Final Year Project

Status Report

Brandon Walsh - 13428222

Contents

Table of Figures

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 in regards to how fast the system can encrypt information, the hardware costs involved and power consumption.

SIMON; is a collection of block ciphers released to the public by the NSA (National Security Agency) in mid 2013[1]. Each individual cipher works very similar to its counterparts, with only the message and key input bit lengths varying (thus affecting some details of the workings of the cipher) SIMON itself is a balanced Feistel cipher, capable of encrypting blocks of data from 32bits up to 128bits in a single execution.

SIMON was designed to fill a cryptographic gap in the market. Few encryption methods had been designed to work in the relatively new field of "lightweight cryptography"[2], an area concerning smaller, 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.

Devices such as pacemakers, or mobile phones need a way of communicating with the world around them, while keeping the user safe. At the same time however; the encryption methods used need to be physically small, energy efficient and quick enough to not affect the device's ability to communicate with the outside too much.

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, throughput and ability to be serialised at various levels. Decryption of Simon encrypted data is a process very similar to encryption; infact it is almost the very reverse of it, though 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 giving developers an easier step towards encryption of their designs.

For this project, I intend to create an efficient and cost effective implementation of the cipher. Moreover, I intend 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. 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.

Findings

VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) is a programming 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 a programming language used to design circuits instead of just documenting them.

In 1988, the rights to VHDL were transferred to the IEEE as standardisation 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 I have become familiar with creating large-scale logic systems with nothing more than paper; I believe that working with this language will be a welcome boon, allowing a rapid transition to simulation and testing of my designs, as I tinker with them.

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 [9][10]

I'm restricting myself from viewing that VHDL version, though have used a Java implementation I found to get my JavaScript version working[11]. 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 useful way, will help others in their implementation of the cipher.

Solutions

At current, I have three ideas of how to implement the system in my design:

Double Pipe

Looking at the system, the encryption is performed by expanding the key into a much longer "generated key", then using segments of this generated key to encode the message data multiple times.

Both processes can be seen as assembly lines working side by side. The key is split up into segments (2, 3 or 4). The system then passes these segments down an encryption line where new keys are created at multiple stages, but all generation relies on the previous results. These segments are used to encrypt the message multiple times.

For example, let's say that the system receives the key "00112233" which has the four segments '00', '11', '22' and '33'. Using these segments, the first key generator machine on our "key generator" assembly line creates the segment '44'.

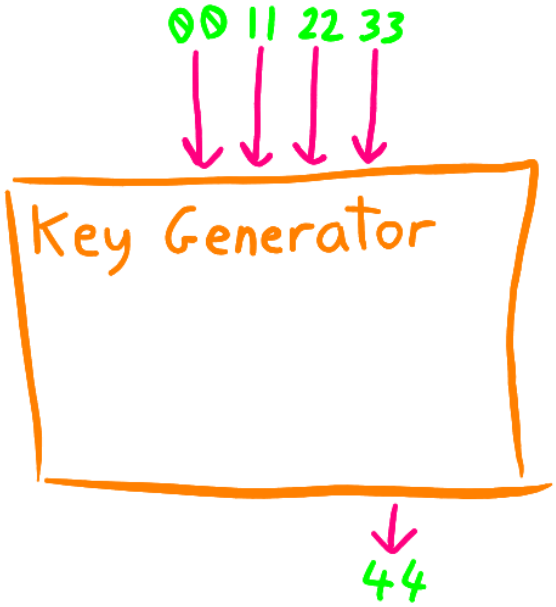

Figure 1 - Basic Key Generator

So now we have the key 0011223344. The next machine on the line, uses the most recent 4 segments (the same number of segments that were in the original key), namely '11', '22', '33' and '44' to create another segment; '55'. The line goes on like this for a predefined number of generations, creating a key much larger than our original.

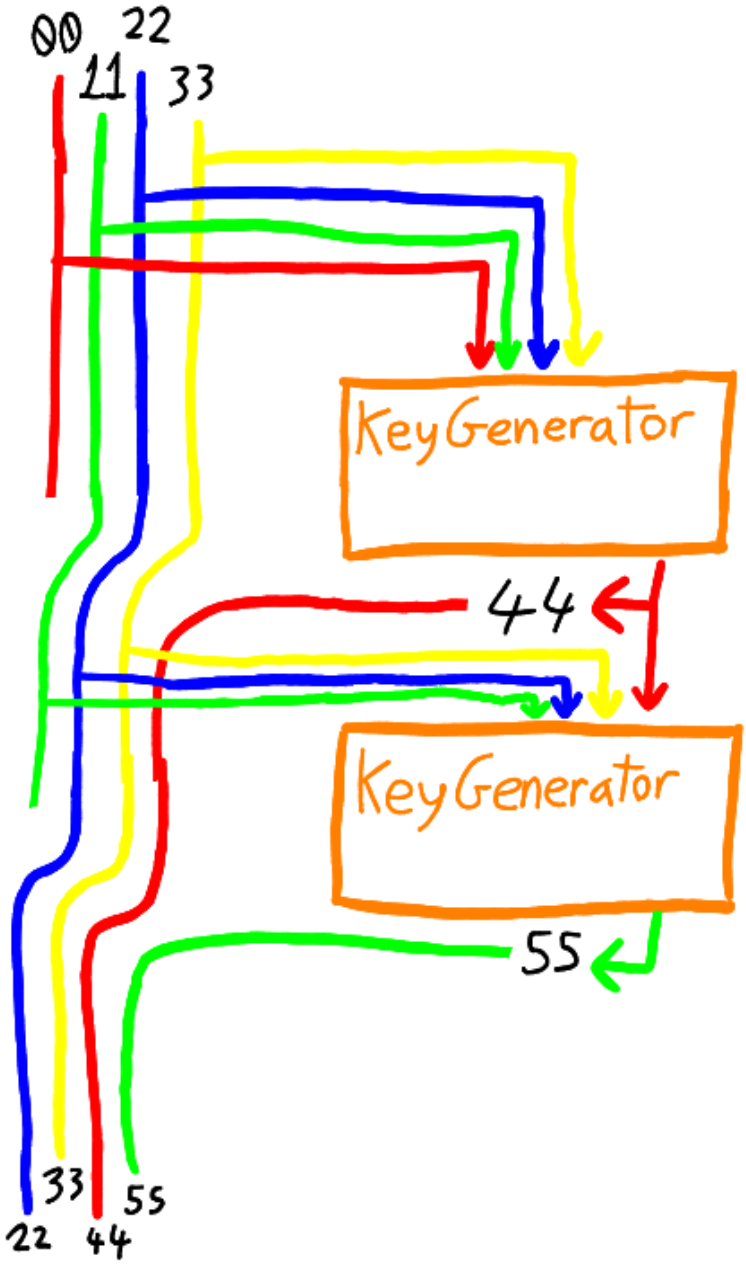

Figure 2 - Key Expansion

Now, as this is happening, a second "message encryption" assembly line is running alongside. This line has the same number of machines that the "key generator" assembly line has (plus 4), but these machines are used for the actual encryption. Each segment the "key generator" assembly line makes (including the first four that were provided, hence

why this line is 4 machines longer) is passed to the corresponding encryption machine. The message is passed into the first encryption machine, encrypted with the first key segment, then that machine's output it passed into the second machine, and so on. Thus the message is encrypted, then the encrypted message is encrypted, then the encrypted encrypted message is encrypted, then the encrypted encrypted encrypted message is encrypted, etc.

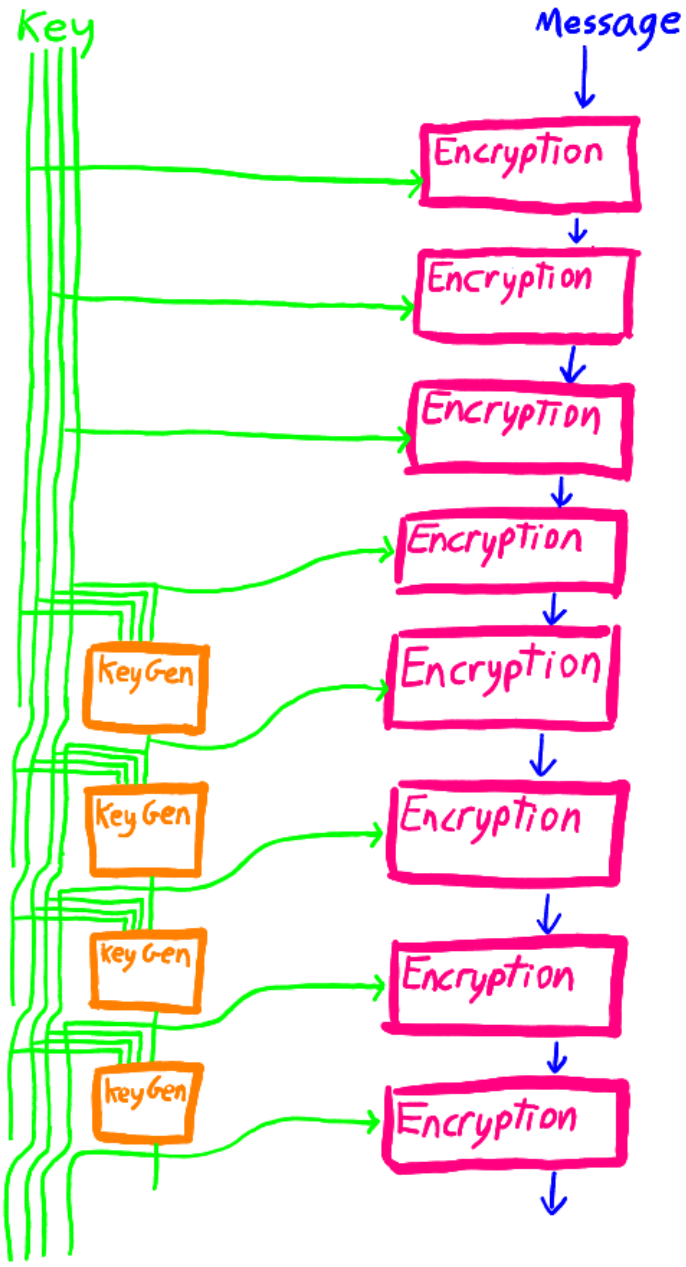

Figure 3 - Key Expansion and Encryption Process

(It is important to note that the numbers used here are purely illustrative)

Currently, my developed system does something very similar to this; creating the next key segment just before the next encryption is performed, with each stage holding onto the previous 4 key segments. In this way double-piping is implied, though may not actually be generated by the VHDL compiler.

Double Pipe with Locks

This idea, is actually a continuation/development of the previous one. As you can see from the previous design, encryption is quite linear. As such, as the last "machines" are doing their work, the ones before are sitting idle. By implementing a series of 'flip-flops' (or 'Pipelining' as it is known in VHDL) which can hold the encrypted data at different stages; multiple messages and keys can be sent into the system, even before the first encryption has completed and returned a result.

To grasp this concept, one has to understand that the system as it is; is a reactionary one. Each "machine" is reliant on the output of the machines that feed into it for what they use to do their job. So, one has to wait until the last machine completes its work and outputs a result, before you can send in new data for the system to work on.

With pipelining, we add check-points to the assembly line. Let's say half way through the lines, a new "process locking" machine is there (like a water lock on a canal) between the "key generator" and "message encryption" machines of each line. As the key segments and encrypted messages pass through this halfway mark, the "locking machine" takes note of what data exists between the machines of the first and second half, and cuts off the second half's access to the first half's results. Instead supplying the second half with the data that it noted. The second half's machines carry on as if nothing has happened, producing a result at the output end of the line, but as they are doing their work; a new set of inputs can be sent into the first half of the line. Once this new job makes it all the way down the line to the "locking machine", the previous job has made it out the other side, so the machine just notes this new data, passes it to the second half, and we start all over.

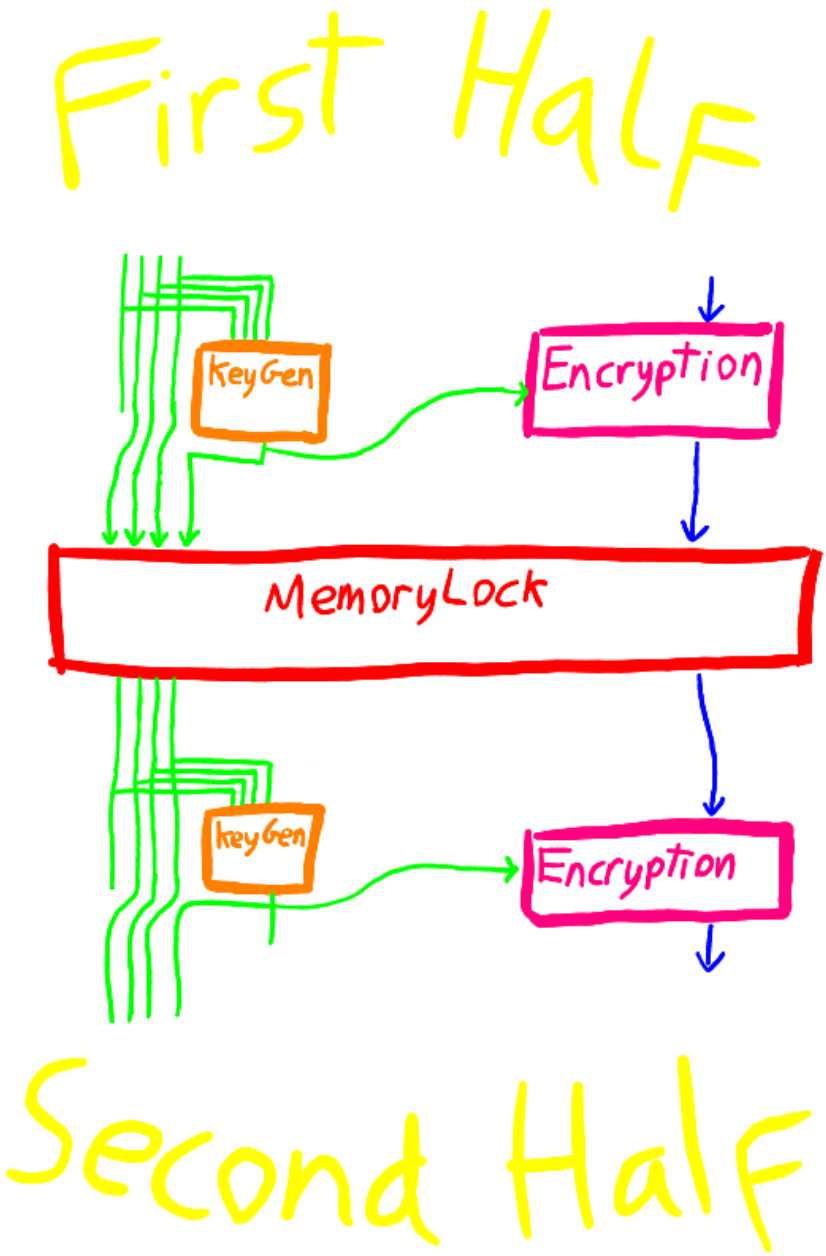

Figure 4 - Position of the Memory Lock within the System

In this way; two encryption processes can happen at the same time, doubling the throughput of the system. With more locks, this throughput can increase, though an individual encryption process will take longer than before.

Processor-esk

For this idea, I took inspiration from the architecture of a simple microprocessor. Returning to our production line metaphor; instead of having many many "key generator" and "message encryptor" machines; why not have just the one set and feed their output back into their input? In this way we can reduce the number of "machines" needed. We would need to add on a 'encryption management' system to do this feedback-

ing, watch over how many times it was being done and collect the results, but this could result in a substantial saving in components required.

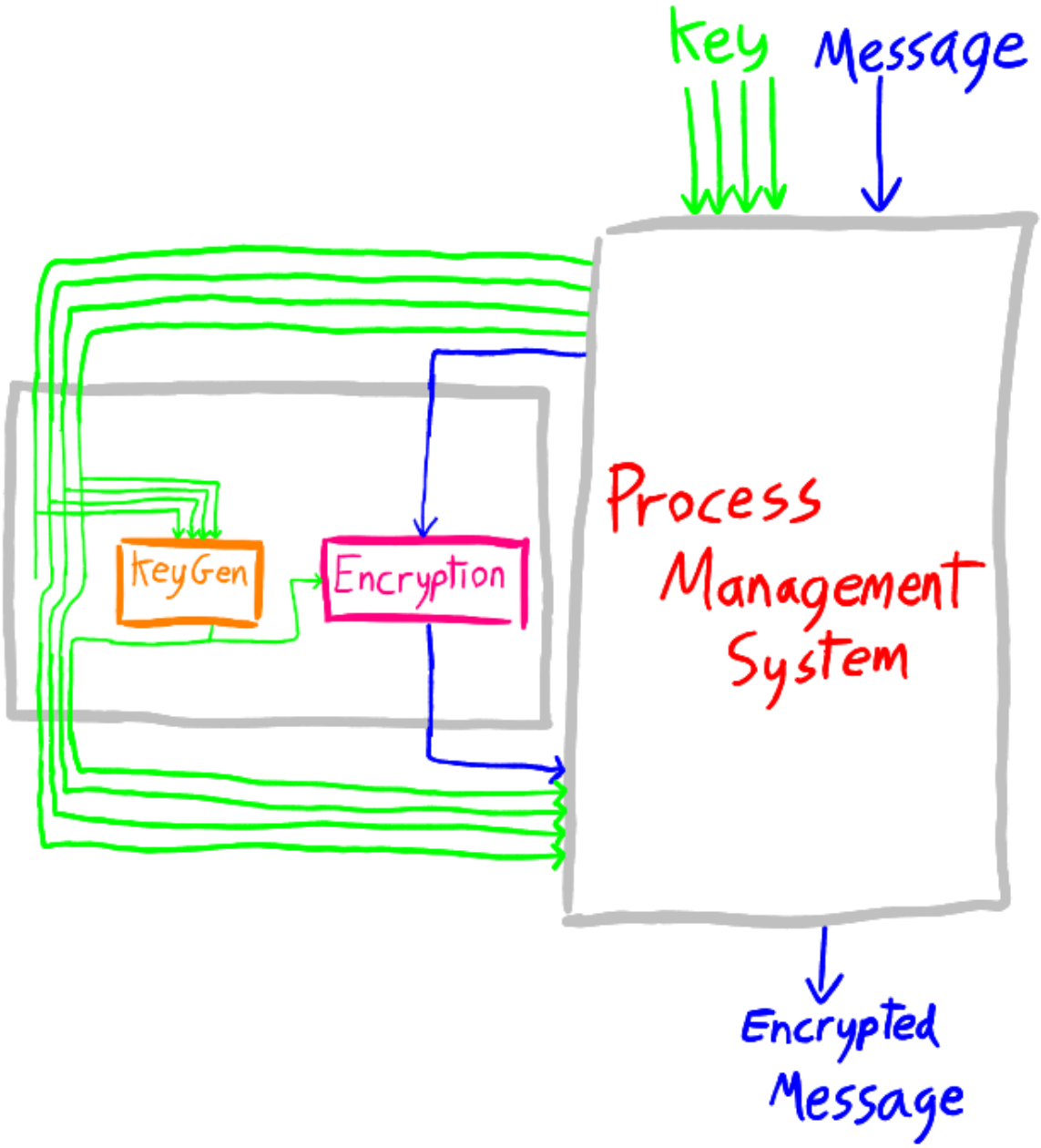

Figure 5 - Outline of Processer-esk System

The lead draw back in this idea would be that the processing time of a single encryption could possibly take much longer than in the original "double pipe" concept, but in the interests of saving parts, I feel it's a worthy endeavour.

In addition, with the saving of parts, one could build multiple versions of this system into a single chip which would allow for true parallel processing to take place.

Future Plans

Having worked out how the cipher works, and creating a JavaScript prototype in early semester one[12]; I intend to use semester two developing the VHDL implementation - perhaps creating slightly different versions for different performance results (as mentioned before) - while writing a paper on how to implement the cipher for others to use and learn from.

Considering that the report is to be submitted at the start of week 11 of semester 2 (10/4/2017) I will consider this to be my project deadline. From the week of the deadline of this report; week 10 of semester 1 (28/11/2016), that gives me a total of 19 weeks.

At the time of writing; I have a fully working implementation of the cipher in JavaScript[13], and a functional version of its most basic method in VHDL in the logic-pipe style described before. I have a VHDL IDE at home, and intend to use that in developing this version further over the Christmas break. After this (and the return to college) I plan on working on the "logic-pipe with water locks" and "Processor-esk" versions and writing the instructional paper on implementing and using SIMON.

My supervisor has offered to lend me a Zybo FPGA development board[14], so I shall also be using that to perform some real world tests of my implementations (in conjunction with some external development equipment, such as Arduinos, etc)

Project Time Plan

Project Week	Date	Other Events	Project Activities
Week 0	21/11/2016		Finish Status Report
Week 1	28/11/2106	Status Report Due	
Week 2	05/12/2016	Last Week of College	
Week 3	12/12/2016	Exam Prep Begins	
Week 4	19/12/2016	Christmas Week	
Week 5	26/12/2016		Develop "Double-Pipe" version
Week 6	02/01/2017		
Week 7	09/02/2017	Semester One Exams Begin	
Week 8	16/01/2017		Oral Presentation Work
Week 9	23/01/2017		

Project Week	Date	Other Events	Project Activities
Week 10	30/01/2017	Semester Two Begins	Borrow Zybo board - test design developed over the break
Week 11	06/02/2017	Oral Presentations Begin	
Week 12	13/02/2017		Report writing begins, also develop of “Double Pipe with Locks” version
Week 13	20/02/2017		
Week 14	27/02/2017		Develop “Processor-esk” version
Week 15	06/03/2017		
Week 16	13/03/2017		
Week 17	20/03/2017		Create paper on implementing and using Simon
Week 18	27/03/2017		
Week 19	03/04/2017		Finalising report
Week 20	10/04/2017	Report Due (Week 11)	Project Finished

References

[1] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, Louis Wingers, “The Simon and Speck Families of Lightweight Block Ciphers”, NSA, Fort Meade, MD, 2013

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

[3] “A Brief History of VHDL”, doulos.com, 2014, [Online]. Available: https://www.doulos.com/knowhow/vhdl_designers_guide/a_brief_history_of_vhdl [Accessed: 22- Nov- 2016]

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

[5] “The Computer Engineering Handbook”, CRC Press, Boca Raton, FL, 2002, p. 12-6, 12.2

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

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

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

[9] Guangyan Song (GSongHashrate), "Simon & Speck block cipher implementation open source code in C", github.com, 2016 [Online] Available: https://github.com/GSongHashrate/SimonSpeck [Accessed: 24- Nov- 2016]

[10] Calvin McCoy (inmcm), "Implementations of the Simon and Speck Block Ciphers", github.com, 2016 [Online] Available: https://github.com/inmcm/Simon_Speck_Ciphers [Accessed: 24- Nov- 2016]
[11] Tim Whittington (timw), "The Simon family of block ciphers", github.com, 2016 [Online] Available: https://github.com/timw/bc-java/blob/feature/simon-speck/core/src/main/java/org/bouncycastle/crypto/engines/SimonEngine.java [Accessed: 24- Nov- 2016]
[12] Brandon Walsh (metasophiea), “SIMON Cipher JavaScript - Pure System”, metasophiea.com, 2016 [Online] Available: http://metasophiea.com/lib/js/SIMON/pureSystem/SIMON.js [Accessed: 24- Nov- 2016]
[13] Brandon Walsh (metasophiea), “SIMON Cipher JavaScript - Test Page”, metasophiea.com, 2016 [Online] Available: http://metasophiea.com/lib/js/SIMON/test.html [Accessed: 24- Nov- 2016]
[14] “ZYBO Zynq™-7000 Development Board”, digikey.ie, [Online]. Available: http://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- Nov- 2016]