Dublin City University – School of Electronic Engineering

Implementation of the Simon Block Ciphers

Brandon Walsh – April 2017

Bachelor of Engineering in Electronic and Computer Engineering

Supervised by Dr. X. Wang

Acknowledgements

I would like to thank my supervisor and VHDL teacher Xiaojun Wang; for their help, support, lending me the Zybo board for this project and for telling me that I didn't need to implement every aspect of the cipher into one design, because it's needless and no-one would have wanted that anyway. It sounds small; but I was really panicked over that, so I'm pretty grateful.

I'd also like to apologise in advance for having him read this report last minute, and hounding him for reviews so I can get it fixed up and printed.

I would also like to thank my technical supervisor Robert Clare, for lending me that ribbon cable.

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the DCU Academic Integrity and Plagiarism at https://www4.dcu.ie/sites/default/files/policy/1%20-%20integrity_and_plagiarism_ovpaa_v3.pdf and IEEE referencing guidelines found at https://loop.dcu.ie/mod/url/view.php?id=448779

Name: _____ Date: _____

Contents

Table of Figures

Abstract

This report details the development of three implementations of the lightweight Simon block ciphers in hardware of various attributes and specialities, and the corresponding connectivity to a C++ program which is capable of encrypting and decrypting messages using the systems.

Alongside this, an online explanation paper was created to create an easy and friendly point of access for those who wish to use the cipher in their own designs, detailing the inner workings of segments and with extensive process data.

The systems are written in VHDL and tested using a FPGA development board, in conjunction with a Raspberry Pi which could communicate and control the developed hardware.

Introduction

For my final year project, I have been asked to develop an implementation of the Simon block ciphers in VHDL, and review my design's performance regarding logical efficiency and hardware usage.

Simon is a collection of ten lightweight block ciphers released to the public by the United States National Security Agency (NSA) in mid-2013 [1]. Overall each cipher is almost identical, with different key and message bit lengths diversifying each implementation. Simon is a balanced Feistel cipher, capable of encrypting blocks of data from 32bits up to 128bits in a single execution.

The cipher was designed to fill a cryptographic gap in the market. Few encryption methods have been designed to work in the relatively new field of "lightweight cryptography" [2], an area concerning small, technically constrained devices. As the world moves towards a more connected and computerised one; increasingly devices are being built with networking capability. It is important therefore, that such devices should have some level of protection when connecting to each other, or to larger control machines. A hacker should not be able to take control of the security in your house or the breaks of your car, for example.

Simon is a well-tested encrypting algorithm with these constraints in mind. Designed to be realised in hardware - though perfectly functional in software - the designers proclaim Simon's small hardware footprint, high throughput and its ability to be serialised at various levels. Decryption of Simon encrypted data is a process very similar to encryption; in fact, it is almost the very reverse of it, thus it is entirely possible to use much of the encryption components for decryption.

Simon is also general use, giving developers the flexibility to integrate it into their designs; thus, providing developers with an easier step towards encryption of their designs.

For this project, I intended to create an efficient and cost effective implementation of the cipher. Moreover, I intended to develop a toolbox for developers that might be interested in using the cipher, so that they can get a boost in integrating encryption into their designs. Upon review of the papers around Simon, I have not been able to find one that explicitly describes the design of each segment of the ciphers, nor one that contains full encryption process data in a clear and explicit way. This is my lead motivation for this project; not to create the greatest implementation of the cipher, but to create an easy and friendly point of access for those who wish to use it.

Technical Background

For my project, I will be using mainly three technologies to develop and test my design

VHDL

VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) is a description language developed by the US Department of Defence in the early 80's, initially as a means of documenting the designs of integrated circuits that supplier companies were creating for them [3] Soon afterward, simulators were created to test designs written in this language, and slowly the language morphed into more of a programming language used to design circuits instead of just documenting them.

It is important to note however, that VHDL is not a programming language; but a description language. VHDL code is designed to be synthesized into a logic gate and connection layout, instead of an executable program. With that said of course, some simulators (namely GHDL, which I used for the initial development of my designs) can create executables from VHDL code. Though possible, this is not the intention of the language.

In 1988, the rights to VHDL were transferred to the IEEE as standardization of the language took place [4]

VHDL allows a developer to create large-scale logic systems, while utilising many of the project management aspects of a software programming language. Indeed, the language itself is based on the Ada programming language [5] which was created for the US Department of Defence in a bid to supersede the ~450 programming languages the department used at the time [6]

As a modern-day engineer, the language and tools have developed into a robust, and (somewhat) easy to use development tool, with IDEs such as Vivado [7] and ModalSim [8] covering the writing, debugging, simulation and logic-gate map generation processes. Though one could easily draw out one's designs on paper, working with this language is a welcome boon, allowing a rapid transition to simulation and testing of designs as one tinkers with them.

Zybo

For testing the developed designs in real hardware; my supervisor allowed me to borrow one of his Zybo Zynq-7000 ARM/FPGA development boards. [9] This in conjunction with the company's hardware design suite 'Vivado'; one is able to create VHDL designs and have them implemented in a field programmable gate array (FPGA), which can be connected to a range of standardized ports, such as VGA or Ethernet.

Though fitted with an Advanced RISC Machine (ARM) processor, I decided to use the provided ports to interface with an external device. This eased the development tool's learning curve, and also opened up testing possibilities as discussed in the following section.

C++ and Raspberry PI

A raspberry PI 3 model B microcomputer [10], loaded with the Raspbian OS [11] (which is based on the Debian OS [12] (which itself is a Linux distribution)) was used to interface the Zybo's Peripheral Module (Pmod) [13] ports with the PI's General-Purpose Input/Output (GPIO) pins.

On top of this; C++ programs were written to communicate with the GPIO pins and perform the actual testing of the encryption/decryption capabilities of any implemented design. Designing tests in this way (instead of solely using the Zybo) allowed for rapid development of the testing program, and made use of the cipher as a module a central feature from the outset.

Beyond this, developing a program that could communicate with the design from a Linux system; means that more advanced and useful features can be created and integrated with a wider system. For example; encrypting an entire document, or for use as a Trusted Platform Module (TPM) [14] to encrypt an entire system.

Previous Work on the Subject

Though I haven't been able to find any express uses of the cipher in the real world (I suspect as secrecy has been employed for an added layer of security) I have been able to find a number of software implementations, including one in VHDL [15] [16]

I restricted myself from viewing that VHDL version, though I have used a Java implementation which I found in order to get my JavaScript version working [17], I mainly used this to analyse and correct my encryption process.

It was from these searches that I determined one of the main focuses of my project; to create an accessible engineering explanation of the cipher for all to use. Having the encryption process data (of what the data looks like at different stages of the encryption) helped my understanding of the system, and I believe that documenting and formatting this data in an accessible way will help others in their implementation of the cipher.

Simon

As mentioned before; Simon is a balanced Feistel cipher, capable of encrypting blocks of data from 32bits up to 128bits in a single execution. In its most basic form, Simon is a collection of three different circuits; a key expander, a message encryptor and a message decryptor, referred to in this report as 'modules'. These circuits vary a little in response to the message and key bit lengths defined in the NSA's paper [1], but operation is mainly the same for all. There are 10 different message and key bit lengths, defined in this paper as 'methods'.

For encryption; a segment of the key provided is given to an encryption circuit along with the provided message. The circuit uses this segment to encrypt the message, producing a new message of the same length. The provided key is also mutated by a key expander circuit, producing a new key. A segment from this new key is given to another message encryptor circuit along with the message produced by the previous message encryptor circuit, producing another message.

This pattern is repeated over and over again a set number of times (defined in this paper as 'stages') which is defined by the method, until the message encryption is complete. Decryption is a very similar process bar two differences; first the encryption circuit is swapped for a decryption circuit, and second; the first decryption circuit needs the last mutated key (the second needing the second last, etc.) in an action akin to pushing the encrypted message backwards through the system.

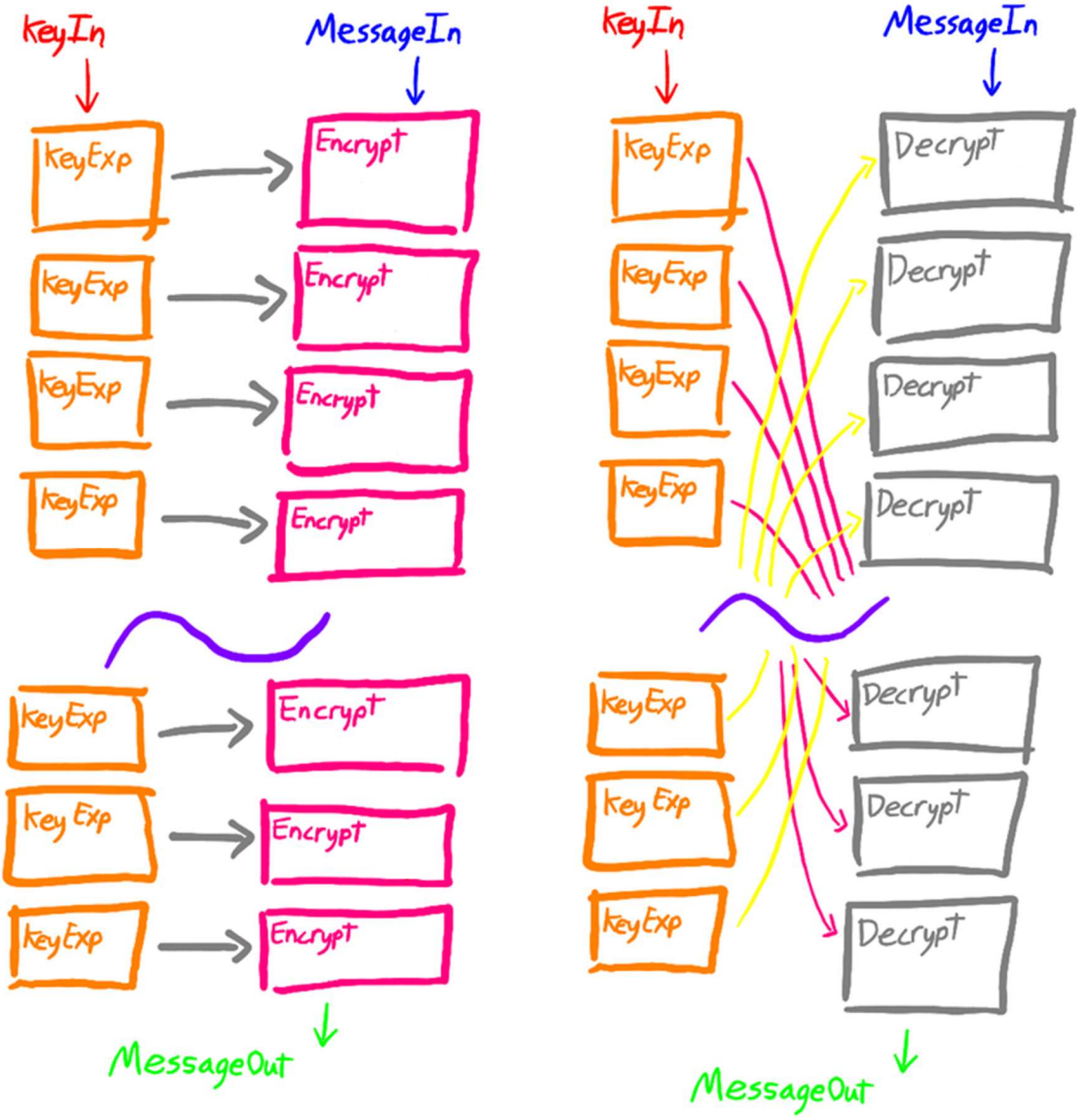

Figure 1 - Layout of System for Encryption and Decryption

As said before, the number of stages required is defined by the method;

Method	Number of Stages
1	32
2	36
3	36
4	42
5	44
6	52
7	54
8	68
9	69
10	72

Now let's look at the circuits themselves.

Key Expander

The key expander module uses a provided key and a predefined 'Z' value as a seed to produce a new key to be used by an encryption or decryption module. The newly produced key is then used by further key expansion modules.

A key can come in a variety of lengths and segments, as defined by the method;

Method	Key Length (bits)	Key Segments
1	64	4
2	72	3
3	96	4
4	96	3
5	128	4
6	96	2
7	144	3
8	128	2
9	192	3
10	256	4

The initial 'Z' value to be used is also defined by the method

Method	Selected Z
1	0
2	0
3	1
4	2
5	3
6	2
7	3
8	2
9	3
10	4

Z Types	Values
0	11111010001001010110000111001101111101000100101011000011100110
1	10001110111110010011000010110101000111011111001001100001011010
2	10101111011100000011010010011000101000010001111110010110110011
3	11011011101011000110010111100000010010001010011100110100001111
4	11010001111001101011011000100000010111000011001010010011101111

'Z' values are always 62 bits long, regardless of method. Only one bit of the 'Z' value is used in each key module, which is selected based on the stage in the key expansion process; most left bit first, the bit to its right second, etc. with the selection looping back to the start if more than 62 stages are used.

The received key is expanded like so;

```
newKeySegment = rightRotate( keySegment(0) )threeTimes
if(keySegments = 4)then -> newKeySegment = newKeySegment XOR ( keySegment(2) )
newKeySegment = newKeySegment XOR rightRotate(newKeySegment)once
newKeySegment = newKeySegment XOR keySegment(keySegments-1)
newKeySegment = newKeySegment XOR (keySegmentLength-1 of 0's, with the selected Z bit)
newKeySegment = newKeySegment XOR -4;
```

The produced key is now ' newKeySegment' with the first (keySegments-1) segments of the received key attached

Message Encryptor

The message encryptor module takes a key segment and uses it to encrypt a provided message, producing an encrypted message which is used by subsequent encryption stages or produced as the system's encrypted message.

Messages come in two segments and in a variety of lengths, as defined by the method

Method	Message Length (bits)
1	32
2	48
3	48
4	64
5	64
6	96
7	96
8	128
9	128
10	128

It is worth noting that for each method; the length of each key segment and message segment is the same.

The received message is encrypted like so;

```
newMessageSegment = leftRotate( messageSegment(0) )once
                        AND leftRotate( messageSegment(0) )eightTimes
newMessageSegment = newMessageSegment XOR messageSegment(1)
newMessageSegment = newMessageSegment XOR leftRotate( messageSegment(0) )twice
newMessageSegment = newMessageSegment XOR keySegment(keySegments-1)
```

The produced message is now newMessageSegment with messageSegment 0 attached

Message Decryptor

The message decryptor module takes a key and uses it to decrypt a provided message, producing a decrypted message which is used by subsequent decryption stages or produce as the system's decrypted message. The key must be fully expanded at the beginning of decryption process, as the expanded stage keys are used in reverse order.

Messages come in two segments and in a variety of lengths, as defined by the method

Method	Message Length (bits)
1	32
2	48
3	48
4	64
5	64
6	96
7	96
8	128
9	128
10	128

It is worth noting again that for each method, the length of each key segment and message segment is the same.

The received message is decrypted like so;

```
newMessageSegment = leftRotate( messageSegment(1) )once
                    AND leftRotate( messageSegment(1) )eightTimes
newMessageSegment = newMessageSegment XOR messageSegment(0)
newMessageSegment = newMessageSegment XOR leftRotate( messageSegment(1) )twice
newMessageSegment = newMessageSegment XOR keySegment(keySegments-1)
```

The produced message is now messageSegment 1 with newMessageSegment attached

Please see the appendix for full examples of the operation of the key expander, message encryptor and message decryptor. Also, one can find a full stage-by-stage encryption and decryption process for method one.

Developed Designs

For this project, I have developed three different designs for the Simon cipher; Flow Logic, Register Transfer Level and the Crypto-Processor. Each design is split into subtypes; unified, methods (and in design three, modeAndMethods) These different subtypes split out the functionality of the cipher, allowing a designer the ability to choose the level of encryption complexity they want, or the range of encryption functions available.

Subtypes

unified	These allow the system to encrypt or decrypt with any of the methods
methods	These are a collection that specialize in one particular method, though allow encryption or decryption. They are much smaller than the unified version
modeAndMethods	These are a collection that specializes in one particular method and mode. They are the smallest of the collections.

Designs

Type One - Flow Logic

The flow logic design lays out the cipher in its entirety, allowing the user to pass data into the input and have it flow through all the required modules to produce a result. It is a pure combinational logic design, and as such no clock signal is needed. In this developed implementation, the encrypted and decrypted output is computed at the same time, with and additional 'mode' input determining which result to output.

Though not as optimised for through-put or size as the designs ahead; its simple layout and similarity to the basic model presented in the NSA's paper [1] servers as a good starting point for understanding the system.

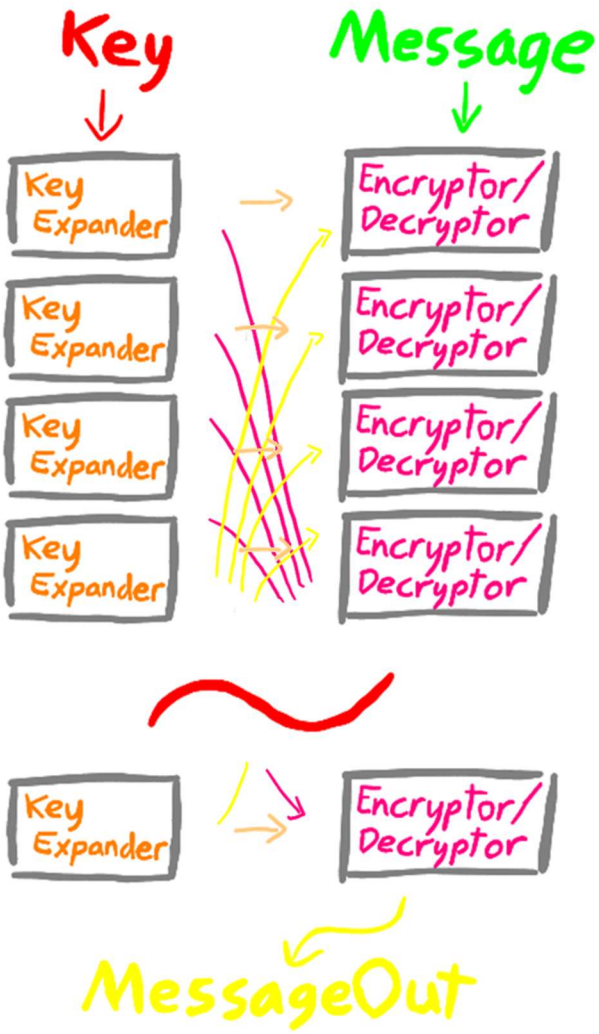

Figure 2 – Type One Flow Logic Design Layout

The encryption and decryption modules are tied together, so for the unified subtype an additional 4-bit input determining the method is used. This method value is passed to each encryptor/decryptor module, wherein the module can determine whether to process the message or let it pass through. This is done because the unified subtype must have enough encryptor/decryptor modules for the largest possible required stage number - 72, but different methods need a different amount of stages. Thus giving the encryptor/decryptor modules the ability to decide whether they should perform work or not, automatically adjusts the number of stages. Though more complicated than other subtypes, devices incorporating this “multi-method” design would be able to communicate with a device using any form of the cipher.

This decision-making ability is only useful in the unified subtype and is removed in the other subtypes, where the number of stages is hardwired in.

Below is a diagram of the unified subtype encrypting and decrypting a message of method 1. You can notice how the encryption and decryption modules decide whether to process the message or allow it to pass through, dependent on their position in the system.

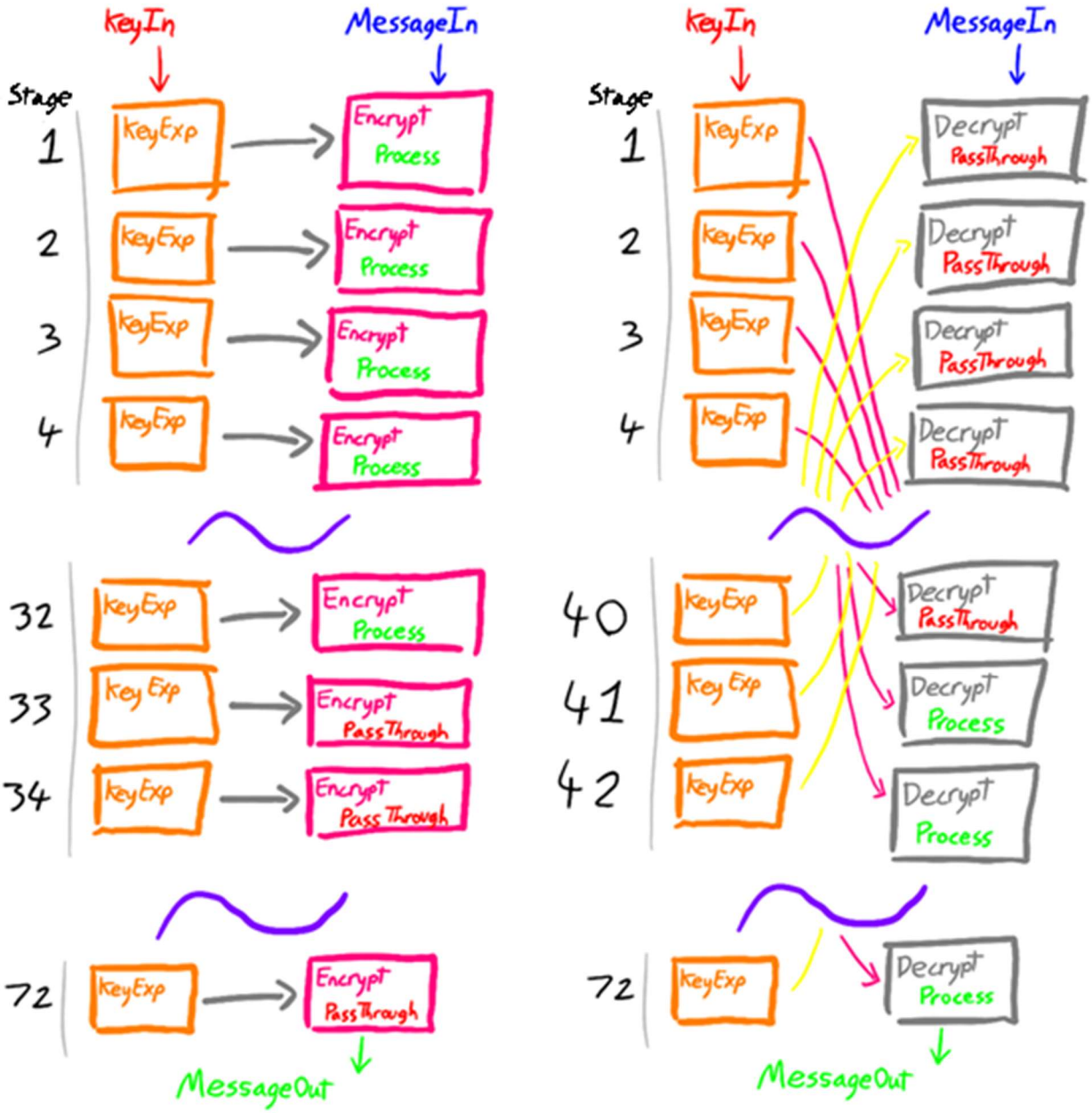

Figure 3 - Operation of the Flow Logic Unified Subtype Encrypting And Decrypting a Message

Benefits/Disadvantages

Advantage	can allow encryption/decryption to occur in a single clock cycle, however this restricts the maximum clock frequency
Advantage	encrypting/decrypting messages with the different keys takes no longer than encrypting/decrypting with the same key
Disadvantage	as all the modules are being laid out in their entirety, the design takes up a large space
Disadvantage	much of the circuitry does nothing for the majority of the flow time

Type Two – Register Transfer Level

The Register Transfer Level design expands upon the previous design, addressing the issue of inactive modules. Here, every encryption and decryption module is separated out between registers, which can store the intermediate results of the encryption/decryption progress of a message. The key is completely expanded at the very first stage, and this data is passed through to the registers as well. In this way, all parts of the circuit can be utilised for processing different messages in a pipeline.

Similar to the previous design, the unified subtype has a 4-bit input determining the method to be used. This information is also stored by each register block and passes through the system alongside the message. This data is given to each encryptor/decryptor module as before.

In addition; anther one bit input connection is used. This is for the external clock, which controls the progress of the messages through the system. For every tick; a message progresses by one stage.

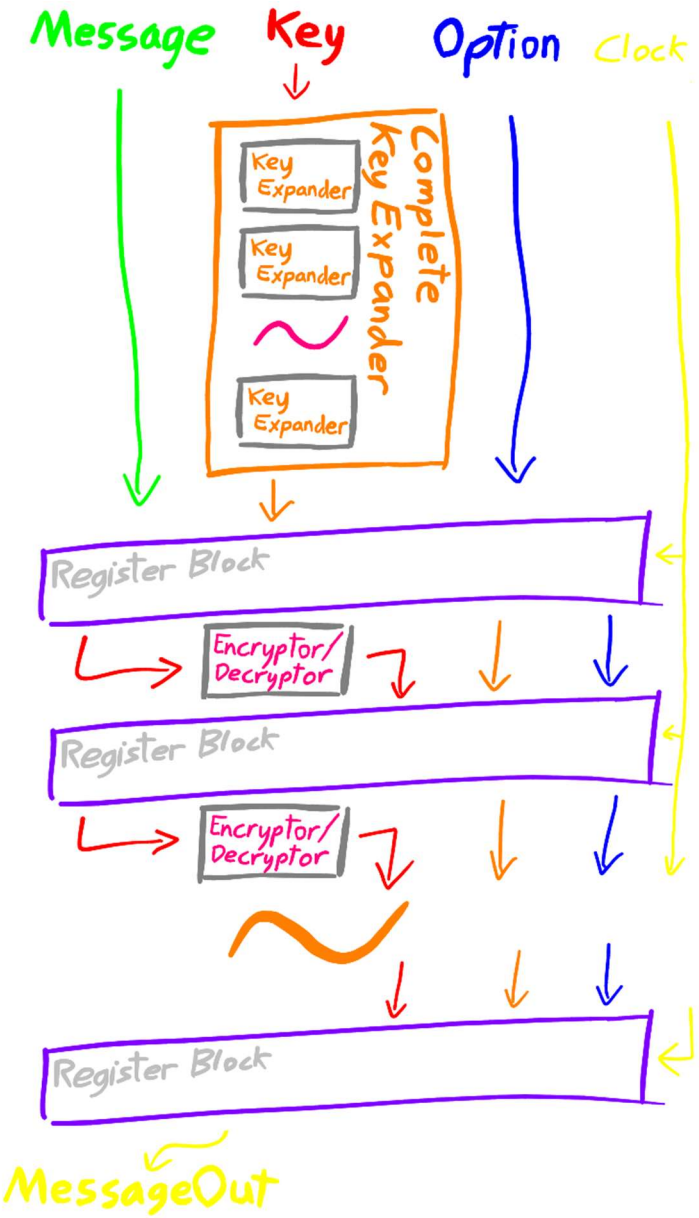

Figure 4 – Type Two Register Transfer Level Design Layout

Benefits/Disadvantages

Advantage	multiple messages can be encrypted/decrypted in pipeline and almost simultaneously allowing for a greater throughput of messages.
Disadvantage	this is the largest of the designs, as not only does it contain all the circuitry of the previous design, but now there is a block of pipeline registers for each stage.
Disadvantage	having the key completely expanded at the first stage can cost a lot of time, and the amount of individual registers in the register blocks required to store this information for each stage of encryption/decryption, means that the design grows quite large. This design decision was made, as for decryption; the first decryptor module requires the last expanded key, thus the key must be expanded in its entirety before any work can begin. As the lead concept in this design was to improve flow-through in the system, the encryption mode has to run at the same speed as decryption. Thus, the initial key expansion is also done for encryption. As you would expect, an encryption mode only subtype wouldn't require such initial processing.
Advantage	though the initial key expansion costs time; encrypting/decrypting batches of messages with the same key is quite efficient. As no key expansion time has to be factored in; the system could be run at a higher clock rate, in addition to benefiting from the near simultaneous encryption/decryption.

Type Three – Crypto-Processor

This design takes the concept of multi-staged encryption from the previous design, and distils it down into a single repeatable process. Instead of having a set of encryption and decryption modules for each stage; only one encryption and decryption module is implemented and are used repeatedly to perform encryption/decryption operations. This means that the finished implementation can take up a much smaller hardware footprint than previous designs.

Similar to the Register Transfer Level design, the provided key must be fully expanded before any encryption or decryption can begin; however, this particular design contains a "modeAndMethods" subtype for encryption only, which does not need such complete key expansion time and thus only requires a fraction of the key expansion logic and registers.

Like the previous two versions, for the unified subtype an additional 4-bit input determining the method is used. This information is stored by a register block for use with the encryptor/decryptor modules similar to before. This version also has a one bit input for the external clock, which controls the progress of the messages through the system. As before, for every tick; a message progresses by one stage.

Importantly; this version has a final single bit input for loading the register block. The main package for this system splits the input message, key and control data into two register blocks; one for the incoming message, and one for the currently processing message. When the load connection is activated, the first set of data is copied into the second. In this way; the internal processing can go unaffected while a new job is loaded in.

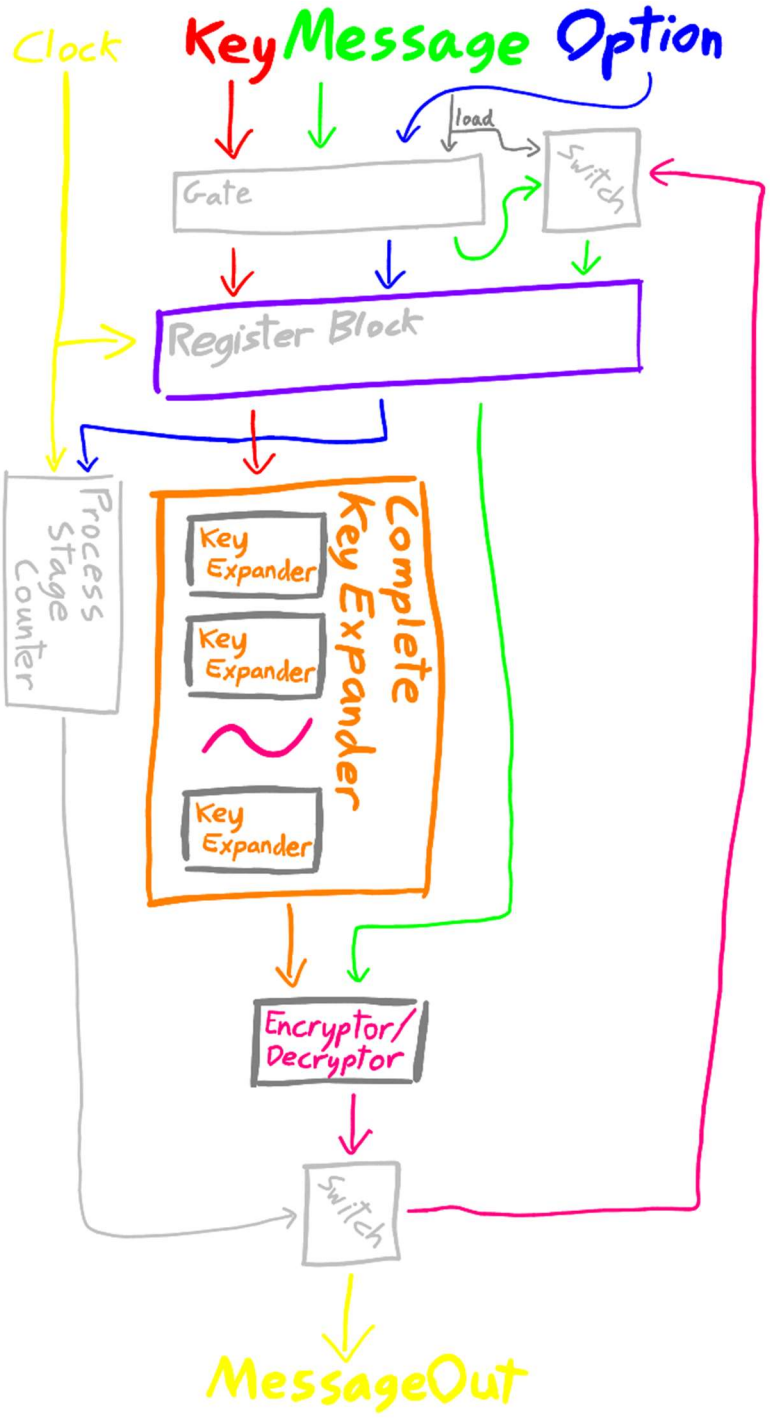

Figure 5 – Type Three Crypto-Processor Design Layout

Benefits/Disadvantages

Advantage	smallest hardware footprint
Disadvantage	only one message can be processed at a time
Disadvantage	complete key expansion time (for 'unified' and 'methods' subtypes)

Implementation paper

To accompany and in addition to this report; a paper on the subject of the cipher has been made for the purposes of helping any developer who wished to use the Simon cipher in their design, so to understand the system in a clear and friendly way [18]. This paper contains clear explanations on how each module of the cipher operates, and many test vectors for each module, plus full encryption and decryption systems; spanning all the message and key bit lengths described in the original Simon paper [1].

Git Repository

This entire project, including its code and test results, is being developed and stored in a git hosted by the website 'github.com' [19], and publicly available to the world. Here one can find the code for each design, subtype and version of the cipher described in this report, plus the C++ code used to communicate with the development board.

Implementations, Testing and Results

JavaScript Implementation

To begin on this project; I first developed a JavaScript version of the cipher [20] [21], using a Java implementation by Tim Whittington [17] in order to flesh out my knowledge of the system and confirm how the encryption and decryption were performed.

During this development, I created a small variety of web apps using the cipher, including a translation page for creating your own encoded messages [22], and a way of encrypting websites, having them decrypted on the client's machine and then run [23]. Both sites run the same JavaScript cipher code, which can take ASCII characters and produce a hex string as the encrypted message. Encrypted websites are simply encoded using the translator, stored, then accessed and decrypted using the siteviewer. An example of an encrypted site can be found here [24]

VHDL Implementation

Having the workings of the system confirmed, I began on developing the VHDL version. Initially using an open-source simulator 'GHDL' to develop design one, I transitioned into using the IDE 'Vivado', which had the capabilities to load the designs onto the development board.

Modularisation

Each design and each subtype have the same basic layout. VHDL allows for a modular design, and thus the designs can easily reuse modules from each other.

Let's take design one's unified subtype as an example

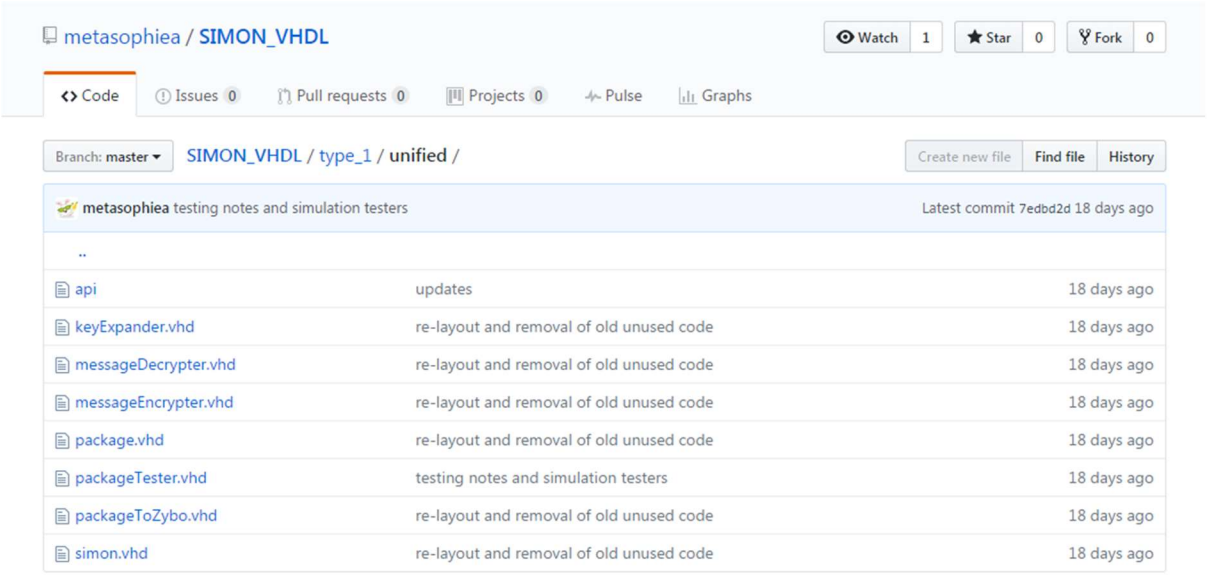

Figure 6 - Screenshot of a Section of the Project Repository

Which can be arranged like so;

Figure 7 - Hierarchy of Files in the Design One Unified Folder

Firstly, the 'simon.vhd' file is the main architecture file. It uses the keyExpander, messageEncryptor and messageDecryptor modules (which are stored in VHDL files of the same name) to perform the actual encryption/decryption of any message. This architecture unfortunately has 517 input/output connections and so needs to have this number reduced to a more reasonable one for connection with an external device.

This is where 'package.vhd' comes in. The architecture here uses the Simon architecture as a component, and reduces the 517 connections down to 24. It is here that the 'api' file becomes useful. This file doesn't contain any code; but instead a complete description of what signals need to be applied to the package architecture's connections to access the various parts of the Simon module's connections.

Following this, the package's connections need to be connected to the development board's real world connections, thus another design file is used with the package design as a component; to map the package architecture's connections to the development board's connections (Zybo being the short name for this board).

The final architecture file here is solely for simulation purposes. It uses the 'packageToZybo' architecture as a component, but produces no connections of its own. It instead utilises the simulation features of the language which allow one to test whether the architecture works properly. This file is removed before the architecture is sent to the development board.

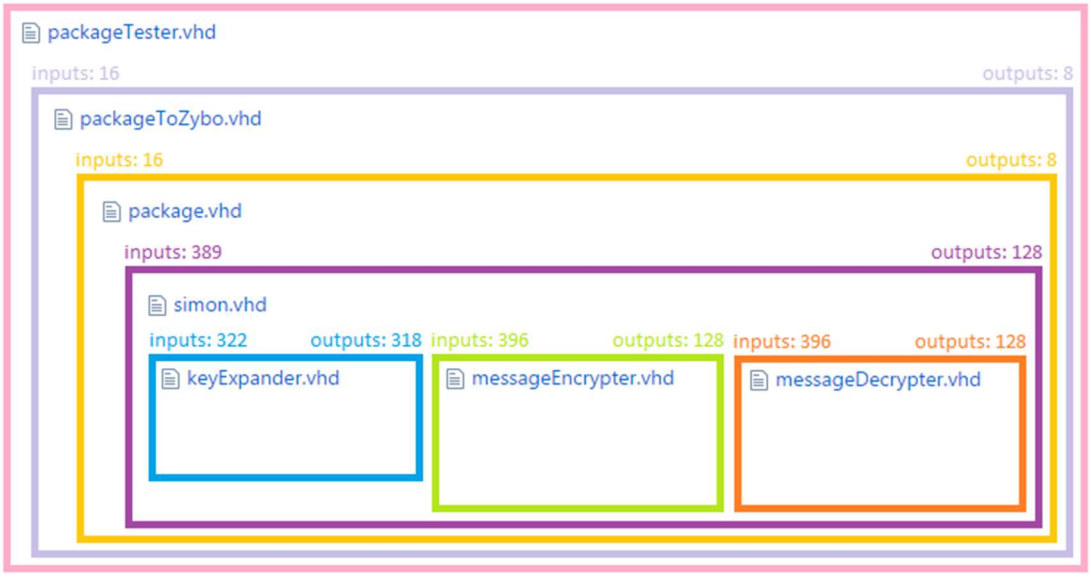

Figure 8 - Visual Layout of How the Modules of Design Type One Unified are Packaged

This description is actually a little misleading, as in reality the Simon architecture uses ~72 instances of the keyExpander, messageEncryptor and messageDecryptor modules, but I've removed them here for aesthetic purposes.

Some designs have additional architecture files. Design type two and three have a file named "completeKeyExpander.vhd", which contains architecture that uses the keyExpander module to completely expand a key in one go. To aid the creation of subtypes other than 'unified', which have different methods hardcoded; a file called "constants.vhd" is used. For these implementations, all the architecture files use named constants to define how long connections should be, the size of register blocks or how many stages should be used. The values for these constants are defined in the "constants.vhd" file. Simply changing which batches of definitions are commented out adjusts the entire architecture of the system to the desired method. This was done to allow simultaneous development of each version, and so that the code repository could be much smaller overall.

Hardware API

As mentioned in the 'Modularisation' section, to communicate with the Simon cipher, one must use the APIs that the packaging architecture provided. As there were 24 connections available, they were neatly split into 3 groups of 8; the Input pins, Output pins and the Control pins.

Input Pins

These connections are used solely for sending data to the system.

Output Pins

These connections are used solely for receiving data from the system.

Control Pins

These connections determine what the previous two groups referred to. For example; in the case of method 1, there are 64 bits in a key. Splitting that up into groups of eight; that's eight groups. To access group 1 (the rightmost eight bits) one must send the hex code '0x51' to the control pins. Now, the eight input pins are connected to group 1, to be edited as desired.

Different codes access different parts of the key, as well as the input message. Sending the codes to access the different parts of the output message, those segments would be sent to the output pins. The codes for accessing this data are the same for all designs, subtypes and versions (though of course, for versions with smaller messages and keys; the higher access codes will return nothing)

For designs that require a clock input; the leftmost Control connection is used for connecting to an external clock (and as a result, only the right 7 bits are used for accessing the message and key segments)

In addition to connecting to the input/output messages and key; one can connect to the 'System Option Bits'. These options are roughly the same for each subtype, and mainly allow one to set which method or mode to use. See the sections ahead for descriptions and details of each.

Design One

unused | unused | unused | method(3) | method(2) | method(1) | method(0) | mode

For this subtype; the rightmost bit controls whether the system is to encrypt or decrypt the message. The four bits to its left determine which method to use, while the rest go unused. For subtypes that had their method set; the connections for setting the method do nothing.

Design Two

unused | unused | unused | method(3) | method(2) | method(1) | method(0) | mode

For this subtype; the rightmost bit controls whether the system is to encrypt or decrypt the message. The four bits to its left determine which method to use while the rest go unused. For subtypes that had their method set; the connections for setting the method do nothing.

Design Three

unused | unused | load | method(3) | method(2) | method(1) | method(0) | mode

For this subtype; the rightmost bit controls whether the system is to encrypt or decrypt the message. The four bits to its left determine which method to use, while the bit to their left activates the load function and the rest go unused. For subtypes that had their method or/and mode set; the connections for setting the method/mode do nothing.

Testing and Results

Here, you can find testing results for each design, subtype and all the versions therein. When testing if each system could perform encryption and decryption correctly; the test was considered a success if the architecture could encrypt and decrypt the provided test messages from the NSA paper [1], matching the required output.

The simulated tests were performed in the manor described in the modularisation section, using a VHDL module testing file, which could provide the message and key to the system, simulate the logic, and display the returned message. The real-world tests were done using the Zybo development board and Raspberry PI, as described in the 'Technical Background' section, with the architecture loaded onto the development board, and a C++ program on the Raspberry Pi communicating the encryption/decryption tasks.

In the following image; one can see this set-up, with the Pi on the left and the Zybo on the right. One can notice the three Pmod connectors in use on the Zybo, which are wired into the ribbon cable attached to the Pi.

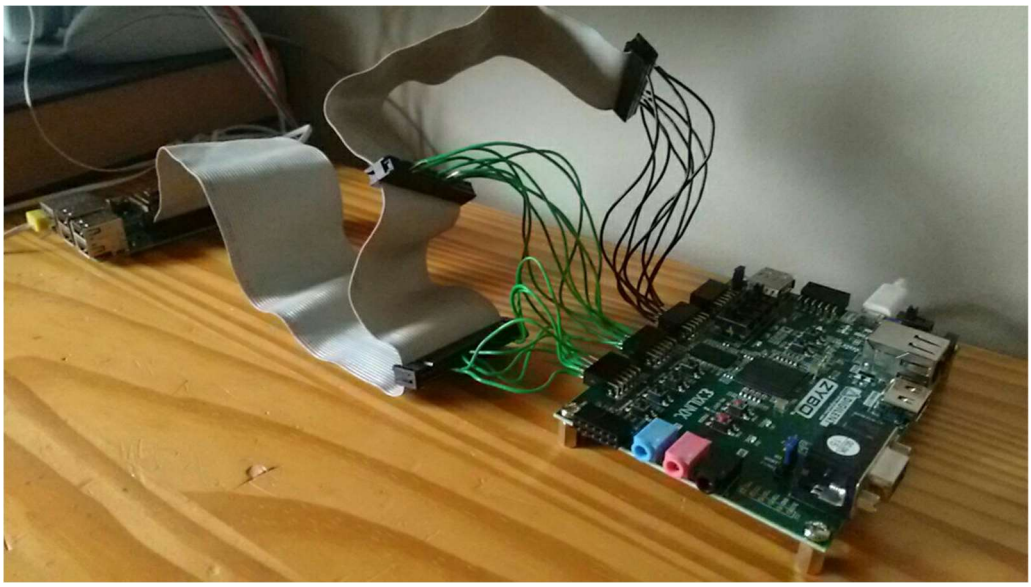

Figure 9 - Photograph of the Test Set-up

The C++ Tester Program

This program is used to test whether the implemented systems can communicate with an external device correctly, with the communication rules set out in the hardware API. It has all the test messages from the NSA paper [1] preloaded, and can send these to the system to gather an encrypted or decrypted result. The user then must decide whether the process has been a success or not. In addition, other messages can be entered into a console interface and sent to the system for further tests.

Thanks to the similarity of the APIs, the C++ tester program is roughly the same for each design, subtype and version. It's split up into six sections;

utilityFunctions

This section consists solely of hexadecimal/binary conversion functions, in both directions.

raspberryPI_api

Here, you can find all the low-level functions that work with the Linux file system to set-up and work with the Raspberry Pi GPIO pins. Reducing the complexity down to a couple of simple "set pin x to y" and "read pin z" functions.

chipAPI

Located here, are the main functions for dealing with the system. Abstracting things further, this section uses the functions from above to create "setControl", "setInput" and "readOutput" functions that can take/return strings of hexadecimal. Building on this, there are functions "writeModeAndMethod", "writeKey" and "readMessage" which can send and receive entire messages and keys to the system.

messageProcessing

This section is more changeable between versions, depending on how interaction must be performed with the system (such as sending clock signals, etc.) Here is a single function, with which one can provide the mode, method, key and message, and receive the complete response message as a string. This function is the main manager of encryption; sending the key, message and options to the system, providing clock and load signals (if necessary) and reading the returned message back.

fileProcessing

Similar to the section above, though a little more experimental; this section contains a function that takes mode, method, key and a file address. The function will then open said file, and encrypt or decrypt its contents, creating an output file in the same directory.

This function is deemed experimental, as it isn't as finished as other sections, and can only take files that have a certain data format. It exhibits however, that with a little work it is entirely possible to have a program that could take any file and encrypt it.

main

The lead section, this code runs a console program with which a user can provide the key and message they wish to encrypt or decrypt. It also contains all the test messages from the NSA paper [1]. Below, is a screenshot of the console output of this testing on a unified subtype

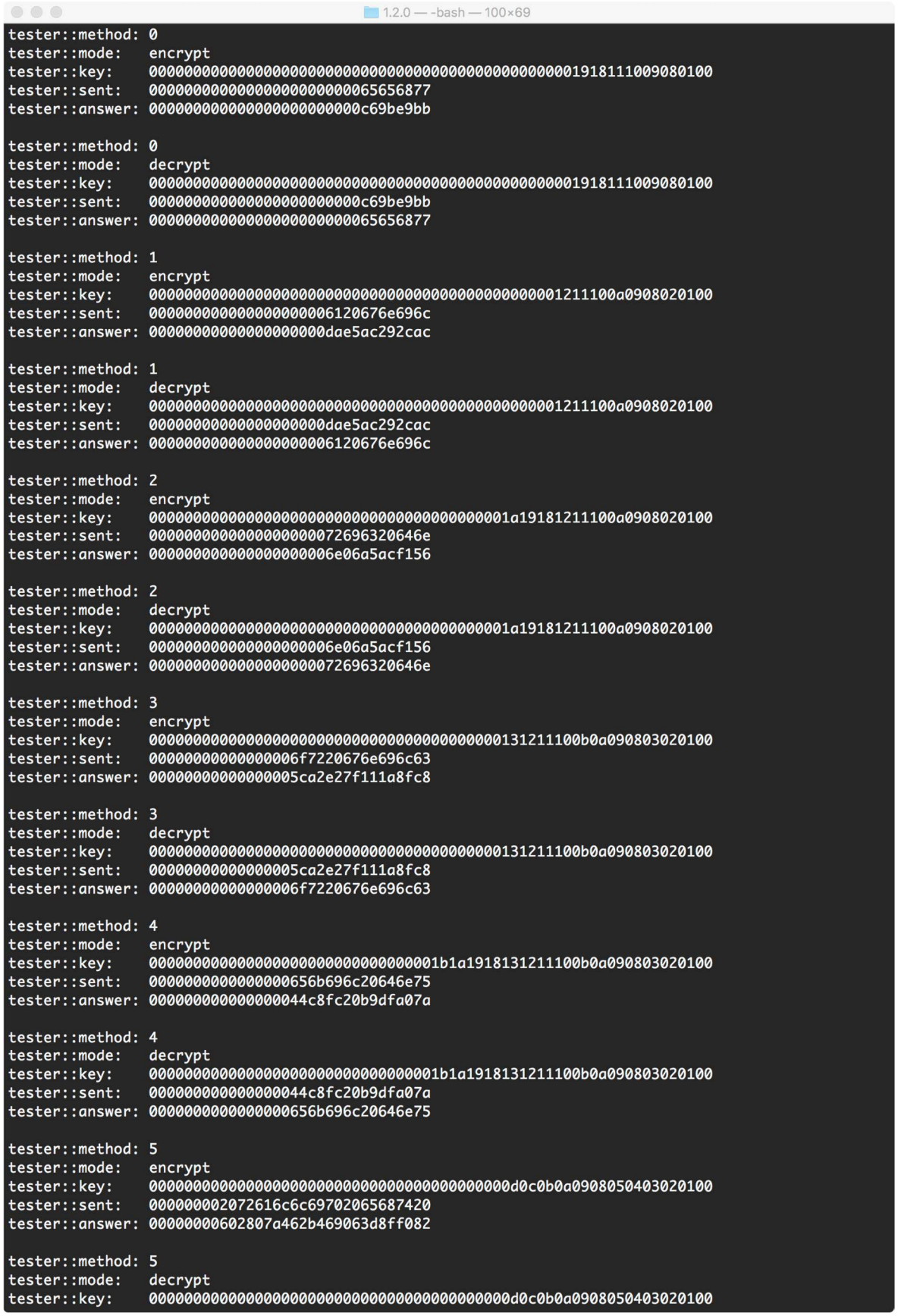

Figure 10 - Screenshot of the Tester Program Running some of the NSA Provided Test Messages

Test Results

On top of determining whether the implemented system could perform correctly or not; the development software also logged the LUT, REG and BUFG usage of each design.

- LUT (Look Up Table) refers to a programmable logic unit, which is used to simulate the desired logic gates.
- REG (Registers) refers to the number of register blocks used in the implementation. You'll notice that even in design one - which shouldn't have any registers - there is a non-zero amount. This is due to the package architecture, which needs to store the message and key data while the external operator uses the control input to access and change it.
- BUFG (Global Buffer), these components are used to distribute high fan-out clock signals through-out a device.

Timing results were unfortunately not captured, as the developed designs operated faster than the C++ testing program could measure.

Type One

Subtype	Version	Synthesised LUT Requirements	Synthesised REG Requirements	Synthesised BUFG Requirements	Successful Simulated Encryption/Decryption	Successful Real World Encryption/Decryption Using the Pi and Zybo
unified	n/a	99436	531	0	Yes	No*
methods	1	1560	97	0	Yes	Yes
methods	2	2336	121	0	Yes	Yes
methods	3	2567	145	0	Yes	Yes
methods	4	3377	161	0	Yes	Yes
methods	5	4168	193	0	Yes	Yes
methods	6	6338	193	0	Yes	Yes
methods	7	6532	241	0	Yes	Yes
methods	8	11004	257	0	Yes	Yes
methods	9	11133	321	0	Yes	Yes
methods	10	13759	385	0	Yes	Yes

*though these versions were synthesised by the development software, their hardware requirements were beyond what the development board could provide

Type Two

Subtype	Version	Synthesised LUT Requirements	Synthesised REG Requirements	Synthesised BUFG Requirements	Successful Simulated Encryption/Decryption	Successful Real World Encryption/Decryption Using the Pi and Zybo
unified	n/a	111581	28978	0	Yes	No*
methods	1	2835	3234	1	Yes	Yes
methods	2	4513	5402	1	Yes	Yes
methods	3	3641	5426	1	Yes	Yes
methods	4	5721	8354	1	Yes	Yes
methods	5	6394	8770	1	Yes	Yes
methods	6	11567	15362	1	Yes	Yes
methods	7	1976	15896	1	Yes	No**
methods	8	23129	26626	1	Yes	No*
methods	9	23266	27238	1	Yes	No*
methods	10	25770	28290	1	Yes	No*

*though these versions were synthesised by the development software, their hardware requirements were beyond what the development board could provide

** though these versions were synthesised by the development software and their hardware requirements were theoretically within what the development board could provide; the development software was unable to produce an actual design that would fit

Type Three

Here, for the modeAndMethods subtype, only the encryption version of the subtype was tested, as the hardware and performance difference between the decryption version of this subtype and the methods subtype were negligible. This is due to the decryption version of the modeAndMethods subtype using the complete key expansion module that the methods subtype has, whereas the encryption version does not. Removing this complete key expansion module is the main design concept around the modeAndMethods subtype.

Subtype	Version	Synthesised LUT Requirements	Synthesised REG Requirements	Synthesised BUFG Requirements	Successful Simulated Encryption/Decryption	Successful Real World Encryption/Decryption Using the Pi and Zybo
unified	n/a	53099	1089	2	Yes	No*
methods	1	1686	280	1	Yes	Yes
methods	2	2887	353	1	Yes	Yes
methods	3	1846	401	1	Yes	Yes
methods	4	1727	464	1	Yes	Yes
methods	5	2385	531	1	Yes	Yes
methods	6	7500	562	1	Yes	Yes
methods	7	7884	658	1	Yes	Yes
methods	8	5647	773	1	Yes	Yes
methods	9	5776	903	1	Yes	Yes
methods	10	7302	1031	1	Yes	Yes
modeAnd Methods	1	123	317	1	Yes	Yes
modeAnd Methods	2	152	381	1	Yes	Yes
modeAnd Methods	3	181	429	1	Yes	Yes
modeAnd Methods	4	192	477	1	Yes	Yes
modeAnd Methods	5	226	541	1	Yes	Yes
modeAnd Methods	6	254	573	1	Yes	Yes
modeAnd Methods	7	283	669	1	Yes	Yes
modeAnd Methods	8	324	733	1	Yes	Yes
modeAnd Methods	9	369	861	1	Yes	Yes
modeAnd Methods	10	444	989	1	Yes	Yes

*though these versions were synthesised by the development software, their hardware requirements were beyond what the development board could provide

Power Usage

Using the power estimation tools available in the development software; estimations were run for one complete encryption cycle. Below you can find details on the power usage of some of the implementations, in regard to the energy usage of internal connections, logic gates and input/output connections. These are known as 'dynamic' usage, where 'static' refers to the energy usage of the development board itself. All usage is recorded in milliwatts (mW)

Implementation	Signals	Logic	I/O	Dynamic	Static	Total
Design Three, subtype methods, version 1	10 (47%)	9 (41%)	3 (12%)	22 (28%)	102 (82%)	124
Design Three, subtype methods, version 10	205 (49%)	208 (50%)	6 (1%)	418 (80%)	106 (20%)	524
Design Three, subtype modeAndMethods, version 1	6 (47%)	3 (25%)	3 (28%)	12 (11%)	102 (89%)	114
Design Three, subtype modeAndMethods, version 10	31 (62%)	13 (25%)	6 (13%)	50 (33%)	102 (67%)	152

Conclusions Drawn

It is clear from these tests; that design three, of the subtype 'modeAndMethods' is logically the smallest of all the architectures, using just 123 logic units. However, design one of the subtype 'methods' has the least register requirements at 97.

As expected; design two has the largest hardware needs, becoming too large to fit on the development board for subtype 'methods' over method 7-8.

All implementations were capable of passing the encryption/decryption in simulation, and all of the architectures that would fit on the development board encrypted/decrypted successfully.

Reduction of Circuit Size

Upon reviewing these results, it is clear that the subtype 'modeAndMethods' of design three, could have its register requirements reduced by rewriting the way the keyExpander module works. Currently, it takes a key and Z value, morphing both and passing both out. However, this morphing of Z is only there to aid the modular design of the system. Instead, one could simply hold the Z value static and have the keyExpander decide which part of it to use; thus saving 62 registers.

Comparison to Other Works

For perspective, the table below compares the hardware utilizations of some of the designs in this work, against a design produced in 2014 by Soheil Feizi, Arash Ahmadi and Ali Netmati [25]. Their design focuses primarily on creating the smallest implementation possible.

Work	IO Connections	LUT Requirements	REG Requirements	BUFG Requirements
S. Feizi - A. Ahmadi - A. Nemati [25]	32	177	182	2
Design One, subtype methods, version 1	24	1560	97	0
Design Two, subtype methods, version 1	24	2835	3234	1
Design Three, subtype modeAndMethods, version 1	24	123	317	1

One can notice that 'Design Three, subtype modeAndMethods, version 1' has the smallest LUT requirements, though the S. Feizi, A. Ahmadi and A. Nemati design has almost half the REG requirements.

Upon review of the designs, it is clear the S. Feizi, A. Ahmadi and A. Nemati design uses a more efficient method of storing the working data; however they are wasteful in their use of multiple encryption and key expansion modules.

Ethics

Wastefulness of the Need

Encryption brings with it a problem of efficiency. So much hardware logic is already spent on giving a device the ability to commutate reliably with the outside world; but in addition to this we have the fear of malicious interference from intentionally hostile sources. This need for security costs us in hardware logic, power consumption and time in encrypting, decrypting, generating keys and passwords, sharing passwords, determining if the keys and messages were sent from the right source, making sure that our messages and keys are getting to the right receiver without interference, etc.

This report does not claim that all encryption should be done away with; though it is a shame that so much of networking has to be devoted to these processes; which is time, power and hardware that could be better spent on communicating more of the actual message. Ultimately, the primary flaw in all cryptographic systems is the need for the receiver to be able to decode the message. This single and unavoidable issue means that every code is breakable by a hacker, thus no cipher can be completely secure. In this way, security is more of an ongoing experiment than a logical possibility.

Use of Encryption in General

Back in March of 2016 [26]; the FBI requested that Apple assist in unlocking an iPhone owned by a criminal, in order to access the information inside. The iPhone has a feature that disables the device after ten sequential, incorrect passwords have been entered. To complete this task; Apple would have had to create a program that could break their own encryption and security systems. This was a task that Apple's CEO at the time – Tim Cook – flatly denied to carry out as to do so would be like creating "a master key, capable of opening hundreds of millions of locks", causing understandable controversy between those who believed that law enforcement should be given every assistance in order to carry out their job; and those who thought that personal privacy should not be given away so easy.

Revelations from the leaked documents by Edward Snowden [27], show that such privacy previously thought inherent, is in fact in question; and recent decisions by the American government [28] are showing that the official stance on this, is leading people towards less privacy. And in turn; forcing companies to create holes in their systems through which legal police action can take place, opens those systems to unlawful attack though cracks in the security of these forced insecurities.

One can argue that people have nothing to fear of this mass surveillance, if they've done nothing wrong. This is ostensibly true, and in the case of unlocking a criminal's home computer, phone or even wall safe, very strong arguments can be made to justify gathering that information. But to have such widespread and extensive access to not only every individuals' data, but the "big-data" that goes along it, can have stranger affects rarely talked about. For example; knowing the demeanour of a person, who their friends are, what products they buy, what content they consume, etc.; one can build a very accurate picture of what stimulus will affect them the most, potentially inciting political unrest in some. Beyond this, it is the opinion of the author that private communication between a person and their parent should by right be private. It is an unspoken understanding.

To remedy this, we all should be more conscious of our own security. Seeking out secure applications and devices to communicate with other people, or browse the internet with. Pressuring companies to deliver encryption and secure content as standard, and voting towards political representatives whom respect the individual's right to privacy are all a start.

In the end; Apple didn't help the FBI, but the FBI did eventually break into the iPhone by themselves. The NSA continues to survey the American people, and there is talk of further bills being drawn to remove restrictions on accessing private information. If things continue, encryption will become more and more disparaged by governments, leaving the door open to hackers stealing private information, such as bank details and identities. There can never be a guarantee of only lawful surveillance.

Conclusions

In conclusion; I believe I have successfully implemented the Simon block ciphers in VHDL, surpassing the original project specification by creating two additional designs, as well as creating an implementation paper for others to use in future.

For all developed code (VHDL and C++) please see the git for this project [19]

Improvements and Future Research

Upon completion of the practical aspects of this project; a number of future research avenues have become clear

Circuit Size Reduction

As mentioned before, certain aspects of the design are a little wasteful. For example, the needless storage of the Z value in the subtype 'modeAndMethods' of design three. It is also worth researching a more efficient complete key expander that wouldn't use the key expander module, or finding a way to remove its need entirely. Perhaps a reverse to the key expander module could be created, which should reduce the register requirements of design two and three substantially.

Another circuit reducing improvement would be to replace the inner workings of the complete key expander, found in the design of type two and three. Currently, the complete key expander contains lots of key expander modules; but one could replace these with a single module which would be reused, akin to the overall concept of design three. In this way, the hardware requirements could be greatly reduced for both designs. In addition; one could fully expand the key for a subsequent message, while the current message was being encrypted/decrypted, saving on the overall processing time.

Additional Subtypes

The creation of a 'modeAndMethods' encryption only subtype for the type two 'Register Transfer Level' design, would allow the complete key expander to be removed; placing the key expansion modules alongside the individual encryption modules. This would remove the wait time for a new message, allowing for faster processing of messages.

Better C++ Program for Encrypting Files

As mentioned previously, the C++ file encryptor/decryptor is still very much in a development phase, lacking the ability to work with files outside of a particular format. Beyond that, one could also consider more direct communication with a computer's GPIO pins; the Linux file system method is easy to understand, but writing a driver module that would bypass this could raise efficiency.

Ultimately, a program could be written, that could take any file and a password, and encrypt that file using a hardware cipher.

Evolutionary Circuit Design

Much research has been done into the fields of machine learning and neural networks. Another field similar to these is known as Evolvable Hardware, which concerns the use of artificial intelligence to create electronic circuitry automatically.

Conceivably, one could design a program that would use a design described in this report, to produce random encrypted messages; and use these messages to train a neural network to perform

the same work, eventually creating an electronic circuit that would perform all the same tasks as the original system, but with much greater efficiency and at a lower hardware cost.

Suggested Usage of Designs

Broadcast and Response Layout

It is my opinion, that for device network layouts consisting of many satellite devices and a central communication node; one should fit the satellite devices with a Simon module of type 3 (crypto-processor) encryption only. Then use Simon module type 2 (Register Transfer Level) for the central node, running decryption only.

The encryption only version of type 3 has by far the smallest footprint of any of the designs, and in turn the lowest power consumption. Though one must accept a delay in the encryption time; I believe this layout of many devices communicating with a central node, allows for the satellite devices' encryption time to be slightly slower.

Type 2 has a much larger footprint, but would allow the system to decrypt batches of messages faster (allowing for the initial key expansion processing time)

In action; the satellite devices would encrypt any message it wanted to send to the central node, where it would be decrypted. For returned messages; the central node would decrypt any message it wanted to send to a satellite, where the satellite can then encrypt it, and in effect decrypt the message.

In this way, two-way communication can be achieved for the lowest hardware costs on the satellite, and with a key common to all the satellites; the highest throughput for the central node. Also using a common key for all devices, would allow the central node to both 'encrypt' messages to send to satellite devices while decrypting received messages.

A disadvantage of this design is that no two satellites could communicate with each other. It is also important in this situation, to be mindful of how this common key is created. Simply having a hardwired key in all devices is not very secure, so one should investigate using time-dependent key generation.

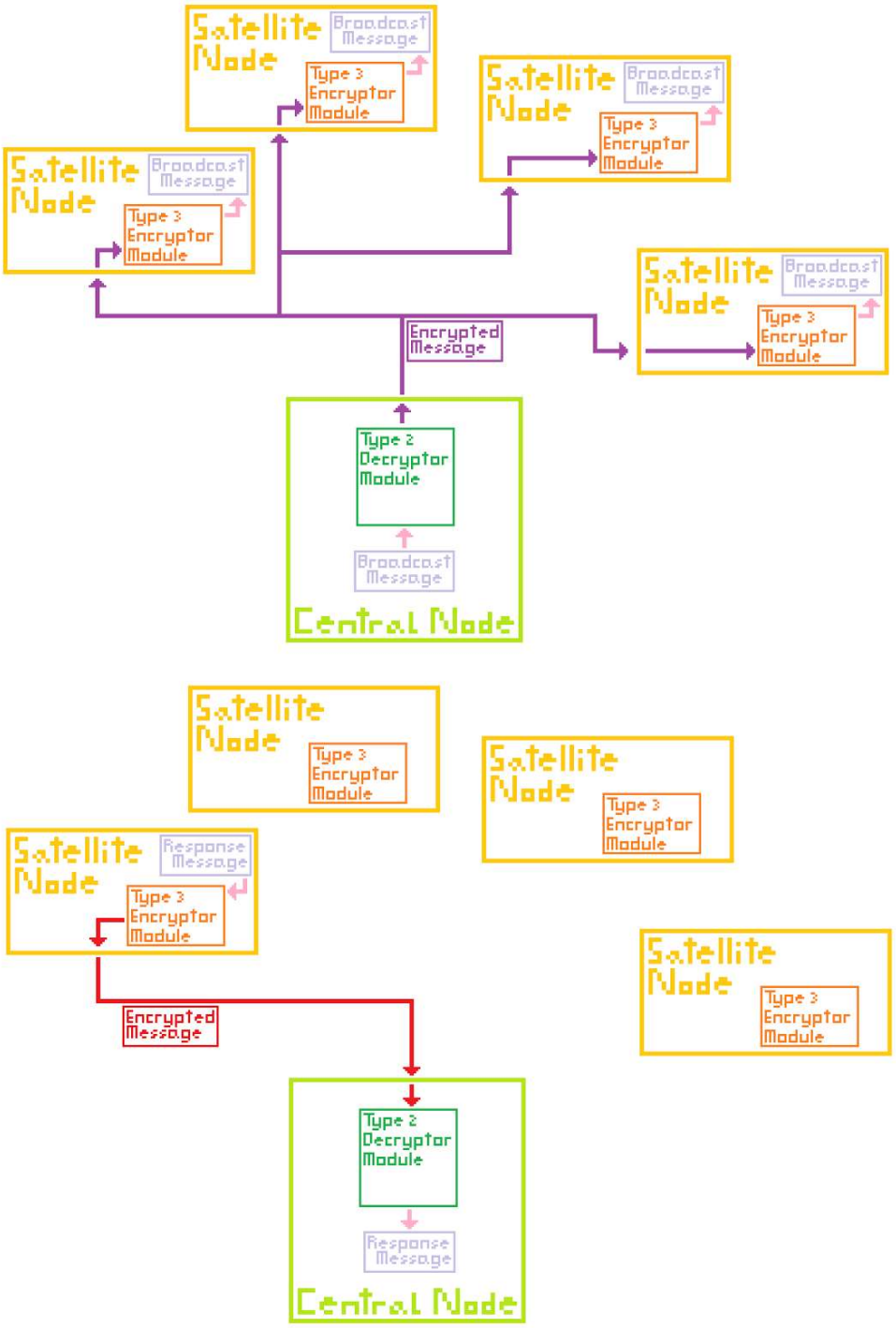

Figure 11 - Broadcast and Response Layout Example

Mesh Layout

For a mesh layout, where all nodes are of a similar size and must communicate with each other; I would recommend type 2, and then type 3 for very size constrained devices. Both types have the key expansion delay, though for long messages between nodes, this becomes negligible with type 2. Type 3's lead advantage is its hardware size, so it is recommended to use type 2 here if possible.

Encryption Only

Another potential usage of the designs presented here, could be for the creation of “Cipher Feedback” or “Output Feedback” encryption. These methods of encrypting only need the encryption portion of the cipher, though would take longer to encrypt or decrypt a message. As such they would allow devices that only implemented the encryption circuitry to communicate with one another, at the cost of communication speed.

Cipher Feedback

Cipher Feedback uses an “initialization vector”, encrypts it and XORs this with the message. For stronger security, this XOR’ed message can be encrypted again and XOR’d again with the message multiple times. Decryption is the very same system, expect of course the encrypted message is provided in place of the plain message.

Below, is a diagram of this system, using the smallest method of the cipher. One could save implementation size, by reusing the same encryption and XOR circuitry, akin to the crypto-processor.

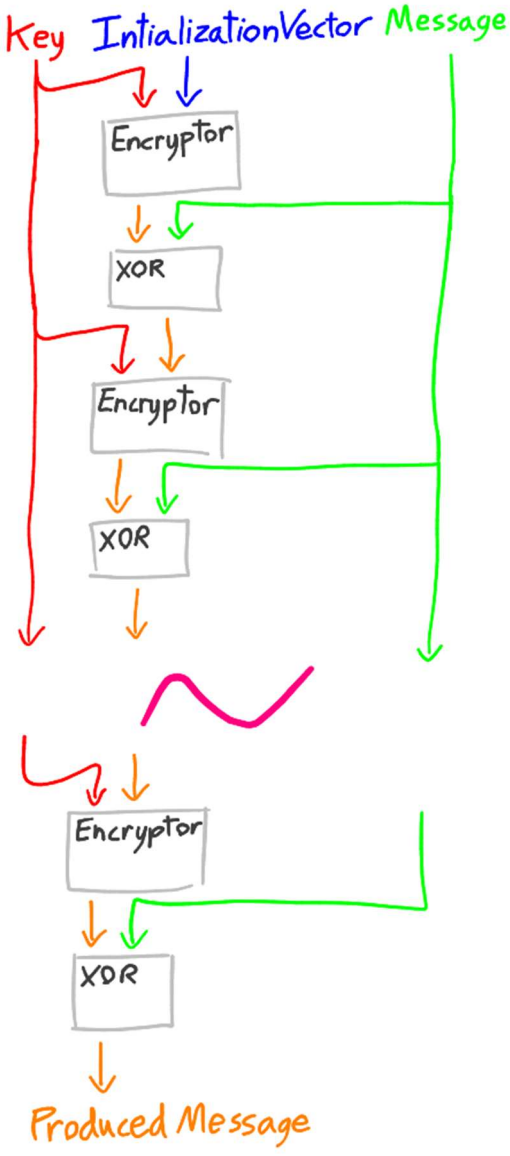

Figure 12 - Example Cipher Feedback Layout

Output Feedback

Output feedback encryption is very similar to cipher feedback, but after the initialization vector is encrypted and XOR'd with the message; the vector is encrypted again and the result is XOR'd with the next message to be encrypted. Similar to above, decryption is the very same system, expect the encrypted messages are provided in place of the plain messages.

Below, is a diagram of this system, using the smallest method of the cipher. One could save implementation size, by reusing the same encryption and XOR circuitry, akin to the crypto-processor.

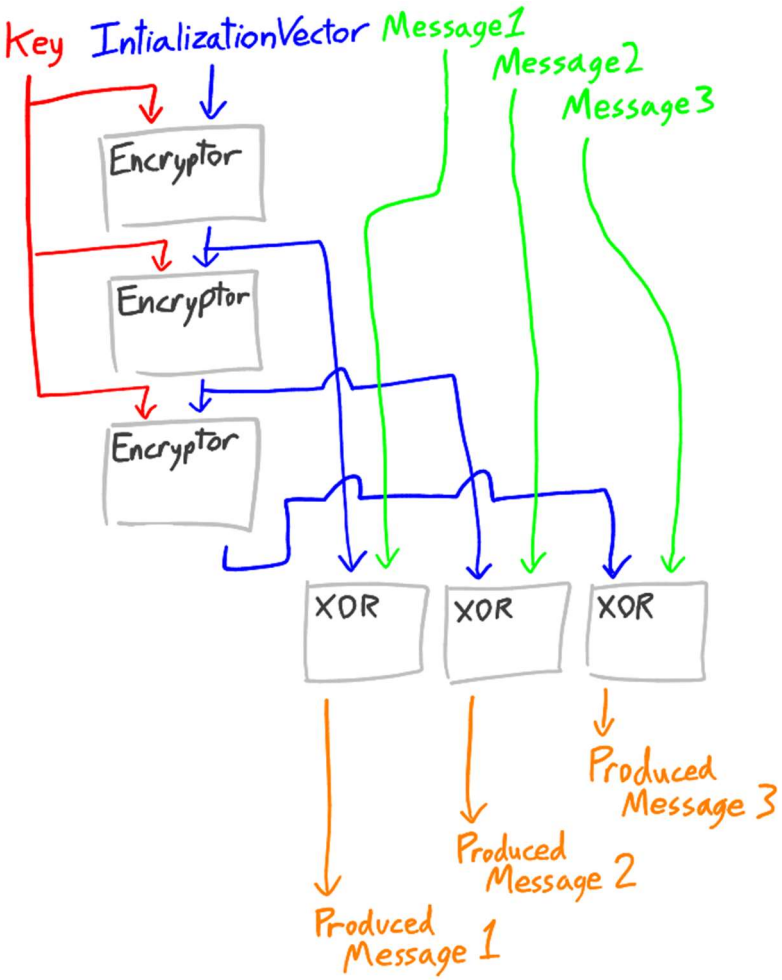

Figure 13 - Example Output Feedback Layout

It is conceivable, that one could implement a system that used both these techniques together, to boost security.

Security

As this is a system designed to be used in the world of secret communication, it is reasonable to assume that someone will attempt to break the encryption used by the cipher. Though the implementations developed in this project can't inherently be hacked, it's a potential concern for any developer who would want to use these designs in their device. Once a hacker knew that a device used the Simon cipher in its communications, and the keys used to encode information; it would be only a matter of time before the messages sent and received could be decoded, allowing for imposter devices to communicate with it.

It is important therefore, that a developer investigate advanced key generation methods, such as time-dependent generation, or sequential counter mode generation with a seed determined while in contact with the device, or/and utilising a Public Key Infrastructure type system or Key Distribution Centre, for the remote sharing and administration of seeds/keys.

Beyond this, a developer must also be wary of keeping what cipher they are using secret in the first place. Obviously, one could hide the cipher in unmarked silicon integrated with the larger system; but side-channel attacks, such as monitoring the timing of sections of a device, its power consumption and electromagnetic leaks during situations where communication with an external entity is taking place - or any such situation where the cipher might come into use – could reveal to a hacker what encryption is being used, or/and key generation details. In this way, certain characteristics of the designs in this project can become a risk.

For example; in designs two and three, in initiating a new message, one must wait a period of time for the complete key to be expanded, before message encryption/decryption can begin. This initial waiting time is controlled by the external clock; thus a hacker could monitor the timing information and notice this pause in clock signal. Also, once a key has been expanded, subsequent messages using the same key do not need to repeat this expansion. Again, a hacker could use this clock information to determine that the messages being transmitted or received by the device are using the same key.

Ultimately, it is up to the developer as to how cautious the system should be. They could slow the clock down so that a clock step is the same whether the key needs to be expanded or not. They would also need to vary the key used for each message, never using a single key for batches of messages. This constant processing would mitigate changes in power consumption or electromagnetic leaks.

Bibliography

[1] R. Beaulieu, D. Shors, J. Smith, B. Weeks, S. Treatman-Clark and L. Wingers, *The Simon and Speck Families of Lightweight Block Ciphers,* Fort Meade: NSA, 2013.

[2] E. Chabrow, "Encrypting the Internet of Things," 2016. [Online]. Available: http://www.bankinfosecurity.com/encrypting-internet-things-a-9382. [Accessed 22 11 2016].

[3] Doulos, "A Brief History of VHDL," 2014. [Online]. Available: https://www.doulos.com/knowhow/vhdl_designers_guide/a_brief_history_of_vhdl/. [Accessed 22 11 2016].

[4] "1076-1987 - IEEE Standard VHDL Language Reference Manual," 1988. [Online]. Available: http://ieeexplore.ieee.org/document/26487. [Accessed 22 11 2016].

[5] "The Computer Engineering Handbook," Boca Raton, CRC Press, 2002, pp. 12-6.

[6] "The Ada Programming Language," [Online]. Available: http://groups.engin.umd.umich.edu/CIS/course.des/cis400/ada/ada.html. [Accessed 22 11 2016].

[7] Xilinx, "Vivado Design Suite," Xilinx, [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html. [Accessed 22 11 2016].

[8] Mentor, "ModelSim PE Student Edition," Mentor, [Online]. Available: https://www.mentor.com/company/higher_ed/modelsim-student-edition. [Accessed 22 11 2016].

[9] Digikey, "ZYBO ZynqTM-7000 Development Board," Digikey, [Online]. Available: https://www.digikey.ie/en/product-highlight/d/digilent/zybo-zynq-7000-development-board?WT.srch=1&mkwid=sil5IkmbQ&pcrid=76482552558&pkw=_cat%3Adigikey.ie&pmt=b&pdv=c. [Accessed 24 11 2016].

[10] Raspberry PI Foundation, "Raspberry PI 3 Model B," Raspberry PI Foundation, [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Accessed 3 4 2017].

[11] "About Raspbian," [Online]. Available: http://raspbian.org/RaspbianAbout. [Accessed 3 4 2017].

[12] Software in the Public Interest, Inc., "Debian "wheezy" Release Information," Software in the Public Interest, Inc., [Online]. Available: https://www.debian.org/releases/wheezy/. [Accessed 3 4 2017].

[13] Digilent, "Pmod Modules," Digilent, [Online]. Available: http://store.digilentinc.com/pmod-

modules/. [Accessed 3 4 2017].

[14] International Organization for Standardization, “Information Technology -- Trusted Platform Module -- Part 1: Overview,” International Organization for Standardization, [Online]. Available: https://www.iso.org/standard/50970.htm. [Accessed 3 4 2017].

[15] G. Song, “Simon & Speck block cipher implementation open source code in C,” [Online]. Available: https://github.com/GSongHashrate/SimonSpeck. [Accessed 24 11 2016].

[16] C. McCoy, “Implementations of the Simon and Speck Block Ciphers,” [Online]. Available: https://github.com/inmcm/Simon_Speck_Ciphers. [Accessed 24 11 2016].

[17] T. Whittington, “The Simon family of Block Ciphers,” [Online]. Available: https://github.com/timw/bc-java/blob/feature/simon-speck/core/src/main/java/org/bouncycastle/crypto/engines/SimonEngine.java. [Accessed 24 11 2016].

[18] B. Walsh, “The Simon Cipher,” 2017. [Online]. Available: http://metasophiea.com/projects/simon/webPaper. [Accessed 3 4 2017].

[19] B. Walsh, “SIMON_VHDL Github Repository,” [Online]. Available: https://github.com/metasophiea/SIMON_VHDL. [Accessed 3 4 2017].

[20] B. Walsh. [Online]. Available: http://metasophiea.com/lib/js/SIMON/pureSystem/SIMON.js. [Accessed 3 4 2017].

[21] B. Walsh, “JavaScript Simon Cipher Test Page,” [Online]. Available: http://metasophiea.com/lib/js/SIMON/test.html. [Accessed 3 4 2017].

[22] B. Walsh, “Simon Translator,” [Online]. Available: http://metasophiea.com/apps/encryption/SIMON/translater/. [Accessed 3 4 2017].

[23] B. Walsh, “Encrypted Site Viewer,” [Online]. Available: http://metasophiea.com/apps/encryption/SIMON/siteviewer/. [Accessed 3 4 2017].

[24] B. Walsh, “Encrypted Light Cycles Game,” [Online]. Available: http://metasophiea.com/lib/js/SIMON/encryptedSites/encrypted_LightCycles.crypt. [Accessed 3 4 2017].

[25] S. Feizi, A. Ahmadi and A. Nemati, “A Hardware Implementation of Simon Cryptography Algorithm,” IEEE, 2014.

[26] A. Kharpal, “Apple vs FBI: All you need to know,” CNBC, 29 3 2016. [Online]. Available: http://www.cnbc.com/2016/03/29/apple-vs-fbi-all-you-need-to-know.html. [Accessed 04 04 2017].

[27] “Edward Snowden: Leaks that exposed US spy programme,” BBC, 17 1 2014. [Online]. Available: http://www.bbc.com/news/world-us-canada-23123964. [Accessed 4 4 2017].

[28] K. Reilly, “President Trump Signs Bill Overturning Internet Privacy Protections,” Time, 3 4 2017. [Online]. Available: http://time.com/4724128/donald-trump-internet-history-isp-privacy-browser-history/. [Accessed 4 4 2017].

Appendix

Key Expansion Example

Using method 1 for this example, our inputs will be:

keySegments: 4
selected Z: 0
stage: 1st
keyIn (0 1 2 3): 0001100100011000 0001000100010000 0000100100001000 0000000100000000
Z: 11111010001001010110000111001101111101000100101011000011100110

1. Triple right rotate key segment 0
   newKeySegment = rightRotate3(0001100100011000)
   = 0000001100100011
2. There are four segments here, so XOR newKeySegment with key segment 2
   newKeySegment = 0000001100100011 XOR 0000100100001000
   = 0000101000101011
3. XOR newKeySegment with one-bit right-rotated version of itself
   newKeySegment = 0000001100100011 XOR 0000100100001000
   = 1000111100111110
4. XOR newKeySegment with keySegment (keySegments-1), which here is segment 3
   newKey = 1000111100111110 XOR 0000000100000000
   = 1000111000111110
5. As this is the first stage, XOR Z's 1st bit (most left) with newKeySegment 's rightmost bit
   newKey = 1000111000111110
   Z's first bit: 1
   = 1000111000111110 XOR 0000000000000001
   = 1000111000111111
6. XOR newKeySegment with the value: -4
   newKey = 1000111000111111 XOR 1111111111111100
   = 0111000111000011

The key produced from this module is
0111000111000011 0001100100011000 0001000100010000 0000100100001000

(Notice how the leftmost three segments here, are the rightmost three segments from the input)

In this example (as in the designs produced in this report) to select the correct Z bit, the Z value is left rotated by one bit inside each key expander module and this Z value is passed to the subsequent key expander module. In this way, the leftmost bit is always the correct bit.

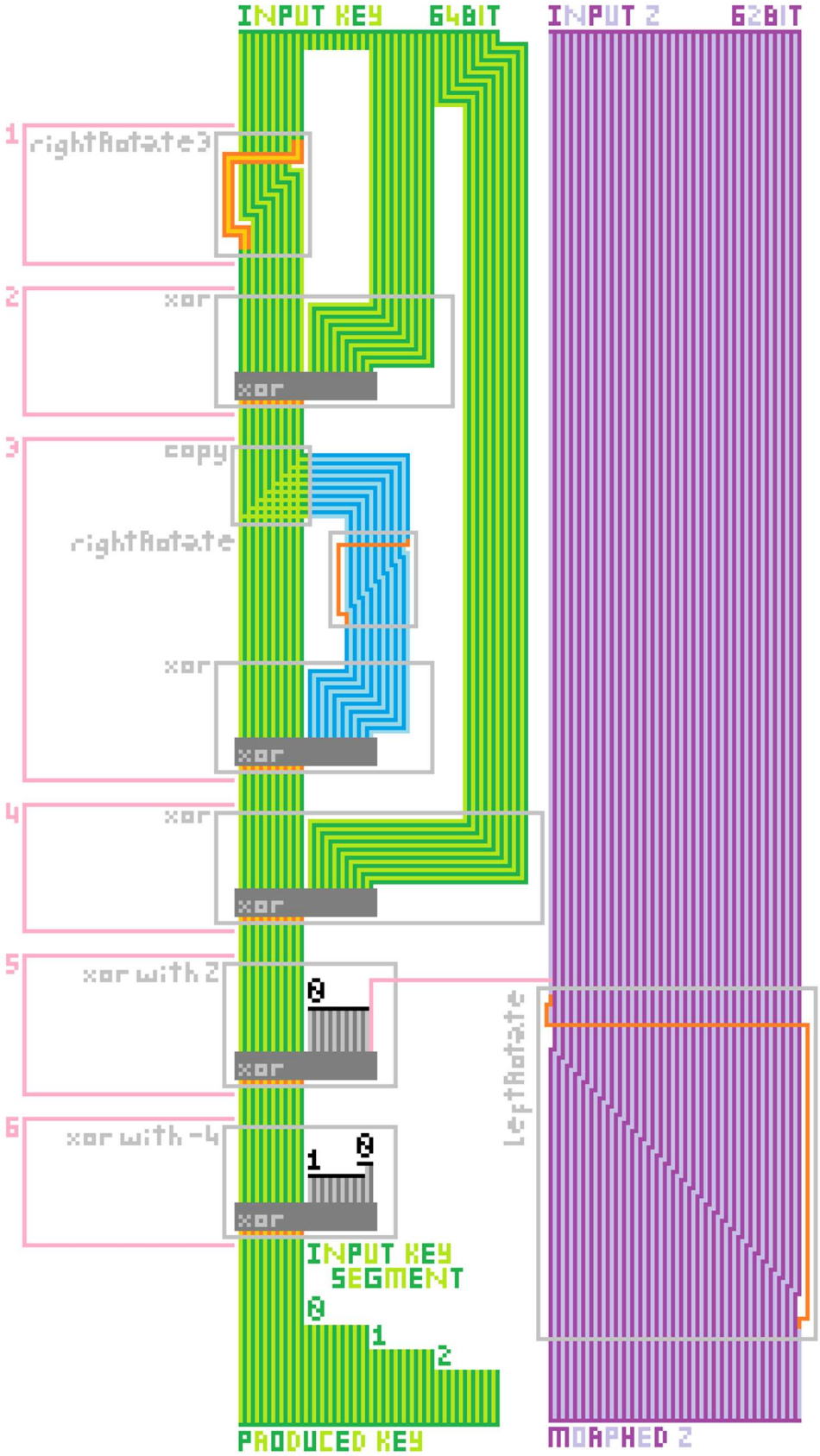

Figure 14 - Key Expander Example for Method One

Message Encryption Example

Using method 1 for this example, our inputs will be:

messageIn (0 1):0110010101100101 0110100001110111
keyIn (0 1 2 3): 0001100100011000 0001000100010000 0000100100001000 0000000100000000
keySegmentCount: 4

1. Right rotate message segment 0 once and eight times, then AND these two together 2
   newMessageSegment = leftRotate(0110010101100101)once
   AND leftRotate(0110010101100101)eightTimes
   = 1100101011001010 AND 0110010101100101
   = 0100000001000000
2. XOR newMessageSegment with message segment 1
   newMessageSegment = 0100000001000000 XOR 0110100001110111
   = 0010100000110111
3. XOR newMessageSegment with a twice left rotated message segment 0
   newMessageSegment = 0010100000110111
   XOR leftRotate(0110010101100101)twice
   = 0010100000110111 XOR 1001010110010101
   = 1011110110100010
4. XOR newMessageSegment with the last key segment (key segment (keySegmentCount-1))
   newMessageSegment = 1011110110100010 XOR 0000000100000000
   = 1011110010100010

The message produced from the module is

1011110010100010 0110010101100101

(Notice how the rightmost segment here, is the leftmost segment from the input)

Figure 15 - Message Encryptor Example for Method One

Message Decryption Example

Using method 1 for this example, our inputs will be:

messageIn (0 1):1100011010011011 1110100110111011
keyIn (0 1 2 3): 0001100100011000 0001000100010000 0000100100001000 0000000100000000
keySegmentCount: 4

1. Right rotate message segment 0 once and eight times, then AND these two together 2

   newMessageSegment = leftRotate(1110100110111011)once
   AND leftRotate(1110100110111011)eightTimes
   = 1101001101110111 AND 1011101111101001
   = 1001001101100001

2. XOR newMessageSegment with message segment 1

   newMessageSegment = 1001001101100001 XOR 1100011010011011
   = 0101010111111010

3. XOR newMessageSegment with a twice left rotated message segment 0

   newMessageSegment = 0101010111111010
   XOR leftRotate(1110100110111011)twice
   = 0101010111111010 XOR 1010011011101111
   = 1111001100010101

4. XOR newMessageSegment with the last key segment (key segment (keySegmentCount-1))

   newMessageSegment = 1111001100010101 XOR 0000000100000000
   = 1111001000010101

The message produced from the module is

1110100110111011 1111001000010101

(Notice how the leftmost segment here, is the rightmost segment from the input)

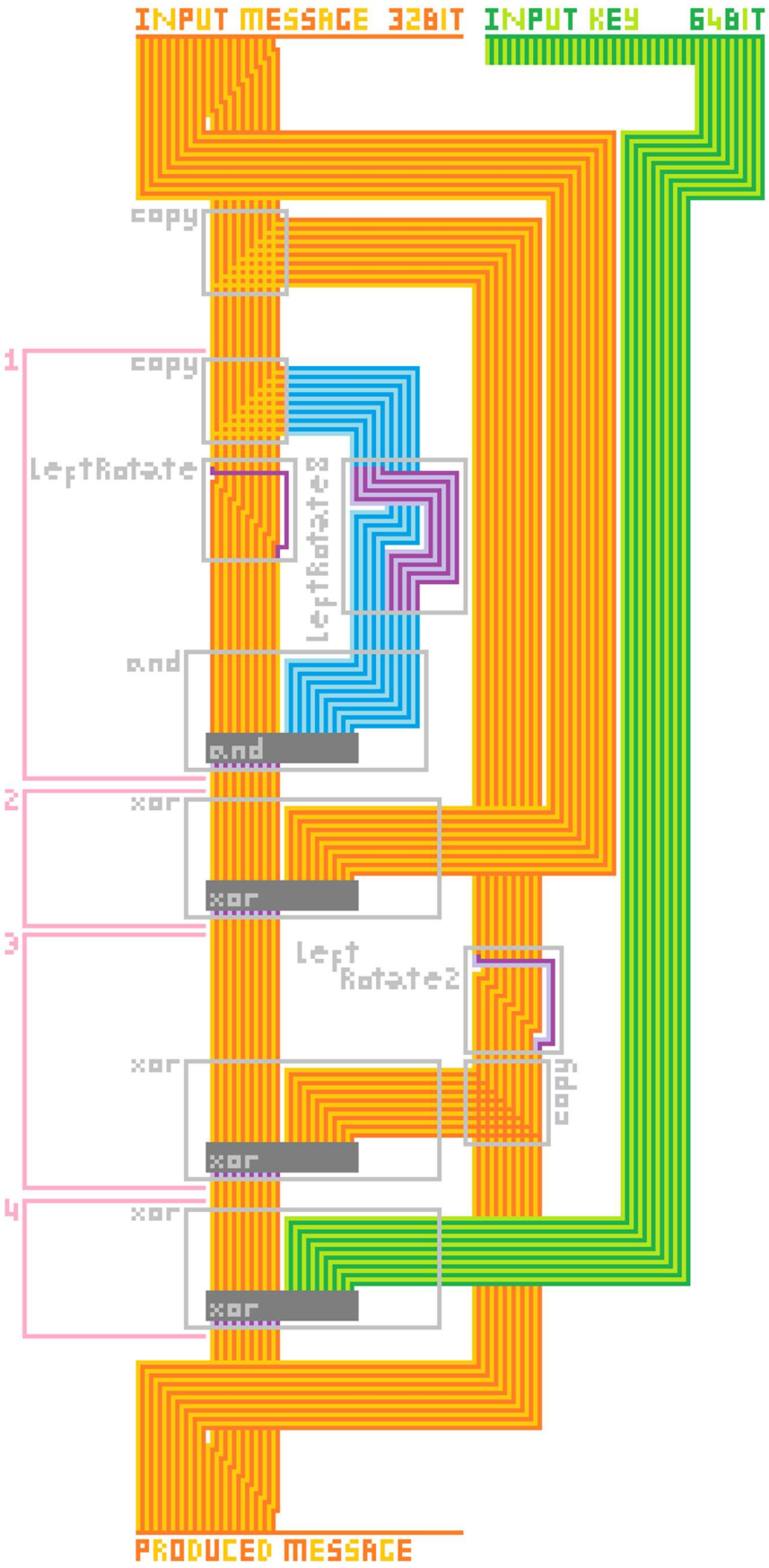

Figure 16 - Message Decryptor Example for Method One

Full Encryption Example

Provided Key: 0x1918111009080100
Message to Encrypt: 0x65656877
Method: 1
Number of stages: 32

Key is split into four segments: 0x1918, 0x1110, 0x0908 and 0x0100
Message is split in two: 0x6565 and 0x6877

Stage	Key Expansion	Key Segment Used	Message Passed to Encryptor Module	Message Out of Encryptor Module
1	0x1918111009080100	0x0100	0x65656877	0xbca26565
2	0x71c3191811100908	0x0908	0xbca26565	0xbee3bca2
3	0xb64971c319181110	0x1110	0xbee3bca2	0x37babee3
4	0x56d4b64971c31918	0x1918	0x37babee3	0x532737ba
5	0xe07056d4b64971c3	0x71c3	0x532737ba	0x2ca65327
6	0xf15ae07056d4b649	0xb649	0x2ca65327	0x57fa2ca6
7	0xc535f15ae07056d4	0x56d4	0x57fa2ca6	0x8fcf57fa
8	0xdd94c535f15ae070	0xe070	0x8fcf57fa	0x873b8fcf
9	0x4010dd94c535f15a	0xf15a	0x873b8fcf	0x687c873b
10	0x250a4010dd94c535	0xc535	0x687c873b	0xb397687c
11	0x6f66250a4010dd94	0xdd94	0xb397687c	0x7c95b397
12	0xe96b6f66250a4010	0x4010	0x7c95b397	0x90fa7c95
13	0x4bd8e96b6f66250a	0x250a	0x90fa7c95	0x3ae590fa
14	0x0fe54bd8e96b6f66	0x6f66	0x3ae590fa	0x71023ae5
15	0x7c470fe54bd8e96b	0xe96b	0x71023ae5	0x15877102
16	0xe0ef7c470fe54bd8	0x4bd8	0x15877102	0x6fc21587
17	0x3e21e0ef7c470fe5	0x0fe5	0x6fc21587	0x676f6fc2
18	0x065b3e21e0ef7c47	0x7c47	0x676f6fc2	0xc07e676f
19	0x438c065b3e21e0ef	0xe0ef	0xc07e676f	0x86bbc07e
20	0xf26a438c065b3e21	0x3e21	0x86bbc07e	0xedb786bb
21	0xb5c0f26a438c065b	0x065b	0xedb786bb	0xa552edb7
22	0x8609b5c0f26a438c	0x438c	0xa552edb7	0x79d4a552
23	0x9f8e8609b5c0f26a	0xf26a	0x79d4a552	0x604179d4
24	0xd8bf9f8e8609b5c0	0xb5c0	0x604179d4	0x0d116041
25	0x09acd8bf9f8e8609	0x8609	0x0d116041	0xc20c0d11
26	0xe81209acd8bf9f8e	0x9f8e	0xc20c0d11	0x9eacc20c
27	0x2710e81209acd8bf	0xd8bf	0x9eacc20c	0x4c199eac
28	0x2caa2710e81209ac	0x09ac	0x4c199eac	0xbf654c19
29	0x8d142caa2710e812	0xe812	0xbf654c19	0x3d16bf65
30	0xfa048d142caa2710	0x2710	0x3d16bf65	0x7e013d16
31	0x32f2fa048d142caa	0x2caa	0x7e013d16	0xe9bb7e01
32	0x7db932f2fa048d14	0x8d14	0xe9bb7e01	0xc69be9bb

From this we see the encrypted message is: 0xc69be9bb, which matches the NSA paper's [1] expected outcome.

Full Decryption Example

Provided Key: 0x1918111009080100
Message to Decrypt: 0xc69be9bb
Method: 1
Number of stages: 32

Key is split into four segments: 0x1918, 0x1110, 0x0908 and 0x0100
Message is split in two: 0xc69b and 0xe9bb

Stage	Key Expansion	Key Segment Used	Message Passed to Decryptor Module	Message Out of Decryptor Module
1	0x1918111009080100	0x8d14	0xc69be9bb	0xe9bb7e01
2	0x71c3191811100908	0x2caa	0xe9bb7e01	0x7e013d16
3	0xb64971c319181110	0x2710	0x7e013d16	0x3d16bf65
4	0x56d4b64971c31918	0xe812	0x3d16bf65	0xbf654c19
5	0xe07056d4b64971c3	0x09ac	0xbf654c19	0x4c199eac
6	0xf15ae07056d4b649	0xd8bf	0x4c199eac	0x9eacc20c
7	0xc535f15ae07056d4	0x9f8e	0x9eacc20c	0xc20c0d11
8	0xdd94c535f15ae070	0x8609	0xc20c0d11	0x0d116041
9	0x4010dd94c535f15a	0xb5c0	0x0d116041	0x604179d4
10	0x250a4010dd94c535	0xf26a	0x604179d4	0x79d4a552
11	0x6f66250a4010dd94	0x438c	0x79d4a552	0xa552edb7
12	0xe96b6f66250a4010	0x065b	0xa552edb7	0xedb786bb
13	0x4bd8e96b6f66250a	0x3e21	0xedb786bb	0x86bbc07e
14	0x0fe54bd8e96b6f66	0xe0ef	0x86bbc07e	0xc07e676f
15	0x7c470fe54bd8e96b	0x7c47	0xc07e676f	0x676f6fc2
16	0xe0ef7c470fe54bd8	0x0fe5	0x676f6fc2	0x6fc21587
17	0x3e21e0ef7c470fe5	0x4bd8	0x6fc21587	0x15877102
18	0x065b3e21e0ef7c47	0xe96b	0x15877102	0x71023ae5
19	0x438c065b3e21e0ef	0x6f66	0x71023ae5	0x3ae590fa
20	0xf26a438c065b3e21	0x250a	0x3ae590fa	0x90fa7c95
21	0xb5c0f26a438c065b	0x4010	0x90fa7c95	0x7c95b397
22	0x8609b5c0f26a438c	0xdd94	0x7c95b397	0xb397687c
23	0x9f8e8609b5c0f26a	0xc535	0xb397687c	0x687c873b
24	0xd8bf9f8e8609b5c0	0xf15a	0x687c873b	0x873b8fcf
25	0x09acd8bf9f8e8609	0xe070	0x873b8fcf	0x8fcf57fa
26	0xe81209acd8bf9f8e	0x56d4	0x8fcf57fa	0x57fa2ca6
27	0x2710e81209acd8bf	0xb649	0x57fa2ca6	0x2ca65327
28	0x2caa2710e81209ac	0x71c3	0x2ca65327	0x532737ba
29	0x8d142caa2710e812	0x1918	0x532737ba	0x37babee3
30	0xfa048d142caa2710	0x1110	0x37babee3	0xbee3bca2
31	0x32f2fa048d142caa	0x0908	0xbee3bca2	0xbca26565
32	0x7db932f2fa048d14	0x0100	0xbca26565	0x65656877

From this we see the decrypted message is: 0x65656877, which matches the NSA paper's [1] expected outcome, and the message encrypted by the previous section.