Electronics and Computer Science
Faculty of Physical and Applied Sciences
University of Southampton

Author: Lewis Smith

April 30, 2018

Low Power Hardware Accelerated Internet of Things Cryptography

Project Supervisor: Mark Zwolinski
Second Examiner:

A project report submitted for the award of MEng Electronic Systems with Computer Systems

Abstract

Contents

Acknowledgements

Chapter 1

Introduction

As the speed and global reach of the internet has expanded over the years the number of devices connected to it has rapidly increased. These devices are no longer just the servers and the PC's connected to them, they now include consumer devices like smartphones, tablets, and games consoles. However, even more recently the idea of connecting the internet to various 'dumb' appliances like: simple light switches; kettles; fridges and many more; to make them 'smart' devices that can be controlled through small embedded processors has emerged. The idea of connecting such devices to internet has been dubbed 'Internet of Things' or 'IoT' and has the aim to make our lives simpler. The 'IoT' concept is also being explored for more industrial applications such as: automated factories; city electrical grids; and even a network of self-driving cars[1] but this is far more advanced than the basic 'smart' home.

Due to the wide range of products that a connected IoT device can be applied to improve the efficiency and/or usefulness, it has been predicted that billions of devices will be in use by 2020[2]. This also means that the complexity of the devices varies greatly. The one thing that all of these devices have in common though is that they need to be secure as they communicate sensitive and private data through an open channel on the internet between the user and the device. To keep the potential adversaries from accessing the data and possibly controlling numerous connected devices, maliciously or not, an encryption algorithm can be used.

POSSIBLY CHANGE/IMPROVE BELOW

There are many encryption algorithms that perform this function and most can be implemented in both software and dedicated hardware such as an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). As a majority of IoT devices are implemented on small embedded processors which have limited resources, the hardware option might possibly be a better solution for IoT devices. However, due to the fact that most IoT devices are always on, and likely battery powered, power consumption is a very important factor when considering options for adding hardware accelerated encryption and for battery powered devices it is often more critical than the actual encryption.

The goals of this project are to explore various encryption algorithms and compare their performance based on data throughput, accuracy, security and power consumption when implemented in software and hardware. To evaluate these parameters the

same algorithms can be coded in C or C++ for the software versions and a Hardware Development Language (HDL) such as System Verilog can be first simulated in ModelSim, before programming a FPGA for the hardware version. These comparisons can then be used to match the algorithms to the appropriate IoT device as they all have different requirements for relative security level and power consumption, as for example a light switch does not necessarily need to be protected from the same level of attack as a set of digital locks or private data storage. In order for the hardware to work with IoT devices it will also need a communication protocol like I^2C or SPI to work with embedded processors, and possibly Ethernet or WiFi to act as the gateway to internet for the device. Some of these protocols are available on FPGA development boards but can be implemented in System Verilog code.

Chapter 2

Background Research & Literature

2.1 Internet of Things

CHANGE/IMPROVE BELOW [100]

As mentioned in chapter 1 there are many IoT devices that require varying levels of security and have to be protected against different of attacks, like side channel attacks. The purpose of this subsection is to discover what these devices are and what properties are required in the encryption algorithm in terms of throughput, power or energy consumption, software or hardware restrictions and security. Due to IoT devices having limited resources a provisional limit of 2000 Gate Equivalents in hardware is the maximum size for most embedded platforms but even that might be to big for devices like RFID tags[**?**]. Power consumption should also be kept to as little as possible but a limit of tens of micro Watts (μW) for RFID tags is suggested[**?**].

2.2 Cryptography

POSSIBLY CHANGE/IMPROVE BELOW

After evaluating the conditions required to provide the appropriate level of security in section 2.1 this subsection explores the cryptographic principles and algorithms available that satisfy those conditions. The primary objective of cryptography is to convert, or encrypt, a readable message known as plaintext into an unreadable form, ciphertext, so that adversaries cannot read the contents, but over the years the scope of cryptography has widened. Throughout history encryption has been has been used allow people and groups to exchange secret messages, especially in times of war. Since the early transposition and substitution ciphers, where each character in a message are rearranged and replaced by others a certain number further down the alphabet respectively, encryption has evolved to include techniques for identity authentication, integrity checks and much more[insert reference]. Cryptography is therefore the study of encryption and other techniques, including identity authentication and integrity

checks. Its counterpart: the study of breaking the encryption to find the original message, is known as cryptanalysis[?]. Eventually, for most encryption techniques a weakness is found, and subsequently exploited, so more complex techniques are conceived and with the invention of computers the complexity of the algorithms has increased greatly. However, the computer power is also available for cryptanalysis so the cycle of continuous improvement of the algorithms hasn't stopped.

In cryptography there are two main concepts that the algorithms are based on: symmetric and asymmetric keys[insert reference]. In asymmetric key cryptography a unique key is used to encrypt data and different, but a related key, is used to decrypt it. The relationship between the two keys is often defined by maths problem that is very difficult to solve which is the basis of the encryption[insert reference]. On the other hand, symmetric key cryptography uses the same key for both encryption and decryption, hence symmetric, with the security usually provided by a combination of simple logic operations[insert reference].

2.2.1 Asymmetric Key

POSSIBLY CHANGE/IMPROVE BELOW

Asymmetric key cryptography can be referred to as public key due to the fact that one of the related keys can be publicly available without compromising the security of the encrypted data. This is because the keys are usually generated based on mathematical problems that have no solution or the solution is impossible for a computer to solve efficiently, such that solving it takes longer than an exhaustive key search[?]. There are many problems that fit this criteria but the most popular in use today are the integer factorization and elliptic curve problems used by the RSA[insert reference] and the family of Elliptic-curve cryptography (ECC) techniques[insert reference] respectively. Public key cryptography can be used in two different modes as if data is encrypted with the intended recipients public key only they can decrypt it with their private key, thus encryption. However, if a private key is used for encryption then using the public key to decrypt it ensures the senders identity, authentication[?].

2.2.2 Symmetric Key

POSSIBLY CHANGE/IMPROVE BELOW

Similar to the symmetric/private comparison symmetric key cryptography is also known as private key, as in order to keep the encrypted data secure the key used must be kept secret. There are two main types of private key algorithms that operate on the plaintext differently: block ciphers which uses a fixed number of bits, block; or stream ciphers which encrypts data bit by bit[?].

Modern block ciphers are based on Claude Shannons work on product ciphers[?], in which he suggested that iterating a cipher for multiple rounds, with subkeys, improves the security. Hence, the cipher to be iterated didn't need to be complex operations and simple logic operations such as XOR, substitution or permutation of the plaintext could be used[insert reference]. The base cipher that is iterated is

known as the round function and it takes as an input a block of plaintext and a subkey, which is generated from the main key by a separate key expansion function, and outputs a block of ciphertext. The output is usually the result of the round function XORed with the subkey. The round functions are mostly designed using either a Feistel network[?] (F network) or a Substitution Permutation network (SP network)[?].

The Fesitel network was named after physicist Horst Feistel who was a integral part of the team at IBM that developed the early block cipher Lucifer, which of course used a Feistel network[insert reference]. The F network works by splitting the input plaintext into two equal words, known as the left (MSB) and right (LSB) words. The round function is then applied to the right word before the result is XORed with the left word and then the words are swapped over and iterated as in Equation 2.1 and 2.2, with the ciphertext being equal to (R_{n+1}, L_{n+1}) where n is the number of rounds iterated. The advantage of using a F network is that decryption is just applying the same algorithm but with the sub keys in reverse as in Equation 2.3 and 2.4, with the ciphertext (R_{n+1}, L_{n+1}) as the input and the plaintext (L_0, R_0) returned.

$$L_{i+1} = R_i \tag{2.1}$$

$$R_{i+1} = L_i \oplus F(R_i, K_i) \tag{2.2}$$

$$R_i = R_{i+1} \tag{2.3}$$

$$L_i = R_{i+1} \oplus F(L_{i+1}, K_i) \tag{2.4}$$

On the other hand, Substitution Permutation networks operate on the whole plaintext block using S-boxes for substitution and P-boxes for permutation. Individually, these operations aren't particularly strong as a S-box and a P-box can be thought of as simple substitution and transposition ciphers respectively. However, when combined in a SP network over multiple rounds the security can be very strong due to Shannon's confusion, provided by the S-boxes, and diffusion, P-boxes, properties being satisfied[?]. The S-boxes usually take in a certain number of bits and outputs the same number of bits but of a different value. P-boxes are then used to spread the bits around such that the output of the S-boxes are used by as many S-boxes in the next round. After the S-boxes and P-boxes and before the next round occurs the block is XORed with the round key so the round equation is Equation 2.5. Decryption, Equation 2.6 is achieved using inverted S-boxes and P-boxes and the round keys in reverse order which means that different hardware or operations are needed.

$$B_{i+1} = F(B_i) \oplus K_i \tag{2.5}$$

$$L_i = R_{i+1} \oplus F(L_{i+1}, K_i) B_i = F'(B_{i+1}) \oplus K_i \tag{2.6}$$

CHANGE/IMPROVE BELOW

Stream ciphers were initially designed to approximate the One Time Pad (OTP) cipher that was proved to be completely unbreakable by Claude Shannon[insert reference]. The OTP works by combining each digit of the plaintext with a completely random keystream. The stream ciphers work by generating a pseudo-random keystream to combine with the plaintext[?]. Because the keystream is pseudo-random

and not completely random a stream cipher is breakable. The keystream is created by a pseudo-random number generated with a cryptographic key used as a seed.

There are also some modes of operation for block ciphers, in [?], that provide better security by using feedback of the ciphertext to the next block. Some of these modes of operation also allow block ciphers to behave similar to stream ciphers as they encrypt an initialization vector with the key and the resulting ciphertext can be combined with the plaintext.

2.2.3 Decisions

POSSIBLY CHANGE/IMPROVE BELOW

Due to the fact that asymmetric key algorithms are hard to solve they require complex hardware or software to implement which is undesirable for this project. Also, with the exception of ECC the key sizes needed for the security can be very large so with the limited IO pins available on FPGAs they could prove difficult to program. On the other hand, many private key algorithms are designed to be efficient in hardware especially Feistel networks as an inverted round function isn't required. While a stream cipher can be useful to encrypt serial data that will most likely be the source, the modes of operation available for block ciphers provide more flexible functionality including stream cipher modes. Therefore, the algorithm chosen for this project will most likely be a block cipher with a Feistel network.

2.3 Conventional Algorithms

POSSIBLY CHANGE/IMPROVE BELOW

There are many block ciphers that are considered very secure and therefore popular, they include: DES[?], AES[?], Blowfish[?]. DES operates on a block of 64 bits for 16 rounds using a key length of 64 bits but it has an effective key length of 56 bits as 8 bits were used for parity. AES, an upgrade to DES, is far more secure as uses a 128 bit block and has the flexibility of using three different key lengths: 128, 192 and 256. The number of rounds that AES iterates depends on the key length with 10 rounds used for a 128 bit key, 12 for 192, and 14 for the largest key. Blowfish, like DES, operates on a 64 bit block and iterates for 16 rounds, but it can use a variable key length in the range 32 to 448 bits.

2.3.1 Standardization

DES, which stands for Data Encryption Standard, is one the earliest block ciphers used in the computer age and it was developed by IBM in the 1970s based on their earlier cipher Lucifer[insert reference]. As with Lucifer it was designed around a Feistel network but the round function used also has a SP network structure to it[insert reference],but the S-boxes aren't a one-to-one function but rather output 4 bits from a 6 bit input. It has the name Data Encryption Standard as it was

accepted as the standard encryption algorithm by the US National Bureau of Standards (NBS), now the National Institute of Standards and Technology (NIST), in 1977 after it was altered by the National Security Agency (NSA), which caused some controversy[?].

DES was used for about two decades but in the 1990s several successful attacks proved its weakness[?] so in 1997 NIST started a selection process to find its replacement. Due the controversy of the NSAs involvement and comments from the cryptography community the selection process was as transparent as possible[insert reference]. Many algorithms were submitted as candidates for the standard but the finalists were: MARS, RC6, Rijndael, Serpent, and Twofish[insert reference]. It took three years to decide on the algorithm to be set as the standard which was announced as Rijndael in 2000 and the standard was et in 2001, with the 128, 192, 256 bit keys being used in the standard[?]. Unlike DES, AES uses a SP network as it is efficient, in time, in both hardware and software. The round function treats the block as a $4x4$ byte matrix and performs multiple steps on the data: sub bytes; shift rows; mix columns and add round key[insert reference].

Since its standardization in 2001 AES has been used almost exclusively because its security is trusted. Because of this there are many different software and hardware implementations produced with some concentrating on side-channel attack resistance[?] or efficient S-box implementations[?]. However, even area optimized designs like [?] use 2400 gate equivalents which is too much for lightweight applications.

2.3.2 Other Algorithms

The US standardized algorithms quickly became very popular and can be considered the unofficial global standard. Although, there are many other algorithms that are considered secure and are commonly used. One of these algorithms, TripleDES, is actually based around DES which increases the security by encrypting the plaintext three times with separate keys making the effective key length of 168 bits. These algorithms might be used because there is still some distrust of the NSAs involvement in the algorithms and they are more open source. This is the case of the Blowfish algorithm as it is unpatented and can therefore be used in any product without legal consequence.

Blowfish was designed by Bruce Schneier in the early 1990s as he, and many others, noticed the insecurity of DES particularly with the 56 bit key length making a brute force attack more plausible[?]. The design of the algorithm is based around a Feistel network with, similar to DES, the round function using S-boxes. The sub keys and S-box lookup tables are generated using the hexadecimal digits of pi which are provided by the designers[insert reference]. As with AES there are many FPGA and ASIC implementations of the Blowfish algorithm, [?] and [?], but they require too many FPGA resources to be considered for the lightweight nature of this project.

2.3.3 Decisions

Due to the conventional algorithms not being explicitly designed for hardware and definitely not for lightweight applications they are not appropriate for this project. Although, they are useful for comparison with the lightweight algorithms in section 2.4 in terms of security and throughput.

2.4 Lightweight Algorithms

After deciding that the conventional algorithms might not be suitable for the low power devices targeted by this project, some more lightweight algorithms were found including: PRESENT[**?**], PRINCE[**?**] and the SIMON and SPECK algorithms[**?**]. However, as IoT is an emerging technology and is the main reason for lightweight cryptography there isn't a standard set by NIST, but the process has begun in [**?**]. These algorithms are considered lightweight because they make sacrifices in and security or throughput, or both, to achieve small area and low power designs.

2.4.1 SIMON & SPECK

The SIMON and SPECK family of algorithms are the lightweight techniques proposed by the NSA that were designed to perform well in both software and hardware while still being secure; and to be flexible in terms of block and key size, listed in Table 2.1. The algorithms are similar but SIMON was optimised for hardware implementations and SPECK for software. As with AES the number of rounds iterated depends on the key size but as the block size varies as well it also has an effect as shown in Table 2.1. The structure of both algorithms is a Feistel network and thus it works on words of N bits, where $2N$ is the block size, and with a key of $M * N$ bits. This lends it self to the naming format of SIMON or SPECK $2N/N * M$, which means, for example, SIMON48/72 has a word size of 24 and uses 3 words for the key[insert reference].

Block Size	Key Size	N	M	SIMON Rounds	SPECK Rounds
32	64	16	4	32	22
48	72	24	3	36	22
48	96	24	4	36	23
64	96	32	3	42	26
64	128	32	4	44	27
96	96	48	2	52	28
96	144	48	3	54	29
128	128	64	2	68	32
128	192	64	3	69	33
128	256	64	4	72	34

Table 2.1: A table of the modes of operation for the SIMON & SPECK Algorithms. Adapted from [**?**].

Even though they are relatively new algorithms there are still a few FPGA implementations available for review, including [**?**] and [**?**] as well as those provided in [**?**]. These all show that very small designs are possible with even the 128/256 versions fitting below the 2000 GE limit.

2.4.2 Other Algorithms

The PRESENT cipher is another option for lightweight cryptography as it achieves a 1570 GE FPGA design with a 80 bit working on a 64 bit block. there is also an version that uses 128 bit key. The design of the cipher is based around a SP network, **??**, with 64 bit subkeys.

PRINCE is a lightweight cipher that can encrypt data in one clock cycle with an unrolled SP network, as SP networks require less rounds than a F network. The steps include S-boxes and a matrix layer as well as the XORing of the round key and a round constant. Due to the unrolled nature of the algorithm the FPGA implementations are mainly combinational so the register count is lower than other algorithms.

2.4.3 Decisions

Based on the research explored in chapter 2 I chose the SIMON and SPECK algorithms from the NSA, mainly because of their flexibility in security levels, with different key lengths, that could be applied to the different devices explored in section 2.1. This means that multiple algorithms don't need to be developed and I could concentrate on making my code as efficient as possible. When compared in terms of power consumption the SIMON64/96 version in [**?**] also shows lower power consumption than the other algorithms explored. Also, while both PRESENT and PRINCE meet the lightweight specification they are also SP networks and in subsection 2.2.3 it was decided that a Feistel network is preferable.

After deciding on the SIMON and SPECK family I explored how each version works in order to make a more informed decision on which to work with in this project. As SIMON was designed primarily for hardware it only makes use of XOR ($\oplus$), AND ($\&$) and circular rotate operations ($R^j[x]$) on the n bit wide words. For the rotate operation the word x is rotated by j bits to the left or right if j is negative. The encryption and decryption functions take the Feistel network form described in **??** with the round function Equation 2.7.

$$F(x) = (R^1[x] \& R^8[x]) \oplus R^2[x] \tag{2.7}$$

SPECK on the other hand, being optimised for software implementations uses XOR ($\oplus$), modulo 2^n addition ($+$) and circular rotate operations ($R^j[x]$), with the circular rotate being equivalent to what is used in SIMON. The encryption and decryption functions take a slightly different form to the basic Feistel network, **??**, and are shown in Equation 2.8 and 2.9 where $\alpha = 7$ and $\beta = 2$ if $n = 16$, but $\alpha = 8$ and $\beta = 3$ otherwise.

$$L_{i+1} = (R^{-\alpha}[L_i] + R_i) \oplus K_i \tag{2.8}$$

$$R_{i+1} = R^{\beta}[R_i] \oplus (R^{-\alpha}[L_i] + R_i) \oplus K_i = R^{\beta}[R_i] \oplus L_{i+1} \tag{2.9}$$

As the aim of this project is to compare how an algorithm performs in hardware and not software SIMON was chosen even though SPECK shows almost as good performance in hardware and much better in software.

Chapter 3

Previous Work & Initial Design Approaches

Unfortunately, unlike some of the algorithms explored in chapter 2 there isn't a standard SIMON/SPECK software library available for use and benchmarking with my design. For that a reason a software version needed to be developed that offers similar functionality to the hardware version, so both software and hardware were developed in parallel. Also, by working on the software version, in C or C++, it allowed me to increase my familiarity with the algorithm and provided some insight in to how the hardware version, in System Verilog, would be designed, and vice versa.

For all versions of the SIMON algorithm developed for this project the most important factor is efficiency in terms of power consumption and resource use, with time efficiency not being an initial priority. All versions of this algorithm were developed not only with the description but also the pseudocode[insert reference]. In that document there is also a set of test vectors that define the ciphertext the algorithm should produce with given key and plaintext. Due to Feistel network structure of SIMON the decryption is just the same as encryption but with the key schedule reversed. This means that the encryption can be done in parallel to the key expansion but decryption requires the key to be pre-expanded. As some of the block cipher modes of operation[insert reference] only require encryption, two variants might be useful: one that can just encrypt data; and one with full functionality.

3.1 First Approach

The first approach for this project was to develop the software and hardware to be flexible to the various data block and key sizes defined for SIMON, shown in Table 2.1. As each system would only be computing with one mode at a time the decision of the variable sizes could be made at compile time and not during in runtime which be introduce some unnecessary inefficiency to the system. This could be achieved with C preprocessor macros for the software and, even though something similar is possible in System Verilog, parameters were passed between the modules. The macros and parameters were used to define the n and m values for each mode so

that the input and output variables of each function in software and each module in hardware would be correct. They were also used to define the number of rounds that the round function should iterated and the value of j used in the key expansion.

3.1.1 Software - Hosted C

Initially the software was only developed to be used in a hosted environment but future approaches could made be for an embedded processor or something similar. With most languages used in software development the variable sizes and constrained to the data types available, with C offering only 8, 16, 32 and 64 bit variable types. This produced some difficulty with the 24 and 48 bit sizes required for this algorithm, but with the use of typedefs and bitfields in a structure it was solved. Using the values of N and M a *word* data type was defined based on unsigned integers, uint$N$_t, and the block, key and key schedule data types were setup using arrays of the base *word* type. All of the code used to setup the data sizes and type can be seen in Listing 3.1.

```
#if      defined S32_64
    #define    N    16
    #define    M    4
    #define    T    32
    #define    j    0
#elif    defined S48_72
     ...
#elif    defined S48_96
...

typedef uint32_t     uint24_t;
typedef uint64_t     uint48_t;
#define TYPE_(x)     uint ## x ## _t
#define TYPE(x)      TYPE_(x)
#define UINT(x, n)   typedef struct n    {    TYPE(x) v : x;   }    n;
UINT(N, word);

typedef word         block[2];
typedef word         key[M];
typedef word         keys[T];

```

Listing 3.1: Macro definition of the word, block, key and key schedule types.

After the data types were defined the rotate functions, which are used frequently in algorithm, were made to rotate the bits in a *word* around a given number of bits. These rotate functions were used in the Feistel and round functions which is iterated for both encryption and decryption. The code used for these functions can be seen in Listing 3.2 but a full listing is available in the Appendix.

CHANGE/IMPROVE BELOW

```
TYPE(N) F       (word x                    )
{
    return (ROTL(x,1) & ROTL(x,8)) ^ ROTL(x, 2);
}

void    ROUND   (block b,   word k        )
{
    word tmp = b[0];                       //  SAVED TO SWAP
    b[0].v = b[1].v ^ F(b[0]) ^ k.v;       //  GENERATE NEW WORD
    b[1] = tmp;                            //  SWAP FROM SAVED
}
```

Listing 3.2: Round and Feistel functions

```
void    KEXP_PRE    (keys ks,   key k         )
{
    word tmp;
    int8_t i;
    //  LOAD KEY
    for(i=M; i>0; i--)  ks[i-1].v = k[M-i].v;

    //  GENERATE NEW KEYs
    for(i=M; i<T; i++)
    {
        tmp.v = ROTR(ks[i-1], 3);

#if (M == 4)    tmp.v ^= ks[i-3].v; #endif

        tmp.v ^= ROTR(tmp, 1);
        ks[i].v = ~ks[i-M].v ^ tmp.v ^ (z[j][(i-M) % 62]) ^ 3;
    }
}

void    KEXP_INL    (key k, TYPE(8) i         )
{
    //  GENERATE NEW KEY
    word tmp;
    tmp.v = ROTR(k[M-1], 3);

#if (M==4)  tmp.v ^= k[M-3].v;  #endif

    tmp.v ^= ROTR(tmp, 1);
    tmp.v ^= ~k[0].v ^ (z[j][i % 62]) ^ 3;

    //  SHIFT KEYS
    k[0] = k[1];

#if (M>2)   k[1] = k[2];        #endif

#if (M>3)   k[2] = k[3];        #endif

    k[M-1] = tmp;
}
```

Listing 3.3: Key Expansion Functions

The key expansion functions, Listing 3.3, also use the rotate functions but they are more complicated than the round function as they differ slightly depending the number of words in the key, M, but more compiler directives can handle that problem. There are two separate key expansion function: one that can expand the key at the same time as the encryption in the iterative loop (INL); and one that computes the schedule before encryption or decryption occurs in its own iterative loop (PRE).

For simplicity in the early stages the the software didn't have the capability of reading a file and encrypting or decrypting it so it could only use data stored in the code. This still allowed for the test vectors to be used and therefore correct functionality could still be confirmed. By simply changing the desired mode at compile time and setting up the test vectors for each mode they could be tested with the same code.

The results of testing the code, in mode 0 (32/64), can be seen in [reference figure]. For the in loop key expansion the test plaintext and key was inputted and encrypted with the result compared with the expected ciphertext. With the pre loop key exapansion the encrypted ciphertext could also be decrypted so when that produced the original plaintext then the whole system was confirmed to function correctly.

INSERT TESTING RESULTS FIGURE

3.1.2 Hardware - System Verilog

As mentioned in section 3.1 the parameter system was used to pass the values for word and key size and also the number of rounds to be iterated. Similar to the functions used in subsection 3.1.1 modules were used for the Feistel function, round function, key expansion function before being combined in the top level control module. Although, unlike the software version the rotate and logic operations used could be done with just combinational logic so the result is available almost instantaneously not after a few CPU instruction cycles.

```
1  module SIMON_function
2  #(
3  parameter               N =     16,
4  parameter               M =     4
5  )
6  (
7  input logic [N-1:0]      in,
8  output logic [N-1:0]     out
9  );
10
11 always_comb
12 begin
13     out = ({in[N-2:0], in[N-1]} & {in[N-9:0], in[N-1:N-8]}) ^ {in[N
       -3:0], in[N-1:N-2]};
14 end
15
16 endmodule
17
```

Listing 3.4: The feistel function

Listing 3.4 shows how the Feistel function is implemented in hardware with just combinational logic and wire shifts. A similar method is used for the shifts in the key expansion. The round function, Listing 3.5, was then designed using an instance of the Feistel function with the input being the the top half of the input block of the round function and the result being assigned to the f logic block. The result of the round function is then calculated in another combinational block.

```
1  module SIMON_round
2  #(
3  parameter               N =     16,
4  parameter               M =     4
5  )
6  (
7  input logic [2*N-1:0]   in,
8  input logic [N-1:0]     key,
9  output logic [2*N-1:0]  out
10 );
11
12 logic [N-1:0] f;
13
14 SIMON_function func(.in(in[2*N-1:N]), .out(f));
15
16 always_comb
17 begin
18     out[N-1:0] <= in[2*N-1:N];
19     out[2*N-1:N] <= in[N-1:0] ^ key ^ f;
20 end
21
22 endmodule
23
```

Listing 3.5: The Round function

```
1  always_comb
2  begin
3      unique case(current)
4      s0:
5      begin
6          if(newData)          next = s1;
7          else                 next = s0;
8      end
9      s1:                      next = s2;
10     s2:
11     begin
12         if(count != T-1)     next = s2;
13         else                 next = s3;
14     end
15     s3:
16     begin
17         if(readData)         next = s0;
18         else                 next = s3;
19     end
20     endcase
21 end
22
```

Listing 3.6: The next state logic for the (INL) control module

The main functionality of the system was provided by the control module which provides the sequential operation with the clock (clk), and active low reset signal (nR). As with the software version two control modules were created: one that expands they key while encrypting (INL) and one that waits for the key to be fully expanded before the data processing begins (PRE). They both achieve this by operation as a state machine with with different states: waiting for data; loading data; processing data; and writing the output data. The state is controlled by the combinational block which is dependant on the current state; the inputs: *newData*, *readData* and *enc_dec*; and the internal *count* variable and the output *doneKey*. The combinational blocks that implement the next state logic is shown in Listing 3.6 and 3.7.

```
always_comb
begin
    unique case(current)
    s0:
    begin
        if(newData && doneKey)  next = s1;
        else                                    next = s0;
    end
    s1:                                         next = s2;
    s2:
    begin
        if(count != T-1)                next = s2;
        else                                    next = s3;
    end
    s3:
    begin
        if(readData)                        next = s0;
        else                                    next = s3;
    end
    endcase
end

```

Listing 3.7: The next state logic for the (PRE) control module

To test the individual modules various input data was selected and the expected outputs were calculated either manually for the simple operations or using the software. As with the software when this functionality was correct they were combined into the top level control module to be further tested. Again, this was only for the most basic 32/64 mode of SIMON. To do this a testbench was created that with a clock that had a $100ns$ period, or $10MHz$ [CHECK], and with the initial inputs setup during the reset of the module.

After this the reset signal could be disabled and then the *newData* signal could be set $HIGH$ to indicate to the state machine to move into the loading and processing states. After an appropriate delay the *newData* signal is reset and then testbench waits for a rising edge of the *doneData* signal before asserting the *readData* signal. The testbench for (INL) then just resets the *readData* signal and finishes as the encryption is finished. Although, the other testbench reads the outputted ciphertext is read into the the plaintext input and the *enc_dec* signal is inverted. This informs the module to decrypt the plaintext when *newData* is next set. The same process is the repeated and the testbench finishes after the decryption is complete. When

the module finished processing the data the ciphertext outputted could read by the testbench and then compared with the expected results.

INSERT TESTING RESULTS FIGURE

As this approach was developed in the early stages of the project and didn't offer much functionality or perfomance enhancements it was never synthesised for an FPGA.

3.2 Second Approach

With the basic functionality working in section 3.1 improvements could be made to the power consumption and resource use with methods like: code minimisation; logic minimisation; clock gating. Also more functionality could be added to increase the usability of the algorithm. As the aim of this project is to design an efficient version of this algorithm in hardware not software, most of the changes in this approach were made in the System Verilog code.

3.2.1 Software - Hosted C

There were only some minor changes to the software version for this approach which were mainly what was learnt from developing the System Verilog version, subsection 3.1.2. One of these changes was ensuring that the key schedule was stored, before or during the main loop, and only expanded once. This was done with a *doneKey* flag variable which is set to true at the end of the key expansion.

One other improvement was to add basic file reading capability to the software which allowed any '*.txt*' file to be read and stored in data blocks to be processed. Although, the same test data was used for the key and first data block to ensure that the cipher was still performing correctly.

ADD C LISTINGS BELOW [100]

ADD C TESTING BELOW [200]

3.2.2 Hardware - System Verilog

To add the clock gating to the control module wasn't simple as the clock should not be halted while data is being processed or written to the output, so it could only be updated during the wait state. This was attempted with the clock gating signal only updating in this state on the same rising edge as the state machine. However, this caused glitches because the state machine moved onto the next state before the clock was stopped and was thus no longer in the correct state the clock gating when required. This was because the same edge that is needed to update the control signal moves the machine onto the next state. One option to solve this is to add more signals and a latch to have the gated clock to only stop when the clock is *LOW*. A similar method to this was eventually used as the clock gating signal sensitivity was

changed to the falling edge of the clock, hence updating half a clock cycle before the system changes state.

```
1  //  GATED CLOCK LOGIC
2  logic clkCipherGo, clkCipher;
3  logic clkKeyGo, clkKey, clkAll;
4
5  assign clkCipher = clk && clkCipherGo;
6  assign clkKey = clk && clkKeyGo;
7  assign clkAll = clkCipher || clkKey;
8
9  always_ff @(negedge clk, negedge nR)
10 begin
11     if(~nR)
12     begin
13         clkCipherGo <= 1'b0;
14         clkKeyGo <= 1'b0;
15     end
16     else
17     begin
18         unique case(current)
19         INIT:
20         begin
21             clkCipherGo <= newData;
22             clkKeyGo <= newKey;
23         end
24         LOAD:
25         begin
26             ENCDEC <= enc_dec;
27             if(enc_dec) clkCipherGo <= newData && (newKey || doneKey);
28             else        clkCipherGo <= newData && doneKey;
29             clkKeyGo <= newKey;
30         end
31         EXECUTE:
32         begin
33         end
34         WRITE:
35         begin
36         end
37         endcase
38     end
39 end
40
```

Listing 3.8: The next state logic for the (PRE) control module

Gating the clock also enabled a few more enhancements to the code. One of these was because the halted clock provided waiting for new data functionality, the waiting state was no longer necessary. The clock gating was moved onto the loading state and the waiting state was just used for initialisation to improve the reliability of the system. Another change was the option to clock both the encryption/decryption as well as the key expansion and have them both in the same module. This also removed the need for two variants needed for in loop and pre key expansion as the clock gating could also be dependant on the *enc_dec* signal. It also helped with the logic minimisation as the key and the expanded key didn't need to be passed between multiple modules.

Another addition to the system was the ability to load a new key after the initial loading was done. This was done with another input signal *newKey* which operated in the same way as the *newData* signal did for data. Also a few more signals were outputted that indicate when the data and keys were loaded into the system and thus the inputs could change to be ready for the next set of data.

For this new approach all ten modes of SIMON were tested in hardware with each having it's own top level module containing an instance of the control module as well as the key expansion module which had a new input of the z constant that varies with each mode. This top level module for the basic mode 32/64 is shown in Listing 3.9 and the other modes are setup in the same way.

```
1  module SIMON_3264
2  #(
3  parameter N = 16,
4  parameter M = 4,
5  parameter T = 32,
6  parameter Co = 5
7  )
8  (
9  input logic clk, nR,
10 input logic newData, newKey,
11 input logic enc_dec, readData,
12 input logic [2*N-1:0] plain,
13 input logic [M-1:0][N-1:0] key,
14 output logic ldData, ldKey,
15 output logic doneData, doneKey,
16 output logic [2*N-1:0] cipher
17 );
18
19 //  KEY EXPANSION LOGIC
20 logic [M-1:0][N-1:0] pKeys;
21 logic [N-1:0] oKey;
22 logic [Co-1:0] count;
23
24 reg [61:0] z = 62'
       b01100111000011010100100010111110110011100001101010010001011111;
25
26 SIMON_keyexpansion #(N,M,Co) ke(.count(count), .keys(pKeys), .z(z), .
       out(oKey));
27
28 SIMON_control #(N,M,T,Co) control(.*);
29
30 endmodule
31
```

Listing 3.9: The next state logic for the (PRE) control module

To test this new approach the same testbench was used for each version, only differing in the input data taken from the test vectors. The same clock This testbench first inputs the plaintext and the key at the same time by putting the *newData*, *newKey* and *enc_dec* signals $HIGH$. The new input signals are then reset after each rising edge of the load signal, *ldData* & *ldKey* respectively, using *always* blocks in the testbench and short delays. Then in another *always* block, which is sensitive to the rising edge of the *doneData* signal, the *readData* signal set $HIGH$, the output is read back into the input and he *enc_dec* signal is inverted. Then before the *always*

block completes the *newData* signal is asserted again. The *readData* is then reset after a falling edge of the *doneData* signal which the control module should enforce. With these *always* blocks, Listing 3.10. the simulation can just continue always alternating between encrypting and decrypting the data.

```
always @(posedge ldData)
begin
    repeat(2)   @(posedge clk);
    #20ns
    newData <= 1'b0;
end

always @(posedge doneData)
begin
    repeat(2)   @(posedge clk);
    #10ns
    readData <= 1'b1;
    plain <= cipher;
    enc_dec <= ~enc_dec;

    repeat(2)   @(posedge clk);
    #10ns
    newData <= 1'b1;
end

```

Listing 3.10: The *always* blocks of the testnches

Mode	N	M	T	Logic	Registers	Pins
0	16	4	32	437	656	138
1	24	3	36	717	1049	178
2	24	4	36	730	1073	202
3	32	3	42	1123	1585	234
4	32	4	44	1123	1681	266
5	48	2	52	1905	2801	298
6	48	3	54	2024	2945	346
7	64	2	68	3266	4754	394
8	64	3	69	3627	4882	458
9	64	4	72	3498	5138	522

Table 3.1: A table of the synthesis of all ten modes of SIMON.

With this new approach the hardware as compiled and synthesised in Quartus Prime 17.1 Lite Edition for the DE1-SoC development board[insert reference] with just the basic settings and 'Analysis and Synthesis' stage used. This produced some statistics about the design including the number or registers and combination logic lookup tables required to implement it. As expected these varied greatly with the mode of SIMON with the registers needed increasing substantially because the key schedule is stored and the not only does the word width increase but also the number rounds. Table 3.1 shows the results of this synthesis and **??** is the resultant RTL netlist view of the system.

Chapter 4

Final Design Approach

For the final approach the idea was to ensure that the control module remains in the *execute*, or processing, state as much as possible to ensure that the system is always processing data and thus increase the throughput. Having the hardware active all the time does increase the overall power consumption and might negate the effects of the clock gating added in section 3.2, but as the throughput is increased the power consumed per bit or byte of data processed is lower. This is done by sending packets of data instead of just individual blocks of data and having extra functions or modules to handle the input and output of these packets. This added another system to be tested and compared in both hardware and software but even if it performs worst than the just inputting blocks, it does represent a more usable and complete system.

In a full system the packets would be sent using a serial protocol like I^2C but the they would also include information about what and how much data is being sent in a packet. As there are ten modes of the SIMON algorithm, labelled as $0-9$, four bits of the information are used to identify what mode of SIMON the data is intended for. This leaves another 4 bits to be used in a byte that were used individually for different parameters of the packet. One bit was used to indicate whether the data is to be processed or a new key to be loaded. Another tells the hardware to encrypt or decrypt the data, obviously irrelevant if a new key is being sent. As for most modes the number of words in the key are more than the two words in a block it was decided that a constant four words, or two blocks, would be sent in each packet but an option to just send one block would be controlled by one bit. The last bit in this byte tells the hardware that the packet is either meant to be inputted to it or outputted from it. To help the hardware keep track of the packets another byte was used to as a packet ID, which increments with each new packet.

ADD PACKET DIAGRAM FIGURE

Having this information attached to each block of data being processed could introduce a weakness to the cipher that might be exploited by an attacker. Although, ensuring and increasing the security of the algorithm is not the main purpose of this project.

4.1 Software - Hosted C++

4.1.1 Changes to the Code

As the packets need to be setup by the a processor and includes a few data types and with some individual bits needing to be controlled it made sense to switch to an object orientated language instead of C so naturally C++ was chosen. While it would be possible to develop the functionality required in C it would take time and effort to get functionality already provided by C++.

```
1  class           WORD
2  {
3  public:
4      //  CONSTRUCTORS
5      WORD();
6      WORD(TYPE(N) x);
7      //  MUTATORS
8      void            assign          (   TYPE(N) x
           );
9      void            assign          (   TYPE(8) x, TYPE(8)  i   );
10     TYPE(8)     addBYTE         (   TYPE(8) x
       );
11     void            flush               (
                       );
12     //  ACCESSORS
13     void            test                (
                       );
14     TYPE(N)     get_v               (
                   );
15     TYPE(8)     get_b               (
                   );
16     TYPE(8)     get_b               (   TYPE(8) i
           );
17     TYPE(16)    get_B               (   TYPE(8) i
           );
18     TYPE(16)    size_v          (
                  );
19     TYPE(16)    size_b          (
                  );
20     string      HEX_WORD        (
                  );
21     string      CHR_BYTES       (
                  );
22     string      HEX_BYTES       (
                  );
23     //  OPERATORS
24     WORD            operator~       (
                      );
25     WORD            operator&       (   const   WORD    x
           );
26     WORD            operator^       (   const   WORD    x
           );
27     WORD            operator^=  (   const   WORD    x
       );
28     WORD            operator<<  (   const   TYPE(8) i               );
29     WORD            operator>>  (   const   TYPE(8) i               );
30 private:
31     TYPE(8)     w_nxtBYTE;  //  NEXT BYTE TO WRITE TO
32     TYPE(8)     r_nxtBYTE;  //  NEXT BYTE TO READ FROM
33     union
34     {
35         TYPE(N) val :   N;      //  VALUE STORED AS ONE BYTE
36         TYPE(8) bytes[N/8]; //  VALUE STORED IN BYTES
37     };
38 };
39
```

Listing 4.1: Declaration of WORD Class

Listing 4.1 shows the basic word class with the same $TYPE(N)$ as in the previous C software. With C++ some of the operators needed could be overloaded, including the shift operators (<< & >>) which were used to provide the rotation functions. The other basic data type used in the previous approaches were setup with similar classes and constructor, mutator and accessor functions. All of the classes developed also used similar functions, especially for printing the data to strings and getting the data stored.

The packet class stored the packet information, Listing 4.2, packet count and the four words of data, Listing 4.3. It also has an array of one byte unsigned integers that compacts the data into the smallest possible size for transmission. This is required because the information and count can always be stored in two bytes but depending on the mode the words vary in size and don't always fit exactly into the data types available. So before being sent a function is used to copy the data from the words byte by byte into the byte array which results in no redundant data being sent by accident. The setting up of these packets can be done separately or as part of the encryption, or decryption, process. The input and output packet counts were stored in static variable as they are meant to increment with each new packet. This was done when a packet is declared as an input or an output and not in the constructor as the actual object could be used again. For example, an output packet with some cipher text stored could be used to test decryption by simply declaring it an input packet while keeping the same data.

```
typedef struct  _INFO_
{
    //  PACKET INFO
    TYPE(8)      mode        :   4;  //  CIPHER MODE
    TYPE(8)      in_out      :   1;  //  INPUT/OUTPUT
    TYPE(8)      data_key    :   1;  //  DATA/KEY
    TYPE(8)      enc_dec     :   1;  //  ENCRYPT/DECRYPT
    TYPE(8)      nBlocks     :   1;  //  NUMBER OF BLOCKS (1/2)
}                _INFO_;

```

Listing 4.2: Declaration of Packet information

```
static U_64 inputCount;
static U_64 outputCount;

union
{
    TYPE(8) iBYTE;
    _INFO_  iDATA;
};
TYPE(8)      count;             //  DATA COUNT
WORD         wDATA[4];          //  PACKET DATA

//  PACKET IN BYTES
TYPE(8)      pBYTES[2+(N/2)];

```

Listing 4.3: Data stored in Packet class

A class was used to reading files because the data were read into a temporary byte

array before being stored in a the four words in a packet. From there with the packet setup, using Listing 4.4 where the *readWORD*() function reads a *WORD* from the file and also keeps track of where in the file it has read to. With the packet read it could be added to a vector of input vectors or sent straight to the cipher used, either software or hardware. If the vector method is used then the packets could be stored in another file to be sent to the cipher at a later point. The class is also used for reading the packets returned to it by the cipher and then written to a file with the same as the original name but with an encrypted or decrypted indicator. This class is also capable of just reading blocks of data from the file so that the new approach to the SIMON could be compared with the previous approaches.

```
1  PACKET        DATA::readPACKET      (                                    )
2  {
3      PACKET bufferPACKET;
4
5      _INFO_ input;
6      input.mode = MODE;
7      input.in_out = 0;
8      input.data_key = 0; //  DEFAULT
9      input.enc_dec = 1;  //  DEFAULT
10     input.nBlocks = 1;
11
12     bufferPACKET.input();          bufferPACKET.assign(input);
13
14     bufferPACKET.addWORD(readWORD());
15     bufferPACKET.addWORD(readWORD());
16     bufferPACKET.addWORD(readWORD());
17     bufferPACKET.addWORD(readWORD());
18
19     bufferPACKET.pack();
20
21     return bufferPACKET;
22 }
23
```

Listing 4.4: Function used for reading a packet

```
class           CIPHER
{
public:
    //  CONSTRUCTORS
    CIPHER();

    //  MUTATORS
    PACKET      compute     (   PACKET  p                       );
    BLOCK           compute     (   BLOCK   b, TYPE(8)  enc_dec );
    void            expandKEY   (   KEY     x                   );
    void            encryptDATA (   WORD    x0, WORD    x1      );
    void            encryptDATA (   BLOCK   x                   );
    void            decryptDATA (   WORD    x0, WORD    x1      );
    void            decryptDATA (   BLOCK   x                   );
    void            resetCount  (                               );
    BLOCK           round       (   WORD    x                   );
    WORD            expand      (   KEY     x,  TYPE(8) i       );

    void            flush       (                               );

private:
    TYPE(8)     pktCOUNT;
    BLOCK           stateCIPHER;
    KEY_S           scheduleKEY;
    TYPE(8)     roundCOUNT;
    TYPE(8)     doneKEY :   1;

};
```

Listing 4.5: The declaration of the CIPHER Class

The reason for using a class for the cipher object, Listing 4.5, was so that the key schedule and the state of the encrypted/decrypted block could be stored in one variable it similar to how it is stored in the hardware version. There is also a function added for handling the packets to ensure that data is used correctly in Listing 4.6. This function takes the inputted packet and sets up a new output packet with the same information about the data and then checks it for errors before computing the relevant data and addding it to the output packet. As with data class there is also the basic encryption and decryption functions to deal with just blocks as the inputs.

```
PACKET CIPHER::compute ( PACKET x )
{
    PACKET out;
    out.output(x.get_i());

    switch(x.checkIN(pktCOUNT))
    {
        case 0: // ERROR
            break;
        case 1: // DATA
            if(doneKEY)
            {
                if(x.get_i().enc_dec) encryptDATA(x.get_w(0), x.get_w
    (1));
                else decryptDATA(x.get_w(0), x.get_w
    (1));
                out.assign(stateCIPHER.get_w(0), 0);
                out.assign(stateCIPHER.get_w(1), 1);
                pktCOUNT++;
                if(!x.get_i().nBlocks) break;

                if(x.get_i().enc_dec) encryptDATA(x.get_w(2), x.get_w
    (3));
                else decryptDATA(x.get_w(2), x.get_w
    (3));
                out.assign(stateCIPHER.get_w(0), 2);
                out.assign(stateCIPHER.get_w(1), 3);
            }
            break;
        case 2: // KEY
            expandKEY( KEY(x.get_w(0), x.get_w(1), x.get_w(2), x.get_w
    (3)) );
            pktCOUNT++;
            break;
        default : // ERROR
            break;
    }

    out.pack();

    return out;
}
```

Listing 4.6: The function for handling the packets

4.1.2 Testing

ADD C++ TESTING BELOW [200]

As this final approach is in a different language all of the classes and overloaded operators needed to be tested, and for this reason each class has a test function that checks the number bytes used by the variables and then sets up example data and prints them to the console in the various formats. Then before packets and data reading were tested the cipher functions were designed and first tested just on the blocks as with the previous approaches. Then after the packets were functioning

as desired the decoding of them in the cipher class could be tested by inputting dummy packets and checking that the correct case is activated. To check that the data reading is reading the correct data and inserting it into the correct packets, and blocks, a small test file with just two packets worth of data, 32 bytes for the 32/64 mode, was used tha included the testvector plaintext.

From there there functions were used to insert the test vector key into the first packet and then the plaintext into the second packet. These two packets were used to confirm the operation with checks on the outputted cipher text after the processing. They were also inserted at the beginning of each packet stream so that if something went wrong with whole processing, it can be seen if the cipher functions are the source of the error.

4.2 Hardware - System Verilog

With the packets being setup in software, as described in section 4.1, the System Verilog module only needed to be handle the information and hence route the packeted data to the correct part of the system at the right time. The whole system also needed to ability to pass the packet information and count through the modules as each one processed the data to ensure that the correct packet is outputted. To do this two modules were added: one to handle the input packet; and one to handle the output packets.

To handle the input packets the module takes an input of bytes equal to the size of the packets as well as an input control signal $newIN$. The module operates in a similar way to how the control module did in the first approach, subsection 3.1.2, with states for waiting; loading packets; processing packets; and writing the relevant words to the relevant output, the key or the input block of the control module. After the packet is loaded the processing state checks the information byte is in the correct form: as an input, for the correct mode and the packet count; and then decodes what data is in the packet which is used by the writing state. When a packet contains two blocks of data, identified in the information, the first writes two of the words to the the input block and then waits for the control module to be ready for new data before writing the other two words. To do this a $PROCESSING$ flag bit used to notify the next state that when the control module is ready to load new data it should continue to write the other block in the packet before it is ready to load a new packet.

```
always_comb
begin
    unique case(current)
    WAIT:
    begin
        if(PROCESSING)
        begin
            if(newDATA_rise || loadDATA)     next = WAIT;
            else                             next = WRITE;
        end
        else
        begin
            if(newPKT_rise)                  next = LOAD;
            else                             next = WAIT;
        end
    end
    LOAD:                                    next = COMPUTE;
    COMPUTE:                                 next = WRITE;
    WRITE:
    begin
        if(newDATA_rise)                     next = WRITE;
        else                                 next = WAIT;
    end
    endcase
end

```

Listing 4.7: The next state logic used for the input packet handling module.

The control module was also changed slightly to enable the packet information and count to propagate through the module with the connected data. This removed the need for the *enc_dec* signal as that is contained in the information. The input or output flag is also flipped by the control module for the output handling module. One change that was made though was to use extra signals that keep track of when the new data and keys are loaded internally with rising edge sensitive $always\_ff$ blocks. This is done with the new_DATA_rise and $newKEY_rise$ signals that are set $HIGH$ with the rising edge of their relevant inputs and reset with the rising edge of the relevant load output. This enables the control module to not continue loading new data or another key if the new input signals are not reset by the other modules. Similar functionality was also used used in the packet handling modules for their new input signals.

```
//  RISING EDGE NEW CHECK
always @(posedge newDATA, posedge loadDATA, negedge nR)
begin
    if(~nR)                          newDATA_rise <= 1'b0;
    else if(loadDATA)    newDATA_rise <= 1'b0;
    else if(newDATA)     newDATA_rise <= 1'b1;
end

```

Listing 4.8: The $always\_ff$ blocks used for risnig edge new data checks

The output handling module receives the packet information and data from the control module and then writes the data to the relevant part of the output packet. It then checks and decodes the packet information similar to the input dandling

module but then instead of loading, processing and then writing it processes the packet information, reads the data and then writes the packet to the output. If the packet information suggests that there should be two blocks of data in the packet then the module waits for a second block of data to be processed by the control module before writing the output packet.

ADD SYSTEM VERILOG LISTINGS BELOW [100]

ADD SYSTEM VERILOG SIMULATIONS BELOW [200]

As the control module didn't change much from the previous approach its testing wasn't important but the other modules needed to be functionality correctly with absolutely no glitches for them to all interact as a whole system. To this the, as with all of the previous testbenches, the inputs were all setup with the reset signal active and then changed after the reset was deactivated.

For the input handling module the input packet had the basic packet information for a key, initial packet count of 0 and then the key from the test vectors.

ADD SYSTEM VERILOG SYNTHESIS BELOW [100]

Mode	N	M	T	Logic	Registers	Pins
0	16	4	32	635	1029	167
1	24	3	36	971	1542	231
2	24	4	36	997	1590	231
3	32	3	42	1446	2214	295
4	32	4	44	1457	2342	295
5	48	2	52	2337	3654	423
6	48	3	54	2501	3646	423
7	64	2	68	3813	5863	551
8	64	3	69	4115	6055	551
9	64	4	72	4117	6375	551

Table 4.1: A table of the synthesis of all ten modes of SIMON with packet handling modules.

Chapter 5

Experiments & Results

The experiments used in this project mainly focussed on the parameters that were outlined in chapter 1: throughput, power consumption, and resource use; with the accuracy confirmed throughout the development stage and the security level assumed to be good enough. Unfortunately, the hardware was never fully implemented on an FPGA as it was felt that the work required to setup the modules to work with a communication protocol was too much and would not yield any more information that couldn't be discovered with the various simulation tools. As the software and hardware versions do differ in how they compute the results, some of the desired comparisons couldn't be made directly as, for example, the power consumption is not available in software and the resources used are not the same. Even with the throughput the clock speed used in each version is vastly different, but even with that disadvantage the hardware should perform better.

5.1 Throughput

5.1.1 Method

To test the throughput of the algorithm is as simple as putting some data through it and timing how the implementation takes to process that data. From there, using the number of bytes and the time taken, the throughput can be calculated in bits per second ($bits/s$) with [insert equation]. For the both software and hardware the data needed to be setup in software by reading an example file containing approximately $10kB$ of lorum ipsum[insert reference]. As the reading of the file takes some time, and is not part of the hardware testing, it is not included for the throughput testing in software by reading the file and setting up the packets and blocks before recording the start time. To input the data into the ModelSim testbenches it was simply written to a file in a SystemVerilog format and then copied and pasted into the relevant testbench.

5.1.2 Results

As each mode of SIMON processes a different number of bytes depending on the block the throughput of each block size (32, 48, 64, 96 and 128) is shown as it doesn't vary much key size for both software and not at all for hardware. Figure 5.1 demonstrates how the throughput of SIMON in hardware out performs the software with all of the block sizes defined, and used, by SIMON. It also shows that the throughput increases with the block size significantly in hardware. There is a slight improvement in software between 32 and 48 bits but from there it remains almost constant. This is because the processor used to run the software is 64 bits and thus can operate on 64 bit, used for the 96 and 128 bit block sizes, and 32 bit, used for 48 and 64 bit block sizes, data types efficiently, but with the 16 bit data type used only with a 32 bit block size this is not the case.

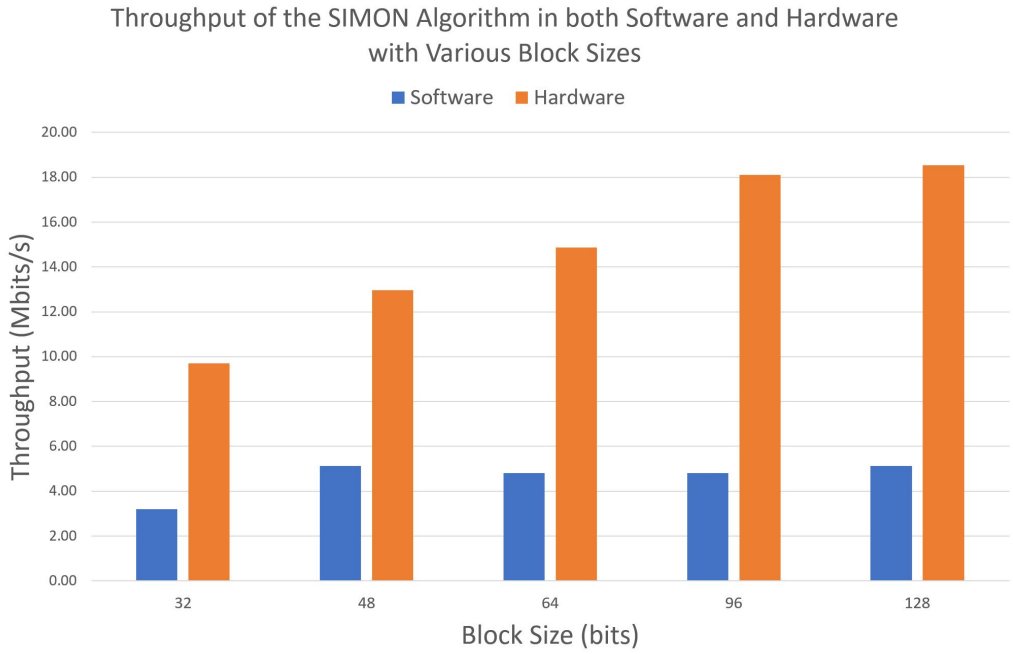

Figure 5.1: The comparison of the throughputs of various modes of SIMON in software and hardware.

Another factor that effects the throughput of the algorithm is the clock speed used, but this suggests that the software should process data quicker than the hardware because it uses a clock speed of over $2GHz$ and the hardware uses a much slower $10MHz$ in the simulations used to gather these results. Therefore, a speedup of approximately 200 is expected when comparing software to hardware, but the actual speedup occurs in the other direction with an average of about 3.3. This is because the software takes many more instructions to compute each cycle of the encryption, or decryption and it also has the overheads of the operating system and any other programs running at that time. Where as because the hardware is customised to this algorithm it can compute the result encryption, or decryption, cycle almost very clock cycle, excluding when the state machine is in the other states.

5.2 Resource Use

5.2.1 Method

Different resource are required and used by the software and hardware versions of SIMON developed for this project, but they are both still quantifiable in terms of memory used, but not with the logic elements used by the hardware as the software uses a number of operations with the ALU to compute the logic. Therefore, the different modes are used for comparisons within the same version with total logic elements, total registers, and input output pins being used for the hardware; and the total memory and number of instructions required for the software. The hardware data is provided by the Quartus Prime 17.1 and the data for both approach two, **??**, the final approach in **??** as they are both synthesised and provide an insight into the difference that using the packets has in terms of resource use.

5.2.2 Results

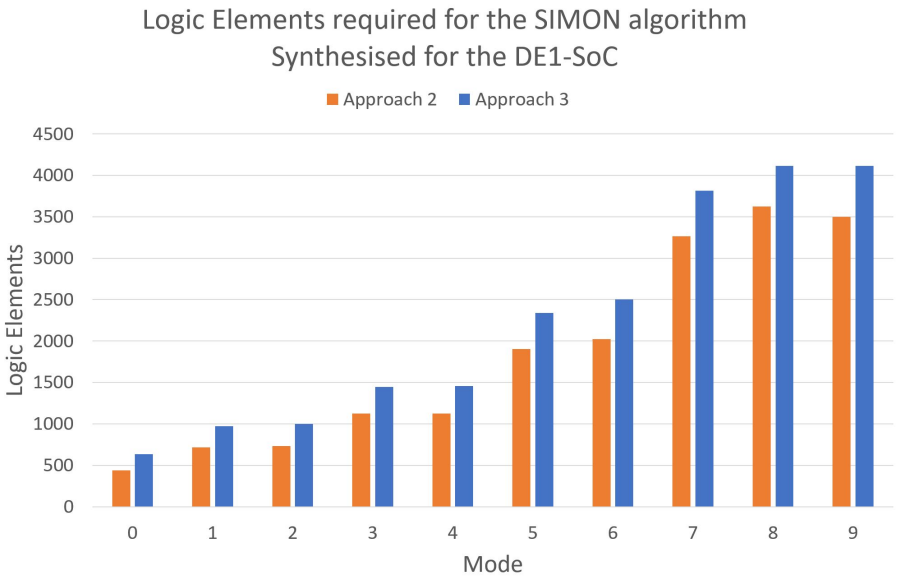

Figure 5.2: The comparison of the throughputs of various modes of SIMON in software and hardware.

Using the data from Table 3.1 and Table 4.1 the approaches resources are compared in Figure 5.2, 5.3 and 5.4. These figures indicate that, for both approaches, the main increases only occur between modes when the block size changes and the resources used only changed slightly with the different key sizes. An exemption to this pattern is the pins used in approach 2 which appears to increase with an approximate linear relationship with the mode. This is different for the third and final approach as the packet handling ensures that the inputs and outputs remain equal for the block sizes.

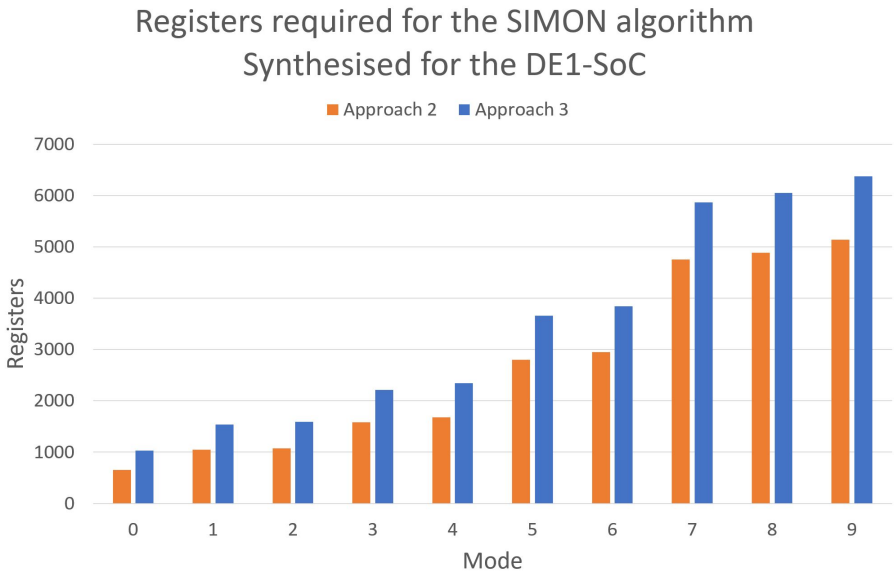

Figure 5.3: The comparison of the throughputs of various modes of SIMON in software and hardware.

When comparing the approaches it is clear that the packet handling modules, as expected, require more resources than just the control module used in the second approach. Although, the percentage changes between the two sets of data, while quite significant, aren't all huge; with the logic elements averaging at 27%, the registers at 36% and the input/output pins at 23%.

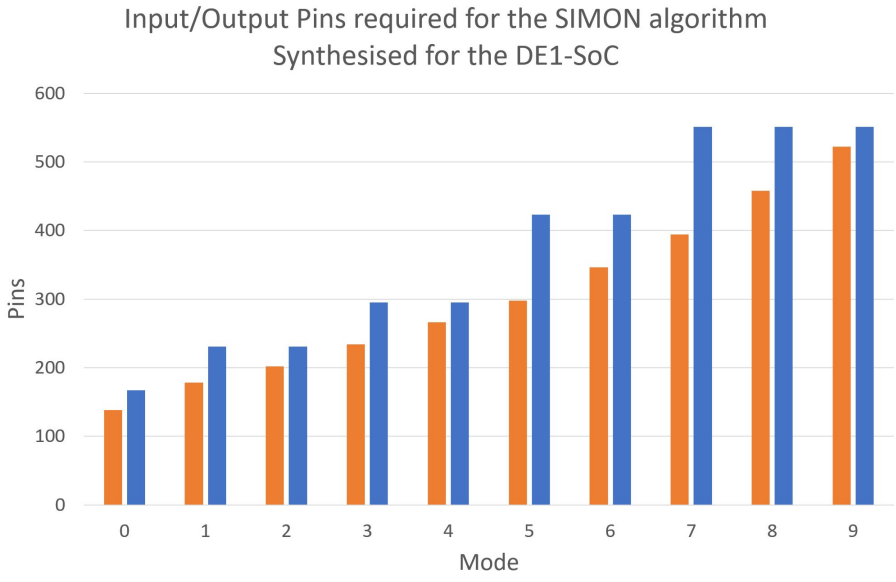

Figure 5.4: The comparison of the throughputs of various modes of SIMON in software and hardware.

5.3 Power Consumption

5.3.1 Method

As the hardware design was never implemented onto a FPGA the power consumption of the system while encrypting and decrypting data was never tested. However, the synthesis software often provide some extra tools that can estimate the power consumption by calculating the switching frequency of the internal nodes based on random input switching. The synthesised design from Quartus Prime is the main source of this data by using it's 'Power Analyzer Tool', but the design was also compiled by the Cadence software[[insert reference] to compare what might be possible if the system is implemented onto an ASIC in the future. The cadence software only provides some basic information about the estimated the power consumption during the synthesis.

5.3.2 Results

ADD POWER TESTING RESULTS BELOW [300]

Chapter 6

Conclusion & Future Work

6.1 Conclusion

Overall this project, while not completely succeeding in the original goal of optimising the algorithm for the low power environment of 'Internet of Things', it has produced a working model for encrypting and decrypting data at a much faster rate than would be possible in the small embedded processors that the 'IoT' devices use. Finding multiple algorithms for the varying levels of performance; in terms of security, throughput, power consumption and resource use; required by the wide range of 'IoT' devices was also a goal in the original project brief but wasn't achieved in this project. However, the algorithm, SIMON, was chosen to be the only algorithm used because it is defined with flexibility in all those performance parameters with the different block and key sizes.

Another potential problem of this project is that it became too software orientated, for reasons explored in section 6.2, but the work done was still very relevant to the project goals as for it to be used in 'IoT' products their would need to be a communication protocol in place. Although, the protocol eventually developed was done so with not much research into similar options that have already been produced and are readily available for use. This resulted in the protocol, based on packets and connected information, possibly being a poor solution to the problem.

6.2 Project Management

Unfortunately, even though the multiple working models produced in this project required working very hard throughout the duration of the project, a consistent project management technique made it difficult for the original goals, which were plausible and possible, being achieved. As, for example, implementing the hardware version onto a FPGA and testing the performance parameters on actual hardware was the main objective at the beginning but when it came to to doing this developed model wasn't ready because too much time and effort was put into adding advanced power saving techniques such as clock gating, something that wasn't required for correct operation, to the system. This meant that the simulation and synthesis tools

were relied upon heavily to gather the results required to evauluate its performance; and these only produce estimates and don't necessarily represent what a FPGA would perform like.

On the other hand, when this was realised a backup plan for continuing the project was reasily available. This plan was to shift some of the development efforts to the software to setup the packet system for sending and receiving data between the various 'IoT' platforms and the now theoretical hardware. Even though this was the back up plan it did require more effort than expected

6.3 Future Work

There is lots of functionality that can be added and testing that can be done to continue and improve on this project in the future. These includes, but not limited to, continuing to improve the efficiency of both the hardware and software; implemented the hardware version onto an FPGA; implementing the hardware as an Application Specific Integrated Circuit (ASIC); optimising the design for clock speed on the FPGA and ASIC; and implementing the software onto the small embedded processors used in IoT.

For the hardware version the improvements needed are mainly just adding more power saving methods like clock gating and also optimising the logic for the fastest clock speed possible. What is needed though is methods to deal with the errors detected by the packet handling modules. As, at this point the errors are detected and stop the loading of packets but are only shown as lines of text printed during in simulation. This could be achieved by having a few outputs assigned to the various errors that occur that when detected by the sending processor could be corrected and resent.

In terms of software, the code should probably be reduced to just the basic functionality and the setup of the packets and the encryption algorithm could be used in separate sets of code.

Mainly that would be putting the hardware design onto an FPGA and performing some tests for throughput and power consumption on the actual hardware. This would involve adding some more modules on top of the SIMON cipher system to enable communications to get the data in and out, which could be a fully parallel method if enough input/output pins are available for both the key and data block. However, this would prove very difficult due the limited I/O options available on most FPGA development boards, and it requiring a total of 96 inputs (64 Key + 32 Data Block) and 32 outputs (32 Data Block) for just the most basic mode without the packet system in place. Another option would be to send a $WORD$, or maybe a $BLOCK$, of data in parallel and then some control signals to route it into the correct register for the key or data block. But the method I would have used, if I had the time in this project, would have been to implement a serial protocol like I^2C or similar in the hardware. This is because this functionality would most likely be already implemented on the host embedded processor and requires only a few I/O pins. The serial protocol would send over the packets which would be stored in a FIFO with the width of the packets.

The developed System Verilog design could also be compiled and implemented on an ASIC. This would almost certainly improve the power consumption as only the absolutely necessary logic would be present and consuming power. It would also improve the throughput performance as it could be designed for an optimal clock speed that should be faster than what an FPGA can achieve.

For a more direct comparison with the small embedded processors used in IoT the software could be altered slightly and compiled for the platforms use. With this the throughput and power consumption could be tested. However, as the hardware has already been proven to perform better than the software running on a full size 64-bit processor at over $2GHz$, a small processor with between 8 and 32-bit operations at a much slower clock speed will perform even worse.

Bibliography

[1] Z. Hegde, "IoT now : how to run an IoT enabled business." [Online]. Available: https://www.iot-now.com/2017/03/09/59388-iot-applications-autonomous-vehicles-smarter-cars-cellular-iot-vehicle-telematics/

[2] D. Evans, "The Internet of Things How the Next Evolution of the Internet Is Changing Everything," 2011. [Online]. Available: https://www.cisco.com/c/dam/en{_}us/about/ac79/docs/innov/IoT{_}IBSG{_}0411FINAL.pdf