,,,

Jacob Herbst (mwr148), Jonas Flach Jensen ()

Crypto in SME for FPGA

Advisor: Kenneth Skovhede

April 12, 2021

Contents

1 Introduction

2 Background

2.1 Field Programmable Gate Arrays

TODO: write something more in-depth.

Why use FPGA's? To get a speed increase by moving frequently used functions and calculations from the CPU to a specialized chip (the FPGA). This will result in less time "wasted" on the CPU, and the specialized chip will ideally also be able to do the computations faster than a generic CPU.

2.2 Synchronous Message Exchange

Synchronous Message Exchange (SME) is a programming model to enable FPGA development 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[1], 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, a `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. 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 every process should have one and possibly multiple "valid"-flags which shows if there is any data on the bus. 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 vivado[] to synthesize the implementation to actual hardware. For the cryptographic library covered in this report we will be using a 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.

[1] sme ref

IBM created their own "IBM 4758 Secure Coprocessor"[2]. 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 an implementation of various cryptographic functions, such as AES and SHA256. 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". One such example is the outdated MD5 algorithm.

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

1. Merkle-Damgård construction One approach that is widely used in cryptographic hashing is 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. 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, and the first 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. Lastly, the hashed value will be produced.

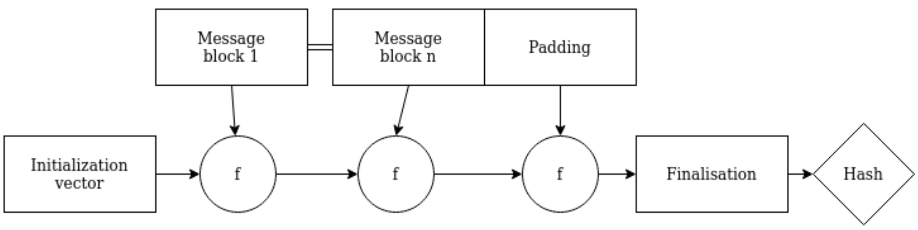

Figure 1: Merkle-Damgård construction

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[3]. MD5 uses a Merkle-Damgård construction. The MD5 algorithm work by partition the input message into blocks of 512 bits. It is

done by always padding the message with a single set bit followed by a series of 0's until the message = 448 mod 512. That is, even when the original message has a length of 448 mod 512, a 1 is followed by 511 bits of 0's. Next, a 64-bit representation of the message length mod 2^{64} is appended to the padded message. The digest will be calculated in a 32-bit 4-word initialization (A, B, C, D), with the initial value:
`[ A: 0x67542301, B: 0xefcdab89, C: 0x98badcfe, D: 0x10325476 ]`
and we use the following functions corresponding to each of the four rounds:
$F(X, Y, Z) = XY \vee \neg X Z$
$G(X, Y, Z) = XZ \vee Y \neg Z$
$H(X, Y, Z) = X \oplus Y \oplus Z$
$I(X, Y, Z) = Y \oplus (X \vee \neg Z)$
These are defined as such to in "bitwise parallel" produce independent and unbiased bits in each of the rounds.

Process each 16-word block (512 bits) by copying it into a buffer X, save the current digest buffer and perform the following rounds: For each round a function [abcd k s i] denoting $a = b + ((a + round(b,c,d) + X[k] + T[i]) \lll s)$, where round denotes the function corresponding to that round.[2]

```
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]
```

Next, increment each of the variables by its starting value.

The Digest will now be (A, B, C, D) in LE format.

It is worth noting that MD5 is not a very good hashing algorithm for cryptography, as collision attacks exist, but still show use for data integrity purposes and such.

2.5 AES

The Advanced Encryption Standard (AES) is a symmetric block cipher which is 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 msotly 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 of a vast variety of devices[4]. Rijndahl is an Substitution-permutation 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 can listed as follows:

[2] this might be unecessary to show

1. KeyExpansion: The key, whether it be 128, 192 or 256 bits is expanded using a keyschedule[3] which will expand a key into the number of rounds + 1 keys.

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[5], followed by the permutation which consists of 2 functions `ShiftRows` `MixColumns`, which will ensure the 4x4 matrix is permuted. 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[4]. 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 $\mathrm{b_i}$ denoting the ith bit of the byte and $\mathrm{c_i}$ denoting the ith 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.

ShiftRows will transform the 4x4 input matrix by rotating the rows 0 to 3 bytes to the left, meaning the first row $\{\mathrm{b_0}, \mathrm{b_4}, \mathrm{b_8}, \mathrm{b_{12}}\}$ will not be rotated, the second row will be rotated one bit to the left, i.e. $\{\mathrm{b_5}, \mathrm{b_9}, \mathrm{b_{13}}, \mathrm{b_1}\}$ 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 2

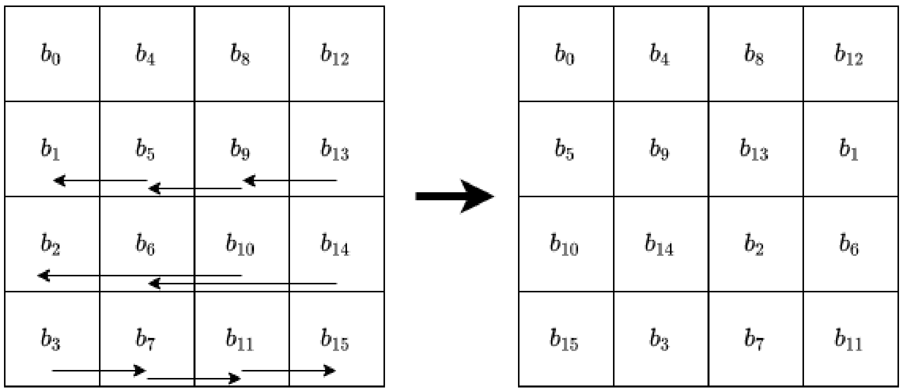

Figure 2: ShiftRows operation

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 addittion is xor. By now the input text should be pretty diffused.

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[5] describes how these different steps can be implemented as 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). This approach are generally considered faster as it reduces each round to 16 lookups and 16 xors compared to the normal approach where we

[3] should we include the schedule
[4] should we include the table?

memory needs to be moved around. As the decryption is truely invertible there exist likewise a lookup table approach for decryption.

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 by following figure 1 and using two busses and one simple process. Overall there are two approaches, Firstly, we could let the arrows in the diagram denote the busses, such that we would have a bus denoting the message-block and one denoting the digest. However, this approach requires an extra bus, thus the alternative. Since we use a c# implementation of SME, we can store the Digest locally inside the simpleProcess. Thus we will only require a bus with the message, corresponding to the downward-facing arrows and one for the Hash (the rightmost arrow). 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 SME busses should have a flag for whether or not a bus has data inside of it. 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.

The Digest on the other hand is simple. It only consists of a Valid flag and the Hash as an array of 4 32-bit words.

Except for the handling of the padding in relation to SME, the simple process works as explained in section 2.4

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. We can thus structure the program as such using a simple diagram. in figure 3 one can see we can split the hash function as a whole up into 5 simple processes and build a pipeline from this. One process for message formatting, and one for each of the 4 rounds. along with an Imessage bus and a Digest bus as described in section 3.1.1. This construction will create a pipeline where each process can run concurrently and potentially execute faster than the naive approach.

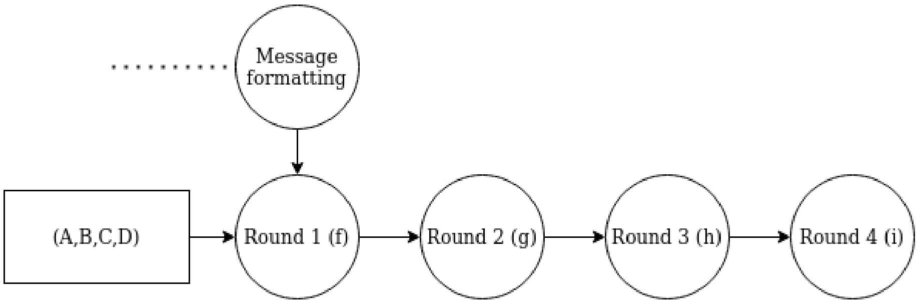

Figure 3: MD5 pipeline

3.2 AES

3.2.1 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 is 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 2.5 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 reduces the resource utilization. If 128 key encryption suffices the overhead from including the 4 extra rounds for 256 is wasteful.

```
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 process we follow the T-box approach described earlier, as we want the throughput of our FPGA to be as efficient as possible and comparable to fast CPU implementations.

3.2.2 optimisation 1

Just like for the other algorithms its is a quite slow approach to use only a single process and we can try to pipeline the different aspects of the algorithm as smaller processes. We notice that AES likewise uses rounds and each round can be divided into its own process. However we can note that each round does not have a lot of computation as we use the T-box implementation. The key-expansion might therefore be the most computationally heavy part and could be split into smaller parts.

4 Benchmarks

5 Discussion

6 Conclusion

References

[1] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666677, August 1978.

[2] 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.

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

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

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