6.375: Complex Digital System Design
Final Proposal, High-level design and Test Plan

Lauren De Meyer - Candace Ross

1 Introduction

In the last few years, the need for small devices with very low computing power has awoken an interest for lightweight cryptography. Consider for example wireless sensors and the rapidly growing number of Internet of Things devices, becoming smaller and smarter.

SPECK [1] is a family of lightweight block ciphers that was introduced by the NSA in 2013. It was designed for flexibility and optimized for hardware implementations. This report describes the design and implementation of a hardware SPECK module for an FPGA. First, we give a background on the SPECK algorithm itself and we describe its implementation. We created both a folded design and pipeline design and explored how changing the number of stages in the pipeline affects the throughput and the FPGA area used. As block ciphers require inputs of a fixed length, the last part of the report describes how we used our SPECK implementation to create an encryption tool for arbirary length messages.

2 SPECK

2.1 Block ciphers

A block cipher is an algorithm that transforms a plaintext into a ciphertext using a symmetric key. The plaintext and ciphertext have a fixed size, typically referred to as a block. The cipher consists of two functions, a round function and a key schedule. The ciphertext block is calculated by T subsequent evaluations of the round function on the plaintext block, where T is a fixed number of rounds. In each round, the round function uses a different key. The T round keys are calculated by the key schedule from the original masterkey.

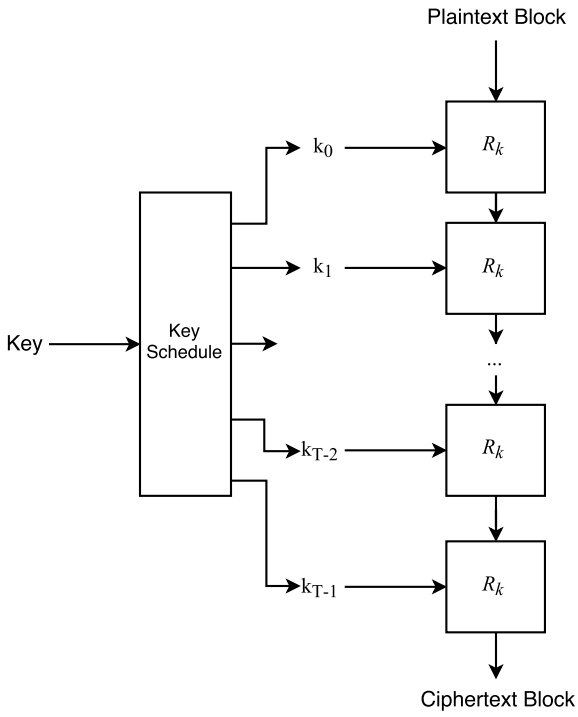

Figure 1: Structure of a block cipher of T rounds with round function R_k

2.2 SPECK's round function

SPECK's round function only requires modular addition (A), rotations (R) and exclusive OR (X). The use of these three efficient operations is popular and the schemes that use them are called ARX ciphers. The round function is shown in Figure 2.

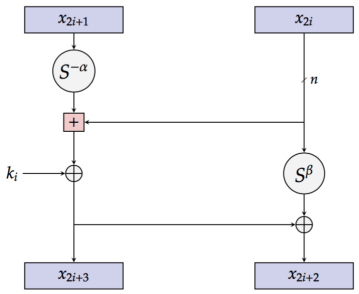

Figure 2: The SPECK Round function

One block of plaintext or ciphertext is $2n$ bits long with n the word size. As a family of block ciphers, SPECK allows for different choices for n, ranging from 16 to 64. For each choice of word size, there are one or two possible key sizes mn. For each pair of parameters

(n, m), the designers have specified the number of rounds to use and the rotation parameters α and β (See Table 1).

word size n	key words m	block size $2n$	key size mn	α	β	# rounds T
16	4	32	64	7	2	22
24	3	48	72	8	3	22
	4		96			23
32	3	64	96	8	3	26
	4		128			27
48	2	96	96	8	3	28
	3		144			29
64	2	128	128	8	3	32
	3		192			33
	4		256			34

Table 1: Parameters for SPECK

The round function depicted in Figure 2 is applied T times, starting with x_1 and x_0 respectively the most significant and least significant n bits of a plaintext block.

$$R_k(x, y) = \big((S^{-\alpha}x + y) \oplus k, S^{\beta}y \oplus (S^{-\alpha}x + y) \oplus k\big)$$

Each round transforms (x_{2i+1}, x_{2i}) to (x_{2i+3}, x_{2i+2}) with R_{k_i} and after the last round, we obtain the 2 words (x_{2T+1}, x_{2T}) which form the ciphertext block.

The round keys k_i are determined from the master key K by a key schedule, which requires exactly the same hardware as the round function. We split the key K in its m words:

$$K = (l_{m-2}, \ldots, l_0, k_0)$$

The first round key k_0 is thus the least significant word of K. Subsequent keys are calculated as follows:

$$l_{i+m-1} = (S^{-\alpha}l_i + k_i) \oplus i$$
$$k_{i+1} = S^{\beta}k_i \oplus l_{i+m-1}$$

This is equivalent to applying the round function with round key i:

$$(l_{i+m-1}, k_{i+1}) = R_i(l_i, k_i)$$

For decryption, we must use the inverse of the round function

$$R_k^{-1}(x, y) = \big(S^{\alpha}(x \oplus k - S^{-\beta}(x \oplus y)), S^{-\beta}(x \oplus y)\big)$$

and we must invert the order of the round keys.

2.3 Why embedded cryptography?

Why is it useful to create cryptographic hardware instead of using software implementations? Firstly and most obviously, a dedicated hardware implementation achieves better performance. More importantly, a separate cryptography processor creates an explicit physical barrier around sensitive information. Cryptographic software performed in an environment with other applications running concurrently can leak information through for example timing

variations or the cache. It is better for secret keys to be handled in their own environment with a private cache. Furthermore, many devices in the Internet of Things have limited resources or even no generalized core at all, in which case cryptography in hardware is the only option.

3 High-level design and Test Plan

3.1 Design

Apart from the fact that it uses the same hardware, another important aspect of the key schedule is that it can be executed "on the fly". There is no need to generate all round keys before the start of encryption. During each round of encryption, we can calculate the next round key in parallel. As a result, we only need to keep the current key k_i in memory and don't need to remember the complete array of round keys. We do however need a register for the array l.

When many blocks are encrypted with the same key, it might seem unefficient to recalculate the round keys for every encryption. The alternative however is to store all round keys (Tn bits) in memory. We prefer only storing the encryption key (mn bits) as the cost of the round key calculation is negligible: the hardware requirements of the round function are minimal and as the key schedule is performed parallel to encryption, we don't lose any time efficiency.

An essential property of a lightweight cipher is its low-area design. We will therefore opt for a folded pipeline rather than a linear pipeline. For many lightweight applications, throughput is not the top priority. The pipeline will contain the hardware for two round functions such that we can perform encryption and key schedule in parallel. In addition, we will need a register for the array l and for the initial key k_0, inputFIFO's for the plaintext and key and and outputFIFO for the ciphertext.
The implementation will use polymorphism to allow for different choices of parameters and we will choose one or two specific sets for synthesis on the FPGA.

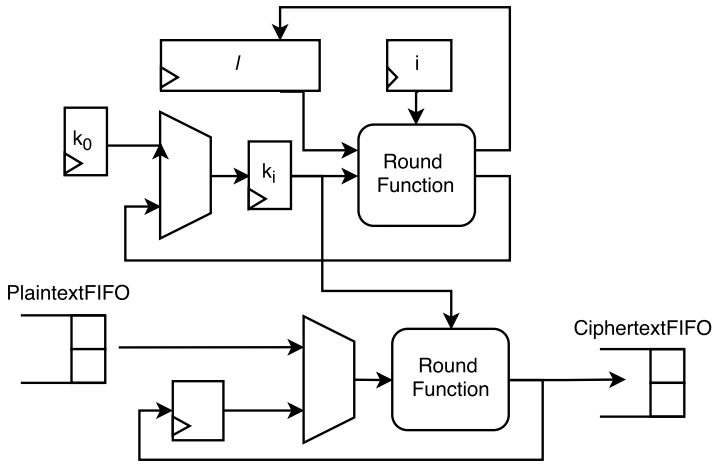

Figure 3: High level design

3.2 Test Plan and Interaction with Host processor

We have a working C implementation and a set of test vectors to check for correctness from the original paper [1]. Since the paper only provides one test for each parameter set, we can use the reference implementation to generate more test vectors for the parameter sets that we use on the actual FPGA.
We use a testing infrastructure similar to that of the Audio Pipeline, using Sce-Mi. A software test bench in c++ feeds the test vectors to port proxies and creates two outputfiles (encryption results and decryption results). We use these to compare the received encryptions to our reference ciphertext and to check that the decryption results match the initial plaintexts. We wrap our SPECK implementation in the `SceMiLayer` to connect it to the Sce-Mi ports. There are three types of communiation between the host processor and the FPGA:

1. Host processor sets key
2. Host processor passes one block of plaintext or ciphertext
3. FPGA passes output to host processor

While there are two separate hardware modules for encryption and decryption, we use the same in- and outputports for both. Our 'Device Under Test' (DUT) combines the two modules and decides which one to use based on an enumeration flag that is sent with the inputs.

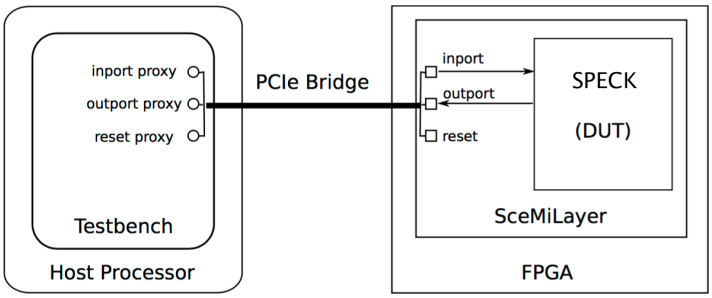

Figure 4: Test infrastructure with Sce-Mi

4 Microarchitectural Description

On the FPGA, our design consists of two modules. The modules, one for encryption and one for decryption, have identical interfaces and nearly identical implementations. They only differ in the definition of the round function (see Figure 3). As their names imply, the encryption module receives plaintext and transforms it into ciphertext. The decryption module receives ciphertext and produces plaintext.

4.1 Interface

SPECK can have a variable word size n, key size mn and number of rounds used for encryption T. The parameters (n, m, T) are used as numeric types for the `EncryptDecrypt` interface, which is implemented by both modules. The interface has three methods:

1. an `Action` method `setKey()` for setting the encryption/decryption key
2. an `Action` method `inputMessage()` for receiving the block of plaintext or ciphertext

3. an `ActionValue` method `getResult()`, returning the encryption/decryption result

These methods were chosen such that a user must only set his key once, when encrypting or decrypting multiple blocks.

4.2 Implementation

4.2.1 Folded design

Each module has one rule implementing the folded pipeline shown in Figure 3. In every cycle, the round function is executed twice (once for the plaintext/ciphertext block and once for the keys) and the round indicator i is incremented. In the first round, when $i = 0$, the plaintext/ciphertext is taken from the inputFIFO instead of a register. In the last round, when $i = T - 1$, the result is put in the outputFIFO, the inputFIFO is dequeued, the round indicator i is reset to 0 and the round key k_i is reset to k_0.

In order to avoid conflicts with the `setKey()` method (both are writing to the key registers k_i), we introduce mutually exclusive guards on the two, using the state of the inputFIFO. It is only logical, that the key should only be allowed to changed when the inputFIFO is empty.

The `setKey()` method receives a key or mn bits or m words. The last three words are stored in registervector l. The round key is initialized with the first word k_0, which is also stored in a separate register for reuse of the key.

The vector l is stored in a vector of registers rather than a register of a vector. This vector is relatively long but is only edited at one index each cycle and at $m - 1$ entries in the `setKey()` method. It is therefore more efficient to implement it with a separate register for each entry.

The `inputMessage()` and `getResult()` methods are straightforward. They respectively receive a block to enqueue into the inputFIFO or dequeue and return the first element of the outputFIFO. They don't conflict with any other method or rule.

4.2.2 Unfolding

We want to explore how introducing parallelism with an unfolded design can increase throughput of the encryption/decryption module and how much we have to pay for this in area.
We thus introduce multiple stages, each with two copies of the round function (one for the key schedule and one for the encryption) and registers keeping the round index, the round key and the intermediate result $\{x_{2i+1}, x_{2i}\}$. We still keep only one copy of the vector l since every stage can write to a different entry.

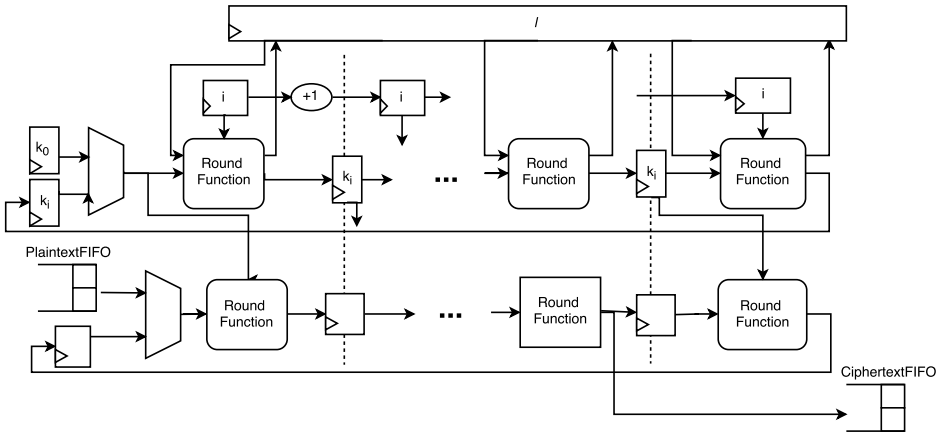

Figure 5: Unfolded pipeline

In the first stage, we need to determine whether to start a new encryption, or to continue with one that is underway. This is indicated by the round of this stage. When it is 0, we must get a new input from the inputFIFO. If there is no input available, we set the round of the next stage to 0. We could use the Maybe type to indicate the validity of a stage input, but since we already have the registers keeping round numbers, we don't need it. A round equal to zero in any stage other than the first stage means that there is nothing to do but passing the round number 0 on to the next stage.
In any stage, a nonzero round number indicates a valid block in the register before the stage. In that case, the stage performs the roundfunctions and puts the results in the register before the next stage (modulo the number of stages of course). When a round register reaches the last round number $(T-1)$, the result is enqueued into the outputFIFO and the next round number is set to 0 instead. This means that computations can start on the next block in the inputFIFO.

5 Synthesis and evaluation

5.1 Folded design

We synthesized for the parameterset $(n, m) = (24, 4)$. With a clock period of 10ns, we obtain a critical path with worst negative slack equal to 3.29ns. Both the decrypt and encrypt module require a total of 940 flip-flops. There are 608 LUT's for the decrypt module and 622 for encryption.

To measure the throughput, we implemented a separate module that passes a fixed number of blocks to the encryption module. As the inputs and outputs have identical formats, there is no need to hardcode a long vector of input plaintexts. The module stores the outputs of the module in a fifo, so they can be feeded back. When the fifo is empty, a random input is chosen such that the encryption module receives something every cycle. With one million inputs, both the encryption and decryption module finish after 0.47 seconds. As each input block consists of $n = 48$ bits in our implementation, this corresponds to a throughput of around 100 Gbps.

5.2 Design exploration

We also synthesized the pipeline with 2 to 5 stages. Table 2 shows the resulting area and throughput.

# stages	# flip-flops	# LUT's	Throughput (Gbps)
1	940	622	100
2	1036	1548	188
3	1132	1994	274
4	1288	2951	355
5	1324	3571	417

Table 2: Area vs Throughput trade off for unfolding the pipeline

5.3 Implementation Evaluation

6 An encryption tool for practical use

6.1 Mode of operation

For practical use of the implementation, we need to encapsulate the encryption module in a block cipher mode of operation for cryptography. Our module can encrypt blocks of $2n$ bits. In the very likely case that one wants to encrypt a message that is longer than that, it is not considered safe to simply split the message into blocks of $2n$ bits and put them through the encryption module independently. This mode is known as Electronic Codebook (ECB) mode (see Figure 6).

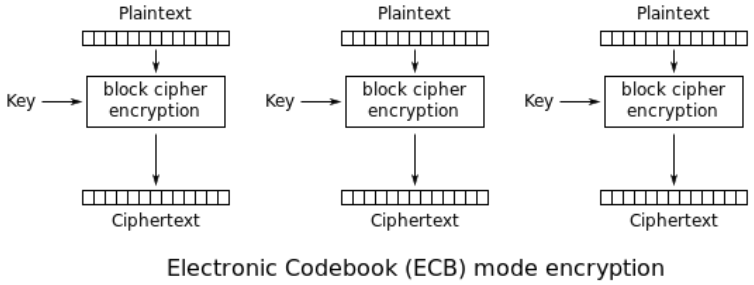

Figure 6: Electronic Codebook (ECB) mode of encryption

Figure 7 demonstrates why ECB is not a good mode of encryption.

(a) Message

(b) What encryption looks like with ECB

(c) What encryption should look like

Figure 7: Why ECB is not a suitable mode of encryption

Instead, we implement the Output Feedback (OFB) mode of encryption, shown in Figure 8.

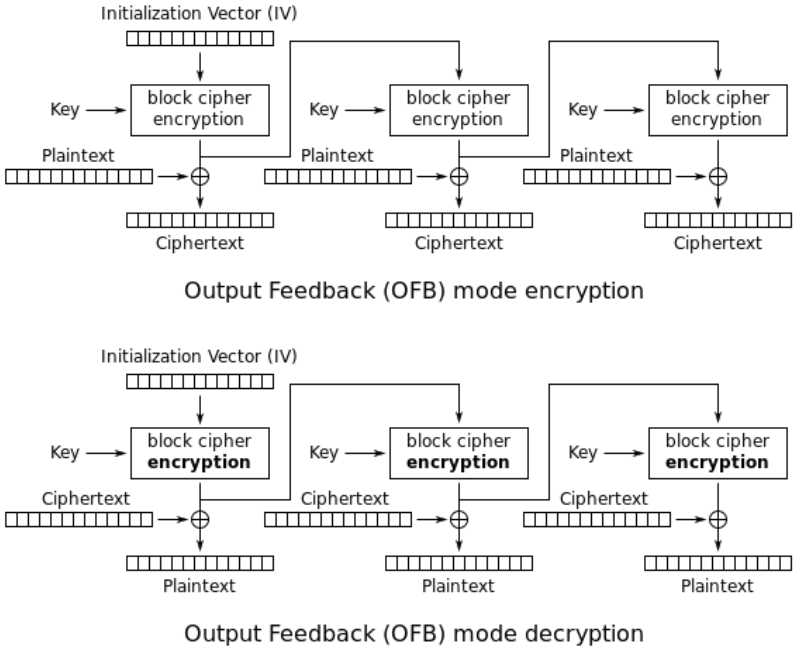

Figure 8: Output Feedback (OFB) mode of encryption

This mode of operation is not parallelizable, but as we still want to keep the area of our design as small as possible, this is not considered a problem. A very convenient advantage of the output feedback mode is that no decryption module is needed to recover the message. Since the mode's approach is to use the encryption module to generate a stream of "random" keys which are XORed with the plaintexts, we can reuse the encryption module when we want to invert this XOR operation. As a result, the hardware for encryption and decryption is identical (see figure 8).

6.2 Interface

The `OperationMode` interface is very similar to the `EncryptDecrypt` interface. It also takes the numeric types (n, m, T) and has the following three methods:

1. an `Action` method `setKeyIV()` for setting the encryption key and initialization vector
2. an `Action` method `inputMessage()` for receiving a block of plaintext or ciphertext
3. an `ActionValue` method `getResult()`, returning the encryption/decryption result

The only difference with the `EncryptDecrypt` interface is that apart from a key, we must also be able to set an IV. Finally, we need one more method to explicitly reset the module. As Figure 8 shows, once an IV is set, it will be encrypted endlessly. We thus also provide an `Action` method `reset()` to allow the module to be used with a new IV. This is for example necessary when one wants to start decrypting after encryption.

6.3 Implementation

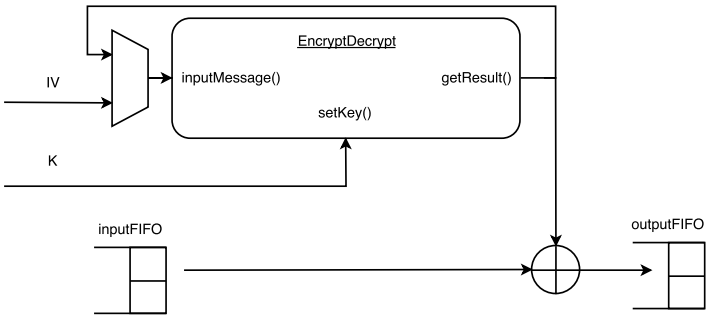

Figure 9: Design of the OFB module

Figure 9 shows a folded version of Figure 8. The `SetKeyIV()` method of the `OperationMode` interface simply passes the key to the `SetKey()` method of the encryption module and introduces the IV as a plaintext block.

There is one rule responsible for getting the encryption result from the encryption module, xoring it with an input, resubmitting it to the encryption module and passing the result of the xor operation to the outputFIFO. The rule is made mutually exclusive with the `SetKeyIV()` method by means of a Boolean flag `started`. The `SetKeyIV()` method, which can only be used when `started` is False, changes the flag to True. The `reset()` method sets the flag to False. In that case, a second rule empties the resultqueue of the encryption module such that remaining encryptions of the IV cannot be mixed with those of the next IV.

The `inputMessage()` and `getResult()` methods are again straightforward. They respectively enqueue inputs into the inputFIFO or dequeue and return the first element of the outputFIFO.

6.4 Synthesis

Also for the OFB implementation, we use a clock period of 10ns, resulting in a worst slack of 2.448ns. The critical path is thus 7.552ns. The overhead of the OFB module causes an

increase of 0.85ns.

The implementation requires 782 LUT's and 1137 flip-flops and the result can encrypt messages at a throughput of 97 Gbps. As expected, this throughtput is very close to the throughput of the 1-stage encryption module.

6.5 Test environment and results

The SceMiLayer and testbench for this module are very similar to the original ones, the only difference being that the testbench must be able to send an IV with the key. Furthermore, as the OFB mode is meant to encrypt/decrypt a series of blocks belonging to one message, we also change the input to our testbench from a file of $2n$-bit blocks to a file containing a message (like for example an e-mail). The testbench encodes the e-mail to ASCII hexadecimal format and splits it into $2n$-bit blocks itself before sending them to the FPGA. Similarly, the FPGA outputs are reconverted to readable text format, though an encrypted message will of course look like gibberish. The testbench then resets the DUT and uses this encrypted file as input for the decryption (or equivalently, another encryption with the same key and IV). The message resulting from decryption perfectly matches the initial message.

References

[1] R. Beaulieu et al. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. `http://eprint.iacr.org/`.