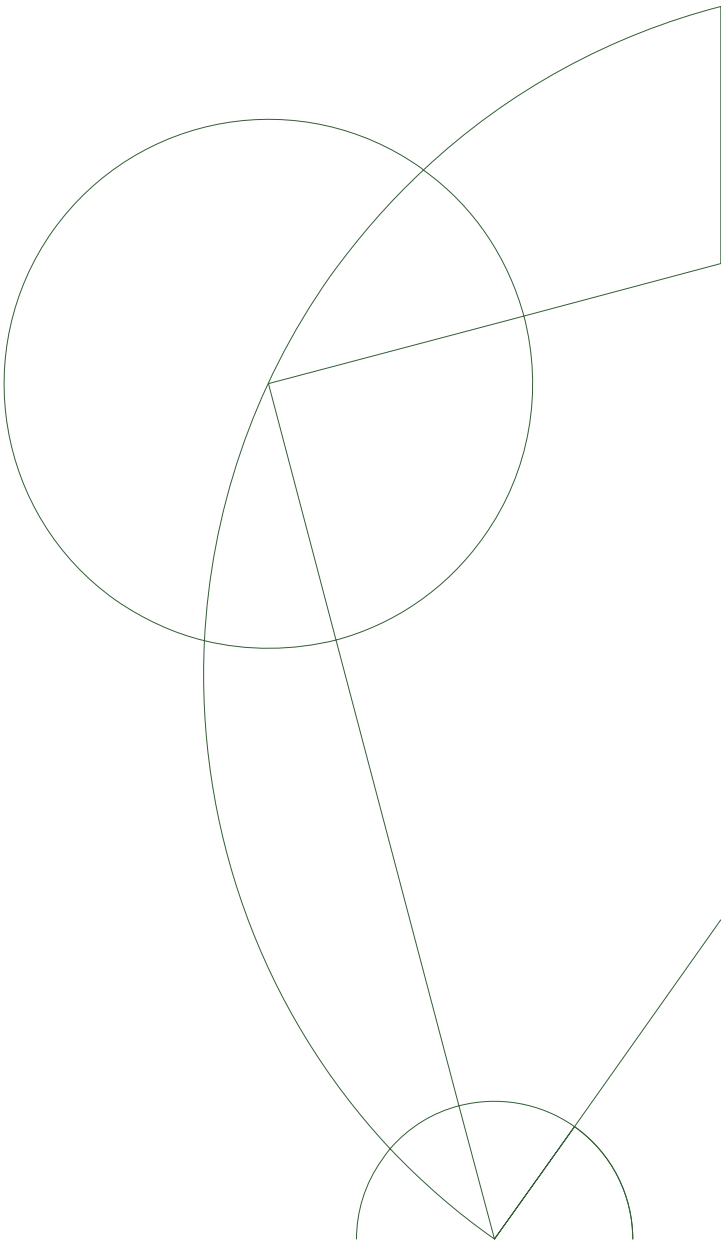
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bachelor Project

Jacob Herbst(mwr148), Jonas Flach Jensen (sjm233)

Cryptographic library for FPGA's



Advisor: Kenneth Skovhede

Abstract

Computer security and cryptography is ubiquitous and is a critical aspect of computer science. Often cryptography is handled in the CPU as with the general majority of computing. Still, in some cases, CPUs might be a suboptimal solution, for instance, when low power consumption is critical. In such a case, a Field Programmable Gate Array (FPGA) is a good alternative. This report will present a library of four cryptographic functions designed for FPGA devices: MD5, SHA256, AES, and ChaCha20. We will present the underlying algorithm of the four functions and how we have implemented these using a high-level programming model Synchronous Message Exchange (SME) in C# instead of the usual approach of using a Hardware Description Language(HDL). We will further present how we, by pipelining, have achieved performance comparable to a CPU of a similar price range at much lower power consumption. In the process, we have tried investigating different parts of FPGA programming and how this can be applied in SME to see if it would further improve performance. We were, however, not able to achieve this and hence included some reflection on how approachable FPGA programming is using a high-level model such as SME. Code for the project can be found at:
https://github.com/Spatenheinz/Bachelor

Contents

List of Illustrations

Figures

Tables

Chapter 1

Introduction

Cryptography is an essential branch of both mathematics and computer science. It is used in wast variety of fields, from security to compression and data validation. Because of their widespread use, cryptographic algorithms should be fast and efficient without exposing vulnerabilities. With this in mind, algorithms are often developed with hardware as a major consideration. With hardware in focus, cryptographic functions are appropriate for a wider variety of devices since the generality of a CPU is not critical. Some examples include low-resource devices, Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs). Devices such as the latter two are similar in that they are designed to compute a specific task instead of doing general computations. This allows the devices to do faster computations at a lower resource cost since the need for generality is omitted. The advantage of an FPGA over an ASIC is that it is generally more approachable for private users and academics and that they can be field programmed, meaning the specific computational "purpose" can be changed, whereas ASICs are "soldered" to its purpose. Based on this, we wanted to explore how accessible FPGA programming is, and i our case using a high level programming model to achieve this, and if it would be possible to get any competitive advantage from an FPGA over CPUs without prior experience with FPGA development.

1.0.1 Project Objective

This project's main objective is to implement a cryptographic library that can be synthesized on an FPGA; hence a secondary objective explores the area of FPGA development. This is done using a high-level programming model, Synchronous Message Exchange (SME) intended for FPGA development. This approach is chosen rather than using a Hardware Description Language (HDL), as it allows for easier implementation and focuses on the algorithmic aspect, omitting the low-level focus. Hence, no need for us to know how to write HDL. The library consists of four cryptographic functions, which cover various cryptographic purposes and their peculiarities in the specific implementation. We have implemented each cryptographic function naively and then tried to improve these by pipelining the computation. Lastly, we have made comparisons of how well a naive version compares to a pipelined one and how well it compares with its CPU alternative.

1.0.2 Report Structure

Chapter 2 presents some background knowledge about the target device, Synchronous Message Exchange, and cryptography. Chapter 3 will present the four cryptographic functions in the library in the following order: MD5, SHA256, AES, ChaCha20. Chapter 4 revolves around the implementation of the functions, focusing on SME and how the design corresponds to FPGAs rather than the actual C# code used in the project. Chapter 5 presents the performance of the functions and the proposed optimizations. Chapter 6 will serve as a discussion on SME as a tool for FPGA development and proposals for future work. Finally, we conclude in Chapter 7.

1.0.3 Requirements for the reader

It is assumed the reader of this report has a knowledge equivalent to that of a 6th semester bachelor student in computer science or higher.

Chapter 2

Background

This chapter will present the theoretical aspect applied in implementing our cryptographic library. Firstly, we will briefly give a high-level overview of the target device, FPGA, followed by an introduction to the programming model which has enabled us to implement the cryptographic functions included in the library. A review of the different cryptographic terms and constructs used in the library is presented for good measure. The algorithm of the four cryptographic functions in the library is presented in detail in the following order: MD5, SHA256, AES, and ChaCha20.

2.1 Field Programmable Gate Arrays

An FPGA is an integrated circuit that can be reconfigured in the "field". It stands as the opposite of an ASIC, which will have a single purpose, whereas FPGA's can be reprogrammed to have different purposes. It means that an FPGA can be configured to work as a CPU, a GPU, or something else entirely. This can be done in an HDL, such as Verilog or Very High-Speed Integrated Circuit Hardware Description Language (VHDL), and is typically run through a program to synthesize/implement the design on the FPGA. FPGA's consist of a set of Configurable Logic Blocks (CLBs) and interconnects between these. CLBs and their interconnects are the reason FPGA's are reprogrammable. They differ from classic logic gates as NAND and such, used in a CPU, since they are constructed by look up tables, which can be reprogrammed, instead of fixed gates. Since these lookup tables can be built for specific purposes, an FPGA can be programmed to do one thing and do it well. This lack of generality is often suitable for both performance and power consumption, compared to a CPU, which needs to be able to do general processing and thus, in general, has more waste.

2.2 Synchronous Message Exchange

Synchronous Message Exchange is a programming model to enable FPGA development for software programmers using high-level languages. SME is based on Communicating Sequential Processes (CSP) and at its core constructs from said process calculi, making use of the elements which has proven useful in hardware design[1]. Using the following concepts from the CSP model[2], SME can be derived:

- A program consists of a set of named processes.
- Each process runs concurrently with no form of sharing with other processes.
- Concurrent processes can communicate using message passing.

SME has a similar notion of processes. There exist two types of SME processes, `simple process` and a `simulation process`. Of these, the simple process corresponds to a process in CSP as described above. Each simple process in SME will only share communication channels and constants with the other processes. Simple processes will consist of a set of input and output busses, an `onTick` function, which will run on every clock tick, and a set of optional variables and functions. Since the model revolves around mapping to hardware, every construct inside a simple process should have a fixed size, which means no dynamic lists, while-loops, etc. On the other hand, simulation processes will not be a part of the actual hardware design, making dynamic constructs legal.

Furthermore, simple processes have an optional property `Clocked Process` which means the process will be triggered in parallel with all the other clocked processes at every clock tick, whereas non clocked processes first will be triggered when all processes it depends on have finished. For the communications channels, SME extends the concepts from CSP by using buses. Instead of using explicit naming for sources and destinations, each process will consist of a set of input and output busses that it can read and write to, respectively. Furthermore, these buses use broadcasting as means of synchronization instead of the blocking non-buffered approach. The broadcasting happens every clock cycle on the internal clock. A bus is essentially just a collection of fields that can be read and written depending on the process's access, merely a data transfer object. Thus a simple (and pointless) process that adds two numbers might have two input busses X{valid,x} and Y{valid,y}, where X and Y are unique bus identifiers, valid, x and y are fields, with denoting whether there is any data on the bus and x and y is the data. Inside the `onTick` function, which will be run every tick of the internal clock of SME, could then add the two values x and y if their valid fields were set to true and write the result to a bus RES{valid, res}. It is worth noting that a process should not necessarily have one, or possibly multiple, "valid"-flags which shows if there is any data on the bus, but this is common in cases where the processes communicate using the ready/valid handshake, for instance, the one specified by the AXI protocols, which is the process communications protocol we will be using. It is easy to see how an SME model can be transformed into a dependency graph with processes being nodes and buses the edges. From the dependency graph, it is possible to create an Abstract Syntax Tree (AST) which can be translated into VHDL code[1], thus creating the bridge from the high-level model to the low-level hardware implementation. This, in turn, can be fed into a tool such as Xilinx Vivado to synthesize the implementation to actual hardware. For the cryptographic library covered in this report, we will be using the C# implementation of SME by the models creators[1].

2.3 A crypto library

Cryptographic functions are used by developers across most branches, whether communicating securely over a network or hashing programs to do version control. So there is a motive for having a crypto library for FPGA's. Such a processor has been made before. IBM created their own "IBM 4758 Secure Coprocessor"[3]. Another point is modern Hardware Security Modules (HSM) which also does this. However, the problem with the existing solutions is that many of them require setting up a royalty-based licensing deal, making it difficult to use for experimental development, small projects, research, and academics. So we set out to create an open-source crypto library.
The crypto library consists of 4 cryptographic functions, two of which are hash functions, MD5 and SHA-256, and two of which are ciphers, AES, and ChaCha.
It should also have an API allowing users to utilize these functions in their projects, as they would with any other library. These implementations should also be optimized in terms of speed to compete with the existing software solutions.

2.3.1 Hashing

Hashing is a mathematical concept referring to using a hash function to map some data of arbitrary size to a value of a fixed size. Cryptographic hash functions are a subset of all hash functions. The reason for this is that for a hash function to be a cryptographic hash function, it needs to uphold several properties to ensure it is secure, such as ensuring that it is hard to find collisions. Computers also have limited space in memory which limits the implementation of hash functions. Lastly and most importantly, computers can't

do true randomness. If a hash function can be implemented with a limited input space, it is pseudo-random, and upholds specific properties listed below, it can be categorized as a "Cryptographic Hash Function".

- It should be deterministic, as the same hash must be computed given the same input.
- It is unreasonably hard to predict the hashed value. One reason for this is the requirement to exercise the avalanche effect, meaning the tiniest change in the input message would resolve significant changes in the hash.
- It is collision-resistant, meaning it is unreasonably hard to find two different messages to have the same hash.

Merkle-Damgård construction
As stated previously, this library includes implementations of MD5 and SHA-256. These are very similar in design and follow a widely known construction method for cryptographic hashing, the Merkle-Damgård construction. One of the reasons this approach is desirable, when developing a cryptographic hashing algorithm, is because the hash function will be collision-resistant given the compression function itself is collision-resistant[4]. From Figure 2.1 one can see the construction of the hashing function. One can see that the message will be padded to have a certain length since any compression function must work on a fixed size. The compression function `f` will initially take two arguments, the Initialization Vector (IV) and the first message block. `f` will then produce a result of the same size as the initialization vector. This result will then be fed into the next iteration of `f` and the second block of the message. This process is repeated until the entire padded message has been processed. From here, a potential finalization function can be applied to improve the hash, and a hashed value is hence produced.

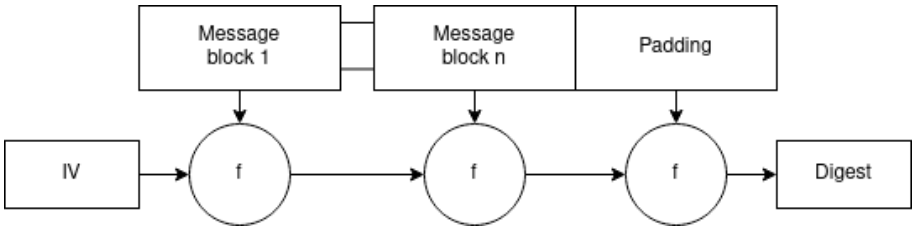


Figure 2.1: Merkle-Damgård construction. Each rectangle presents data, where the messages will be divided into n blocks, of which the size depends on the specific algorithm. Similarly, the IV and the Digest also will have variable lengths, but these will have the same size. The circles `f` denotes the compression function, and the finalization function is optional (depending on the specific algorithm). Each arrow shows the flow of data.

2.3.2 Cipher

Ciphers are algorithms used for symmetric encryption and decryption of data. This means that rather than generating a fixed-sized output like most (if not all) hash functions, a cipher should always output as many bits as its input. There are generally two types of ciphers: block ciphers and stream ciphers. They a similar in that they always have to be a bijective mapping from key/plaintext to ciphertext, such that no two plaintexts can map to the same ciphertext. Furthermore, Claude Shannon defined[5] that

secures ciphers should have confusion and diffusion. Confusion meaning a bit of the ciphertext should depend on the key in multiple ways, such that no connection between those two is easily observable. Diffusion meaning a single change of bit in the plaintext should change most bits in the ciphertext. For their internal workings, the two types of ciphers are, however, vastly different.

Block ciphers
Block ciphers are defined to work on a fixed-sized block of bits, which often, and in the case of AES, is 128 bits. This requires some considerations; firstly, data that is not a multiple of the block size will require some sort of padding. There exists no single standard for padding a block. To handle data that does not directly fit into a single block multiple Modes of operations are defined. They differ quite a lot in detail, but all conceptually turns the block cipher into a stream. Appendix D shows 3 different "popular" modes of operation. The most simple is Electronic Code Book (ECB), which will independently encrypt each data block. It is worth noting that this is not the most secure mode since identical data blocks will produce identical cipher blocks. Another more secure method is Cipher Block Chaining (CBC), which will xor the previous block's ciphertext with the plaintext of the current block before encrypting the block. This approach is an inherently sequential method as block i is dependent on the result of block $i-1$. More parallel and secure modes also exist, such as Counter Mode (CTR) and Galois Counter Mode (GCM). These work by taking a nonce as input to the cipher instead of the plaintext. The result from doing AES encryption on the nonce will then be XORed with the plaintext. Each block after the initial will then take the nonce increased by some fixed size per block. Modes such as ECB and CBC needs a separate decryption algorithm to produce the plaintext from the cipher, whereas CTR can use the same as the plaintext XOR $\text{nonce}_{\text{encrypted}}$ = cipher, and cipher XOR $\text{nonce}_{\text{encrypted}}$ = plaintext.

Stream ciphers
A stream cipher, as the name suggests, works using stream and is thus independent of size. Stream ciphers generate a pseudorandom keystream, which will be combined with the plaintext. Most often, this combination will be by XOR, such that bit 0th of the plaintext will be XORed with the 0th bit of the keystream.

Chapter 3

Library functions

3.1 MD5

The Message-Digest algorithm MD5 is a reasonably simple one-way hashing function that produces a 128-bit digest specified in 1992 in RFC 1321[6]. The MD5 algorithm will thus create a 128-bit digest from an arbitrary-sized message of n bits. Since MD5 uses a Merkle-Damgård construction, it follows Figure 2.1. It will thus partition the n bit message into smaller blocks of 512 bits. This is done by following a fairly common padding scheme, seen in the Merkle-Damgård family. It is done by always padding the message with a single set bit followed by a series of 0's until the message length = 448 mod 512. Thus in situations where the original message has a length of 448 mod 512, a 1 is followed by 511 bits of 0's. Lastly, a Merkle-Damgård strengthening is applied by appending a 64-bit representation of the message length mod 2^{64} to the padded message, resulting in every partition being 512 bits wide. Each partition of the message will then be fed into the compression `f` function in Figure 2.1.

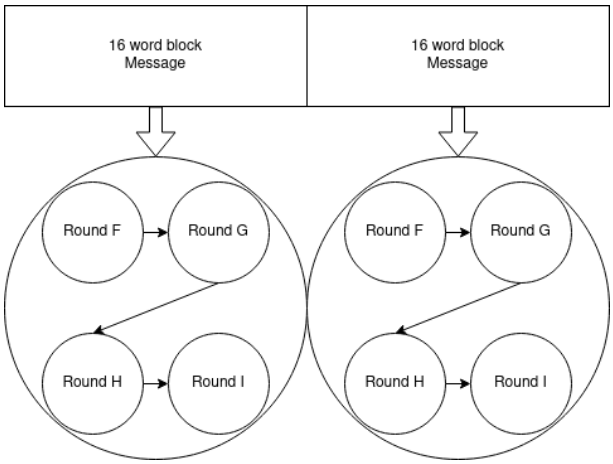


Figure 3.1: Rounds of MD5. Each of the two outer circles, is what corresponds to the f-function in Figure 2.1. Each of f function consists of a set of functions which will hash the (A,B,C,D) vector and forward it to the next round. The last round of an application of f will forward the vector to the next application of f on the next block of the message.

Figure 3.1 shows the expanded compression function `f`. `f` will modify a 128-bit initialization vector (A, B, C, D), with the initial value:
`[ A: 0x67542301, B: 0xefcdab89, C: 0x98badcfe, D: 0x10325476 ]`
`f` will use the following four functions, defined as such, to in "bitwise parallel" produce independent and unbiased bits in each of the rounds.

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z) \tag{3.1}$$
$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z) \tag{3.2}$$
$$H(X,Y,Z) = X \oplus Y \oplus Z \tag{3.3}$$
$$I(X,Y,Z) = Y \oplus (X \vee \neg Z) \tag{3.4}$$

In `f` a total number of 64 rounds will be computed, each of the functions 3.1-3.4 is applied a total of 16 times. Table 3.1 shows each of the specific rounds.

```
round 1  :: F
[ABCD  0  7  1] [DABC  1 12  2] [CDAB  2 17  3] [BCDA  3 22  4]
[ABCD  4  7  5] [DABC  5 12  6] [CDAB  6 17  7] [BCDA  7 22  8]
[ABCD  8  7  9] [DABC  9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12  7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
Round 2 :: G
[ABCD  1  5 17] [DABC  6  9 18] [CDAB 11 14 19] [BCDA  0 20 20]
[ABCD  5  5 21] [DABC 10  9 22] [CDAB 15 14 23] [BCDA  4 20 24]
[ABCD  9  5 25] [DABC 14  9 26] [CDAB  3 14 27] [BCDA  8 20 28]
[ABCD 13  5 29] [DABC  2  9 30] [CDAB  7 14 31] [BCDA 12 20 32]
Round 3 :: H
[ABCD  5  4 33] [DABC  8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD  1  4 37] [DABC  4 11 38] [CDAB  7 16 39] [BCDA 10 23 40]
[ABCD 13  4 41] [DABC  0 11 42] [CDAB  3 16 43] [BCDA  6 23 44]
[ABCD  9  4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA  2 23 48]
Round 4 :: I
[ABCD  0  6 49] [DABC  7 10 50] [CDAB 14 15 51] [BCDA  5 21 52]
[ABCD 12  6 53] [DABC  3 10 54] [CDAB 10 15 55] [BCDA  1 21 56]
[ABCD  8  6 57] [DABC 15 10 58] [CDAB  6 15 59] [BCDA 13 21 60]
[ABCD  4  6 61] [DABC 11 10 62] [CDAB  2 15 63] [BCDA  9 21 64]
```

Table 3.1: All rounds of a single MD5 iteration, where [abcd k s i] denotes $a = b+((a+round(b,c,d)+M[k]+K[i]) \lll s)$, and round denotes the function corresponding to one of the 4 functions corresponding to that round, M denotes the current 16-word buffer of the padded message and K[i] denotes $floor(2^{32} \cdot |sin(i+1)|)$.

When all rounds are completed, the new vector (A_1 , B_1 , C_1 , D_1) added to the vector from before the rounds will store the digest of that block. This digest will then serve as the initial vector for the next block of the message. We can thus see there is a Read After Write dependency (RAW) between the compression function on the block i depending on the result from the compression function on block i-1.

3.2 SHA256

SHA256 is a one-way Secure Hash Algorithm, which is where it gets its name from. It is part of the SHA2 family and was designed and published by the NSA. SHA256, like MD5, is based upon the Merkle-Damgård construction.
The 256 part refers to the output size of 256 bits. SHA256 can take input of any size (depending on the implementation) but works on chunks of 512 bits and then outputs a digest or hash of 256 bits. Other versions from the SHA2 family exist, like SHA512 and the truncated versions like SHA224 and SHA384. All of which are very similar.
The SHA256 routine can be expressed as some initialization and then 3 computation stages. All of which works on the message encoded in binary. All operations are also bitwise.
Firstly all constants and variables get initialized. All members of the SHA family use some preset constants for their initial round of calculations. SHA256 uses an array of size 64, K, consisting of the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers. The IV of SHA256 uses eight variables (A-H) with an initial value of the first 32 bits of the fractional parts of the square roots of the first 8 prime numbers. The IV get updated with each round and contain the final hash after the final round.
The message padding scheme is the same as that of MD5 with the exception of Big-Endian encoding of the message size at the end.
The first stage of the computation is to expand the message.
As mentioned the input block is of 512 bits, so sixteen 32 bit words. These gets extended

to 64 32 bit words. The extention of the input block works as follows:

$$W[i] = \begin{cases} M[i] & \text{for } 0 \leq i \leq 15 \\ \sigma_1(W_{i-2}) + W[i-16] + \sigma_0(W_{i-15}) + W[i-7] & \text{for } 16 \leq i \leq 63 \end{cases} \tag{3.5}$$

$$\sigma_0(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \ggg 3) \tag{3.6}$$

$$\sigma_1(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \ggg 10) \tag{3.7}$$

In the compression function we need some intermediate values Ch, Ma, Σ_0 and Σ_1.

$$Ch = (E \wedge F) \oplus ((\neg E) \wedge G) \tag{3.8}$$

$$Ma = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C) \tag{3.9}$$

$$\Sigma_0 = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22) \tag{3.10}$$

$$\Sigma_1 = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25) \tag{3.11}$$

Each round, as shown in Figure 3.2, is performed a total of 64 times, which constitutes the second stage. Lastly the third stage, which is addition of the IV before and after the rounds can be performed and the digest for a block is complete and fed as IV for the hashing of the next block until the entire message is hashed.

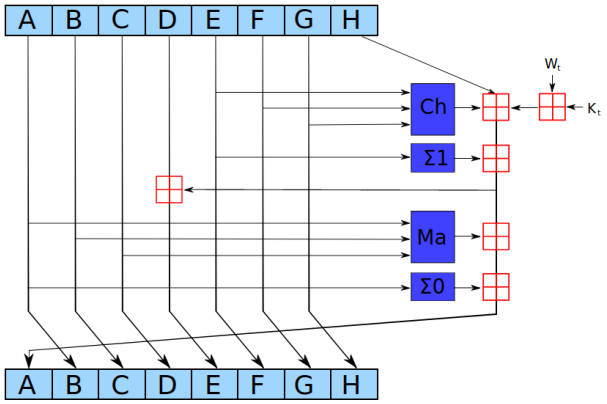


Figure 3.2: A SHA256 round. Shows how the IV (A-H) gets modified over a single round, where a squares with a cross denotes addition, W is shown in equation 3.5, Ch, Ma, Σ_0 and Σ_1 is presented in equations 3.8-3.11. K is the fractional of the ith cube root.[7]

3.3 AES

The Advanced Encryption Standard (AES) is a symmetric block cipher and specified as the standard for encryption by the National Institute of Standards and Technology (NIST). As AES is the standard for encryption, it is used chiefly everywhere and is critical to include in a cryptographic library. The algorithm behind AES is called Rijndahl and was chosen since it had a good balance of security, performance on a vast variety of devices[8]. Rijndael is a Substitution-permutation (SP) network that manipulates a block and key size of any multiple of 32 in the range 128-256 bits. In the exact specification of AES, the block size is fixed to 128 bits, where the key potentially can be 128, 192, or 256 bits. The 128 bits are arranged in a 4 x 4 column-major order matrix. As stated, AES is an SP network, meaning it is constructed as a series of rounds of substitutions and permutations. More precisely, the algorithm is listed as follows:

1. KeyExpansion: The key, whether it be 128, 192 or 256 bits is expanded using a keyschedule which will expand a key into the number of rounds + 1 keys. The schedule look as follows:

$$W_i \begin{cases} K_i & \text{if } i < N \\ W_{i-N} \oplus \text{SubWords}(W_{i-1} \lll 8)) \oplus \text{rcon}_{i/N} & \text{if } i \geq N \text{ and } i \equiv 0 (\text{mod } N) \\ W_{i-N} \oplus \text{SubWords}(W_{i-1}) & \text{if } i \geq N,\ N > 6, \text{ and } i \equiv 4 \text{ (mod } N) \\ W_{i-N} \oplus W_{i-1} & \text{otherwise} \end{cases}$$

where $i = 0...4 \cdot \text{rounds} - 1$, K_i is the i$^{\text{th}}$ 32 bit word of the original key. W is a 32-bit word of the expanded key. N is the number of words in the original key and subword and rcon being defined as follows:

$$\text{SubWord}([b_0 b_1 b_2 b_3]) = [S(b_0)S(b_1)S(b_2)S(b_3)] \tag{3.12}$$

With S being the Substitution box explained later for the `SubBytes` function.

i	1	2	3	4	5	6	7	8	9
rcon[1]	01	02	04	08	10	20	80	1b	36

2. The initial round-key is XORed with the plaintext.

3. SP - round: the rounds of the SP is performed by first doing a substitution which officially is called SubBytes[9], followed by the permutation, which consists of 2 functions `ShiftRows` and `MixColumns`, which will ensure the 4x4 matrix is permuted and diffused. Lastly, the round-key is XORed with the result. This is done 9, 11, or 13 times depending on whether the key size is 128, 192, or 256 bits, respectively.

4. The last round will work like the other, except it will only permute the rows and not the columns.

Subbytes is a non-linear byte substitution and is usually implemented as a lookup table. It is calculated in 2 steps first by taking the multiplicative inverse in the Galois field GF(2^8) followed by an affine transformation over GF(2):

$$b_i = b_i \oplus b_{(i+4)\%8} \oplus b_{(i+5)\%8} \oplus b_{(i+6)\%8} \oplus b_{(i+7)\%8} \oplus c_i$$

with b_i denoting the i$^{\text{th}}$ bit of the byte and c_i denoting the i$^{\text{th}}$ bit of 0x63. Since these and mostly every calculation in AES operates on Galois fields, we can be certain the cipher also will be 128 bits. The lookup table can be seen in Appendix C.

ShiftRows will transform the 4x4 input matrix by rotating the rows 0 to 3 bytes to the left, meaning the first row $\{b_0, b_4, b_8, b_{12}\}$ will not be rotated, the second row will be rotated one bit to the left, i.e. $\{b_5, b_9, b_{13}, b_1\}$ after the rotation. Likewise the 3rd row is shifted 2 and the last row is shifted 3 to the left (or 1 to the right). The transformation can be seen in Figure 3.3

MixColumns takes each column as a polynomial over the GF(2^8) and is multiplied (mod $x^4 + 1$,as it is a finite field) by $a(x) = 3x^3 + x^2 + x + 2$, which can be written as a matrix as:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

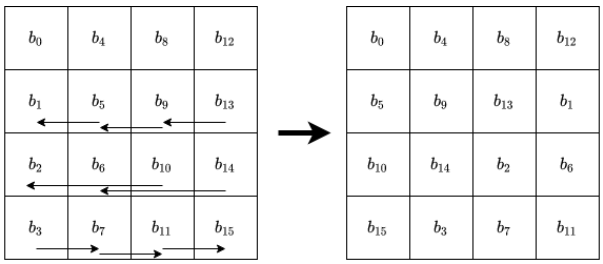


Figure 3.3: ShiftRows operation. Shows the rearrangement of the bytes in the matrix

Where c denotes the column, multiplication is as described above, and addition is XOR. The equivalent inverse functions can be used for decryption, as Rijndael is truly invertible, meaning an implementation in a reversible programming language would result in correct encryption or decryption based on whether the function was called or uncalled. The original paper for Rijndael[9] describes how these different steps can be implemented using lookup tables. This implementation can be realized on any 32-bit system with 4096 bits of memory, as we would need 4 lookup tables of 256 32-bit entries. That is one table for each column with all the 256 values in GF(2^8). The tables can simply be computed:

$$T_0[a] = \begin{bmatrix} S[a] \cdot 02_{16} \\ S[a] \\ S[a] \\ S[a] \cdot 03_{16} \end{bmatrix} T_1[a] = \begin{bmatrix} S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \\ S[a] \\ S[a] \end{bmatrix} T_2[a] = \begin{bmatrix} S[a] \\ S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \\ S[a] \end{bmatrix} T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \end{bmatrix} \tag{3.13}$$

these will then get used in a round transformation as

$$e_j = T_0[a_{0,3}] \oplus T_1[a_{1,2}] \oplus T_2[a_{2,1}] \oplus T_3[a_{3,0}] \oplus k_j \tag{3.14}$$

where $a_{x,y}$ denotes the byte in row x and column y and j is the round transformation. This approach are generally considered faster as it reduces each round to 16 lookups and 16 XORs compared to the normal approach where memory needs to be moved around. This is approach however is more prone to cache timing attacks and since the introduction of AES instruction set in 2010 this method is no longer the fastest on CPUs.

3.4 ChaCha20

ChaCha20 (From hereon just called ChaCha) is a stream cipher intended to be a fast and efficient standby cipher in case AES is compromised[10]. Unlike block ciphers, such as AES, which works on a fixed-sized block of text, stream ciphers work on a per-byte level. This is usually done by combining the plaintext with a pseudorandom stream of digits using XOR. Since the objective of the cipher is to generate a random stream, one first needs a seed. The seed of ChaCha is 16 32 bit words, laid out as such:
It might seem counter-intuitive that the seed would include a word, which holds the current block number. However, the result of each iteration of ChaCha20 will result in 16 words generated for the stream. Since the rest of the seed will stay the same for the entire encryption, the increasing block counter will ensure that no two “blocks” should result in the same cipher, and essentially including the CTR mode of operation for block ciphers into the streaming cipher. The confusion part of the algorithm follows a simple add-rotate-XOR (ARX) structure. Every round is based on only simple arithmetic add, left rotations, and XOR operations. More specifically ChaCha20 consist of 20 round of

expa	nd 3	2-by	te k
KEY	KEY	KEY	KEY
KEY	KEY	KEY	KEY
BC	NONCE	NONCE	NONCE

Figure 3.4: Seed of Chacha20. the layout of the seed is relatively simple and consists of 16 32-bit words that are constructed from 4 parts: A 4 word constant ”expand 32-byte k”, which is a classic case of a ”nothing up my sleeve number”.
A 256 bit key in little-endian order.
A word for the block counter (BC). This is sufficient for up to 256GB of plaintext.
A nonce which spans 3 words in little-endian.

which each consist of 4 quarter rounds will confuse 4 input words. Each quarter round looks as shown in Figure 3.5

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= d; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

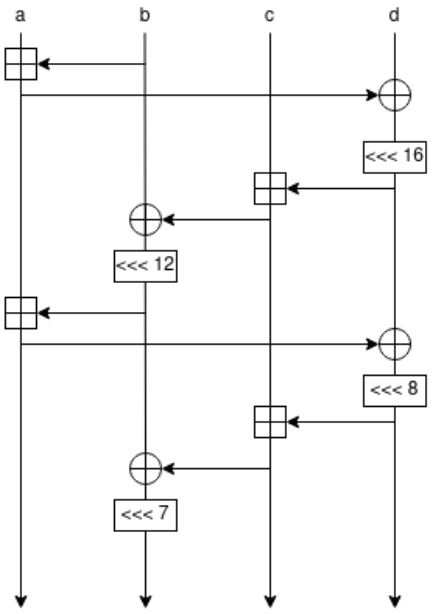


Figure 3.5: ChaCha20 Quarter Round. The left-hand side shows a QR in pseudocode, and the righthand side is a flow diagram of the same operations, where a square is addition, and a circle is XOR, and three arrows is left-shift.

ChaCha will perform 20 rounds consisting of 4 quarter rounds. For a quarter-round (QR) in an even-numbered round it will take a column of the seed as the input, an odd-numbered round will work on diagonals.

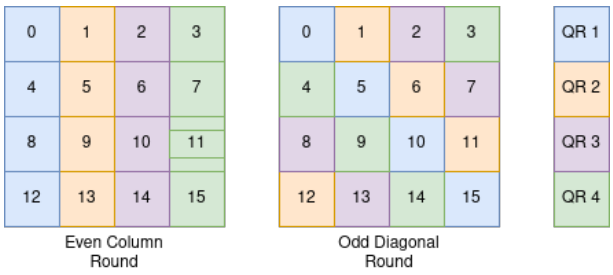


Figure 3.6: ChaCha20 Rounds. The 2 matrices shows how the ”seed” is distributed over the 4 QR’s of each round depending on its an even or an odd round. Such the 4 32-bit values colored blue is arguments for the same QR round, etc.

Lastly, when the 20 rounds have been computed, the initial seed and the modified version are index-wise added, giving a resulting block of 16 words of the stream, and this stream can then be XORed with the plaintext to get the plaintext cipher.

Chapter 4

Implementation

This chapter will go over the implementation-specific details of the algorithms described in the previous chapter. The implementation details will mainly be focused on the usage of SME and how we can structure the algorithms using the different components of SME, to get a working FPGA solution. We describe these implementations first by its naive model, followed by how we can structure a pipeline to improve the performance of each algorithm. We do not explain how the generated VHDL code gets routed to hardware, and we do not describe any interface to use the FPGA from an actual program.
We will be using the streaming AXI protocol[11] (from hereon, just called AXI) for all implementations. AXI is a lightweight protocol for synchronizing data transactions between hardware components. It specifies some standards for a ready/valid handshake between components. This means that whenever a Master component is ready (the valid flag is set) to send data over a bus, and the slave is ready to receive data simultaneously, a transaction is completed. This should ensure that the FPGA version should work with a potential callable interface.

4.1 MD5

4.1.1 naive

As explained in Section 2.2, SME consists of busses and processes. We can define the MD5 algorithm naively using 4 busses and one simple process.

The compression function itself is wholly contained in the single clocked process and works as described in 3.1. The significant difference comes in the data flow. Since our program will be mapped to hardware, we cannot have variable sizes; everything must be static. Thus we have opted for an approach that will receive 512 bits, corresponding to a single message block of the entire message, over the `Message` which we describe later. The process will both handle the padding and the compression and thus stand as an "independent" bus having no dependency on external computations.
For the bus interface between the simulation process and the MD5 process, we considered two overall approaches, Firstly, one could have 2 input buses to the process, one which would contain the message and one which would contain the IV to modify. However, we find this approach unnecessary as the initialization vector is fixed for every hash, thus the alternative. Since we use the C# implementation of SME, we can easily store the Digest locally inside the process as a field. We will only require a single data bus with the message. We can define the Message bus as such:

```
public interface IMessage : IBus {
    [InitialValue(false)] bool Valid { get; set; }
    [FixedArrayLength(MAX_BUFFER_SIZE)]
    IFixedArray<byte> Message { get; set; }
    int BufferSize { get; set; }
    int MessageSize { get; set; }
    [InitialValue(true)] bool Last { get; set; }
    [InitialValue(true)] bool Head { get; set; }
    [InitialValue(false)] bool Set { get; set; }
}
```

A byte array `Message` is used to store the message block itself. `BufferSize` will be updated for every iteration or tick, and denotes how many values in the buffer are set, essentially flag for when the message should be padded. `MessageSize` will be set in

the initial tick and denote the length of the entire message used for the Merkle-Damgård strengthening. The last 3 flags are used to handle some "edge-cases". Head Denotes that the initialization vector should be reconstructed. Last is used to denote when a block is the last in the message. The block cannot be filled with more than 447 bits. Set is used when the initial 1 should be set but where the block is not the last in the message, for instance, when the length of the message is 448.
Since we also need to receive the digest from the process, we also need an output bus. This bus is, however, reasonably uncomplicated. It only consists of a Valid flag and the Hash as an array of 4 32-bit words.
Lastly, we want 2 additional busses to make our design comply with the AXI protocol. This bus will be the most basic of all busses and contain only a single flag to show if the process is ready to receive data. It will thus be wired such that the MD5 process will have an in-going bus to know when it is safe to send the digest to the simulation and an out-going ready bus to let the simulation know when to send the message values.

4.1.2 First optimization approach

To make the algorithm more efficient, the length of the circuit produced in the VHDL code should be reduced. Meaning we want the simple process to do less. For the initial approach, we can notice that the compression function in MD5 works in rounds. Figure 4.1 shows how the hash function as a whole up can be split up into 5 smaller processes and build a pipeline from this. One process for message formatting and one for each of the 4 rounds. In our actual implementation, we further added 2 processes:

- A message-converter process between the message formatting and `round 1 (f)` to convert the message from bytes to unsigned integers.
- A combiner that does the last addition of the two vectors.

Each of these processes should be clocked and thus run concurrently on each clock tick, and the results of each process will be sent to the next.

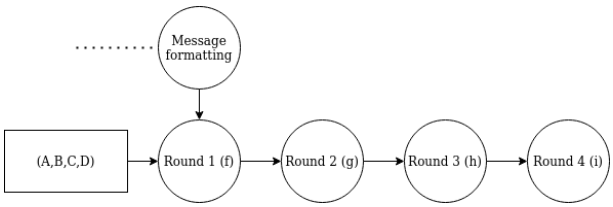


Figure 4.1: Highlevel overview of the MD5 pipeline. Each circle is a process and an arrow can be seen as a data bus. The downwards facing arrow carries the initial message and the leftmost arrow is simply (A,B,C,D). All arrows going out from a round carries both (A,B,C,D) and the block.

One problem we have faced with this general approach is that MD5 is embarrassingly sequential because of the RAW dependency, as described in section 3.1, is that we need a bus between the process which calculates the 16 first rounds (F) and the `combinator` process. As of now, we have not sorted the logic of this "wiring" out. Hence our pipelined version is only able to calculate messages of <448 bits. A bit of a hack that could be implemented easily would be to extend the Top Level Bus (the bus which is exposed to other devices) with a vector (A,B,C,D). Thus one could simply chain the computations together. In any case, a RAW dependency will create a stall in the concurrent pipeline, as seen in Table 4.1. Such a long stall has a big difference in throughput, which will be elaborated in section 5.1.

Independent message blocks									
clock	0	1	2	3	4	5	6	7	8
	P_1	M_1	F_1	G_1	H_1	I_1	C_1		
		P_2	M_2	F_2	G_2	H_2	I_2	C_2	

Dependent message blocks												
clock	0	1	2	3	4	5	6	7	8	9	10	11
	P_1	M_1	F_1	G_1	H_1	I_1	C_1					
		P_2	M_2	-	-	-	-	F_2	G_2	H_2	I_2	C_2

Table 4.1: Shows how the proposed MD5 pipeline would work in case of smaller messages (independence) and larger messages (dependence), where P are the padding process, M is the formatter, F, G, H, I is each of the rounds and C is the last vector addition.

4.1.3 Further optimizations

Because the initial optimization approach showed good promise, as shown in Section 5.1 we wanted to see how well a deeper pipeline performed. Hence we further chose to make versions that propagated the rounds into 8, 16, 32, and 64 processes. The results of this can be seen in Section 5.1. As of now, each of our pipelined versions only work on messages <448 bits. Since we have not been able to fix the additional signal wirering.

4.2 SHA256

The implementation of SHA256 is very similar to that of the implementation of MD5, since they both are hashing algorithms of the Merkle-Damgård construction. The significant difference is that SHA uses big-endian encodings while MD5 uses little-endian.

4.2.1 Naive

The naive unoptimized SHA256 implementation upholds the same general structure in the code, including the padding and block fetching as the naive MD5. The only difference is that the output digest is 256 bits, so an array of eight 32 bit unsigned integers, and the format is changed from little-endian to big-endian in the padding and fetch block routines.
So all the busses are set the same and functions the same as in MD5. The exception of the array in the `Digest` output bus is extended to contain a 256-bit hash.
The SHA256 algorithm is implemented as described in Section 3.2. That is the `OnTick` function will first pad the input buffer, followed by an expansion of the function from 512 bits to 2048 bits and lastly computing the 64 rounds.
The calculations for ch, maj, s1, and s0 could be their own functions, but they have been left as written out since they are simple and shouldn't significantly affect performance.

4.2.2 Optimizations

Since SHA256 is so similar to MD5, we can apply the same pipeline approaches as we did for MD5. One difference, however, is the block expansion, which likewise can be pipelined. Thus we should theoretically be able to create an even deeper pipeline for SHA256. Because the target FPGA runs out of resources even before pipelining the

rounds, we have not looked into this. The problem and the resource usage will be explained further in Section 5.2.

4.3 AES

4.3.1 Naive

Just like for the other algorithms, AES can naively be implemented as a single simple process. That is, we could implement the SME implementation of AES as a single process that can do both encryption and decryption and then have some checks in the `OnTick` function. This, however, poses some unwanted effects. First, we add unnecessary complications to the process as it would have to multiple things at once. Furthermore, and more importantly, a combined encrypter/decrypter reduces the utilization of the library, since not all (and the most useful) modes of operations need a specific decryption function as shown in Section 2.3.2 and Appendix D. Thus in a hypothetical scenario where the design includes both encryption and decryption might take up 40 pct. of an FPGA, and a design with only encryption would take up 20 pct. It is clear to see how many resources are wasted. Thus we have decided that for the base case implementation, encryption and decryption should be separate processes. (For good measure, we implemented it anyways, and it could not even fit on the target board). We will only go over the implementation of encryption as the process for decryption is the exact inverse computationally as described in section 3.3 and the structure thus follows symmetrically. For the design, a single bus with four fields, as seen below, suffices. It consists of two Valid flags, which work similarly to the one described for MD5. Furthermore, there are two-byte arrays with the size of `BLOCK_SIZE` = 128 as this implementation is a 128-bit key AES. We have one array for storing the data and one for storing the key. Once again, we do not want to make the process itself flexible with multiple AES versions as it will reduce the resource utilization on an FPGA. This is because the optionality of additional rounds for a 256-bit key version would map these extra computations to hardware, making the circuit more complex and more demanding for Vivado to route the design and have a harder time meeting the timing constraints. In Section 5.3 we will take a brief look at this. If 128 key encryption suffices, the overhead from including the four extra rounds for 256 is wasteful.
Notice furthermore, the bus is named IPlainText but could just as well have been called IData or something similar as the same bus can be used for both the plaintext and the cipher as the algorithm is symmetrical. However, for the output bus, we do not have to output the key, assuming the result is sent back to the device that called the function.

```
public interface IPlainText : IBus {
    [InitialValue(false)]
    bool ValidKey { get; set; }
    [InitialValue(false)]
    bool ValidData { get; set; }
    [FixedArrayLength(BLOCK_SIZE)]
    IFixedArray<byte> Key { get; set; }
    [FixedArrayLength(BLOCK_SIZE)]
    IFixedArray<byte> Data { get; set; }
}
```

For the actual AES process, we have decided to only support full blocks of 128 bits because of the lack of a padding standard. We have decided to implement both the classic algorithm and the T-box approach described earlier to see if there actually was a significant performance difference.

4.3.2 Optimisation 1

AES has shown to be quite fast even in the naive implementation (see Section 5.3); however, it should still be a reasonably slow approach compared to a pipelined solution, as long as the FPGA can handle the additional logic. Exactly as the other algorithms. We notice that AES likewise uses rounds, and the single naive process from before can be divided such that each round of AES can be contained in its own process. This is easily implemented compared to MD5 and SHA, which have a round dependency. This approach initially seems to have a flaw that might underutilize the LUT's of the FPGA, leading to worse performance. Section 2.2 described how there is no sharing between processes, and henceforth they can not have access to the same lookup tables/T-boxes. Corollary, every process that needs access to one of the lookup tables will have to have it defined for itself. Arguably this is a wasteful approach as the size of lookup tables will be 36 KB instead of 4 KB. This should then be more LUTs used, leading to harder routing, which results in lower performance. This showed not to be a problem (elaborated in Section 5.3). Another approach we tried was to use Block RAM (BRAM), which is enabled through `Components` in SME. The initial idea was to enable 8 of these in each process, such that we would have two copies of each of the T boxes, one in each BRAM. We need two copies of each because using a True Dual Port Memory component only allows for two lookups at a time.

4.4 ChaCha

4.4.1 Naive

Just like AES, ChaCha will work in two phases. The initial phase will be to set up the seed. After the initial setup, the only modification to the initial seed will be the block counter. We will thus have a similar bus to that of AES. The only difference is that we also need to give the nonce with the input bus. Every iteration will perform ChaCha described in 3.4. Like the other algorithms, we have opted for a total solution, meaning the FPGA solution should be as independent as possible. Such that our chacha20 version will not merely produce the keystream but will produce the cipher itself. Thus the input bus should look like this:

```
public interface IState : IBus {
    [InitialValue(false)] bool ValidSeed { get; set; }
    [InitialValue(false)] bool ValidT { get; set; }
    uint Nonce0   { get; set; }
    uint Nonce1   { get; set; }
    uint Nonce2   { get; set; }
    [FixedArrayLength(BLOCK_SIZE)]
    IFixedArray<uint> Key { get; set; }
    [FixedArrayLength(TEXT_SIZE)]
    IFixedArray<byte> Text { get; set; }
}
```

Where `BLOCK_SIZE = 16` and `TEXT_SIZE = 64`.

4.4.2 First optimization

ChaCha encourages concurrency and parallelism, as each chunk of the keystream can be computed entirely independently of each other in a similar fashion to AES. Thus a pipelined version is easy to implement compared to the has functions. We can very easily split the 20 rounds into 20 processes, and even if the FPGAs limitations allow for it, we can reduce down to each quarter round (QR) to its own process.

Chapter 5

Results

This chapter will go over the performance of the implementations described in the previous chapter. The main focus for our benchmarks has been Throughput and power usage. For comparisons, we settled on similar CPU implementations. The different implementations have been tested against the C# standard library equivalent algorithms and ensure that the results are correct. The only exception for this `ChaCha` as it still is a quite uncommon cipher. All implementations is synthesized and implemented using Xilinx Vivado on a Zynq Zedboard, which is a low-end FPGA. For comparisons, we have chosen to include different implementations, in C, C#, and OpenSSL, using `openssl speed -evp "algorithm"`. Unfortunately, we have not been able to get our hands on the board in time, and we stand with some limitations on the benchmarking results. The reported frequency is the results synthesizing and doing place & route on the design through Xilinx Vivado. For comparisons, we settled for a Raspberry pi 4B. The reason being, this having a low-end processor similar to the one on the Zedboard, a Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz. Because of promising results, we further compare our results with an Intel i5-7500. For specific details about waht hardware and software is used see Appendix B.

5.1 MD5

5.1.1 Throughput

Version	f_{max}(Mhz)	$clocks_{hi}$	$TP(MBps)_{hi}$	$clocks_{lo}$	$TP(MBps)_{lo}$	LUT	FF
Naive	2.38	b	152	b	152	11607	2304
$Proc_4$	9.50	hi(6)	266	lo(6)	101	10247	5226
$Proc_8$	19.00	hi(10)	532	lo(10)	122	10087	7538
$Proc_{16}$	33.50	hi(18)	937	lo(18)	119	10206	12162
$Proc_{32}$	65.00	hi(34)	1817	lo(34)	123	10149	21347
$Proc_{64}$	115.00	hi(66)	3209	lo(66)	112	10350	39718

Table 5.1: Performance and statistics over the different MD5 implementations. f_{max} is the clock rate reported from Vivado. Clocks describe how many clock cycles it takes to calculate `b` blocks, where $hi(x) = x+2{\cdot}blocks$ and $lo(x) = 2+6{\cdot}blocks$ describe a best and worst-case scenario, respectively. These are calculated by a schema as in Table 4.1. The throughput (TP) is calculated as $(b_{bits} \cdot f_{max})/(clocks \cdot 8)$. LUT is the number of Look-Up Tables used in the design. FF is the reported amount of Flip Flops used. $Proc_i$ denotes how many i processes the 64 rounds are distributed over.

As can be seen in Table 5.1, there is a monumental difference between the naive and the pipelined versions. Even the most simple of the pipelines has a 74.6 pct increase over the naive version and the highest performing version, which calculates only a single round in each process more than 20 times faster than the naive version. This comes at the cost of a lot more Flip-Flops but with slightly fewer LUTs. It is, however, quite remarkable that such performance increases are achievable without doing specific FPGA optimizations as such but simply applying pipelining. Especially one aspect has been surprising to see. To keep track of the input block, each process simply forwards it from

its input bus to its output bus. Thus one would assume this computation would take a relatively long time compared to calculating a single round value, but this has not been the case. The reason might be because this is optimized away by Vivado. However, one aspect to be aware of is that the pipelined versions actually "theoretically" perform worse than the naive version. The worst case scenario happens when the string to hash is very long as the process thus becomes inherently sequential. Thus the time saved from pipelining will amount to simply 2 cycles, and since the pipelined versions have additional computations, signals, etc., that need to be routed, these will not be as fast as the naive version, when comparing throughput. Thus if the user need to hash long inputs instead of many short ones, the naive version would be preferable. Comparing

Version	Naive	Proc_{64}	C#	C	OpenSLL_{low}	OpenSLL_{high}
Pi	152	3210	287	256	42	293
i5	152	3210	604	622	81	691

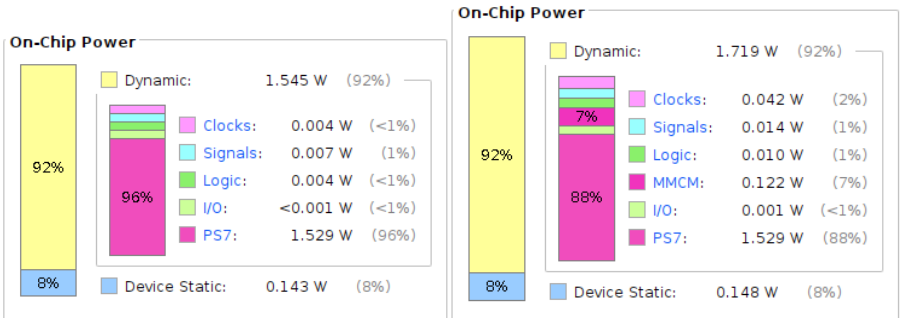
Table 5.2: Performance comparison of the worst and best MD5 FPGA implementations and the various CPU versions. The C# uses the 0`System.Security.Cryptography.MD5`, the C version is our implementation and is optimized with `-O3`. The OpenSSL is from `openssl speed -evp md5`, where high and low corresponds to the lowest reported throughput and high is the highest reported throughput.

the implementations to the CPU versions, the naive only perform quite poorly with the C version on the Pi. Likewise, it seems to beat $\text{OpenSSL}_{\text{low}}$ by quite a margin. The $\text{OpenSSL}_{\text{low}}$ is the worst utilization of the `openssl speed`, which happens on message sizes of 16 bytes. Compared to the worst utilization of OpenSSL, this is a speedup of more than 300 pct. One should keep in mind that OpenSSL only works on inputs of 16 bytes, which is not nearly enough to fill a block, and thus entire blocks of data are not processed, meaning there is a significant spill. Even when running the same benchmarks on the i5, 16 bytes are merely 81 MB/s. Thus to get the full utilization, we should focus the attention on 256-byte blocks or higher, as the 64-byte blocks will have a round of "wasteful" computation as this block is purely padding and not part of the message size. Compared to all the other CPU versions, the naive version performs poorly, especially on the i5, where most versions are around 4 times faster. On the other hand, the best-performing pipeline outperforms all the CPU versions by a significant amount by at least 4.6 times. This is a significant increase in speed, emphasizing how well an FPGA can perform if designed correctly. However, the Throughput of the optimal pipeline will only serve as the theoretical maximum, and in a real-world example, such speed might not be observable depending on how fast the host can provide the data for the FPGA. One thing to note is that according to the specifications for the Xilinx Zedboard[12], then we see that the bus size is 32-bit and the memory controller operates at 533 MHz, so on a perfectly utilized Zedboard we would get a throughput of $533 \cdot 32 = 17056$ Mbps = 2132 MBps, so CPU speeds are theoretically reachable for specific usecases. It however also means the highest perfoming version is bottlenecked by the memory bus.

5.1.2 Power Consumptions

From the previous section, we showed that our FPGA solution could outperform not only low-end CPUs but also mid-end CPUs by quite a margin. But, not only is the FPGA able to achieve high throughput it also does it at very low power consumption. Figure 5.1 shows the TDP power consumption as reported by Vivado. The power consumption of the naive version sums up to 0.016 watts without including the processing system,

which is almost 11 times less than the optimized version using 0.189 watts. Thus, we can see that one needs only 11 times as much power to get a speed increase of 20 times. Interestingly the most of the power usage can be attributed to the clocking. In any case, we can assume this to be very power efficient compared to the power used by a CPU, since the Pi has a TDP of 7.5 W and the i5 has an TDP of 65 W. We can further see the processing system (PS7) uses at least 88 percent of the power.



(a) Naive version (b) Proc_{64} version

Figure 5.1: Powerconsumption of MD5 designs. Note that this is just the TDP and not the actual power usage. To mesure the actual power usage we would need the physical board.

5.1.3 Takeaways

The takeaway from these results is that if one needs to hash long strings, one might as well choose a CPU implementation over our versions, and is one to use our implementations for such one should use the naive version, whereas the pipelined version will be a lot faster (or at least not be the bottleneck). If the power consumption by the computations, on the other hand, is more important than throughput, our naive version is a good alternative over the other implementations presented in the previous sections.

5.2 SHA256

5.2.1 Throughput

Looking at the statistics in Table 5.3 one can see there is no real difference in Mhz between Proc_4 and Proc_8. This, in turn, means there is a negligible performance difference in a best-case scenario, where many small hashes are calculated. On the other hand, the performance in a worst-case scenario is in favor of Proc_4, since the pipeline of Proc_8 is longer and provides no additional speed because of the serialness. Thus in this case the pipeline Proc_4 should be considered over any deeper pipeline for boards of similar size to the Zynq, in cases where there are dependencies. With further analysis in vivado to see which parts of the program is the costly bit the solution could potentially be improved by reducing this cost. We assume the cost happens in the expanding of the message, and that this is reason SHA cannot meet the same magnitude of improvement as MD5. This isalso reflected by the high usage of LUTs. Compared to MD5, SHA uses around 2.3 times as many LUTs (24466/10350), which makes the routing of the design harder. The additional LUT usage comes from the expanding of the input block from 512 bits to 2048 bits. In any case, the pipelined versions do perform better than the naive version in an optimal scenario, by a 66.5 pct. However, in a worst-case scenario,

Version	f_{max}(Mhz)	$clocks_{hi}$	$TP(MBps)_{hi}$	$clocks_{lo}$	$TP(MBps)_{lo}$	LUT	FF
Naive	2.1	b	134.4	b	134.4	24330	2560
$Proc_4$	8.0	hi(6)	223.9	lo(6)	85.3	24466	8938
$Proc_8$	8.0	hi(10)	223.8	lo(10)	51.2	24756	14066

Table 5.3: Performance and statistics over the different SHA implementations. f_{max} is the clock rate reported from Vivado. Clocks describe how many clock cycles it takes to calculate `b` blocks, where $hi(x) = x + 2 \cdot blocks$ and $lo(x) = 2 + 6 \cdot blocks$ describe a best and worst-case scenario, respectively and is calculated by a similar pipelining schema as MD5. The throughput (TP) is calculated as $(b_{bits} \cdot f_{max})/(clocks \cdot 8)$. LUT is the number of Look-Up Tables used in the design. FF is the reported amount of Flip Flops used. $Proc_i$ denotes how many `i` processes the 64 rounds are distributed over.

the naive version is 55.6 pct faster than $Proc_4$. Thus just as for MD5, the best-suited version will be application-specific. Just as with MD5, the throughput of the naive SHA

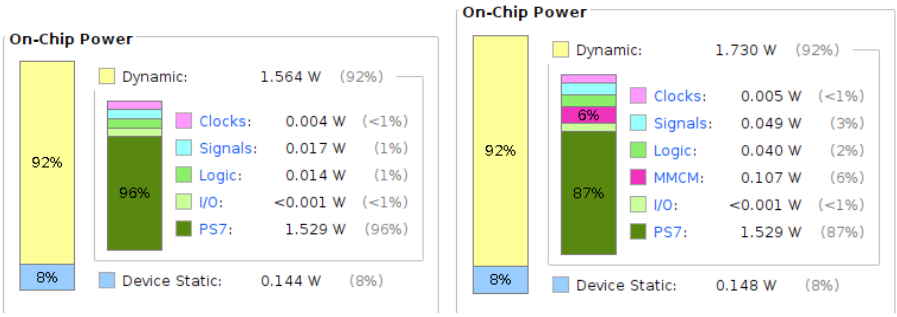
Version	Naive	$Proc_4$	C#	$OpenSSL_{low}$	$OpenSSL_{high}$
Pi	134	224	163	42	165
i5	134	224	438	61	461

Table 5.4: Performance comparison of the worst and best SHA FPGA implementations and the various CPU versions. The C# uses the `System.Security.Cryptography.SHA256`. The OpenSSL is from `openssl speed -evp sha256`, where high and low corresponds to the lowest reported throughput and high is the highest reported throughput.

version is slower than the CPU versions. For the naive SHA version, the naive version is only 30 MBps slower than the CPU versions on the Raspberry Pi, whereas MD5 was only half as fast (difference of 141 MBps). Likewise, the pipelined version performs reasonably well compare to the Pi, with an overall increase of 35.7 pct. Comparing to the i5, the results are not as promising. The pipelined version is only about half as fast as the CPU versions.

5.2.2 Power Consumptions

Figure 5.2 shows the power consumption of the naive version and $Proc_4$. Once again, the results are very similar to those of MD5. The SHA version consumes only about 0.02 watts more than the naive MD5 version. This additional cost is attributed to additional signals and more logic required to compute SHA. Similarly, the pipelined version is a little more power-hungry than MD5. The thing to remember is that the throughput does not follow the same trend. Interestingly, most of the power consumption of the pipelined MD5 goes towards the clocks, whereas the power consumed by SHA is mainly placed in the computations. This might be because the pipeline is far deeper in MD5. The actual power usage for the two SHA versions is 0.036 watts for the naive version and 0.202 watts for $Proc_4$, meaning the $Proc_4$ uses 5.6 times more power, but it only gives a 2-time speedup. Hence, the naive version is more power-efficient than the pipelined version.



(a) Naive version (b) Proc_4 version

Figure 5.2: Powerconsumption of SHA designs

5.2.3 Takeaways

The overall results for SHA are inferior to MD5. The pipelined version could compete with the Raspberry Pi but is still slower than an i5 processor. If power consumption is of concern, the naive version should always be chosen over the pipelined as the throughput increase does not match the increase in power usage.

5.3 AES

5.3.1 Throughput

Version	f_{max}(Mhz)	clocks	TP(MBps)	LUT	FF	BRAM
Naive	22	b	352	10612	3195	0
TBox	25	b	400	16458	3195	0
Proc_4	68	$C(3)$	544	16474	2817	0
Proc_{11}	208	$C(12)$	1663	15659	4383	0
Proc_{22}	217	$C(24)$	1662	15454	7401	0
BRAM_{11}	195	$C(31)$	1556	10012	10398	72

Table 5.5: Performance and statistics over the different AES implementations. f_{max} is the clock rate reported from Vivado. Clocks describe how many clock cycles it takes to calculate b blocks, where $C(x) = x + 2 \cdot blocks$, since there is no dependency high and low should be the same. The throughput (TP) is calculated as $(b_{bits} \cdot f_{max})/(clocks \cdot 8)$. LUT is the number of Look-Up Tables used in the design. FF is the reported amount of Flip Flops used. Proc_i denotes how many i processes AES is distributed over.

From the different implementations as shown in Table 5.5 one can see quite interesting results. First and foremost, the TBox approach has merely 48 MBps higher throughput than the naive approach, and they each have their pros and cons. The naive approach uses less power and uses fewer LUTs but at a little lower throughput. In Section 4.3.2 we described how we would assume the reasonably large lookup tables might be using up quite a lot of the FPGAs LUTs. This is, however, not the case as the naive version is using only 16500 LUTs comparing to the SHA256's 24300. Even more interesting is that splitting the calculations up into different processes does not increase the LUT usage.

This suggests that the Vivado synthesizer recognizes the arrays from each process to be equivalent Read-Only Memory and thus can optimize it to a single table. Against the assumptions, this produces good results without having to use BRAM specifically. By making each round of AES its process, we get a four-fold increase in throughput from the naive TBox approach. We get no further improvement from reducing each process to only a half-round, suggesting that the overhead from signals is becoming the bottleneck. Furthermore we can see the BRAM version is actually slower than our pipelined version which simply uses LUTs. We might not use the BRAM correctly as all the articles we have found using the TBOX approach for FPGAs uses BRAM. Hence it suggests that using the more intrinsic parts of FPGA programming is not as straight forward even with a high level model.
In Section 4.3.1 we described how we rejected to make a solution that was flexible in its key size. The results shown in Table 5.5 hint that this has a good impact on the performance. Comparing our solution to the solution presented in the SME GitHub repository[1], which is more flexible in the key size, our solution outperforms this by a factor of 1.66, as it is reported to have a throughput of 1.92Gbps(240MB/s)[1]. This shows that we can trade off some flexibility for a significant speedup.

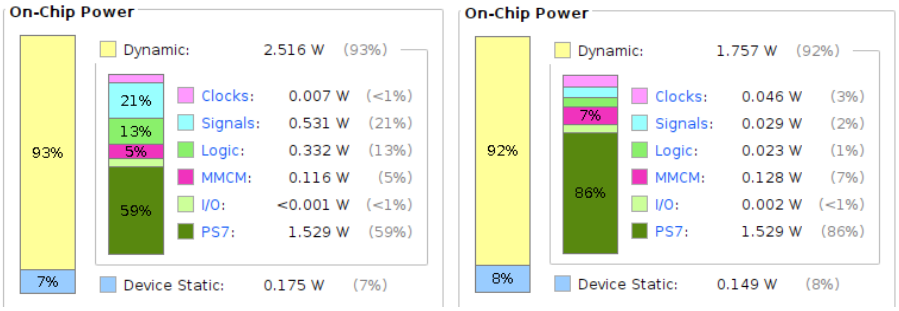
Version	Naive	Proc_{11}	C#	C	OpenSLL_{low}	OpenSLL_{high}
Pi	400	1963	70	198	72	89
i5	400	1963	1699	340	847	5722

Table 5.6: Performance comparison of the worst and best AES FPGA implementations and the various CPU versions. The OpenSSL is from `openssl speed -evp aes-128-ecb`,

The results of AES are interesting compared to our other implementations in the sense that even the naive FPGA version is outperforming the CPU on the Pi. One can notice that our naive version has a throughput of 400 MB/s which is around 4.49 times as much as OpenSSL on its peak performance and that it likewise outperforms C# and our C-version with 5.7 and 6.2 times, respectively. The i5, on the other hand, is quite a lot faster than the naive implementation, where only our C version is slower. The pipelined version Proc_{11} is almost as fast as the C# version and faster than OpenSSL at its lowest performance. However, OpenSSL at its full capacity still has around 3.4 times higher throughput. These results emphasize the results presented from the previous sections, that an FPGA is faster at performing specific tasks than a CPU and shows how an ASIC, such as the AES-NI device in intel CPUs, is even better doing a specific task than an FPGA. One aspect to consider is that a Zynq board and an i5-7500 have about the same cost[13][14]. Hence a Zynq is cheaper than the i5 since it also needs all the other computer components, such as Motherboard, RAM, etc. All things considered, our solution is not achieving the hoped throughput, as Xilinx own AES[15] should be able to run at 16GBps on boards with a cost of around 1100 USD[16] whereas an AMD ThreadRipper 3970X (2600 USD[17]) can only achieve 12.9GBps [18]. Thus we would assume it to be possible for an FPGA to be faster than even an ASIC accelerate CPU of equivalent price. There might, however, be some difference in specific architecture for FPGAs in different price ranges that might disallow such generalization, and the AES version provided by Xilinx is most definitely developed by a larger team with more experience than us. Still, our version to be more than three times as slow is not ideal.

5.3.2 Power Consumptions

Figure 5.3 shows the power usage of the TBox version vs. the power usage of the fastest pipelined version Proc_{11}, which shows that the naive Tbox version is using a lot more power than the Proc_{11}. Thus we can easily say Proc_{11} is to prefer over the Tbox version. It is, however, possible that there is something wrong with the reported power consumption (or at least we cannot pinpoint why it is so high) since the power usage of Proc_{11} is more closely related to the power usage of MD5 and SHA, whereas the Tbox version has an abnormally high power usage.



(a) TBox version (b) Proc_{11} version

Figure 5.3: Powerconsumption of AES designs

5.3.3 Takeaways

Just like for MD5, we have been able to gain some good performance improvement from pipelining. However, the improvements have not been significant enough to compete with CPUs with AES-NI. Interestingly enough, it seems like the pipelined version is more power-efficient and uses less power and hence should always be preferred over the naive Tbox version.

5.4 ChaCha20

Version	f_{max}(Mhz)	clocks	TP(MBps)	LUT	FF
Naive	1.25	b	80	14670	3457
Proc_{11}	40.00	$C(9)$	1279	14736	16898
Proc_{22}	82.00	$C(20)$	2557	17565	32420

Table 5.7: Performance and statistics over the different ChaCha implementations. f_{max} is the clock rate reported from Vivado. Clocks describe how many clock cycles it takes to calculate b blocks, where $C(x) = x + 2 \cdot blocks$. The throughput (TP) is calculated as $(b_{bits} \cdot f_{max})/(clocks \cdot 8)$. LUT is the number of Look-Up Tables used in the design. FF is the reported amount of Flip Flops used. Proc_i denotes how many i processes ChaCha is distributed over.

One would expect ChaCha to perform well because of the simplicity in computation, the fact it was designed for speed, and that the OpenSSL as reported in Table 5.8 is more than three times faster than AES as shown in Table 5.6 when run on the Rasberry Pi (meaning it is faster when ASIC acceleration is not applied). Despite this, the naive

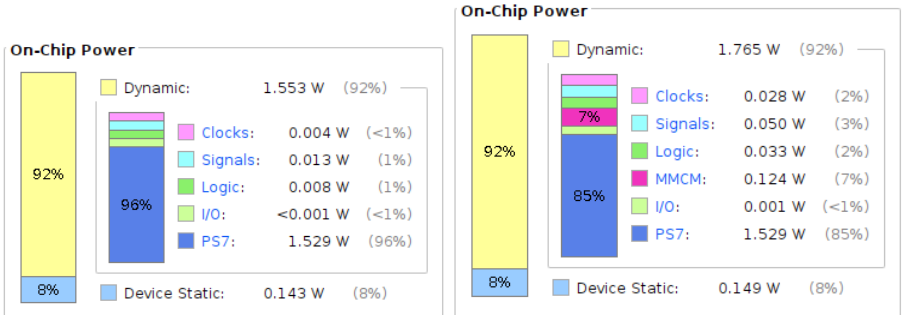
ChaCha has an abysmal throughput of 80 MBps. However, when pipelining ChaCha, the performance increase to expected levels. Splitting the rounds into 11 processes (one process for an even and an odd-numbered round along with an XOR round), we can increase the throughput by 16 times. Even further splitting of the processes can yield a throughput of 2557 MBps. Comparing this to the OpenSSL results as shown in Table 5.8, one can see it is still not quite enough to beat the throughput of an i5 but is far faster than a PI, but once again this would not be possibly because of the memory bottleneck of 2132MBps.

Version	Naive	Proc	OpenSLL_{low}	OpenSLL_{high}
Pi	80	2557	84	307
i5	80	2557	388	3092

Table 5.8: Performance comparison of the worst and best ChaCha FPGA implementations and the various CPU versions. The OpenSSL is from `openssl speed -evp chacha20`, where high and low corresponds to the lowest reported throughput and high is the highest reported throughput.

5.4.1 Power Consumptions

Figure 5.4 shows similarity to MD5 and SHA256. That is, the power is lower for the naive version is lower by using only 0.026 watts compared to Proc_{22}, which uses 236 watts. Thus the power usage is nine times higher, whereas the throughput difference is around 32 times, meaning Proc_{22} is far more power-efficient but at a higher cost. Nevertheless, once again, the power is lower than for common CPUs as described in Section 5.1.2.



(a) Naive (b) Proc_{22} version

Figure 5.4: Powerconsumption of ChaCha designs

5.4.2 Takeaways

From pipelining, we can significantly increase the performance of ChaCha, but not enough to perform as well as a mid-range CPU, and thus if speed is critical, one might be better of with a CPU, whereas the FPGA still can provide decent throughput at a lower cost.

Chapter 6

Discussion

We will, in this section, present some reflections on the tools used and listing work to be done in the future, and showing an example of some further improvements that can be implemented.

6.1 SME as a tool for Software developers

SME has worked well as a tool for FPGA development. As shown in the previous section shown our implementations use few resources, and we have been able to gain some significant speedup that can compete with modern CPU alternatives without knowing how to actually write in a HDL. VHDL being one such hardware description language makes it an abstraction to physically connecting signals together. Abstractions like this are usually created to make tasks easier for humans to write and read, but since VHDL is a "first-level" abstraction from hardware, it can be difficult for untrained people to use. In many ways, it is similar to how C for CPU programming in that the user has a lot more control over every little aspect of the program at the cost of additional logic that the developer has to take into account. In a similar mindset, SME is another level of abstraction from VHDL as C#, for instance, would be to C. Since SME is a framework for C#, it should be easier for a lot of software programmers to use.

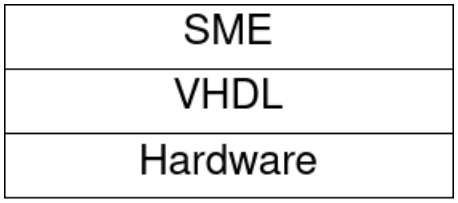


Figure 6.1: Levels of hardware abstraction, note that SME is the top level.

However, one thing to be aware of is that SME needs to compile to an HDL, and hence using SME must necessarily follow the same quirks, such as everything to have a fixed size. As shown, busses, etc., must be user specified. If the data flow of a program is complex, it will still be using SME, but it is comparatively not very much. Besides the dataflow, SME is definitely a lot easier to use than an HDL since one simply writes (a restricted subset) C# instead of defining specific types, etc. For us, this was a big advantage since we have worked in C\ before but had no knowledge of working with VHDL.
In the process of developing this library, we have helped to find some bugs and identified parts of C# which we found missing from in SME's allowed subset, such as pass by reference, methods which take 0 parameters and work directly on fields. As these have now been taken into account in SME, it should be even more accessible to users in the future.
The major problems we encountered was using Vivado to synthesize our implementations. If we wanted alternatives to Vivado we would need to switch to another FPGA than the Xilinx, however, investigating and determining what the advantages/disadvantages are of the different hardware/software would have taken a significant amount of time. Given the scope of our project, we thought that time was better spend on implementation and testing. The usage of synthesis tools is, however, an inevitable part of FPGA programming and will likely require some time to familiarise with. This process

can especially be tedious because of how long it takes to run it through the synthesis tool, ranging all from 20 minutes to multiple hours in extreme cases. An important thing to note is that even though it proved difficult to optimize SHA-2 and MD5, their naive implementations had a reasonably large data throughput and uses few resources. This is important since it was fairly quick to get them implemented using SME. If we look past the Vivado problems, we could have a prototype ready to use relatively quickly. Of course, we had access to expert guidance, given that our supervisor helped create SME; however, SME has still proved to be a useful tool for FPGA programming and was quite manageable to familiarize oneself with.

6.2 Future work

Because we focused on creating a wider variety of cryptographic functions, the project has some shortcomings that could be fixed in future work.

Firstly it is not ideal that SHA-2 and MD5 can only handle short inputs, and thus a pressing suggestion is to make them handle more than one block (448 bits) of data at a time. This could be done by introducing a bus to forward the result of the final stage back into the start stage. As shown previously, this will be effective for small messages, but due to the overhead and the data dependency of SHA-2 and MD5, it would not be better than the naive implementation.

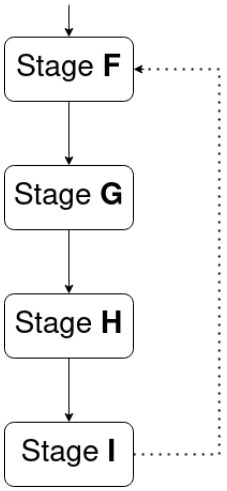


Figure 6.2: A visualization of the pipeline stages of MD5 (SHA-2 is similar). The proposed bus is the dotted line.

Secondly, we had no prospect of getting our hand on a physical FPGA, so we decided not to look into developing a device driver/interface to be able to communicate with the FPGA and call our functions in C. Such a driver would, however, be necessary to actually use the library. Furthermore, we have successfully implemented and tested our implementations through Vivado, so it would be appropriate to test the actual performance instead of just relying on the theoretical limits once the driver, etc., was set up. The next thing we would like is to physically synthesize the implementation on the Zynq Zedboard FPGA. In this context, it would also be interesting to compare with how well these versions compare to GPU accelerated solutions as this is an obvious first choice in many branches of computer science when trying to accelerate the performance

of an algorithm.
Thirdly, the library contains fundamental algorithms, but it would always be good with a more extensive library.
Fourthly, in our research for the project, we have looked at multiple different FPGA implementations of the different functions, but most of the presented results in these papers are conducted on more expensive FPGAs, and it would be interesting to see how well our versions would do compared to versions written in an HDL.
Lastly, our optimizations have mainly focused on the simple pipelining aspects, whereas there are known optimizations that could be implemented. For example, one such optimization is explained in the following subsection.

6.2.1 Known SHA256 improvements on FPGAs

The subject of running and optimizing hash functions on FPGAs is not new. One work on this subject is "Improving SHA-2 Hardware Implementation", by R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis. The most significant increase they documented in their report was a throughput increase of 40% on a VIRTEX II Pro FPGA(XC2P30-7) board[1]. The increase comes from measuring their own implementation against a commercial product (Helion) on the same board.
The improvements they found were some rescheduling of operations. SHA-2 is designed to have data dependency for previous rounds. So values of round i can not be computed until round $i-1$ is finished, making unrolling and pipelining almost impossible. They did find some values and techniques they could optimize without increasing the circuit size much, which is usually a problem.
One improvement they did were to pre-calculate all possible values needed in round $i+1$ in round i. Of the variables A-H, only A and E are dependent on calculations, the rest just gets shifted. So the following was done:

$$E_{t+1} = D_t + \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + H_t + K_t + W_t \tag{6.1}$$
$$A_{t+1} = \Sigma_0(A_t) + Maj(B_t, C_t, D_t) + \Sigma(E_t) + Ch(E_t, F_t, G_t) + H_t + K_t + W_t \tag{6.2}$$
$$\Downarrow \tag{6.3}$$
$$E_{t+1} = D_t + \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + \sigma_t \tag{6.4}$$
$$A_{t+1} = \Sigma_0(A_t) + Maj(B_t, C_t, D_t) + \Sigma(E_t) + Ch(E_t, F_t, G_t) + \sigma_t \tag{6.5}$$

Where $\sigma_t = H_t + K_t + W_t = G_{t-1} + K_t + W_t$.
In the future we could test this strategy, along with the other changes they describe. Such as decreasing the curcuit size by generalizing the computations for the digested message (DM):

$$DM(j+4)_i = E_{t-3+j} + DM(j+4)_{i-1} \qquad \text{for } 1 \leq j \leq 3 \tag{6.6}$$
$$DM(j)_i = A_{t-3+j} + DM(j)_{i-1} \qquad \text{for } 1 \leq j \leq 3 \tag{6.7}$$

[1]This is an article and FPGA, however, the optimizations are still interesting and might still be worth investigating.

Chapter 7

Conclusion

We have in this report presented four known cryptographic functions and how we have implemented them using SME for a library of cryptographic functions for FPGA. We have covered the advantages and disadvantages of using SME over a HDL. In the process of making the library, we have optimised the implementations and have gotten mixed results. The optimised version of MD5 and ChaCha20 have been able to compete with mid range CPU speeds, while using less power. The biggest improvement was seen in MD5 which ran at 3210 MBps around 4.5 times faster than the mid range CPU, where ChaCha20, was only able to run at around the same throughput as the CPU. Despite the promosing results we have also gotten results that werent as good. For instance it was difficult to gain significant improvement to SHA256 as its LUT usage was so high. Similarly, AES was significantly faster than the CPU but failed to compete with hardware solutions such as the AES-NI ASIC. For all the functions, however, we did see some sort of improvement from pipelining them. Furthermore, we have shown how our pipelined implementations increase in both size and power because of the increase in logic and frequency. One aspect that was interesting however, was that the pipelined AES used far less power than the naive version. Another point we have covered is that it is initially difficult to tell how well a function algorithm designed for CPUs is going to perform on a FPGA.

Bibliography

[1] Carl-Johannes Johnsen, Alberte Thegler, Kenneth Skovhede, and Brian Vinter. SME: A High Productivity FPGA Tool for Software Programmers. Expected publication: July 2021, 2021.

[2] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. doi:10.1145/359576.359585.

[3] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. W. Smith. Building the ibm 4758 secure coprocessor. *Computer*, 34(10):57–66, 2001. doi:10.1109/2.955100.

[4] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford, CA, USA, 1979. AAI8001972.

[5] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, 1949. doi:10.1002/j.1538-7305.1949.tb00928.x.

[6] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992. URL: http://www.ietf.org/rfc/rfc1321.txt.

[7] kockmeyer. Sha-2, 2007. [Online; accessed 12-June-2021]. URL: https://commons.wikimedia.org/wiki/File:SHA-2.svg.

[8] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001. doi:https://doi.org/10.6028/NIST.FIPS.197.

[9] Joa Daor, Joan Daemen, and Vincent Rijmen. Aes proposal: rijndael. 10 1999.

[10] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015. URL: https://rfc-editor.org/rfc/rfc7539.txt, doi:10.17487/RFC7539.

[11] Amba axi-stream, protocol specification, Apr. 09 2021. URL: https://developer.arm.com/documentation/ihi0051/b/Introduction/About-the-AXI4-Stream-protocol.

[12] Specifications for the xilinx zedboard. [Online; accessed 12-June-2021]. URL: https://reference.digilentinc.com/_media/zedboard:zedboard_ug.pdf.

[13] Pynq-z1: Python productivity for zynq-7000 arm/fpga soc. [Online; accessed 13-June-2021]. URL: https://www.xilinx.com/products/boards-and-kits/1-hydd4z.html.

[14] Intel core i5-7500 processor. [Online; accessed 13-June-2021]. URL: https://ark.intel.com/content/www/us/en/ark/products/97123/intel-core-i5-7500-processor-6m-cache-up-to-3-80-ghz.html.

[15] Advanced encryption standard (aes) engine v1.1, Jun. 03 2020. URL: https://www.xilinx.com/support/documentation/ip_documentation/aes/v1_1/pg383-aes.pdf.

[16] Xcku035-1sfva784c. [Online; accessed 12-June-2021]. URL: https://www.digikey.com/en/products/detail/xilinx-inc/XCKU035-1SFVA784C/6132037.

[17] Amd ryzen threadripper 3970x processor. [Online; accessed 09-June-2021]. URL: https://www.komplett.dk/product/1147275/hardware/pc-komponenter/processorer/amd-ryzen-threadripper-3970x-processor.

[18] voluntas. openssl speed aes_ctr aes_gcm chacha20 poly1305. [Online; accessed 09-June-2021]. URL: https://gist.github.com/voluntas/fd279c7b4e71f9950cfd4a5ab90b722b.

[19] WhiteTimberwolf. Electronic codebook (ecb) mode encryption, 2013. [Online; accessed 05-June-2021]. URL: https://en.wikipedia.org/wiki/File:ECB_encryption.svg.

[20] WhiteTimberwolf. Electronic codebook (ecb) mode decryption, 2013. [Online; accessed 05-June-2021]. URL: https://en.wikipedia.org/wiki/File:ECB_decryption.svg.

[21] WhiteTimberwolf. Cipher block chaining (cbc) mode encryption, 2013. [Online; accessed 05-June-2021]. URL: https://en.wikipedia.org/wiki/File:CBC_encryption.svg.

[22] WhiteTimberwolf. Cipher block chaining (cbc) mode decryption, 2013. [Online; accessed 05-June-2021]. URL: https://en.wikipedia.org/wiki/File:CBC_decryption.svg.

[23] WhiteTimberwolf. Counter (ctr) mode encryption, 2013. [Online; accessed 05-June-2021]. URL: https://commons.wikimedia.org/wiki/File:CTR_encryption_2.svg.

[24] WhiteTimberwolf. Counter (ctr) mode decryption, 2013. [Online; accessed 05-June-2021]. URL: https://en.wikipedia.org/wiki/File:CTR_decryption_2.svg.

Appendix A

Requirements & How to run

To run the C# code one nneds the .NET core 3.1 compiler and then every project can be run by running `dotnet run` from within the folder. To reproduce the FPGA design should be able to this from the files for the IP which is located in the HDL folders and by the specific versions .bd file using Xilinx Vivado. The code is located at: https://github.com/Spatenheinz/Bachelor

Appendix B

Setup used in benchmarking

B.0.1 Hardware

Raspberry Pi 4 - 8GB

CPU: i5-7500

- Cores: 4
- Threads: 4
- Base Freq: 3.4GHz
- Max Freq: 3.8GHz
- TDP: 65W

RAM:

- Size: 2x4GB
- Type: DDR4
- Speed: 2133 MTps

Disk: Kingston SA400S372

- Size: 240GB
- Form Factor: 2.5
- Interface: SATA Rev 3.0
- Read: 500MB/s
- Write: 350MB/s

Motherboard

- Model: MSI H110I PRO

B.0.2 Software

Operating System:

- Pi: Raspberry PI OS - buster
- i5: Linux 5.9.16-1-MANJARO

Compilers:

- C: gcc 10.2.0
- C: .NET 5.0 .NET 3.1

Other Tools:

- OpenSSL 1.1.1k
- Xilinx Vivado v2020.2

Appendix C

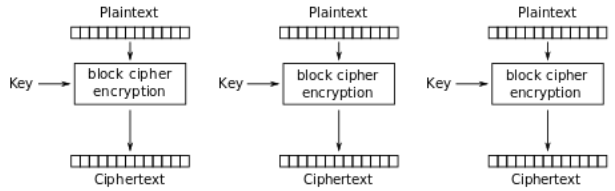
AES Look Up Table

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	A2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	E5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	D6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	Be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	Da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	B8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	Ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	F4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

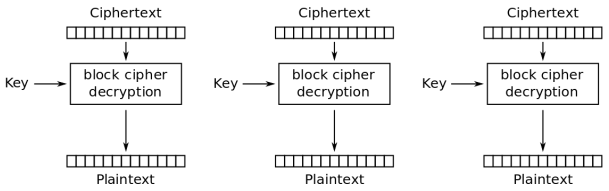
Table C.1: AES Look Up Table. Every value is shown in hex

Appendix D

Modes of operations

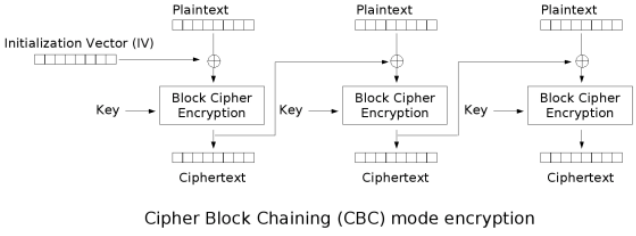


(a) Encryption [19]

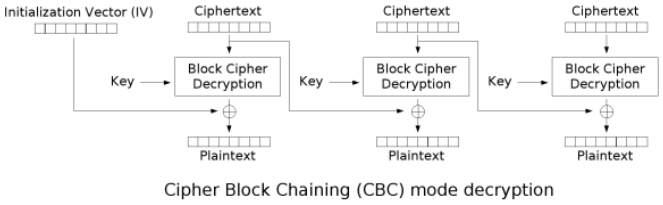


(b) Decryption [20]

Figure D.1: Electronic Codebook (ECB) mode

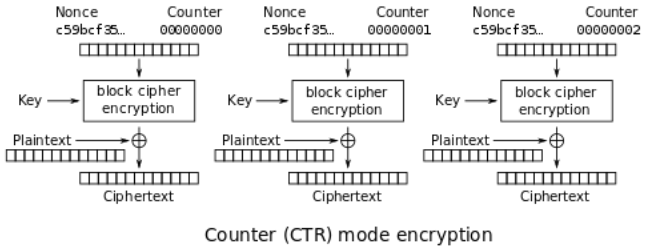


(a) Encryption [21]

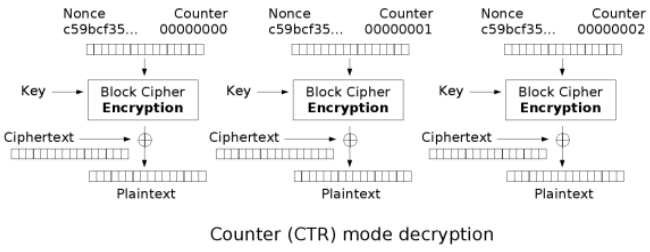


(b) Decryption [22]

Figure D.2: Cipher Block Chaining (CBC) mode



(a) Encryption [23]



(b) Decryption [24]

Figure D.3: Counter (CTR) mode