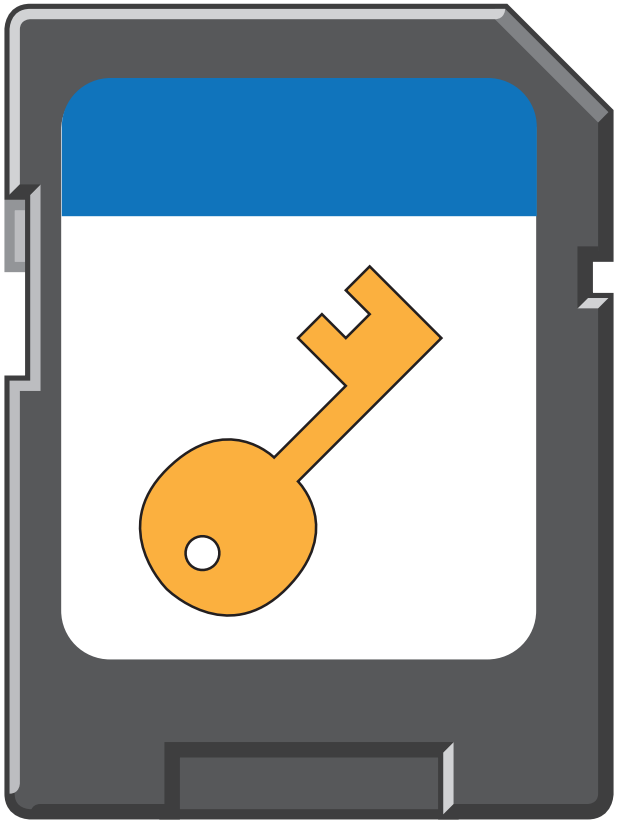
CHALMERS UNIVERSITY OF GOTHENBURG



A VHDL Architecture for Auto Encrypting SD Cards

Master's thesis in Embedded Electronic System Design

ALEXANDER DAVIDSSON
TORBJÖRN RASMUSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2016

MASTER'S THESIS IN EMBEDDED ELECTRONIC SYSTEM DESIGN

A VHDL Architecture for Auto Encrypting SD Cards

ALEXANDER DAVIDSSON
TORBJÖRN RASMUSSON

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2016

A VHDL Architecture for Auto Encrypting SD Cards
ALEXANDER DAVIDSSON
TORBJÖRN RASMUSSON

ISSN 1652-8557
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden 2016

A VHDL Architecture for Auto Encrypting SD Cards
Master's thesis in Embedded Electronic System Design
ALEXANDER DAVIDSSON
TORBJÖRN RASMUSSON
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

This thesis details the design of an encrypted SD-card adapter for journalists to be used in destabilized areas. The SD-card adapter should be designed to protect the journalist's photographs while allowing previewing of the photograph until the SD-card is powered down. The SD-card adapter is designed to appear as a generic SD-card adapter with the exception that it encrypts the data and hides any changes in the file system. The design is based around a FPGA with use of a publicly available IP-core for encryption. The SD-card adapter uses both symmetrical and asymmetrical encryption for protection of individual files. A soft-core is used for generating encryption keys, asymmetrical encryption and general management. Through use of software based emulation, the concept was proven feasible.

Keywords: FPGA, VHDL, Encryption, ChaCha

Acknowledgements

We would like to give our thanks to our supervisors Sally McKee and Fredrik Strömberg for their guidance. We would also like to give our thanks to Joachim Strömbergson for his help with the encryption. We would like to give our thanks to Sebastian Sahlin and Vanessa Vannas for their help with proofreading.

Finally we would like to give a thank you to E-sektionens Teletekniska Avdelning for all the good times, even though they might have delayed our graduation.

Nomenclature

Boot	The process of loading in software to run an operating system.
Bootable	A device that is bootable means that it contains software to be loaded during the boot.
Emulation	Emulation is a simulation where the hardware platform is simulated rather than the behavior of the system. An emulator is designed to ran the software intended to be run on the hardware.
FAT	Means File Allocation Table is a table over all of the clusters in the file system. It's also the name of the file system. In this text will FAT be used to refer to the file system and File Allocation Table to refer to the structure.
Cipher	Is a mathematical algorithm that with the help of a key makes data incomprehensible to someone who does not have access to the right key.
IV	This stands for initialization vector and is a changing vector used together with the key and the cipher to make sure that two blocks of data never get encrypted with the same exact key. This is to prevent a cryptographic attack called a two time pad.
FPGA	Means field-programmable gate array and is a reconfigurable hardware that uses look up tables to build logic gates.
SD	Stands for Secure Digital and is a memory card format.

Contents

1 Introduction

Society's reliance on digital storage media gives users easy access to personal data. A downside with this technology is the lack of a secure way to handle data that is sensitive in nature. Most modern equipment such as cameras will store an image onto a storage media such as SD-cards without any means of protection. This could be a problem, especially for journalists, who are likely to be subjected to threats while working in destabilized regions or dictatorships, where taken images could endanger both the journalist and their source.

One solution to this problem would be if the pictures were to be encrypted on the way to the SD-card, which is what this thesis will address. The reason for the project being initiated now is because the technology of field-programmable gate arrays (FPGAs) has reached a point where the amount of logic in the smaller chips makes this project feasible. If the final design is too big to fit inside the target FPGA, there is a possibility that future generations of FPGAs will contain enough logic to fit the design in an SD-package.

1.1 Prestudy

Before the thesis began the initiating party performed a pre-study [1] of viable options to solve the problem mentioned in the introduction. There were three solutions considered. The first solution was to use a SOC (System On Chip) that transferred the data to an external unit for encryption. It was decided that for fast data transfers this was not an appropriate solution due to power and heat issues. The second solution would be to use a Micro controller Unit/Central Processing Unit (MCU/CPU) to handle the encryption. The problem with this solution was that because of the bandwidth requirements there was a need to have a system clock with a minimal frequency of 357 MHz [1]. This would lead to a level of power consumption that would be outside of the specification. The final option considered was to use an FPGA to make logic encryption. This was considered the most viable solution due to the speed requirements [1].

The solution that we decided to proceed with was using an FPGA with hardware encryption and a soft-core for book keeping. This solution will give two benefits. Firstly we will use hardware because of its ability to handle high speed communication and encryption of the data. Secondly we will have a soft-core that can work as a book keeper and track where the hidden files are stored so that files from a previous session are not overwritten.

1.2 Requirements

The SD-card is intended to be used by journalists for both photography and video. The requirements are listed below:

- The complete system should fit inside a FPGA.
- The encryption should use a public and private key scheme for protection.
- The transfer speed should be at least 10 MB/s.
- The device should hide encrypted files from previous sessions.
- Power consumption should be within the SD-card specifications.

1.3 Limitations

Our goal with this thesis is to develop a proof-of-concept and not a final product. We will not supply a complete PCB design nor an end user desktop interface, which would be required for an end product. There will not be any complete software tools developed, only tools with the required features for the progression of the project. The thesis will not include the evaluation or implementation of different encryption schemes, which will be provided by an external encryption expert.

2 Background

This chapter sums up the background information a reader needs to have in order to understand the thesis. We will give a short background to SD-card protocol followed by some encryption theory. We will sum up with an explanation of storage and file systems.

2.1 SD-card protocol

In most cases when sending and receiving data to and from a SD-card a native protocol[2] is used. To initiate and communicate with the SD-card on the other hand a Serial Peripheral Interface (SPI) bus is used. The SD-card also supports using the SPI bus for data transfer. The drawback with this is that the transfer speed is slower. In cases where the performance requirements are not very high, the SPI protocol is used because it is an open protocol and license free.

2.2 Encryption

In this section we are going to give a brief background to the field of encryption. First we are going to discuss the two main encryption methods, symmetrical and asymmetrical. We will also discuss a method called Diffie Hellman key exchange. Then we will finish with a description of two different implementation methods the first one being stream cipher and the other one being block cipher.

Encryption is a method to make data unreadable to a person that does not possess the right key. To encrypt data we need a cipher and a key, The cipher is in most cases a mathematical formula that is supposed to be fast to compute but hard to reverse engineer. The key is one of the inputs to the mathematical formula the other one being the data that we want to encrypt.

2.2.1 Symmetrical encryption

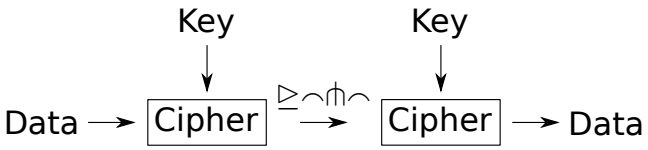


Figure 2.1: Symmetrical encryption uses the same key fore encryption and decryption.

Symmetrical encryption is a method where the encryption and decryption use the same key see Figure 2.1. There are several benefits of using this, one of them being that this method is less computation heavy, hence faster than the asymmetrical encryption [3]. The symmetrical encryption holds a high security level if used in the correct way. It is estimated to take several decades to decipher data with a 256 bit key using brute-force attack, assuming that the key is chosen in a correct way [3].

Disadvantages with symmetrical encryption include that both the receiver and sender need to have the key which means you need to share the key and hence it can be intercepted. There is another issue, if a third part gets hold of the key they will be able to decrypt data in both directions.

2.2.2 Asymmetrical encryption (public key encryption)

The basics of public key encryption is that instead of having one key that can both encrypt and decrypt we have two keys; one for encryption and one for decryption see Figure 2.2. These encryption methods are based on mathematical problems that are unpractical to solve without the private key. The benefit whit asymmetrical

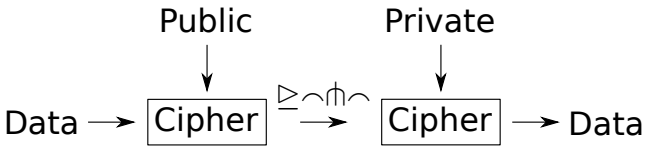


Figure 2.2: Asymmetric encryption uses one key to decrypt and one to decrypt.

encryption is that we do not need to share the unlocking key, which makes intercepting the key impossible. This however relies on heavy calculations which makes it slower than symmetrical encryption.

2.2.3 Diffie Hellman key exchange

This is a method of transferring a key without being intercepted [3]. We will describe this in the classical fashion with colors instead of with numbers. Imagine Alice and Bob wants to send messages between each other but they are concerned that Eve will intercept their messages. They both agree to a common color and a private color. They both mix their private color with the common one and sends it to the other person. The final step is to add their private color to the received mixture. This results in the same color. Now both Alice and Bob have the same color a so-called "shared secret". This secret can now be used to encrypt any messages sent between them and without anyone of the secret colors Eve can not figure out the shared secret. In real applications the colors are numbers and the mixing is a mathematical function that is difficult to reverse.

2.2.4 Stream cipher

Stream cipher encrypts in a serial fashion one bit at a time. This is done simply by adding the bit that one wants to encrypt together with the key bit and then take modulus 2 of the result, see Equation 2.1 where x is the message we want to encrypt, S is the key and y is the encrypted message.

$$(x_i + S_i) mod 2 = y_i \tag{2.1}$$

Stream ciphers are in general more efficient in the sense of hardware usage than a block cipher.

2.2.5 Block cipher

Block ciphers encrypt whole blocks of data at a time instead of bit by bit. The benefit of this is that this cipher is much faster than the stream cipher. The main drawback is that this takes a lot of space in hardware.

2.3 Storage and file system

The file system for photo cameras is specified by the *Design rule for Camera File system* (DCF) standard from Camera & Imaging Products Association, whose members include several of the larger camera manufacturers [4]. A file system is a structure used to allow a user to store multiple files on a single block device such as a hard-drive or SD-card. DCF defines that the storage medium is to be formatted with a FAT file system and that devices larger than 2 GB are to use the FAT32 variant or exFAT [5]. Only the FAT32 file system will be used in this project. FAT file system is named after a structured called File Allocation Table (FAT) and will be refereed as FAT32.

2.3.1 Disk structure

When working with block devices such as hard-drives and SD-cards there is a need to understand how individual bytes are addressed. IBM PC compatible systems uses a scheme with cylinders, heads and sectors. This scheme

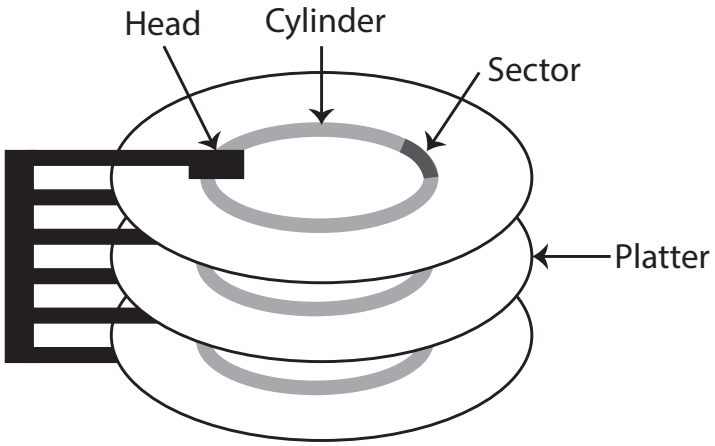


Figure 2.3: A mechanical disk with its heads and platters.

is reminiscent of the mechanical hard-drive which consists of a number of discs called platters and two heads for each platter, as seen in Figure 2.3. The addressing works by first specifying a cylinder which corresponds to a circular region on the platters. Then the head selects the specific platter and platter side[1] where the sector is located [6]. A sector is a block of data with a fixed size, in this project a sector is 512 B since this size is common on IBM PC compatible systems [7].

2.3.2 Disk layout and master boot record

Many modern computers use partitioning to allow one or more file systems on the same hard-drive. A partition table is used to store the size and position of a computer's partitions. One standard for handling partitions is the Master Boot Record (MBR) which is common on the IBM PC. The MBR is located at the first sector, first head and first cylinder and contains both a partition table and a piece of code to be booted by the computer. The code block can be used to boot an operating system or to indicate that the disk is not bootable. The partition table consist of four entries which contain information of size, type and position [8].

2.3.3 FAT32

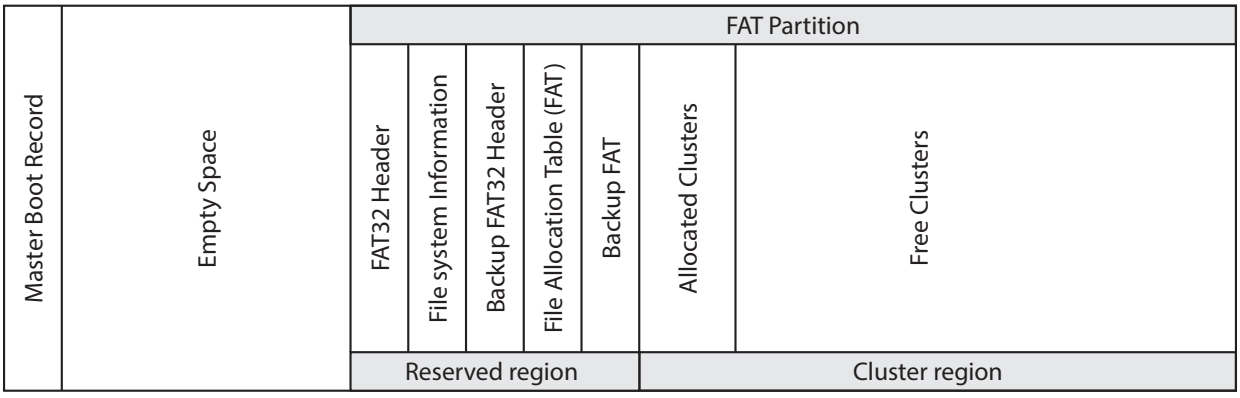


Figure 2.4: A typical layout of a FAT32 disk.

FAT is a file system which was created by Microsoft and which exists in multiple variations. *Design rule for*

[1] Each platter can have either 1 or 2 sides.

Camera File system standard allows usage of either FAT32 or exFAT on devices equal to or larger than 2 GB [5]. This project will use FAT32 as the main file system since in contrast to exFAT it is publicly documented.

Figure 2.4 shows a typical disk layout with a MBR and one FAT32 partition. The FAT32 partition contains two regions: The reserved region and the cluster region. The reserved region contains structures needed by the file system such as the FAT and the FAT32 header. The cluster region contains the file system's clusters. A cluster is a collection of sectors on the partition used for storage of directory tables and file contents [9].

The reserved region starts with the FAT32 Header that consists of a bootstrap and the BIOS Parameter Block (BPB). The header is followed by the File system information (*fs_info*) and a backup FAT32 Header. The final part of the reserved region is the FAT [9].

The BPB structure contains the information required to interpret the file system. The BPB contains data such as how many clusters exists on the drive, how many sectors there are in a cluster and how large a sector is. Additionally, it contains the file system name. The *fs_info* structure contains information such as how many free clusters are available and which is the next free cluster. *fs_info* is used to improve the performance, but the *fs_info* structure is not reliable since it is not updated by all systems that support FAT32 [9].

At the end of the reserved region there are two FAT, where the second FAT is a backup. The FAT consists of a set of 28 bit entries where each entry represents a cluster in the cluster region. A cluster entry with the value 0 indicates that the cluster is unused. Files and folders are represented as a linked list where each cluster entry contains the index to the next cluster entry or an end marker. Folders are a special type of files where the content of the folder is represented as a table. This table contains references to the folder's files and sub folders [9].

3 Design

To solve the problem discussed in chapter 1 we propose a solution of building an SD-card adapter with built in encryption logic. The adapter contains an FPGA for the logic, as well as RAM and flash storage for the adapter's firmware. Figure 3.1 shows the proposed architecture. The architecture contains both an SD slave and an SD host that will act as the interface between the SD-card and the camera. The block management logic will be used to determine which sectors that are encrypted or not, but it will also be responsible for altering the block sector address to support the algorithm running on soft core. The address altering functionality will be handled by slots which define sector range and base address. This allows the adapter to redirect the camera's read and write to another set of sectors on the SD-card.

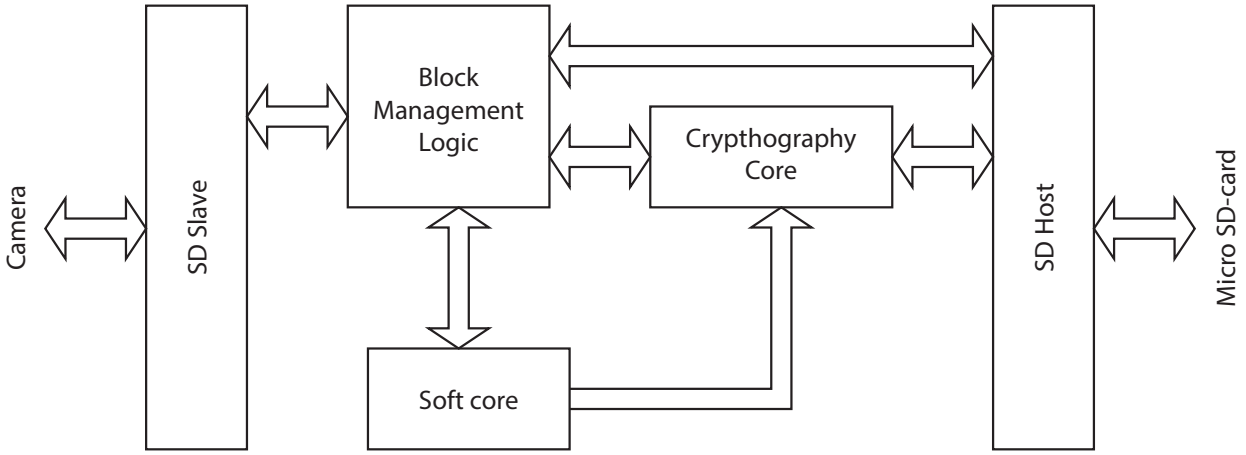


Figure 3.1: Basic architecture of the FPGA.

3.1 Disk layout modifications

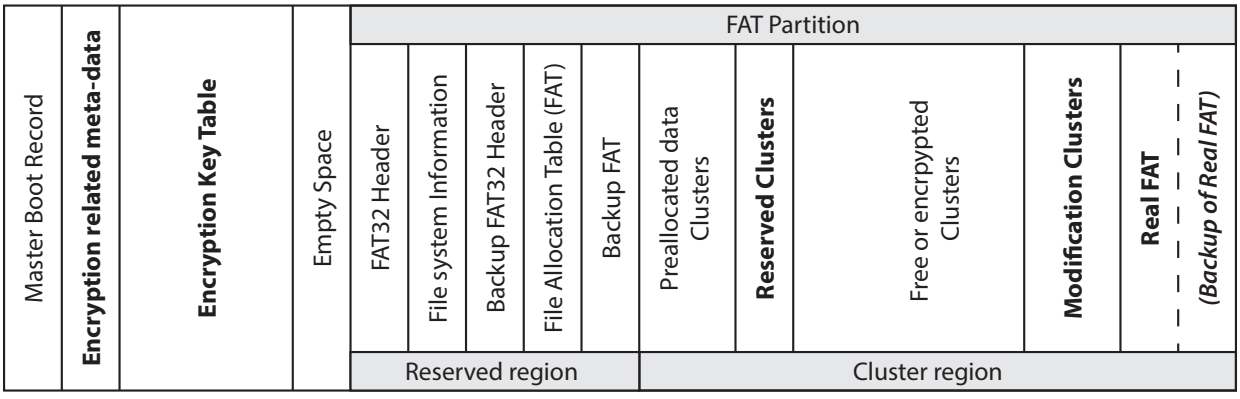


Figure 3.2: The SD-card's custom layout

To support features such as hiding images and encryption there is a need for a custom disk layout. Figure 3.2 shows the proposed layout. These additions are not part of the FAT32 file system, but rather custom additions to the layout. These additions were possible due to the design of the Master Boot Record (MBR) and FAT32 partition. There are multiple new regions added in contrast to the standard layout shown in Figure 2.4 in Chapter 2.

After the MBR there is additional data containing encryption related meta-data that we refer to as the adapter header, as well as a table consisting of previous sessions called an encryption key table.

The FAT32 partition is altered to include four new sub regions located in the FAT32 clusters region. The Real FAT region is a copy of the FAT that is used by the adapter in place of the FAT32 own File Allocation Table. The Real FAT is modified to mark the other sub regions clusters as used. The Reserved clusters sub region is a region that is off limits for the card adapter, this is used to allow some sort of protection against writing over encrypted clusters if the card is used without the adapter. Preallocated data region is intended to be visible without the use of the adapter. The *modification clusters* are used to store alterations to the preallocated folders where the tools have specified to hide encrypted files after power down.

3.2 Selected solution

Our solution uses the boot to initialize the hardware and to make hardware expensive operations on the soft-core while the majority of the normal operation is done in hardware.

The adapter will during the boot sequence generate both an intermediate vector (IV) and an encryption key. The IV and encryption key will then be used by the encryption Intellectual Property(IP)-core until the power is cut. Additionally both the IV and the key will be stored in a secure manner in the Encryption Key Table[1] on the SD-card.

The following step of the boot sequence will be when the Block Management Logic[2] is setup for relocating the real FAT to the address range of the FAT32's File Allocation Table. The real FAT is used to protect the reserved cluster sub region, the modification cluster sub region and previously encrypted blocks.

The final part of the boot sequence will be when the system hides previously created encrypted files from the camera. This is done with the help of the slots in the Block Management Logic, where the original folder is overlaid by a copy in the modification cluster sub region. Due to hardware limitations this feature is limited to three folders. During the boot sequence any folder that is marked as modified will result in a new copy in the modification cluster sub region. As a result the folder will revert to its original state after each boot.

During normal operation reading and writing of the block will be handled by the hardware. The Block Management Logic will determine which block is in plain text or not. It will also generate an interrupt when a relocated block is modified, that is used by the system to mark that copy as modified.

3.3 Encryption

We will in this section discuss the encryption part of the design. First we will discuss the different encryption choices we made during the project. Then we will discuss the encryption and decryption flow.

3.3.1 Design decisions

There are two different parts to the encryption design: the encryption of the data and the encryption of the key that is used for encryption the data. The reason for using two ciphers is that we generate an random key to encrypt the data and then we encrypt that key with an public key that have bin generated externaly.

We start looking at the data encryption. When deciding between symmetrical and asymmetrical encryption we decided to use symmetrical due to the space and power limitations. When we decided between stream and block cipher we decided on stream cipher. The reason is that stream cipher is less computationally heavy and also takes less logical space in the FPGA [3]. Also the data received is serial, which means that to be able to use block cipher we would need to store the data somewhere before encryption. This is bad due to that we will either have to store the data in an on board memory chip or use a part of the FPGA to store data. The first

[1] Encryption Key Table is a region in the modified disc layout as seen in Figure 3.1.
[2] Block Management Logic is a part of the SD-card's architecture as seen in Figure 3.2.

option is because of safety [3] and speed reasons. The second alternative is bad due to that it will use up more logic it the FPGA. For the encryption we will use an IP core called ChaCha20 supplied by Cryptech [10]. The way that ChaCha20 and other stream ciphers work is that a random number or a seed is generated. This seed is then used as starting point for a random generator. This is done several times, and between each cycle a new random number is generated. The number of times this is done depends on how high security is wanted. In our case we would like to use ChaCha20 (because of specification) which means that the cycle is repeated 20 times. However we might have to lower the iterations to be able to meet the speed requirements. At this stage of the project, we think that we will be able to use the ChaCha20 and still meet the requirements. A third party cryptographer will validate the minimum number of cycles needed to achieve the wanted level of security.

Secondly we have the encryption of the keys that is performed in software. This will make it slower than hardware but we do not consider this to be problem because it is only performed at the start of each session and does not affect transfer speed past this point. The software encryption used is xsalsa[11] cipher that is also a stream cipher. It makes sense to use a stream cipher here because we do not have a well defined block size. The reason for not using the ChaCha for this is that this would require us to develop more hardware and since it is only used rarely it will not give a significant improvement. The reason for using xsalsa is that it is included in the communication library TweetNaCL[12] that we use.

3.3.2 Encryption flow

We will here discuss the encryption flow in more detail, see Figure 3.3. The flow of the encryption can be divided into four phases. The first being that we generate a public and private key `[A,a]`[4] outside of the SD-card adapter. The public key `[A]` is then saved on the SD-card.

The second part is a setup phase that is performed in the SD-card adapter. This phase is repeated before each session[5]. During this phase we use a generation program called `curve25519` to generate another set of private and public keys `[B,b]`. We also generate a random `[key]` plus an `[IV]`(Initialization Vector). With the help of both the public keys `[A,b]` and a "random" number `[n]` we create a so called shared secret `[s]`. The `[n]` is a counter that increments with each new encryption. This is to protect against an attack usually called "two time pad". After this the processor encrypts the `[IV, key]` with an xsalsa encryption that uses the `[s]` as input before storing it in the SD-card. We also store the public key `[B]`.

In the encryption phase the `[IV]` and the `[key]` is used as variables for the ChaCha block to encrypt the data before being stored on the SD-card. Finally when the system is powered off the keys that have not been saved `[b,key,IV]` will be discarded.

3.3.3 Decryption flow

Here we will describe the decryption flow. The decryption flow consists of three steps. The first one being the extraction of the data plus the keys from the SD-card.

After extraction is done we use the `[a,B,n]` to decrypt the `[key , IV]`.

Finally we use the `[key & IV]` to decrypt the data.

[3] A third party could extract data from the memory chip
[4] Upper case letters represent public keys and lower case letters represent private keys
[5] We consider each time we start the camera as one session

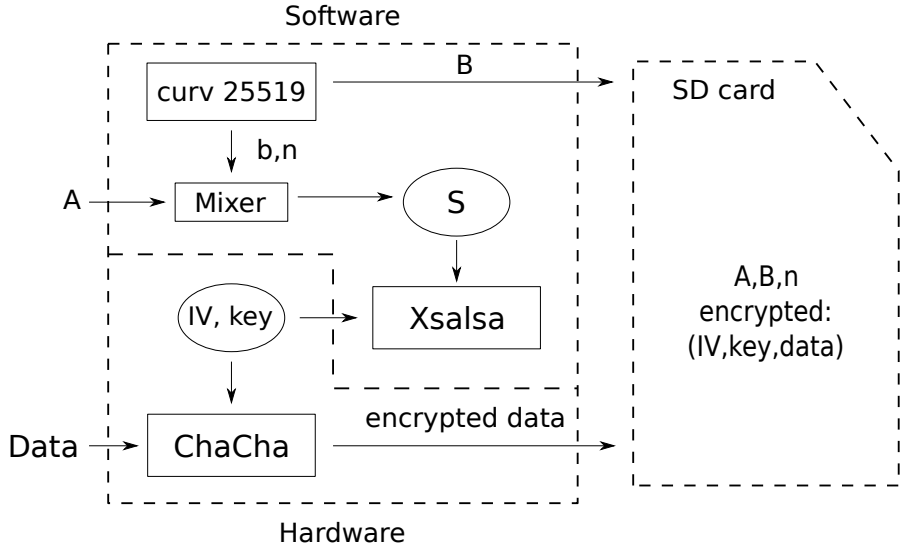


Figure 3.3: The illustration shows the encryption flow and what part of the encryption is done in hardware and what part is done in software.

4 Implementation

In this chapter we are going to discuss the implementation of the design choices made in the previous chapter. The implementation is divided into two parts, the first being the software and the second being the hardware.

We are first going to discuss the software part of the project. This includes an explanation of the file system tools designed to manipulate and experiment with the file system, as well as a description of the firmware implementation. The file system tools were required since there were no appropriate tools available.

Secondly we are going to explain the different parts of the hardware implementation. This begins with a brief explanation of what hardware was used. After this we explain the different parts of IP cores used during the implementation and finally we are going to discuss the VHDL implementation and the different blocks that were designed.

4.1 Software

The project software is divided into tools used during development, the firmware, and test software. The tools are used to prepare or manage the disc image used by the test software. The firmware is the software intended to be executed on the soft-core of the hardware and implement our algorithm. The test software consists of a simulator that was used for evaluation of the firmware's algorithm and an emulator used for firmware development.

4.1.1 File-system tools

During the project several tools were created to handle the adapter. The most important tool is called FATBuilder which is used to setup the custom disk layout of the device.There is also a tool for debugging disk images called FATReader.

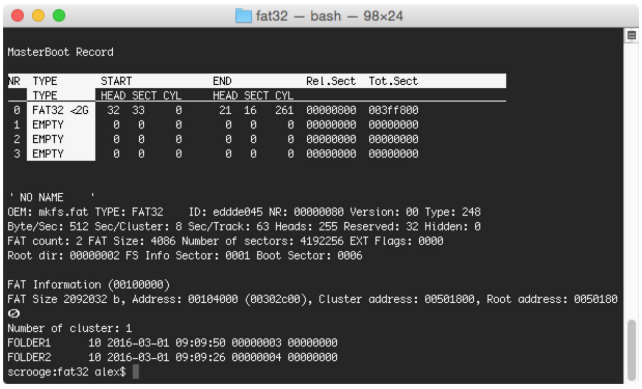


Figure 4.1: Output from FATReader.

FAT partition reader

FATReader is a tool created to debug the structure of a FAT32 partition and was used to get a deeper understanding of the FAT32 file system. The application was written in C language for a UNIX terminal. FATReader prints the partition table, the FAT32 header, structure of the root folder and useful disk layout related information. Figure 4.1 shows FATReader displaying information from an SD-card image.

FAT partition builder

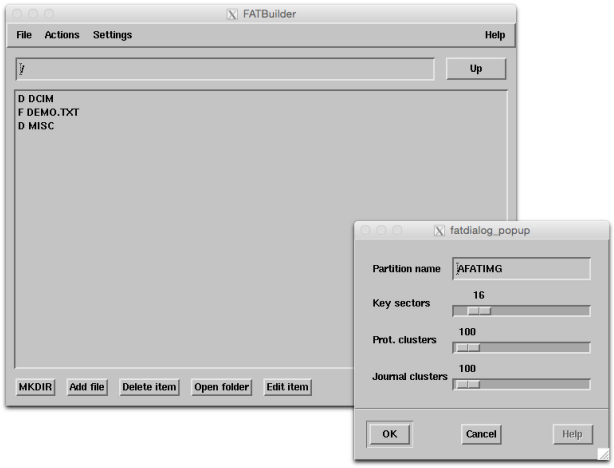


Figure 4.2: Main window and settings for the FAT32 partition and MBR.

FATBuilder is a tool used to format the SD-card and to initialize the required meta-data used by the firmware's algorithm of the adapter. FATBuilder is a graphical tool that can generate a complete disk image with an MBR and a FAT32 partition. It will also generate a key pair used for the asymmetrical encryption of the session keys. The tool generates a raw image that can be used by the simulation tools or written to a SD-card.

FATBuilder is written in C language and includes functions to fine tune the parameters of the generated image. The tool allows for creating folders and adding files. There are settings to modify the pre-allocated size of a folder, this feature is used to allow files to be added without the requirement to allocate new folders.

Another important function of FATBuilder is to select which folders any changes will be hidden after a power-off. Due to limitations in the hardware, there is a limit of 3 folders that can be protected.

4.1.2 Software simulation and emulation

There was a need to create a software prototype of the architecture before designing the actual hardware. The main reason was to simplify the debugging and validation of the selected architecture.

The selected solution was to use Network Block Device (NBD) with a server. NBD is a Linux kernel module that through a simple protocol lets a server act as a native block device. NBD supports two different protocols, a simple legacy protocol and a newer protocol with extra functionality.

The server was written in Java and implements NBD's legacy protocol. The legacy protocol was chosen for its simplicity. The server was designed to use a file as a block device. Additional logic was implemented to simulate the structure of the SD-card.

The first incarnation was a pure Java simulation of the intended solution. Later in the development process a 68000 microprocessor emulator[13] was added to the design to convert the software to an emulator. The main difference between the simulator and the emulator is that the emulator mimics the adapter's hardware and is able to execute the adapter's software, while the simulator only mimics the function of the adapter. This emulator was used during the development of the firmware, enabling easier debugging and faster development iterations.

4.1.3 Firmware

Figure 4.3 shows the basic structure of the firmware. The firmware starts with loading a header created by FATBuilder which contains information required to setup the adapter. The first task in the process is to validate whether the header is correct. If this check fails then the adapter will encrypt with a random key to make the SD-card appear broken since no partition is available. In the event of a successful check, the device will start to initialize the adapter.

The initialization starts with setting up the sector range to encrypt. The parameters for the range are stored in the adapter's header. The next step is to setup the first slot in the Block Relocation Unit (BRU) to redirect read and write from the original File Allocation Table to the adapter's own File Allocation Table, real FAT.

The initialization continues with setting up the rest of the BRU's remaining slots. Figure 4.4 shows how the firmware handles the setup of the slots. The parameters required for these slots are stored in the adapter's header. Slots are enabled as long as their length is larger than zero. The firmware will also create a new copy of the selected range if it is flagged as modified in the adapter's header. Any new copy will be stored in the "Modification Cluster" part of the altered file system. The slot's base address will be set to the pointer to the modification cluster, and then the modification pointer will be incremented with the length of the slot.

The last part of the initialization process is where the system generates the keys and the intermediate vector for the ChaCha hardware core. These keys and the base address for the three slots will be encrypted through asymmetrical encryption and stored on the SD-card.

When the initialization is completed the firmware will enter an endless loop until an interrupt occurs. When the device writes to a sector that is within the range of a slot, an interrupt is raised. Figure 4.5 shows the interrupt routine. The routine will fetch which slot that fired the interrupt and then set the dirty flag in that slot part of the adapter's header.

4.2 Hardware

To test the firmware we used a development board TE0725 [14], that is based around the Artix 100T FPGA. This board is much larger than the one targeted, granting us a larger area to work with during the development. This is beneficial for testing purposes.

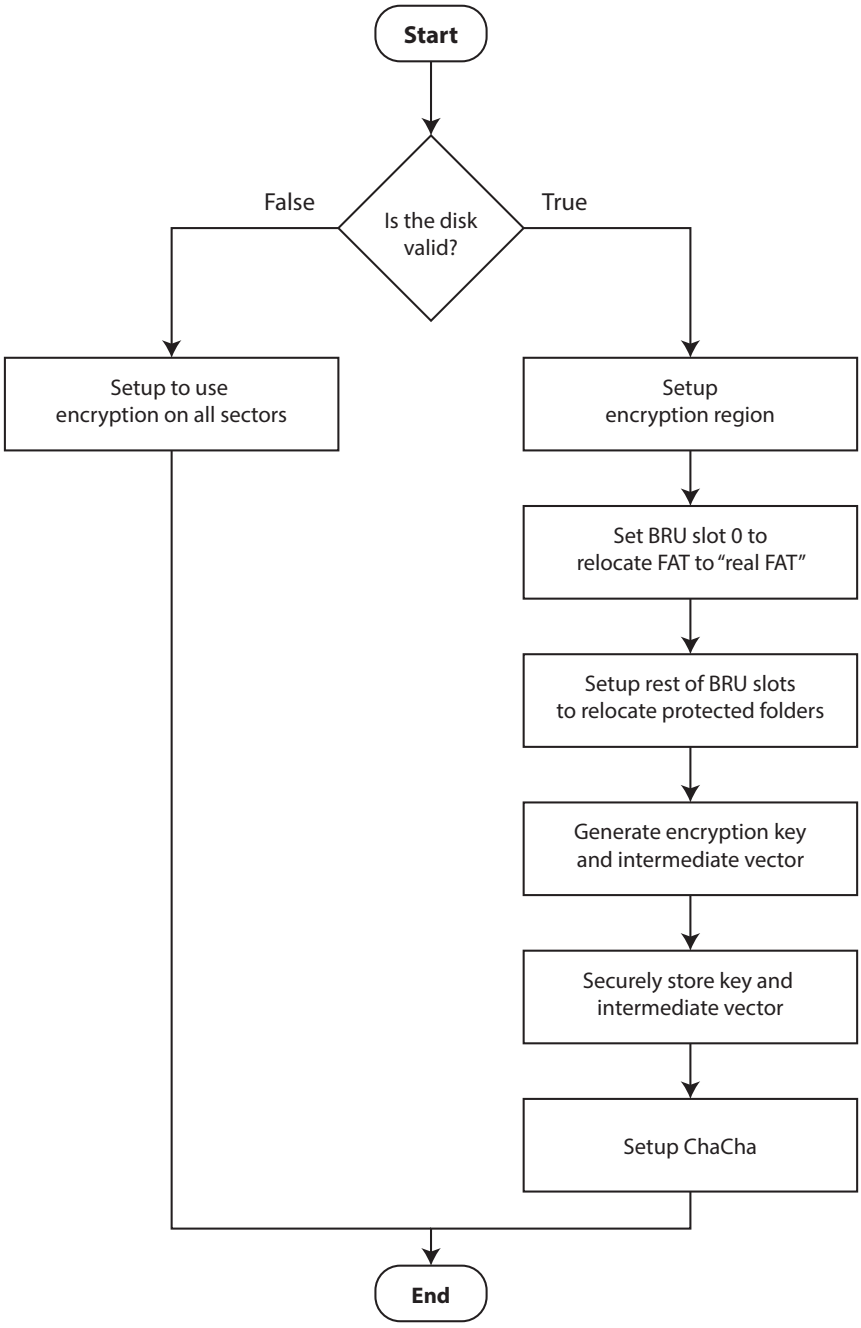


Figure 4.3: Basic flow chart over the firmware's algorithm.

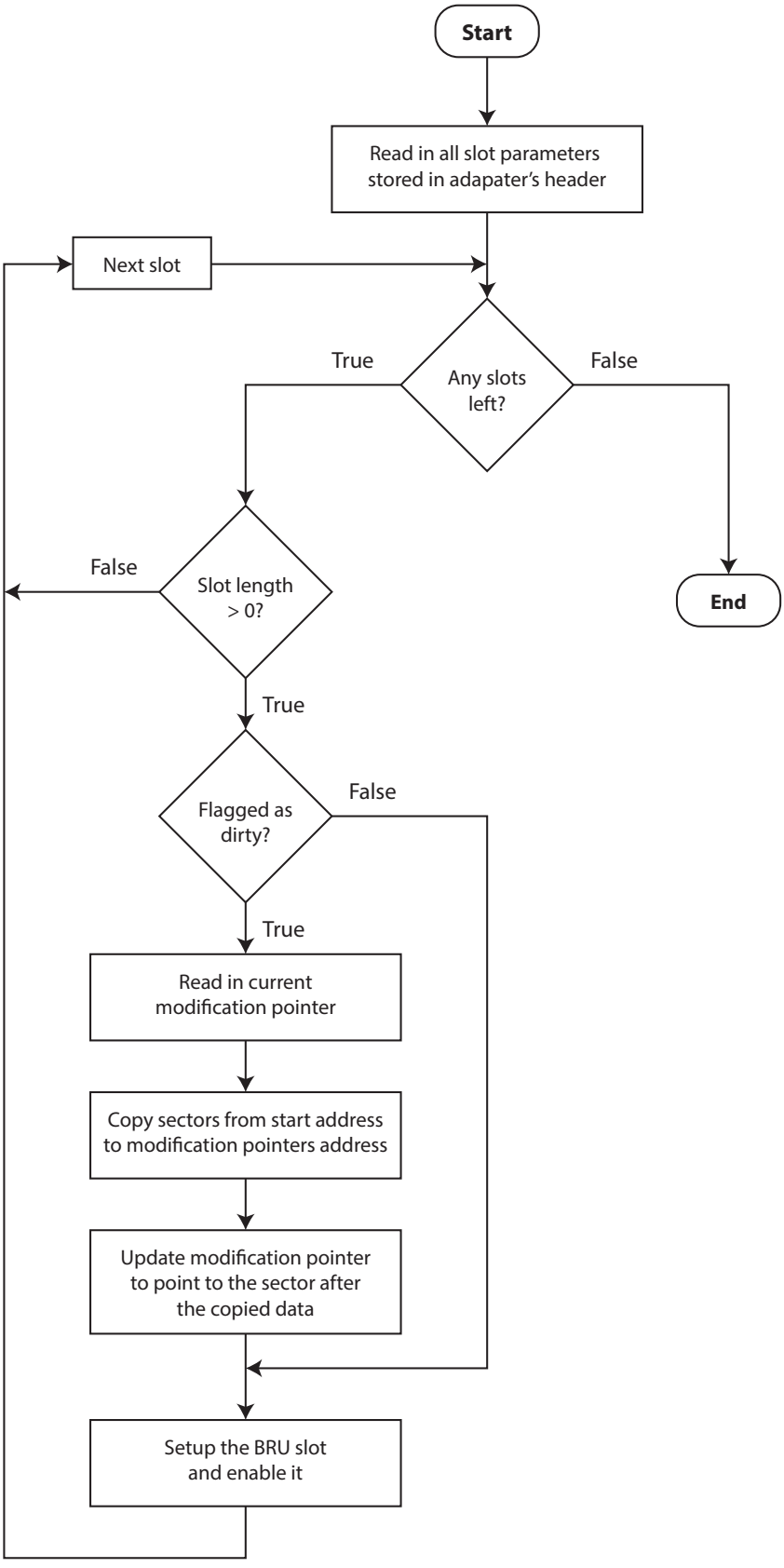


Figure 4.4: Detailed flow chart of the management of the slots.

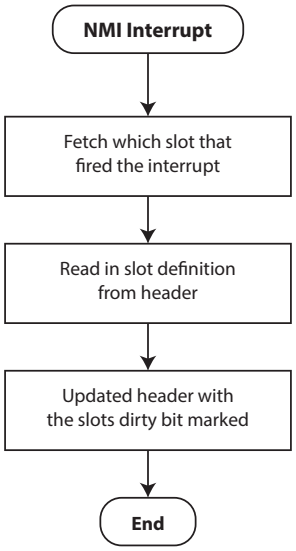


Figure 4.5: Simple flowchart detailing the write interrupt.

4.3 IP cores

In this section we will discuss the IP cores that we use in the hardware implementation. We will also mention some of the decisions made regarding the IP cores.

4.3.1 SD controller

Initially we planned on using an existing SD controller from Googles ProjectVault [15]. After working on the IP for some time we realized that it was incomplete and that even if we would complete the module, it would still not meet the speed requirements.

This meant that we needed a new SD controller. Developing one would be a task that is too big to fit in the time frame of the thesis. So instead we decided to look for alternative solutions.

Two alternatives were discussed. The first alternative was to use a flash memory with an Integrated Drive Electronics (IDE) data bus. This would be relatively easy to implement and would demonstrate the desired functionality. The drawback here is that flash memory was not the intended target device of the project. The second solution that we considered was to interface with the SD-card through an SPI bus. The benefit of the SPI approach is that it is simple to implement. We can also continue working with an SD-card as our storage device.

None of these solutions yield the desired speed, so we had to assess the overall system speed disregarding the bus. In the end we chose the SPI solution because it will yield a design that can work with the intended form factor, only lacking the desired data transfer speed of 10 MB/s.

4.3.2 Encryption block (ChaCha)

The encryption block that we use is the ChaCha IP see section 2.2. This IP is one of the more space and power consuming parts of the system. The original design is estimated to 3093 LUT on the target FPGA architecture. This is about 15% of the space in the FPGA witch makes it one of the larger parts of the design, thus we began to research where the ChaCha could be optimized for size. The most obvious place to cut down on the design was in the updating of the output. This part of the design updated all the 16 output registers in parallel. This method is very fast but it requires 16 pieces of 32bit adders that use up a lot of logic. The way that we try to solve this is to pipeline the adders. The drawback with this is that we will make the update 16 times slower but

the speed should still meet the speed requirements of 10 MB/s. The original transfer speed is above 2.5 Gbps and even if we divide this by 16 we would still exceed the requirement, hence it should not be a problem. To verify that this does not break the functionality we used the original test benches provided with the IP, to test the design after modification. And after some testing the modified ChaCha passed the test benches. After adding the pipeline the estimated logic usage was 200 LUT, a significant improvement.

4.3.3 Soft-core

To be able to manipulate the FAT32 file system to hide the encryption codes and location data we will use an FPGA-implemented soft-core processor. We looked at a number of different processors that we thought would be a good choice, see Table 4.1. The main parameter of interest is the area utilization. Most of the soft-cores have a specified size in their module description. However, Lattice and the Blaze series [16]–[18] were synthesized with the target FPGA to be able to assess the area utilization.

We were also interested in what programming languages the target processor could be programmed in and also what HDL language they were written in. It would be beneficial if we could write the firmware in C instead of assembler to speed up development. The reason for wanting the core in VHDL is that we have most experience in this HDL language.

We validated a couple of interesting cores closer. The soft-cores are shown in Table 4.1. We considered Lattice soft-cores but due to issues with development tools we found them too time consuming. Also, in order to use the smaller 8-bit processor, extra licenses were required. We also considered the Blaze series from Xilinx but the size and the fact that they are not open source is a big drawback. The t65[19] is a small processor but it only handles assembler firmware.

The two most suitable candidates for the project are the t80[20] and tg68[21]. We chose the tg65 architecture because of its size and prior experience of similar architectures. All the values are shown in Table 4.1.

After deciding on a soft-core we first needed to figure out how to use it and then design a development environment. The first test of the soft-core was performed with handwritten opcodes designed as a state machine in VHDL. The test program consisted of a simple counter that drove a pin on the development board.

After this we set up a cross compiler and added an HDL based RAM memory to be able to test our firmware on the development board.

In order to program the processor, the C code was compiled and converted to the `coe`[1] format. The coe file was then integrated in the RAM memory when synthesizing the VHDL project.

4.4 VHDL implementation

This section will explain the different VHDL blocks that were designed during the project.

4.4.1 Bus controller

To be able to use the tg68 soft-core processor we designed a bus controller to handle the data to and from the processor. The controller directs the data to and from the correct memory or module dependent on the address-out from the tg68. To assist with debugging we also included an IO register.

The functionality of this block was tested with the help of a VHDL-based test bench that feeds the module with addresses and asserts that the right memory or module was returned to the processor.

[1] This is a Xilinx specific memory format

Table 4.1: List of plausible soft-cores

Processor types	language	HDL	license	size	url
MicroBlaze	c/c++	encrypted	proprietary	728 Luts 1017 Slices	MicroBlaze
PicoBlaze	c	encrypted	proprietary	69 Slices	PicoBlaze
latticeMicro32	c/c++/assembler	Verilog(VHDL wrapper)	open	2400 Luts	latticeMicro
latticeMicro8	c/assembler	Verilog(VHDL wrapper)	open		latticeMicro
t80	c	VHDL	open	398 Luts Artix	t80
tg68	c	VHDL	open	2500 Luts Artix	tg68
tg68 minimized	c	VHDL	open	2252 Luts Artix	tg68
t65	assembler	VHDL	open		t65

4.4.2 SPI communication

This module is designed to handle the communication between the system and the SD-card. The module supports initiation commands and the read/write block. The timing diagram and the command sequences are based on information from [22].

The read command fetches a 512-byte block of data and stored it in a working memory that the processor has access to. When the processor is done with the block it tells the SPI controller to write it back to the SD-card.

The module consists of two main parts and some glue logic (see Figure 4.6). One part is the `SPI_module` that communicates with the memory card and one part is the `SPI_controller` that handles the communication with the processor. The `SPI_controller` interprets the commands from the processor and then sends the corresponding commands to the `SPI_module` that handles the SPI communication details.

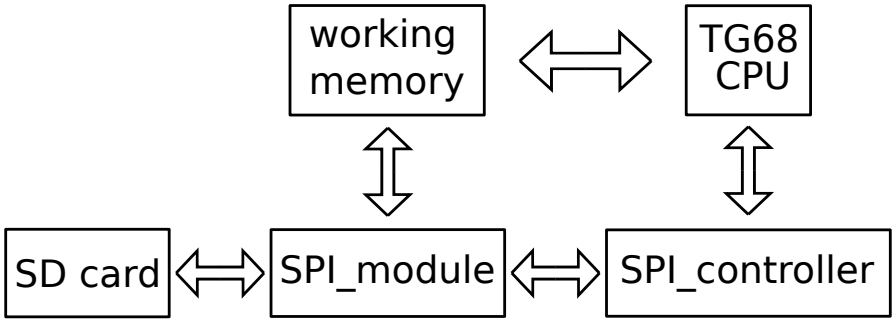


Figure 4.6: This figure shows the separate VHDL modules included in the SPI transfer process.

The functionality of this module is tested with the help of a VHDL based test bench. The test bench feeds the block with the supported SPI commands and checks that the response on the SPI bus is according to expectations.

5 Results

Here we will discuss to what degree we have met the goals of this project. We will first evaluate the functionality and then we will talk about the performance aspects. Due to a shortage of time, the results presented here are only simulated and not tested on hardware.

5.1 Functionality

The firmware's algorithm has been tested through both simulation and emulation of the adapter platform. These tests show that our algorithm is applicable.

5.2 Size

The limitations of the project were that the whole design should fit in an FPGA small enough to fit onto a specially designed circuit board inside an SD-card adapter. The FPGA chosen for this purpose was an Artix 35T. We have verified the size of our design by synthesizing the main part of it in Vivado[23]. The result shows that we use 15 % of the total size available. There will be other parts such as pseudo-random number generator and some more overhead that have to be added to complete the full implementation, but our assessment is that there is enough space left to accommodate this. With these results in mind we consider this goal to be reached.

5.3 Speed

The goal of the system was to be able to reach a transfer speed of 10 MB/s. We can see that in this design we have three main bottlenecks: data transfer, processor operations and encryption.

In the first case we have the transfer of data that should be performed with an SD transfer protocol. In the state the project is right now we do not have a module for the SD controller. We are instead using an SPI bus to transfer the data. This is not fast enough to reach the speed goal, thus a final version would need a real SD controller.

The second part is the processor operations, a relatively slow part of the system. However, we can disregard this when it comes to operation speed due to that it is mainly used in the start-up sequence and will not affect the speed of data transfer after this point.

The final bottleneck is the encryption. The speed of the original design utilizing ChaCha has been measured to have a transfer speed of 2.5 Gbps but in order be able to meet the size requirements we cut down the update block from 16 adders to 1. This would mean that the update phase would take 16 times longer, hence lowering the performance of the system 16 times which would results in a speed of almost 160 Mbps , or 20 MB/s. We can conclude that this part of the system should meet the required speed.

5.4 Power

The power consumption results only considers the consumption of the VHDL part of the project due to the fact that we do not have access to the SD-card adapter platform right now. This result can only be seen as an indication of what range the final consumption will end up in. When the whole system is running we also have passive components, power converters, memory and more that will affect the power consumption.

From the Vivado power assessment chart we can see that the total theoretical consumption of the VHDL system is 0.17 W. This is with the system running at 50 MHz and not with all the blocks included. We still

consider this a good indication because the power intense parts of the design are included and the final system will run at 50 MHz which is the max speed for the encryption core.

6 Conclusions

In this chapter we will first discuss what lessons that we have learned during the project. After that follows a conclusion section that talks about what we achieved during the project.

6.1 Summary

We succeeded to prove that the algorithm works and that the intended firmware is functional in the emulated environment. We also successfully hid the key inside the file system during simulations. An encryption with private and public key was also implemented in simulation. All implemented VHDL blocks were tested and verified to be functional, but not tested together as complete system. The estimated metrics received from the synthesis tools indicate that the system should fit within the area constraint of the FPGA. With the exception of the SD controller, the system performance was up to par with the requirement. Since we used the slower SPI protocol instead of the native SD protocol the system as a whole failed to fulfill the speed requirement. Considering only the VHDL part of the design the simulations show that we will be able to meet the power demands of the SD-card standard. This does not take into consideration the power consumption of any peripheral components.

6.2 Lessons learned

Firstly, we discuss the time management planning of the project. We will then discuss what we would do differently if we would redo the project. Finally we will sum up with what is left to do and possible improvements.

There were several problems that were discovered during the development that we did not anticipate. Originally, we underestimated the amount of software needed to complete the project. The initial plan estimated that both team members would be available for hardware development during the end phase. Much time was devoted to create tools such as FATBuilder and the SD-card platform emulator.

During the process of the project it was discovered that the SD host/slave controller was not functional. This lead to the creation of a much simpler SPI based SD host/slave. To successfully fulfill the speed requirement there is a need to implement an SD-card with support of high-speed modes.

Initially, space was an issue with the system where the encryption core used the majority of available resources. It was later discovered that the encryption core could be decreased in size by lowering the throughput.

In retrospect, there are some decisions that we should have made differently. The most prominent faulty decision was the attempt to implement the system to fit within the Artix7-35T instead of developing a proof-of-concept first. This decision lead to the selection of an undocumented soft-core that proved to be hard to use. Instead we should have used one of the soft-cores provided by Xilinx even though it would not fulfill the space requirement.

We could also have waited with the optimization of the ChaCha encryption core. This was time consuming and space optimization could have waited until after proof of concept. If we had skipped this we would have had more time to reach the proof of concept.

Due to time constraints, the system still includes components that either are not finished or are deemed inadequate for final use. The list below summarizes what we think needs to be done before this can be considered a finished design.

- We need to implement a FPGA specific random number generator to obtain sufficient security.
- We need an external noise source for good encryption.
- We would need to finish interfacing the hardware.
- We need to evaluate power on the full system.

- We want to do a final system test with hardware and firmware.

There is still a need for more work on the hardware before we have a complete proof of concept. For a final product there is also a need for development of user-friendly tools to format the SD-card and to read out encrypted images. In the end we have managed to show that our proposed algorithm is feasible.

6.3 Revised time-plan

In the original time plan as seen in Appendix A we started working on two parts in parallel. The first part being the software, and the second part being the HDL design. And in the original plan the software wold be don early in the project and we wold then both work on the HDL part of the design. However the software was shown to take much longer than expected and there were also the need to develop software tools that was not part of the original plan. So the software part ended up stretching over the whole project and this led to the HDL part not being done in time.

Another part that was unexpected was that the intended SD-card controller was not in a state where we could use it. Because of this we had to develop extra HDL parts that were not part of the original plan.

Lastly we like to mention that our time plan was flawed from the beginning. We started focusing on optimising the design before having an proof of concept. The project would have benefited much more from having an functioning prototype before optimising.

References

[1] F. Strömberg, "Internal research", stromberg@mullvad.net, 2016.

[2] SD-Group, *Sd specifications part 1- physical layer simplified specification*, SD Group and SD Association, Jan. 2013.

[3] C. Paar and J. Pelzl, *Understanding cryptography: A textbook for students and practitioners*. Springer Science & Business Media, 2009.

[4] *CIPA - camera & imaging products association: List of members*, [Accessed 27 January 2016], Camera & Imaging Products Association. [Online]. Available: `http://www.cipa.jp/guide/member-list_e.html`.

[5] *Design rule for camera file system: DCF version 2.0*, CIPA DC-009-2010, Camera & Imaging Products Association, Apr. 2010.

[6] IBM, *IBM 1311 disk storage drive*, File No. 1311-07 Form A26-5991-0 [Accessed 1 June 2016]. [Online]. Available: `http://bitsavers.trailing-edge.com/pdf/ibm/140x/A26-5991-0_1311diskDrive.pdf`.

[7] M. E. Fitzpatrick, "4K sector disk drives: Transitioning to the future with advanced format technologies", Toshiba America Information Systems, Inc, Tech. Rep., 2011.

[8] *MBR (x86)*, [Accessed 7 April 2016], OSDev.org. [Online]. Available: `http://wiki.osdev.org/MBR_%5C%28x86%5C%29`.

[9] *Microsoft extensible firmware initiative fat32 file system specification*, Microsoft Corporation, Dec. 2000.

[10] *Encryption IP*, [Accessed 27 January 2016], Cryptech. [Online]. Available: `https://cryptech.is`.

[11] D. J. Bernstein, "Extending the salsa20 nonce", The University of Illinois at Chicago , Department of Computer Science, Tech. Rep., 2008.

[12] *Tweetnacl*, [Accessed 15 November 2016]. [Online]. Available: `https://tweetnacl.cr.yp.to/software.html`.

[13] *M68k*, [Accessed 15 November 2016]. [Online]. Available: `https://github.com/tonyheadford/m68k`.

[14] *Te0725*, [Accessed 15 November 2016]. [Online]. Available: `http://www.trenz-electronic.de/products/fpga-boards/trenz-electronic/te0725-artix-7.html`.

[15] *Projectvault*, [Accessed 31 January 2016]. [Online]. Available: `https://github.com/ProjectVault`.

[16] *Latticemico32*, [Accessed 15 November 2016]. [Online]. Available: `http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx`.

[17] *Latticemico8*, [Accessed 15 November 2016]. [Online]. Available: `http://www.latticesemi.com/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/Mico8.aspx`.

[18] *Microblaze*, [Accessed 15 November 2016]. [Online]. Available: `https://www.xilinx.com/products/design-tools/microblaze.html`.

[19] *T65 cpu*, [Accessed 15 November 2016]. [Online]. Available: `https://opencores.org/project,t65`.

[20] *T80 cpu*, [Accessed 15 November 2016]. [Online]. Available: `https://opencores.org/project,t80`.

[21] *Tg68 cpu*, [Accessed 15 November 2016]. [Online]. Available: `https://opencores.org/project,tg68`.

[22] *How to use mmc/sdc*, [Accessed 27 January 2016]. [Online]. Available: `http://elm-chan.org/docs/mmc/mmc_e.html`.

[23] *Xilinx vivado*, [Accessed 15 November 2016]. [Online]. Available: `https://www.xilinx.com/products/design-tools/vivado.html`.

A Original time plan

Tasks

Name	Begin date	End date
Research	1/19/16	2/12/16
Reading relavent documents	1/19/16	2/5/16
Evaluation of soft cores	2/1/16	2/12/16
Research of FAT32 and MBR format	1/25/16	2/5/16
Architecture	2/8/16	2/19/16
Software prototype	2/16/16	3/25/16
Create SD card model as a linux remote block device	2/16/16	2/26/16
Create SD card formating tools	2/17/16	3/25/16
Add selective encryption of device blocks	2/29/16	3/2/16
Add block write protection	3/3/16	3/8/16
Implement hiding of files from previous sessions	3/9/16	3/25/16
Software proof of concept	3/28/16	3/28/16
Hardware prototype	2/22/16	5/17/16
SD Host and slave connection	2/22/16	3/11/16
Communication between SD host and slave	3/14/16	3/14/16
Implementation of soft core	3/15/16	3/25/16
Integration of encryption core	3/28/16	4/8/16
Writing firmware	4/11/16	5/17/16
Communication with encryption of blocks	4/18/16	4/18/16
Impliment block management logic	4/18/16	5/16/16
Communication with block relocation added	5/9/16	5/9/16
Proof of concept	5/18/16	5/18/16
Test & Verification	5/9/16	5/27/16
Documentation	1/19/16	6/13/16
Writing planning report	1/19/16	2/1/16
Submit planning report	2/2/16	2/2/16

Tasks

Name	Begin date	End date
Writing report	2/3/16	6/13/16
Submit halftime report	3/22/16	3/22/16
Submit report to opponents	5/24/16	5/24/16
Prepare presentation	6/1/16	6/7/16
Presentation	6/8/16	6/8/16
Submit final report	6/14/16	6/14/16

Gantt Chart

Name	Begin date	End date
Research	1/19/16	2/12/16
Reading relavent documents	1/19/16	2/5/16
Evaluation of soft cores	2/1/16	2/12/16
Research of FAT32 and MBR format	1/25/16	2/5/16
Architecture	2/8/16	2/19/16
Software prototype	2/16/16	3/25/16
Create SD card model as a linux remote blo...	2/16/16	2/26/16
Create SD card formating tools	2/17/16	3/25/16
Add selective encryption of device blocks	2/29/16	3/2/16
Add block write protection	3/3/16	3/8/16
Implement hiding of files from previous sess...	3/9/16	3/25/16
Software proof of concept	3/28/16	3/28/16
Hardware prototype	2/22/16	5/17/16
SD Host and slave connection	2/22/16	3/11/16
Communication between SD host and slave	3/14/16	3/14/16
Implementation of soft core	3/15/16	3/25/16
Integration of encryption core	3/28/16	4/8/16
Writing firmware	4/11/16	5/17/16
Communication with encryption of blocks	4/18/16	4/18/16
Impliment block management logic	4/18/16	5/16/16
Communication with block relocation added	5/9/16	5/9/16
Proof of concept	5/18/16	5/18/16
Test & Verification	5/9/16	5/27/16
Documentation	1/19/16	6/13/16
Writing planning report	1/19/16	2/1/16
Submit planning report	2/2/16	2/2/16
Writing report	2/3/16	6/13/16
Submit halftime report	3/22/16	3/22/16
Submit report to opponents	5/24/16	5/24/16
Submit final report	6/1/16	6/7/16
Presentation	6/8/16	6/8/16
Submit final report	6/14/16	6/14/16