Bachelor Project

Jacob Herbst (jmrv148), Jonas Flach Jensen (sjm233)

\forfatter{...} el. \author{...}

Cryptographic library for FPGA's

Advisor: Kenneth Skovhede

Contents

1 Introduction

2 Background

2.1 Field Programmable Gate Arrays

Fields Programmable Gate Arrays (FPGA's) is an integrated circuit which can be reconfigured in the "field". It stands as the opposite Application Specific Integrated Circuit, which will have the single purpose, whereas FPGA's can be reprogrammed to have different purposes. It means an FPGA can be configured to work as a CPU, a GPU or something else entirely. This is done in hardware description language such as Verilog or VHDL and typically run through a program to sythesize the design on the FPGA. FPGA's consist of a set of Configurable Logic Gates (CLBs) and interconnects between these. CLBs is the reason FPGA's is reprogrammable. They differs from classic logic gates such as Nand, etc. used CPU in that they are constructed by look up tables, which can be reprogrammed, instead of gates which are fixed. Since these lookup tables can be build for specific purposes FPGAs can be a programmed to do one thing and do it really well. This lack of generality is often good for both performance and power consumption, compared to a CPU which needs to be able to do general prosessing, and thus in general has a lot of waste.

2.2 Synchronous Message Exchange

Synchronous Message Exchange (SME) 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 on its own processor 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 static size. Meaning no dynamic lists, while-loops etc. Simulation processes on the otherhand will not be a part of the actual hardware design, hereby making dynamic constructs legal. Furthermore simple processes has an optional property `Clocked Process` which means … TODO. 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 to depending on the process' 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}. 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 was 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 and 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[1], which is the process communications protocol we wil 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 AST which can be translated into VHDL code[**?**], 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[**?**].

2.3 A crypto library

Cryptographic functions are used by developers across most branches, whether it'll be 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. In fact, 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, which makes it difficult to use for experimental development, small projects, and in 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 so that they are competitive with the existing software solutions. Creating a crypto library for FPGA's …

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, is pseudo-random, and upholds certain properties listed below, it can be categorized as a "Cryptographic Hash Function".

- It should be deterministic, as it is important that the same hash is computed given some 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 in big changes in the hash.

[1] AXI ref

- It is collision-resistant, meaning it is unreasonably hard to find two distinct messages to have the same hash.

1. Merkle-Damgård construction As stated previously this library include implementations of MD5 and SHA-256. These are very similar in design and follows a widely 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[2]. From Figure 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 static 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` along with the second block of the message. This is repeated until the entire padded message has been processed. From here a potential finalization function can be used to improve the hash and a hashed value is produced.

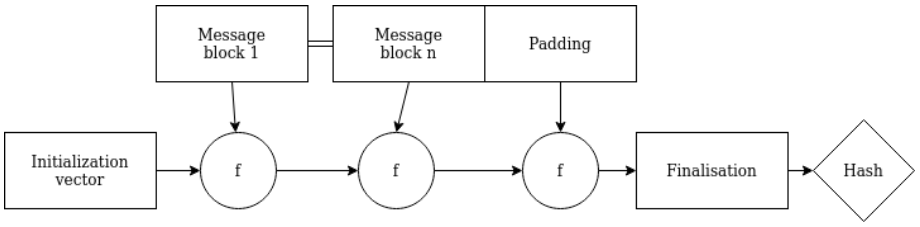

Figure 1: Merkle-Damgård construction

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

1. 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. Obviously this requires some considerations, firstly data that is not a multiple of the block size will require some sort of padding, as the method of choice is not defined by any standard we have not taken uneven sized plaintexts into account. Furthermore there are defined multiple Modes of operations to handle data with more than one block. Of these the most simple is Electronic Code Book (ECB), which simply will encrypt each block of the data independently. It is worth noting that this is not the most secure

[2] ref to the article
[3] ref til confusion difussion

mode since identical data blocks will produce identical cipher blocks. Another more secure method is Cipher Block Chaining (CBC) which will xor the cipher text of the previous block with the plaintext of the current block before encrypting the block. This approach is a embarrisingly sequential method. More parallel and robust 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 will then be XORed with the plaintext. Each block after the initial will then take the nonce increased by some fixed size pr. block.

2. 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.

2.4 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[4]. THe MD5 algorithm will thus create a the 128-bit digest from a arbitrary sized message of n bits. Since MD5 uses a merkel damgaard construction, it follows figure**??**. 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 merkel damgaard 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 merkel damgaard strengthening is applied by appending a 64-bit representation of the message length mod 2^{64} to the padded message, resulting in every partiotion being 512 bits wide. Each partition of the message will then be fed into the compression `f` function in Figure **??**.

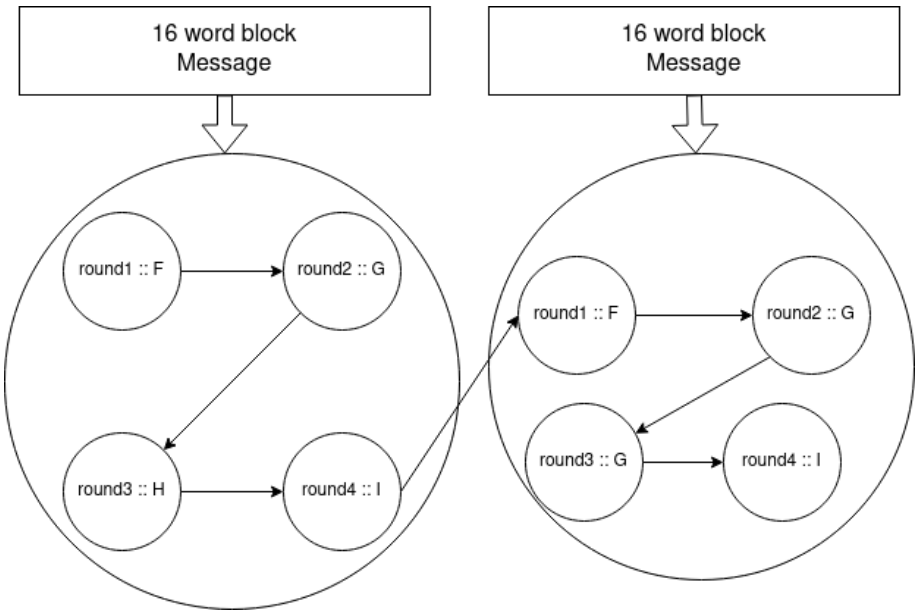

Figure 2: Two iterations of the MD5 rounds.

Figure 2[4] 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 4 functions, defined as such to, in "bitwise parallel" produce independent and unbiased bits in each of the rounds.

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z) G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z) H(X,Y,Z) = X \oplus Y \oplus Z I(X,Y,Z) = Y \oplus (X \vee \neg Z) \tag{1}$$

In `f` a total number of 64 rounds will be computed, each of the functions 1-4 is applied a total of 16 times. Figure `??` shows each of the specific rounds, where `[abcd k s i]` denotes `a = b + (( a + round(b,c,d) + M[k] + K[i]) \lll s)`, where `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 the `floor(2^32 * abs (sin(i + 1)))`.

```
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]
```

when all rounds have are completed the new vector ($A_{1,B1,C1,D1}$), will be added to the vector from before the rounds. This result will store the digest for that round. If this is the result of applying `f` to the last block of message, we have the MD5 digest of the message. Is this however result of applying `f` to any other block this result will be the IV of the next round. It is worth noting that MD5 is not a very good hashing algorithm for cryptography, as collision attacks exist, despite the fact that it uses the merkle damgaard construction, but still show use for data integrity purposes and such.

2.5 SHA-256

SHA256 is a one way Secure Hash Algorithm. Which is where it gets its name from. It is part of the SHA2 familiy and was designed and published by the NSA. SHA256 is also build upon the Merkle-Damgård construction, like MD5.

The 256 part refers to the output size of 256 bit. SHA256 can take in 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 familiy exsits, like

[4] this is a horrible picture and we should prob make a better one

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 4 computation stages. All of which works on the massage encoded in binary. All operations are also bitwise.

Firstly all constants and variables gets initialized. All members of the SHA familiy 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. It also uses eight working variables H_i with initial value of the first 32 bits of the fractional parts of the square roots of the first 8 prime numbers. These H variabels gets updated with each round, and will contain the final hash after the final round.

Then the message gets padded. In SHA256 the message bit-length needs to be a multiple of 512. The padding scheme consists of appending a 1 at the end of the message, and then appending 0's untill the message has a length of $x * 512 + 448$. Finally the length of the original message is appended as a bigendian 64 bit integer. This also results in SHA256 not being able to handle messages that has a lenght of over 2^{64}.

For 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:

$$w[i] = \begin{cases} M[i] & for 0 \leq i \leq 15 \sigma_1(W) + \\ for 16 \leq i \leq 63 \end{cases} \tag{2}$$

The second stage is to update the H variables.

The third stage is to compute ch, maj, $temp_1$ and $temp_2$ for each of the 64 entries in the array K and the expanded message W.

For the last stage the H variables gets updated one last time, so they now contain the final hash. Then they get appended together and returned.

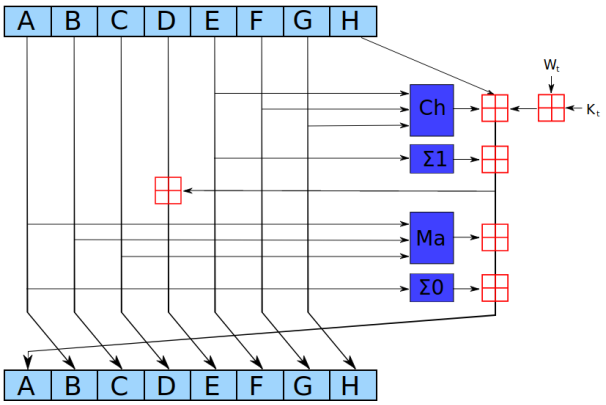

Figure 3: A SHA2 round

2.6 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 mostly 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[5]. Rijndahl is an Substitution-permutation (SP) network which manipulates a block and keysize of any multiple of 32 in the range 128-256 bits. In the exact specification of AES the blocksize is fixed to 128 bits where the key potentially can be 128, 192 or 256 bits. The 128 bits is arranged in a 4 x 4 column-major order matrix. As stated AES is a 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:

$$W_i \begin{cases} K_i & \text{if} i < N \\ W_{i-N} \oplus \text{SubWords}(W_{i-1} \lll 8)) \oplus \text{rcon}_{i/N} & \text{if } i \geq N \text{ and } i \equiv 0 \pmod N \\ W_{i-N} \oplus \text{SubWords}(W_{i-1}) & \text{if } i \geq N,\ N > 6, \text{ and } i \equiv 4 \pmod N \\ W_{i-N} \oplus W_{i-1} & \text{otherwise} \end{cases} \tag{3}$$

   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 beeing defined as follows:

$$\text{SubWord}([b_0 b_1 b_2 b_3]) = [S(b_0)S(b_1)S(b_2)S(b_3)] \tag{4}$$

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

i	1	2	3	4	5	6	7	8	9
rcon[5]	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[6], 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 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 ny 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 **??**[6]

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 4

[6] TODO insert substitution box

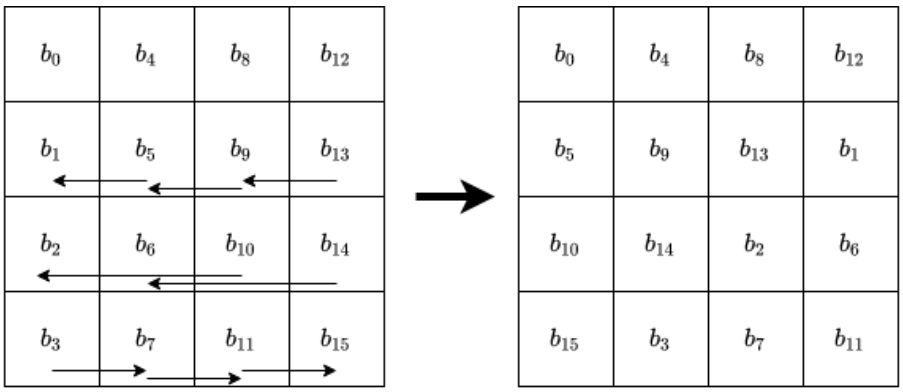

Figure 4: ShiftRows operation

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:

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

where c denotes the column. Multiplication is as described above and addition is XOR.

For decryption the equivalent inverse functions can be used, as Rijndahl is truely 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 Rijndahl[6] describes how these different steps can be implemented using lookup tables. This implementation can be realised 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:

$$T_0[a] = \begin{bmatrix} S[a] \cdot 02_{16} \\ S[a] \\ S[a] \\ S[a] \cdot 03_{16} \end{bmatrix} T_1[a] = \begin{bmatrix} S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \\ S[a] \\ S[a] \end{bmatrix} T_2[a] = \begin{bmatrix} S[a] \\ S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \\ S[a] \end{bmatrix} T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \end{bmatrix} \tag{5}$$

these will then get used in a round transformation as

$$e_j = T_0[a_{0,3}] \oplus T_1[a_{1,2}] \oplus T_2[a_{2,1}] \oplus T_3[a_{3,0}] \oplus k_j \tag{6}$$

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 2008[**?**][7] this method is no longer the fastest on CPUs.

2.7 ChaCha20

ChaCha20 is a stream cipher, which intend is to be a fast and efficient standby cipher in case AES is compromised. unlike block cyphers, such as AES, which works on a

[7] TODO insert reference

fixed sized block of text steam ciphers work on a per byte level. This is usually done by combining the plaintext with a pseudorandom stream of digits using XOR. As 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, layed out as such:

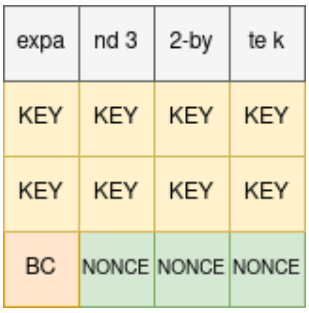

Figure 5: seed of ChaCha20

As one can see, the layout of the seed is fairly simple, and constitutes 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), which is sufficient for up to 256GB of plaintext.
- A nonce which spans 3 word in little endian.

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 chacha 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 blockciphers into the streaming cipher. The confusion part of the algorithm follows a simple add-rotate-XOR (ARX) structure. That is every round is based on only simple arithmetic add, left rotations and XOR operations. more specifically ChaCha20 consist of 20 round of which each consists of 4 quarter rounds will confuse 4 input words. Each quarter round looks as such: As said chacha20 will perform 20 round consisting of 4 quarter rounds. For the a quarter round in an even numbered round will take a column of the seed as the input. where an odd numbered round will work on diagonals. Lastly when the 20 rounds has been computed, the intial seed and the modifed version is index-wise added giving a resulting block of 16 words of stream and this stream can then be XORed with the plaintext to get the cypher.

3 Implementation

3.1 MD5

3.1.1 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 completely contained in the single clocked process and works as described in 2.4. The major difference comes in the data flow. Since our

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

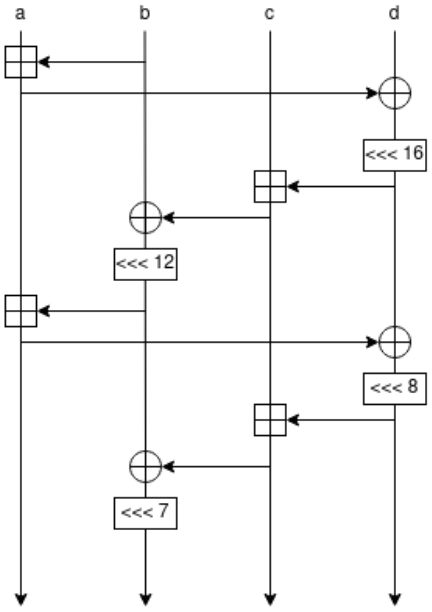

Figure 6: ChaCha Quarter Round

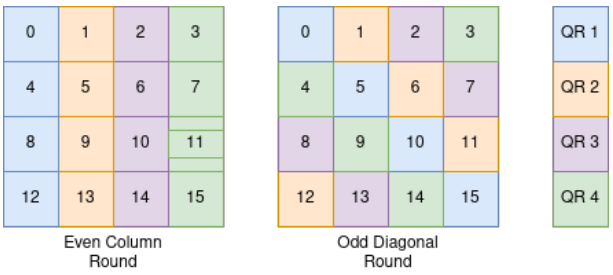

Figure 7: ChaCha Rounds

program will be mapped to hardware, we cannot have a variable sizes, everything has to be static. Thus we have opted for an approach which 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 a "independent" bus without depending 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 inputbusses to the process, one which would contain the message and one which would contain the IV to modify. However, we find this approach unecessary as the initialisation 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; }
}
```

One can see there are multiple things to keep track of. First and foremost, all busses we will be working with should have a flag for whether or not a bus has data inside of it, since we try to adhere to the AXI protocol, which specifies some standards for a ready/valid handshake between the processes. Secondly, A byte-array 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 in the cases where 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 recive the digest from the process we also need an output bus. This bus is however fairly simple. 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 ingoing bus to know when it is safe to send the digest to the simulation and an outgoing ready bus to let the simulation know when to send the message values. [8] We will be using the AXI protocol for all implementations.

3.1.2 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 8 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 which does the last addition of the two vectors.

[8] should we have a section in the background for AXI?

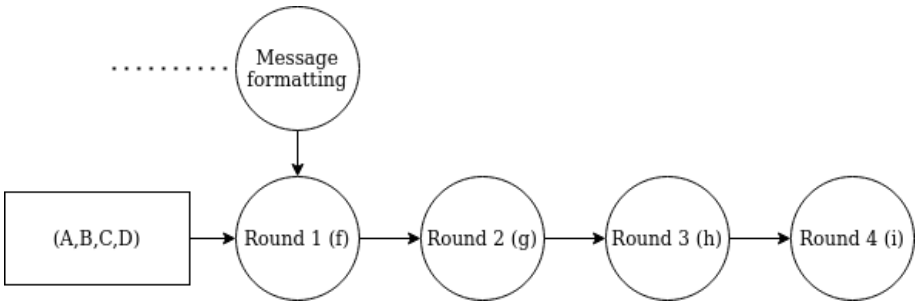

Figure 8: MD5 pipeline

One problem we however have faced with this general approach is that MD5 is embarisingly sequential in the sense that `round 1 (f)` of the compression of the second message block 2, will depend of the result of `combiner` of the first message block's compression. This will create a stall. Thus for long messages the speed-up from the naive version should not be too considerable. However for very small messages (<448 bits) the throughput should be noticable faster as the propogation time should be decreased considerably. Hereby we expect to see a possibility to increase the clock frequency, which would tradeoff energy use for throughput.

3.2 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 major differences is that SHA uses big endian encodings while MD5 uses little endian.

3.2.1 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. With the exception of the array in the `Digest` output bus being extended to contain a 256 bit hash.

The SHA256 algorithm is located in the `processBlock()` routine. The routine has the input block stored in the first 16 entires of the array `blockD[]`. Then these blocks get expanded and the hash calculated as described in Section 2.5.

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.

3.3 AES

3.3.1 naive

Just like for the other algorithms AES can naively be implemented as a single simple process. That is we could implement the SME implentation 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 unecessary complication to the process as it would have to multiple things at once. Furthermore

and more importantly a combined encrypter/decrypter reduces the utilizations of the library. Since AES is a block cypher and rarely only need to encrypt a block of 128 bits a sequence of blocks needs to be encrypted. This naive approach is not necessarily bad for some of the modes of operations such as Electronic Codebook (ECB) (which never really should be used anyway), Cipher Block chaining (CBC) (which eliminates parallelism) etc. as these will need a decryption algorithm. However is the programmer using this library choosing to operate under a Counter mode (CTR) or Galois-counter mode (GCM) the decryption algorithm itself would be unecessary as these modes uses the encryption function to both encrypt and decrypt. Thus in a hypotetical 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 ressources are wasted. Thus we have decided that for the basecase implementation that encryption and decryption should be seperate processes. We will only go over the implementation of encryption as the process for decryption is the exact inverse computationally as described in section 2.6 and the structure thus follows symmetrically. For the design a single bus with 4 fields as seen below suffices. It consists of two Valid flags which works in a similar matter to the one described for MD5. Furthermore there is 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 dont want to make the process itself flexible with multiple AES versions as it will reduce the resource utilization on an FPGA. The reason for this is the optionality of additional rounds for a 256 bit key version would map these extra computations to hardware which will make the circuit more complex and harder for vivado to route the design and will have a harder time meeting the timing constraints. In section **??** we will take a short look at this. If 128 key encryption suffices the overhead from including the 4 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 cypher as the algorithm is symmetrical. For the output bus we however dont really have to output the key, assuming the result is send back to the device who called the function.

```
public interface IPlainText : IBus {
    [InitialValue(false)]
    bool ValidKey { get; set; }

    [FixedArrayLength(BLOCK_SIZE)]
    IFixedArray<byte> Key { get; set; }

    [InitialValue(false)]
    bool ValidData { get; set; }
    [FixedArrayLength(BLOCK_SIZE)]
    IFixedArray<byte> Data { get; set; }

}
```

For the actual AES process we follow the T-box approach described earlier, as we want the throughput of our FPGA to be as efficient as possible despite lacking AES specific instructions such as, `GF2P8AFFINEINVQB`, `GF2P8AFFINEQB` and `GF2P8MULB`.

3.3.2 optimisation 1

AES have shown to be quite fast even in the implementation, however its should still be a quite 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 process from before can be divided such that each round can is its own process. We have tried two solutions this way.

The first solution will merely be the original process split up into smaller parts. This is fairly easy to implement, but it has one flaw which might underutilize the LUT's of the FPGA, since each process will only know about each other through a bus and henceforth not have access to the same lookup tables/T-boxes. Coroally every process that needs access to one of the lookup tables will have to have it defined for itself. Argueably this is an wasteful approach as each process will have to use its own 4KB of T-boxes, which sums up to 36KB.

If we however want a solution that uses less memory we can use the block RAM (BRAM) to our advantage. We can utilize BRAM through SME using `Components`. as we only have to use the BRAM for lookups we can settle for Single Port Memory. One thing to be aware of when using BRAM is that it will take 2 clock cycles for a data transaction to happen. With this restriction, there are some considerations to have in mind, should the Tboxes be crammed into a single BRAM, thus having more clockcycles as we can only read a single address at a time, or should we distribute them out on multiple BRAMs to be able to do muliple lookups in each cycle. If the 4 T-boxes is distributed out on 4 BRAMs we can achieve the following pipeline:

3.4 ChaCha

3.4.1 naive

Just like AES ChaCha will work in two phases. The initial phase will be to setup the seed. After the initial setup, the only modification to the intial 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 **??**. Just like for any of the other algorithms we have opted for a full 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:

```
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`. This design however have posed some challenges.

3.4.2 First optimization

As it should hopefully be clear from the high level description the computation of the rounds are quite simple compared to AES, which both needs computation in the key-expansion, and in the T-box version of AES a lot of table lookups. Furthermore ChaCha encourages concurrency and parallelism, as each "block" can be computed completely independently of each other in a similar fashion to AES. Thus a pipelined version is easy to implement compared to the has functions. In our initial version we split the process

into 10 processes, each of which will perform 2 of the rounds, one column-wise round and one diagonal-wise. The reason we start with 10 processes is that we dont want each of the rounds to do too much work while we on the other hand dont want the processes to be som small that the handshake become the overhead.[9]

4 Results of implementation

All of the different implementations have been tested against the C# standard library eqivalent algorithms and ensures that the results produced is correct. The only exception for this `Chacha` as it still is a quite uncommon cipher. All implementations synthesized 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 OpenSLL, using `openssl speed -evp "algorithm"`. Unfortunately we have not been able to get our hands on the board in time and we thus stand with some limitations on the benchmarking results. Firstly, the reported frequency is the results from vivado, which might be a little different had it been an actual test from calling the function from a C library. Secondly, we settled for a raspberry pi 4B for the comparisons. 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.[10] All the "raw" stdout results from the benchmarks can be found in Appendix **??**.

4.1 MD5

MD5 naive: 2.38 Mhz
throughput: 512*2.38Mhz=1.218GBit=152,3MB/s

4.1.1 Throughput

Looking at Table /ref{} we can see our naive implementation of performs adequatly. Our naive version runs at around 150 MB/s, thus performing about as well as our own C-version optimized with -O3. When comparing to a threaded version C_t, which uses 7 threads to calculate the 7 smaller messages of the C-version, the naive version performs quite poorly. This is to be expected as a completely sequential should in general not be faster than a parallel version. 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 OpenSLL this is a speedup of more than 300 pct. There are some things to keep in mind from this:

- 16 bytes is not nearly enough to fill a block and thus full blocks of data is not process meaning there is a lot of spill. Even when running the same benchmarks on a i7-7500 the result of 16 bytes is merely 78 MB/s. Thus to get the full utilization we should focus the attention to 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.

Comparing to $\text{OpenSSL}_{\text{high}}$ one notices that our naive version cant really compare as its only about half as fast. The same is true for the C# version, which just as well could be using openssl.

[9] is this possible?
[10] write some descriptive text for the results

Version	Naive	C#	C	C_t	$OpenSSL_{low}$	$OpenSSL_{high}$
Throughput (MB/s)	152.33	287	154.33	255.53	41.84	292.53

Benchmarking results for MD5.

4.1.2 Power Consumptions

From the previous section we showed that our FPGA version was overall not as fast as its CPU counterparts, however the FPGA version shows promises in power usage. from Figure **??**

4.2 SHA256

SHA naive: 2.10 Mhz
Throughput: 512*2.1Mhz=1.075GBit=134,4MB/s

4.2.1 Throughput

Since SHA256 is closely related to MD5 we would expect the results to be relatively similar, however percentage-wise the FPGA version performs relatively better as our Naive version has a throughput of 134.4 MB/s whereas $OpenSSL_{high}$ is 30 MB/s faster, corresponding to ?? percent.

Version	Naive	C#	$OpenSSL_{low}$	$OpenSSL_{high}$
Throughput (MB/s)	134.4	163	26.3	164.97

Benchmarking results for SHA.

4.3 AES

AES naive: 25 Mhz
Throughput: 128*25Mhz=3.2GBit=400MB/s

4.3.1 Throughput

The results of AES is interesting compared to our other implementations in the sense that even the naive FPGA version is outperforming the CPU. One can notice that our naive version has a throughput of 400 MB/s which is around 4.49 times as much as OpenSLL on its peak performance and that it likewise outperforms C# and our own C-version with 5.7 and 6.2 times respectively. Even the threaded version is only half as fast as the FPGA solution. These results are quite promising in itself and clearly shows that the FPGA is very suitable for solutions as this one and in cases where large amounts of data needs to be encrypted or decrypted the FPGA is preferable over an arm processor. It is however still worth noting that this still only outperforms processors that does not have AES-NI. For instance running the same OpenSSL benchmark on an intel i7-7500 the worst case has a throughput of 788 MB/s and an optimal solution of 5.61 GB/s. Even though there is a significant price difference between the ARM and Intel processor it still hints to how AES-NI, essentially a dedicated ASIC, outperforms more general CPU solutions by an order of magnitude. One aspect which would have been interesting to measure is how well the implementation synthesized on a high-end FPGA would perform compared to cabable CPU's. An article from 2020 which compares both pipelined and non-pipelined versions of AES on FPGA's shows that even on expensive FPGA's such as the Virtex-7 family only runs between 2.06-6.34 Gbps (257.5-792.5 MB/s)[**?**]. Thus a nonpipelined version will never be able to compete with an ASIC. One aspect that

would have been interested exploring is how well our highlevel version would compare to a solution which has been optimized directly in one of the HDL languages. We have not been able to do this but we can hint at the fact that some of the solutions described previously performs worse than our solution despite the difference in chipset. In the implementation section we described how we rejected to make a solution that was

Version	Naive	C#	C	C_t	$OpenSSL_{low}$	$OpenSSL_{high}$
Throughput (MB/s)	400	70	64.49	198.28	72.4	89.06

Benchmarking results for AES.

flexible in its key-size. The results hint that this have good impact on the performance. Comparing our solution to the solution presented in the SME github repository, 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 tradeoff some flexibility for a significant speedup.

4.4 ChaCha20

ChaCha naive: <5 Mhz ????
Throughput: ? fails nets before timing, too much data for a small board. Even though one would expect ChaCha20 to perform well, because of the simplicity in computation, our FGPA version of ChaCha20 performs extremely poorly. As one can see from the Table our version is X times slower than the openSSL solution. The culprit of ChaCha20's

Version	Naive	$OpenSSL_{low}$	$OpenSSL_{high}$
Throughput (MB/s)	?	84.03	306.81

Benchmarking results for Chacha.

poor peformance is found in the high amount of nets. Nets is sythetic datapath in vivado, which will be transformed into a wire when mapped to hardware. This suggests that we have too much data on the busses between the interlectual property (IP) and the register transfer level (RTL) of the design. This seems quite a reasonable argument as the input bus itself takes in 1152 bits and the output bus carries 544 bits to output the cipher. To have a concrete proof of this we also implemented a version which only generates the keystream. When running the keystream version through vivado we get a reported speed of 200 Mhz, which might suggest that Vivado is optimizing away some computations, and thus the results of this will be scewed. Anyways, we can from Figure**??** see that the naive version is taking up a lot of space on the FPGA and this will make timing even harder, especially when the net usage is too high.

5 Discussion

- SME vs VHDL
- How easy is FPGA programming
- ???

6 Conclusion

References

[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.

[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.

[4] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[5] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.

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

%%% Cha Cha er smart fordi den ikke bruger nogen “advanceret” operationer. Så specielle instruktioner på CPU’en kan ikke blive udnyttet %%% Så den burde virke lige så hurtig på hyper moderne optimeret CPU’er som på low power alternative CPU’er %%% Carl sagde vi kunne bruge: https://www.ctan.org/pkg/ieeetran til mødet %%% Nævn at både SHA256 og MD5 har samme opbygning.