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

Author: Lewis Smith

December 12, 2017

Low Power Hardware Accelerated Internet of Things Cryptography

Project Supervisor: Mark Zwolinski
Second Examiner: Frederic Gardes

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

Abstract

This project aims to develop a FPGA system that can encrypt and decrypt data that is being used and transmitted between 'Internet of Things' devices. It does this using the SIMON algorithm that was selected from many during the research stage because of its flexibility and its lightweight characteristics. The FPGA development was done in the System Verilog language and simulated using ModelSim before being synthesised in Quartus Prime. A software version was also developed in C for comparison with the hardware version. Both versions will be compared with similar projects found in literature in terms of data throughput, power consumption and circuit area. So far the software version is ready for testing and the hardware is working in simulation but not tested on a FPGA.

Contents

Chapter 1

Introduction

Over the last few years there has been a shift in type of devices connected to the internet from just servers, PC's and later smartphones, to small embedded processors that can control many devices. 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. 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[1]. This also means that the complexity of the devices varies greatly with simple light switches and even kettles being given the IoT treatment, but at the other end of the scale a network of connected self-driving cars is being considered[2].

To keep the potential adversaries from accessing the data, transmitted over an open internet channel, and possibly controlling numerous connected devices, maliciously or not, an encryption algorithm can be used. There are many encryption algorithms that perform this function and most can be implemented in both software and dedicated hardware such as a 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, 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.

Chapter 2

Background Research and Literature

2.1 Internet of Things

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. Due to IoT devices having limited resources [3] suggests that 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 [4] outlines a limit of tens of micro Watts (μW) for RFID tags.

2.2 Cryptography

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. 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[5]. 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.

2.2.1 Asymmetric Key

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[6].

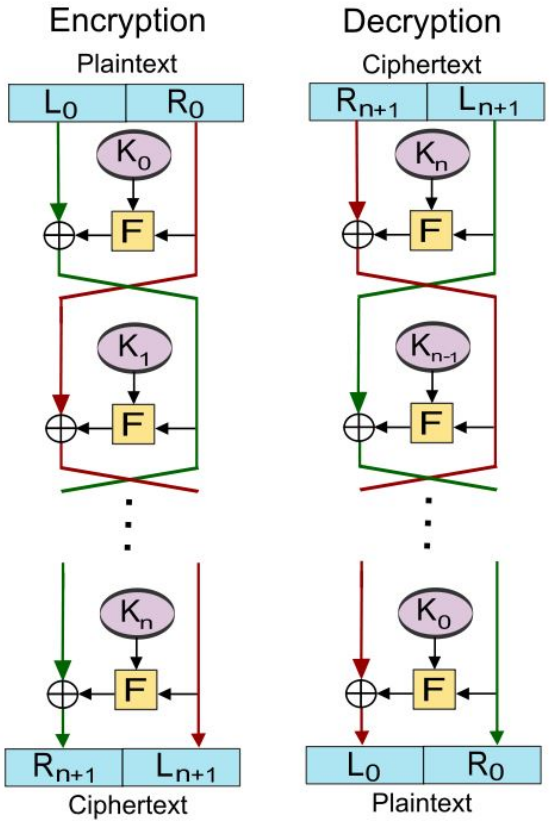
Public key cryptography can be used in two different modes as if data is encrypted with the intended recipients public key only they can decrypt it with their private key, thus encryption. However, if a private key is used for encryption then using the public key to decrypt it ensures the senders identity, authentication[5].

2.2.2 Symmetric Key

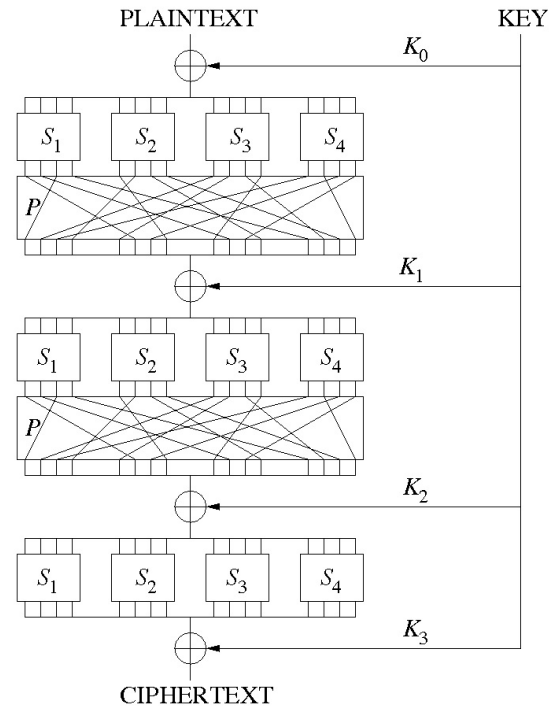
Similar to the symmetric/private comparison symmetric key cryptography is also known as private key, as in order to keep the encrypted data secure the key used must be kept secret. There are two main types of private key algorithms that operate on the plaintext differently: block ciphers which uses a fixed number of bits, block; or stream ciphers which encrypts data bit by bit[5].

Modern block ciphers work by iterating a basic cipher function with the ciphertext used as the input for the next round[7]. To increase the security of a block cipher subkeys, or round keys, are generated for each round of the iteration using similar operations to those used in the actual cipher. The round functions are mostly designed using either a Feistel network[8] (F network) or a Substitution Permutation network (SP network)[5].

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. It works by splitting the input plaintext into equal words and applying the round function to it right, or LSB, word before swapping the words for the next iteration. This functionality, for both encryption and decryption, is described in Figure 2.1a.



(a) Basic Feistel network.



(b) Basic Substitution Permutation network.

On the other hand, Substitution Permutation networks operate on the whole plaintext block using S-boxes for substitution and P-boxes for permutation. There can be more steps that treat the block slightly differently but those are the basic steps and when combined they are enough to provide Shannon's confusion and diffusion[7]. The basic structure of an SP network is in Figure 2.1b.

Stream ciphers work by generating a pseudo-random keystream to combine with the plaintext[9]. 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 modes of operation for block ciphers, in [10], 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

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

There are many block ciphers that are considered very secure and therefore popular, they include: DES[11], AES[12], Blowfish[13]. 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. 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[11].

DES was used for about two decades but in the 1990s several successful attacks proved its weakness[14] so in 1997 NIST started a selection process to find its replacement. 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[12]. Unlike DES, AES uses a SP network as it is efficient, in time, in both hardware and software.

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[15] or efficient S-box implementations[16]. However, even area optimized designs like [17] 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. These algorithms might be used because there is still some distrust of the NSAs involvement in the algorithms and they are more open source. 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[13].

As with AES there are many FPGA and ASIC implementations of the Blowfish algorithm, [18] and [19], 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[20], PRINCE[21] and the SIMON and SPECK algorithms[22]. 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 [23].

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 mn bits.

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 [24].

Even though they are relatively new algorithms there are still a few FPGA implementations available for review, including [25] and [26] as well as those provided in [22]. 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, Figure 2.1a, 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 [27] 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. The encryption and decryption functions take the Feistel network form described in Figure 2.1a with the round function Equation 2.1.

$$F(x) = (R^1[x] \& R^8[x]) \oplus R^2[x] \tag{2.1}$$

SPECK on the other hand, being optimised for software implementations uses XOR ($\oplus$), modulo 2^n addition ($+$) and circular rotate operations ($R^j[x]$). The encryption and decryption functions take a slightly different form to the basic Feistel network, Figure 2.1a, and are shown in Equation 2.2 and 2.3 where $\alpha = 7$ and $\beta = 2$ if $n = 16$, but $\alpha = 8$ and $\beta = 3$ otherwise.

$$L_{i+1} = (R^{-\alpha}[L_i] + R_i) \oplus K_i \tag{2.2}$$

$$R_{i+1} = R^{\beta}[R_i] \oplus (R^{-\alpha}[L_i] + R_i) \oplus K_i = R^{\beta}[R_i] \oplus L_{i+1} \tag{2.3}$$

As the aim of this project is to compare how an algorithm performs in hardware and software SIMON was chosen even though SPECK shows almost as good performance in hardware and much better in software.

Chapter 3

Progress

Due to the fact that there isn't a standard library or program for SIMON, a software version was required for benchmarking as well as the main System Verilog version. Starting in software also increased by familiarity of the algorithm and provide a good starting point for System Verilog development. The code for the Hosted C version is available in the Appendix.

For all versions of the SIMON algorithm (table 2.1) efficiency, in terms of power consumption and resource use, is very important but time efficiency is not an initial priority. All versions of this algorithm were developed not only with the description but also the pseudocode provided in [22]. That document also has a set of test vectors for all modes that define the ciphertext that the algorithm should produce from the given key and plaintext. Due to the Feistel network structure of this algorithm encryption can be done in parallel to the key expansion but for decryption requires the keys to be pre expanded. For this reason, and because some modes of operation only require encryption, two variants of the algorithm were developed: one for just encryption and one with full functionality.

3.1 Hosted C

As there are ten versions of this algorithm that all differ slightly with the block key sizes they could all be developed individually. However, as ten versions of the similar functions would be required I felt it was better to have one version that is flexible. However, checking which mode the program is before most operations would be very inefficient and as the program would rarely changing mode during runtime the decisions could be made at compile time using preprocessor macros as in Listing 3.1. The macros were used to define the variable type used for the words based on the values for n with the uint$n$_t type. Also the value of m was used with macro if statements in the key expansion functions where extra code is required if $m = 4$.

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

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 word type

The round function and Feistel functions are shown in Listing 3.2 and they are iterated in the encryption and decryption functions. These functions were tested with the test vectors from [24] producing the results in

```
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];
  b[0].v = b[1].v ^ F(b[0]) ^ k.v;
  b[1] = tmp;
}
```

Listing 3.2: Round and Feistel functions

```
|       TESTING PRE KEY EXPANSION              |
|       KEY:            1918111009080100       |
|       PLAIN->CIPHER:  65656877->00000000     |       START   |
|       PLAIN->CIPHER:  65656877->C69BE9BB     |       ENCRYPT |
|       PLAIN->CIPHER:  65656877->C69BE9BB     |       TEST    |
|       PLAIN<-CIPHER:  65656877<-C69BE9BB     |       DECRYPT |
|       TESTING IN LOOP KEY EXPANSION          |
|       KEY:            1918111009080100       |
|       PLAIN->CIPHER:  65656877->00000000     |       START   |
|       PLAIN->CIPHER:  65656877->C69BE9BB     |       ENCRYPT |
|       PLAIN->CIPHER:  65656877->C69BE9BB     |       TEST    |
```

Figure 3.1: Results from SIMON32/64 Test Vectors encrypted and decrypted in Hosted C program.

3.2 System Verilog

After the Hosted C version was working with the test vectors the System Verilog development began. Similar to the function used in the software developed in section 3.1 modules were created and tested with individual testbenches before being combined in the top level control modules. Most of the modules, including the rotate operations, were done in combination logic blocks for initial simplicity to ensure correct simulations. While this might work in simulations, it could be unreliable and inefficient when implemented in an FPGA. Also parameters, demonstrated in Listing 3.3, were used so the different modes could use the same code but the only testing done so far is with the SIMON32/64 version.

```
1  module SIMON_control
2  #(
3  parameter N = 16,
4  parameter M = 4,
5  parameter T = 32
6  )
7  (
8  input logic clk,
9  output int count,
10 output logic done,
11 input logic [2*N-1:0] plain,
12 output logic [2*N-1:0] cipher,
13 input logic [M-1:0][N-1:0] key
14 );
```

Listing 3.3: Encryption control module

Figure 3.2 shows the ModelSim simulation results of the control module, Listing 3.3 set in the 32/64 mode, for encryption with the correct ciphertext appearing on the cipher port only after the done signal is high.

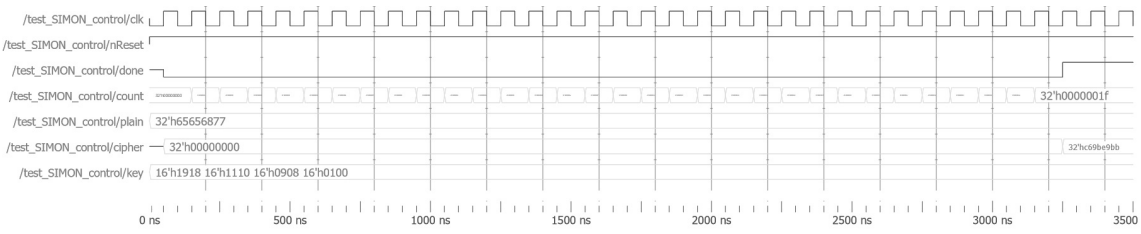


Figure 3.2: Simulations Results for encryption in System Verilog.

The synthesis results, using Quartus Prime, shown in Appendix A, show a RTL netview of the module. The synthesis shows that this module requires 161 pins and 131 registers and when a GE multiplier of 4 is used this encryption module has a GE area of 524 which is well below the 2000 limit set in section 2.1 and similar to that in [22]. Although, the module doesn't provide complete functionality so those values will most likely increase.

Chapter 4

Future Plan

To continue this project the System Verilog code needs to be improved with more sequential operations to improve reliability and efficiency. Serializing the algorithm at the byte level could also be explored which could have some interesting results similar to the bit serial version presented in [28]. Work will also have to be done to implement the System Verilog top module onto an FPGA and interface it with communication module, that could be provided by the FPGA development board used or from an OpenCores.org project[29]. When the FPGA is functioning correctly tests will need to be developed to determine the throughput in hardware and software, preferably operating at similar clock frequencies to ensure the results will be as comparable as possible. The hosted C program, section 3.1, should also be adapted to be programmed onto an 8 or 16 bit microcontroller and then the same parameters will be tested and compared.

Bibliography

[1] 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

[2] 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/

[3] A. Juels and S. A. Weis, "Authenticating Pervasive Devices with Human Protocols," *Advances in CryptologyCRYPTO*, 2005. [Online]. Available: http://www.arijuels.com/wp-content/uploads/2013/09/JW05.pdf

[4] M. David, D. C. Ranasinghe, and T. Larsen, "A2U2: A stream cipher for printed electronics RFID tags," in *2011 IEEE International Conference on RFID*. IEEE, apr 2011, pp. 176–183. [Online]. Available: http://ieeexplore.ieee.org/document/5764619/

[5] P. C. v. O. Alfred J. Menezes and S. A. Vanstone, "Overview of Cryptography," in *Handbook of Applied Cryptography*. CRC Press, 1996. [Online]. Available: http://cacr.uwaterloo.ca/hac/

[6] Bruce Schneier, "Self-Study Course in Block Cipher Cryptanalysis," *Cryptologia*, vol. 24, no. 1, pp. 18 – 34, 2000. [Online]. Available: https://www.schneier.com/academic/archives/2000/01/self-study{_}course{_}in.html

[7] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, oct 1949. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6769090

[8] H. Feistel, "Cryptography and Computer Privacy," *Scientific American*, vol. 228, no. 5, pp. 15 – 23, 1973.

[9] M. J. B. Robshaw, "Stream Ciphers," *RSA Laboratories*, no. 2, 1995.

[10] M. Dworkin, "Recommendation for block cipher modes of operation: methods and techniques," *NIST Special Publication*, pp. 800–38, 2001. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[11] N. Computer Security Division, "FIPS 46-3, Data Encryption Standard (DES) (withdrawn May 19, 2005)," *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION*, 1999. [Online].

Available: https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf

[12] ——, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," 2001. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[13] Bruce Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," in *Fast Software Encryption, Cambridge Security Workshop Proceedings*, 1994, pp. 192 – 204.

[14] T. S. Team, "COPACOBANA - Special-Purpose Hardware for Code-Breaking." [Online]. Available: http://www.sciengines.com/copacobana/

[15] M. Strachacki and S. Szczepanski, "Implementation of AES algorithm resistant to differential power analysis," in *2008 15th IEEE International Conference on Electronics, Circuits and Systems*. IEEE, aug 2008, pp. 214–217. [Online]. Available: http://ieeexplore.ieee.org/document/4674829/

[16] O. d. S. M. Gomes and R. L. Moreno, "A Compact S-Box Module for 128/192/256-bit Symmetric Cryptography Hardware," in *2016 9th International Conference on Developments in eSystems Engineering (DeSE)*. IEEE, aug 2016, pp. 94–97. [Online]. Available: http://ieeexplore.ieee.org/document/7930630/

[17] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang, "Pushing the Limits: A Very Compact and a Threshold Implementation of AES." Springer, Berlin, Heidelberg, 2011, pp. 69–88. [Online]. Available: http://link.springer.com/10.1007/978-3-642-20465-4{_}6

[18] S. R. Chatterjee, S. Majumder, B. Pramanik, and M. Chakraborty, "FPGA Implementation of Pipelined Blowfish Algorithm," in *2014 Fifth International Symposium on Electronic System Design*. IEEE, dec 2014, pp. 208–209. [Online]. Available: http://ieeexplore.ieee.org/document/7172778/

[19] K. N. Prasetyo, Y. Purwanto, and D. Darlis, "An implementation of data encryption for Internet of Things using blowfish algorithm on FPGA," in *2014 2nd International Conference on Information and Communication Technology (ICoICT)*. IEEE, may 2014, pp. 75–79. [Online]. Available: http://ieeexplore.ieee.org/document/6914043/

[20] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," 2007. [Online]. Available: http://lightweightcrypto.org/present/present{_}ches2007.pdf

[21] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Kneževi, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçn, "PRINCE A Low-latency Block Cipher for Pervasive Computing Applications," 2012. [Online]. Available: https://eprint.iacr.org/2012/529.pdf

[22] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK Families of Lightweight Block Ciphers," 2013. [Online]. Available: https://eprint.iacr.org/2013/404.pdf

[23] K. A. Mckay, L. Bassham, M. Sönmez, and T. N. Mouha, "Report on Lightweight Cryptography." [Online]. Available: http://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf

[24] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*. New York, New York, USA: ACM Press, 2015, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2744769.2747946

[25] A. Aysu, E. Gulcan, and P. Schaumont, "SIMON Says: Break Area Records of Block Ciphers on FPGAs," *IEEE Embedded Systems Letters*, vol. 6, no. 2, pp. 37–40, jun 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6782431/

[26] A. Shahverdi, M. Taha, and T. Eisenbarth, "Silent Simon: A threshold implementation under 100 slices," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, may 2015, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7140227/

[27] S. Banik, A. Bogdanov, and F. Regazzoni, "Exploring the energy consumption of lightweight blockciphers in FPGA," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, dec 2015, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7393308/

[28] J. d. C. G. Vasquez, F. Borges, R. Portugal, and P. Lara, "An Efficient One-Bit Model for Differential Fault Analysis on Simon Family," in *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, sep 2015, pp. 61–70. [Online]. Available: http://ieeexplore.ieee.org/document/7426153/

[29] OpenCores, "Home :: OpenCores." [Online]. Available: https://opencores.org/

Appendix A

SIMON32/64 Synthesis

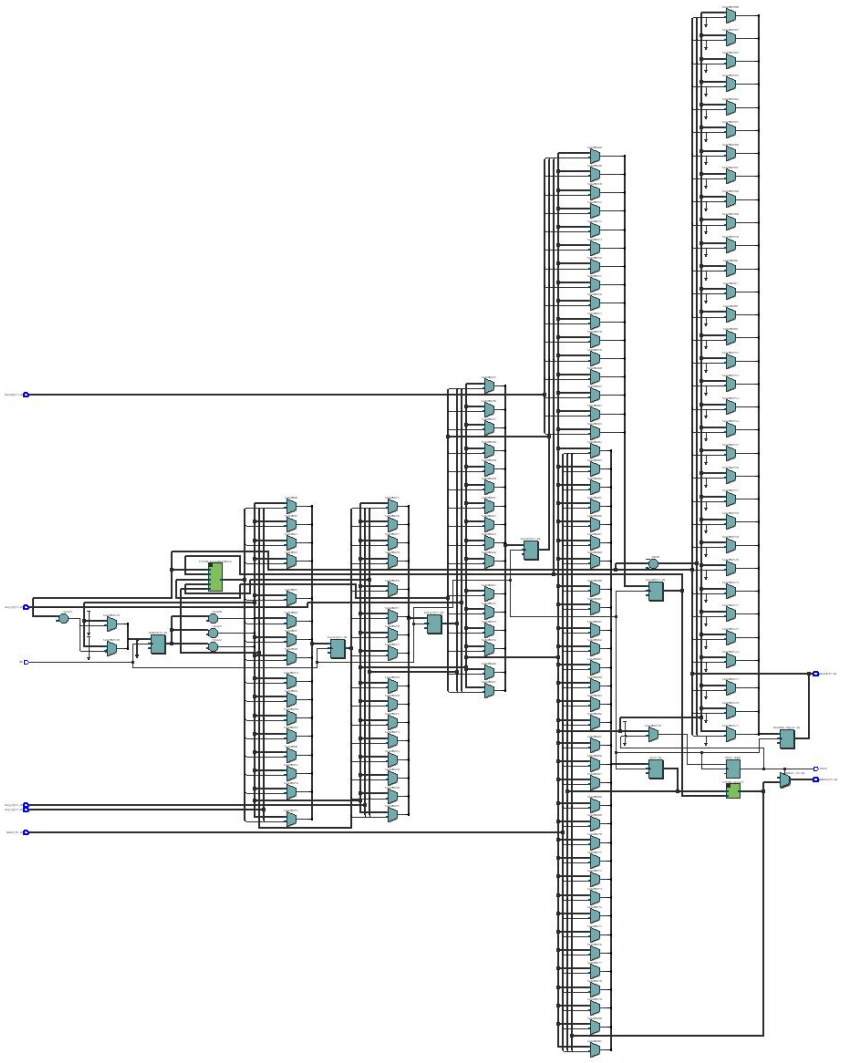
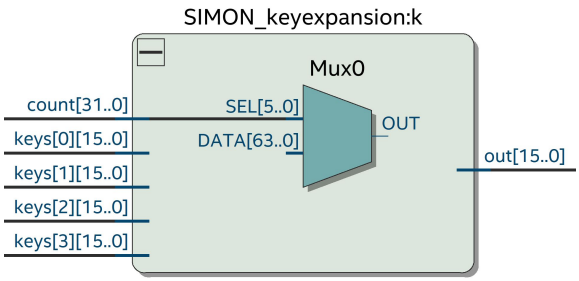
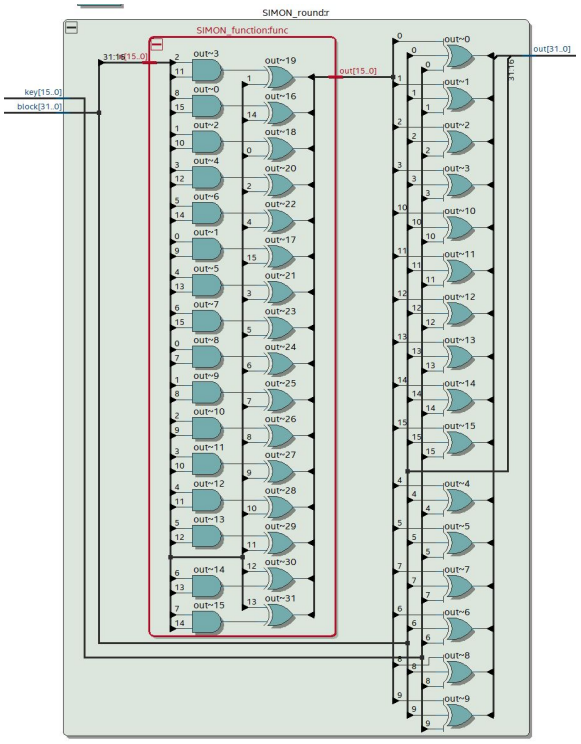


Figure A.1: Synthesis of SIMON control module.



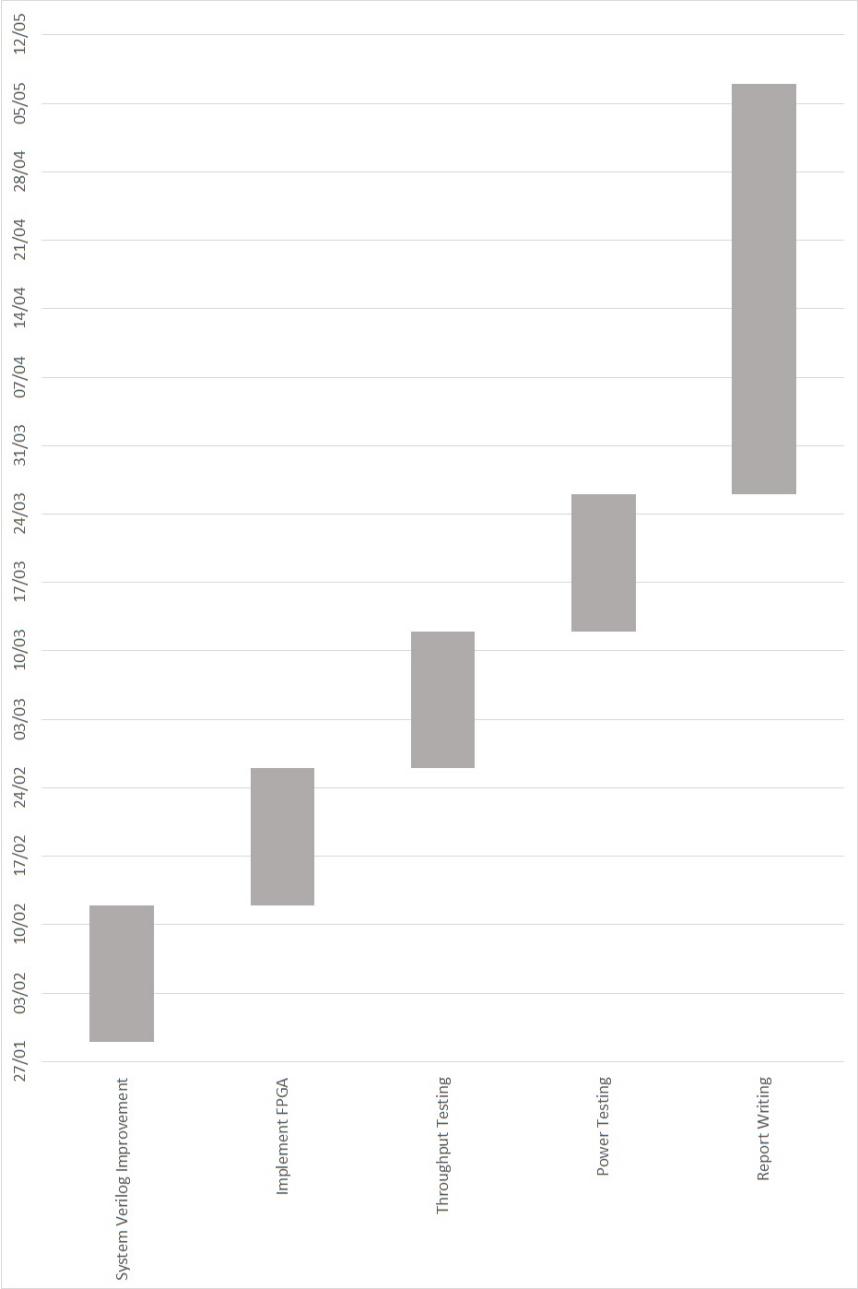
(a) Synthesis of the key expansion module.



(b) Synthesis of round function module.

Appendix B

Gantt Chart



Appendix C

Hosted C

SIMON_host.h

```
#ifndef SIMON_HOST_H
#define SIMON_HOST_H

#include <stdint.h>

#if   defined S32_64
  #define N 16
  #define M 4
  #define T 32
  #define j 0
#elif defined S48_72
  #define N 24
  #define M 3
  #define T 36
  #define j 0
#elif defined S48_96
  #define N 24
  #define M 4
  #define T 36
  #define j 1
#elif defined S64_96
  #define N 32
  #define M 3
  #define T 42
  #define j 2
#elif defined S64_128
  #define N 32
  #define M 4
  #define T 44
  #define j 3
#elif defined S96_96
  #define N 48
  #define M 2
  #define T 52
  #define j 2
#elif defined S96_144
  #define N 48
  #define M 3
```

```
39   #define T 54
40   #define j 3
41 #elif defined S128_128
42   #define N 64
43   #define M 2
44   #define T 68
45   #define j 2
46 #elif defined S128_192
47   #define N 64
48   #define M 3
49   #define T 69
50   #define j 3
51 #elif defined S128_256
52   #define N 64
53   #define M 4
54   #define T 72
55   #define j 4
56 #else
57   #error INVALID MODE
58 #endif
59
60 typedef uint32_t  uint24_t;
61 typedef uint64_t  uint48_t;
62 #define TYPE_(x)  uint ## x ## _t
63 #define TYPE(x)   TYPE_(x)
64 #define UINT(x, n)  typedef struct n  { TYPE(x) v : x;  } n;
65 UINT(N, word);
66
67 typedef word    block[2];
68 typedef word    key[M];
69 typedef word    keys[T];
70
71 TYPE(N) ROTL    (word x,  uint8_t n );
72 TYPE(N) ROTR    (word x,  uint8_t n );
73
74 TYPE(N) F     (word x         );
75 void  ROUND   (block b, word k    );
76 void  KEXP_PRE  (keys ks, key k   );
77 void  KEXP_INL  (key k, TYPE(8) i     );
78 void  ENCRYPT_PRE (const block p, block c,  key k   );
79 void  ENCRYPT_INL (const block p, block c,  key k   );
80 void  DECRYPT_PRE (const block c, block p,  key k   );
81
82 #endif
```

SIMON_host.c

```
1 #include "SIMON_host.h"
2 #include <stdio.h>
3
4 const TYPE(8) z[5][62] =
5 {
6   { 1,1,1,1,1,0,1,0,0,0,1,0,0,1,0,1,0,1,1,0,0,
7     0,0,1,1,1,0,0,1,1,0,1,1,1,1,1,0,1,0,0,0,1,
8     0,0,1,0,1,0,1,1,0,0,0,0,1,1,1,0,0,1,1,0   },
9   { 1,0,0,0,1,1,1,0,1,1,1,1,1,0,0,1,0,0,1,1,0,
10    0,0,0,1,0,1,1,0,1,0,1,0,0,0,1,1,1,0,1,1,1,
11    1,1,0,0,1,0,0,1,1,0,0,0,0,1,0,1,1,0,1,0   },
12  { 1,0,1,0,1,1,1,1,1,0,1,1,1,0,0,0,0,0,0,1,1,0,
13    1,0,0,1,0,0,1,1,0,0,0,1,0,1,0,0,0,0,1,0,0,
14    0,1,1,1,1,1,1,1,0,0,1,0,1,1,0,1,1,0,0,1,1   },
15  { 1,1,0,1,1,0,1,1,1,0,1,0,1,1,0,0,0,1,1,0,0,
16    1,0,1,1,1,1,0,0,0,0,0,0,1,0,0,1,0,0,0,1,0,
17    1,0,0,1,1,1,0,0,1,1,0,1,0,0,0,0,1,1,1,1 },
18  { 1,1,0,1,0,0,0,1,1,1,1,0,0,1,1,0,1,0,1,1,0,
19    1,1,0,0,0,1,0,0,0,0,0,0,1,0,1,1,1,0,0,0,0,
20    1,1,0,0,1,0,1,0,0,1,0,0,1,1,1,0,1,1,1,1   }
21 };
22
23 TYPE(N) ROTL  (word x,  uint8_t n )
24 {
25   return (x.v << n) | (x.v >> (N-n));
26 }
27
28 TYPE(N) ROTR  (word x,  uint8_t n )
29 {
30   return (x.v << (N-n)) | (x.v >> n);
31 }
32
33 TYPE(N) F   (word x          )
34 {
35   return (ROTL(x,1) & ROTL(x,8)) ^ ROTL(x, 2);
36 }
37
38 void  ROUND (block b, word k    )
39 {
40   word tmp = b[0];
41   b[0].v = b[1].v ^ F(b[0]) ^ k.v;
42   b[1] = tmp;
43 }
44
45 void  KEXP_PRE  (keys ks, key k   )
46 {
47   word tmp;
48   int8_t i;
49   for(i=M; i>0; i--)  ks[i-1].v = k[M-i].v;
50
51   for(i=M; i<T; i++)
52   {
53     tmp.v = ROTR(ks[i-1], 3);
54 #if (M == 4)
55     tmp.v ^= ks[i-3].v;
56 #endif
```

```
57      tmp.v ^= ROTR(tmp, 1);
58      ks[i].v = ~ks[i-M].v ^ tmp.v ^ (z[j][(i-M) % 62]) ^ 3;
59    }
60  }
61
62  void  KEXP_INL  (key k, TYPE(8) i    )
63  {
64    word tmp;
65    tmp.v = ROTR(k[M-1], 3);
66  #if (M==4)
67    tmp.v ^= k[M-3].v;
68  #endif
69    tmp.v ^= ROTR(tmp, 1);
70    tmp.v ^= ~k[0].v ^ (z[j][i % 62]) ^ 3;
71
72    k[0] = k[1];
73  #if (M>2)
74    k[1] = k[2];
75  #endif
76  #if (M>3)
77    k[2] = k[3];
78  #endif
79    k[M-1] = tmp;
80  }
81
82  void  ENCRYPT_PRE (const block p, block c,  key k    )
83  {
84    c[0].v = p[0].v;
85    c[1].v = p[1].v;
86
87    keys ks;
88    KEXP_PRE(ks, k);
89
90    TYPE(8) i;
91    for(i=0; i<T; i++)
92    {
93      ROUND(c, ks[i]);
94    }
95  }
96
97  void  ENCRYPT_INL (const block p, block c,  key k    )
98  {
99    c[0].v = p[0].v;
100   c[1].v = p[1].v;
101   key ks;
102
103   TYPE(8) i;
104   for(i=M; i>0; i--)  ks[i-1].v = k[M-i].v;
105
106   for(i=0; i<T; i++)
107   {
108     ROUND(c, ks[0]);
109     KEXP_INL(ks, i);
110   }
111 }
112
113 void  DECRYPT_PRE (const block c, block p,  key k    )
114 {
115   p[0].v = c[1].v;
```

```
116   p[1].v = c[0].v;
117
118   keys ks;
119   KEXP_PRE(ks, k);
120
121   TYPE(8) i;
122   for(i=T; i>0; i--)
123   {
124     ROUND(p, ks[i-1]);
125   }
126   word tmp = p[0];
127   p[0].v = p[1].v;
128   p[1].v = tmp.v;
129 }
```

SIMON_test.h

```
#ifndef SIMON_TEST_H
#define SIMON_TEST_H

#include "SIMON_host.h"

#if (N<=32)
#define PRINTHEX(x) printf("%0*X", N/4, x )
#else
#define PRINTHEX(x) printf("%0*llX", N/4, x )
#endif

#define NEWLINE() printf("\n\r"      )
#define TAB()    printf("\t"        )

void  test();

#endif
```

SIMON_test.c

```
#include "SIMON_test.h"
#include "SIMON_host.h"
#include <stdio.h>

#if defined S32_64
  key   k = { 0x1918, 0x1110,
              0x0908, 0x0100                };
  block   p = { 0x6565, 0x6877                  };
  block   c = { 0xC69B, 0xE9BB                  };
#elif defined S48_72
  key   k = { 0x121110, 0x0A0908,
              0x020100                    };
  block   p   = { 0x612067, 0x6E696C              };
  block   c = { 0xDAE5AC, 0x292CAC              };
#elif defined S48_96
  key   k = { 0x1A1918, 0x121110,
              0x0A0908, 0x020100              };
  block p = { 0x726963, 0x20646E                };
  block c = { 0x6E06A5, 0xACF156                };
#elif defined S64_96
  key   k = { 0x13121110, 0x0B0A0908,
              0x03020100                  };
  block p = { 0x6F722067, 0x6E696C63            };
  block c = { 0x5CA2E27F, 0x111A8FC8            };
#elif defined S64_128
  key   k = { 0x1B1A1918, 0x13121110,
              0x0B0A0908, 0x03020100            };
  block p = { 0x656B696C, 0x20646E75            };
  block c = { 0x44C8FC20, 0xB9DFA07A            };
#elif defined S96_96
  key   k = { 0x0D0C0B0A0908, 0x050403020100       };
  block p = { 0x2072616C6C69, 0x702065687420       };
  block c = { 0x602807A462b4, 0x69063D8FF082       };
#elif defined S96_144
  key   k = { 0x151413121110, 0x0D0C0B0A0908,
```

```
36              0x050403020100                  };
37    block p = { 0x746168742074, 0x73756420666f       };
38    block c = { 0xECAD1C6C451E, 0x3f59C5DB1AE9       };
39 #elif defined S128_128
40    key   k = { 0x0F0E0D0C0B0A0908, 0x0706050403020100   };
41    block p = { 0x6373656420737265, 0x6C6C657661727420   };
42    block c = { 0x49681B1E1E54FE3F, 0x65AA832AF84E0BBC   };
43 #elif defined S128_192
44    key   k = { 0x1716151413121110, 0x0F0E0D0C0B0A0908,
45              0x0706050403020100                  };
46    block p = { 0x206572656874206E, 0x6568772065626972   };
47    block c = { 0xC4AC61EFFCDC0D4F, 0x6C9C8D6E2597b85B   };
48 #elif defined S128_256
49    key   k = { 0x1F1E1D1C1B1A1918, 0x1716151413121110,
50              0x0F0E0D0C0B0A0908, 0x0706050403020100   };
51    block p = { 0x74206E69206D6F6F, 0x6D69732061207369   };
52    block c = { 0x8D2B5579AFC8A3A0, 0x3BF72A87EfE7b868   };
53 #endif
54
55 void  test()
56 {
57    ///*
58    printf("|\tTESTING PRE KEY EXPANSION\t\t|");
59
60    NEWLINE();
61    printf("|\tKEY:\t\t");
62    TYPE(8) i;
63    for(i=0; i<M; i++)  { PRINTHEX(k[i].v); }
64    printf("\t|\n");
65
66    block cipher, plain;
67    cipher[0].v = 0;
68    cipher[1].v = 0;
69    printf("|\tPLAIN->CIPHER:\t");
70    PRINTHEX(p[0].v);    PRINTHEX(p[1].v);
71    printf("->");
72    PRINTHEX(cipher[0].v);  PRINTHEX(cipher[1].v);
73    printf("\t|\tSTART\t|\n");
74
75    ENCRYPT_PRE(p, cipher, k);
76    printf("|\tPLAIN->CIPHER:\t");
77    PRINTHEX(p[0].v);    PRINTHEX(p[1].v);
78    printf("->");
79    PRINTHEX(cipher[0].v);  PRINTHEX(cipher[1].v);
80    printf("\t|\tENCRYPT\t|\n");
81
82    printf("|\tPLAIN->CIPHER:\t");
83    PRINTHEX(p[0].v);    PRINTHEX(p[1].v);
84    printf("->");
85    PRINTHEX(c[0].v); PRINTHEX(c[1].v);
86    printf("\t|\tTEST\t|\n");
87
88    ///*
89    plain[0].v = 0;
90    plain[1].v = 0;
91    DECRYPT_PRE(c, plain, k);
92    printf("|\tPLAIN<-CIPHER:\t");
93    PRINTHEX(plain[0].v);    PRINTHEX(plain[1].v);
94    printf("<-");
```

```
95    PRINTHEX(c[0].v); PRINTHEX(c[1].v);
96    printf("\t|\tDECRYPT\t|\n");
97    //*/
98
99    printf("|\tTESTING IN LOOP KEY EXPANSION\t\t|");
100
101   NEWLINE();
102   printf("|\tKEY:\t\t");
103   for(i=0; i<M; i++)  { PRINTHEX(k[i].v);}
104   printf("\t|\n");
105
106   cipher[0].v = 0;
107   cipher[1].v = 0;
108   printf("|\tPLAIN->CIPHER:\t");
109   PRINTHEX(p[0].v);    PRINTHEX(p[1].v);
110   printf("->");
111   PRINTHEX(cipher[0].v);  PRINTHEX(cipher[1].v);
112   printf("\t|\tSTART\t|\n");
113
114   ENCRYPT_INL(p, cipher, k);
115   printf("|\tPLAIN->CIPHER:\t");
116   PRINTHEX(p[0].v);    PRINTHEX(p[1].v);
117   printf("->");
118   PRINTHEX(cipher[0].v);  PRINTHEX(cipher[1].v);
119   printf("\t|\tENCRYPT\t|\n");
120
121   printf("|\tPLAIN->CIPHER:\t");
122   PRINTHEX(p[0].v);    PRINTHEX(p[1].v);
123   printf("->");
124   PRINTHEX(c[0].v); PRINTHEX(c[1].v);
125   printf("\t|\tTEST\t|\n");
126   //*/
127
128   /*
129   word a;
130   a = k[0];
131   PRINTHEX(a.v);
132   //*/
133 }
```