# AN IMPLEMENTATION OF CHACHA20 STREAM CYPHER IN ALL-PROGRAMMABLE SoCs

by

IGOR SEMENOV

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in
The Department of Electrical and Computer Engineering
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2020

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Igor Semenov (date)

## THESIS APPROVAL FORM

Submitted by Igor Semenov in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering.

| | | |
|---|---|---|
| *Dr. Aleksandar Milenković* | (Date) | Committee Chair |
| *Dr. David Coe* | (Date) | |
| *Dr. Jeffrey Kulick* | (Date) | |
| *Dr. Ravi Gorur* | (Date) | Department Chair |
| *Dr. Shankar Mahalingam* | (Date) | College Dean |
| *Dr. David Berkowitz* | (Date) | Graduate Dean |

# ABSTRACT

School of Graduate Studies
The University of Alabama in Huntsville

Degree Masters of Science in Engineering College/Dept. Engineering/Electrical and Computer Engineering

Name of Candidate Igor Semenov

Title An implementation of ChaCha20 stream cypher in all-programmable SoCs

An increased reliance on services offered online is an inherent aspect of the Information Age. Such services often handle sensitive customers' data and therefore must ensure its confidentiality, for example, by using cryptographic algorithms. Some algorithms, such as AES, have long been used for this purpose, but they impose additional cost and/or performance overheads. However, new lightweight stream cyphers, such as ChaCha20, are emerging as a faster alternative to conventional algorithms without sacrificing the security. Development of hardware cryptographic accelerators has been widely used to reduce performance overheads of cryptographic algorithms. In the past, this approach was extremely expensive, but nowadays, widely available all programmable SoCs make it more affordable. In this thesis, we present an IP-core for ChaCha20 acceleration in a low-cost all-programmable SoC, Cyclone V. We describe the design of our core, consisting of a ChaCha20 accelerator and a custom DMA. We also present a software framework that employs multiple CPU cores or hardware accelerators for file encryption. We explore different configurations of this framework in order to find the optimal one. Our experiments show that two

accelerators clocked at 50 MHz working concurrently provide the throughput of 120.5 MiB/s, a five-fold throughput improvement over the baseline software file encryption executed on two ARM Cortex-A9 processor cores running at 800 MHz.

Abstract Approval: Committee Chair _____
*Dr. Aleksandar Milenković*

Department Chair _____
*Dr. Ravi Gorur*

Graduate Dean _____
*Dr. David Berkowitz*

# ACKNOWLEDGMENTS

First, I would like to thank the Department Electrical and Computer engineering for awarding me the position of a teaching assistant, which helped me to greatly improve my teaching skills and fund my education and research at UAH.

Second, I would like to express my deepest gratitude Dr. Aleksandar Milenković for serving as my advisor and providing constant support throughout this work. He has been a great inspirator and mentor for me during the last 2 years.

Additionally, I am also very grateful to Dr. Jeffrey Kulick and Dr. David Coe for serving as my committee members. Their challenging questions during the defense procedure and follow-up feedback helped me to improve this work.

I would like to sincerely thank my lab mates Mr. Ranjan Hebbar, Mr. Prawar Poudel, Dr. Mounika Ponugoti and Mr. Amirahmad Ramezani. They have been my true friends who I could always reach if I needed help with academic or life issues.

Finally, I am expressing my gratitude to my parents for their love and moral support. Special thanks goes to my wife, who treated my time-consuming work on this thesis with understanding.

## Contents

## LIST OF FIGURES

FIGURE PAGE

## LIST OF TABLES

TABLE PAGE

## LIST OF SYMBOLS

| SYMBOL | DEFINITION |
|---|---|
| AEAD | Authenticated encryption with associated data |
| AES | Advanced encryption standard |
| ASIC | Application-specific integrated circuit |
| CSR | Control and status register |
| DRAM | Dynamic random-access memory |
| FPGA | Field-programmable gate array |
| HPS | Hard processor system |
| IP-core | Intellectual property core |
| IRQ | Interrupt request |
| LUT | Lookup table |
| OS | Operating system |
| OTP | One-time pad |
| RAM | Random-access memory |
| SoC | System on a chip |
| UIO | User-space input/output |

# Chapter 1

# INTRODUCTION

Data confidentiality has become especially important in the Information Age. Nowadays many services are provided online: governments, financial institutions, and e-commerce companies interact with their customers remotely. Such interaction is extremely convenient for all parties. However, service providers are required to store and transfer sensitive customer's data, which makes them a target for hacker attacks. Sometimes customer's data becomes exposed even without malicious actions, leading to data leaks. To prevent detrimental consequences of unauthorized data access, sensitive information can be encrypted using strong cryptographic algorithms. Such algorithms are usually computationally difficult, which, together with ever growing amounts of data, leads to the need of increasing the efficiency of data encryption. That is why new lightweight cryptographic algorithms are being developed. An example of such an algorithm is ChaCha20, which can outperform its conventional competitors. Additionally, the performance of the encryption process can be improved by using hardware accelerators, designed specifically to facilitate a certain cryptographic algorithm. Even though fabrication of such accelerators is extremely expensive, some technologies, such as field-programmable gate arrays, make it much more affordable.

The goal of this work is to explore any performance improvements of implementing a hardware accelerated ChaCha20 algorithm on a system-on-a-chip (SoC) platform that contains an FPGA and a CPU. We focus on file encryption in the Linux environment as the final application of our solution.

## 1.1 Technology trends

This thesis focuses on hardware implementation of ChaCha20 cypher in the context of all-programmable SoCs. Our motivation for pursuing this topic arises from the following technological trends.

The first trend we recognize is extensive application of hardware accelerators for different tasks, including data encryption. Since strong cryptographic algorithms are usually computationally intensive, replacing their software implementations with hardware modules dramatically improves their performance. This is especially important for embedded systems where resources are limited due to size, cost, or power consumption requirements. Many companies whose products target embedded market use hardware accelerators. For example, STMicroelectronics embeds an AES crypto-processor in some families of their STM32 microcontrollers [1]. Texas Instruments also uses AES hardware accelerators in some of their MSP430 microcontrollers [2].

Development of new stream cyphers is another technology trend. Many cryptographic applications these days rely on block cyphers. Recently, stream cyphers started gaining more attention because they are capable of providing higher security level with the same performance. An example of such a cypher is ChaCha20 [3]. According to

Langley [4] ChaCha20 shows better performance than Advanced Encryption Standard (AES) algorithm [5], a de facto industry standard for encryption.

Heterogeneous computing is one more technological trend. Heterogeneous systems may combine different types of computational units, suitable for different tasks. For example, some computers these days contain a CPU to solve general purpose tasks and a GPGPU that helps to accelerate parallelizable tasks. Another instance is all-programmable system-on-a-chip products. These devices contain a hard processor system and field-programmable gate array fabric on the same silicon die. They are capable of solving the same problems as GPGPU-based platforms, but have a more fine-grained structure that gives the developer more freedom for building a custom architecture.

## 1.2 Contributions

This thesis presents the design and evaluation of a ChaCha20 cypher in heterogeneous all-programmable SoCs. Specifically, the thesis makes the following contributions:

- Presents a design and an implementation of an IP-core for efficient ChaCha20 computation in FPGA. To the best of our knowledge, this is the first freely available ChaCha20 accelerator for FPGA, easily connectible to an HPS.
- Describes a software framework based on the Linux environment that allows instantiation of our IP-core and its evaluation and comparison to software cyphers

running on multiple CPU cores in order to evaluate and compare performance of different configurations.

- Evaluates the performance of different configurations of our hardware accelerator, demonstrating its superiority over the software solution. The throughput of our hardware solution is 120.5 MiB/s, which is 5 times higher than the maximum throughput of a software solution running on two CPU cores.

- Identifies DRAM bandwidth as the main bottleneck, limiting the performance of our IP-core.

## 1.3 Thesis outline

The remaining sections of this thesis are organized as follows. Chapter 2 gives background information that is essential for understanding the other sections of this thesis. Chapter 3 overviews existing work that is related to the topic. Chapter 4 explains design choices we made in order to build an efficient hardware accelerator. Chapter 5 describes the design of our IP-core for accelerating ChaCha20. It also describes our software framework for file encryption with hardware acceleration or without it. Chapter 6 lists experiments we carried out, presents their results and gives some analysis of those results. Finally, Chapter 7 summarizes the work, draws conclusions, and discusses future work. Additionally, Appendix A describes the procedure of enabling the SDRAM-to-FPGA bridge, which was essential for the practical part of this thesis.

# Chapter 2

# BACKGROUND

This chapter contains information that is useful for better understanding of the other chapters of this thesis. First, we explain what stream cyphers are and how they are different from block cyphers. Then, we describe the stream cypher of our interest, ChaCha20. We also give some information about the platform we used for our experiments, as well as some technologies we used to implement ChaCha20 in hardware and interface our implementation to software.

## 2.1 Block vs. stream ciphers

According to Christof Paar [6], encryption algorithms can be divided into two groups: symmetric and asymmetric. In the former, the same secret key is used to encrypt and decrypt the data, whereas for the latter, encryption and decryption keys are different.

Symmetric cryptography algorithms are subdivided into two subgroups: block ciphers and stream ciphers. In both cases a certain transformation is applied to a plaintext (the original message that needs to be encrypted), yielding a cyphertext (the encrypted message). This transformation is assumed to be irreversible in practice

without knowing the secret key. The way such a transformation is applied to the plaintext is different for block and stream ciphers.

In block ciphers the plaintext is split into blocks, size of which is algorithm-specific. Then, each block of the plaintext is transformed into a block of the ciphertext by applying some encryption function as follows:

$$CB = E(Key, PB) \tag{2.1}$$

where $CB$ is a block of the ciphertext; $PB$ is a block of the plaintext; $Key$ is the secret key; and $E$ is the encryption function.

In real block ciphers each bit of $CB$ depends of each bit of $PB$. Moreover, changing a single bit of $PB$ alters $CB$ in an unpredictable way. This property differentiates block ciphers from stream ciphers.

In stream cyphers the encryption function is applied independently to each bit of the plaintext. In other words, the encryption process is described as follows:

$$C_i = E(Key, P_i, i) \tag{2.2}$$

where $C_i$ is the $i$-th bit of the ciphertext; $P_i$ is the $i$-th bit of the plaintext; and $E$ is the encryption function.

The result of $E$ depends on the index, $i$, of the bit being encrypted. If $E$ was a function of $P_i$ only, the ciphertext would be either a bitwise inverted or non-modified plaintext. This, obviously, would give no protection at all.

Usually, the $E$ function is implemented as a bitwise XOR operation between $P$ and a pseudo-random number derived from $Key$. It is important to choose this number such that it is not shorter than $P$ and make sure it is used only once. Moreover, it should be impossible to get $Key$ from the pseudo-random number. Without these two properties a stream cypher cannot be secure.

As opposed to block ciphers, changing $P_i$ in a stream cipher affects only $C_i$, but not $P_j$ when $j \neq i$. This may give a malicious party a much better chance to attack a specific part of the plaintext by altering the cyphertext. That is why if the attacker can potentially modify the cyphertext, stream cyphers must be used together with authentication algorithms.

According to [6], currently, block ciphers are used more frequently than stream ciphers despite the fact that stream ciphers can provide similar or better performance. That is why, stream cyphers have a great potential for improving performance of cryptographic applications.

There are encryption algorithms that were designed as block ciphers, but can be turned into stream ciphers. AES [5], a de facto industry standard for symmetric encryption, is an example of such an algorithm. When used in the electronic codebook mode (ECB), it works as a block cypher. In the counter mode, AES works as a stream cypher.

The next section describes a relatively new stream cypher, ChaCha20. Even though it is less flexible than AES, because it can only be used in the stream mode, it is more efficient and secure, and its computation involves only relatively low-complexity

operations. These properties make it promising for cryptographic applications, including file encryption.

## 2.2 ChaCha20

ChaCha20 is a stream encryption algorithm proposed in 2008 by Daniel J. Bernstein. The algorithm is based on Salsa20: another encryption algorithm proposed by Bernstein earlier. According to the author, Salsa20 can be used instead of AES in applications where the confidence in the cypher's security can be sacrificed in favor of speed [7]. ChaCha20 has even better characteristics: it provides higher confidence, while being consistently faster than AES on machines without hardware accelerators [4]. These properties allowed ChaCha20 to earn recognition in the cryptographic community [3]. Moreover, the algorithm was standardized in two documents RFC7539 [8] (now obsolete) and RFC8439 [9]. The novelty of ChaCha20 and its superb characteristics make it a good candidate for hardware implementation for this research.

ChaCha20 is a classic stream cypher: it produces a stream of pseudo-random bytes (one-time pad or OTP) that is XORed with the plaintext to produce the cyphertext. Figure 2.1 demonstrates how a 512-bit chunk of such a stream is generated according to RFC8439. First, the initial 512-bit state of ChaCha20 must be formed. This state consists of a few fields that are explained in Table 2.1. Second, the initial state is transformed by 20 round functions of two types: even and odd. Each round function converts its 512-bit input into a 512-bit output. Finally, the initial state is added to the result of the last round function. To do the summation stage, both

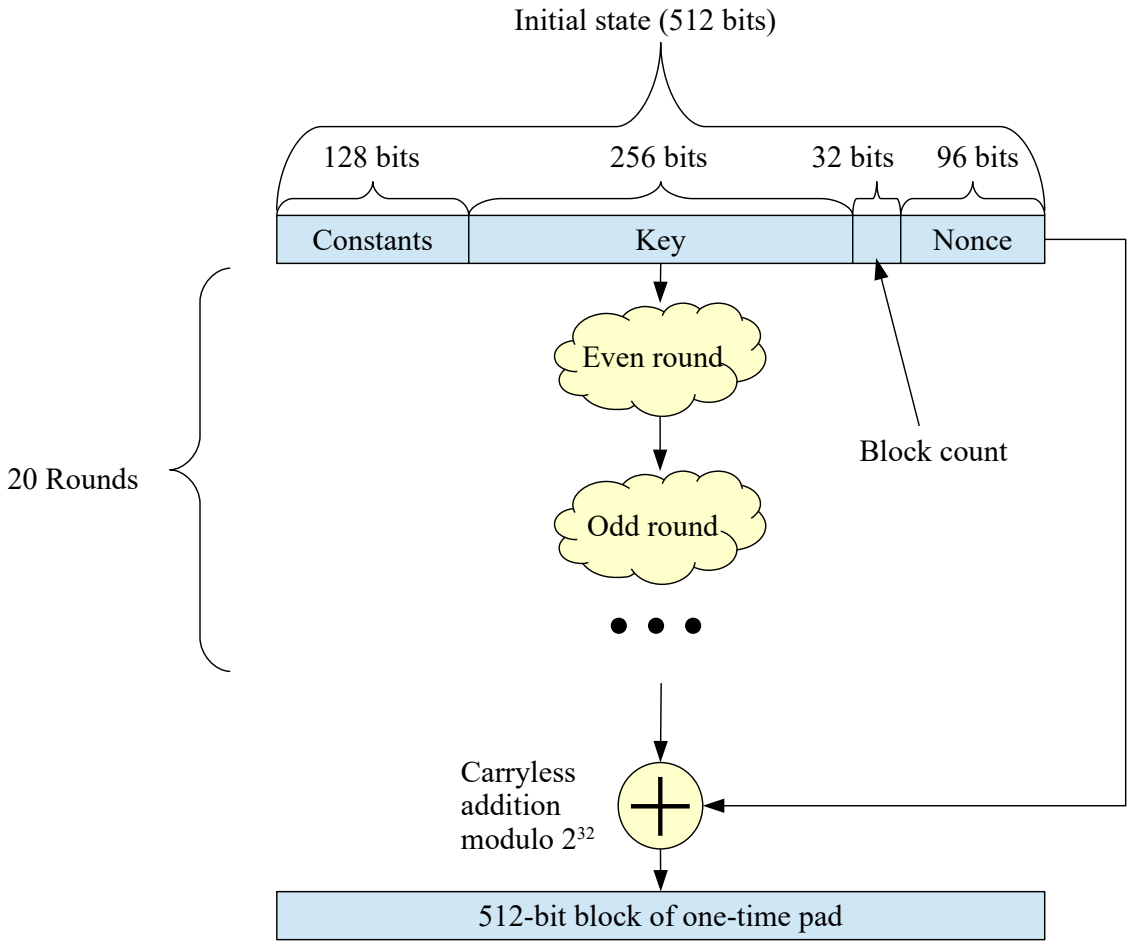



**Figure 2.1**: Graphical illustration of ChaCha20 algorithm

operands are viewed as arrays of 32-bit integer numbers with 16 elements. The summation modulo $2^{32}$ is done element-wise between the arrays. The result of the summation is used to do encryption as follows:

$$C_i = P_i \oplus PAD_i \tag{2.3}$$

where $C_i$ is the $i$-th bit of the ciphertext; $P_i$ is the $i$-th bit of the plaintext; and $PAD_i$ is the $i$-th bit of ChaCha20 result.

**Table 2.1**: Fields of ChaCha20 initial state

| Name | Bit size | Byte size | Description |
|---|---|---|---|
| Constant | 128 bits | 16 bytes | A constant part of the state. It is always initialized with the ASCII values of the characters of the following string: 'expand 32-byte k' |
| Key | 256 bits | 32 bytes | The secret key. |
| Block count | 32 bits | 4 bytes | A counter that starts from 0 and is incremented to generate the next chunks of OTP. There are $2^{32}$ unique values of this field, so the maximum length of the entire OTP with the same secrete key and nonce is 256 GiB. |
| Nonce | 96 bits | 12 bytes | A unique number that can be changed to generate a new OTP with the same key. Using each value of nonce no more than once is crucial for providing a high level of security. |

To generate the next 512-bits of OTP, the the block count field of the initial state is incremented and the process repeats.

The final result of ChaCha20 is extremely sensitive to changes of the initial state: flipping even a single bit of the input leads to an unpredictable change of the result. Moreover, despite the round functions being reversible, it is impossible to convert the result of ChaCha20 back to the initial state, because of the summation stage. This two properties make ChaCha20 suitable for stream encryption.

The round functions consist of one layer of quarter rounds, each working on 1/4 of the round's input. Figure 2.2 shows the structure of the quarter round. The input of the quarter round consists of 4 32-bit words taken from the input of the round. Even and odd rounds supply the input to the quarter rounds differently. Tables 2.2

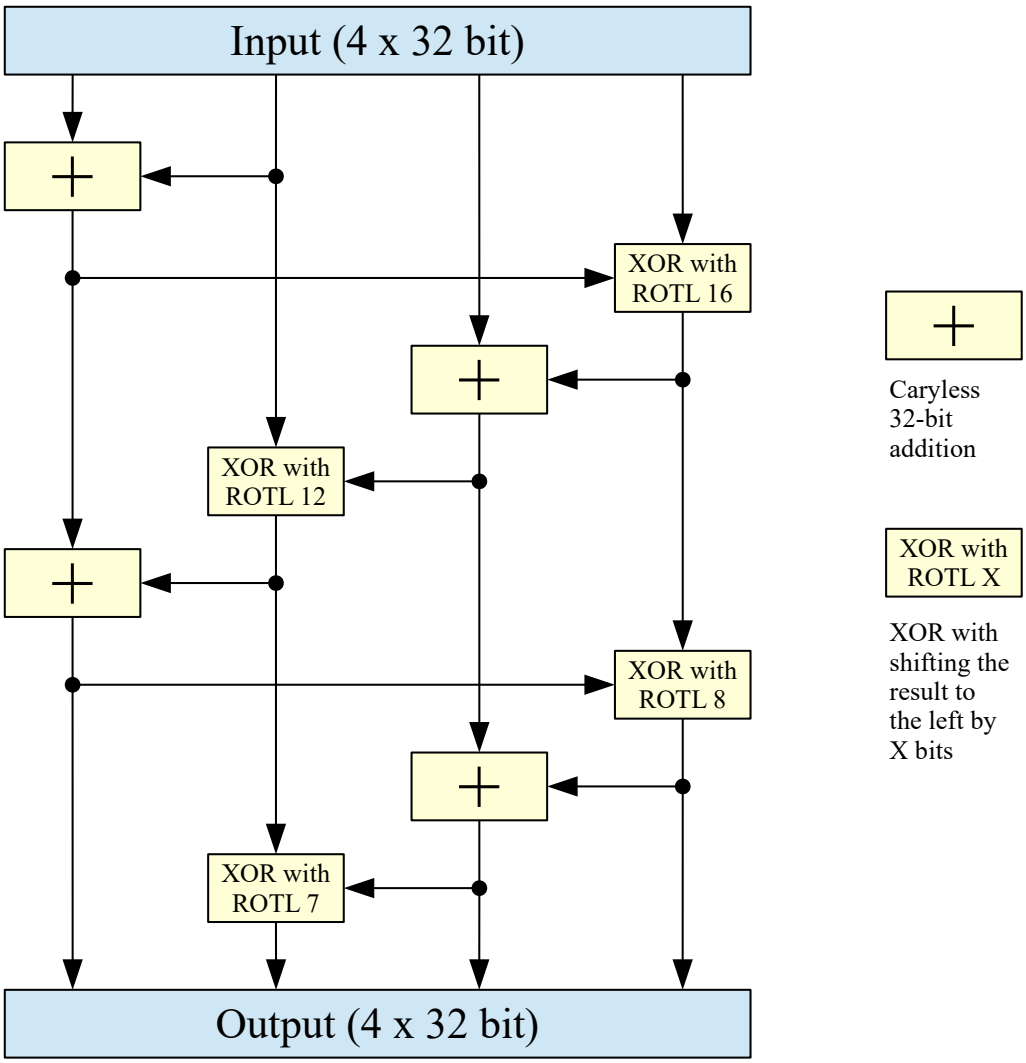



**Figure 2.2**: Graphical illustration of ChaCha20 round

and 2.3 show how the input words of the even and odd rounds are supplied to their quarter rounds.

High performance of ChaCha20 can be ascribed to the fact that it is based on a few primitive operations that execute in a minimum number of clock cycles on modern 32-bit and 64-bit processors:

- Carryless addition of two 32-bit numbers: $a + b \mod 2^{32}$.

**Table 2.2**: ChaCha20 even round input words

| Quarter round number | Input words for the quarter round |
|---|---|
| 1 | 0, 5, 10, 15 |
| 2 | 1, 6, 11, 12 |
| 3 | 2, 7, 8, 13 |
| 4 | 3, 4, 9, 14 |

**Table 2.3**: ChaCha20 odd round input words

| Quarter round number | Input words for the quarter round |
|---|---|
| 1 | 0, 5, 10, 15 |
| 2 | 1, 6, 11, 12 |
| 3 | 2, 7, 8, 13 |
| 4 | 3, 4, 9, 14 |

- Exclusive OR between two 32-bit numbers: $a \oplus b$.
- Rotation of a 32-bit number by a fixed number of positions: $ROTL^n(a)$.

Even though having only such simple operations in an encryption algorithm may seem insecure, Bernstein points out that his operators can simulate any circuit and therefore provide the same security level as a set of more complex operations [7].

Having only such simple operations makes ChaCha20 attractive for implementation in hardware. We used Intel Cyclone V FPGA device for this purpose. The next section describes the characteristics of this device.

## 2.3 Intel Cyclone V

Intel Cyclone V is a family of affordable field-programmable gate array (FPGA) devices. Some sub-families of Cyclone V are called System-on-a-Chip (SoC) because they combine an FPGA and an ARM-based hard processor system (HPS) on a single die. The following sub-families are available, targeting various applications with different requirements for cost and performance [10]:

- Cyclone V E — lowest power, low cost, general logic and DSP applications
- Cyclone V GX — additionally contain 3.125 GiB/s transceivers
- Cyclone V GT — contain 6.144 GiB/s transceivers instead
- Cyclone V SE — same as E but with HPS
- Cyclone V SX — same as GX but with HPS
- Cyclone V ST — same as GT but with HPS

The chips are manufactured using the 28 nm TSMC Low-Power technology and require only 1.1 V core voltage. The FPGA part of Cyclone V allows the engineer to build custom hardware accelerators out of basic blocks called ALMs (Adaptive Logic Modules). Each ALM contains an 8-input adaptive look-up table (LUT), 4 flip-flops, and two full adders. In addition to ALMs, Cyclone V provides digital signal processing (DSP) blocks, which include a 64-bit accumulator, a hard pre-adder supporting 18 and 27-bit modes, and cascaded output adders for systolic finite-impulse-response filters. Cyclone V also contains embedded memory blocks, each providing 10 Kibit of space in

a fast dual-port static RAM (SRAM) [10]. Although the ability to reconfigure FPGA fabric introduces some overhead compared to application-specific integrated circuits (ASICs), it is ideal for prototyping and building accelerators for products with low production quantity.

The HPS part of the chip consists of an ARM-based processor, a shared multi-port DRAM controller, and a few peripheral devices. These devices include a SD/MMC card controller, an Ethernet MAC, a USB OTG, a NAND Flash controller, and a DMA controller. The HPS provides additional performance boost in case the developer needs a CPU and various external interfaces in the system. Even though FPGA technology allows building an entire CPU out of ALMs and the other building blocks, it is not the most efficient solution due to the overhead inherent in FPGAs. Having the HPS eliminates this overhead. Additionally, designing a custom high-performance CPU is not a trivial task. Instead, a third-party IP-core may be purchased, but such a solution may be costly. Cyclone V gives the developer a powerful processor for a low price.

The HPS part of Cyclone V can be interfaced with the FPGA fabric through one of the following bridges:

- FPGA-to-HPS bridge. This bridge allows hardware modules implemented in FPGA to access peripherals on the HPS side. It is also possible to have coherent access to DRAM. This means that the CPU's cache will be aware of DRAM content changes and will invalidate corresponding cache lines. The bridge supports burst transactions, so that multiple contiguous reads or writes can

be done in a sequence. The maximum data width supported by this bridge is 128-bit. Having all these features, the bridge is suitable for high-throughput data transfers initiated from the FPGA side.

- HPS-to-FPGA bridge. This bridge allows ARM core and HPS-side DMAs to access data on the FPGA side. The bridge supports burst transactions and has maximum data width of 128-bit. Thus, it is suitable for high-throughput data transfers initiated from the HPS side.

- Lightweight HPS-to-FPGA bridge. This bridge also allows the ARM core to access the data on the FPGA side, but it is primarily used for non-intensive traffic. A good example of such traffic is reading and modification of control and status register of custom hardware peripherals. Having a dedicated interface for such communication offloads the high-speed HPS-to-FPGA bridge and improve overall performance.

- FPGA-to-SDRAM bridge. This bridge allows FPGA peripherals to non-coherently access DRAM. That is, the CPU's cache will not be aware of changes in DRAM. The interface supports burst transactions and its maximum data width for unidirectional (read or write only) access is 256 bit. Thus, this interface is suitable for high-throughput data transfers initiated from the FPGA side.

## 2.4 Terasic DE10-Nano Kit

Terasic DE10-Nano Kit is a Cyclone-V-based FPGA development board suitable for application in embedded systems due to its compact size and low cost. The photo

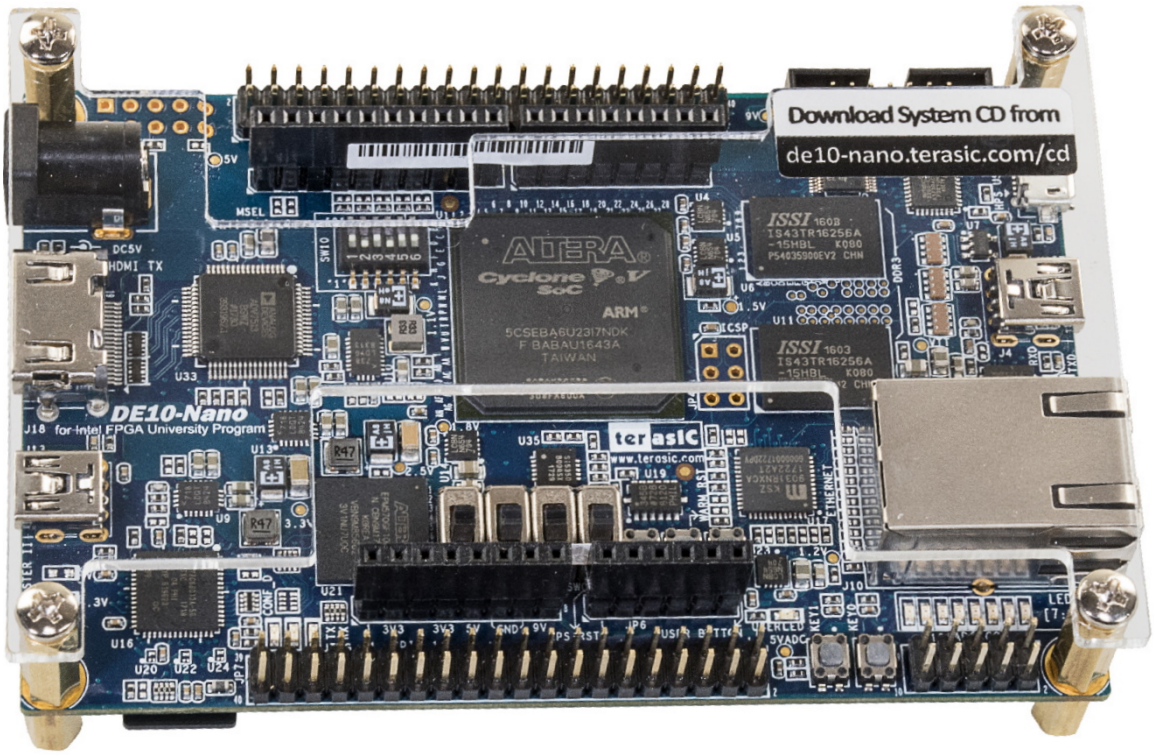



**Figure 2.3**: Photo of DE10-nano board

of the board is shown in Figure 2.3. Figure 2.4 depicts the resources that the board has. The most important of them are the following:

- Cyclone V SE 5CSEBA6U23I7NDK FPGA chip, consisting of two parts:
  - FPGA with 32,070 ALMs [12]. The number of ALMs is equivalent to 85,000 logic elements. The FPGA part is relatively big and has even more logic than is needed for this work.
  - 800MHz dual-core ARM Cortex-A9 CPU. The processor is capable of running a fully-fledged Linux-based operating system.

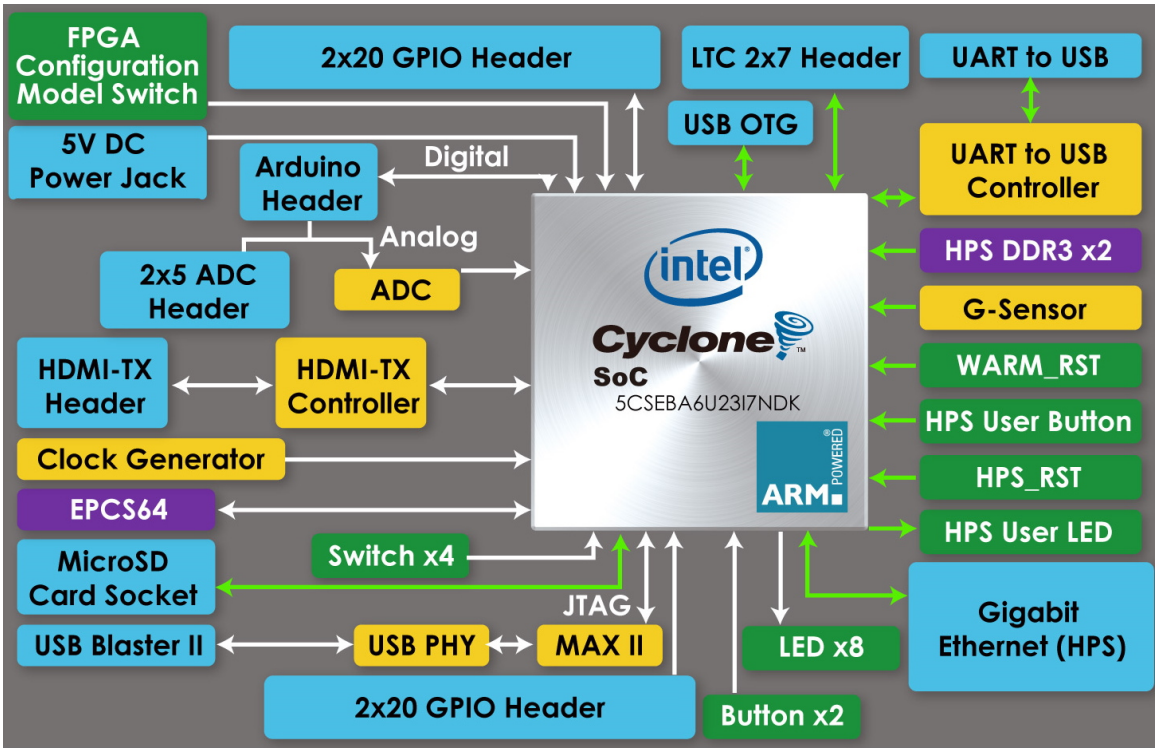



**Figure 2.4**: Block diagram of DE10-nano board [11]

- 1GB of DDR3 memory with a 32-bit data bus. This memory can be used as RAM for Linix-based OS. It can also work as a storage for data produced by custom FPGA modules.

- 1 Gigabit Ethernet PHY. The network connection this PHY provides is useful for interacting with Linux running on the board remotely using SSH protocol.

- Micro SD card socket. SD card can be used to store Linux kernel and root file system. It is also possible to store there FPGA configuration as an `.rbf` file and load it to the FPGA fabric before booting Linux.

- UART to USB converter, connected to a UART on the HPS side. Even though having a network interface covers most needs for the user interaction with Linux,

the network is only initialized after Linux has booted. Thus, it is useful to have access to UART for observing boot process or changing boot configuration in U-boot, a Linux loader. In this work U-boot was helpful to activate FPGA-to-SDRAM bridge (see Appendix A).

- Three 50 MHz clock generators. These clock sources can be used to clock custom FPGA logic. If needed, the frequency of 50 MHz can be multiplied using on-chip phase locked loop (PLL) modules.

## 2.5 Avalon interfaces

Avalon is a group of interfaces available for use in Intel FPGAs. These interfaces unify interaction between hardware modules. The group includes the following interfaces that accommodate various needs [13]:

- Avalon-MM (Avalon Memory Mapped Interface) — an interface for reading/writing data from/to devices that are or look like memory.
- Avalon-ST (Avalon Streaming Interface) — an interface for handling unidirectional streams of data.
- Avalon Interrupt Interface — an interface for notifying hardware components about events that they need to react on.
- Avalon Conduit Interface — an interface that encapsulates a custom set of signal that are not encompassed by any other Avalon interfaces.

The following subsections describe Avalon interfaces that are essential for this work: Avalon-MM, Avalon-ST and Avalon Interrupt.

### 2.5.1 Avalon-MM interface

Avalon-MM interface connects memory-like peripheral devices to a CPU or a DMA. Memory-like means that such a device must support reads or/and writes at a certain range of addresses. However, it does not mean the the device must have internal memory. The memory-like device is called the Slave (S) and the controlling device is called the Master (M).

Data transfers through Avalon-MM happen as follows. On the first clock cycle M asserts either `read` or `write` signal to indicate that it wants to read or write data to S. If it is a read transaction, on the same clock cycle M sets the address it wants to read from. If it is a write transaction, M also sets the data it wants to write on the `writedata` bus. On the next clock cycle, M observes the `waitrequest` signal, controlled by S. If this signal is asserted, M keeps all the signals it controls for one or more cycles until `waitrequest` is deasserted. Once this happens, M can read the `readdata` in the case of a read transaction. For a write transaction, having `waitrequest` deasserted means that the transaction was successful.

Table 2.4 lists most important signals that are used in the Avalon-MM interface. Additional information about the interface's features and signals can be found in [13].

In our system we used the Avalon-MM interface to give the CPU access to control and status registers of our peripheral devices. Through these registers the CPU can configure peripheral devices and observe their current state. More details

**Table 2.4**: Signals of Avalon-MM interface

| Name | Direction | Description |
|---|---|---|
| `address` | M→S | Master uses this signal to let the slave know which address it is going to read or write. |
| `read` | M→S | Master uses this signal to signalize that it is going to read data from the slave. |
| `write` | M→S | Master uses this signal to signalize that it is going to write data to the slave. |
| `readdata` | S→M | Slave outputs the data for reading by the Master through this bus. |
| `writedata` | M→S | Master outputs the data for writing to the Slave through this bus. |
| `waitrequest` | S→M | Slave uses this signal to indicate that it cannot process the current read or write request from the Master. The Master must repeat its request on the next clock cycle if this signal is asserted. |

about control and status registers of our peripherals can be found in Sections 5.1.1 and 5.1.2.

### 2.5.2 Avalon-ST interface

Avalon-ST interface is used to connect a data provider to a data consumer. The data provider is called source (SRC) and the data consumer is called sink (SNK). Usually a single data item is transmitted per clock cycle. Its size is static, but can vary for different sources and sinks. SRC has the option not to send the data on a specific clock cycle. It deasserts the `valid` signal to notify SNK about this. If a valid data item is transmitted, the `valid` signal must be asserted. SNK has the option not to accept the data on any clock cycle. It deasserts the `ready` signal on the next clock

**Table 2.5**: Signals of Avalon-ST interface

| Name | Direction | Description |
|---|---|---|
| `valid` | SRC→SNK | SRC uses this signal to indicate that its trying to transfer valid data to SNK. |
| `data` | SRC→SNK | SRC uses this bus to transfer data to SNK. |
| `ready` | SNK→SRC | If SNK is unable to accept data from SRC, it asserts this signal to stall SRC. |

cycle to notify SRC about this. SRC must repeat data transfer until SNK asserts the `ready` signal.

Table 2.5 lists most important signals that are used in the Avalon-ST interface. Additional information about the interface's features and signals can be found in [13].

In our system we used the Avalon-ST interface to output data from ChaCha20 accelerator (see Section 5.1.1). This interface allows us to supply data to the next stage, a custom DMA, only if the data is ready to be transmitted: the accelerator produces valid data only once every 20 clock cycles. Moreover, if the consumer cannot handle the data (for example when DRAM controller is busy and DMA cannot access it) the `ready` signal can stall the accelerator. More information about functioning of our DMA ChaCha20 accelerator can be found in Sections 5.1.1 and 5.1.2.

### 2.5.3 Avalon interrupt

Avalon interrupt interface connects an interrupt sender to an interrupt receiver. Such a connection is useful when a slave device needs to notify a master device about some event as soon as possible. Usually this notification is handled by the master

device in a timely manner, because master pauses its previous routine and processes the signal.

We used the Avalon interrupt interface in our system to connect our IP-core to the CPU. This allows the CPU to schedule a new computation as soon as the previous one is finished and its result is transferred to DRAM. Using an interrupt in this scenario minimizes the idle time of the IP-core. Section 5.1.2 contains more detailed information about interrupts in our system.

## 2.6 Userspace I/O device drivers

In Linux-based operating systems (OS) applications are executed in a virtual address space and with limited privileges. This restricted environment is called user space. By keeping applications in the user space, the OS protects itself and other applications from unauthorized access. This approach results in inability of regular applications to directly interact with memory-mapped hardware devices: they are controlled by reading and writing at dedicated addresses of CPU's physical address space, which is unavailable to user space programs.

Contrary to user-space applications, Linux kernel runs with higher privileges and thus can directly control devices (can function as a device driver). However, it is not always feasible to modify the kernel to add support of a custom device. That is why Linux-based OSs can be extended by adding kernel modules. Such modules can be loaded dynamically (when the kernel already runs). Even though the perspective of developing a device driver as a kernel module looks attractive, this task is hard: a bug in the module can crash entire OS.

Generic Userspace Input/Output (UIO) driver [14] with interrupt handling support (a.k.a. `uio_pdrv_genirq` driver) is a good alternative to custom kernel modules. This module is a part of the Linux kernel code base and can be enabled by configuring the kernel before its compilation.

This device driver covers most needs for custom memory-mapped hardware devices. Through this driver user space programs can access device's control status registers (CSR) as well as handle interrupts. As a result, the major part of the device interaction logic can be moved to the user space, making the debugging task much easier as program errors do not crash entire system. The UIO driver even makes possible controlling devices directly from a program written in a higher level language such as Python.

Whenever the `uio_pdrv_genirq` module is loaded, it creates a device file `/dev/uioX`, where `X` is the device number. This file serves for both CSR access and interrupt handling. To access device registers the users space program opens this file using `open()` function and makes `mmap()` system call. The `mmap()` function returns a pointer to a virtual memory region that is mapped to the physical addressees of device CSRs. By modifying and reading that region, the program can control the device and observe its status.

To react on interrupt requests (IRQs), the user space program does a blocking read from the `/dev/uioX` file by calling to the `read()` function. This operation suspends the calling thread until the device triggers an interrupt. An interrupt event unlocks the suspended thread, so it can process the interrupt request. Usually, when reacting on an IRQ, device drivers write to certain CSR to deassert the interrupt

line an be able to leave the interrupt service routine. However, the generic UIO driver is not aware of the functionality of CSRs of a specific device. Thus, to clear the interrupt, it disables the channel of the interrupt controller that belongs to the processed interrupt. This means that the users space program, which is aware of control and status register functionality, has to modify one of the registers to clear the pending interrupt. Finally, to be able to accept new interrupts the users space program does a file write operation on the `/dev/uioX` file. This operation activates the interrupt channel that was previously disabled.

There is an alternative to the generic UIO driver: user space programs can access physical address space by calling `mmap()` on the `/dev/mem` device file. This method provides access to any physical addresses including those that are mapped to CSRs of a custom device. The main advantage of this method is its availability for virtually any precompiled kernel (`/dev/mem` is enabled by default). To enable UIO support one often needs to configure and build their own kernel. However, the `/dev/mem` has a serious drawback: it does not support handling interrupts. That is why we used the generic UIO driver in our work instead of `/dev/mem`.

## 2.7 User space mappable DMA Buffer

Direct Memory Access (DMA) is an important feature in a system that is expected to have a hight throughput. DMA can unload the CPU from moving memory content and reduce overhead of this process. DMA modules usually expect to have access to a physically contiguous region of memory. Operating system needs to be aware of this buffer so that it does not use it to store some other data. One may think

of using regular `malloc()` call to allocate space for a DMA buffer. This, howerer, will not work: `malloc()` allocates a virtually contiguous region, but does not guarantee its physical contiguity. `/dev/mem` will not work either: although it can give a user space program access to physical memory in a contiguous manner, it cannot prevent the OS from using the buffer for other purposes.

This problem is usually solved by using `kmalloc()` function, which can provide physical contiguity. However, it is only available in the kernel space, which requires developing a kernel module.

An alternative solution is the udmabuf Linux device driver developed by Kawazome Ichiro [15]. This kernel module can reserve a contiguous chunk of memory of required size. The user space code can easily map this buffer to its virtual address space and find out its physical address to properly configure DMA using a UIO driver. Since udmabuf provides a buffer that satisfies all the requirements for DMA, we used it in our work.

# Chapter 3

# RELATED WORK

Developing custom cryptographic accelerators in hardware is a popular topic in scientific and engineering literature. Some articles feature an FPGA-based chip or a hybrid SoC as the target platform. Other use ASIC (application-specific integrated circuit) or do not mention any platform at all, discussing only a general architecture of the proposed design. Many publications feature AES as the accelerated algorithm. A few projects are devoted to acceleration of ChaCha20.

Cowart *et al.* present and discuss the results of their experiments where they measure the performance of hardware AES accelerators implemented in an all-programmable SoC [16]. Their target platform is Zedboard, a Zynq-7000-based development board. The authors discuss two AES IP-cores placed in the FPGA fabric: a non-pipelined core for the ECB and CBC AES modes, and a fully pipelined core for the CTR (counter) AES mode. They compared the performance of these cores to that of the OpenSSL library compiled for the ARM architecture. As a result, they did not manage to see a significant speedup for the non-pipelined core, but demonstrated that the pipelined one is almost 7 times faster than OpenSSL. The maximum performance

observed for the non-pipelined core was 25 MiB/s and 350 MiB/s for the pipelined one.

Baskaran and Rajalakshmi present an AES accelerator fabricated using a 0.18-µm CMOS technology [17]. Their design supports the ECB (electronic codebook), OFB (output feedback), and CBC (cipher block chaining) modes of operation of AES and runs at the frequency of 330 MHz. The accelerator can be configured using a memory-mapped interface of a LEON 32-bit (SPARC V8) processor. The maximum throughput they manage to get in such setting is 480 MiB/s.

Silex Insight implemented ChaCha20-Poly1305 Crypto Engine as an IP-core [18]. The core supports the Authenticated Encryption with Associated Data (AEAD) mode. In such a mode only a part of the input message is encrypted. The other part is transmitted as is. However, both parts are authenticated [19]. The core is claimed to have multi Gibit/s speeds. However, the company's official website does not give any information about the core's exact throughput [20], so it is difficult to evaluate it.

Kanda and Ryoo propose an accelerator supporting ChaCha20-Poly1305-based AEAD [21]. Their target platform is Virtex 7, a high-end family of FPGAs from Xilinx. Their design can run at the clock frequency of 161.02 MHz, giving the resulting throughput of 515 MiB/s for ChaCha20 encryption without message authentication. The resource utilization of their design is 1692 LUTs and 566 registers. No information is provided about the way this core can be connected to a CPU.

Another work describes a compact co-processor that facilitates ChaCha computation as well as the computation of BLAKE and Skein — ChaCha-based hashing algorithms [22]. Instead of directly implementing ChaCha rounds, the authors propose

**Table 3.1**: Summary of the performance characteristics for various hardware cyphers

| Source | Algorithm | Platform | Frequency | Performance |
|---|---|---|---|---|
| Cowart *et al.*[16] | AES CBC | Zynq-7000 | 100 MHz | 25 MiB/s |
| Cowart *et al.*[16] | AES CTR[1] | Zynq-7000 | 100 MHz | 350 MiB/s |
| Baskaran and Rajalakshmi[17] | AES | 0.18-µm CMOS | 330 MHz | 480 MiB/s |
| Kanda and Ryoo[21] | ChaCha20 | Virtex 7 | 161 MHz | 550 MiB/s |
| At *et al.*[22] | ChaCha20 | Virtex 6 | 266 MHz | 362 MiB/s |
| Strömbergson[23] | ChaCha20 | Cyclone V | 60 MHz | 159 MiB/s[2] |
| This thesis | ChaCha20 | Cyclone V | 50 MHz | 120.5 MiB/s[3] |

an architecture that can efficiently pipeline the primitive arithmetic operations that the algorithm involves. They argue that such an approach can significantly reduce resource utilization, but keep a high throughput. As a result, their architecture provides the throughput of 266 MiB/s at the clock frequency of 362 MHz for ChaCha20 algorithm while consuming only 49 slices and 2 block RAMs on Virtex-6 FPGA.

One more ChaCha20 implementation is available in a GitHub repository [23]. This project contains a standalone ChaCha20 module. According to its author the module has a non-pipelined design. The latency of producing one OTP is 23 clock cycles. When compiled for Cyclone V, this design is capable of working at 60 MHz and consumes 1939 ALMs (basic logic blocks of Cyclone V). This results in 159.22 MiB/s of theoretical throughput. Unlike our solution, this design does not have any processor interfaces, so it cannot be connected to HPS systems without additional modification.

[1] a fully pipelined design is used
[2] theoretical throughput
[3] the result of using two IP-cores for file encryption
[4] custom DMA is not included

**Table 3.2**: Summary of the resource consumption for various hardware cyphers implemented in FPGA

| Source | Algorithm | Platform | Registers | Logic |
|---|---|---|---|---|
| Kanda and Ryoo[21] | ChaCha20 | Virtex 7 | 566 | 1692 LUTs |
| At *et al.*[22] | ChaCha20 | Virtex 6 | — | 49 slices |
| Strömbergson[23] | ChaCha20 | Cyclone V | 1940 | 1939 ALMs |
| Silex Insight [18] | ChaCha20-Poly1305 | Zynq-UP-MPSoC | — | 3769 LUTs |
| This thesis | ChaCha20 | Cyclone V | 1040 | 1440 ALMs[4] |

This thesis is different from the above-mentioned works in several aspects. First, we offer a ready-to-use open-source solution for hardware ChaCha20 acceleration that can easily be integrated in existing processor systems. Second, we provide a software framework that allows interfacing multiple hardware modules and using them for file encryption in the Linux environment. Finally, we carry out experiments that demonstrate the performance of our core under realistic conditions and show how it compares to a multi-threaded software implementation. The experiments have been done for a real-world application, file encryption.

Table 3.1 summarizes the performance of all related implementations mentioned in this section, including the implementation described in this thesis. Table 3.2 gives the summary of the resource consumption of all FPGA-based projects mentioned above. Both tables exclude implementations that do not mention the parameters of interest.

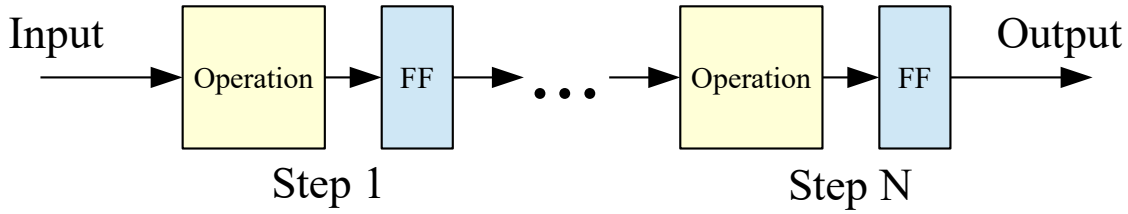
# Chapter 4

# DESIGN CONSIDERATIONS

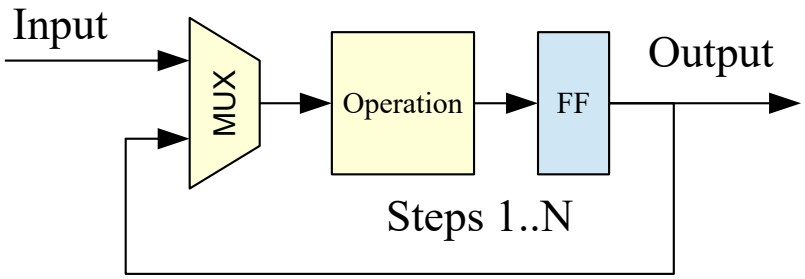
In order to see how beneficial hardware realization of ChaCha20 encryption algorithm in all-programmable SoC can be, we developed a ChaCha20 IP-core in SystemVerilog. In this chapter we describe design decisions made to build a cost-effective and high-throughput IP core.

## 4.1 Pipelining

If an algorithm consists of a sequence of steps and this sequence does not depend on the input data, building a pipeline can greatly increase the throughput of a system. At the same time pipelining may require considerably more resources. If the data produced by a pipelined module cannot be utilized properly by the other modules, the pipeline will stall, leading to poor resource utilization. Alternatively, if all the steps of an algorithm are identical (the algorithm consists of a loop), a non-pipelined design with a feedback loop can be used. On one hand, such design will demonstrate $N$ times lower performance, where $N$ is the number of steps in the algorithm. On the other hand it will consume almost $N$ time less resources. The pipelined and non-pipelined designs are shown in Figure 4.1. Blue rectangles depict




(a) A pipelined design




(b) A non-pipelined design with a feedback loop

**Figure 4.1**: A pipelined design vs a non-pipelined design with a feedback loop

registers and yellow rectangles depict combinational logic. The non-pipelined design contains a multiplexer that feeds the input data to the operation logic on the first step and connects the result of the operation to its input for the following steps. Once $N$ steps are computed, the output becomes valid.

The decision between a pipelined design and a non-pipelined must be justified. ChaCha20 is a perfect candidate for pipelining as it consists of a static sequence of steps. A pipelined ChaCha20 module can produce 512 bits of OTP every clock cycle. Considering the clock rate of 50 MHz, it provides the data rate of:

$$64 \text{ bytes} \cdot 50 \text{ MHz} = 3.2 \text{ GiB/s} \tag{4.1}$$

At the same time, a non-pipelined implementation that computes one round per clock cycle produces 512 bits of OTP every 20 clock cycles. This results in the data rate of:

$$\frac{64 \text{ bytes} \cdot 50 \text{ MHz}}{20 \text{ cc}} = 160 \text{ MiB/s} \tag{4.2}$$

Let us consider the DRAM of the DE1-SoC board as the storage for OTP. The board has two DDR3 memory chips that are connected to the memory controller by a 64-bit data bus. The clock rate of the memory is 400 MHz. Since DDR3 is a double data rate memory (two data transfers happen per clock cycle) the theoretical throughput of this DRAM is:

$$8 \text{ bytes} \cdot 400 \text{ MHz} \cdot 2 = 6.4 \text{ GiB/s} \tag{4.3}$$

Even though DRAM throughput is twice as high as the data rate of the pipelined ChaCha20, it does not mean that the pipelined version can be efficiently utilized. First, it is just a theoretical upper limit: the real speed may be a few times lower due to long memory timings [24]. Second, ChaCha20 module is not the only device accessing DRAM: the software part that does encryption needs to access OTP with the same rate.

Even if DRAM could handle the required data throughput, the FPGA-to-SDRAM bridge (see Section 2.3) would cause an additional limitation. Its maximum data width is 256 bit and its clock rate equals to the clock rate of the FPGA fabric. This means that if a pipelined ChaCha20 accelerator produced 512 bits of data every

clock cycle, the bridge would require 2 clock cycles to send this data to the DRAM controller. Thus, the accelerator would stall every second clock cycle, waiting for the bridge.

One can argue that once a pipelined version is implemented, it can be successfully used in systems without sufficient throughput if the pipeline stalls properly. This can make the module more universal. However, pipelining is costly in terms of chip area. Let us assume that $R$ is the number of logic gates required to build ChaCha20 round function and $S$ is the number of flip-flops needed for a register, holding one ChaCha20 state. In this case, resource consumption of a pipelined module that computes 20 rounds will be $20R$ logic elements and $21S$ flip-flops. At the same time a non-pipelined implementation will take roughly $R$ logic elements and $2S$ flip-flops. That is, for systems with a limited throughput, 95% of logic gates and 90% of flip-flops will be wasted.

Considering the problem with low DRAM bandwidth, an insufficient width of the FPGA-to-SDRAM bridge, and low resource utilization of the pipelined ChaCha20 version, we have chosen a non-pipelined approach.

## 4.2 XOR stage

The XOR stage of ChaCha20 is used to encrypt and decrypt data (see Section 2.2). This operation is unique comparing to the other 21 stages (20 rounds plus the summation stage) of the algorithm, so it requires additional resources that are not used for the rest of the algorithm. As a result, we could expect a low utilization ratio

for the XOR gates involved:

$$\frac{\text{1 clock cycle (XOR stage)}}{\text{22 clock cycles (total number of stages)}} = 4.54\% \tag{4.4}$$

To prudently use FPGA resources, the XOR stage can be handled by software. Keeping this stage on the HPS side should not significantly affect the performance: in Section 6.2.3 we will show that file access together with XOR operation is far from the critical path. Moreover, having XOR stage in HPS avoids moving plaintext (for encryption) or cypertext (for decryption) to the FPGA side and reading back the result: only moving OTP from FPGA to HPS is necessary. As a result, our design choice can reduce HPS-to-FPGA bridge traffic by two times.

## 4.3 Summation stage

The summation stage adds 32-bit words of the initial state to the result of 20 rounds (see Section 2.2). This operation is unique comparing to the other 21 stages (20 rounds plus the XOR stage) of the algorithm, so it requires additional resources that are not used for the rest of the algorithm. As a result, we could expect a low utilization ratio for the adders involved:

$$\frac{\text{1 clock cycle (summation stage)}}{\text{22 (total number of stages)}} = 4.54\% \tag{4.5}$$

To prudently use FPGA resources, the summation stage can be handled by software. Keeping this stage on the HPS side should not significantly affect the

performance. The summation stage requires the same number of instructions as the XOR stage (16 `add` instructions versus 16 `xor` instructions per 512 bits of OPT), which is also handled by software (see Section 4.2). According to the data presented in Section 6.2.3, even if the number of instructions on the CPU side doubles, the CPU part will still be far from the critical path, so we expect no significant performance degradation caused by this design decision.

## 4.4 Summary

As the result of careful consideration the following design decisions have been made:

- A non-pipelined architecture has been chosen;
- Only the round function will be implemented in hardware;
- The summation and the XOR stages will be handled in software.

# Chapter 5

# DESIGN DESCRIPTION

In order to see how beneficial hardware realization of ChaCha20 encryption algorithm in all-programmable SoC can be, we developed a ChaCha20 IP-core in SystemVerilog and designed a software framework to employ it for file encryption and compare it with our software implementation. This chapter describes the design of the IP core and the software framework for evaluating its effectiveness, using a file encryption/decryption as an exemplar workload.

## 5.1 FpgaCha IP-core

FpgaCha is a hardware IP-core for ChaCha20 algorithm acceleration that we implemented to do this research. FpgaCha is packaged as an Intel Platform Designer (an Intel's tool for system integration) subsystem, so it can easily be connected to different processor systems such as Nios II or an ARM-based HPS. Internal modules of FpgaCha are implemented in SystemVerilog. According to the Timing Analyzer, the $F_{max}$ parameter of this core is 54.38 MHz for the Slow 1100 mV 100 C° model (a conservative model). We use a slightly lower frequency of 50 MHz to clock the core as it is easier to derive. The source code of the IP-core is available at [25].

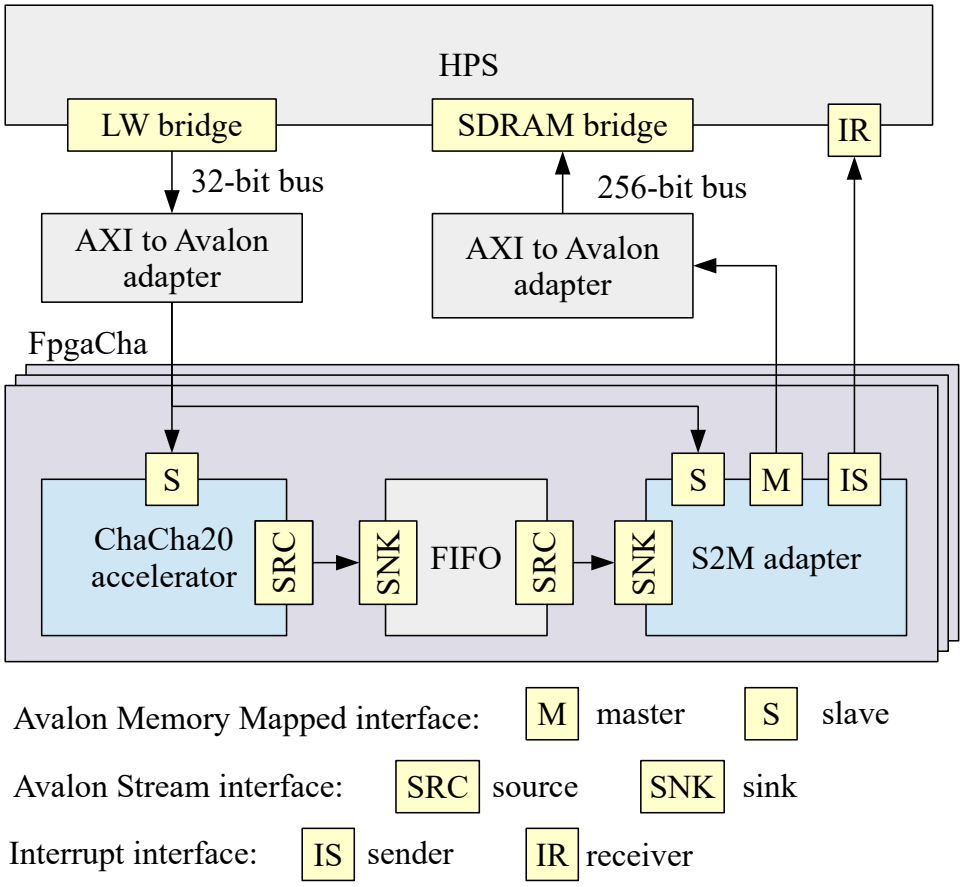



**Figure 5.1**: Hardware components of FpgaCha

Figure 5.1 shows all components of the core and how it is interfaced to to the HPS. Blue rectangles represent custom modules and gray rectangles represent modules already available in the Intel Platform Designer.

FpgaCha IP-core functions as follows. The HPS configures ChaCha20 accelerator and S2M (Avalon-ST to Avalon-MM) adapter through the lightweight HPS-to-FPGA bridge (LW bridge in Figure 5.1). AXI to Avalon adapter connects a simpler Avalon-MM slave interface to the AXI bus that is native to the bridge. ChaCha20 accelerator configuration includes setting encryption parameters (key, nonce, initial block count) and the number of OTP blocks that needs to be produced. S2M adapter

configuration includes setting the target address of SDRAM and the number of 256-bit chunks to transfer (twice the number of OTP blocks set in ChaCha20 accelerator). Right after configuring, ChaCha20 accelerator starts producing OTP blocks. They go to the S2M adapter through a FIFO buffer with the capacity of 16 512-bit items. The buffer helps to avoid stalling ChaCha20 accelerator if S2M adapter cannot accept data for a short period of time. S2M adapter accepts data from the queue and moves it to SDRAM through the FPGA-to-SDRAM bridge. One more AXI to Avalon adapter is used here for connecting the Avalon-MM master interface to AXI bus of the bridge. Once all scheduled transfers are finished, the adapter sends an interrupt request to the HPS. Upon accepting the request, the HPS clears pending interrupt flag in S2M adapter and the cycle repeats if more OTP blocks are needed. Now HPS can access generated blocks in the DRAM.

The following two subsections describe the functionality of our custom modules: ChaCha20 accelerator and S2M adapter in more detail.

### 5.1.1 ChaCha20 accelerator

ChaCha20 accelerator is a custom SystemVerilog module that is capable of computing 20 rounds of ChaCha20 algorithm in 20 clock cycles. The module has a non-pipelined architecture and does not include the summation stage.

From the programmer's point of view the module looks like a set of control registers. Their functionality is summarized in Table 5.1. To configure the module the programmer, first, needs to set up the initial ChaCha20 state (see Section 2.2). Second, the programmer should decide how many OTP blocks the module will generate

**Table 5.1**: Register map of ChaCha20 accelerator

| Name | Byte offset | Description |
|---|---|---|
| `INIT_STATE[0:15]` | 0x0000 – 0x001F | A 16-word (32 bits per word) initial state for ChaCha20 algorithm. This register consists of multiple fields such as key, nonce, and block count (see Table 2.1). |
| `PAD_COUNTER` | 0x0020 | The number of one-time pads (without the summation stage) to generate. The module start computation upon modification of this register. |

and set the `PAD_COUNTER` register. The bigger this number is, the less significant the configuration overhead will be. On the other hand, bigger numbers require more memory to store OTP. Upon modification the `PAD_COUNTER` register, the module starts producing OTP blocks and outputs them through Avalon-ST interface. When generating blocks the module automatically increments the `block count` field of the `INIT_STATE` register.

The data path of the module is shown in Figure 5.2. This diagram demonstrates how a single OTP (without the summation stage) is calculated by the module. Blue rectangles represent registers and yellow clouds depict combinational logic. Initially, the round number register (not accessible by the programmer) contains 0. The register is incremented by 1 every clock cycle until it reaches 19. The current state register is also updated every clock cycle and contains the last result of the round function. In the first round, the current state is computed based on the initial state (configured by the programmer). In the following rounds, the previous value of the current state

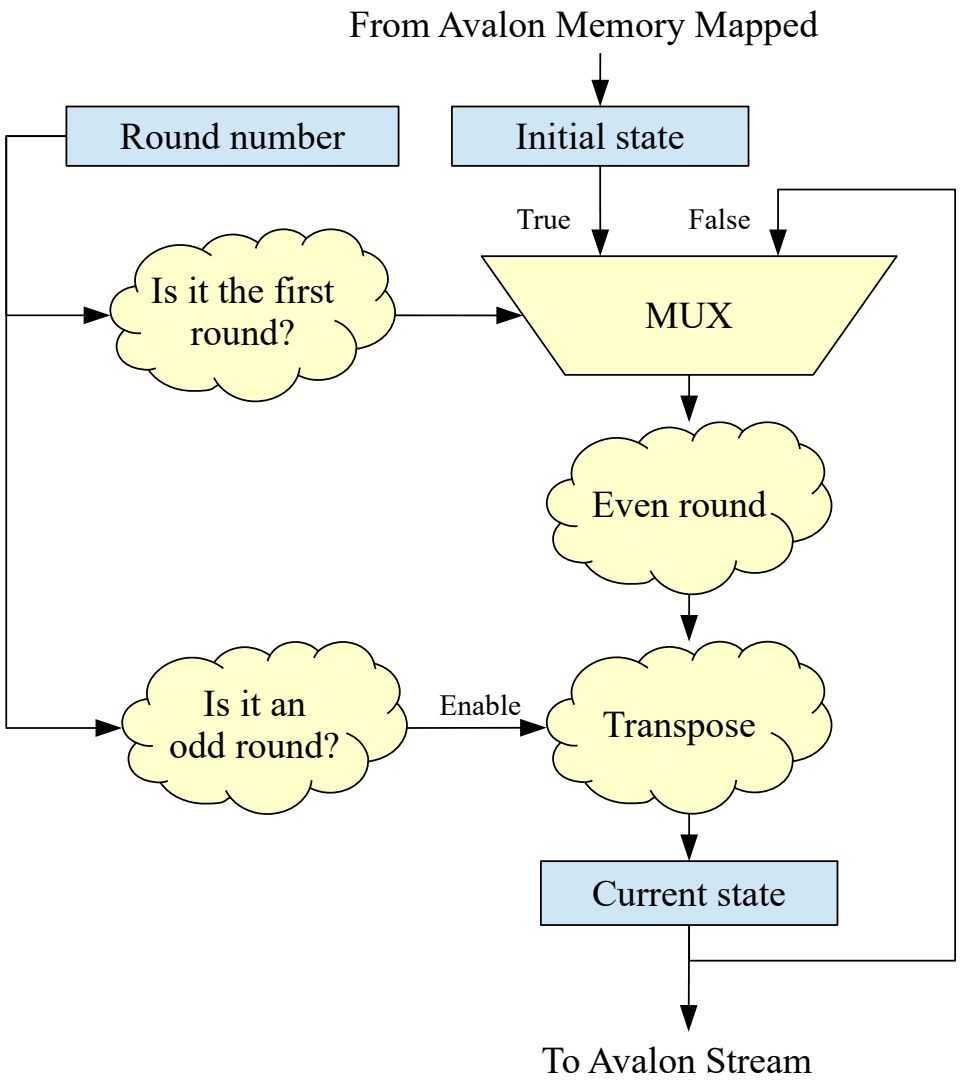



**Figure 5.2**: The data path of ChaCha20 accelerator. Its feedback loop allows computation of multiple ChaCha rounds iteratively

serves as the input for the round function. Once 20 rounds have been computed, the result is available through the Avalon-ST source interface. If Avalon-ST (see Section 2.5.2) sink deasserts the `ready` line, signalizing its inability to accept data, the module stalls, so that no data gets lost. If Avalon-ST sink accepts the data, the block

**Table 5.2**: FPGA resource utilization of ChaCha20 accelerator

| Resource type | Available [12] | Used | Utilization ratio |
|---|---|---|---|
| ALMs | 32,070 | 1,440 | 4.5% |
| Registers | 128,300 | 1,094 | 0.9% |
| Block memory bits | 4,065,280 | 0 | 0.0% |
| DSP blocks | 87 | 0 | 0.0% |

count field in the initial state register is incremented, the `PAD_COUNTER` is decremented, and the process of computing 20 rounds is repeated until `PAD_COUNTER` reaches 0.

The module is quite compact as it takes only 4.5% of the logic resources available in the FPGA chip. Its resource utilization is summarized in Table 5.2

### 5.1.2 S2M adapter

S2M adapter is a custom SystemVerilog module that accepts 512-bit data items through the Avalon-ST sink interface, splits them into two 256-bit chunks and transfers these chunks through the 256-bit Avalon-MM master interface at a certain address configurable by the programmer. To reach better performance, the module does those transfers in bursts of two. In the IP-core the module is used to transfer data from ChaCha20 accelerator to DRAM. Basically, the adapter functions as a DMA, but has a smaller size as it contains only essential functionality. It also supports a 512-bit Avalon-ST interface, which is not supported by the DMAs available in Platform Designer.

From the programmer's point of view the module looks like a set of control registers. Their functionality is summarized in Table 5.3. To configure the module the

**Table 5.3**: Register map of S2M adapter

| Name | Byte offset | Description |
|---|---|---|
| `LENGTH` | 0x0000 | The number of 256-bit chunks of data to be transferred. Modification of this register initiates a data transfer |
| `ADDRESS` | 0x0004 | Starting address in the DRAM for transferring data at |
| `IRQ` | 0x0008 | Modification of this register clears a pending interrupt request |

programmer, first, needs to chose the destination address in DRAM and set it to the `ADDRESS` register. Second, the programmer has to decide how many 256-bit chunks of data should be transfered from ChaCha20 accelerator to DRAM, and set this number to the `LENGTH` register. Since each block of OTP is 512-bit long, this number must be 2 times bigger than the number used in the `PAD_COUNTER` register (see Section 5.1.1) of ChaCha20 accelerator. Improperly chosen value of `LENGTH` will cause undefined behavior. Modification of `LENGTH` starts data transfer. Once `LENGTH` 256-bit chunks have been transferred, the module generates an interrupt, signalizing about the end of the transfer. The programmer is supposed to modify the `IRQ` register to clear the pending interrupt.

S2M adapter supports the `waitrequest` signal from the Avalon-MM slave interface. This signal can stall the module if the DRAM controller is busy. Moreover, if necessary, the module can stall the Avalon-ST source interface, by deasserting the `ready` signal. This feature prevents data loss when the DRAM controller experiences high load. Also, it allows connection of multiple FpgaCha modules to the same DRAM

**Table 5.4**: FPGA resource utilization of S2M adapter

| Resource type | Available [12] | Used | Utilization ratio |
|---|---|---|---|
| ALMs | 32,070 | 186 | 0.6% |
| Registers | 128,300 | 354 | 0.3% |
| Block memory bits | 4,065,280 | 0 | 0.0% |
| DSP blocks | 87 | 0 | 0.0% |

port: the arbitration logic introduced by Platform Designer uses `waitrequest` signal to stall the Avalon-MM masters that lose arbitration.

The module is quite compact as it takes only 0.6% of the logic resources available in the FPGA chip. Its resource utilization is summarized in Table 5.4

## 5.2 Software organization

In order to carry out experiments and compare the hardware solution to the software one, we have implemented a small software framework. The framework is written in C++ programming language. Its source code is freely available at [25].

We designed the framework to satisfy the following requirements:

- support for multi-threading;
- support for multiple hardware accelerators;
- simplicity of adding or removing hardware accelerators and varying the number of threads;
- capability of testing performance of data consumers and producers separately;

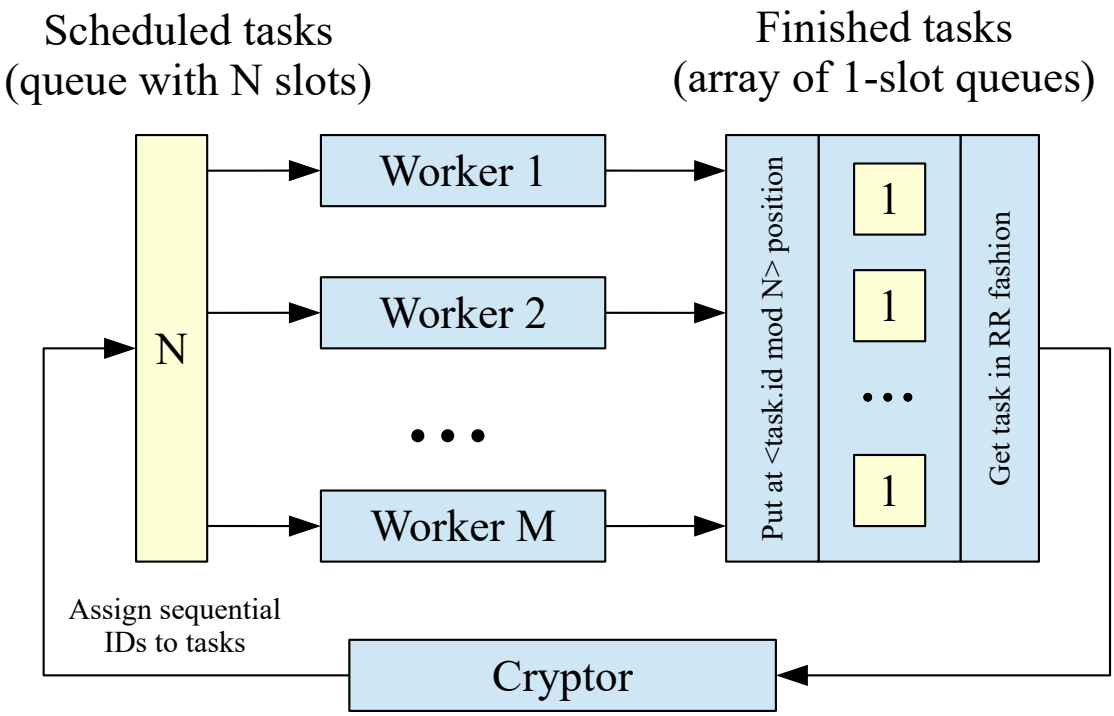



**Figure 5.3**: Structure of our software framework

- the system must be self-balancing: if data producer or data consumer is faster than its counterpart, it must stall.

The structure of the framework is shown in Figure 5.3. It consists of the following components: tasks (objects that represent the need for OTP blocks or produced OTP blocks; the flow of tasks is depicted as arrows), queues (buffers temporary storing tasks; depicted as yellow rectangles), workers (threads that perform tasks by generating OTP blocks), and cryptors (threads that use solved tasks to do data encryption). The following subsections describe those components in detail and explain how they function as a whole.

### 5.2.1 Task

Task is a C++ struct that represents a job that needs to be done on some data. Arrows in Figure 5.3 represent the flow of tasks. The type of work that needs to be done is determined by the consumer of a task. In Figure 5.3 the tasks that enter workers represent the need of the cryptor for OTP. The tasks that exit the workers represent ready-to-use OTP blocks that can be used for data encryption.

Each task contains the following fields:

- id: the sequence number of a task. There are no two tasks with the same id in the system.
- buffer: a pointer to the buffer with the input data (if any) for the task. This buffer is used to store the result (if any) of the task as well.
- length: the number of words available in the buffer.

There is a constant number of tasks in the system. This property helps to prevent excessive memory consumption if the producer of tasks produces them faster than the consumer can handle.

### 5.2.2 Queue

In our framework, queue is a fixed capacity blocking FIFO (first in first out) buffer with the support of a shutdown mode. Such a queue can be efficiently used to synchronize consumer and producer threads even if there are many consumers and producers. In Figure 5.3 queues are depicted as yellow rectangles. The number inside each rectangle specifies the the capacity of the queue.

If a consumer thread tries to extract an element from an empty queue, the thread is blocked until the queue becomes non-empty. If a producer thread tries to place an element in a queue that is full, the thread is blocked until at least one element is extracted from the queue.

The queue supports a shutdown mode. Upon activation of this mode all threads waiting on the queue are notified about its new status. This signal means that no more data needs to be consumed or produced. The shutdown mode is used to safely terminate all threads involved when no more tasks need to be processed.

### 5.2.3 Worker

Worker is a data producer that generates OTP for encryption and decryption. Each worker contains one or more threads. A worker consumes a task from the queue of scheduled tasks and generates a series of OTP blocks, placing them in the buffer. The key and nonce are set when the worker is created. Block count is chosen based on task id such that two consequent task ids match consequent series of OTP blocks.

The following types of workers are available for use:

- StubWorker. This worker takes a task from the queue of scheduled tasks and immediately inserts it in the queue of finished tasks. As a result, the buffer of a finished task keeps its old invalid data. Even though this worker does not produce an OTP that can be used for real encryption, it has a relatively high throughput, and thus, can help to measure the performance of other parts of the system.

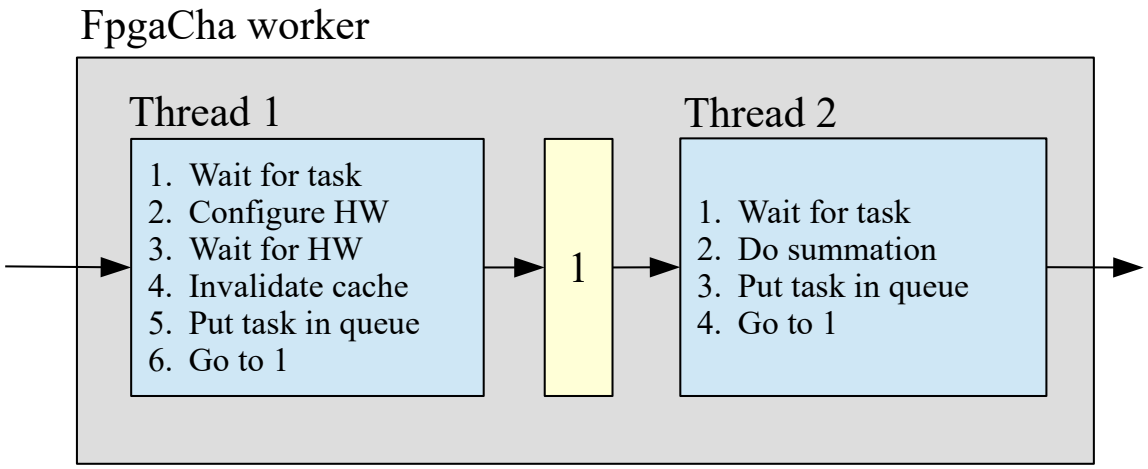



**Figure 5.4**: Structure of FpgaChaWorker

- ChaCha20Worker. This worker takes a task from the queue of scheduled tasks and generates a series of OTP blocks, placing them in the task's buffer. After that, the task is placed into the the queue of finished tasks. This worker employs the software ChaCha20 implementation. More than one ChaCha20Worker can be used to exploit multi-threaded execution.

- FpgaChaWorker. This worker is shown in Figure 5.4. It does the same job as ChaCha20Worker, but employs a single instance of FpgaCha instead of the software ChaCha20 implementation to compute the result of 20 rounds. The summation stage (see Section 4.3) is computed in software.

  FpgaChaWorker consists of two threads. Thread 1 takes a task from the queue of scheduled tasks, configures FpgaCha to calculate a series of OTP blocks (only 20 rounds without the summation stage) to fill the buffer of the task. After that, Thread 1 goes into the wait mode until the computation in hardware is finished. While being in the wait mode, Thread 1 consumes no CPU time, effectively

offloading the CPU. Once hardware computation is over, Thread 1 wakes up, invalidates the content of the cache lines (if any) holding data of the buffer and puts the task in the the intermediate 1-element queue. After that, it repeats the process.

Thread 2 consumes a task from the intermediate queue and does the summation stage of ChaCha20 algorithm over the task's buffer content. The result goes back to the buffer and the task is placed in the queue of finished tasks. This process repeats again until the shutdown mode is activated.

Having two threads in FpgaChaWorker overlaps the summation stage with the hardware computations and thus increases the throughput.

### 5.2.4 Cryptor

Cryptor is a data consumer that consumes OTP blocks produced by workers to do data encryption. Cryptor also reschedules the tasks it processed, giving them new constantly increasing IDs.

The following types of cryptors are available:

- StubCryptor. This cryptor takes a task from the queue of finished tasks and immediately inserts it in the queue of scheduled tasks, assigning it a new ID. Even though this cryptor does not do any useful work with the OTP it consumes, it provides basically unlimited throughput and thus can be used to measure the performance of the workers.

- FileCryptor. This cryptor encrypts or decrypts a file using OTP blocks received from workers. In order to do this it opens two files: input and output. It maps these files onto the virtual memory of its process to improve file access performance by reducing the number of system calls and avoiding copying data from the kernel space to the user space. Then, it waits on the finished tasks queue for the next finished task. Upon getting one, it uses its buffer to XOR it word by word with the content of the input file and writes the result in the output file. The processed task is rescheduled after assigning it a new ID. Finally, it starts waiting for the next finished task and repeating the cycle until all words of the input file are processed.

### 5.2.5 Queue of finished tasks

The framework contains a special queue: a queue of finished tasks. Basically, this data structure is an array of single-element blocking queues. Workers pick a queue to insert a finished task into, based on its id according to the following formula:

$$\text{Queue index} = \text{task.id} \mod N \tag{5.1}$$

where N is the number of queues in the array. If the destination index contains a full queue, the worker stalls until the queue is empty.

The cryptor takes tasks from the queues in the round robin manner. If the next queue it needs to access is empty, the cryptor stalls until the queue is full.

By organizing the queues this way we allow the cryptor to process finished tasks in the ascending order of their IDs. This guarantees that all generated OTP blocks are used consequently even if two workers finish their tasks out of order.

### 5.2.6 Overall framework description

The framework functions as follows. Initially the queue of scheduled tasks contains $N$ tasks. with the following IDs: $0, 1, ..., N-1$. Workers consume those tasks and generate OTP blocks. The tasks with ready-to-use OTP blocks are placed in the queue of finished task to reorder them. The cryptor extracts finished tasks one by one, and performs file encryption/decryption. After using a task with OTP, the cryptor assigns it a new ID: $LastUsedId + 1$. Then, it puts it in the queue of scheduled tasks. This process repeats until the cryptor reaches the end of the input file. Upon reaching the end, the cryptor turns on the shutdown mode for all queues in the system, which leads to the termination of all threads waiting on the queues.

Figure 5.5 shows the framework, configured to have 3 slots in the queue of scheduled tasks and 6 slots in the queue of finished tasks. The total number of task objects circulating in the system is also 6 (Block 21, Block 22, Computing block 23, Computing block 20, Request for block 24, and Request for block 25). There are two workers, processing the tasks. The figure depicts the system at some arbitrary moment of time. At this moment the cryptor is waiting on Queue 2 for OTP block 20, which is being computed by Worker 1. Once Worker 1 finishes the computation, it will place the block in Queue 2, unlocking the cryptor. Queue 2 will be chosen for Block 20 because $20 \equiv 2 \mod 6$. After consuming and processing Block 20, cryptor, based

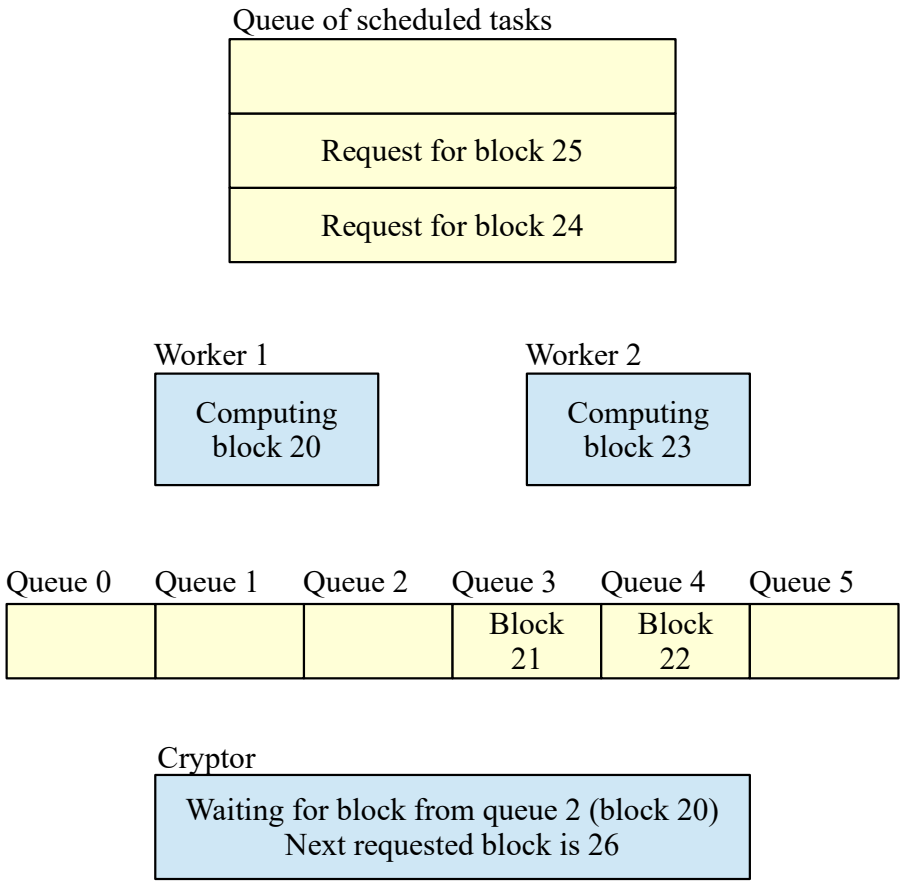



**Figure 5.5**: Example of the framework with 2 workers and a 6-slot queue

on its internal counter, will place a request for block 26 in the queue of scheduled tasks and increment its internal counter to 27. After that it will go to the next queue, Queue 3, and try to fetch OTP block 21, which is already there. The cryptor continues this process iteratively, wrapping around from Queue 5 to Queue 0. Once Workers 1 and Worker 2 finish their current computations, they will fetch the next requests for OTP blocks from the queue of scheduled tasks. If the cryptor is slower than the workers, they will eventually process all the tasks and will slow down by locking on the queue of scheduled tasks and waiting for the cryptor to schedule more requests.

In the opposite situation, when the cryptor is faster than the workers, the cryptor will lock on the queue of finished tasks, waiting while workers process more tasks. The capability of the faster part to slow down its counterpart, makes this system self-balancing.

# Chapter 6

# EXPERIMENTS AND RESULTS

This chapter presents the results of functional testing and performance measurements done for our software and hardware ChaCha20 implementations. The chapter also contains discussion of the results.

## 6.1 Correctness of software and hardware

In order to prove that both software and hardware ChaCha20 modes produce correct results, we carried out two experiments: functional testing of the single-threaded software mode (reference mode) and file encryption testing of the other modes to verify that they produce the same results as the reference mode.

### 6.1.1 Functional testing of the reference mode

We did functional testing of the reference mode to prove that it functions according to the standard. We used the test vector from [9] to generate the first 512 bits of OTP. This test vector is listed in Table 6.1. Based on the test vector our reference implementation produced the result shown in Table 6.2. This output exactly

**Table 6.1**: Test vector for reference mode functional testing

| Field name | Field content |
|---|---|
| Key | 0x03020100 0x07060504 0x0b0a0908 0x0f0e0d0c<br>0x13121110 0x17161514 0x1b1a1918 0x1f1e1d1c |
| Block count | 0x00000001 |
| Nonce | 0x09000000 0x4a000000 0x00000000 |

**Table 6.2**: The result of the reference ChaCha20 implementation operating on the test vector

| | | | |
|---|---|---|---|
| e4e7f110 | 15593bd1 | 1fdd0f50 | c47120a3 |
| c7f4d1c7 | 0368c033 | 9aaa2204 | 4e6cd4c3 |
| 466482d2 | 09aa9f07 | 05d7c214 | a2028bd9 |
| d19c12b5 | b94e16de | e883d0cb | 4e3c50a2 |

matches the exemplary OTP specified in [9], proving the correctness of the reference mode.

### 6.1.2 File encryption test of the other modes

To make sure that the other modes produce the same results as the reference one we did the file encryption test. We created a 256 MiB text file with random content using the following command:

```
base64 /dev/urandom | head -c 256M >./ramdisk/in
```

After that we generated a reference file by encrypting the random file using the reference mode. Next, we decrypted the reference file using the other ChaCha20 modes: a two-threaded software mode, and hardware modes with 1, 2, and 4 FpgaCha modules. We saved the results of decryption in separate files and computed SHA-2

**Table 6.3**: SHA-2 hashes of files decrypted using different modes

| Decryption mode | File size | SHA-2 hash (16 lower bytes) |
|---|---|---|
| Original file | 256 MiB | `84822395bbf0471da6fb0107bc3e5e89` |
| ChaCha20Worker ×2 | 256 MiB | `84822395bbf0471da6fb0107bc3e5e89` |
| FpgaChaWorker ×1 | 256 MiB | `84822395bbf0471da6fb0107bc3e5e89` |
| FpgaChaWorker ×2 | 256 MiB | `84822395bbf0471da6fb0107bc3e5e89` |
| FpgaChaWorker ×3 | 256 MiB | `84822395bbf0471da6fb0107bc3e5e89` |
| FpgaChaWorker ×4 | 256 MiB | `84822395bbf0471da6fb0107bc3e5e89` |

hashes of the decrypted files. Then we compared the hashes to the hash of the original random file. As a result, all hashes matched, allowing us to deem all our implementation to be correct. Even though we only tested decryption, this conclusion holds true for encryption as well, because there is no difference between encryption and decryption for stream cyphers. Table 6.3 lists SHA-2 hashes resulting from our experiments.

## 6.2 Throughput evaluation

In order to measure performance of our solution and identify bottlenecks, we carried out a series of experiments. Table 6.4 summarizes those experiments. The meaning of the content in its columns should be clear after reading Section 5.2. All software components were compiled with the `-Ofast` flag. The task length (see Section 5.2.1) in all experiments was 256 KiB. All FPGA modules were running at the clock rate of 50 MHz. The HPS was running at 800 MHz. The throughput evaluation was done by measuring the execution time of the program using Linux `time` utility.

**Table 6.4**: Experiments for throughput evaluation of software and hardware ChaCha20 implementations

| Name | Worker | Cryptor |
|---|---|---|
| A | StubWorker | FileCryptor |
| B1 | ChaCha20Worker ×1 | StubCryptor |
| B2 | ChaCha20Worker ×2 | StubCryptor |
| C | ChaCha20Worker ×2 | FileCryptor |
| D1 | FpgaChaWorker ×1 | StubCryptor |
| D2 | FpgaChaWorker ×2 | StubCryptor |
| D3 | FpgaChaWorker ×4 | StubCryptor |
| E1 | FpgaChaWorker ×1 | FileCryptor |
| E2 | FpgaChaWorker ×2 | FileCryptor |
| E3 | FpgaChaWorker ×4 | FileCryptor |

For experiments A, C, and Ex we allocated a `tmpfs` (a file system residing in RAM) with the size of 600 MiB, and created a file with pseudo random content with the size of 256 MiB. This file is called the input file. The input file is supplied to the compiled program for encryption. The result of the encryption is saved in the output file located in the same `tmpfs`. By using `tmpfs` in our experiments we eliminated speed limitations imposed by non-volatile storage.

The results of the experiments are summarized in Figure 6.1. The bars of the same color depict similar in nature experiments. The following subsections describe the experiments and present the analysis of their results.

### 6.2.1 File I/O throughput (Experiment A)

File I/O can be a limiting factor for the file encryption process. In order to understand where this limit is on the current platform, we conducted Experiment A.

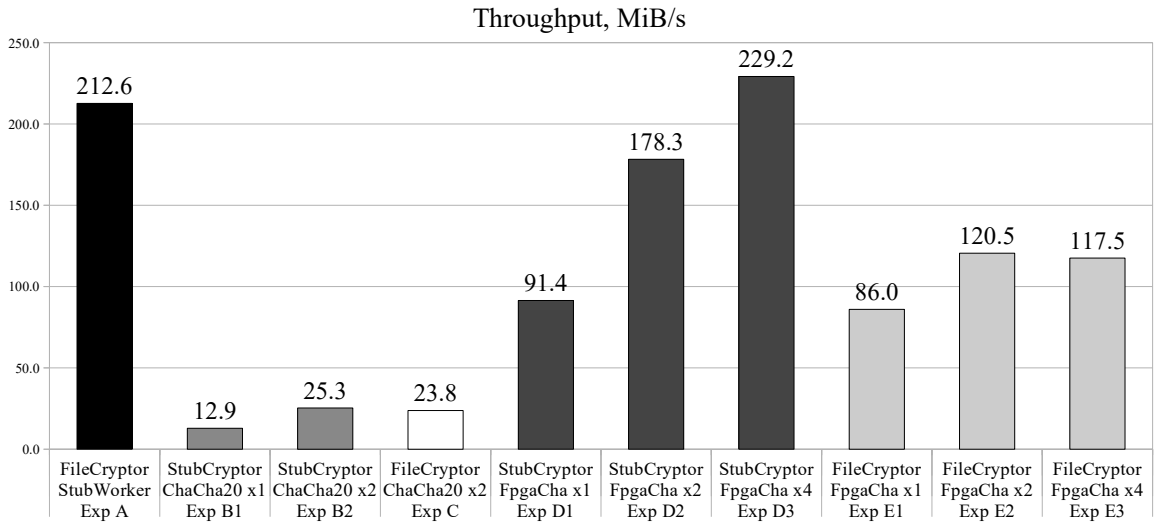



**Figure 6.1**: Overall system throughput for different combinations of data producers and data consumers

Conditions of this experiment (StubWorker as data supplier and FileCryptor as data consumer) simulate the situation when OTP is generated almost instantly, and major time is spent reading data from the input file, XORing it with OTP and writing it in the output file.

The experiment finished in 1.2 s. Considering the input file size of 256 MiB, this gives us the encryption throughput of 212.6 MiB/s.

### 6.2.2 Software ChaCha20 throughput (Experiments Bx)

In order to reveal the limitations of our software ChaCha20 cypher we conducted experiments B1 and B2. These experiments use 1 and 2 CPU cores to generate OTP, but the output data is not saved anywhere and rather discarded. Such conditions allow us to estimate pure performance of the software ChaCha20 without limitations imposed by file access.

Experiment B1 finished in 19.9 s and experiment B2 finished in 10.1 s. Considering the input file size of 256 MiB, this gives us the encryption throughputs of 12.9 MiB/s and 25.3 MiB/s. The increase in performance almost by two times when moving from one core to two cores suggests that ChaCha20 is perfectly parallelizable on 2 CPU cores. Additionally, the fact that the throughput in Experiment A exceeded the throughput in Experiment B2 by 8.54 times indicates that there is an opportunity for accelerating ChaCha20 by implementing it in hardware.

### 6.2.3 Software-based file encryption throughput (Experiment C)

Experiments A and Bx demonstrate throughput limits for the cases when only the data producer or the data consumer does the work. Those experiments allow us to understand the limits of different parts of a cryptographic application. However, the situations those experiments simulate are obviously useless in a real application. It is also not clear whether the real application can show the performance of its weakest part as a separate component or it will be even worse than that because of competition for shared resources. To understand the properties of a real file encryption process we conducted Eexperiment C, which features software ChaCha20 working on two CPUs as the data producer and FileCryptor as the data consumer.

Experiment C finished in 10.8 s. Considering the input file size of 256 MiB, this gives us the encryption throughputs of 23.8 MiB/s. This speed is only 11% of the maximum throughput possible on this platform (see Section 6.2.1), so such result indicates that software ChaCha20 can be accelerated by a hardware module.

Given only 7% performance degradation comparing to Experiment B2, we can conclude that the data producer and the data consumer almost do not compete for shared resources such as DRAM bandwidth and CPU time. This fact, as well as the data consumer being much faster than the data producer (see Section 6.2.2), suggests that only OTP computation is on the critical path, so XOR operation can stay in software without hurting performance.

### 6.2.4 Hardware ChaCha20 throughput (Experiments Dx)

In order to reveal the limitations of our hardware ChaCha20 cypher we conducted experiments D1, D2 and D3. These experiments use 1, 2, and 4 FpgaCha cores to generate OTP, but the output data is not saved anywhere and rather discarded. Such conditions allow us to estimate pure performance of the hardware ChaCha20 without limitations imposed by file access.

Experiment D1 finished in 2.8 s, experiment D2 in 1.4 s and experiment D3 in 1.1 s. Considering the input file size of 256 MiB, this gives us the encryption throughputs of 91.4 MiB/s, 178.3 MiB/s and 229.2 MiB/s. This data clearly demonstrates that performance of a single FpgaCha core by far exceeds the performance of the software solution which runs on two CPU cores. Additionally, this data shows that adding the seconds FpgaCha core improves the the performance by 2 times. However, adding two more cores (4 cores in total) does not significantly improve the situation. This observation can be ascribed to the limitations of the DRAM throughput because all cores work independently and DRAM is the only shared resource.

### 6.2.5 Hardware-based file encryption throughput (Experiments Ex)

Similarly to Experiment C we conducted experiments E1, E2 and E3 to understand how hardware ChaCha20 accelerator behaves in a realistic scenario. These experiments use 1, 2, and 4 FpgaCha cores to generate OTP, which is used to encrypt the input file.

Experiment E1 finished in 3.0 s, experiment E2 in 2.1 s and experiment E3 in 2.2 s. Considering the input file size of 256 MiB, this gives us the encryption throughputs of 86.0 MiB/s, 120.5 MiB/s and 117.5 MiB/s.

Experiment E1 clearly shows that adding real file encryption to a single FpgaCha core does not significantly hurt performance (it reduced only by 5.9% comparing to Experiment D1). This means that at this data rate, FpgaCha and CPU almost do not compete for shared resources. Additionally, E1 indicates that even a single instance of our hardware accelerator running at 50 MHz can beat the software solution running on 2 CPU cores with the clock rate of 800 MHz. The speedup of the hardware solution over the software one is 3.6 times.

Efficiency of the computation in Experiment E2 comparing to Experiment E1 is 0.7. This indicates that two FpgaCha cores in a realistic scenario are not very efficient. At the same time E2 demonstrates the speedup of 1.4 over Experiment E1, so two cores instead of one still can considerably increase the overall performance. Moreover, E2 shows the speedup of 5 times over the best software solution, which is quite a good result. However, this performance is still quite far from the platform's

limitation: the experiment showed only 56.6% of the maximum possible performance (see Experiment A in Section 6.2.1).

Efficiency of the computation in Experiment E3 comparing to Experiment E1 is 0.34. This indicates that four FpgaCha cores in a realistic scenario are not efficient at all. Moreover, E2 demonstrates the speedup of 1.37 over Experiment E1, which is less than in experiment E2. This means that it does not make sense to use 4 FpgaCha cores in a real situation on the current platform. This observation can be ascribed to the fact that the data producers and the data consumer compete for DRAM bandwidth.

None of the realistic Ex experiments managed to demonstrate the results that are close to ideal scenario of Experiment A. This means that neither FgaCha, nor file encryption process is the bottleneck, and the overall performance is limited by a shared resource: DRAM bandwidth.

# Chapter 7

# CONCLUSION

Efficient cryptographic algorithms are especially important nowadays because they can protect ever growing amounts of data at lower cost. New lightweight stream cyphers are being developed in order to satisfy this demand. Development of hardware accelerators for cryptographic algorithms is another way to make data protection more efficient. In this work we use both approaches together to accomplish better results: we picked a state-of-the-art stream cypher, ChaCha20, and improved its performance by making for it a hardware accelerator in FPGA.

## 7.1 What has been done

As our target platform we chose Cyclone V: a low-cost heterogeneous SoC with an FPGA and a CPU on a single die. Having both parts on the same chip reduces the cost associated with system integration. We designed a hardware ChaCha20 IP-core for this platform, consisting of two parts: the cryptographic accelerator itself and an auxiliary custom DMA that transfers data from the accelerator to DRAM. We justified the optimality of our design decisions. For example, we kept the XOR and the summation stages of ChaCha20 in software because this improves resource

utilization of our core without sacrificing its performance. As a result, we managed to develop an IP-core that requires only about 5% of the on-chip FPGA resources available on the given platform. Additionally, we developed software for a Linux-based OS to interface multiple instances of our core and employ them for file encryption. This framework also has the capability of employing a software ChaCha20 running on multiple CPU cores in order to compare the software and the hardware solutions. Finally, we tested our IP-core to prove that it is implemented correctly and carried out multiple experiments to asses its performance.

## 7.2 Results

Our tests include functional testing and performance measurements. Functional testing showed that our implementation is correct. Performance measurements demonstrated that in a realistic environment, a single hardware accelerator working at the frequency of only 50 MHz is 3.6 times faster than a software solution running on two 800 MHz CPU cores. Two hardware accelerators provide the speedup of 5 over the same software solution. The data we gathered shows that the throughput of file encryption cannot be higher than 212.6 MiB/s on the given platform. Software ChaCha20 can only reach 11% of this limit, whereas our hardware cores could go up to 56.6%. Attempts to increase the throughput by employing more than 2 accelerators did not give better results. Thus, we conclude that the maximum encryption rate on the given platform is 120.5 MiB/s. We ascribe this constraint to a limited bandwidth of the DRAM because the DRAM is the only shared resource that our IP-cores use.

## 7.3 Future work

The following efforts can constitute a continuation of this work.

First, an attempt to improve the performance of our IP-core can be made. This can be achieved by optimizing the critical path of the accelerator and thus increasing its clock frequency. In the case when two FpgaCha cores are used, a higher clock frequency may not help to improve the maximum encryption throughput because of DRAM bandwidth limitations on the current platform. However, a higher clock frequency can possibly improve the performance of a single IP-core, so it is still worth pursuing. Using high-end SoC chips such as Stratix V from Intel or Virtex UltraScale+ from Xilinx, is another way to improve performance. Not only can they provide higher clock frequency due to faster logic, but also they may have a faster DRAM controller and may be available on development boards with a faster DRAM.

Second, ChaCha20 alone has limited application: the algorithm only provides data confidentiality without data integrity, which is especially important for stream cyphers as they produce a malleable cyphertext. That is, a malicious party can modify the cyphertext and cause predictable changes in the plaintext. In order to address this issue, ChaCha20 needs to be accompanied by a cryptographic message authentication algorithm. RFC8439 [9] proposes Poly1305 for this purpose. The combination of these two algorithms is called ChaCha20-Poly1305. It is possible to use ChaCha20-Poly1305 as AEAD [19]. Thus, modifying our IP-core to enable support for Poly1305 can widen its application scope.

Finally, comparison of hardware ChaCha20 to hardware AES may be of interest for people who are considering moving to faster cryptographic solutions. AES has long been a de facto industry standard, so hardware manufacturers embed AES accelerators in their products. At the same time, to the best of our knowledge, no ChaCha20 accelerators are available in mass commercial products. Since ChaCha20 involves only simple operations as opposed to AES, it is reasonable to assume that a hardware ChaCha20 can outperform a hardware AES. If this assumption comes out to be true, it can create an incentive for hardware manufacturers to consider ChaCha20 as an alternative for AES.

# APPENDICES

# Appendix A

## ENABLING FPGA-TO-SDRAM BRIDGE USING U-BOOT

The FPGA-to-SDRAM bridge in Cyclone V is useful for accessing the DRAM from the FPGA side (see Section 2.3). Unfortunately, this bridge is disabled by default. In order to enable it, one needs to modify certain registers inside HPS. In general, HPS control registers can be accessed from Linux OS. However, the registers responsible for the FPGA-to-SDRAM bridge have a special requirement: they cannot be modified after the SDRAM controller is started being used. Since Linux relies on storing data in DRAM, enabling the bridge after the OS boots will violate this requirement.

Fortunately, this problem can be solved using U-Boot, a boot loader that is often used to start Linux on the ARM platform. U-Boot does not require access to the DRAM controller, so using it for enabling the bridge is acceptable. The command interface of U-Boot has `mw` command (memory write). The command accepts two arguments: hexadecimal address of the memory location that needs to be accessed and a value that will be placed there [26]:

```
mw address value
```

Table A.1 contains information about the registers that need to be modified in order to enable FPGA-to-SDRAM bridge in Cyclone V [27]. When configuring

**Table A.1**: Register map of some registers of HPS

| Name | Address | Bits | Description |
|---|---|---|---|
| `sdr.fpgaportrst.portrstn` | `FFC25080` | 13:0 | This register should be written to with a 1 to enable the selected FPGA port to exit reset. Writing a bit to a zero will stretch the port reset until the register is written. Read data ports are connected to bits 3:0, with read data port 0 at bit 0 to read data port 3 at bit 3. Write data ports 0 to 3 are mapped to 4 to 7, with write data port 0 connected to bit 4 to write data port 3 at bit 7. Command ports are connected to bits 8 to 13, with command port 0 at bit 8 to command port 5 at bit 13. |
| `sdr.staticcfg.applycfg` | `FFC2505C` | 3 | Write with this bit set to apply all the settings loaded in SDR registers to the memory interface. This bit is write-only and always returns 0 if read. |

the `sdr.fpgaportrst` register we set bits from 4 to 7 to enable the write ports (our IP-core only needs to write data to DRAM). If the read or command ports are needed the corresponding bits should be set as well. When configuring the `sdr.staticcfg` register, we set bit 3 in order to make the configuration of the previous register effective. Since `mw` does not support bit masking, it is impossible to keep the values of the other bits in the registers, so we read the old values of both registers using `mr` command, modified necessary bits in the velues we got and wrote the new values using `mw`. The final `mw` commands to enable the the bridge look as follows:

```
mw 0xFFC25080 0x1F0

mw 0xFFC2505C 0xA
```

# REFERENCES


[1] (Jun. 15, 2019). "STM32 Advanced Encryption Standard hardware accelerator," STMicroelectronic, [Online]. Available: http://www.st.com/resource/en/product_training/stm32l4_security_aes.pdf (visited on 06/15/2020).

[2] (Mar. 2018). "MSP430x5xxand MSP430x6xx Family User's Guide," Texas Instruments, [Online]. Available: https://www.ti.com/lit/ug/slau208q/slau208q.pdf?ts=1592364725411 (visited on 06/16/2020).

[3] D. Bernstein, "ChaCha, a variant of Salsa20," Jan. 2008. [Online]. Available: https://cr.yp.to/chacha/chacha-20080120.pdf (visited on 06/19/2020).

[4] A. Langley. (Feb. 27, 2014). "TLS Symmetric Crypto," ImperialViolet, [Online]. Available: https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html (visited on 06/24/2020).

[5] S. Heron, "Advanced Encryption Standard (AES)," *Network Security*, vol. 2009, no. 12, pp. 8–12, 2009, ISSN: 1353-4858. DOI: https://doi.org/10.1016/S1353-4858(10)70006-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1353485810700064.

[6] C. Paar, *Understanding cryptography: a textbook for students and practitioners*. Berlin London: Springer, 2009, ISBN: 978-3-642-04100-6.

[7] D. J. Bernstein, *The Salsa20 Family of Stream Ciphers*, M. Robshaw and O. Billet, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 84–97, ISBN: 978-3-540-68351-3. DOI: 10.1007/978-3-540-68351-3_8.

[8] Y. Nir and A. Langley, "RFC7539: ChaCha20 and Poly1305 for IETF Protocols," May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7539 (visited on 06/25/2020).

[9] ——, "RFC8439: ChaCha20 and Poly1305 for IETF Protocols," Jun. 2018. [Online]. Available: https://tools.ietf.org/html/rfc8439 (visited on 06/25/2020).

[10] A. Milenković, *CPE523: Hardware/Software Co-Design*, University Lecture, The University of Alabama in Huntsville, 2019.

[11] (Jun. 18, 2020). "Terasic — SoC Platform — Cyclone — DE10-Nano Kit," Terasic, [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1046 (visited on 06/18/2020).

[12] (May 7, 2018). "Cyclone V Device Overview," Intel, [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf (visited on 06/15/2020).

[13] (May 7, 2020). "Avalon Interface Specifications," Intel, [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf (visited on 05/15/2020).

[14] H.-J. Koch. (Dec. 2006). "The Userspace I/O HOWTO," The Linux Kernel documentation, [Online]. Available: https://www.kernel.org/doc/html/v4.13/driver-api/uio-howto.html (visited on 05/15/2020).

[15] K. Ichiro. (2020). "User space mappable dma buffer device driver for Linux," GitHub, [Online]. Available: https://github.com/ikwzm/udmabuf (visited on 05/15/2020).

[16] R. Cowart, D. Coe, J. Kulick, and A. Milenković, "An Implementation and Experimental Evaluation of Hardware Accelerated Ciphers in All-Programmable SoCs," in *Proceedings of the SouthEast Conference*, ser. ACM SE '17, Kennesaw, GA, USA: Association for Computing Machinery, 2017, pp. 34–41. DOI: 10.1145/3077286.3077297. [Online]. Available: http://www.ece.uah.edu/~milenka/docs/rc_acmse17.pdf.

[17] S. Baskaran and P. Rajalakshmi, "Hardware-Software Co-Design of AES on FPGA," in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ser. ICACCI '12, Chennai, India: Association for Computing Machinery, 2012, pp. 1118–1122, ISBN: 9781450311960. DOI: 10.1145/2345396.2345575.

[18] (Nov. 5, 2018). "Silex Insight unveils high-throughput version of Chacha20-Poly1305 authenticated encryption," Silex Insight, [Online]. Available: https://www.silexinsight.com/silex-insight-unveils-high-throughput-version-chacha20-poly1305-authenticated-encryption (visited on 05/18/2020).

[19] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm," *Journal of Cryptology*, vol. 21, pp. 469–491, 4 2008. DOI: 10.1007/s00145-008-9026-x. [Online]. Available: https://cseweb.ucsd.edu/~mihir/papers/oem.pdf.

[20] (Dec. 10, 2019). "ChaCha20-Poly1305 Crypto Engine: Device Implementation Matrix," Xilinx, [Online]. Available: https://www.xilinx.com/products/intellectual-property/1-onj5rx.html#productspecs (visited on 05/18/2020).

[21] G. Kanda and K. Ryoo, "High-Throughput Low-Area Hardware Design of Authenticated Encryption with Associated Data Cryptosystem that Uses ChaCha20 and Poly1305," *International Journal of Recent Technology and Engineering (IJRTE)*, vol. 8, pp. 86–94, 2S6 2019, ISSN: 2277-3878. DOI: 10.35940/ijrte.B1017.0782S619. [Online]. Available: https://www.ijrte.org/wp-content/uploads/papers/v8i2S6/B10170782S619.pdf.

[22] N. At, J. Beuchat, E. Okamoto, İ. San, and T. Yamazaki, "Compact Hardware Implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 2, pp. 485–498, 2014.

[23] J. Strömbergson. (2016). "Verilog 2001 implementation of the ChaCha stream cipher," GitHub, [Online]. Available: https://github.com/secworks/chacha (visited on 05/15/2020).

[24] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999, pp. 222–233.

[25] I. Semenov. (2020). "FpgaCha — ChaCha20 implementation in FPGA," GitHub, [Online]. Available: `https://github.com/Goshik92/FpgaCha` (visited on 05/15/2020).

[26] (Jun. 15, 2019). "U-Boot documentation: memory commands," Denx software engineering, [Online]. Available: `https://www.denx.de/wiki/publish/DULG/to-delete/UBootCmdGroupMemory.html` (visited on 06/15/2020).

[27] (Jun. 15, 2020). "Cyclone V HPS Memory Map," Intel, [Online]. Available: `https://www.intel.com/content/www/us/en/programmable/hps/cyclone-v/hps.html` (visited on 06/15/2020).