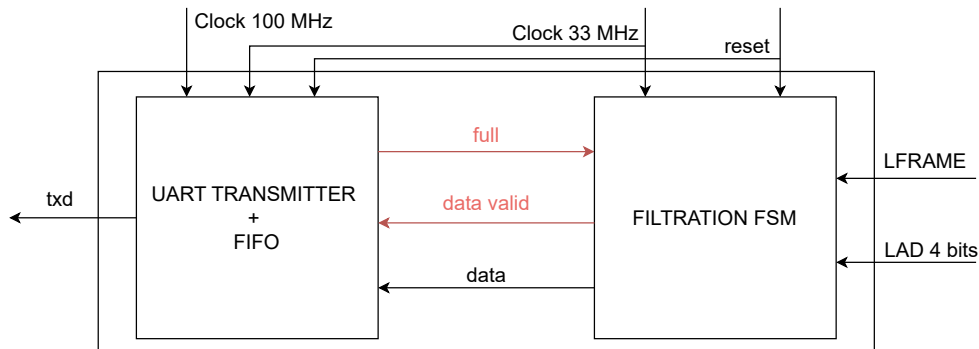


Figure 3.4: Connection between the *transmitter and FIFO* module and the *filtration FSM* creates the *final module*. One new signals are added, *full* signals to condition the *data\_valid* signal .

Just as it was necessary to prevent reading from an empty FIFO, it is also the case with writing into a full FIFO. Therefore, another small adjustment must be made. The FIFO provides a *full* output. This signal connects to a new



input in the filtration FSM. In the source code, only one condition is added. In the *FTAR2* case, the assertion of the *data\_valid* signal is conditioned with the incoming *full* signal being driven low.

```
S_FTAR2:
begin
  if (~full)
    begin
      data_valid_reg <= 1; //Added for writing to FIFO
    end
    state <= S_IDLE;
end
```

Code snippet 3.2: Code snippet from *fsm\_lpc.v* file, which contains the code of *filtration FSM*. This snippet shows the condition of *data\_valid* signal in the *FTAR2* state.

### 3.3 Processing the Serial Data

To receive the data and write them to a file or any other output in the desired way, it is necessary to write a short script. For this purpose, the Python programming language with the *serial* package is utilized. The *Serial* object has to be initialized properly with the baud rate of *115200*, one stopbit, and with the size of eight bits. Then, the script just reads the received bytes and prints them onto the console and into the text file. The complete script is listed in the appendix (A.3).



---

## Simulation & Testing

Each of the three modules described in previous sections (*transmitter*, *FIFO* and *filtration FSM*) has to be tested to make sure it works properly. At first, a module is tested separately, and after connecting it with other component, the newly created module also goes through testing. This is very important, if the testing took place after assembling more components together and some errors occurred, it would not be clear which component caused the issue.

Firstly, *Vivado* simulation is used for simple behavioral testing. After that, the design is tested on the physical board in two stages - with internally generated input and externally generated input using Arduino UNO board<sup>9</sup>.

### 4.1 Behavioral Simulation Using Vivado

The most convenient solution to test each module separately is to use the behavioral simulation feature provided by Vivado. There are multiple advantages, all signals of all used components can be viewed and inspected to a great detail. The simulation also loads very quickly. Performing the synthesis and loading the generated bitstream into a physical FPGA board takes a few minutes, whereas the behavioral simulation is completed in a matter of seconds. Thanks to this, it can even be used for debugging during the implementation.

Data for each simulation are defined in special files called *testbenches*. There exists a separate *testbench* file for the simulation of each module. In these files, it is possible to instantiate the modules and define custom input values.

---

<sup>9</sup>“Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs - ... - and turn it into an output. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language (based on Wiring), and the Arduino Software (IDE), based on Processing.” [30]. Detail information about Arduino is out of scope of this thesis, more can be learned at [www.arduino.cc](http://www.arduino.cc)

Each module needs a clock input that is normally generated by an oscillator on a physical board. For simulation purposes, the clock signal can be easily generated in *Verilog*. It is possible to set a desired frequency, as shown in the code below (code snippet 4.1). In the simulation of the *FIFO* and *filtration FSM* modules, clock signals are generated at both 100 and 33 MHz. The *transmitter* expects only one clock input, which originates from the Basys3 board, therefore its frequency is 100 MHz.

```
parameter c_CLOCK_PERIOD_NS = 10;
parameter c_CLOCK_SLOW = 30;

always
    #(c_CLOCK_PERIOD_NS/2) clock_100 <= !clock_100;
always
    #(c_CLOCK_SLOW/2) clock_33 <= !clock_33;
```

Code snippet 4.1: Code snippet used in *testbench* files to generate two clock signals.

#### 4.1.1 Simulating the Filtration Finite State Machine

To test this module, it is necessary to fabricate various LPC transactions according to the protocol described in the analysis (section 2.3.1). The situations that need to be simulated are as follows:

- A correct TPM transaction with matching addresses, both read and write.
- A TPM transaction in which the addresses do not match.
- A configuration that utilizes only the read or write branch.
- A transaction that is not related to TPM.
- A transaction with invalid TYPE value.
- A transaction where the abort mechanism is activated.

In the testbench, the module is instantiated and input and output signals are defined. It is only necessary to periodically change these signals as if they originated from an LPC bus.

For this purpose, a two-dimensional register is established. Multiple test vectors are stored there. Each test vector is 52 bits wide, which corresponds to 13 4-bit *LAD* values. Each 4-bit value represents one field of the LPC protocol.

```

for( i = 13; i > 0; i = i - 1 )
begin
  @(posedge clock);
  #5;
  if(i == 13) begin
    LFRAME = 0;
  end
  else begin
    LFRAME = 1;
  end
  LAD = test_cases[j][ (i*4 - 1) -: 4];
end
end

```

Code snippet 4.2: The code snippet shows a test step of the filtration FSM. Each iteration represents one LPC transaction field and updates the input data for the filtration FSM as set in the particular test step.

A for loop iterates through each of these test vectors. As mentioned in 3.1.1, the data on the bus are valid slightly after the rising clock edge. That is the reason for the 50 ns delay. The *LFRAME* signal is driven low at the beginning of each transaction. The behavior of the module signals can be inspected in the simulation window. After the simulation is run, a waveform window opens up as shown in the following picture. There, the behavior of the module signals can be inspected in great detail.

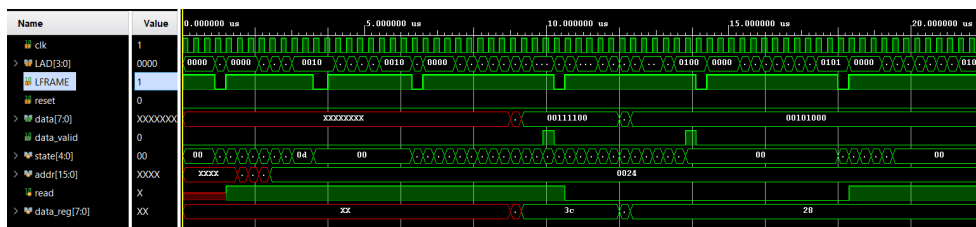


Figure 4.1: A waveform window opens after running a simulation. All the simulated signals can be inspected there.

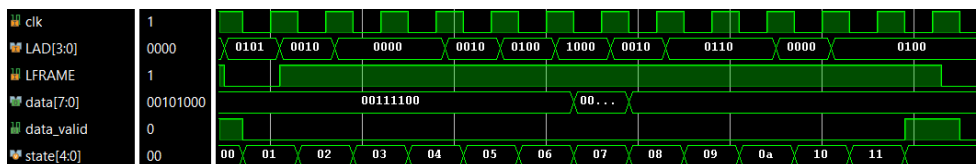


Figure 4.2: The waveform window can be zoomed in. This image shows the signals in course of one LPC transaction.

### 4.1.2 Simulation of FIFO

Simulating the single FIFO is performed mainly to confirm the anticipated behavior, with the FIFO being just a configurable module from the IP library. It is sufficient to establish two clock domains and write and read a few bytes of data. In the waveform window of the simulation, it is then possible to check whether the configuration is correct and the FIFO behaves as expected. The code sample from a testbench shows a writing and reading of 1 byte.

```
//Write
@(posedge s_Clock);
data_in = 8'b01010101;
data_valid = 1;
@(posedge s_Clock);
data_valid = 0;
//Read
@(posedge s_Clock);
read_en = 1;
@(posedge r_Clock);
read_en = 0;
```

Code snippet 4.3: Writing of one byte into a FIFO is performed by triggering the *data\_valid* signal for one clock cycle. Subsequently, the same byte is read from the FIFO by triggering the *read\_en* signal for one clock cycle.

### 4.1.3 Simulation and Testing of UART Transmitter

There are multiple factors to observe in the transmitter simulation. The main one is the correctness of the data that are being transmitted. It is visible from the waveform in the *txd* signal. However, one has to know the number of clock cycles necessary to transmit one bit. Without that, it may be difficult to distinguish particular bits that are transmitted via the *txd* output. That is also another aspect of the simulation. It is necessary to check whether a bit maintains its value for the correct number of clock cycles due to synchronization with the serial receiver.

## 4.2 Functional Testing on Basys3 Board

After the behavioral simulation of all components is complete, it is time to test the design on physical hardware. First, it is necessary to test the transmitter to ensure that the outputs of other modules that connect to the transmitter are correct. It is very important because the correct evaluation of further testing is strongly based on the output data coming via the serial line from the Basys3 board.

Only the three modules that include the *transmitter* module will be tested on physical hardware, the *transmitter* itself, the *FIFO* connected to the transmitter and the *final module*. It would be possible to generate input data into the other separate components, however, there is no easy method to evaluate the output and behavior of the modules.

In addition to writing the source code for the modules, to run the design on the board, it is necessary to define a *constraint file*. In these files, the inputs and outputs defined in the module are paired with specific ports on the FPGA board.

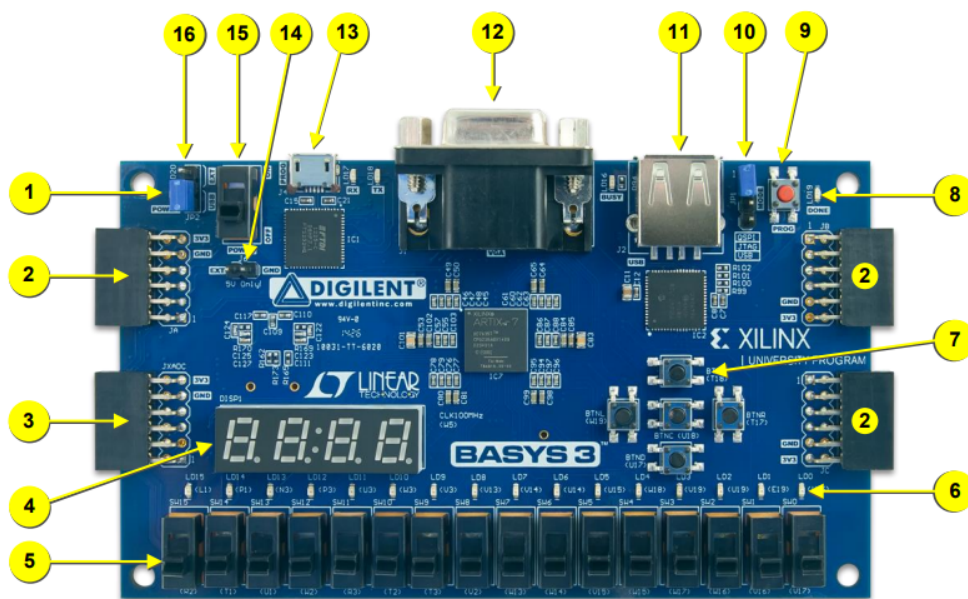


Figure 4.3: There are several ports on the Basys3 board that are used in the design. [29] The switches (number 5) are used for testing the serial line communication (4.2.1), the usb output (number 13) serves as the serial output for the *txd* signal, the pmod pins (number 2) serve as LPC bus inputs (4.3.1) and a button (number 7) is used as a *reset* input throughout multiple modules.

#### 4.2.1 Testing the Serial Line Communication

To provide input, switches and buttons (5 and 7 in the 4.3 scheme) on Basys3 are used. Each switch represents one bit of the *data* input and sets the value that will be sent through the UART to the Python script. Two buttons represent the *transmit* input and the *reset* input. The *clock* input is provided by the internal Basys3 clock. For better button functionality, a debouncing module was used.[28] Below are the constraints used for reset and transmit outputs.

```
set_property PACKAGE_PIN U18 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports reset]
set_property PACKAGE_PIN T18 [get_ports transmit]
set_property IOSTANDARD LVCMOS33 [get_ports transmit]
```

Code snippet 4.4: Example of setting constraints for ports. In this example, the inputs *reset* and *transmit* are assigned to two buttons.

After clicking the *transmit* button, a series of values presented in hexadecimal form is displayed in the terminal as output from the Python script. They are also saved to a text file. The values are correct, thus both the transmitter module and the Python script playing the role of serial receiver work correctly.

### 4.2.2 Creating a Slow Clock Module

Because the design operates with two clock domains and the external clock signal is not yet available, it is necessary to create a module that will provide the slower signal for testing purposes. The Basys3 has only one 100 MHz oscillator, so it is necessary to create a small module that will convert this signal into a lower frequency.

The slow clock input does not have to be a perfect 33 MHz signal like the LPC clock, because it is an external input and the design will adapt to the incoming frequency. The only requirement for the clock is to be slower than 100 MHz, so the FIFO does not fill up that quickly.

In the *custom\_clock* module, there is only one input, *clk*, which is the regular clock input, and one output *clk\_out* which is the slow clock. The output is driven high in one of five cycles of the input clock, resulting in a frequency of 20 MHz when used on the Basys3.

```
module custom_clock(input clk, output clk_out);
    reg [3:0] counter = 0;
    reg clk_slow = 1;
    always @(posedge clk)
    begin
        counter <= counter+1;
        if(counter == 4)
        begin
            counter <= 0;
            clk_slow = ~clk_slow;
        end
    end
    assign clk_out = clk_slow;
endmodule
```

Code snippet 4.5: Module generating a clock signal 5 times slower than the input clock signal.



### 4.2.3 Testing the FIFO and Transmitter

To test that the FIFO and transmitter connected module also works well on the board, some data writes and data reads must be performed. It is then possible to check all the transmitted data as the output of the Python script on the command line or in the file which is generated.

For the purposes of testing of this module, a data generator was created. This module contains a data register which is incremented in each clock cycle. Its maximum value is 255. Every fifth clock cycle, the *data\_valid* signal is triggered and the value in the register is therefore written into the FIFO. This continues until the FIFO is full.

```
always @(posedge clk) begin
    if(full) begin
        write <= 0;
    end
    counter <= counter+1;
    data <= data + 1;

    if(counter == 4)
    begin
        if(write) begin
            data_valid <= 1;
        end
        else begin
            data_valid <= 0;
        end
        counter <= 0;
    end else begin
        data_valid <= 0;
    end
end
```

Code snippet 4.6: Code of a module that generates sample data to test the *FIFO* and *transmitter* connected module. A counter is incremented in each cycle, and every five cycles, a *data\_valid* signal is driven high to write to *FIFO*. The writing stops once *FIFO* fills up.

The *FIFO and transmitter* module is instantiated in the *internal\_test\_fifo.v* file, together with the *FIFO data generator* and the *slow clock* module described earlier. The slow clock is utilized as the clock input of the data generator, as the data have to be written into the FIFO using the slower clock domain.

#### 4.2.4 Testing the Final Design

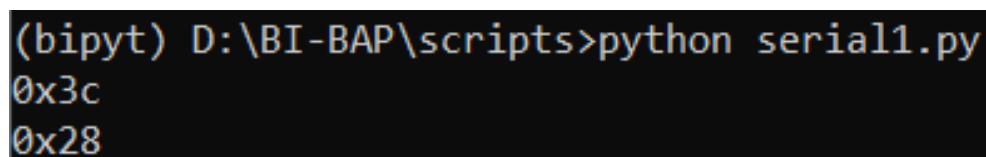
The final design on the Basys3 board is tested here. Again, three modules are instantiated in the design file. The *slow clock* are used once again to imitate the LPC clock, the *final module* is instantiated and, finally, an LPC data generator was created.

This module initializes multiple constant values, which simulate LPC behavior similarly to the final module simulation. However, for the purpose of synthesis, different approach has to be taken to set the output data in the correct order and time. The test values are again stored in the *test\_cases* register, and the for loop from the filtration FSM testbench is rewritten in a different way, using an *always* block and two counters, *counter\_cases* and *counter\_fields*, to control the state of the data presented to output.

```
always @(posedge LCLK) begin
if (counter_cases < 4) begin
if (counter_fields == 13) begin
LFRAME = 0;
end else begin
LFRAME <= 1;
end
if (counter_fields == 1) begin
counter_fields <= 13;
counter_cases <= counter_cases + 1;
end else begin
counter_fields <= counter_fields - 1;
end
LAD <= test_cases[counter_cases][counter_fields*4-1 -:4];
end
end
```

Code snippet 4.7: Similar test input generator as in code snippet 4.1.1. Each iteration represents one LPC transaction according to the test vector.

Five test transactions were run, with only two of them being valid - one read and one write. Therefore, two captured values should be seen in the output of the script, *0x3c* and *0x28*.



```
(bipynt) D:\BI-BAP\scripts>python serial1.py
0x3c
0x28
```

Figure 4.4: Output of the Python script. The data are generated by the internal test module and successfully transmitted via the serial line.

The same output can be also found in the output text file, therefore the test is successful. The final module consisting of *filtration FSM*, *FIFO* and *UART transmitter* is therefore tested while loaded onto the FPGA board together with a special module that generates input for the final module.

### 4.3 Testing the Design with Arduino

Before connecting the FPGA board to a real LPC bus, it is convenient to test not only the design itself but also the connection of the pins and their assignment to the right input. For this purpose, the Arduino UNO board is used. Its output pins are connected to the input pins of the Basys3 board with wires. Arduino imitates the output of an LPC bus, therefore the FPGA behaves exactly the same as if it were connected to a real bus.

Another advantage of using Arduino for testing is that the input values can be controlled and changed very quickly. Defining new test values directly in the Verilog code as shown in the previous section is not that convenient, mainly because the synthesis has to run again every time the data are changed. This takes a few minutes, whereas loading the code into Arduino takes just a few seconds.

#### 4.3.1 Wiring Arduino and Basys3 Together

As described in previous chapters, there are 5 inputs and 1 output in the top module. The output (*txd*) is already tested properly, so the focus is on the inputs.

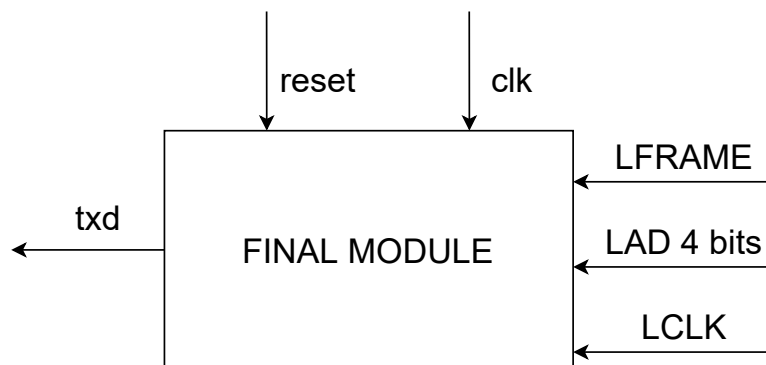


Figure 3.2: The final module has three inputs from the LPC bus, along with the clock signal from the FPGA board, and a reset signal. It processes the input values and outputs the correct TPM data via the *txd* UART output.

The *clk* input is connected to the inner 100 MHz clock of Basys3 in the same way as in the internal testing part. A button on the FPGA serves as the

#### 4. SIMULATION & TESTING

---

reset input. Therefore, there are the three main inputs left for the Arduino to provide.

The *LFRAME* and *LCLK* are only one-bit signals, whereas the *LAD* is 4 bits wide. 6 output and input wires are therefore necessary to connect the two boards.

There are 14 digital pins present on the Arduino UNO, it is necessary to define which of them will serve which input. The pins 4-7 act as the *LAD* inputs, respectively, from the least to the most significant bit. *LFRAME* input comes from the pin 8 and *LCLK* from the pin number 2.

Now, it is necessary to connect them to the Basys3 board. The Basys3 board contains 4 pmod headers, in each header there are 12 pins - 2 power supplies, 2 ground, and 8 data pins. It is possible to map each of the individual pins to an input of a module.

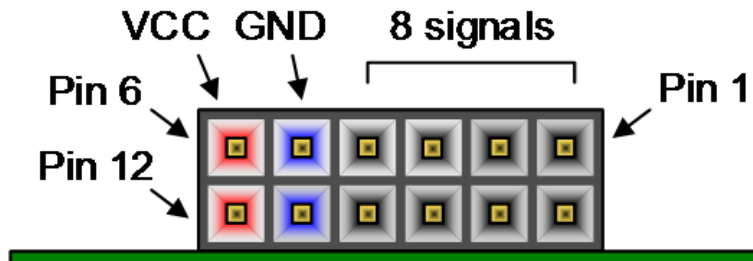


Figure 4.5: Scheme of one pmod head on Basys3 board. It contains 12 pins including 8 data pins, 2 power (3.3V) pins and 2 ground pins. [29]

Two pmod heads are utilized for connecting the inputs, *JA* - the upper left header (number 2 on the Basys3 scheme) and *JC* - the bottom right header. The *LAD* pins are connected to pins 1-4 of the *JA* header, the *LFRAME* is connected to the pin number 7. The clock is connected to pin number 3 in the header *JC*, because there are few pins capable of handling an external clock input. Moreover, it has to also be connected to the P side of the clock-capable input pin, because the clock is a single-ended input<sup>10</sup>, which took some time to figure out. Naturally, the ground pins of the two boards have to be connected. Constraints must be set to assign ports to matching inputs and outputs.

---

<sup>10</sup>This topic is out of scope of this thesis, some details can be found at [omega.com](http://omega.com) [31]

```

set_property PACKAGE_PIN J1 [get_ports LAD[0]]
set_property IOSTANDARD LVCMOS33 [get_ports LAD[0]]

set_property PACKAGE_PIN N17 [get_ports {clk_33}]
set_property IOSTANDARD LVCMOS33 [get_ports {clk_33}]

```

Code snippet 4.8: Snippet of a constraint file. The first two lines describe the assignment of the pmod pin number 1 to the least significant bit of the *LAD* input. The remaining lines assign the clock input to the clock capable pin.

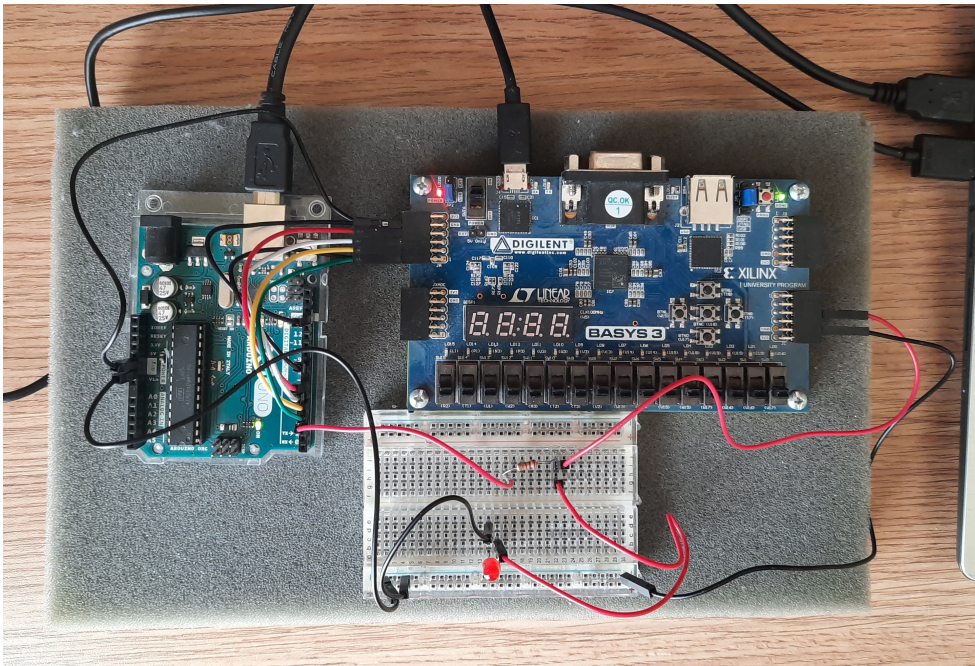


Figure 4.6: Connection of Arduino and Basys3. The Arduino pins for *LAD* and *LFRAME* outputs are connected directly to pmod pins of the Basys3 board. The *LCLK* output is connected to another pmod head containing a clock capable pin, while also connecting to an LED. The LED is present just to indicate that the clock is active.

### 4.3.2 Programming the Arduino

After connecting the Arduino and Basys3 boards, it is necessary to program the Arduino to produce the correct values similar to the LPC bus. An important topic is the clock. As described earlier, the LPC clock frequency is around 33 MHz. It is not possible to produce this frequency because the internal clock of the Arduino operates on a 16 MHz signal. Fortunately, this exact frequency is not needed. The clock from the LPC bus is an external

input, so the FPGA design behaves the same way at any frequency.

The only difference is the time it takes for the FIFO to fill up. Because the clock is slower on the Arduino than on the LPC bus, it only slows down the process and the test data size is not that large to fill the FIFO anyway.

Regarding the clock signal, another advantage of using Arduino is the ability to dynamically control the clock frequency, even slow it down to the point where it is observable which values are sent into the FPGA board in real time (for example, after setting the clock frequency to one second).

To generate the clock, an oscillator on the Arduino ATmega328P microchip was used. Multiple frequencies can be generated and for the testing purposes, 8 MHz or 1 Hz clock can be selected. While operating with the 1 Hz clock, it is possible to view the data processing slowly and, for example, set the *LAD* and *LFRAME* signals as outputs on the Basys3 LEDs. A function that reacts to the clock is established and desired test values are written there to the digital output pins on the Arduino. The source code of the clock generators and the testing function can be found in the appendix (A.4, A.5 and A.6, respectively).

---

## Forensics of Captured Data

There are various applications that can utilize the TPM chip. However, an illustrative example of such software is the previously mentioned Windows BitLocker. It is used by a very large number of users who may not even know about it because it is configured on Windows operating systems by default.

Within the process of unsealing a key, it has to be read from the TPM and travel through the LPC bus. This exact process is performed by BitLocker during the boot phase, where *volume master key (VMK)* is unsealed. While researching sources for this thesis, a blog publication was discovered that discussed this idea [37]. Together with the supervisor, it was decided to explore it and confirm that *VMK* can be indeed *sniffed*<sup>11</sup> from the LPC bus. The data sample regarded in this chapter was provided by Filip Štěpánek [24].

### 5.1 Configuration of FPGA Design

When there is a specific piece of data that is intended to be read from the LPC bus, it is useful to take advantage of the parameters provided before loading the design onto a physical board. In this case, only the *read* branch is used, as the unsealed key is read from the TPM by BitLocker. Another parameter that can be set is *tpm\_addr*, where it can be specified which address to focus on for data capture.

#### Choosing the Correct Address

As described in section 2.3.1.1, the address consists of 16 bits. That is a lot of possible addresses. To find the right one, it is necessary to examine the TPM Client Specification ([17]). There, in section 6.5.2.2.2, it is stated that “*when a command completes, the TPM puts the results into the data FIFO, which is read via the TPM\_DATA\_FIFO\_x register*”.

---

<sup>11</sup> “*Sniffing is when packets passing through a network are monitored, captured, and sometimes analyzed.*” [36]

There is no need to know how the commands are processed. It is assumed that a command for unsealing the *VMK* is invoked by BitLocker, performed by the TPM, and the key is loaded into the *TPM\_DATA\_FIFO\_x* register.

After further inspection of the specification, it is revealed that there are five of these registers, each belonging to a different locality (0-4). In this case, the data belong to locality 0, therefore, to the *TPM\_DATA\_FIFO\_0* register. As described in section 3.2 of the specification, locality 0 regards the static root of trust and measurement (SRTM), which is used during the boot phase before the OS is loaded. That is when the *VMK* unsealing takes place.

According to Table 19 in the specification, *TPM\_DATA\_FIFO\_0* can be accessed in the address range of 0027h to 0024h (h meaning hexadecimal). As written in the note, however, these addresses are aliased into one in the TPM, therefore it is enough to read only the **0024h** address.

When configuring the FPGA design, the *tpm\_addr* parameter is therefore set to **0024h**.

## 5.2 Searching for the VMK

After setting the correct parameters, loading the design onto a physical board, connecting it to the LPC bus, and capturing the data during the boot phase, it is necessary to inspect the text file that contains the captured data and find *volume master key*. To achieve that, it is necessary to know what to look for, thus look into the structure of the *VMK*.

### 5.2.1 Structure of the VMK on the LPC Bus

The Bitlocker drive encryption (BDE) format specification is publicly available on Github. It was created by Joachim Metz by combining public work on the specification and analyzing test data. [22]

The data that are sent from the TPM in the process of unsealing the *VMK* resemble the full volume encryption (FVM) key, with standard FVE metadata entry header. [23], [22] The format is described in the BDE format specification, sections 5.4 and 5.3, respectively.

Knowing this, it is necessary to create a byte sequence to look for in the captured data. According to the specification, the key structure should look as indicated in the table below. The expected values are written in hexadecimal form.



|                 | Size | Expected value  | Description        |
|-----------------|------|-----------------|--------------------|
| Metadata header | 2    | 00 2C           | Data size in bytes |
|                 | 2    | xx xx           | Entry type         |
|                 | 2    | 00 01           | Value type         |
|                 | 2    | xx xx           | Version            |
| Key             | 4    | 00 00 20 0[0-5] | Encryption method  |
|                 | 32   | ...             | Key value          |

Table 5.1: This table shows the expected structure of the *VMK* between captured data. The following list describes the individual fields of the metadata header and the key.

Description of individual fields:

- **Data size** - Contains the sum of sizes of all fields, which is 44 decimal or 2C hexadecimal.
- **Entry type** - It is not clear which data should be in this field, other fields will be used to determine the key.
- **Value type** - Here, according to section 5.3.2 in the BSD specification the value should be 1, as the key is already decrypted.
- **Version** - In this field, values may vary.
- **Encryption method** - The VMK uses AES-CCM 256 bit encryption, therefore, according to section 5.2.1, the value can range from 20 00 to 20 05 hexadecimal.
- **Key** - The *VMK* itself is located in this field.

It is important to note that the header data are stored in little-endian, therefore for each field, the least significant byte is transferred first through the LPC bus. Therefore, the desired byte sequence of the header should look like this: 2C 00 xx xx 01 00 xx xx 0[0-5] 20 00 00, where *x* stands for an unknown value.

### 5.2.2 Finding the VMK Between Captured Data

While searching for the VMK header, the 0x2C data size served as a point of reference. There were 23 0x2C entries in the captured data chunk, so it was possible to go through them manually. The majority of these entries were surrounded by values that did not match the VMK header at all. Finally, only one entry corresponded to the header.

The hexadecimal VMK value found between the data sample is:

```
f5 c0 83 bd f9 61 cf a0 9a ea a9 6d a3 e2 7a 1d 45 78 71 e0 1e 99 72 ab d7
3b d3 a3 e5 d7 30 2b
```

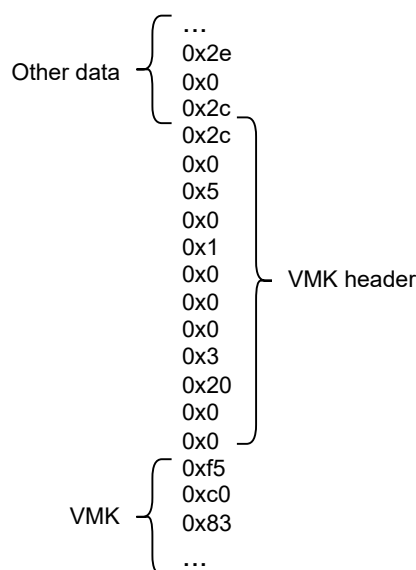


Figure 5.1: The VMK was found between the captured data from the LPC bus. The header corresponds with the anticipated structure.

### 5.3 Conclusion of the Forensic Analysis

Considering all the sections above, it is possible to perform an *evil maid*<sup>12</sup> attack. After extracting *volume master key*, it is possible to decrypt and mount the disk. A tool called Dislocker [26] can be utilized, it supports drive decryption after providing *VMK*, so it is not necessary to decrypt *FVEK* manually.

It is important to mention that certain conditions must be met for the attack to work. First, the targeted Windows machine must operate with a discrete TPM chip that communicates with the motherboard using the LPC bus.

Second, the Bitlocker has to be configured in the TPM only authentication mode. The chances are fairly high for this to be the case, as it is the default Bitlocker configuration.

Most importantly, the adversary has to get hold of the device so that they can probe it in somewhat laboratory conditions. In terms of tools, it is only necessary to possess an FPGA and solder some wires.

---

<sup>12</sup>“An evil maid attack is an act of hacking a device through physical access. The name refers to a scenario where a hotel employee compromises a laptop, smartphone, or tablet left in a room.”[25]

## 5.4 Mitigation

There are multiple methods to prevent this attack. The simplest one is to change the Bitlocker configuration and use the PIN or the startup key authentication mode alongside the TPM. As mentioned earlier, the *volume master key* is not released to the bus before entering the second authentication factor.

While this thesis was being written, an article addressing the issue with this attack was published. It reported about new security features implemented by Microsoft for Windows 11. One of the features is the usage of a new security chip called Pluton, which implements the TPM functionality directly into the central processing unit. It mitigates this exact type of attack, as there is no bus connection between the TPM and the motherboard. [27]



---

# Conclusion

The goals of the thesis were to provide an introduction to the trusted platform module, implement an FPGA design to capture only TPM related data from the LPC bus, and analyze these data.

In the first part of the thesis, important theoretical information about the functionality and design of the trusted platform module was presented, as well as some basic information about the LPC bus and BitLocker. In the analysis part, the modules necessary to create the device were identified. The UART interface and protocol was described and a serial line transmitter was implemented. Moreover, the LPC protocol specification was examined and reflected in the design of a finite-state machine with the purpose to obtain only TPM related data from the LPC bus.

Multiple problems were encountered during the implementation part. Even before starting to work on the implementation, the setup of the development environment took a lot of time and effort. Another challenge that came was to cross the two clock domains, which could be solved using FIFO. Connecting the modules was also interesting, the intention of designing the modules separately and connecting them afterwards was only partially successful, as it was necessary to perform small, yet noticeable adjustments in the modules.

The FPGA design was first tested only with the help of the Vivado design suite and its simulation feature. This was perfect for debugging. Another challenge presented was to test the FPGA design on a physical board even before having access to a real LPC bus. For this purpose, Arduino was used to simulate the expected behavior of the data on the LPC bus. It had another advantage, being that particular components of the final module could be tested separately before completing and testing the final module itself. The idea of using Arduino to simulate external inputs to test FPGA designs has potential to be further explored and developed.

Upon researching the sources, it was discovered that, under certain conditions, it could be possible to perform an *evil maid* attack with the developed FPGA design, as it is possible to extract BitLocker *volume master key* from

## CONCLUSION

---

the data captured from the LPC bus. With this key, it is possible to decrypt the BitLocker-protected disk.

Together with the supervisor, it was decided to explore this idea and confirm that the attack could be carried out, which was successful. The TPM-related data captured from the LPC bus were analyzed and *volume master key* was found. Mitigation recommendations are also mentioned in the thesis.

---

# Bibliography

- [1] Arthur, Will; Challener, David. *A Practical Guide to TPM 2.0*. Berlin: Springer, 2015. 978-1-4302-6583-2.
- [2] SIMPSON, Daniel et al. *Trusted Platform Module (TPM) fundamentals (Windows) - Windows security — Microsoft docs* [online]. 2021. [Accessed 20 February 2022]. Available from: <https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/tpm-fundamentals>
- [3] SIMPSON, Daniel et al. *Understanding PCR banks on TPM 2.0 devices (Windows) - Windows security — Microsoft docs* [online]. 2021. [Accessed 20 February 2022]. Available from: <https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/switch-pcr-banks-on-tpm-2-0-devices>
- [4] SIMPSON, Daniel et al. *Trusted Platform Module Technology Overview (Windows) - Windows security — Microsoft docs* [online]. 2022. [Accessed 20 February 2022]. Available from: <https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview>
- [5] TRUSTED COMPUTING GROUP. *Trusted Platform Module Library Part 1: Architecture* [online]. 2019. [Accessed 24 February 2022]. Available from: [http://trustedcomputinggroup.org/wp-content/uploads/TCG\\_TPM2\\_r1p59\\_Part1\\_Architecture\\_pub.pdf](http://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf)
- [6] "TPM based attestation - how can we use it for good?" - Matthew Garrett (LCA 2020) [online]. Linux.conf.au, 17 January 2020. [Accessed 3 March 2022]. Available from: <https://www.youtube.com/watch?v=FobfM9S9xSI&t=540s>
- [7] TOMLINSON, Allan. *Introduction to the TPM* [online]. 2007. [Accessed 3 March 2022]. Available from: [https://www.researchgate.net/publication/227039163\\_Introduction\\_to\\_the\\_TPM](https://www.researchgate.net/publication/227039163_Introduction_to_the_TPM)

## BIBLIOGRAPHY

---

- [8] OUTLAW, Robert. *TpmAttestation interface*. *Microsoft Docs* [online]. 2022. [Accessed 3 March 2022]. Available from: <https://docs.microsoft.com/en-us/javascript/api/azure-iot-provisioning-service/tpmattestation?view=azure-node-latest>
- [9] BONEH, Dan et al. *TCG: Trusted Computing Architecture* [online]. 2010. [Accessed 3 March 2022]. Available from: <https://crypto.stanford.edu/cs155old/cs155-spring11/lectures/08-TCG.pdf>
- [10] KAMAL, Nandhitha. *How to TPM - Part 2 : TPM Software Stack* [online]. 2020. [Accessed 6 March 2022]. Available from: <https://dev.to/nandhithakamal/how-to-tpm-part-2-55ao>
- [11] TRUSTED COMPUTING GROUP. *Trusted Platform Module Library Part 1: Architecture* [online]. 2020. [Accessed 6 March 2022]. Available from: [https://trustedcomputinggroup.org/wp-content/uploads/TCG\\_TSS\\_TCTI\\_v1p0\\_r18\\_pub.pdf](https://trustedcomputinggroup.org/wp-content/uploads/TCG_TSS_TCTI_v1p0_r18_pub.pdf)
- [12] HOPKINS, Jessica. *What is the eSPI Protocol and How Does it Improve Upon LPC? - Total Phase Blog* [online]. 2021. [Accessed 10 March 2022]. Available from: <https://www.totalphase.com/blog/2021/09/what-is-the-espi-protocol-and-how-does-it-improve-upon-lpc/>
- [13] SUN MICROSYSTEMS. *Sun Netra™ CP3250 Blade Server User's Guide* [online]. 2009. [Accessed 10 March 2022]. Available from: <https://docs.oracle.com/cd/E19233-01/820-5195-11/820-5195-11.pdf>
- [14] TEXTRONIX. *Logic Analyzer Fundamentals* [online]. 2010. [Accessed 15 March 2022]. Available from: <https://assets.testequity.com/te1/Documents/pdf/logic-analyzer-fundamentals.pdf>
- [15] TOUGER, Evan. *What is an FPGA and why is it a big deal? Prowess Consulting* [online]. 2018. [Accessed 15 March 2022]. Available from: <https://www.prowesscorp.com/what-is-fpga/>
- [16] BOBROWICZ, Sam. *Basys 3 reference*. Basys 3 Reference - Digilent Reference [online]. 2022. [Accessed 15 March 2022]. Available from: <https://digilent.com/reference/basys3/refmanual>
- [17] TRUSTED COMPUTING GROUP. *TCG PC Client Platform TPM Profile Specification for TPM 2.0* [online]. 2020. [Accessed 29 March 2022]. Available from: [https://trustedcomputinggroup.org/wp-content/uploads/PC-Client-Specific-Platform-TPM-Profile-for-TPM-2p0-v1p05p\\_r14\\_pub.pdf](https://trustedcomputinggroup.org/wp-content/uploads/PC-Client-Specific-Platform-TPM-Profile-for-TPM-2p0-v1p05p_r14_pub.pdf)
- [18] INTEL. *Intel® Chipsets Low Pin Count Interface Specification* [online]. 2002. [Accessed 29 March 2022]. Available from:



- <https://www.intel.com/content/dam/www/program/design/us/en/documents/low-pin-count-interface-specification.pdf>
- [19] SIMPSON, Daniel et al. *Bitlocker (Windows 10) - Windows security — Microsoft docs* [online]. 2021. [Accessed 10 April 2022]. Available from: <https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview>
- [20] MICROSOFT. *Bitlocker Drive Encryption Technical Overview — Microsoft docs* [online]. 2012. [Accessed 10 April 2022]. Available from: [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-r2-and-2008/cc732774\(v=ws.10\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-r2-and-2008/cc732774(v=ws.10))
- [21] SIMPSON, Daniel et al. *Bitlocker Key Management FAQ (Windows 10) - Windows security — Microsoft docs* [online]. 2021. [Accessed 10 April 2022]. Available from: <https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-key-management-faq>
- [22] METZ, Joachim et al. *libbde/BitLocker Drive Encryption (BDE) format.asciidoc at main · libyal/libbde · GitHub* [online]. 2022. [Accessed 17 April 2022]. Available from: [https://github.com/libyal/libbde/blob/main/documentation/BitLocker%20Drive%20Encryption%20\(BDE\)%20format.asciidoc](https://github.com/libyal/libbde/blob/main/documentation/BitLocker%20Drive%20Encryption%20(BDE)%20format.asciidoc)
- [23] BASU ROY, Joymalya. *Bitlocker Unlocked With Joy - Behind The Scenes Windows 10 - Part 1 HTMD Blog* [online]. 2020. [Accessed 18 April 2022]. Available from: <https://www.anoopcnaair.com/bitlocker-unlocked-behind-the-scenes-windows-10/>
- [24] ŠTĚPÁNEK, Filip. *Zachycená data* [email]. Message to: mandima1@fit.cvut.cz. 8 April 2022. [cit. 19 April 2022]. Personal communication.
- [25] KASPERSKY. *What is an evil maid attack? Kaspersky IT Encyclopedia* [online]. 2022. [Accessed 19 April 2022]. Available from: <https://encyclopedia.kaspersky.com/glossary/evil-maid/>
- [26] AORIMN. *GitHub - Aorimn/dislocker: FUSE driver to read/write Windows' BitLocker-ed volumes under Linux / Mac OSX* [online]. 2022. [Accessed 19 April 2022]. Available from: <https://github.com/Aorimn/dislocker>
- [27] VIJAYAN, Jai. *Microsoft Details New Security Features for Windows 11* [online]. April 06 2022. [Accessed 19 April 2022]. Available from: <https://www.darkreading.com/remote-workforce/microsoft-details-new-security-features-for-windows-11>

- [28] WONGLIK, Alex. *Debonucing Button on Basys 3, Xilinx FPGA Development Board : 6 Steps (with Pictures) - Instructables* [online]. 2017. [Accessed 24 April 2022]. Available from: <https://www.instructables.com/Debonucing-Button-on-Basys-3-Xilinx-FPGA-Developme/>
- [29] DIGILENT. *Basys 3™ FPGA Board Reference Manual* [online]. 2017. [Accessed 24 April 2022]. Available also from: [https://hwlab.fit.cvut.cz/\\_media/pripravky/fpga/basys3/basys3\\_rm.pdf](https://hwlab.fit.cvut.cz/_media/pripravky/fpga/basys3/basys3_rm.pdf)
- [30] ARDUINO. *What is Arduino?* [online]. 2017. [Accessed 25 April 2022]. Available from: <https://www.arduino.cc/en/Guide/Introduction>
- [31] OMEGA. *Differential Signal Vs Single-Ended Inputs* [online]. 2019. [Accessed 25 April 2022]. Available from: <https://www.omega.com/en-us/resources/differential-or-single-ended>
- [32] AMANDAGHASSEI. *Arduino Timer Interrupts* [online]. 2021. [Accessed 25 April 2022]. Available from: <https://www.instructables.com/Arduino-Timer-Interrupts/>.
- [33] GAMMON, Nick. *Using an Arduino as a crystal oscillator* [online]. 2016. [Accessed 25 April 2022]. Available from: <https://arduino.stackexchange.com/questions/30695/using-an-arduino-as-a-crystal-oscillator>
- [34] LUTKEVICH, Ben *cryptographic nonce* [online]. 2021. [Accessed 28 April 2022]. Available from: <https://www.techtarget.com/searchsecurity/definition/nonce>
- [35] WAGNER, Bill et al. *Cryptographic signatures — Microsoft Docs* [online]. 2021. [Accessed 29 April 2022]. Available from: <https://docs.microsoft.com/en-us/dotnet/standard/security/>
- [36] NAKUTAVIČIŪTĖ, Jomilė. *What is a sniffing attack? — NordVPN* [online]. 2021. [Accessed 6 May 2022]. Available from: <https://nordvpn.com/blog/sniffing-attack/>
- [37] ANDZAKOVIC, Denis. *EXTRACTING BITLOCKER KEYS FROM A TPM* [online]. 2019. [Accessed 3 March 2022]. Available from: <https://pulsesecurity.co.nz/articles/TPM-sniffing>
- [38] LEWIS, James. *Logic Analyzer Tutorial and Introduction - Bald Engineer* [online]. 2016. [Accessed 7 May 2022]. Available from: <https://www.baldengineer.com/logic-analyzer-tutorial-introduction.html>
- [39] DEVADAS, Srinivas et al. *Logic synthesis in a nutshell* [online]. 2009. [Accessed 7 May 2022]. Available from: <https://www.sciencedirect.com/science/article/pii/B9780123743640500138>

---

## Source Code Fragments

```
counter <= counter + 1;
read <= 0;
if (counter == NCYCLES - 2) //One cycle before we indicate
begin //sync, we want to read data from FIFO
    if(load)
    begin
        read <= 1;
    end
end
if (counter >= NCYCLES) //If the counter indicates sync,
begin //conditions change according to the state machine
    counter <= 0;
    state <= nextstate;
    if(load)
    begin
        shift_reg <= {1'b1, data, 1'b0};
    end
    if(clear)
        transmit_counter <= 0;

    if(shift)
    begin
        shift_reg <= shift_reg >> 1;
        transmit_counter <= transmit_counter + 1;
    end
end
```

Code snippet A.1: Code fragment of the logic section of the transmitter. It includes the synchronization of the baud rate and value changes conditioned by the FSM.

## A. SOURCE CODE FRAGMENTS

---

```
case(state)
0: begin //Idle state
    if(transmit) //When transmit input is detected
    begin
        load <= 1; // indicate loading data into shift_reg
        shift <= 0;
        clear <= 0;
        txd <= 1;
        nextstate <= 1;
    end
    else begin //Not transmitting data
        load <= 0;
        shift <= 0;
        clear <= 0;
        txd <= 1;
        nextstate <=0;
    end
end
1:begin //Transmit state
    if(transmit_counter < 10) //If data bits are remaining
    begin // 10 => to transfer one byte, start and stop
        load <= 0; // bits have to be aded
        shift <= 1; //indicate shifting of shift register
        clear <= 0; //to transfer next bit
        txd <= shift_reg[0]; //Send lsb to output
        nextstate <= 1;
    end
    else begin //when all bits are sent,
        load <= 0; //stop transmitting data
        shift <= 0;
        clear <= 1;
        txd <= 1; //indicate that no data
        nextstate <= 0; //is being transferred
    end
end
end
```

Code snippet A.2: Code fragment of the finite state machine in the *UART transmitter* module. When the FSM is in the idle state, it is waiting for the transmit input to be triggered. When that happens, it sets the load signal up, so that the data input is loaded into the shift reg. After that, the state changes to transmit and sets the registers for the logic section accordingly.

---

```
import serial
```

```
s = serial.Serial( \\
    'COM4', \\
    115200, \\
    stopbits=serial.STOPBITS.ONE, \\
    bytesize=serial.EIGHTBITS, \\
    timeout=2)
```

```
f = open("result.txt", "w")
```

```
while 1:
```

```
    if s.in_waiting > 0:    # buffer contains more than
        str = s.read(1)[0] # one character
        print(hex(str) + ' ')
        f.write(hex(str) + '\n')
```

Code snippet A.3: Code of *serial\_receiver.py* script, which handles data incoming from the FPGA board via the serial line. The correct baud rate of 11500 is set and the serial port where the Basys3 board is connected is the COM4 port. The received data is then printed to the output and saved into a text file.

```
void oneHz(){
    TCCR1A = 0; // set entire TCCR1A register to 0
    TCCR1B = 0; // same for TCCR1B
    TCNT1  = 0; // initialize counter value to 0
    OCR1A  = 15624; // =(16*10^6) / (1*1024)-1 (must be < 65536)
    TCCR1B |= (1 << WGM12); // turn on CTC mode
    TCCR1B |= (1 << CS12) | (1 << CS10);
    TIMSK1 |= (1 << OCIE1A); // timer compare interrupt
}
```

Code snippet A.4: This code fragment generates clock with frequency of 1 Hz in Arduino UNO. [32]

```
void eightMHz(){
    TCCR1A = bit (COM1A0); // toggle OC1A on Compare Match
    TCCR1B = bit (WGM12) | bit (CS10); // CTC, no prescaling
    OCR1A  = 0; // output every cycle
    TIMSK1 |= (1 << OCIE1A);
}
```

Code snippet A.5: This code fragment generates clock with frequency of 8 MHz in Arduino UNO. [33]

## A. SOURCE CODE FRAGMENTS

---

```
ISR(TIMER1_COMPA_vect){
    if (state){
        digitalWrite(LCLK,HIGH); //Drive LCLK HIGH
        //Change LPC values
        if (firstCycle == false){
            if(fieldIndex == 0){
                LFRAMELow();
            } else {
                LFRAMEHigh();
            }
            writeLAD( testData[dataIndex][fieldIndex] );
            fieldIndex = getNextField(
                testData[dataIndex],
                fieldIndex);

            if(fieldIndex == 0 && dataIndex < dataCount - 1){
                dataIndex++;
            }
        }
        firstCycle = false;
        state = 0;
    }else{
        //LCLK is LOW
        digitalWrite(LCLK,LOW);
        state = 1;
    }
}
```

Code snippet A.6: This code fragment shows the function which react to clock change. Each time, it changes the clock output signal value and if the clock is being driven high, it also changes the output values accordingly to the test cases. [33]

---

## Acronyms

|             |                                   |
|-------------|-----------------------------------|
| <b>AES</b>  | Advanced encryption standard      |
| <b>AK</b>   | Attestation key                   |
| <b>API</b>  | Application programming interface |
| <b>BIOS</b> | Basic input-output system         |
| <b>CPU</b>  | Central processing unit           |
| <b>DMA</b>  | Direct memory access              |
| <b>ECC</b>  | Elliptic-curve cryptography       |
| <b>EK</b>   | Endorsement key                   |
| <b>HDL</b>  | Hardware description language     |
| <b>FPGA</b> | Field-programmable gate array     |
| <b>FSM</b>  | Finite-state machine              |
| <b>FVEK</b> | Full volume encryption key        |
| <b>IP</b>   | Intellectual property             |
| <b>ISA</b>  | Industry standard architecture    |
| <b>LPC</b>  | Low pin count                     |
| <b>OS</b>   | Operating system                  |
| <b>PCR</b>  | Platform configuration register   |
| <b>RNG</b>  | Random number generator           |
| <b>RSA</b>  | Rivest, Shamir, Adleman           |

## B. ACRONYMS

---

**SPI** Serial peripheral interface

**SRK** Storage root key

**TPM** Trusted platform module

**UART** Universal asynchronous receiver-transmitter

**VMK** Volume master key



---

## Contents of Enclosed SD Card

```
readme.txt ..... the file with contents description
src ..... the directory of source codes
├── thesis ..... the directory of LATEX source codes of the thesis
├── tpm_lpc_analyzer ..... Vivado project with implementation sources
├── arduino_tester.ino ..... source code of Arduino tester
├── captured_data.txt ..... text file with captured TPM data
├── serial1.py ..... source code of python serial line script
text ..... the thesis text directory
├── thesis.pdf ..... the thesis text in PDF format
```