TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Hardware Acceleration of Adiantum Cryptography Algorithm on PYNQ

Author:
Konstantinos
AMPATZIDIS

Thesis Committee:
Prof. Apostolos DOLLAS
Assoc. Prof. Vasilios
SAMOLADAS
Assoc. Prof. Sotirios
IOANNIDIS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

April 22, 2021

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Hardware Acceleration of Adiantum Cryptography Algorithm on PYNQ

by Konstantinos AMPATZIDIS

As technology is closely interwoven with everyday reality, an exponentially increasing volume of information is exposed to potential data breaches. In that context, the field of cryptography offers the necessary confidentiality and accuracy to sensitive data handling. Existing options such as AES can often lead to significant performance expense, yet recent appearance of more lightweight alternatives, like Adiantum, resolves the dilemma of choosing between speed and security. The connection between hardware development and cryptography is inevitable as hardware offers high parallelism which in turn results in faster deployments with balanced power consumption. In this thesis we present the first attempt at accelerating the entire Adiantum algorithm for big plaintexts with FPGAs. This thesis comprises of three parts: profiling of the Adiantum algorithm in order to determine the most computationally intensive parts; implementation of the ChaCha12 core which accounts for some 86% - 96% of the total load; and, full implementation of the ChaCha12 core and the Adiantum algorithm on a PYNQ Z1 FPGA board. Despite technology-related limitations the results are most encouraging. Specifically, the ChaCha12 core is 10,731 times faster and 77,000 times more energy efficient than the Intel i5-3230M processor. If it were to run on a present-day system with an Intel i5-3230M processor having a tightly-coupled FPGA, the Adiantum algorithm, including I/O overhead would run at speeds approaching the theoretical limits posed by Amdahl's Law. However, because the processor on the PYNQ Z1 is 15 times slower than the Intel i5-3230M processor, the full-Adiantum-algorithm performance of the Intel i5-3230M CPU is 4x times faster than our Pynq-z1 system, but at a 2x higher energy cost.

Acknowledgements

First, I would like to express my deepest appreciation to my supervisor Prof. Apostolos Dollas for his guidance both throughout my studies as a college student and during the course of this thesis. All of the ideas and expertise that he offered me, will continue to broaden my horizons. But most importantly, I would like to thank him for giving me the knowledge and the opportunity to be a part of the research community in the field of hardware.

Also, I would like to extend my deepest gratitude to my advisor Dr. Dimitrios Theodoropoulos for his guidance and continuous support in this project. I would like to thank him for giving me the opportunity to expand my knowledge at re-configurable computing and enlightening me in the field of cryptography. He has been a great mentor for me during the last year both at a professional and a personal level.

Furthermore, I would like to thank my thesis committee, Prof. Vasilios Samoladas, and Prof. Sotirios Ioannidis, for evaluating my work.

I would also like to acknowledge the creators of Adiantum, Paul Crowley and Eric Biggers for their immediate response to my initial queries and substantial help in understanding Adiantum.

Last but not least, I am grateful for my family for their love and moral support all of these years. I am also thankful of all my friends who stood by me at all times during my academic years. Finally, special thanks to my girlfriend who has been understanding during the recent years that involved a lot of studying and most importantly who was by my side at all times, making it possible to hold this thesis in the midst of a pandemic while continuing to make me happy.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
AES	**A**dvanced **E**ncryption **S**tandard
BRAM	**B**lock **R**andom **A**ccess **M**emory
CBC	**C**ipher **B**lock **C**haining
CLB	**C**onfigurable **L**ogic **B**lock
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
DDR4	**D**ouble **D**ata **R**ate 4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
ECB	**E**lectronic **C**ode **B**ook
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
GPU	**G**raphic **P**rocessor **U**nit
GUI	**G**raphical **U**ser **I**nterface
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
IDE	**I**ntegrated **D**evelopment **E**nvironment
ILA	**I**ntegrated **L**ogic **A**nalyzer
ISE	Xilinx **I**ntegrated **S**oftware **E**nvironment
LUT	**L**ook **U**p **T**able
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
MAC	**M**essage **A**uthentication **C**ode
PL	**P**rogrammable **L**ogic
PLD	**P**rogrammable **L**ogic **D**evice
PS	**P**rocessing **S**ystem

RTL	**R**egister **T**ransfer **L**evel
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
TDP	**T**hermal **D**esign **P**ower
URAM	**U**ltra **R**andom **A**ccess **M**emory
USB	**U**niversal **S**erial **B**us

Dedicated to my family and friends…

Chapter 1

Introduction

Nowadays, technology essentially intertwines with daily reality, thus affecting not only the greatest fields of academic research but nearly all aspects of ordinary life in general. Since the invention of the first computers, there has been an exponential growth both in their use for research as well as in everyday life. The more people use electronic devices the more personal information is expressed in computer language and, as a result, the need for security arises.

There are many types of computer systems in terms of performance and capabilities and obviously cannot be compared with each other. For example, the computing power of a modern Central Processing Unit(CPU) is clearly more powerful than the CPU of a mobile phone or a smartwatch. In addition, as science evolves, simple electronic files like documents or photographs and many others, consume increasingly more storage space. To be more specific based on the National Security Agency, at 2013 internet processed 1.8 Petabytes per day [1]. In 2018 based on Forbes, 2.5 quintillion bytes of data were created every day [28] while it is estimated that until 2025 the total amount of data created worldwide will rise to 163 ZettaBytes [29]. The larger the file size is, the longer it takes to encrypt or decrypt the data.

Nevertheless, as the great Roman Stoic philosopher, known as Seneca said: *"It's not that we have little time, but more that we waste a good deal of it"*.
That is, no-one should have to be stuck in front of a computer system just waiting for encryption/decryption process to finish. Concerning cryptography approaches, there are numerous encryption methods, some of which target disk encryption. However, disk encryption cannot replace file encryption as it does not protect data within an operating system from malware or physical access. The optimal outcome in terms of security is achieved with a combination of the two encryption methods.

1.1 Motivation

Cryptography as a concept originates from the Greek words "*krypto*" and "*grafo*", which means "hidden" and "to write" respectively. Cryptography can be thought of as “the art of writing secrets" [22]. It is impressive and important to reflect on the fact that 4000 years ago this was done by hand [20]. Today, cryptography is considered a branch of number theory and computer science and the use of electronic devices is required as the complexity and the size of mathematical operations make it impossible in terms of time for humans.

In practice, cryptography is the process of converting data ("*plaintext*") into an unreadable form for a human eye through mathematics ("*ciphertext*"), using a secret key. The above operation is called encryption while the reverse where we convert the ciphertext back to the plaintext is called decryption ([32], [22]). Cryptography ciphers are mainly divided into symmetric-key ("*secret-key ciphers*") and asymmetric-key("*public-key ciphers*"). In the first one, the same private key is needed both for encryption and decryption while in the second one a public key is used for the encryption part and a private key for decryption (figure 1.1).

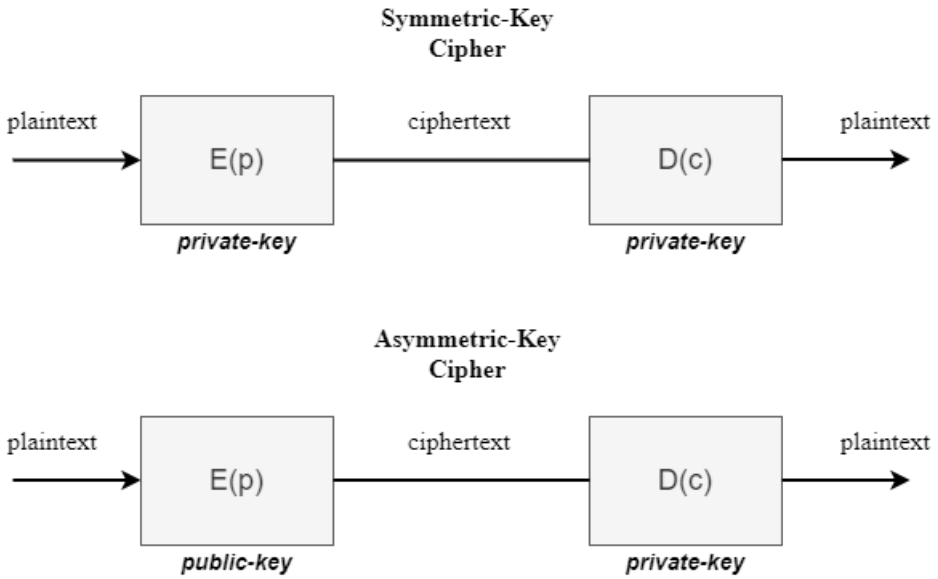

FIGURE 1.1: Symmetric and Asymmetric key ciphers

Apart from the theory of cryptography, the real challenge in practice is the construction, implementation and time execution of such algorithms. It is important to note that encryption algorithms usually run in the background of a computer system so that the user does not waste time with that part of their computer, let alone notice it. In addition, as already mentioned, there are many different types of processing systems, so a brief overview of some of them is essential.

Central Processing Units(**CPUs**) are the least efficient solution for running complex algorithms. The execution times are affected due to low parallelism and in combination with the high power consumption, make CPUs a poor decision as a hardware choice. On the other hand, Application Specific Integrated Circuits(**ASICs**) are the exact opposite of the CPUs. ASICs are a good hardware choice as they offer high parallelism and the lowest power consumption. However they can only serve the application for which they were designed while they do not offer future flexibility and are really expensive.

Graphical Processor Units(**GPUs**) have upgraded the Single Instruction Multiple Data(SIMD) architecture of CPUs to a whole new level dedicated to offering parallelism of tasks. The much simpler control logic in addition to small per core memory has led to simpler computing cores allowing GPUs to contain a greater amount of cores per chip than a CPU. GPU architectures can perform very well on workloads that have no data dependencies or branching conditions. Additionally, even though a GPU has to communicate with a CPU, their memory is specialized to support high-speed data streaming. Until recently, the high power consumption of GPUs rendered them unsuitable as hardware accelerators [13], yet the situation seems to be changing drastically.

Finally Field-Programmable Gate Arrays(**FPGAs**) blend the performance of an ASIC system with the flexibility of a microprocessor. FPGAs can offer high parallelism with balanced power consumption. One of the most significant advantages of the FPGAs is that they can be further optimised or redesigned every time, incorporating the needs of each application. Even so, in order for the designer to grasp and utilize the capabilities of an FPGA, he must be able to tackle both hardware and software issues. The designer needs to handle the available gates/hardware resources for performing the requested application computations but at the same time watch the software part that supports the whole process [17].

While, assumedly, the ongoing battle between FPGAs and GPUs comes to an end with the latter standing out as a better choice, the continuous research by Intel and AMD for a hybrid model of a CPU-FPGA device presents new opportunities for FPGAs. Intel has also designed a Development Kit(Arria 10 GX FPGA - figure 1.2) that can highlight the benefits of this hybrid design. By merging the logic of an FPGA with a CPU, they achieved lower latency and higher bandwidth between the communication of the two of them, while they can also share resources such as cache and system memory. Moreover, the hybrid model has also opened up discussions in the research community for new technologies and especially for a future CPU-GPU-FPGA hybrid design.

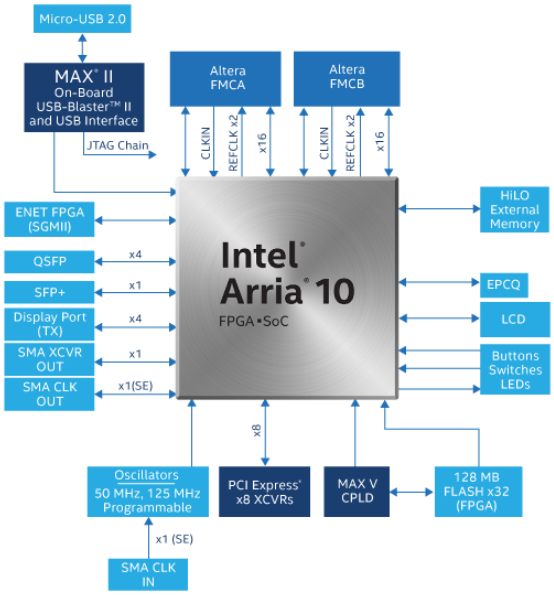

FIGURE 1.2: Arria 10 GX FPGA Development Kit Block Diagram URL

The amazing field of cryptography combined with the hardware perspective is really intriguing. This thesis analyzes and discusses the benefits of FPGAs for the hardware acceleration of Adiantum[14] cryptography algorithm. Adiantum is a relatively new cryptography algorithm(published on December of 2018) that belongs to symmetric ciphers with the addition of a second key that is called tweak [26]. Adiantum is intended by Google for low-end devices and specifically for disk encryption. Nonetheless, Adiantum's ability to handle very big messages makes it an interesting choice for optimization and acceleration for similar tasks, thus broadening the spectrum of use possibilities.

1.2 Scientific Contributions

The goal of this thesis was to study and explore how the Adiantum algorithm performs with plaintexts bigger than 4096 bytes and how an FPGA can improve the performance and time execution. Adiantum was chosen because it is a relatively new cryptography algorithm that consists of existing cryptography models/methods/primitives which, in combination with the security they offer, make it very interesting for research. In addition, Adiantum can easily be improved in the future with more diffusion and security.
Towards achieving Adiantum acceleration we followed the procedure bellow:

1. Profiling of Adiantum algorithm in order to reveal the most computationally intensive parts.
2. Implementation of the time consumable parts with hardware (In our case ChaCha12).
3. Full execution of the whole Adiantum algorithm after incorporating our implementation on a PYNQ Z1 FPGA.

The most outstanding outcome is that if our design were to run at a strong CPU like Intel-i5 3230M with a tightly-coupled FPGA, including I/O overhead, would meet the theoretical limits posed by Amdahl's Law.
In comparison to Intel-i5 3230M CPU our **ChaCha12 design** is:

- 10,731x faster
- 77,000x more energy efficient

However, given that software improvements led to a better standalone version of Adiantum and since PYNQ processor is 15 times slower than Intel-i5 3230M, consequently Intel-i5 3230M processor is:

- 4x time faster
- 2x higher energy cost

Finally, our design offers flexibility for further improvements as the outcome of this thesis introduces novel concepts and areas for future research.

1.3 Thesis Outline

- **Chapter 2 - Related Work & Tools:** Description of all related work for Adiantum and its points of interest as well as an explanation of all the tools that were utilized.
- **Chapter 3 - Architecture Analysis:** Detailed presentation of the theoretical background for Adiantum, software improvements and a theoretical analysis of the hardware implementation.
- **Chapter 4 - FPGA Implementation:** Detailed analysis of our design and the hardware implementation.
- **Chapter 5 - Results:** Here all the results from the different platforms and designs are compared and discussed thoroughly. Specifically metrics like throughput, latency, power and energy efficiency are visualised with MATLAB tools [30].
- **Chapter 6 - Conclusions and Related Work:** The last chapter concludes all outcomes of this thesis and some ideas for further work and optimizations.

Chapter 2

Related Work & Tools

Notably, so far there is no relevant research on the Adiantum encryption algorithm either from the software or the hardware (implementation/ acceleration) point of view. Therefore, everything presented in this thesis is a new approach aimed at accelerating the algorithm through hardware. However, Adiantum has been constructed by already existing cryptography primitives. Furthermore, as this thesis result came with the cooperation of software & hardware and specifically with the hardware implementation of ChaCha12 part that Adiantum uses, it is important that some relevant research, based on the ChaCha algorithm be analyzed ([4] , [40], [21] , [38]).

2.1 Compact hardware implementations of Blake, Skein, ChaCha & Threefish

In the work of Nuray At. et al. [4] compact implementations of the cryptographic hash functions of Blake [5] and Skein [27] are presented. Blake and Skein are based on ChaCha stream cipher and Threefish block cipher respectively and as a result, the paper authors also implemented the latter two. The main idea of these compact implementations is to take advantage of the parallelism that these algorithms already offer. Additionally, by designing pipeline Arithmetic Logic Units(ALUs) with VHDL to avoid data dependencies, the authors achieved lightweight co-processors that specialized in these algorithms. Finally, the result was tested with place & route logic on Virtex-6 FPGAs.

First of all, a separate ALU design for BLAKE and ChaCha was proposed and after that a compact implementation of the two of them. The main idea is a register-file implemented as a dual port memory, an ALU and a control Unit.

The user interacts by choosing the ideal algorithm and also gives the message, plaintext or ciphertext blocks at one port of the register file. While the co-processors are hashing/encrypting the message, the intermediate results are written to the other port of the register file. Secondly, in order to define words of at most 18 bits for instruction memory, a simple compression algorithm that also generates the VHDL description for the decompression circuit has been designed with C-language. Finally, with the control unit as a counter and an Finite State Machine(FSM) that terminates the procedure when the hash completes, they achieved 46 MB/s for ChaCha12 encryption.

2.2 FPGA implementation of HS1-SIV

Hash Stream 1 - Synthetic Initialization Vector(thus HS1-SIV) [24] uses HS1 as a Pseudo-Random Function(PRF) to provide deterministic authenticated encryption with SIV mode [36]. The algorithm consists of six sub-routines HS1-SIV Encrypt, HS1-SIV Decrpyt, HS1-Hash, HS1-Subkeygen and ChaCha. In general HS1-SIV takes some parameters as input which in turn are used to initialize collision level, number of bytes for hash part, rounds of Stream cipher and the byte-length of synthetic IV. However for the hardware implementation, Gerben Geltink and Sergei Volokitin chose these parameters as static and thus they named it HS1-SIV-med [40].

Specifically in the ChaCha stream cipher paper [40] , authors chose 12 rounds same as in Adiantum. Implemented with VHDL language and tested on Virtex-7 the ChaCha part is composed by nine blocks (Set_State, Inner_Block (1..6), Add_States and Serialize). The Set_state is the first block that receives the input (nonce [96-bits], block-count [32-bits] and key [256-bits]) and it is responsible for the initialization of the ChaCha block. After initialization, the six inner_blocks run for the 12 rounds of ChaCha cipher. Finally the add_state adds the current vector (after the inner blocks) with the first version of it (after the Set_state) and through Serialize block the resulted stream goes out of the ChaCha block with each 32-bit word in little-endian order.

Although, this implementation is similar to the one presented in our thesis, the ChaCha block is not an autonomous block as it is contained in the logic/ implementation of HS1-SIV-med design. Additionally, the block-counter of ChaCha block is 32-bits while for Adiantum is 64-bits.

2.3 Hardware Design of ChaCha20 & Poly1305

The combination of ChaCha20 stream cipher and Poly1305-Message Authentication Code(MAC) is very common [31], especially for low-end devices. Moreover, Adiantum uses the variation XChaCha12 instead of ChaCha20 with Poly1305. In the paper "High-Throughput Low-Area Hardware Design of Authenticated Encryption with Associated Data Cryptosystem that Uses Cha Cha20 and Poly1305" [21], G.Kanda and K.Ryoo designed efficient hardware for these two algorithms with Hardware Descriptive Language (HDL)-Verilog and tested it both on a Virtex-7 FPGA and an ASIC system.

Concerning the ChaCha part, the design consists of the blocks Little Endian Serializer, Init_State_Matrix, ChaCha20 State Generator and a Controller. Key (256-bits) and nonce (96-bits) go into Little Endian Serializer which converts them in little-endian order. The result and the block counter enter the Init_State_Matrix which is responsible for generating the 4x4 Initial Matrix of ChaCha algorithm. After the initialization part, ChaCha20 State Generator block operates the Rounds-logic procedure of Cha-Cha algorithm and the result is added to the initial matrix. Finally, an xor operation with the plaintext/message completes the encryption.

Furthermore, ChaCha20 State Generator can run four Quarter-Rounds either in parallel or serialized. Additionally, the Controller block includes an FSM implementation for the termination of the procedure once the whole message has been encrypted. The results from both the FPGA and the ASIC can be seen at the follow (table 2.1):

TABLE 2.1: ChaCha20 implementations: FPGA vs ASIC [21]

Hardware Technology	Design	Frequency(MHz)	Throughput(MB/s)
FPGA	Serialized QR	182.40	135
FPGA	Parallel 4xQR	161.02	479
ASIC	Serialized QR	420	312
ASIC	Parallel 4xQR	312	929

2.4 Implementation of ChaCha20 in SoCs

A relevant thesis based on ChaCha20 stream cipher and its hardware implementation has recently (year 2020) been publish by Igor Semenov [38] of The University of Alabama in Huntsville. Aiming for hardware acceleration of ChaCha20 with low resources Igor Semenov has chosen to implement only its Round logic in hardware, leaving the summation and xor part of ChaCha to processing system(PS). Additionally, after a theoretical analysis for achieving low hardware resources he made clear that a pipeline system would not lead to the desired result. Utilizing only that the Round part of ChaCha can be pipeline due to its own logic, he designed an efficient hardware acceleration.

The design has been implemented with System Verilog and tested at Terasic DE10-Nano Kit which belongs to the family of Intel Cyclone-V [18] field programmable gate arrays(FPGAs). The accelerator (or FpgaCha-Ip core as named by the designer) consists of three blocks (ChaCha20 accelerator, FIFO, and S2M adapter). Except for the FIFO block which belongs to the Intel Platform Designer, the other two modules are custom. The transactions between PS and PL take place through AXI to Avalon adapter blocks. ChaCha20 accelerator is the first block that receives the input (nonce [96-bits], block-count [32-bits], and key [256-bits]) as 32-bit words and runs the 20 rounds of ChaCha cipher. With an additional input (named as pad_counter) the hardware design is aware of how many ChaCha blocks/results the software needs which also translates into how many times the ChaCha20 accelerator will run. The ChaCha results are then transferred to the FIFO block and through that to the S2M adapter. The S2M adapter takes 512-bit each time through Avalon ST-sink interface from FIFO and then splits them into two words of 256-bit each one. Finally the two words are both transferred in burst mode at the same cycle to the processing system. They used different approaches with 1, 2 and even 4 FpgaCha cores for encrypting one message (table 2.2).

TABLE 2.2: ChaCha20 throughput comparison

Hardware Technology	Design	Frequency(MHz)	Throughput(MB/s)
FPGA	1 FpgaCha core	50	90
FPGA	2 FpgaCha core	50	126
FPGA	4 FpgaCha core	50	123

2.5 The FPGA Perspective

FPGAs are programmable devices/integrated circuits and a part of the Programmable Logic Devices (PLDs) family. Unlike all hardware options, the main advantage of FPGAs is that they are not fixed to a specific hardware but they can be reconfigured and thus in general are referred to as re-configurable computing. This asset offers much flexibility as the designer can construct different hardware implementations depending on each application requirement.

The thousands of building blocks known as Configurable Logic Blocks (CLBs) connected trough programmable interconnections enable the designer to reconfigure new digital circuits every time. Additionally, Digital Signal Processing blocks (DSPs) which are a part of FPGA fabric can optimize hardware, especially for mathematical operations like multiply-accumulate and division. The latter makes the representation of floating-point, fixed-point, and even integer values feasible.

Furthermore, FPGAs can have hard processor cores in the same chip offering much faster processing speeds since they are not limited by fabric. By combining hard processors lots of data can be processed which is fundamental for communications, scheduling and data pre-post processing. Additionally, as FPGAs also offer high energy efficiency and low latency they have been established in the field of hardware.

Concerning the cryptography field, many algorithms and especially block ciphers are based on bit-wise logic operations and their implementation fits remarkably well with the FPGA perspective. Also, as the most fundamental operation in cryptography is substitution which needs a lot of memory resources, the pipeline design logic in addition to BRAM and URAM that an FPGA can offer leads to significant results. Some cryptographic algorithms as already seen (ChaCha algorithm) have been used for various applications and also for different cryptographic constructs. As a result, the dynamic configuration and flexibility of FPGAs make them an optimal choice.

2.6 Tools Used

The main tool used for the implementation and optimization of Adiantum Cryptography Algorithm in FPGA platform was Xilinx Vivado Design Suite and specifically HL-System Edition 2019.2 [41]. The use of Vivado Design Suite facilitates acceleration of the design implementation via place-and-route tools which analytically optimize for several and synchronous design metrics, for instance timing, total wire length , utilization and certainly power.

Developed by Xilinx Company, Vivado Design Suite is a software through which analysis and synthesis of Hardware Description Language(HDL) like VHDL or Verilog can take place. Interaction between the user and the software is feasible with a Graphical User Interface(GUI) and Tcl commands too. Xilinx ISE overtakes previous tools with many additional features for System on Chip(SoC) designs and High-Level Synthesis(HLS).

Furthermore a new open source development by Xilinx named PYNQ [34] was released providing easier interaction between the user and the FPGA board as it is using Python libraries and Jupyter Notebook [19]. For carrying out the practical part of this thesis Xilinx Vivado IDE, Xilinx Vivado HLS and Python productivity for Zynq(PYNQ) tools have been used.

2.6.1 Vivado IDE

Vivado Integrated Design Environment(IDE) serves as basis for all Xilinx tools as it is a front-end GUI of Vivado Design Suite. Combining tcl commands and a graphical interface Vivado IDE can compile, synthesize, implement, place & route hardware designs. Apart from HDL-languages (i.e. VHDL and Verilog) other languages that can be compiled too such as C/C++ and SystemC.

Vivado IDE also offers an IP Integrator tool. By utilizing IP Integrator the user has the ability to connect IP blocks with ease solely by making use of the GUI without having to write more code for every connection. Additionally, some basic connections between IPs can even be automated, making tasks like hardware acceleration more manageable for any hardware designer. These IPs can be constructed both from Vivado IDE with VHDL but also Vivado HLS in which high level languages such as C/C++ can automatically converted in VHDL or Verilog and exported as an IP block. Besides user IPs, Xilinx also offers many IPs with or without charge and therefore users

do not necessarily have to build everything from zero. Moreover, custom IP designs can be made available for public use.

Furthermore, the designer is able to export the whole block design into a bitstream file downloadable for an FPGA device. When the bitstream file is loaded at the target FPGA, PL and PS are reinitialized utilizing hardware resources such as Digital Signal Processor(DSP) slices, B-Ram, Logic Cells etc. The result can be executed either in PL/PS or in a combination of the two of them.

Taking the user's needs into consideration, Vivado IDE provides different testing and debugging approaches. There is an integrated simulator for testing, though any RTL simulator can also been used. Finally, apart from built-in debuggers, it is possible to utilize specific IPs such as Integrated Logic Analyzer(ILA) [39] dedicated for deep debugging. ILA is also used in the course of this thesis making monitoring of internal signals feasible. Within Vivado GUI, user specifies which signals trigger ILA IP in order to keep track of their values during software execution.

2.6.2 Vivado High Level Synthesis (HLS)

HLS is a tool that can be combined with Vivado IDE for generating IPs from high level languages like C/C++ and SystemC. Any program written in C based languages like HandelC, MATLAB can be transformed into a register transfer level(RTL) for synthesis and implementation onto PL of any Xilinx Field Programmable Gate Array(FPGA) or Zynq device.

In simple terms HLS provides the functionality of a processor compiler. The main difference between the two of them is the execution target. By setting the FPGA as a target, Vivado HLS gives the opportunity for optimization that can result in throughput, power and latency without limited computational resources. These can be achieved by utilizing a set of directives and timing constraints provided by HLS. However only educated use of each directive can achieve software/hardware optimization.

Lastly, before synthesis user can perform testing by compiling or debugging their code with a C/C++ test-bench. Testing files usually model a typical scenario providing test inputs in order to confirm whether the output is as expected or not. Furthermore, Vivado HLS offers a C/RTL co-simulation that applies to the same test-bench file.

Synthesis Report

Every time a code is synthesized, Vivado HLS generates a synthesis report containing performance metrics and an estimation of the hardware resources. Based on this report hardware designer can identify operations that cause bottlenecks and as a result achieve further optimization. More specifically the report summarises the following:

- **Latency**: this is fundamental for performance both for processors and FPGAs. Latency shows how many cycles an instruction or a set of instructions need to generate a result. Thus when HLS creates the synthesis report, latency represents the number of cycles for the main function and therefore the whole design.
- **Iteration Interval (II)**: The number of cycles needed before the design can accept a new value.
- **Loop Latency**: The number of clock cycles needed to finish all iterations of the loop
- **Loop Iteration Latency**: The number of clock cycles needed for one iteration of the loop
- **Loop Initiation Interval**: The number of clock cycles needed before the next iteration of the loop can process data
- **Interface**: All the signals of the design and their protocols. Usually, as also in our case, the Advanced eXtensible Interface(AXI) protocol [6].
- **Area/Utilization Estimates**: They show an estimation about hardware resources that will be utilized from the target FPGA for implementing the design. They specifically demonstrate the amount of Block RAMs(BRAMs), Ultra Rams(URAMs), Digital Signal Processing Units(DSP48s), Flip Flops(FFs) and Lookup Tables(LUTs).

Optimization Directives

As already stated, HLS offers optional directives with whom the designer can apply high level control over the implementation of the designed code. Designer can either keep directives integrated to their design as pragmas or as a separate file within the Vivado HLS project. In our case directives have been added as pragmas to the design as it is an easier way of improving

specific parts of the code. Some types of directives both for block-level and port-level interfaces are:

- **Array Map**: Several arrays are being combined into one large array thus achieving less FIFO or RAM resources.
- **Array partition**: Separate array interfaces into several smaller sections. The use of this directive, leads to an expanded set of ports, control signals and implementation resources.
- **Array Reshape**: An array is partitioned into smaller arrays. An interesting note here is that these small arrays can be recombined to form an array with less and wider data elements.
- **Interface**: Using this directive the user can specify a port level interface protocol. It is customary to use with the main function of HLS project as it is mapping which protocol each input and output argument will have in the resulted RTL and consequently our IP block.
- **Resource**: With this HLS specifies the resource for the implemented interface. If left unspecified by the user it is auto-specified by Vivado HLS.
- **Stream**: In general top level arguments are being implemented as RAM. However by specify them as streaming port they are constructed as FIFOs and moreover user can determine the depth for each FIFO.
- **Dataflow**: Enables parallel execution of functions and loops and as a result it increases throughput.
- **Unroll**: This directive also needs a factor number which determines the amount of multiple instances of a loop. These instances can run in parallel.
- **Pipeline**: Pipeline directive is of utmost importance as it reduces the initiation interval of a whole function or a loop. Given a number x as interval, each x clock cycles a loop or a function can process new inputs.
- **Allocation**: Limits the number of hardware resources.
- **Loop_Tripcount**: This Directive is only mentioned because it helps in our case otherwise is not so important. It does not affect the resulting RTL file but when loop index as an argument is undefined, HLS has to

know the minimum and maximum possible rounds of the loop. Otherwise synthesis report will not include anything with reference to the latency of design.

2.6.3 PYNQ and Jupyter Notebook

PYNQ is an open source project by Xilinx aimed to work on any computing platform and operating system and to subsequently make it easier for designers to use programmable logic and microprocessors. The main difference from all the tools that Xilinx has offered thus far is that the main high level language for using PL and download the bit-stream file to FPGA is Python [34].

By using pynq tools, programmers and engineers are able to use ZYNQ devices, without being bound to use ASIC style design tools. For pynq, programmable logic circuits are called overlays and they can be referred to as hardware libraries. These hardware libraries can be imagined like software libraries as they can be accessed through an application programming interface(API). Although overlays can be re-used and easily applied by any programmer, a new overlay design requires engineers with knowledge and skills in designing PL circuits.

Xilinx has merged PYNQ tools with the open source Jupyter Notebook as an Interactive Python(IPython) kernel [19]. By utilizing that kernel and a web browser the user is able to directly program the ARM cores of the zynq device but also to use/implement overlays based on the needs of each software.

Jupyter Notebook incorporates:

- **Notebook Web Application**: Interactive web application that enables a user to write and run a code.
- **Kernels**: Kernels are separate processes. These processes can run the user's code in the given language and are started by the notebook web application. Results are demonstrated through the notebook web application.
- **Notebook Documents**: Self-contained documents that accommodate all context in the web application including inputs & outputs of computations, images and media representations. Each notebook document has its own kernel.

2.7 FPGA Platform

Our architecture was tested and implemented with the PYNQ-Z1 board.

2.7.1 PYNQ-Z1 Specifications

Pynq-z1 is a general purpose, programmable platform for embedded systems [35]. It offers flexibility either at hardware or software applications. From PS perspective, it incorporates ZYNQ XC7Z020-1CLG400C with DDR3 memory controllers and from PL perspective the Artix-7 family. The table bellow is a representation of the main PL features:

TABLE 2.3: PYNQ-z1 Specifications

Logic Cells	B-RAM(KB)	DSP SLices	Flip-Flop(FF)	LUTs
13,300	630	220	106,400	53,200

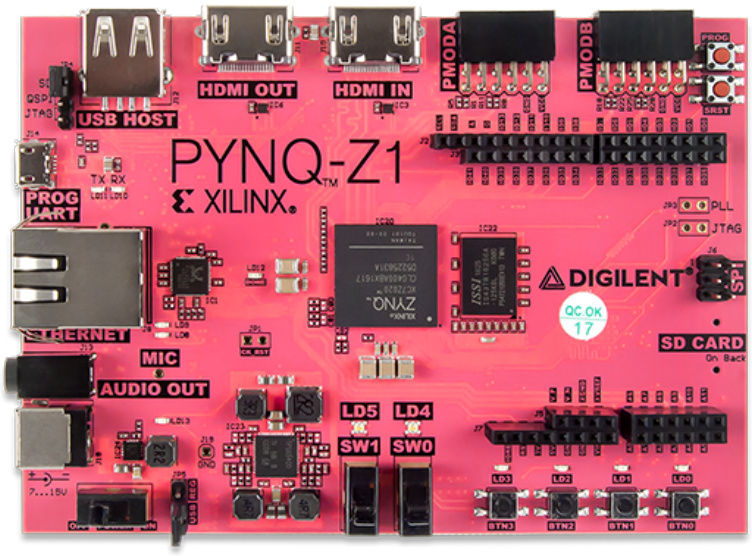

FIGURE 2.1: Pynq-z1 Top-Down

2.8 Thesis Approach

This thesis aims to analyze how the Adiantum algorithm works with very large plaintext sizes and how an FPGA can lead to hardware acceleration of these plaintexts.
The accomplished acceleration is the result of the following three processes:

1. Profiling of the Adiantum algorithm for determining its most time consumable parts.
    - Section 3.3: Profiling
2. Implementation of ChaCha12 core by utilizing FPGA benefits.
    - Section 4.3: Vivado Hardware Design
    - Section 4.4: IP Implementation with HLS
3. Full execution of ChaCha12 implementation & Adiantum algorithm on Pynq-z1 FPGA board.
    - Section 3.5: Hardware Approach
    - Section 4.1: Top-Down Strategy
    - Section 4.2: Pynq Configuration & Software Changes

Chapter 3

Architecture Analysis

3.1 Block & Stream Ciphers

Symmetric cryptography can be divided into Block ciphers and Stream ciphers. Adiantum(3.2) utilises both ciphers therefore a brief summary shall prove useful.

Block Ciphers are used for the encryption and decryption of an entire block of plaintext bits and not individual bits. Each time a whole block of bits is encrypted with the same key so each bit inside a block depends only on the bits inside the same block [23].
Advanced Encryption Standard(AES) which is used by Adiantum has a fixed block cipher of 16-bytes and the key can be either 16, 24 or 32 bytes [16].

FIGURE 3.1: Block Ciphers

Stream Ciphers encrypt bits individually and the main idea is adding a key stream to a plaintext bit. Stream Ciphers can also be divided into synchronous and asynchronous. The main difference between the two of them is that in asynchronous stream ciphers the key stream depends also on the ciphertext while in synchronous it does not.

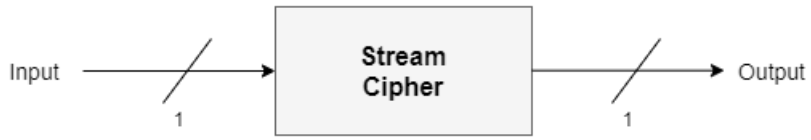

FIGURE 3.2: Stream Ciphers

Except for the two primitives of symmetric cryptography, Adiantum also uses Hash functions and Message Authentication Code(MAC) which are very popular in cryptography.

Hash functions can be perceived as the fingerprint of a message and are widely used to map an arbitrary size of message into a bit string of fixed length. The result is called hash value and is being used to index a hash table. Hash functions do not have a key.

Message Authentication Code(MAC) is also known as a cryptographic checksum, tag or a keyed hash function. In simple terms MACs can be correlated with digital signatures as they are used to authenticate a message, meaning to confirm that a message came without undergoing any change trough transmission. In order to be verified the same key is also needed in authentication.
Unlike digital signatures MACs are symmetric key schemes and much faster as they depend either on hash functions or block ciphers.

TABLE 3.1: Cryptographic Primitives Comparison.

Cryptographic Primitive	**Hash**	**MAC**	**Digital Signature**
Integrity	Yes	Yes	Yes
Authentication	No	Yes	Yes
Non-repudiation	No	No	Yes
Keys	None	Symmetric	Asymmetric

3.2 Adiantum

Adiantum or **HBSH** consists of the following parts

- **H**ash Function: Hash consists of Poly1305 [11] followed by NH [25]
- **B**lock Cipher: Single AES-256 invocation
- **S**tream Cipher: XChaCha12
- **H**ash Function: Hach consists of Poly1305 followed by NH

Adiantum [14] is a variable-input-length [7], tweakable block cipher [26] which can handle any message within the allowed range (128 to 2^{73} bits). The procedure needs the *Message-Plaintext*, *Cryptographic Key* and the *Tweak Key*.
Each stage of Adiantum demands a different size of key and for that purpose an extra instance of XChaCha12 is being used at the beginning to create all the keys [31]. More precisely with XChaCha a long random stream of 1136-bytes is created and as a result Adiantum has the following keys :

- 32-byte AES
- 16-byte Poly1305 for tweak
- 16-byte Poly1305 for message
- 1072-byte NH

Stream Cipher uses the same key both for the generation of the other keys and for the middle stage of Adiantum execution. As for the tweak key, in order to keep it simple it is similar to an initialization vector for a CBC mode (Cipher Block Chaining) or a nonce for OCB mode (Offset Codebook) [37].

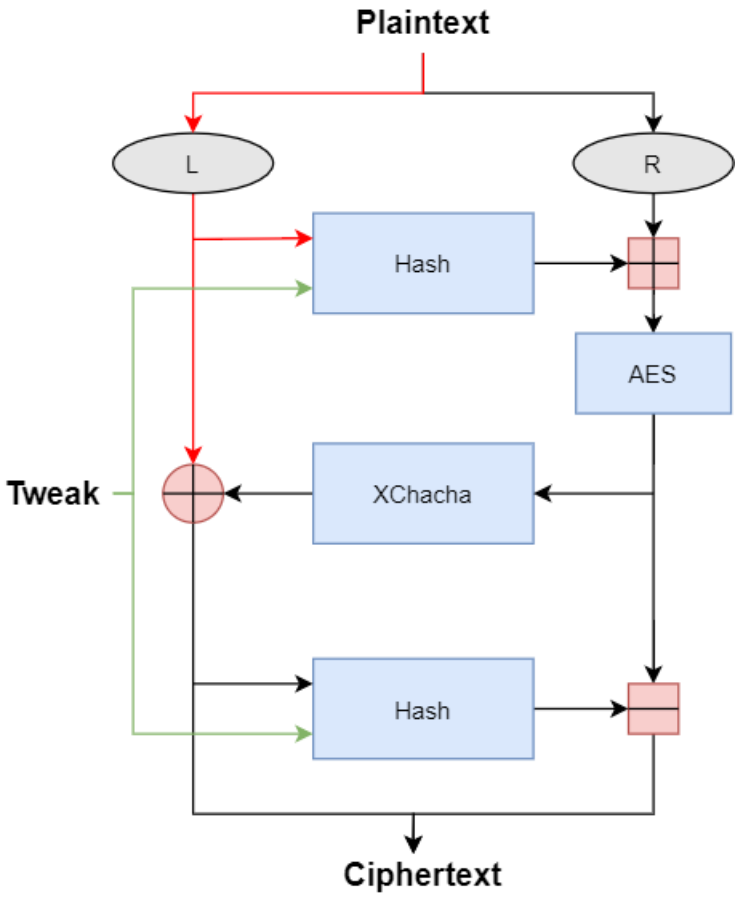

FIGURE 3.3: Adiantum

As seen in figure 3.3 the message (or plaintext) is split between Left and Right. Assuming a plaintext of x bytes, left part will be $(x - 16)$ as the last 16 bytes of message go to the right part. Also at the right part there are group operations of addition, subtraction whereas at the left a simple XOR operation with the plaintext.

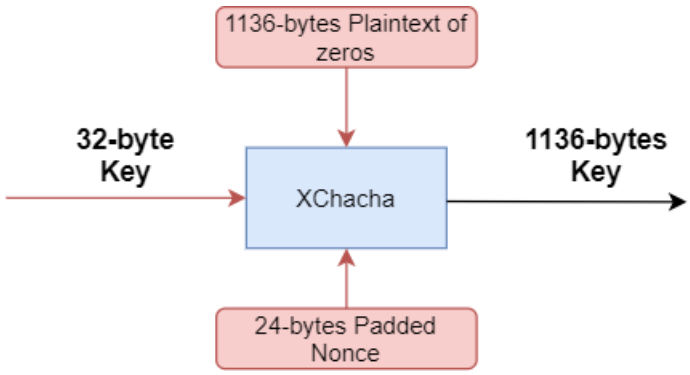

FIGURE 3.4: Adiantum Key Generation

Key generation needs a padded nonce of 24-bytes that consists of a bit with value one followed by zeros and a plaintext filled only with zeros just for the stream cipher.

3.3 Profiling

From a practical view Adiantum was designated by Google for disc encryption , meaning it was tested for up to 4096-bytes message for encryption and decryption. However, as already stated, from a theoretical view Adiantum can be used with a plaintext up to 2^{73} bits with the same key.

For these reasons profiling came up as a first step towards achieving acceleration. However, profiling has various meanings, and is sometimes misinterpreted as benchmarking. Therefore, in terms of profiling we want to measure the performance of the algorithm so as to narrow down where optimization would be more useful. In order to proceed with running the Adiantum algorithm, deep understanding of the python code was essential. The python code published by Google was not limited to the Adiantum logic described in the previous section but also runs many variations of it that were tested, resulting to the final version. Additionally, the published code runs different trials for Stream Ciphers(like Salsa, ChaCha, XChaCha) and Block Ciphers like AES. As a result, we had to minimize the path for running only the final version of Adiantum and also add numbers for bigger plaintexts than 4096 bytes.

For measuring Adiantum's performance on Intel-i5, a Windows operating system was used with an Integrated Development Environment(IDE) known as PyCharm [33]. PyCharm is an IDE specific for running python projects and the premium version which is free for academic students offers different profiling methods. In our case, the cProfiler method was used as it is the most common method for profiling Python code. cProfile is a built-in python module and using it within PyCharm results in a pstat file that gives us the following stats for every function inside the code:

- **Call Count**: Number of calls of the chosen function
- **Time**: Execution time of the chosen function plus all time taken by functions called by this function.
- **Own Time**: Own execution time of the chosen function.

Subsequently, we had to run and profile the Adiantum algorithm in the PS system of our FPGA. The first step was to establish the connection between Pynq and the PC. Through a Universal Serial Bus(USB) connection and a web based architecture of Pynq we were able to set up the communication. In order to handle and run our code, Pynq incorporates the open-source Jupyter notebook infrastructure to run an Interactive Python (IPython) kernel and a web server directly on the ARM processor of the Zynq device. The web server gives access to the kernel via a suite of browser-based tools that provide a dashboard, bash terminal, code editors, and Jupyter notebooks.

By utilizing the IPython kernel we transferred all of our files from our PC to the external memory(Secure Digital card) of Pynq board and were finally able to also run the code. Before measuring the performance of the ARM core we checked that the encryption and decryption were running without problems and that produce the same results as Intel-i5. It should be noted at this point, that for testing purposes, Google has provided a python code wherein a pseudo-random plaintext is created every time the algorithm is running based on the size of the plaintext. For a specific size of plaintext, the same plaintext is created every time and so it was possible for us to ascertain that encryption and decryption give the same results both at Intel-i5 and the ARM cores. The profiling again took place using the python's cProfiler module through the web-interactive terminal of Jupyter notebook.

The general use of cProfiler shows the following information through the terminal at the end of the algorithm execution:

- **function**: the function name
- **ncalls**: the number of calls for the specific function
- **cumtime**: Execution time of the chosen function plus all time taken by functions called by this function.
- **tottime**: Own execution time of the chosen function excluding calls to subfunctions.
- **percall**: how long each call took (However, gives an average value)

TABLE 3.2: Adiantum Profiling on Intel-i5 & Software only Execution on Pynq z1

	Intel-i5		Pynq-z1	
Plaintext(Bytes)	**Encrypt(sec)**	**Decrypt(sec)**	**Encrypt(sec)**	**Decrypt(sec)**
4,096	0.124	0.123	2.014	1.910
8,192	0.212	0.210	3.589	3.538
16,384	0.411	0.416	6.746	6.701
32,768	0.807	0.812	13.054	12.990
65,536	1.590	1.580	26.107	26.027
131,072	3.120	3.089	51.132	51.069
262,144	6.157	6.067	103.052	103.306
524,144	13.034	13.040	210.515	210.611
1,048,576	26.970	25.921	439.894	437.460
2,097,152	62.970	62.583	979.237	977.724
4,194,304	158.740	158.333	2,276.121	2,275.240
8,388,608	384.130	383.840	5,811.407	5,812.610
16,777,216	1,138.147	1,137.861	16,805.768	16,798.543
33,554,432	4,076.847	4,050.760	54,257.313	54,198.785

Finally, table 3.2 shows the specific values of Adiantum encryption and decryption time both for Intel-i5 and Pynq-z1 board for different sizes of plaintext as profiling gave us. The first observation from this table is that timing for Adiantum encryption and decryption is nearly the same and for obvious reasons for the rest of this thesis when encryption time is mentioned the same stands for decryption time too. The resulting time of encryption and decryption is reasonable as decryption has the reverse computations of encryption. Another thing worth mentioning here is the difference in time as the plaintext goes from 4KB to 4MB and finally to 32MB. Moreover, if we compare the encryption time between the Intel-i5 and the ARM cores, the latter is much slower than Intel-i5. Specifically for a plaintext of 4,096 bytes, Intel-i5 is 16x times faster than the ARM while at 32 MBytes is 13x faster.

The important question to be resolved is which of all the components that Adiantum utilises would actually take up the most time.

TABLE 3.3: Encrypt Profiling on Intel-i5 3rd generation.

Plaintext(Bytes)	Encrypt(sec)	Stream Cipher-XChaCha12(sec)	Stream Cipher-XChaCha12(%)
4,096	0.124	0.112	**90.32**
8,192	0.212	0.191	90.09
16,384	0.411	0.370	90.02
32,768	0.807	0.715	88.59
65,536	1.590	1.460	91.82
131,072	3.120	2.760	88.84
262,144	6.157	5.470	88.84
524,144	13.034	11.620	89.15
1,048,576	26.970	24.215	89.78
2,097,152	62.970	57.058	91.17
4,194,304	158.740	147.570	92.96
8,388,608	384.130	361.244	94.04
16,777,216	1,138.147	1,091.271	95.88
33,554,432	4,076.847	3,976.198	**97.53**

TABLE 3.4: Encrypt Profiling for Software only Execution on Pynq z1

Plaintext(Bytes)	Encrypt(sec)	Stream Cipher-XChaCha12(sec)	Stream Cipher-XChaCha12(%)
4,096	2.014	1.745	**86.64**
8,192	3.589	3.105	86.51
16,384	6.746	5.847	86.67
32,768	13.054	11.279	86.40
65,536	26.107	22.586	86,51
131,072	51.132	44.237	86.51
262,144	103.052	89.406	86.67
524,144	210.515	183.218	87.03
1,048,576	439.894	383.996	87.29
2,097,152	979.237	868.185	88.65
4,194,304	2,276.121	2,051.956	90.15
8,388,608	5,811.407	5,367.038	92.35
16,777,216	16,805.768	15,889.808	94,54
33,554,432	54,257.313	52,215.273	**96,23**

In the last two tables(3.3 & 3.4) we have the encryption time of Adiantum. The last two columns, Stream Cipher (in sec and %) show the time and the percentage respectively that XChaCha consumes of all the encryption operation. Despite the fact that Adiantum uses a combination of two hash functions twice, the results demonstrate that Stream Cipher takes up the most time. Specifically XChaCha takes about 86% of encryption time between 4KB and 1MB size of Plaintext and after that grow vastly up to 96% at 32MB.

By looking at these tables one could assume that the comparison of Intel-i5 and Pynq-z1 board seems meaningless as Intel-i5 is 16 times faster than the ARM cores. Regardless of how much we minimize the time needed for the stream cipher part by utilizing programmable logic possible acceleration, the remaining part that continues to run at the processing system is much slower than Intel-i5. Additionally, by calculating the Amdahl's law formula (3.1) the maximum theoretical speedup can be 7,14% - 25%.

$$MaxSpeedup = \frac{1}{1-p} \tag{3.1}$$

- **p**: task's portion that can be benefited by resource enhancement (fourth column of tables 3.3 & 3.4).

As a result, the best outcome for small plaintexts is that Pynq will continue to be slower than Intel-i5, whereas for bigger plaintexts Pynq will not be more than 3x or 4x times faster than Intel-i5. However, this thesis is the first scientific research based on Adiantum algorithm that attempts to discover if an FPGA can accelerate Adiantum and how this can be achieved. The comparison between Pynq and Intel-i5 is only taking place in order to evaluate the differences and the outcome of this thesis. Moreover, as already stated in the first chapter of this thesis the new hybrid models of CPU-FPGA present new opportunities and thus not limit us to systems with low processing performance.

3.4 Software

Evidently, after profiling analysis, Stream Cipher of Adiantum takes up most of the execution time. As the plaintext is getting bigger the same goes for the time needed for completion. Adiantum has already been explained but in order to go further in this thesis and its main purpose, an explanation of how the Stream Cipher(3.4.1) works comes as a necessity.
Adiantum uses XChaCha12. Specifically XChaCha [9] came from ChaCha [8] which is a variation of Salsa Stream Cipher [12]. ChaCha aimed to achieve better diffusion per round and performance with relation to Salsa. Either one functions very similarly and in bibliography they are referred to as XChaCha (8/12/20) [10]. This represents the number of rounds each will do hence the notation XChaCha12. More rounds result in better security but more execution time.

3.4.1 XChaCha12

XChaCha12 consists of the following:

- ChaCha: The usual algorithm
- HChaCha: Intermediate step before the procedure goes to ChaCha

ChaCha Algorithm: Initial State

- Constant: 16-bytes which is "expand 32-byte k"
- Key: 32-bytes
- Counter: 8-bytes
- Nonce: 8-bytes

Constant	Constant	Constant	Constant
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Counter	Nonce	Nonce

FIGURE 3.5: Initial State Of ChaCha

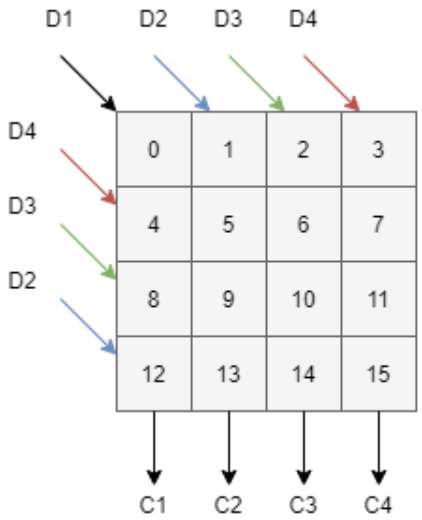

FIGURE 3.6: Double Round of ChaCha

The procedure is rather simple if broken down into stages. The initial state of Chacha is iterated in six loops, that is from zero to twelve with a step of two. Each one of the six loops in XChacha12 first accesses the columns(C) and then the diagonals(D). This dual workload justifies why every loop is referred to as a Double-Round(3.6). Furthermore, each double-round consists of a total of eight Quarter-Round(QR) calls, four for the columns (C) and four for the diagonals(D).
A graphic explanation in figure 3.6 demonstrates the course of action in the case of a diagonal. As to which elements form a diagonal, consider that (D2) is formed by boxes (1,6,11,12). Each Quarter-Round function includes a number of operations that will be subsequently analysed in depth.

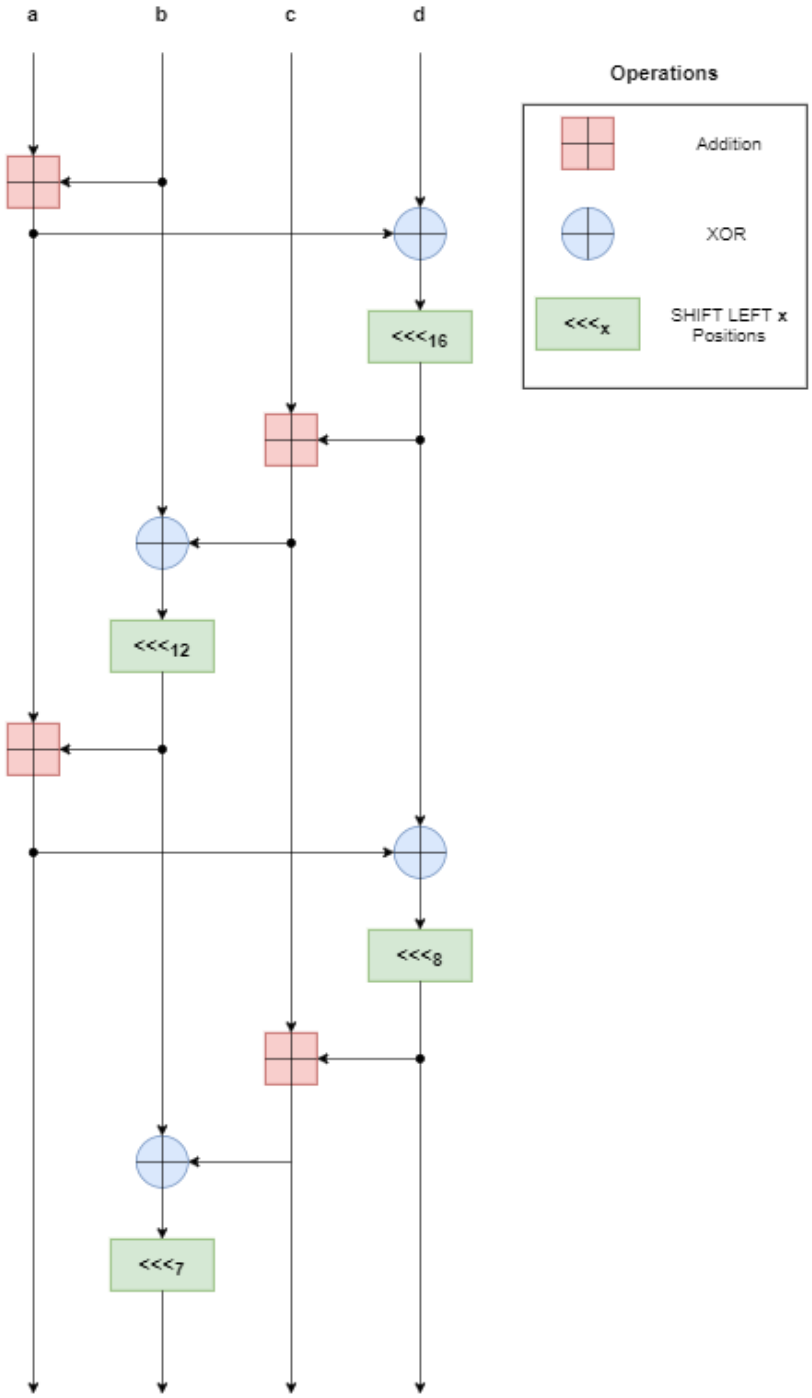

FIGURE 3.7: Quarter Round of ChaCha

In conclusion Chacha12 works in rounds and each round computes eight Quarter-Round(3.7) four through each column and four through each diagonal of the state table. Consequently, the state table changes with each call of Quarter Round. At the end, the result is obtained by adding each block of the initial table to the final version.

HChaCha: Intermediate State

HChaCha uses nearly the same initial table as ChaCha, meaning:

- Constant: 16-bytes which is "expand 32-byte k"
- Key: 32-bytes
- Nonce: 16-bytes

In HChaCha nonce is 16-bytes and the table of block counter takes the first 8-bytes of the 16-bytes nonce. After this initialization the procedure goes through ChaCha rounds as usual. When the ChaCha rounds completes the first 16-bytes and the last 16-bytes of the state table, both in little-endian(3.4.2) format, are concatenated creating a 32-bytes subkey.

Finally **XChaCha** is constructed by ChaCha and HChaCha [15]. More specifically HChaCha is used at first for creating the subkey in ChaCha algorithm. Also it is of essence to mention here that when Adiantum was explained a 24-bytes nonce was mentioned. The first 16-bytes are for the HChaCha step and the last 8-bytes for the usual ChaCha algorithm.

3.4.2 Little and Big Endian numbers

Computers have two different ways of storing data [3]. The method is called endianness and is distinguished in "Little Endian" and "Big Endian". In simple terms big endian can be thought of as a human who begins reading a line from left to right and the other way around for little endian.
When a machine stores data in big endian order the most significant bytes are stored in the first memory location and all the other bytes follow. On the contrary, in little endian machines the least significant bytes are stored in the first memory location up to the most significant bytes that are stored in the last one. Notably, despite the endianness inside each byte, bits are in big endian order.
A major advantage of little endian order is that least significant bytes do not change positions as more digits are added to higher addresses making some operations much faster.

Supposing that an integer is saved as 4 bytes then a toy example of a variable 0x0123456776543210 is in figure 3.8 where letter (a) represents the first memory location.

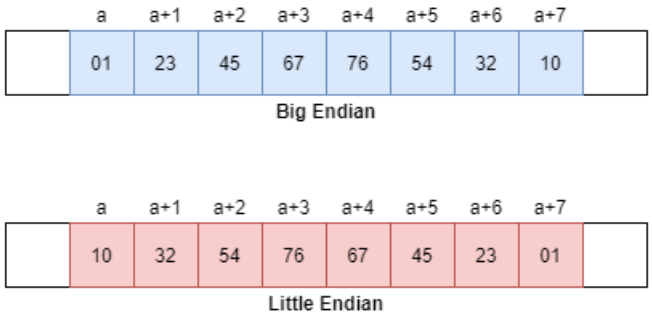

FIGURE 3.8: Example of Endianness

XChaCha algorithm functions with little-endian logic and as in all the related work, we decided to keep this approach for our practical implementation and for academic learning.

3.4.3 Software acceleration of XChaCha12

In this thesis apart from hardware logic which will be analysed in the next sections, software changes have also had a great impact and therefore need to be mentioned.

Algorithm 1 XChaCha Encryption

```
def encrypt(plaintext,offset=0,key,nonce):
    result = []
    while plaintext:
        stream = gen_output(key,nonce,offset)
        offset += 1
        result.append(bytes(x^y for x, y in zip(stream, plaintext)))
        plaintext = plaintext[len(stream): ]
    return b''.join(result)

# Here we have a simple format of what gen_ouput() do
def gen_output(key,nonce,offset):
    subkey = HChaCha(key,nonce[0:16])
    return ChaCha(key=subkey,nonce[16:],offset)
```

Algorithm 1 shows that XChacha encryption operates in a while loop. For every 64-bytes of the plaintext gen_output() is called and returns 64 bytes which are the result of XChaCha12. After the end of gen_output(), as is common with most stream ciphers, an XOR operation between the result and the plaintext chunk is taking place. Finally, plaintext is shifted for the next loop. Moreover, offset is the block counter of XChaCha state(3.5).

Although instinctively one can deduce that gen_output() takes up the most time, profiling revealed that encrypt() is actually the most time-consuming function without taking gen_output() into consideration. An initial approach was based on the idea that python needs to perform XOR operations byte per byte between the plaintext and the result of XChaCha and naturally the bigger the plaintext size, the more XOR operations are needed. Furthermore, an essential modification for the XOR operation is to convert the byte variable into an integer variable. Taking everything into consideration, initially we tried to exploit the hardware in order to perform XOR operations for the whole chunk and many chunks simultaneously. Unfortunately, after a number of different approaches it was clear that the XOR operation was not the part that consumes the greatest amount of time. After these trials it was obvious that we had to do deeper profiling in order to understand which lines of the code consume the most time and why. In order to achieve a deeper profiling of the execution times for each function the kernprof library for line-by-line profiling was used and specifically for the function of the stream cipher part of Adiantum. Eventually, line-by-line profiling showed that Python libraries for simultaneously shifting and deleting bytes take up a lot of time depending on the size of the plaintext.

Apart from the changes based on how Python operates, another important thing to consider is that gen_output() which is called every time inside the while loop computes the subkey from HChaCha step every time. There is no reason for computing the subkey again as the key and nonce do not change until the next message. Therefore the subkey needs to be calculated for ChaCha cipher only the first time that gen_output() is called.

Algorithm 2 XChaCha Encryption with Software Changes

```
def loopsforwhile(number):
        if number // 64 == number / 64:
            loops = number // 64
        else:
            loops = number // 64 + 1
        return loops

def encrypt(plaintext,offset=0,key,nonce):
    result = []
    numofloops = loopsforwhile(len(plaintext))
    for i in range(numofloops):
        stream = gen_output(key,nonce,offset)
        if i == numofloops - 1:
            plain = plaintext[i * 64:]
        else:
            plain = plaintext[i * 64:(i * 64) + 64]
        offset += 1
        result.append(bytes(x^y for x, y in zip(stream, plain)))
    return b''.join(result)

def gen_output(key,nonce,offset):
    if offset == 0:
        subkey = HChaCha(key,nonce[0:16]    #subkey is global variable
    return ChaCha(key=subkey,nonce[16:],offset)
```

Algorithm 2 resolved all previously mentioned issues concerning subkey and Python bytes shift. There is one extra function loopsforwhile() which calculates how many loops encrypt() needs depending on the size of each plaintext. Variable plain is just taking 64-bytes of plaintext each time until the last one yet possibly plaintext has fewer bytes. Finally, subkey in gen_output() is being calculated only the first time, thus subkey is a global variable in order to keep its value for the next calls of gen_output(). Apart from these modifications XChaCha12 cipher still consumes the most time while plaintext gets bigger and therefore hardware shall aid in achieving acceleration.

3.5 Hardware Approach

Software modifications definitively improved Adiantum but the real target from the beginning was to accelerate it utilising hardware tools that an FPGA can offer. This section serves as a theoretical overview of what is implemented using programmable logic(PL) and why, and what will remain in the processing system(PS).

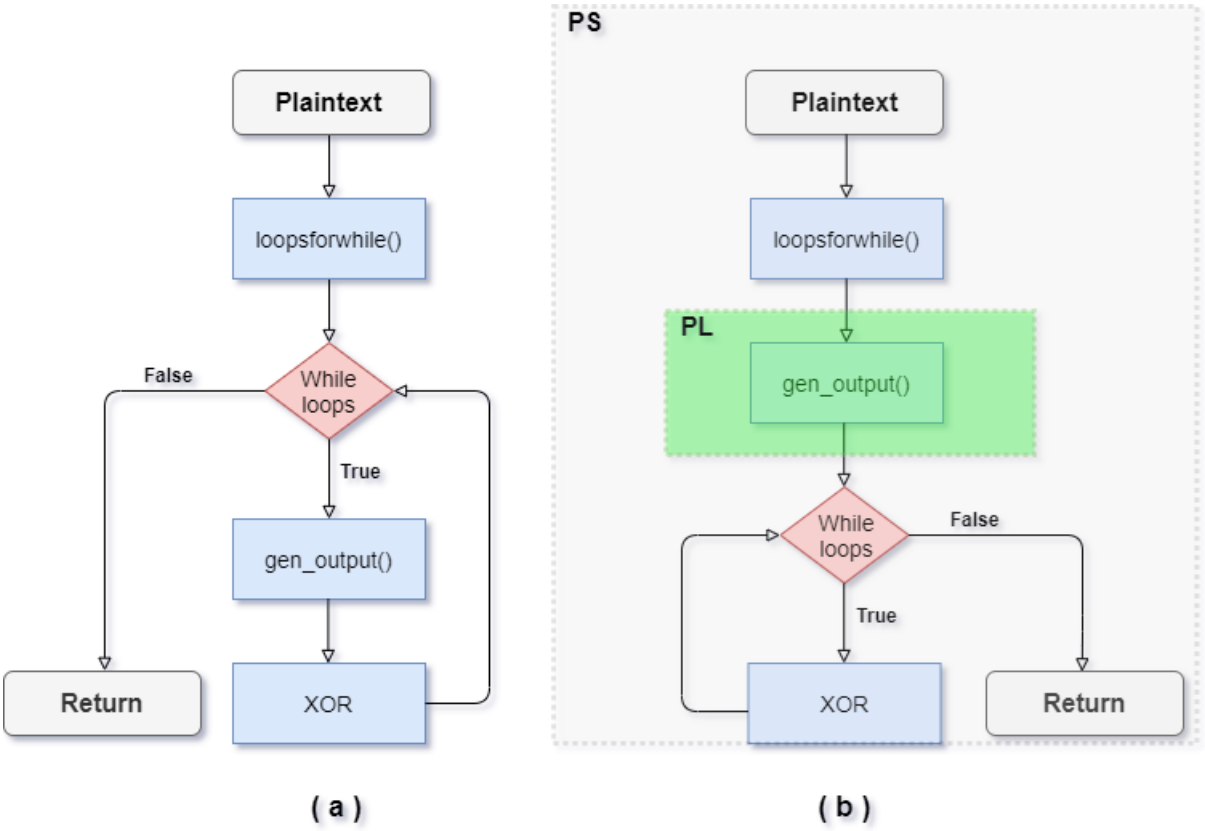

FIGURE 3.9: (a)XChaCha Encrypt (b)XChaCha Encrypt(PL-PS)

Figure 3.9 demonstrates at the left(a) a very simple flowchart of how the XChaCha encrypt() function works and at the right side(b) how it is modified and which parts are implemented either in PL or PS of our FPGA. Although XChaCha still takes up the most time, the difference is that now time is being consumed at gen_output() function. This is reasonable as this function contains nearly the whole logic and computations of Adiantum stream cipher. Despite subkey is being calculated only once, the main issue is that for every chunk of 64-bytes of plaintext the usual ChaCha logic has to be repeated. For a very big plaintext this leads to many cycles as for every chunk

of bytes the block counter is increased and the initial table of ChaCha has to be reinitialized and run through all computations.

By utilizing hardware benefits such as pipeling and many others, gen_output() function can be improved impressively. The basic idea of using pipeline can be best comprehended by thinking of the target function as a counter. For every increment a series of mathematical computations takes place. There is no need, however, for these calculations to be completed in order for the counter to be incremented. Consequently, the ChaCha algorithm inside gen_output() has been implemented in the PL of the ZYNQ device.

Notably, we have chosen to keep the xor part at the PS system so as to not consume time in communication between PL and PS. Had we decided to include the xor part, then every run of ChaCha in Pl would have also needed 512-bit of the plaintext. By implementing it as such, the PS can now send one stream at the PL and receives many 64-bit chunks of ChaCha algorithm each time. Moreover, as communication between PL and PS is limited to an upper limit of bytes, the need for the plaintext from PL would have also resulted to receiving about half of the data that PL now sends to PS. Also, due to the different trials for implementing only the xor part at PL as we mentioned in the previous section and after the deeper profiling with kernprof library took place, it was clear that the xor part consumes an insignificant amount of time.

Furthermore, it is important to clarify that ChaCha algorithm is also executed for generating the subkey through the HChaCha step though as it runs only once there is no need for that part to be transferred to PL. With the implementation of ChaCha algorithm using PL, gen_output() is called only once inside the encryption part and returns a list of all the required stream chunks for the appropriate length of each plaintext.Further details of the hardware implementation will be analyzed in the next chapter of this thesis(4).

Chapter 4

FPGA Implementation

4.1 Top-Down Strategy

A top-down approach of the implementation is analyzed in the following sections. Using the top-down method the explanation begins from the general/big picture and as the explanation goes further inside/deep information is shown and explained.

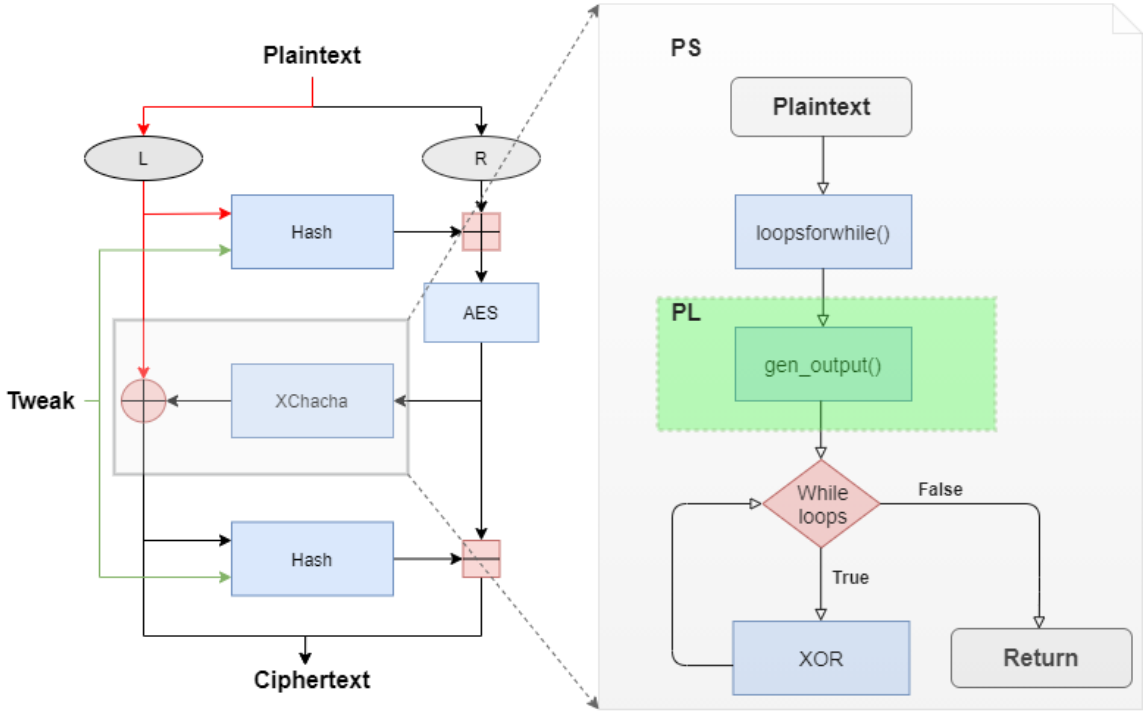

FIGURE 4.1: Adiantum - PS & PL

Figure 4.1 shows the part of Adiantum that has been changed in order to utilize the hardware benefits. As already explained in the last chapter, the most time consuming part of Adiantum is the stream cipher part (XChaCha12). Additionally, by leaving the xor part at the PS we have the benefit of not sending much data to PL but also using all hardware transfer limits for the receiving part from PL to the PS. The upper hardware limit for a single transaction between PL and PS is better explained at the DMA configuration in the next subsection.

The analysis of top-down strategy consists of the following sections:

- **Pynq Configuration & Software Changes:** This section describes the changes required by the software implementation for incorporating our ChaCha12 design within the whole Adiantum process.
- **Vivado Hardware Design:** Here all Vivado configurations for connecting processing system to programmable logic are discussed and analyzed.
- **IP Implementation with HLS:** The last section describes all details about the ChaCha12 implementation and examination of all the pragmas that were utilised for achieving hardware acceleration.

4.2 Pynq Configuration & Software Changes

Algorithm 3 XChaCha Encryption for using Hardware implementation

```
def loopsforwhile(number):
        if number // 64 == number / 64:
            loops = number // 64
        else:
            loops = number // 64 + 1
        return loops

def encrypt(plaintext,offset=0,key,nonce):
    result = []
    numofloops = loopsforwhile(len(plaintext))
    stream = self.gen_output(offset, numofloops, key, nonce)
    for i in range(numofloops):
        if i == numofloops - 1:
            plain = plaintext[i * 64:]
        else:
            plain = plaintext[i * 64:(i * 64) + 64]
        offset += 1
        result.append(bytes(x^y for x, y in zip(stream[i], plain)))
    return b''.join(result)
```

Algorithm 3 shows how the encrypt() function has been changed in comparison to algorithm 2. The main difference is that the call for gen_output() function is outside of the for-loop and now it is called only once. Also, an extra parameter (numloops) has been added to the call of gen_output and based on that the gen_output() function will know how many chunks of ChaCha runs are needed for each size of plaintext. The stream value is now a list of object type variables of 64 bytes. The 64-byte variables are the chunks/results from the ChaCha algorithm that our hardware implementation returns to PS system.

Algorithm 4 Gen_Output() for using Hardware implementation

```
overlay = Overlay("./ChaChaIP.bit")
dma = overlay.my_axi_dma_engine
ChachaMax = 1000000
def hard_loops(number):
        if number // ChachaMax == number / ChachaMax:
            loops = number // ChachaMax
        else:
            loops = number // ChachaMax + 1
        return loops

def gen_output(offset, softloops, key, nonce):
        subkey = ks.hash(key=key, nonceoffset=nonce[:nl])
        result = []
        if softloops > ChachaMax:
            hardloops = self.hard_loops(softloops)
        else:
            hardloops = 1
        for i in range(hardloops):
            start = i*ChachaMax
            input_buffer = allocate(shape=(1,), dtype='S64')
            if i == hardloops - 1:
                endoff = start + softloops
                output_buffer = allocate(shape=(softloops,), dtype='S64')
            else:
                softloops = softloops-ChachaMax
                endoff = start+ChachaMax
                output_buffer = allocate(shape=(ChachaMax,), dtype='S64')
            buffer = endoff+start+nonce[nl:]+subkey
            # after converting buffer to object type 'bytes'
            input_buffer[:] = np.array(buffer)  # transfer() takes ndarray

            dma.sendchannel.transfer(input_buffer)
            dma.recvchannel.transfer(output_buffer)
            asyncio.get_event_loop().run_until_complete(
                asyncio.ensure_future(dma.sendchannel.wait_async()))
            asyncio.get_event_loop().run_until_complete(
                asyncio.ensure_future(dma.recvchannel.wait_async()))
            result.extend(list((output_buffer).copy()))
            del input_buffer,output_buffer
        return result
```

Algorithm 4 has changed a lot with relation to its previous version in algorithm 2. As already explained pynq uses programmable logic circuits as hardware libraries that are called overlays. By calling the function Overlay() with the bit file that has been created through Xilinx Vivado IDE as a parameter, each PL implementation can be loaded to the FPGA. After the PL implementation has been loaded, there are ready-made functions from the Pynq community in order to make the communication between PS and PL feasible.

The first thing that has to be analyzed is the new function hard_loops(). The encrypt() function sends the number of ChaCha chunks required by Gen_Output(). However, due to hardware limitations (AXI DMA configuration) each transaction between PS and PL can be up to 2^{26} bytes or about 64 MB. Therefore, the hard_loops() function is used to calculate for how much time the PL logic has to be called. The number of one million has been calculated based on the 2^{26} limitation as each run of ChaCha results in 64 bytes, then one million runs of ChaCha will result in 61 MB. So the best case scenario is to send 64bytes to PL and receive one million results of ChaCha algorithm.

For every transaction between PL and PS, the input_buffer and output_buffer have to be initialized. By using pynq's allocate() function, the specification for the amount of memory that PL needs is allocated. The input_buffer contains the data which will be transmitted through DDR memory to AXI DMA and respectively the output_buffer receives the data from AXI DMA towards PS. After initialization takes place, the input_buffer can take data and through pynq function dma.sendchannel.transfer(input_buffer) the data can be transferred with AXI master (HP or ACP) ports from DDR memory to the DMA.

The initialization block of XChaCha requires the subkey, the constant, the block counter, and the nonce. Constant is like an authentication that the design is free of intentional harware backdoors which is why it is hard-created within the design. The buffer variable takes the subkey, the nonce, and the numbers of the block counter for the first and last chunk of ChaCha that PL will return. By sending both the first and the last block counter numbers, the PS can communicate again with the PL for getting the next one million or less results. Finally, by using both the python library of asyncio and pynq function wait_async(), the PS can continue to run as PL operates until an interrupt signal is triggered from DMA and sent back to PS.

4.3 Vivado Hardware Design

In general, data has to pass to DDR of the processor and after that there are three ways for sending them through PL.

- **Streaming(AXI-4)**: Streaming method uses AXI-4 protocol and can be implemented with Xilinx's IP(AXI DMA). By utilizing this IP, a continuous bus is being created and without requests data is travelling through it as in a large FIFO. Given that the requests part does not exist, no time is consumed into that part and, therefore, more space is acquired for the pipeline by covering the DDR interval.

- **B-RAM**: BRAMs are used for storing data inside an FPGA and usually they are between many KB and a few MB. The process here goes in bursts. Data is transferred either as stream or memory mapped in chunks and is saved in BRAMs exploiting their huge bandwidth. In order to achieve this approach, data must have small memory footprint. Lastly, there are three configurations of BRAMs, namely single/double port and FIFO.

- **Memory-mapped I/O(MMIO)**: Another approach is MMIO which performs I/O between CPU and peripherals of an FPGA. It can be implemented through Vivado IDE by using Xilinx's IP known as Data-Mover. In principal MMIO method uses the same addresses both for memory and I/O devices as they are mapped together. Since it uses random accesses it cannot drive many requests at the same time therefore a lot of time(30 to 50 cycles) is being consumed at each request. Considering this, there is no reason for this approach to be implemented in a design that targets acceleration and methods like pipelining.

Given that our target is to accelerate Adiantum, the streaming method is more efficient and by utilizing it we can take advantage of the whole DDR high bandwidth memory(HBM). A very simple block diagram follows just to demonstrate the main idea of how connections and data transfers are taking place.

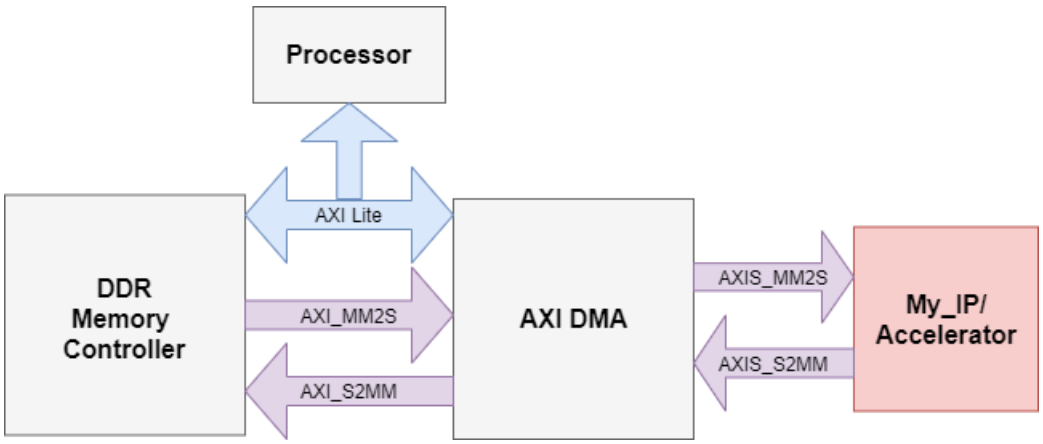

FIGURE 4.2: PS-PL Block Diagram

Figure 4.2 shows how the accelerator is connected with ZYNQ-PS. Processor and DDR Memory Controller are placed inside Zynq PS whereas AXI DMA and our IP are implemented in Zynq PL. The first thing to be mentioned is that through AXI lite bus, the processor can communicate with the AXI DMA. It resembles a human brain as in the case of DMA it controls the setups, initiate and monitor data transfers. Aside this, AXI Memory-Mapped to Streaming(AXI_MM2S) and AXI Streaming to Memory-Mapped(AXI_S2MM) make it possible for DMA to communicate or, better yet, access DDR memory for fetching and returning data. Afterwards DMA takes the data and streams them using AXI4-streaming buses and specifically AXIS_MM2S for sending the data and AXI_S2MM for receiving data. One major difference between AXI and AXIS is that in streaming mode(AXIS) the addresses are not being used.

AXI DMA is the key for communication between PL and PS. By taking data from DDR memory and transferring them to PL, a software algorithm can be further optimized. Besides everything concerning hardware utilization and the way it works, it should be noted that while PL logic is in use, PS can still continue. The outcome is that PS and PL can work in parallel until PL is finished, which can also occur with an interrupt signal from DMA to PS, thus ensuring even more precious time. The configuration of AXI DMA is essential for a design to function properly. In our case the upper limit of 2^{26} bytes for a single transaction is used as well as the data width of 512bits both for the Read and Write Channel.

In addition to the previous block diagram, a more precise block design is provided with the appropriate signals and blocks.

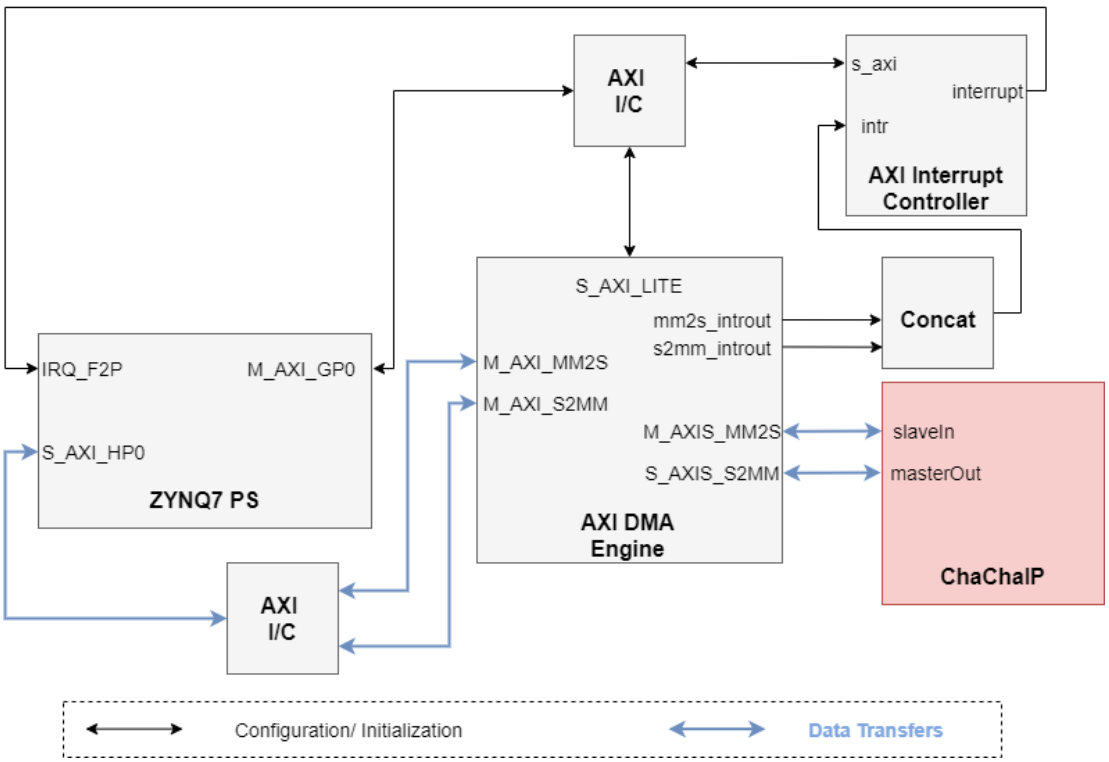

FIGURE 4.3: Vivado Block Design

Figure 4.3 adds more detail to the previous one 4.2. At first we see two AXI Interconnects(AXI-I/C) and IPs which are responsible for Interrupts(Concat and AXI Interrupt Controller). In addition to the blocks, some ports and their names are now visible as well.

In PS, High Performance Port(HP0) is enabled for data transferring from PS to PL and vice versa and from fabric interrupts the first PL-PS interrupt port bits have been enabled. HP ports support high rate communications between PL and memory elements in the PS. Zynq devices have four of these ports but in our case only one is needed. Moreover these ports contain FIFOs for read and write data in burst mode. Data width can be either 32 or 64 bits(in our case 64bits) but PL always acts as the master of these communications.

Apart from PS, from the perspective of PL logic there are two things to be mentioned. Firstly, at DMA engine signals mm2s_introut and s2mm_introut

are shown, which are used for interrupts. Secondly, Concat block and AXI Interrupt Controller connect interrupts between PL and PS.

Concat block combines signals of different width into a single bus. Typically a designer can connect the out port of concat block directly to PS. However it is more accurate to nest AXI interrupt controller between them and for that reason pynq-z1 demands it. AXI Interrupt Controller concentrates multiple interrupt inputs from peripheral devices to a single interrupt output to the system processor.

A further clarification of what these buses transfer is needed for understanding the connection between the Vivado Block Design (figure 4.3) and the algorithm (4). The HP port is used to transfer the input data (buffer data in algorithm 4) from DDR memory of PS to AXI DMA of PL and through it to our ChaChaIP design. However, PS sends and receives packets of 64 bytes while the HP port can transmit/receive either 32 or 64 bits (in our case 64 bits). Therefore, the option of transferring in burst mode offered by AXI I/C has been enabled and as a result, the Vivado auto-configures FIFO implementations for holding the data until the desirable size of data has been received/sent. In our case, the desirable size is 512 bits data both for receiving from PS and sending to PS.

After AXI I/C receives the 512 bits from PS, they are sent to the DMA and through M_AXI_MM2S are finally transferred to our ChaChaIP design. When ChaChaIP receives the data the process explained in the next section is initiated. Our design sends the ChaCha results as streams of 512 bits per stream through masterOut to S_AXIS_S2MM of DMA. The received data are converted from a stream to memory-mapped and sent from DMA (with M_AXI_S2MM bus) to AXI I/C and finally to the PS (with S_HP_HP0) port.

4.3.1 Multiple Clocks Configuration

It is essential before we move forward to how our IP is implemented, to mention that in the above figure 4.3 there are two different clock frequencies for PL logic. One clock at 100MHz is for configuration of AXI I/C, Concat, etc. and the other clock at 150MHZ is for our IP/Accelerator and related connections with AXI DMA engine.

By using two clocks there are some pros and cons depending on the final target of each design. The apparent advantage is that by using a faster clock, design will also run faster and for that reason we have used 150MHz for our IP. On the other hand, more hardware resources need to be consumed for the implementation on FPGA. Considering our thesis target is acceleration and given that our IP does not need many resources we chose to implement this way. Notably these clock frequencies were not selected randomly. Basic frequency for PL logic for Artix-7 and Speed Grade(-1) where our FPGA belongs is at 100MHz and after testing our design with it, we increased to 150Mhz which is the maximum possible frequency. Finally as AXI-Lite clock can be up to 120MHz, our AXI DMA works in asynchronous mode.

4.4 IP Implementation with HLS

The next step of our design was the implementation of the accelerator and more specifically the ChaCha algorithm. In order to convert this part from Python in C++ language, a deep knowledge of how Adiantum uses ChaCha and the mathematics was deemed necessary. Vivado HLS was used for this purpose after having previously understood its tools.

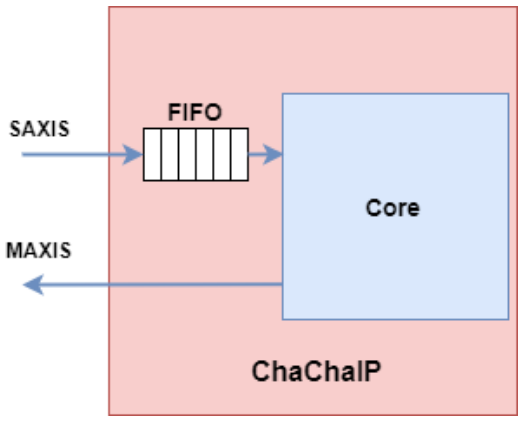

FIGURE 4.4: IP Block Design

Figure 4.4 shows the basic concept of communication which our IP uses with DMA. Two axi streams have been used, one as a slave and the other one as a master for data transferring. As data comes inside our IP block a stream FIFO is implemented for keeping and transporting data to Core function. By utilizing a stream FIFO, each arrival of data from DMA will be kept until they can be processed by Core so as to not get lost. It is essential to mention that both input and output of the accelerator use a stream object.

Stream is a Vivado HLS construct and in simple terms it makes an interface through which data is exchanged in streaming manner. In bibliography stream can be implemented either as a FIFO or a shift register. Due to stream being a template class, it was used in combination with a struct which is an easy way to group variables of different size or type into a single one. The same struct containing two ap_uint<> variables has been used both for input and output. Vivado HLS also offers some C-based data types which helps hardware designers to construct variables of specific size like ap_uint<x> where x is the number of bits.

In this case stream struct contains a variable of 512 bits for receiving or sending out data and a variable of one bit which represents the TLAST signal of AXI protocol. The TLAST signal is fundamental for the proper function of the whole design as it informs the receiver (in this case AXI DMA) that communication is almost terminated.

The main function of our IP uses Pragmas for converting input and output variables into AXI protocol interfaces and DATAFLOW. By using Dataflow pragma in the main function, Vivado HLS is instructed to execute in parallel all sub-functions contained within main. In this case FIFO implementation and Core are executed in parallel as they are part of main function.

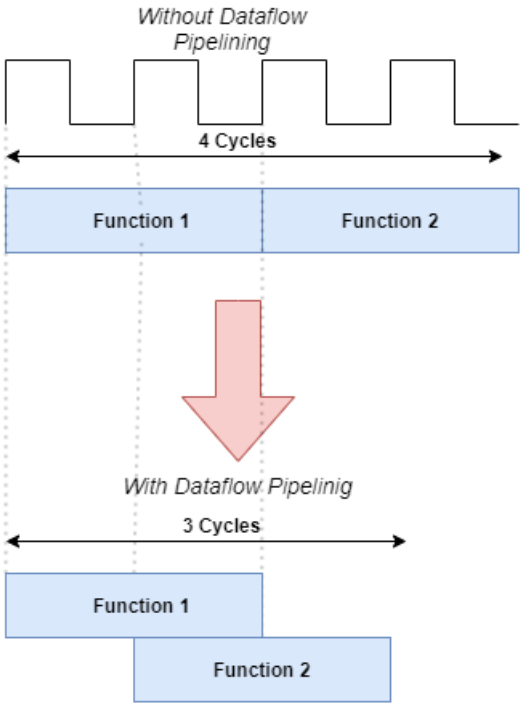

FIGURE 4.5: Dataflow Example

4.4.1 Core Function Analysis

A simple flowchart of how the Core function operates is shown below.

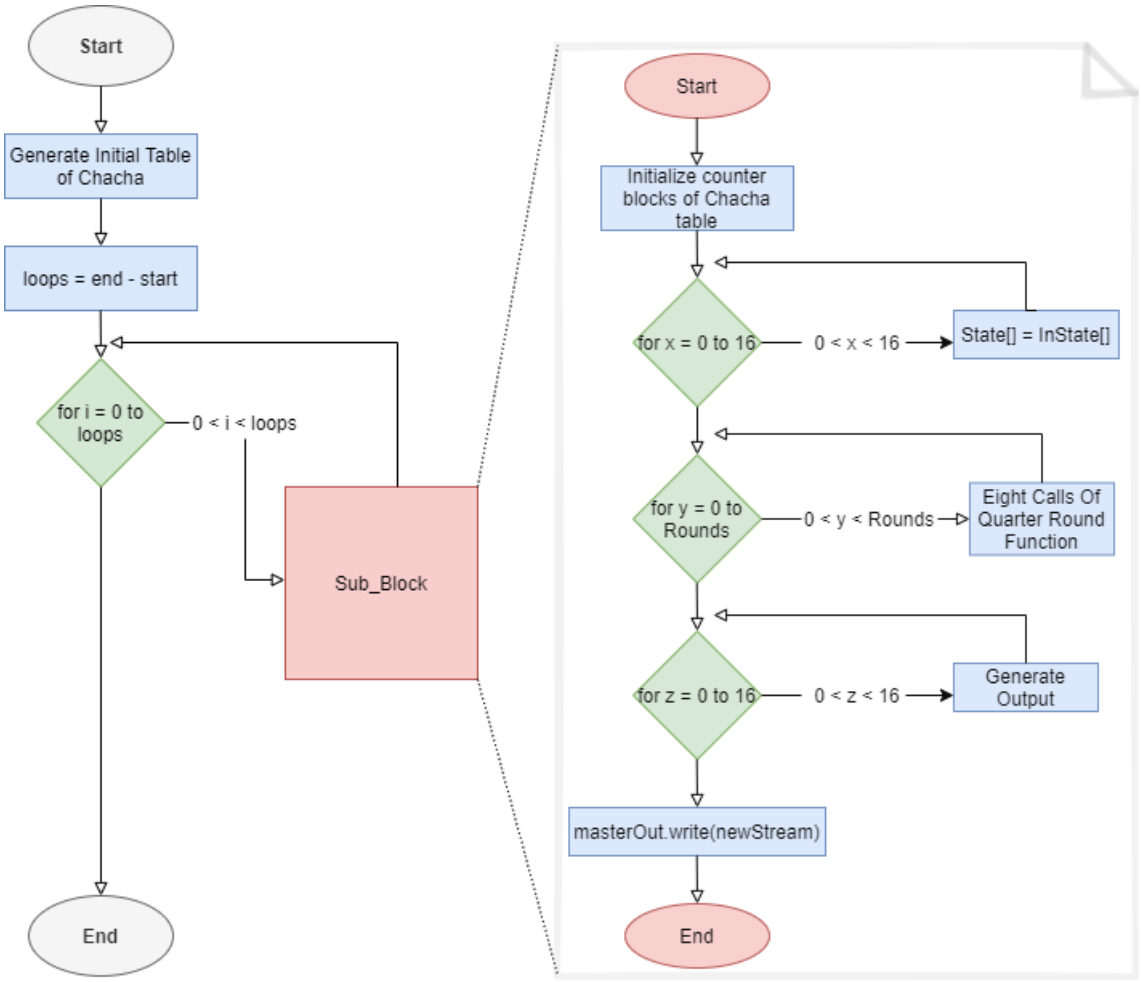

FIGURE 4.6: Core Function Flowchart

Figure 4.6 is one flowchart shown in two parts. The way Core function works is shown on the left side, whereas on the right side there is the flowchart of what is taking place inside the Sub_Block. Additionally, besides the Quarter Round function which will be explained later, there are also LITTLE_INT() and MOD_OV() functions which are not demonstrated in the figure. These functions are not at all complicated yet each one of them is requisite. As already explained(3.4.1) the initial table of Chacha algorithm is consisted of 32-bit words. However these words must be in little endian order(3.4.2).

Consequently, LITTLE_INT() converts 32-bit words from big to little endian and vice versa. Finally, MOD_OV() is just to ensure that whenever an add operation takes place there is no overflow bit and our variables will remain 32-bit.

Before proceeding with more details about figure 4.6, pragmas/directives that are used need to be further analyzed.

Array Partitioning

HLS offers this feature to counterbalance the disadvantage of B-RAMs having only two memory channels which results in memory access limitations. By using array partitioning a B-RAM array can be separated into many parts. There are three ways of partition:

- **Cyclic**: Create smaller arrays and more precisely the array is partitioned cyclically by putting each element into each new array before returning to the first array. This process is repeated until the array is fully partitioned.
- **Block**: Combining with a factor N separates the original array into smaller arrays of size N.
- **Complete**: Finally all elements of the array are separated into different variables.

In order to achieve separation multiplexers are utilized for converting the arrays. In our case complete mode was used for state[] & inState[] arrays and as a result the algorithm can have multiple accesses to these arrays at the same cycle resulting in better throughput.

Pipeline & Unroll

Pipeline directive is one of the most important features that FPGAs offer. The optimal pipeline can be achieved by interval(II=1) in which case every cycle the process can take a new input. In the occasion of a loop every cycle a new iteration can take place. Apart from pipeline, unroll directive can also help significantly with loops. It can be specified by the designer but it is also an automated configuration from HLS when the pipeline directive is used. When a loop is being unrolled either partially or fully this is translated into many copies of loop body in the RTL and therefore the entire loop can run

simultaneously. In our case pipeline pragma is used within the main for-loop of the Core function. For Sub_Block this was translated in fully unrolling of the first and last for-loops and partial unroll with a factor 4 of the middle for-loop which holds the eight calls of Quarter Round function. The middle loop could not be fully unrolled as the last four calls(diagonals) of QR function demand that the first four have already completed.

Algorithm 5 Vivado HLS: Double Round

```
1:  uint32  state[16]
2:  #pragma  HLS  array_partition  variable = state  complete
3:
4:  for i ← 0, i < Rounds, i ← i + 2 do          ▷ QR function take pointers
5:      #pragma  HLS  unroll  factor = 4
6:      QR(state[0], state[4], state[8], state[12])        ▷ column 1 (figure 3.6)
7:      QR(state[1], state[5], state[9], state[13])        ▷ column 2
8:      QR(state[2], state[6], state[10], state[14])       ▷ column 3
9:      QR(state[3], state[7], state[11], state[15])       ▷ column 4
10:
11:     QR(state[0], state[5], state[10], state[15])  ▷ diagonal 1 (main diagonal)
12:     QR(state[1], state[6], state[11], state[12])       ▷ diagonal 2
13:     QR(state[2], state[7], state[8], state[13])        ▷ diagonal 3
14:     QR(state[3], state[4], state[9], state[14])        ▷ diagonal 4
15: end for
```

Algorithm (5) shows a simplified example for the use of array_partition and pipeline logic for Double Round. The use of the array_partition pragma for the state[] table allows for multiple accesses to it at the same cycle. Also, with the addition of unroll pragma, this code results in unrolling with factor 4 the first four QR calls(lines 5..8) and the last four calls(lines 10..13). In fact, this for-loop is inside another bigger loop that consists of the pipeline pragma which results in unrolling these QR calls, but also achieving interval II=1. Interval one means that the whole logic runs in parallel and gives results in every cycle.

In order to further demonstrate the incredible hardware benefits and specifically array partitioning and unrolling, a visual example of them based on the double round is shown bellow.

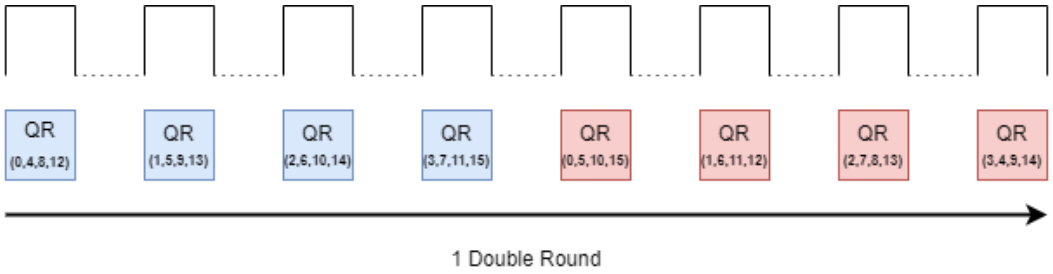

FIGURE 4.7: Double Round Time-Chart: Without Unrolling & Array Partitioning

Figure 4.7 shows how Vivado HLS translates the general C/C++ code of Double Round(blue color: QR calls of columns & red color: QR calls of diagonals) without the appropriate pragmas. The result will not differ at all when the same code runs at the PS system. Assuming that each QR call takes one cycle to complete, then an implementation like this will take 8 cycles for one Double Round and as a result, 6 Double Rounds for the ChaCha12 algorithm will consume $6 * 8 = 48$ cycles.

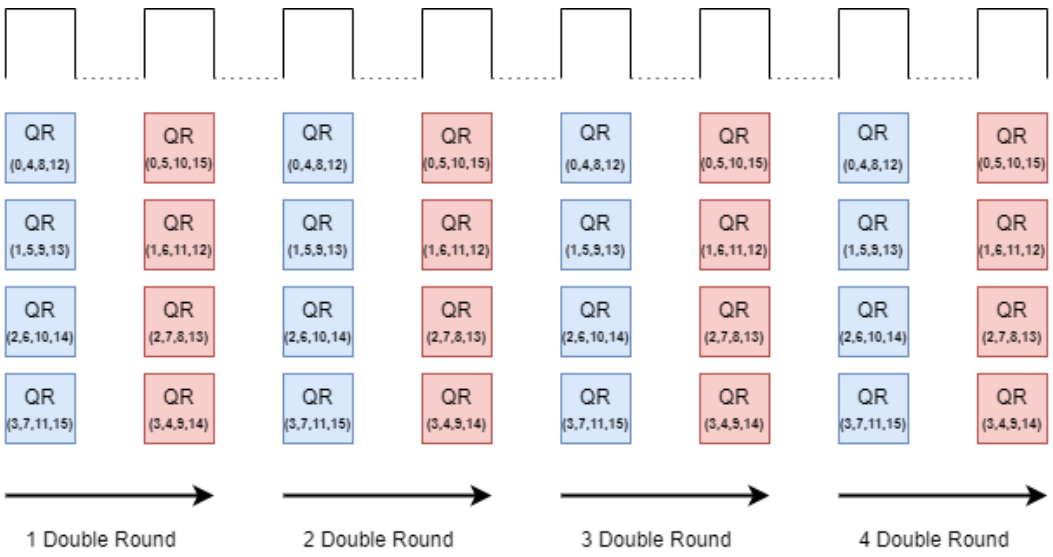

FIGURE 4.8: Double Round Time-Chart: With Unrolling & Array Partitioning

On the other hand, figure 4.8 shows how the appropriate pragmas and an efficient design can result in a much faster implementation by exploiting the hardware benefits. As before, blue boxes represent QR column calls whereas red ones represent QR diagonal calls. The first thing to note is that by unrolling the Double Round with a factor of 4 now in each cycle there are 4 QR calls. This was our goal from the beginning as there is no reason for an ongoing QR column call to wait for the previous QR column call to complete. Of course the same theory is applied to the diagonal QR calls. However, in our case, unrolling requires array partitioning since each QR call utilizes elements from a single array. Therefore, by using array partitioning we achieve multiple access to the same array within the same cycle. Finally, with this implementation, 2 cycles are consumed in each Double Round and $6 * 2 = 12$ cycles are consumed for 6 double rounds of ChaCha12 algorithm.

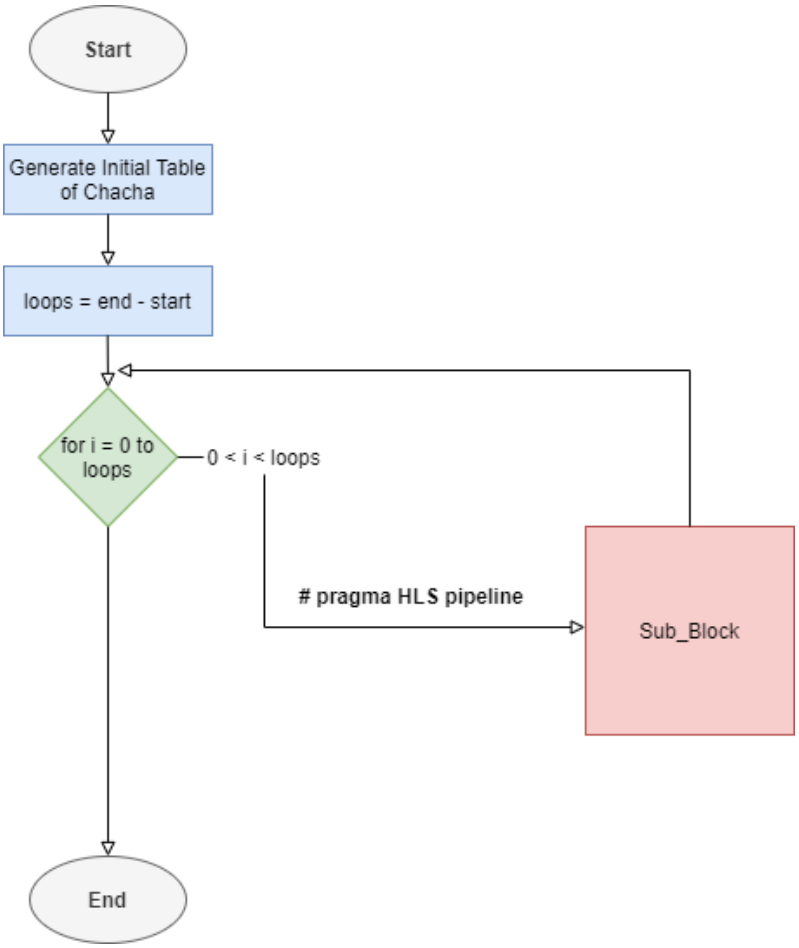

FIGURE 4.9: Pipeline Logic in ChaChaIP flowchart

Figure 4.9 is one of the most important pictures in this thesis as the correct position of the pipeline statement is key for gaining the desired acceleration of ChaCha12 algorithm.

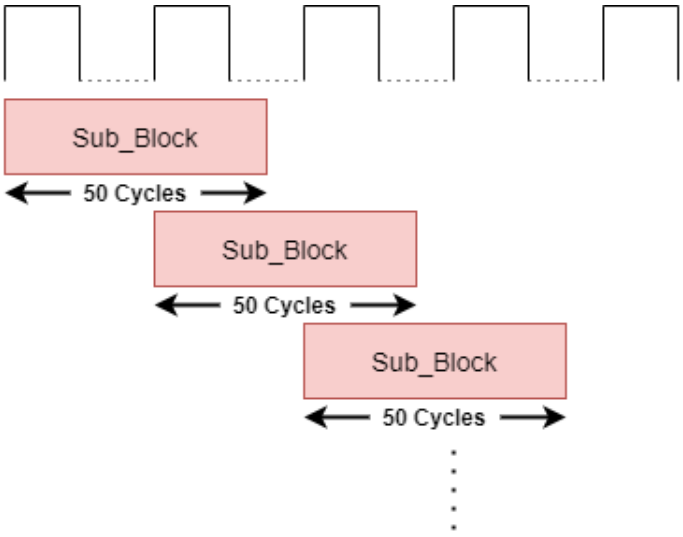

FIGURE 4.10: Pipeline Time-Chart for ChaChaIP

By achieving Pipeline Interval 1 (II=1) for each cycle a new process of Sub_Block part begins and as a result in every cycle there is a ChaCha result (figure 4.10). As already explained at AXI DMA limitations there can be one million ChaCha runs at each transaction between PL and PS. This means that the potency of our acceleration is highlighted when PS asks PL for one million ChaCha runs. One million ChaCha runs means one million Sub_Block runs. One Sub_Block needs 50 cycles to complete so, naturally, a generic system would need $1million * 50 = 50million$ cycles. In contrast, by utilizing pipeline logic in our implementation, hardware advantages shine out and consequently, for 1 million ChaCha runs our design needs only $1million + 50$ cycles.

Flowchart analysis

To begin with, the input word is received and is separated into the appropriate variables. By the term 'separated' it is meant that from 512-bits of the input word some of them are the key, nonce, start counter and the number of the last chunk. When we explained ChaCha and the reasons for implementing it with an FPGA we had correlated it with a usual counter and thus the number of the last 512-bit chunk which is needed has to be sent from PS. The number of loops of the Chacha algorithm is obtained by subtracting the start variable from the value of the last chunk.

After these steps Sub_Block is repeated. Through each iteration from the initial ChaCha array the only change is the two blocks that contain the counter. Although the second array(State[]) may seem unnecessary it should be noted that the last step of ChaCha algorithm is adding each block from the initial table to the same block of the table after Quarter Rounds has taken place, and therefore a copy of the initial table is needed. The procedure continues with the loop that holds the QR calls. Finally in the last loop the final table(512-bits or 64bytes) of ChaCha is being computed by adding each block from the two tables and convert the result from little to big endian. The result is sent immediately through the AXI stream interface back to the AXI DMA and finally to the PS system.

Algorithm 6 Quarter Round Function

1: **function** QUARTERROUND($uint32\&\ a, uint32\&\ b, uint32\&\ c, uint32\&\ d$)
2: $a \leftarrow MOD\_Ov(a + b)$ ▷ MOD_Ov() checks for overflow
3: $d \leftarrow d \oplus a$ ▷ bitwise XOR operation
4: $d \leftarrow d.lrotate(16)$ ▷ lrotate() is left rotation
5: $c \leftarrow MOD\_Ov(c + d)$
6: $b \leftarrow b \oplus c$
7: $b \leftarrow b.lrotate(12)$
8: $a \leftarrow MOD\_Ov(a + b)$
9: $d \leftarrow d \oplus a$
10: $d \leftarrow (d.lrotate(8)$
11: $c \leftarrow MOD\_Ov(c + d)$
12: $b \leftarrow b \oplus c$
13: $b \leftarrow b.lrotate(7)$
14: **return** ▷ Function arguments are pointers
15: **end function**

Lastly, the inner part of our design is the Quarter Round process which is shown in algorithm 6. Inside this function there is a set of three operations that is repeated four times. The first one is an addition of two 32-bit variables which may result in 33-bit if we have an overflow bit. In order to drop the overflow bit, a small function named MOD_Ov() takes the result of the addition as input and returns only the last 32-bits. After the addition a simple bit-wise xor operation take place and finally a left rotation.

Bit Rotation is similar to shift operation with the difference that the bits that are shifted out are pushed in sequence to the opposite side of the bit stream.

- **Left Rotation:** The bits that are shifted out from the left side are put back at the right side.
- **Right Rotation:** The bits that are shifted out from the right side are put back at the left side.

In QR process we have a left rotation(of 16, 12, 8 and 7 bits successively). Despite in general C or C++ language does not offer support such operations, Vivado HLS offers some advanced functions which in our case is the lrotate().

The first approach of the QR function was to take four variables as parameters and not as pointers. However, a struct object that groups four ap_uint<32> variables was necessary in order for the QR function to return four variables at once. Consequently, it was consuming more memory as for every simultaneous iteration of pipeline logic an extra struct was needed apart from the already array that held the specific values. Finally, the pointer logic that C language offers helps decrease the necessary resources and also make our design more efficient.

Chapter 5

Results

This chapter demonstrates all the results both from software approach and most importantly hardware implementation. It is fundamental to not only specify timing but also power and energy differences between the compared platforms.

5.1 Specification of Compared Platforms

The equipment used for all the experiments during this thesis was a generic laptop CPU and PYNQ-z1(Python Productivity for Zynq-7000 ARM/FPGA SoC). Specifically the CPU is an Intel i5 3230M which was released at January 2013 and PYNQ-z1 contains ZYNQ XC7Z020-1CLG400C and was released at the start of 2018. Since 2018 there has been ample improvement on PYNQ-z1.

5.1.1 Intel i5 3230M

Intel i5 3230M is a mobile processor. Some basic information is shown below.

TABLE 5.1: Intel i5 3230M Specifications

Cores	2
Threads	4
MAX Turbo Frequency	3.2GHz
TDP	35W
MAX Memory Bandwidth	25.6GB/s
Lithography	22nm

Thermal Design Power(**TDP**): represents the average power, in watts, the processor dissipates when operating at Base Frequency with all cores active under an Intel-defined, high-complexity workload. Moreover, MAX Turbo frequency is the maximum single core frequency at which the processor is capable of operating whereas Max Two-core processing is 3GHz.

5.1.2 PYNQ-Z1 Resource Utilization

All specifications about PL logic are referred at 2.3 where implementation is analyzed. In the following table hardware utilization of the final implementation is presented.

TABLE 5.2: Pynq z1 Resource Utilization

PS Clock	650 MHz
PL Clocks	100 & 150 MHz
BRAMs	15,53%
FFs	48,69%
LUT	47,74%
DSP	0%

5.2 Power Consumption

Electronic devices require energy/power in order to be in use. More precisely, by the term of power consumption we are referring to the energy consumed per unit time for the completion of a task. It is usually measured in Watt(W) or kilowatt(kW). Every hardware designer has to manage and keep power consumption on specific levels depending on the design needs but also the available tools. In order to minimize energy losses and increase the system's energy efficiency, the designer has to keep the average power consumption at the lowest possible. It is clear that, for the designs to be kept simple and economic, low power consumption is the most effective way.

5.3 Energy Consumption

Although power consumption shows power per task, it is essential to calculate the amount of energy consumed during specific time. Therefore, energy comsuption is easily calculated: As a result, energy consumption is an easy way for the calculation:

$$Energy = Power * time \tag{5.1}$$

Formula 5.1 demands:

- **Power**: Required power
- **Time**: Amount of time to complete the task

Lastly, energy is usually measured in Joule(J) or kilo-joule(kJ) and, as with power consumption, energy is also preferred to be as low as possible so as to minimize the operational costs. Taking everything into account, aside from time acceleration, energy/power consumption should be considered and compared between the two platforms.

5.4 Throughput and Latency Speedup

Latency and throughput are two fundamental concepts both in computer science and hardware. Additionally, speedup is used to compare the workload/time between two or more systems utilizing/running the same task. Speedup is a notion that was first established by Amdahl's law [2]. Despite that Amdahl's law was first mentioned only on parallel computing(meaning multiple processors), it can show performance improvement after any resource enhancement.

Latency is the time needed for single task to be completed by a specific system and is calculated as follows:

$$Latency = \frac{T}{W} \tag{5.2}$$

- **T**: execution time of task
- **W**: execution workload of task

Throughput shows the maximum rate of processing a specific problem:

$$Throughput = r * v * A = \frac{r * A * W}{T} = \frac{r * A}{L} \tag{5.3}$$

- **r**: execution density
- **A**: execution capacity

Finally, speedup can be calculated by comparing each value of the previous formulas(5.2 & 5.3) with the corresponding one from the other systems.

$$S_{Latency} = \frac{L_1}{L_2} = \frac{T_1 * W_2}{T_2 * W_1} = \frac{1}{(1-p) + \frac{p}{s}} \tag{5.4}$$

- **s**: speedup of the improved task
- **p**: task's portion that is benefited by resource enhancement

$$S_{Throughput} = \frac{Thr_2}{Thr_1} \tag{5.5}$$

5.5 ChaCha Performance

Having analyzed all the previous scientific concepts, apart from timing improvements it is also essential to compare latency, throughput and energy efficiency between CPU-i5 3230M and our design in this section . Nonetheless, it is fundamental before we proceed with the tables to distinguish what differs CPU in relationship with our design.

As already explained our design is working like a counter and by that it is meant that with every initialization many results come back from PL to PS. These results are simply running of the ChaCha algorithm as usual, numerous times and during each run only the block counter block of the initial table is changed. Due to Vivado AXI DMA IP limitations, a simple transaction can only be achieved with transfers up to 64MB each time and for that reason with every initialization, our design can return up to one million results at a time(1,000,000 * 64 = 61.0352MB).

Based on the idea that our design can return up to one million ChaCha runs, a testing approach of different packets of runs at a time is essential to clarify which size yields the best throughput.

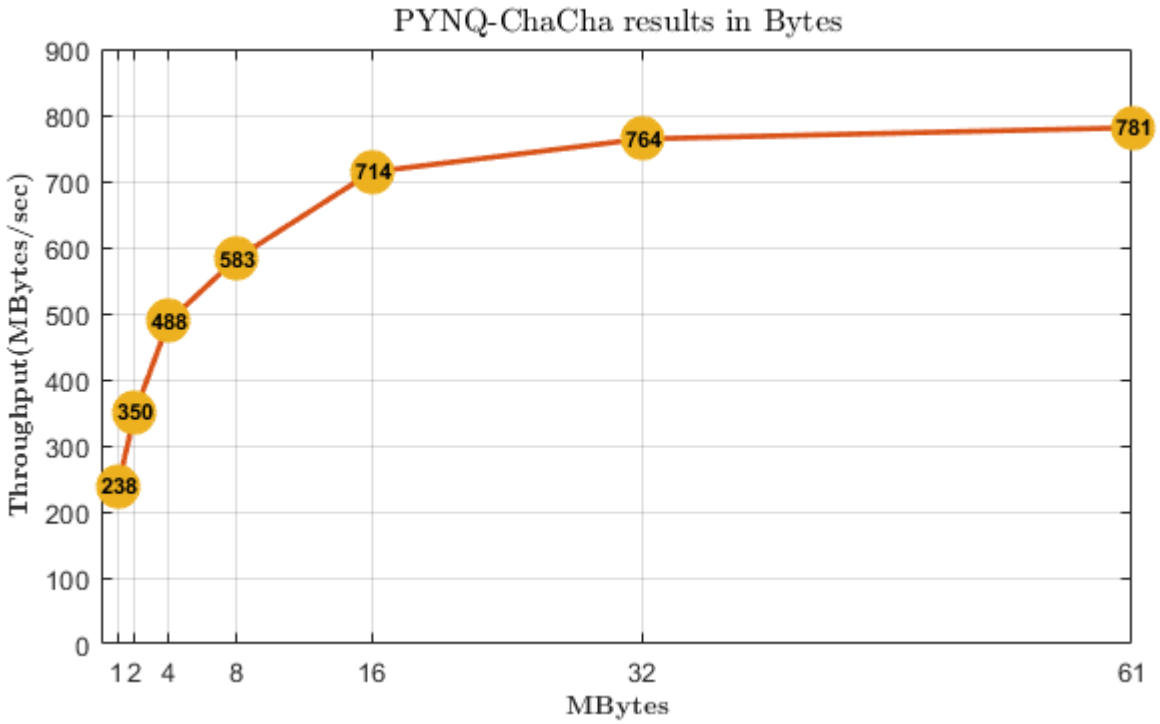

FIGURE 5.1: PYNQ Throughput (MBytes/sec)

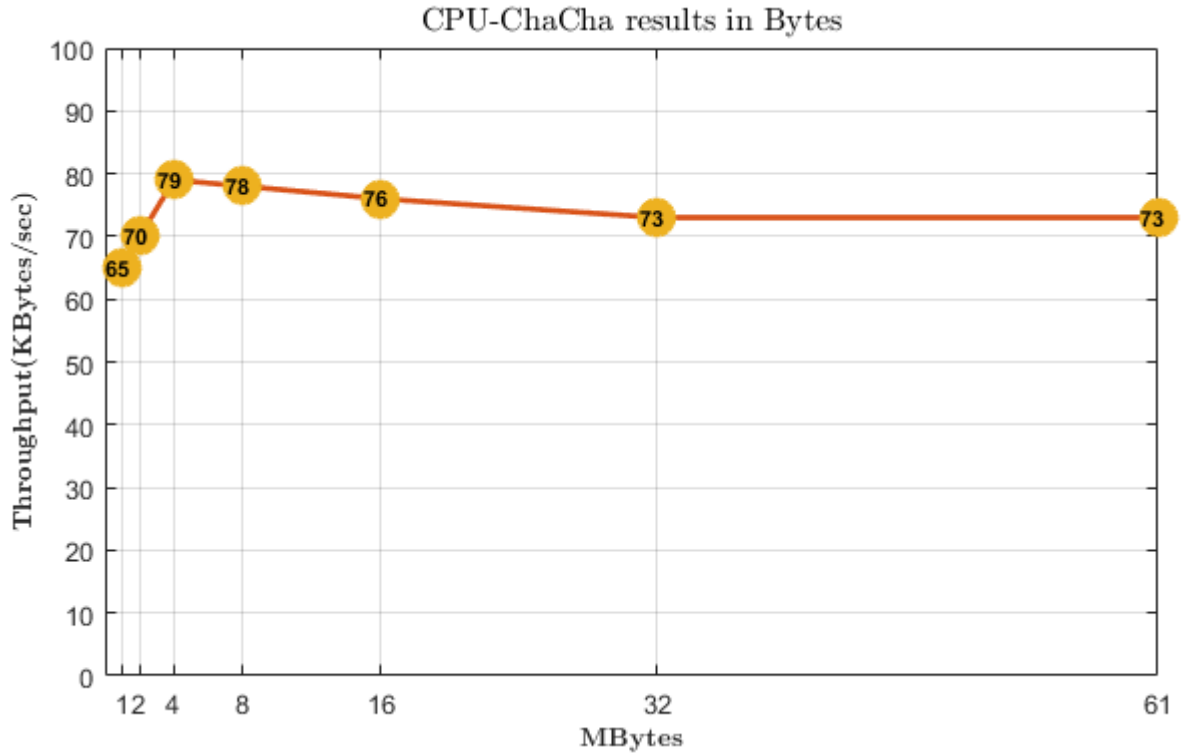

FIGURE 5.2: CPU-i5 Throughput (KBytes/sec)

Figure 5.1 and 5.2 show the results of our hardware design and CPU-vanilla software code for different sizes of ChaCha runs respectively. Specifically, if we run the design each time for 1MB results (1MB/64-byte each run = 16,384 runs of ChaCha) in 4.2ms there is a total throughput of 238MByte/sec (3,899,392 runs of ChaCha/sec). The best throughput was achieved with the utilization of all AXI DMA hardware limits and as a result run our design for one million ChaCha results at a time. This approach gives 781MByte/sec (12,795,904runs of ChaCha/sec) while each run needs only 82ms. In contrast to the CPU which needs 880 seconds with the throughput of 73KByte/sec for one million runs of ChaCha we achieved a big impact on timing results, especially for big messages.

TABLE 5.3: ChaCha acceleration: Pynq-z1 vs Intel-i5

Plaintext(Bytes)	**ChaCha Runs**	**Intel-i5(msec)**	**Pynq-z1(msec)**
4,096	64	71	2.85
8,192	128	102	2.99
16,384	256	158	3.04
32,768	512	327	3.16
65,536	1,024	677	3.38
131,072	2,048	1,296	3.88
262,144	4,096	2,595	3.88
524,288	8,192	5,219	3.99
1,048,576	16,384	15,994	4.20
2,097,152	32,768	29,257	5.71
4,194,304	65,536	51,848	8.19
8,388,608	131,072	105,025	13.7
16,777,216	262,144	215,578	22.4
33,554,432	524,288	447,709	41.8
67,108,864	1,048,576	910,041	103

Table 5.3 is the most interesting table as it presents the outcome of our hardware design and our whole work. As already explained, for every 64 bytes of the plaintext the ChaCha algorithm has to run through all of its computations.

$$ChaCha_{Runs} = \frac{Plaintext_{size}(Bytes)}{64} \tag{5.6}$$

The second column (ChaCha Runs) of the table 5.3 is computed based on the equation 5.6. Notably, these timings have been extracted once more using the same profiling methods that have been analyzed during section 3.3. Moreover, the xor part between the ChaCha result and the plaintext is not included as the xor equation does not exist in our design and is reserved for the PS system. It is extraordinary how much our design prevails over CPU. Even for small plaintexts like 4,096 bytes, our design is 25x times faster than Intel-i5 and the speedup grows vast as the plaintext gets bigger. For a plaintext of 67,108,864 bytes, our design has to be called twice from PS system and it is 8,835x faster than Intel-i5. Therefore, the best case scenario for our design is to run for one million ChaCha runs in each call from the PS.

TABLE 5.4: ChaCha Architecture Comparison: PYNQ vs CPU

	PYNQ	**CPU**
Clock(MHz)	150	3200
Latency(msec)	2.71	3
Power Consumed(Watt)	2	14
Energy(Joule)	0.16	12.32K

Latency: refers to one ChaCha run
Energy: refers to one million runs of ChaCha

Table 5.4 shows latency, power and energy. Latency is based on the worst case scenario in which our design works for only 64Byte result(one run of ChaCha). Energy has been calculated considering that our design aims for big messages for which we choose each PL run to return one million runs of ChaCha. More precisely CPU consumes 12.3KJoule for one million runs of ChaCha while PYNQ needs only 0.16Joule.

TABLE 5.5: ChaCha Speedup over CPU

	PYNQ vs CPU
Throughput Speedup	10,698x

Throughput Speedup: for one million runs ChaCha

TABLE 5.6: ChaCha Energy & Power efficiency over CPU

	PYNQ vs CPU
Power Efficiency	7x
Energy Efficiency	77,000x

Energy Efficiency: for one million ChaCha runs

Tables 5.5 and 5.6 shows how much our design prevails over CPU with relation to above definitions. Firstly, in the case of one million results of ChaCha, throughput is 10,698x better than CPU. Additionally, for the worst-case scenario of the design which correlates to one run of the ChaCha algorithm, our design is a little better than CPU. Secondly, PYNQ is 7x more power-efficient and as a result for one million runs of ChaCha, energy is 77,000x more efficient than CPU.

TABLE 5.7: ChaCha Comparison with Related Work

Design	**Frequency(MHz)**	**Throughput(MB/s)**
G.Kanda and K.Ryoo [21]	161	479
Igor Semenov [38]	50	126
Our Design	150	**781**

Finally, our implementation is 17-times faster than the work of Nuray At. et al. [4] but this was to be expected as their design is intended for a co-processor which resulted in 46 MB/s for ChaCha12 encryption. On the other hand, it is more interesting if we compare our work with the research of G.Kanda and K.Ryoo [21] (table 5.7). Although in our case we have ChaCha12 instead of ChaCha20 and we did not include the xor part at the hardware level, it is still remarkable that our design can compete with their ASIC implementation and it is nearly 2-times faster than their FPGA implementation. Moreover, Igor Semenov [38] designed the implementation of ChaCha20, but as he tried to achieve acceleration without pipeline logic and with low resources, it is only natural that our design is 6-times faster. However, as each implementation poses different standard result metrics, those differences should be taken into consideration.

5.6 Adiantum Performance

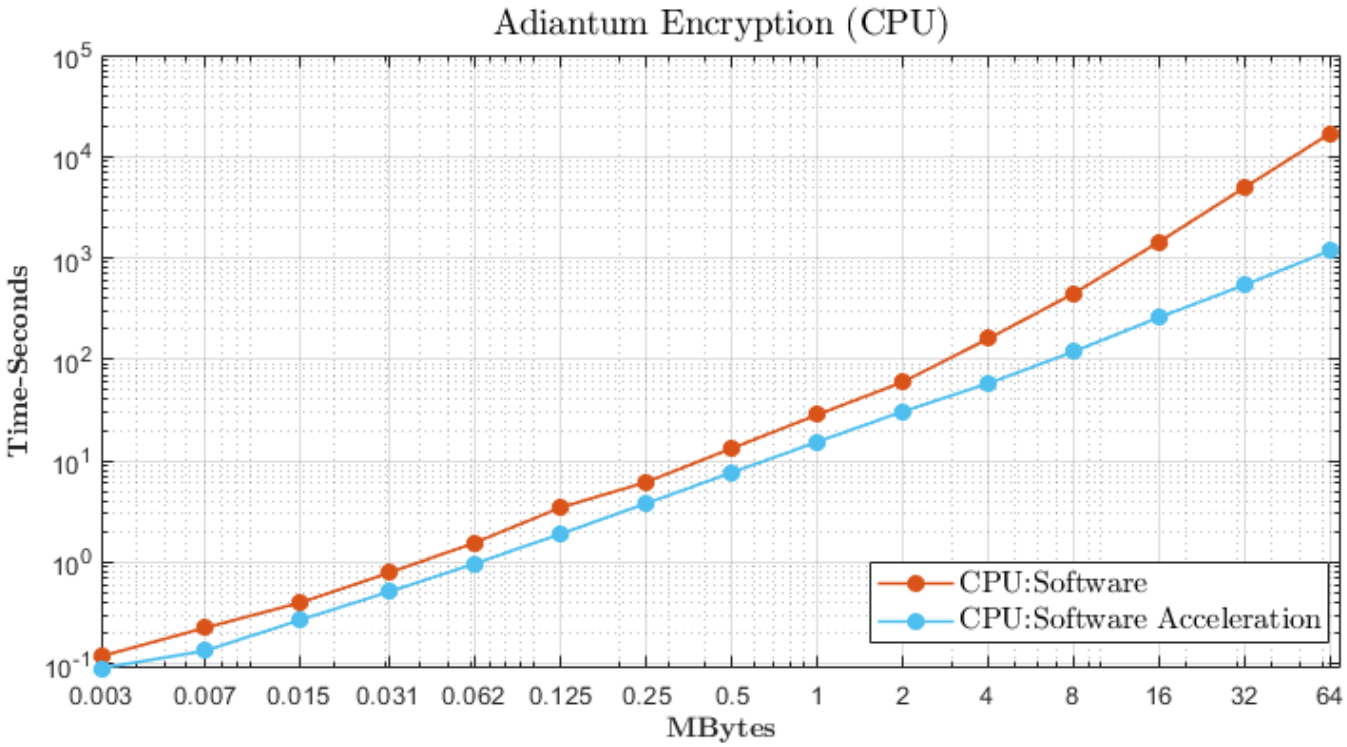

FIGURE 5.3: CPU Encryption

Figure 5.3 shows how much time Adiantum encryption takes in our CPU as the plaintext/message gets bigger. There are two waveforms, one blue and one green. The blue waveform shows the encryption time before the original software implementation (vanilla) whereas the green one is the result of all software changes. Evidently, before reaching the 2MB sized plaintext there is no big difference, as was expected due to profiling. As a brief reminder, profiling showed that after a plaintext of 2MB size, stream cipher takes more and more time. It is essential to note at this point how much difference our software changes achieved in big messages such as 64MB and realize at what extent certain Python libraries may affect a code with relation to time. Specifically, at 64MB size of plaintext Adiantum encryption took about 17,000sec whereas with our changes takes 1,100sec.

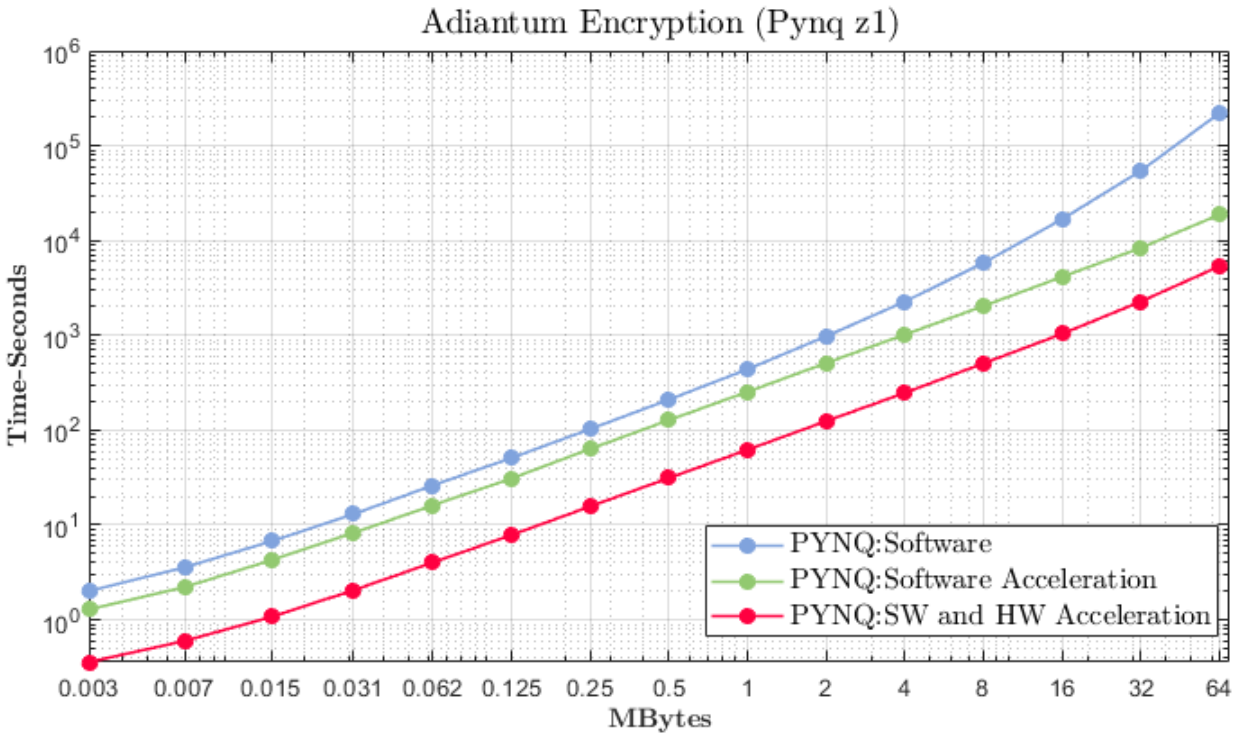

FIGURE 5.4: PYNQ Encryption

Figure 5.4 includes all the execution times of Adiantum encryption based on PYNQ z1 device. All three curves depict the amount of time our FPGA needs for running Adiantum encryption for each approach and messages of different sizes. More specifically the blue curve is the initial code and it is essential to observe that for 64MB size of plaintext PYNQ needs about 3.5 times more than our CPU for this size. The green curve depicts encryption time after software changes and as we see with 64MB size plaintext a decrease in order of magnitude has been achieved.

Finally the last and the most important, red curve is the combination of running Adiantum encryption both with PS and PL. PS executes our new version of Adiantum encryption while PL is used for the ChaCha12 algorithm of the Stream Cipher part. Even for small messages like disc encryption of 4096 bytes, PYNQ needed about 2.5 seconds while in the last version 0.3 seconds. Remarkably, our design is more impactful for big messages. Specifically, for 64MB PYNQ required more than 200,000 seconds at first, whereas now it only needs 5,000 seconds. The most outstanding outcome is that our design meets the maximum theoretical speedup calculated by the Amdahl's law formula.

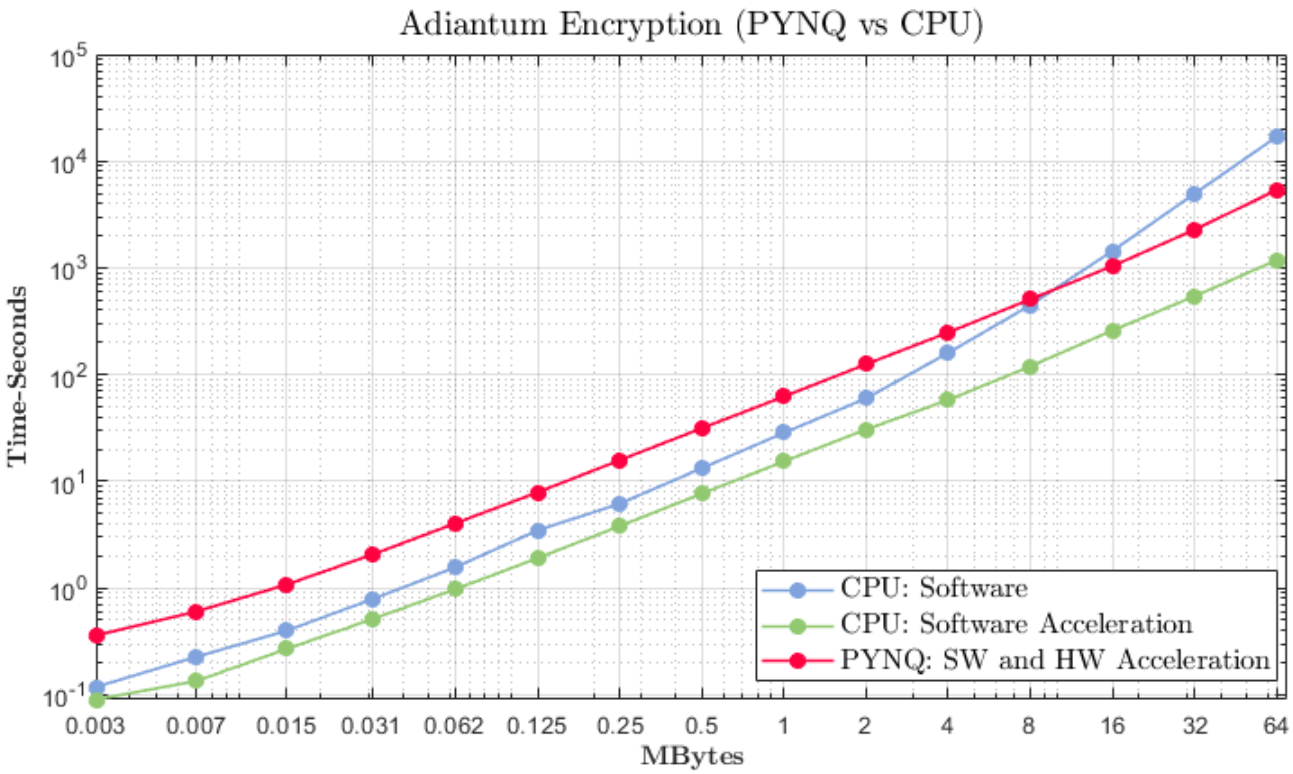

FIGURE 5.5: Adiantum Encryption (PYNQ vs CPU)

Figure 5.5 demonstrates the final comparison of the Adiantum cryptography algorithm with PYNQ and Intel-i5. As already stated in (Chapter 2: Profiling Section 3.3) even if we fully minimize the stream cipher part of Adiantum, the PYNQ PS system is too slow in comparison to Intel-i5 CPU, and only 4x speedup at most could be achieved. As a result, in comparison, between the vanilla version of Adiantum encryption and our PYNQ version, for big messages and specifically for 64MB plaintext, PYNQ is 3.5x times faster and 23x more energy efficient (figure 5.5). However, our software improvements also had a big impact on Intel-i5 and finally, CPU is also faster for bigger plaintexts. Although this comparison is not beneficial to our PYNQ implementation, in fact our hardware design (ChaCha implementation) is unbelievably faster than CPU. The uprising CPU-FPGA hybrids that have been recently put on the market can overthrow all obstacles that an existing PS system may cause to such implementations.

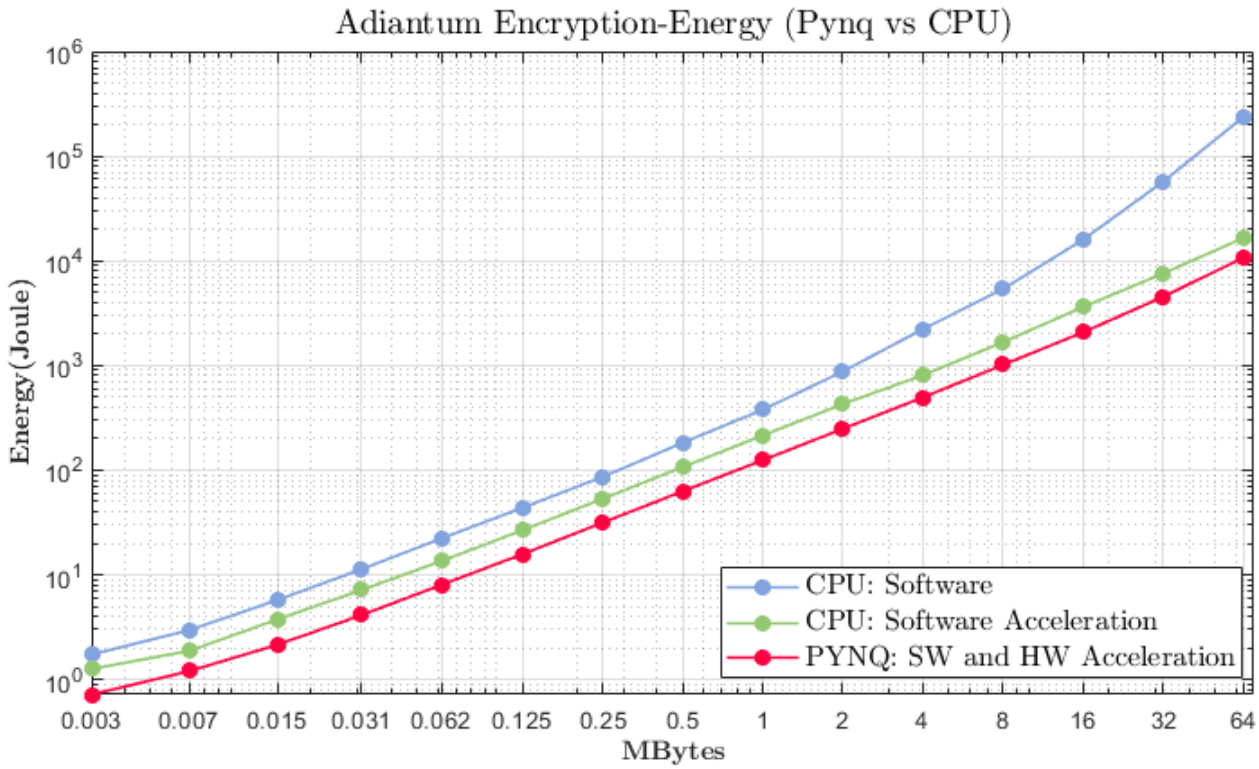

FIGURE 5.6: PYNQ vs CPU Energy

Finally, any embedded implementation is strongly related to energy consumption and thus, figure 5.6 is presented. Despite CPU being faster due to the PS system that PYNQ incorporates, it is less energy efficient. More specifically, Adiantum on PYNQ is 2x more energy efficient than our CPU.

Chapter 6

Conclusions and Future Work

The sixth and last chapter of this thesis serves as a summary and evaluation of our work. Furthermore, some ideas for future work are proposed and explained with hope to intrigue more people into this subject and specifically in Adiantum cryptography algorithm.

6.1 Conclusions

Over the years, mankind uses technology more and more, from the smallest things of everyday life to more advanced ones like jobs, scientific research, etc. The higher the rate of technological use, the greater the need for privacy and security. Besides the security aspects, it is also a fact that electronic files are getting bigger in terms of size in bytes. As the file size gets bigger, more time is needed for encrypting and decrypting these files. However, time is of the essence and should not be wasted on mundane everyday tasks. As a result every approach that aims to achieve acceleration on these subjects becomes ever so important.

The purpose of this thesis was to explore how Adiantum cryptography algorithm can be accelerated by utilizing and exploiting FPGA advantages such as parallelism. Profiling methods made it clear that one should aim for acceleration on the stream cipher part (ChaCha12) of the Adiantum algorithm as big messages consumed the most time. Notice here that, given the time needed for ChaCha12 stream cipher, we achieved remarkable results.

Precisely, in comparison to Intel-i5 3230M, our **ChaCha12 design** is:

- 10,731x faster
- 10,698x throughput Speedup
- 77,000x times more energy efficient

It is remarkable that including I/O overhead if our Adiantum design were to run at a strong CPU with tightly-coupled FPGA, would meet the theoretical speedup as calculated by Amdahl's law. Nonetheless, the Adiantum encryption algorithm is not entirely implemented with hardware logic. Additionally, software changes have made a big impact on our CPU with relation to the time needed for Adiantum encryption. This resulted in CPU being faster but not energy efficient.
Specifically, with our **New software version of Adiantum: Intel-i5 3230M** is:

- 4x times faster
- 2x times less energy efficient

However, it would be more interesting if we were able to test our Adiantum design with an FPGA that incorporates a much better CPU system or, better yet, with the new CPU-FPGA hybrids as PYNQ's CPU is the only reason why Intel-i5 3230M proving faster. Finally, Google tested Adiantum on ARM Cortex-A7 but in our case (ARM Cortex A9) it should be noted that with our changes we achieved to encrypt 64kbyte at about the same time that the initial code requires for 4kbyte.

6.2 Future Work

Adiantum Cryptography algorithm is relatively new and as no other relevant research has been done on it so far it makes sense that this thesis opens up numerous new paths.

Google proved that from a theoretical point of view, Adiantum can encrypt a message of size up to 2^{73} bits. Also, as the algorithm consists of many different methods of symmetric cryptography such as hash functions, stream ciphers and block ciphers(specifically AES), there is the prospect of a very good algorithm in terms of security. All of this results in Adiantum being able to be used in a wider range and not just in disc encryption for low-end devices. Therefore endeavors towards acceleration in execution time, not

only on the hardware side but in software as well, can lead to its integration on many electronic devices in our daily lives.

In terms of hardware, the Adiantum algorithm is undoubtedly a very fast algorithm that does not require as much complexity as the equivalent algorithms that already exist. The question, however, is how much larger the message can be in a reasonable amount of execution time. To resolve the previous question further deepening into acceleration can result in very interesting results.
In this thesis we tried to minimize the time consumed in the stream cipher part of Adiantum and we already have a significant improvement in the execution time. A further approach could be the implementation of hash function(Poly1305 with NH) which now takes about the 90% of encryption time into hardware and as a result the whole Adiantum algorithm. Finally in the world of cryptography, there is the view that regardless of the mathematical complexity of an algorithm, the real metric of its security is the amount of time it remains uncompromised. It is therefore very interesting to continue to see the course of Adiantum in the future.

References

[1] National Security Agency. "US National Security Agency 'is surveillance leviathan". In: *BBC* (Aug. 2013). URL: https://www.bbc.com/news/technology-23669003.

[2] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[3] Mohit Arora. "Handling Endianness". In: *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*. New York, NY: Springer New York, 2012, pp. 155–168. ISBN: 978-1-4614-0397-5. DOI: 10.1007/978-1-4614-0397-5_7. URL: https://doi.org/10.1007/978-1-4614-0397-5_7.

[4] Nuray At et al. "Compact Hardware Implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 61.2 (2014), pp. 485–498. DOI: 10.1109/TCSI.2013.2278385.

[5] J.-P. Aumasson et al. "SHA-3 proposal BLAKE". In: (2008). URL: https://www.aumasson.jp/.

[7] Mihir Bellare and Phillip Rogaway. "On the Construction of Variable-Input-Length Ciphers". In: *Fast Software Encryption*. Ed. by Lars Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 231–244. ISBN: 978-3-540-48519-3. URL: https://cseweb.ucsd.edu/~mihir/papers/lpe.pdf.

[8] Daniel J. Bernstein. "ChaCha, a variant of Salsa20". In: *State of the Art of Stream Ciphers Workshop, SASC 2008, Lausanne, Switzerland* (Jan. 2008). URL: https://cr.yp.to/chacha/chacha-20080128.pdf.

[9] Daniel J. Bernstein. "Extending the Salsa20 nonce". In: *Workshop record of Symmetric Key Encryption Workshop 2011* (2011). URL: https://cr.yp.to/snuffle/xsalsa-20110204.pdf.

[10] Daniel J. Bernstein. "Salsa20/8 and Salsa20/12". In: (2006). URL: https://cr.yp.to/snuffle/812.pdf.

[11] Daniel J. Bernstein. "The Poly1305-AES message-authentication code". In: *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, revised selected papers* (Feb. 2005). URL: https://cr.yp.to/mac/poly1305-20050329.pdf.

[12] Daniel J. Bernstein. "The Salsa20 family of stream ciphers". In: *New Stream Cipher Designs: The eSTREAM Finalists* (2008). URL: https://cr.yp.to/snuffle/salsafamily-20071225.pdf.

[13] Sylvain Collange, David Defour, and Arnaud Tisserand. "Power Consumption of GPUs from a Software Perspective". In: *Computational Science – ICCS 2009*. Ed. by Gabrielle Allen et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 914–923. ISBN: 978-3-642-01970-8.

[14] Paul Crowley and Eric Biggers. "Adiantum: length-preserving encryption for entry-level processors". In: *IACR Transactions on Symmetric Cryptology* 2018.4 (Dec. 2018), pp. 39–61. DOI: 10.13154/tosc.v2018.i4.39-61. URL: https://tosc.iacr.org/index.php/ToSC/article/view/7360.

[15] Frank Denis. "XChaCha20". In: *libsodium* (2018). URL: https://doc.libsodium.org/advanced/stream_ciphers/xchacha20.

[16] Morris Dworkin et al. *Advanced Encryption Standard (AES)*. Nov. 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[17] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. 2008. ISBN: 978-0-12-370522-8. DOI: https://doi.org/10.1016/B978-0-12-370522-8.X5001-8.

[20] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Dec. 1996. ISBN: 978-0-684-83130-5.

[21] Guard Kanda and Kwangki Ryoo. "High-Throughput Low-Area Hardware Design of Authenticated Encryption with Associated Data Cryptosystem that Uses ChaCha20 and Poly1305". In: *International Journal of Recent Technology and Engineering (IJRTE)* 8 (July 2019). URL: https://www.ijrte.org/wp-content/uploads/papers/v8i2S6/B10170782S619.pdf.

[22] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Mar. 1995. ISBN: 978-0-13-061466-7.

[23] Lars R. Knudsen. "Block Ciphers". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 152–157. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_549. URL: https://doi.org/10.1007/978-1-4419-5906-5_549.

[24] Ted Krovetz. "HS1-SIV (version 2)". In: *CAESAR competition: 2nd Round* (July 2015). URL: https://competitions.cr.yp.to/round2/hs1sivv2c.pdf.

[25] Ted Krovetz. "UMAC: Message Authentication Code using Universal Hashing". In: *RFC 4418, RFC Editor* (Mar. 2006). URL: https://www.rfc-editor.org/rfc/rfc4418.txt.

[26] Moses Liskov, Ronald L. Rivest, and David Wagner. "Tweakable Block Ciphers". In: *Advances in Cryptology — CRYPTO 2002*. Ed. by Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 31–46. ISBN: 978-3-540-45708-4.

[27] S. Lucks and Jon Callas. "The Skein Hash Function Family". In: 2009. URL: https://www.cs.rit.edu/~ark/20090927/Round2Candidates/Skein.pdf.

[28] Bernard Marr. "How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read". In: *Forbes* (May 2018). URL: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=5af00f6760ba.

[29] Nicole Martin. "How Much Data Is Collected Every Minute Of The Day". In: *Forbes* (Aug. 2019). URL: https://www.forbes.com/sites/nicolemartin1/2019/08/07/how-much-data-is-collected-every-minute-of-the-day/?sh=6e06a8113d66.

[31] Yoav Nir and Adam Langley. "ChaCha20 and Poly1305 for IETF Protocols". In: *RFC 7539, RFC Editor* (May 2015). URL: https://www.rfc-editor.org/rfc/rfc7539.txt.

[32] Christof Paar and Jan Pelzl. *Understanding Cryptography*. 2010. ISBN: 978-3-642-44649-8. DOI: https://doi.org/10.1007/978-3-642-04101-3.

[36] Phillip Rogaway and Thomas Shrimpton. "Deterministic Authenticated-Encryption: A Provable-Security Treatment of the Key-Wrap Problem". In: *Cryptology ePrint Archive, Report 2006/221* (2006). URL: https://eprint.iacr.org/2006/221.

[37] Phillip Rogaway et al. “OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption”. In: vol. 6. Jan. 2001, pp. 196–205. DOI: 10.1145/501983.502011.

[38] Igor Semenov. “An Implementation Of ChaCha20 Stream Cypher in All-Programmable SoCs”. In: *The Department of Electrical and Computer Engineering of The University of Alabama in Huntsville* (2020). URL: http://www.ece.uah.edu/~milenka/docs/igor.semenov.thesis.pdf.

[40] Sergei Volokitin and Gerben Geltink. “Hardware Implementation of HS1-SIV”. In: Oct. 2017, pp. 179–194. ISBN: 978-3-319-67875-7. DOI: 10.1007/978-3-319-67876-4_9.

External Links

[6] *AXI Protocol*. URL: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[18] *Intel Cyclone V*. URL: https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-v.html.

[19] *Jupyter Notebook*. URL: https://jupyter.org/index.html.

[30] *Matlab*. URL: https://www.mathworks.com/products/matlab.html.

[33] *PyCharm*. URL: https://www.jetbrains.com/pycharm/.

[34] *Pynq Design*. URL: http://www.pynq.io/.

[35] *Pynq-z1*. URL: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start.

[39] *Vivado-ILA IP*. URL: https://www.xilinx.com/products/intellectual-property/ila.html.

[41] *Xilinx Vivado design Suite*. URL: https://www.xilinx.com/products/design-tools/vivado.html.