Linköping University | Department of Electrical Engineering
Master's thesis, 30 ECTS | Datateknik
2025 | LiTH-ISY-EX--25/5733--SE

Exploring the possibility of cracking Argon2 hashes with FPGA technology

En undersökning om hur FPGA kan användas för att effektivisera Argon2

Gustav Eriksson
Oskar Magnusson

Supervisor : Olle Hansson
Examiner : Kent Palmkvist

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

As data security is continuously becoming more important to society, the interest in secure cryptographic hash functions is on the rise, as they play a vital role in the field. The winner of the Password Hashing Competition in 2015 was Argon2, which OWASP recommends as the standard method of password storage. Specifically, it tries to target the performance of GPU-reliant cracking devices by requiring a large amount of memory. Despite this, there is some potential for other pieces of hardware to take on this algorithm, such as ASICs and FPGAs.

In this project, an FPGA implementation of Argon2 written in VHDL is presented. The system was designed with the purpose of being used in conjunction with the Hashcat advanced password guessing tool. The Spartan 6 FPGA proved to be too small in terms of area for the proposed design. In order to ensure a programmable solution, the design was repurposed for the Virtex 6 FPGA. Based on simulation data, performance was estimated and showed a clear disadvantage compared to the reference CPU. Some interesting propositions for further investigation of the topic are presented at the end of the report.

Acknowledgments

We would like to thank our supervisor Olle Hansson and examiner Kent Palmkvist for their valuable feedback and support during this thesis. We would also like to thank the Swedish National Forensic Centre for suggesting this interesting topic. Special thanks to our external supervisors at NFC for their continuous support and guidance throughout the project.

Contents

List of Figures

List of Tables

1 Introduction

Computer passwords have been around since the 1960s [1], but remain the most common way to protect user data to this day. A very important concept in the field of password management is cryptographic hash functions [28]. They are used to hash passwords before storing them, rendering them completely unrecognizable. As this process is practically impossible to reverse, the password is kept a secret to everyone except the user and protects it from being obtained by a malicious actor even in the event of a data leak. Because of this, a commonly used cryptographic hash function discovered to contain vulnerabilities could constitute millions of passwords becoming cracked. An example of this is the LinkedIn leak in 2012 where millions of passwords were leaked in clear text after being stored as unsalted SHA1 hashes, a hash function considered to be very weak. It is therefore crucial to choose the right hashing function in order to keep passwords safe.

As the world's available computational power increases and running a hashing algorithm takes less and less time, hashed passwords become more vulnerable to being recovered. With the emergence of GPUs and their increased parallel execution ability, new algorithms need to account for this new changing landscape. During the Password Hashing Competition (PHC) in 2015, *Argon2* was crowned the winner. Classified as a memory-hard algorithm, Argon2 hinders the relative efficiency of GPUs by exploiting their weakness of having little Random Access Memory (RAM) capacity compared to CPUs. It is also very versatile; with several customizable parameters, Argon2 provides the possibility for users to adjust it to meet performance- or resource-specific criteria. It comes in three different versions: *Argon2d* (data-dependent memory access), *Argon2i* (data-independent memory access) and Argon2id, which is a mix of both. Currently, the password-storing hash algorithm recommended according to the Open Web Application Security Project (OWASP) is Argon2id [28]. It is also used as the key derivation function in the Linux Unified Key Setup (LUKS2) [24].

There are however, other pieces of hardware that could pose a threat to Argon2, namely *Field Programmable Gate Arrays* (FPGA). As designs are implemented directly as hardware circuitry in these types of boards they too can perform parallelized computations, and thus be quite efficient in doing tasks where parallelism or pipelining options are available. With a larger RAM capacity than GPUs, it could potentially outperform them when computing Argon2 hashes.

1.1 Motivation

The thesis subject was proposed by the *Swedish National Forensic Centre* (NFC), and they acted as an external supervisor for the project. The idea is to investigate the potential of using FPGAs for computing Argon2 hashes, and possibly produce some estimates of specifications necessary for it to pay off. It should be compatible with an efficient password recovery tool such as *Hashcat*. However, since it does not currently support FPGA crackers, a solution for that would also be investigated.

1.2 Aim

The purpose of this thesis project is to explore the possibility and performance of using FPGA support to crack Argon2 hashes in conjunction with password recovery tools such as Hashcat. As no official FPGA implementation of Argon2 could be found, it would have to be implemented within the scope of the project.

1.3 Research Questions

Two research questions have been formulated. These questions are meant to provide enough insight to determine if FPGAs are suitable for cracking Argon2 hashes.

1. How much area would be required for a basic implementation of Argon2 on a Spartan 6 FPGA?
2. What performance could be expected of a basic implementation of Argon2 on a Spartan 6 FPGA?

1.4 Delimitations

This thesis project has only access to a specific FPGA board, the Spartan6 - SP605 - XC6SLX45T. This board will therefore be the one connected to interface with Hashcat. Due to the risk that the proposed design will not be synthesizable due to size, some Argon2 parameters' size constraints will be hard-coded and some parameters will be hard-coded outright. If the proposed design of the hashing algorithm does not fit and/or cannot be routed on this FPGA even after the previously mentioned steps, a bigger FPGA will be chosen for evalutation. As the larger FPGA cannot be physically accessed, the performance will only be estimated and measured in simulations. Due to time limitations, only Argon2d will be implemented as it was evaluated to be the least time-consuming alternative.

1.5 Method Overview

The project is comprised of four distinct phases:

1. **Implement Argon2 on a FPGA.** As of today, no official implementation of Argon2 exists for FPGA. An unfinished VHDL implementation developed by Christian Krieg will be used as a starting point [3].
2. **Implement a Hashcat core that is designed for FPGA cracking.**
3. **Establish a communication link between the Host and the FPGA.** For simplicity reasons, the UART protocol is chosen for this task.
4. **Evaluation.** The size and performance of the implementation will be evaluated. The goal is to have a working prototype to run, but simulations and approximations will also be used to estimate performance.

2 Related Work

Although no complete FPGA-based implementation of Argon2 could be found, other algorithms relevant to this project have been implemented. In a paper written by Percival [35] he introduced a set of definitions for memory-hardness, as well as the memory-hard algorithm *scrypt*.

This algorithm first generates a number of blocks equal to a parallelization factor using the *Key Derivation Function* (KDF) $PBKDF2$ with $HMAC\_SHA256$ as its pseudo-random function. The blocks are then passed through the *ROMix* function which will modify each block through a series of operations that depend on previous operations on the same block, potentially requiring a large amount of memory. These blocks are then concatenated into a salt that is passed through $PBKDF2_{HMAC-SHA256}$ with the same *message* as before.

An implementation of scrypt on a Virtex 7 FPGA for the purpose of cryptocurrency mining has been proposed by Duong Le et. al [25]. This hardware solution splits up the *ROMix* algorithm into pipelined Processing Elements (PEs). There were two different synthesized versions of this implementation, one with 32 PEs and one with only one PE. The parameters were those used in most mining environments, which translates to a memory usage of 128 KiB. Comparing the FPGA implementation with the highest hash rate CPU and GPU resulted in the extrapolated table below [25].

Device	kHash/s	Hash/J	LUTs	FFs	MHz	No. cycles per hash
Virtex 7 (1 PE)	2.74	4986	9389	10535	156.05	57012
Virtex 7 (32 PE)	89.38	12721	159434	156934	156.05	1746
Intel I9-10940X	1.8	15	n/a	n/a	n/a	n/a
Nvidia RTX 3090	81.3	246.6	n/a	n/a	n/a	n/a

Table 2.1: Scrypt comparisons.

Here the FPGA outperforms the CPU and GPU in energy efficiency and the number of hashes per second. The scrypt parameters resulted in a quite small amount of memory used by the KDF in comparison to the recommended parameters for Argon2, presented further in Section 3.5.5.

Another interesting paper is about a study conducted by Atiwa et. al. [4] which investigates how the cryptographic hash function *BLAKE2b* can be accelerated with an FPGA. It is

designed to be used with the memory-hard algorithm *Equihash* [4]. In the paper, Atiwa et. al. focuses on the usage of hash functions in the blockchain. In general, SHA-2 is the most known hash function used in this context. However, as a high-speed ASIC implementation of it has become available and the integrity of its predecessor SHA-1 has been compromised, the adoption of it in blockchain solutions has become questionable. Next-generation blockchain networks have in fact already started to adopt new Proof-of-Work (PoW) methods in order to protect against ASIC acceleration. The memory-hard function Equihash, built on BLAKE2b, is one of them [4].

The paper proposes a design of BLAKE2b implemented on the *Zynq 7000 ZC706* and *Virtex 7 VC707* boards. For the former board, the implementation utilized 9517 LUT:s (4.35% of the total amount) and 7350 FF:s (1.68%). The Virtex 7 board used 63549 LUTs (20%) and 84869 FF:s (13%). This design consists of BLAKE2b and an IP core that enables communication with a CPU through PCI Express. It was also designed using High-level synthesis (HLS), instead of e.g. VHDL or Verilog. As the author points out, this could negatively affect the area, clock speed and number of clock cycles needed as a trade-off for a faster design time. The maximum achievable clock frequency wasn't stated in the paper but the plots show hash rate comparisons of up to 100 MHz with the BLAKE2b implementation taking 33 clock cycles. The metric used for performance is hashes per second and calculated by dividing the clock speed in Hz by the number of clock cycles needed. In the final graph, the two FPGA implementations are compared with an unspecified Intel Xeon processor where the former vastly outperforms the latter. Specifically where the clock frequency reaches 100 MHz, the CPU sits at what looks like 0.4 Mh/sec, compared to the FPGA implementations which reach approximately 2.5 - 3.1 Mh/sec.

Although a comparison is hard to make between the Spartan 6 and the Virtex 7 used in the paper by Atiwa et. al, it is at least feasible that the design could outperform a CPU. This paper is however, a bit unclear with its achievable clock frequency of the design and its CPU comparison where the CPU performance stays unchanged with an increase in clock frequency.

Even the least parallelized Virtex 7 implementation of scrypt outperformed a modern CPU by 65%, albeit with a small amount of memory. The physical space occupied by scrypt and BLAKE2b is quite low in terms of LUTs and FFs. In the latter case, the numbers are inflated due to the usage of HLS; a pure VHDL implementation can therefore be expected to be smaller. This indicates that there could be some potential for an Argon2 FPGA implementation that would be able to outperform a CPU in terms of hashes per second.

3 Theory

This chapter provides a theoretical background to the report. Some important concepts are introduced, such as cryptographic hash functions, algorithm complexity, and memory-hard algorithms. The hardware and software used are also presented here.

3.1 Computational Complexity

Computational complexity, often just complexity, is a term describing the necessary resources required by an algorithm to run it. There are two distinct but highly overlapping fields in computational complexity, one being *computational complexity theory*, which focuses on the complexity of mathematical problems. The other, *algorithm analysis*, focuses on the complexity of explicitly given algorithms. In this thesis, the latter is of greater interest [10].

There are several resource types that affect the complexity of an algorithm, such as bit complexity, time, space, and even communication in the case of distributed algorithms. While time is perhaps the most commonly considered resource, space can be of great importance too as apparent by the need for memory-hard cryptographic hash functions [10].

3.1.1 Memory-hard Algorithms and Functions

While the field of algorithm analysis can be used in order to further enhance and optimize algorithms, it can also used in a reversed situation where a slower-performing algorithm is actually preferable; password hashing and key derivation are two common examples of this. As mentioned in the recommendations for password Storage by OWASP [27], passwords should always be stored as hashes in order to mitigate the fallout in case of a data leak. As an attacker gets a hold of the password hashes, the only feasible way to obtain the passwords in plaintext (assumed that they were hashed with a cryptographically secure hashing function) is to try and guess those passwords with different guessing techniques. The faster the attacker can try different guesses the chances of them recovering the passwords increases. Therefore it is desirable to not only use a hashing function that is cryptographically secure but also one that takes a certain time to compute. As the computation time increases it lessens the attacker's ability to recover the passwords, but it also affects the legitimate user's experience when entering the password. This introduces a trade-off situation, where the security of a service must be tuned relative to the user experience.

One simple way of increasing the computational cost of an algorithm is to increase the number of required operations. Even though this increases the computing time on a per-hash basis, the asymptotic cost of hashing n passwords can be significantly reduced by using Graphic Processing Units (GPUs) through their parallel computational capability. Lower asymptotic cost entails higher throughput. One exploitable limitation of these types of hardware is their limited access to RAM in comparison to a central processing unit (CPU). Introduced as concepts by Percival [35], the purpose of memory-hardness is to negatively affect the potential asymptotic cost reduction that parallelized hardware provides by increasing the memory required. He introduces the following definition.

Definition 1. A memory-hard algorithm on a Random Access Machine is an algorithm that uses $S(n)$ space and $T(n)$ operations, where $S(n) \in \Omega(T(n)^{(1-\epsilon)})$.

In other words, a memory-hard algorithm can be described as one that asymptotically requires almost as many memory locations as operations, or as an algorithm that maximizes the memory requirements for a certain amount of operations. An example of a memory-hard algorithm that follows this definition is *HEKS*, introduced by Reinhold in 1999 [18, 35]. However, this is not enough to prevent an attacker from gaining a very significant advantage with parallelized hardware, as can be shown by Percival when comparing the dollar-seconds cost to a naive sequential implementation. Therefore, a second definition is introduced [35].

Definition 2. A sequential memory-hard function is a function which

(a) can be computed by a memory-hard algorithm on a Random Access Machine in $T(n)$ operations; and

(b) cannot be computed on a Parallel Random Access Machine with $S'(n)$ processors and $S'(n)$ space in expected time $T'(n)$ where $S'(n)T'(n) = \mathrm{O}(T(n)^{2-x})$ for any $x > 0$.

A sequential memory-hard function can in other words not only be described as a function where the fast algorithm is memory-hard, but also where it is impossible for a parallel algorithm to asymptotically reach a significantly lower cost [35].

In the report *Memory hard functions and persistent memory hardness* by Kornerup [19], another set of definitions is presented in addition to Percival's. The purpose of these definitions is to add the requirement that a well-designed memory-hard function should require a large amount of memory in a significant number of steps. This prevents functions that have a spike in memory utilization from being classified as memory-hard.

3.2 Cryptographic Hash Function

A hash function is a mathematical function that maps arbitrary input data to output data of a fixed length, usually called a *hash*, or in some cases, a *digest*. In order for a hash function h to be considered a strong cryptographic tool, it must possess the following properties [11].

- **Pre-image resistance**: Given output y, it should be very difficult to calculate an input x such that $h(x) = y$.
- **Second Pre-image resistance**: Given input x and its corresponding hash y, it should be very difficult to find another input z such that $h(z) = h(x) = y$.
- **Collision resistance**: It should be very difficult to find two arbitrary inputs x and z such that $h(x) = h(z) = y$. Note that the collision resistance property implies second pre-image resistance.

These functions play a significant role in computer security in the context of password storage. In order to protect the integrity of the user base, even in the case of a data leak, it is essential that they have been stored in such a way that the attacker is prevented from obtaining them. By storing the hash of a password rather than the password itself, a system can verify a password by hashing it and then comparing the hashes. A randomly generated sequence of data, called *salt*, is often added to the password before hashing it. The resulting hash is then stored together with the salt. This is done in order to ensure that two users with the same password do not produce the same hash, which effectively renders attacks using look-up tables (rainbow attacks) useless. These types of attacks involve creating tables that contain a mapping between all common passwords and their hashes for a specific hash function. Then if a data leak does happen, a large portion of the leaked password hashes could be cracked by simply doing look-ups in this table [28].

3.3 Key Derivation Function

A *Key Derivation Function* (KDF) is another type of function very similar to a cryptographic hash function, albeit with a slightly different goal. They similarly utilize a pseudo-random function to produce scrambled data in order to achieve one of their two primary goals. Typically, KDFs are designed to target either one of these [23].

- **Cryptographic key derivation**: Given an input x, it derives a cryptographic key suitable to be used as input to an encryption function. PBKDF2 for example is a password-based key derivation function suited for deriving encryption keys [31].
- **Password storage**: Uses a computationally expensive function to produce a hash of the password instead of storing it in plaintext. Scrypt for example is a KDF designed for this task [35].

KDFs, in contrast to hash functions, are by design favored to be computationally expensive, hence why they sometimes perform a certain amount of iterations in order to slow the execution down. This is known as key stretching and is a strategy used to protect against attackers by limiting the number of guesses possible to perform in a given timeframe. While different from hash functions, they are built upon cryptographic hash functions internally as part of the data scrambling process. As will be explained later in Section 3.5, this is the case with Argon2 which heavily depends on the cryptographic hash function *BLAKE2b* [22].

3.4 BLAKE2b

BLAKE2 is a cryptographic hash function built upon BLAKE, a finalist for the SHA-3 hashing competition. Although not the winner of said competition, in terms of speed it outperforms other algorithms like MD5, SHA-1, SHA-2, and even the winning candidate *Keccak*, now known as SHA-3. BLAKE2 is a further improved version that consumes less RAM and gives the user more customization options, among other things [7].

Furthermore, there are two flavors of BLAKE2: BLAKE2s is optimized for 8- to 32-bit platforms and BLAKE2b for 64-bit platforms. They can produce outputs of 1 to 32 bytes and 1 to 64 bytes respectively, and are both believed to be highly secure. BLAKE2b is used in the Argon2 password hashing function, where it is defined as the fixed length hashing function. How it is used in Argon2 is described further in Section 3.5. The input parameters of BLAKE2b and their respective ranges and can be seen in Table 3.1 [34].

Value	Parameter Name & Range (bytes)
Bits in word	w = 64
Rounds	r = 12
Block size (bytes)	bb = 128
Hash size (bytes)	$0 \leqslant nn \leqslant 64$
Key size (bytes)	$0 \leqslant kk \leqslant 64$
Input size (bytes)	$0 \leqslant ll < 2^{128}$

Table 3.1: Input parameters for BLAKE2b.

BLAKE2b uses an initialization vector IV in F_{BLAKE2} as well as in the main algorithm. Let p_i be the i:th prime number, and $frac(x)$ is the fractional part of x, then IV is mathematically defined as

$$IV[i] = \lfloor 2^w \cdot frac(\sqrt{p_{i+1}}) \rfloor.$$

A constant defined in BLAKE2b (and BLAKEs) is $SIGMA$. It contains the round permutation scheme for the internal permutation function. It is utilized in F_{BLAKE2}.

$$SIGMA = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 14 & 10 & 4 & 8 & 9 & 15 & 13 & 6 & 1 & 12 & 0 & 2 & 11 & 7 & 5 & 3 \\ 11 & 8 & 12 & 0 & 5 & 2 & 15 & 13 & 10 & 14 & 3 & 6 & 7 & 1 & 9 & 4 \\ 7 & 9 & 3 & 1 & 13 & 12 & 11 & 14 & 2 & 6 & 5 & 10 & 4 & 0 & 15 & 8 \\ 9 & 0 & 5 & 7 & 2 & 4 & 10 & 15 & 14 & 1 & 11 & 12 & 6 & 8 & 3 & 13 \\ 2 & 12 & 6 & 10 & 0 & 11 & 8 & 3 & 4 & 13 & 7 & 5 & 15 & 14 & 1 & 9 \\ 12 & 5 & 1 & 15 & 14 & 13 & 4 & 10 & 0 & 7 & 6 & 3 & 9 & 2 & 8 & 11 \\ 13 & 11 & 7 & 14 & 12 & 1 & 3 & 9 & 5 & 0 & 15 & 4 & 8 & 6 & 2 & 10 \\ 6 & 15 & 14 & 9 & 11 & 3 & 0 & 8 & 12 & 2 & 13 & 7 & 1 & 4 & 10 & 5 \\ 10 & 2 & 8 & 4 & 7 & 6 & 1 & 5 & 15 & 11 & 9 & 14 & 3 & 12 & 13 & 0 \end{bmatrix}$$

BLAKE2 uses two internal functions, one of which is the mixing function which will be called G_{BLAKE2} here. It takes as input a 16-word message block vector v and mixes two input words, x, and y, into four words in v indexed by integer arguments a, b, c, and d. Some pseudo-code for this can be found in Algorithm 1 [34].

The second internal function, the compression function, will be called F_{BLAKE2} here. It takes as arguments an 8-word state vector h, a 16-word message block vector m, a 2-word offset counter t, and a final block indicator flag f. Some pseudo-code for this can be found in Algorithm 2. Lastly, some pseudo-code for the BLAKE2b function in its entirety can be found in Algorithm 3 [34].

Algorithm 1 $G_{BLAKE2b}(v[0..15], a, b, c, d, x, y)$

$v[a] \leftarrow (v[a] + v[b] + x) \mod 2^w$
$v[d] \leftarrow (v[d] \oplus v[a]) \ggg 32$
$v[c] \leftarrow (v[c] + v[d]) \mod 2^w$
$v[b] \leftarrow (v[b] \oplus v[c]) \ggg 24$
$v[a] \leftarrow (v[a] + v[b] + y) \mod 2^w$
$v[d] \leftarrow (v[d] \oplus v[a]) \ggg 16$
$v[c] \leftarrow (v[c] + v[d]) \mod 2^w$
$v[b] \leftarrow (v[b] \oplus v[c]) \ggg 63$
Return $v[0..15]$

Algorithm 2 $F_{BLAKE2b}(h[0..7], m[0..15], t, f)$

$v[0..7] \leftarrow h[0..7]$
$v[8..15] \leftarrow IV[8..15]$
$v[12] \leftarrow v[12] \oplus (t \mod 2^w)$
$v[13] \leftarrow v_[13] \oplus (t \gg w)$
if f = TRUE **then**
    $v[14] \leftarrow v[14] \oplus 0xFF..FF$
end if
for $i \leftarrow 0; i < r; i \leftarrow i + 1$ **do**
    $s[0..15] \leftarrow SIGMA[i \mod 10][0..15]$
    $v \leftarrow G_{BLAKE2}(v, 0, 4, 8, 12, m[s[0]], m[s[1]])$
    $v \leftarrow G_{BLAKE2}(v, 1, 5, 9, 13, m[s[2]], m[s[3]])$
    $v \leftarrow G_{BLAKE2}(v, 2, 6, 10, 14, m[s[4]], m[s[5]])$
    $v \leftarrow G_{BLAKE2}(v, 3, 7, 11, 15, m[s[6]], m[s[7]])$
    $v \leftarrow G_{BLAKE2}(v, 0, 5, 10, 15, m[s[8]], m[s[9]])$
    $v \leftarrow G_{BLAKE2}(v, 1, 6, 11, 12, m[s[9]], m[s[11]])$
    $v \leftarrow G_{BLAKE2}(v, 2, 7, 8, 13, m[s[10]], m[s[13]])$
    $v \leftarrow G_{BLAKE2}(v, 3, 4, 9, 14, m[s[11]], m[s[15]])$
end for
for $i \leftarrow 0; i < 8; i \leftarrow i + 1$ **do**
    $h_i \leftarrow h_i \oplus v_i \oplus v_{i+8}$
end for
return $h[0..7]$

Algorithm 3 BLAKE2b (dd, ll, kk, nn)

$h[0..7] \leftarrow IV[0..7]$
$h_0 \leftarrow h_0 \oplus 0x01010000 \oplus (kk \ll 8) \oplus nn$
if $dd > 1$ **then**
    for $i \leftarrow 0; i < dd - 1; i \leftarrow i + 1$ **do**
        $h \leftarrow F_{BLAKE2b}(h, d_i, (i + 1) \cdot bb, FALSE)$
    end for
end if
if $kk = 0$ **then**
    $h \leftarrow F_{BLAKE2b}(h, d_{dd-1}, ll, TRUE)$
else
    $h \leftarrow F_{BLAKE2b}(h, d_{dd-1}, ll + bb, TRUE)$
end if
return $h_{[0..nn]}$

3.5 Argon2

Argon2 is a memory-hard key derivation function and the winner of the Password Hashing Competition in 2015. It is the recommended standard for password storage by OWASP [28]. It is very versatile thanks to its several customizable parameters, which can be tailored to suit specific infrastructures and security demands. Furthermore, it is available in three different versions [5].

- **Argon2d** is the faster version which uses data-dependent memory access which denies the possibility of trade-off attacks. It is however vulnerable to timing attacks and is therefore suited for crypto-currency and backend-server implementations.
- **Argon2i** is the slower version which uses data-independent memory access. This is better suited for client-side password hashing since it is resistant to side-channel attacks.
- **Argon2id** is a combination of the two as it uses data-independent indexing for the first pass and then switches to data-dependent indexing for the rest of the passes.

Argon2 specifies a number of integer input parameters that are classified as either primary (message and salt) or secondary. They can be viewed in Table 3.2.

Value	Parameter Name & Range (bytes)
Message	$0 \leqslant M < 2^{32}$
Salt	$8 \leqslant S < 2^{32}$
Parallelism degree	$0 < p < 2^{24}$
Memory size	$8p \leqslant m < 2^{32}$
Iteration count	$0 < t < 2^{32}$
Hash length	$4 \leqslant \tau < 2^{32}$
Secret value	$0 \leqslant K < 2^{32}$
Associated data	$0 \leqslant X < 2^{32}$
Type	$0 \leqslant y \leqslant 2$
Version	1

Table 3.2: Input parameters for Argon2.

The most relevant parameters for this thesis are *memory size*, *parallelism degree*, and *iteration count*. The memory size parameter m indicates how much memory is to be used, measured in KiB as it is the algorithm's internal block size. The actual amount of blocks allocated though is actually m', which is m rounded down to the nearest multiple of p.

The parallelism degree p indicates how many parallel threads the algorithm supports. This has an effect on the memory layout, as will be explained further in Section 3.5.4. The iteration count parameter t indicates how many iterations, or *passes*, are to be performed in the compression phase.

A description of how the algorithm and its internal parts work is provided below, inspired by the detailed explanation found in the official *RFC* 9106 [6]. A reference implementation, written to be executed on a CPU, can be found in the official Argon2 Github repository [30]. An incomplete VHDL implementation is also available which is described further in Section 4.1 [40].

3.5.1 Variable Length Hash Function - H′

H' is the internal hashing function of Argon2. An algorithmic description in pseudo-code can be found in Algorithm 4. It operates on two different input parameters, M and T. M is the

message that is to be hashed, and T is the desired tag (hash) length. Let V_i be a 64-byte block, and W_i the first 32 bytes of that block. H is BLAKE2b.

Algorithm 4 $H'(M, T)$

if $T \leqslant 64$ **then**
  $tag \leftarrow H^T(le32(T) \parallel M)$
else
  $r \leftarrow \lceil \frac{T}{32} - 2 \rceil$
  $V_1 \leftarrow H^{64}(le32(T) \parallel M)$
  $V_2 \leftarrow H^{64}(V_1)$
  ...
  $V_{r+1} \leftarrow H^{T-32r}(V_r)$
  $tag \leftarrow W_1 \parallel W_2 \parallel ... \parallel W_r \parallel V_{r+1}$
end if

3.5.2 Permutation Function - P

The *permutation function P* is based on the round function of BLAKE2b. It takes a 128-byte block S as its single input parameter and returns a 128-byte block R. These blocks will be viewed as $4x4$ matrices, with an element size of 8 bytes. For example, S will be viewed as the matrix displayed in Figure 3.1. It makes use of an internal destructive function called F_G, which takes four 8-byte inputs, a, b, c, and d. This is very similar to the mixing function in BLAKE2b, and the behavior of F_G is defined in Algorithm 5. P will apply the F_G function on each row of S, subsequently applying it on each diagonal. It is defined in Algorithm 6.

S_0	S_1	S_2	S_3
S_4	S_5	S_6	S_7
S_8	S_9	S_{10}	S_{11}
S_{12}	S_{13}	S_{14}	S_{15}

Figure 3.1: A graphical representation of how block S is viewed as a matrix, where the first block S_0 is the 64 most significant bits (8 bytes).

Algorithm 5 $F_G(a, b, c, d)$

$a \leftarrow (a + b + 2 \cdot trunc(a) \cdot trunc(b)) \mod 2^{32}$
$d \leftarrow (d \oplus a) \ggg 32$
$c \leftarrow (c + d + 2 \cdot trunc(c) \cdot trunc(d)) \mod 2^{32}$
$b \leftarrow (b \oplus c) \ggg 24$
$a \leftarrow (a + b + 2 \cdot trunc(a) \cdot trunc(b)) \mod 2^{32}$
$d \leftarrow (d \oplus a) \ggg 16$
$c \leftarrow (c + d + 2 \cdot trunc(c) \cdot trunc(d)) \mod 2^{32}$
$b \leftarrow (b \oplus c) \ggg 63$

Algorithm 6 $P(S)$

$F_G(S_0, S_4, S_8, S_{12})$ # Rowwise applicaton of F_G
$F_G(S_1, S_5, S_9, S_{13})$
$F_G(S_2, S_6, S_{10}, S_{14})$
$F_G(S_3, S_7, S_{11}, S_{15})$
$F_G(S_0, S_5, S_{10}, S_{15})$ # Diagonalwise applicaton of F_G
$F_G(S_1, S_6, S_{11}, S_{12})$
$F_G(S_2, S_7, S_8, S_{13})$
$F_G(S_3, S_4, S_9, S_{14})$

3.5.3 Compression Function - G

The *compression function* is based on the compression in BLAKE2b. It takes two 1024-byte blocks, X and Y, as input parameters and returns a single 1024-byte block W. It also contains intermediate state blocks R, Q, and Z, all with a size of 1024 bytes. These blocks will be viewed as $8x8$ matrices, with an element size of 8 bytes. For example, R will be viewed as the matrix displayed in Figure 3.2. How it works is described in Algorithm 7.

	R_{c0}	R_{c1}	R_{c2}	R_{c3}	R_{c4}	R_{c5}	R_{c6}	R_{c7}
R_{r0}	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7
R_{r1}	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}
R_{r2}	R_{16}	R_{17}	R_{18}	R_{19}	R_{20}	R_{21}	R_{22}	R_{23}
R_{r3}	R_{24}	R_{25}	R_{26}	R_{27}	R_{28}	R_{29}	R_{30}	R_{31}
R_{r4}	R_{32}	R_{33}	R_{34}	R_{35}	R_{36}	R_{37}	R_{38}	R_{39}
R_{r5}	R_{40}	R_{41}	R_{42}	R_{43}	R_{44}	R_{45}	R_{46}	R_{47}
R_{r6}	R_{48}	R_{49}	R_{50}	R_{51}	R_{52}	R_{53}	R_{54}	R_{55}
R_{r7}	R_{56}	R_{57}	R_{58}	R_{59}	R_{60}	R_{61}	R_{62}	R_{63}

Figure 3.2: A graphical representation of how block R is viewed as a matrix, where the first block R_0 is the 64 most significant bits (8 bytes). R_{ri} and R_{ci} are labels for the rows and columns, respectively.

Algorithm 7 G(X, Y)

$R \leftarrow X \oplus Y$
for $i \leftarrow 0; i < 8; i \leftarrow i + 1$ **do**
  $Q_{ri} \leftarrow P(R_{ri})$
end for
for $i \leftarrow 0; i < 8; i \leftarrow i + 1$ **do**
  $Z_{ci} \leftarrow P(Q_{ci})$
end for
$W \leftarrow Z \oplus R$

3.5.4 Argon2d Operation

A complete description of how Argon2d operates is described below. The explanation is divided into eight defined steps according to RFC 9106 [6].

1. Calculate the initial hash value H_0 as

$$H_0 = H^{64}(le32(p) \parallel le32(T) \parallel le32(m) \parallel le32(t) \parallel le32(v) \parallel le32(y) \parallel le32(m) \parallel le32(|P|) \parallel P \parallel le32(|S|) \parallel S \parallel le32(|K|) \parallel K \parallel le32(|X|) \parallel X),$$

2. Allocate m′ blocks of size 1024 bytes each, where

$$m' = 4p \cdot \lfloor \frac{m}{4p} \rfloor.$$

This memory will be viewed as a matrix B of p rows and $q = m'/p$ columns, where every element is a 1024-byte block. The columns are grouped into 4 *slices*. An intersection of a row and a slice is called a *segment*. Note that the slice width varies depending on the input parameters, and thus also the segment size. As an example, let $m = 32$, $p = 3$. The amount of allocated blocks will then be

$$m' = 4 \cdot 3 \cdot \lfloor \frac{32}{4 \cdot 3} \rfloor = 24,$$

and the amount of columns q will be

$$q = \frac{m'}{p} = \frac{24}{3} = 8.$$

The resulting matrix B can be viewed in Figure 3.3. The blocks are labeled as B_i according to how they are sequentially stored in memory.

Segment	Slice 0		Slice 1		Slice 2		Slice 3	
Lane 0	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
Lane 1	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}
Lane 2	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}

Figure 3.3: Memory layout matrix B of Argon2 with parameters $m = 32$ and $p = 3$.

3. Calculate the initial values in column 0 as

$$B[i,\, j] = H'^{1024}(H_0 \,||\, i_{le32} \,||\, j_{le32}),$$

for every $0 \leqslant i < i_{max}$ and $j = 0$.

4. Calculate the initial values in column 1 as

$$B[i,\, j] = H'^{1024}(H_0 \,||\, i_{le32} \,||\, j_{le32}),$$

for every $0 \leqslant i < i_{max}$ and $j = 1$.

5. Calculate the remaining blocks of matrix B as

$$B[i,\, j] = G(B[i,\, j-1]),\, B[l,\, z]),$$

for every $0 \leqslant i < i_{max}$ and $2 \leqslant j < j_{max}$. Indexes l and z, which determines the second input block, are derived from the previous block $B[i,\, j-1]$. Let J_1 and J_2 be the 32 most and 32 next most significant bits of $B[i,\, j-1]$ respectively. Then

$$l = J_2 mod p,$$

which is the index of the lane the second input block will be fetched from. If the block that is to be calculated, B[i, j], is in the first slice, then $l = i$ instead. Then, let W be a set of indices. If l is the index of the current lane ($l = i$), W contains the indices of all blocks from the 3 last finished segments. W additionally contains the indices of all computed blocks from the current segment from the current pass, except the first input block $B[i, j-1]$. If l is not the index of the current lane ($l \neq i$), then W includes the indices of all blocks from the 3 last finished segments in lane l. Then calculate the value n as

$$n = |W| - 1 - \frac{\frac{J_1^2}{2^{32}}|W|}{2^{32}}.$$

Finally, z is fetched from W as

$$z = W[n].$$

6. If the number of passes is $t = 1$, skip this step. Else, recalculate all blocks of matrix B as described by

$$B[i,\, j] = G(B[i,\, j-1]),\, B[l,\, z])\ xor\ B[i,\, j],$$

for every $0 \leqslant i < i_{max}$ and $0 \leqslant j < j_{max}$. Perform this step $t-1$ times. The values l and z are derived in the same way as in the previous step, with the only difference that l will be calculated as

$$l = J_2 mod p,$$

regardless if B[i, j] is in the first slice.

7. Calculate the final block C as

$$C = B[0,\, j_{max}-1]\ xor\ B[1,\, j_{max}-1]\ xor\ ...\ xor\ B[i_{max}-1,\, j_{max}-1].$$

8. Calculate the output tag (hash) as

$$Tag = H'^{T}(C).$$

3.5.5 Recommended Parameters

Biryukov et al. [5] provide some guidelines for choosing parameter values. The parameters m, p, and t should be chosen such that each Argon2 call uses as many threads and as much memory as can be allowed while still finishing in a specified time frame x. The execution time is modified by adjusting the parameter t. If the execution time exceeds x even for $t = 1$, reduce m accordingly. In the report, it is stated that a hash size of 128 bits should be enough in most cases, which is the chosen default hash size. With regards to the Argon2 type, if side-channel attacks are considered a viable threat then Argon2id should be chosen. Otherwise, Argon2d is the optimized option when it comes to cryptocurrencies and backend servers [5].

A number of general recommendations are also presented if a non-tailored solution is acceptable. For cryptocurrency mining, the recommended parameters are $m = 2^{18}$ (approximately 250 MB), $p = 2$, and $t = 1$, translated to an execution time of 0.1 seconds on a 2Ghz CPU with 1 core. In the case of password hashing, there are two general recommendations. The primary recommendation is Argon2id with parameters $t = 1$, $p = 4$, $m = 2^{21}$ (2 GiB), 128-bit salt, and 256-bit hash size. If much less memory is available, the secondary recommendation is Argon2id with $t = 3$, $p = 4$, $m = 2^{16}$ (64 MiB), 128-bit salt, and 256-bit hash size [6].

3.6 Field Programmable Gate Array

A *Field Programmable Gate Array* (FPGA) is a form of hardware circuit that can be programmed after it has been manufactured. It consists of a number of Configurable Logic Blocks (CLBs) that contain look-up-tables (LUTs), flip-flops (FF), multiplexers and carry logic. The different CLBs all have pins that can be routed to other CLBs through channel wires. With the help of connection and switch boxes, these connections are also highly configurable. The channel wires are connected to input/output blocks (IOBs) that connect the internal logic to the external parts of the board. More components of an FPGA are described further in the FPGA synthesis report heading [13].

A desired design is described with a *Hardware Descriptive Language* (HDL) such as VHDL and Verilog. This design is then synthesized into a netlist that contains the logic and interconnection required by the CLBs. This is then mapped onto the actual circuit where the available resources and their physical location are taken into account when creating a new blueprint for how the FPGA should be programmed. Although similar, neither the amount of resources nor the physical area of a board can be exceeded in order for a design to be realizable.

3.6.1 FPGA Synthesis Report

An FPGA has a number of resources available. The synthesis will determine how these are utilized to achieve the given design. After synthesizing a design that targets a specific FPGA with ISE, a resource usage report will be filled. This report details seven different measurements, some show the available amount of a given resource and others are a measurement of how well different resources are utilized [43, 39].

- **Slice Registers** are composed of flip-flops which are circuits constructed by logic gates that can store one bit of information.
- **Slice LUTs** refers to the usages of available LUTs divided by the number of slices. Multiple LUTs are contained within one slice. LUTs are a collection of tables that hold a

mapping between a number of bits and an output. This enables them to effectively implement any function from a specific number of input bits.

- **Fully used LUT-FF pairs** refer to the utilization of LUTs and Flip-flops pairs. Inside the CLB, LUTs and FFs are contained together within a slice.
- **Bonded IOBs** refer to how many *Input/Output Blocks* (IOBs) are utilized.
- **Block RAM/FIFO** are dedicated collections of registers that can store more data than the previously mentioned slice registers.
- **BUFG/BUFGCTRLs** or *Global Clock Control Buffers* are multiplexers with two inputs that are used for glitch-free clock switching.
- **DSP48E1s** or *Digital Signal Processing blocks* are components dedicated to arithmetic operations. While their functionality could be implemented using CLBs, it comes at the cost of a greater amount of required LUTs and FFs.

3.7 Xilinx Development Boards

Xilinx provides a type of hardware specifically designed for streamlining the development process of FPGA circuitry. They are referred to as development boards and are composed of an FPGA mounted on a *Printed Circuit Board* (PCB) along with a multitude of helpful components.

Two of these were of great use for this project, namely the *SP605* development board containing a *Spartan-6* FPGA and the *ML605 development board* containing a Virtex-6 FPGA. The Virtex-6 is considerably larger than the Spartan-6, and its related development board contains a few more advanced components, as well as a larger memory. The relevant resources for this project were the oscillators used for clocking, the DDR3 RAM, communication interfaces, and the buttons and LEDs used for debugging. A list of their key components defined in the introductory user guides *Getting Started with the Xilinx Spartan-6 FPGA SP605 Evaluation Kit* and *Getting Started with the Xilinx Virtex-6 FPGA ML605 Evaluation Kit* can be found in Table 3.3 and 3.4 respectively [14, 15].

SP605	
Spartan-6 FPGA	• XC6SLX45T-3C in FGG484 package
Configuration	• Onboard JTAG configuration circuitry
Memory	• 128 MB DDR3 • 32 MB parallel BPI flash memory
Communications	• USB To UART bridge • PCI Express® x1 edge connector
Clocking	• 200 MHz oscillator (differential) • 27 MHz socketed oscillator (single-ended)
Control	• 4 Pushbuttons • 4 DIP switches
Power	• 12V wall adapter or ATX power supply • Voltage and current measurement capability of 2.5V, 1.5V, and 1.2V supplies

Table 3.3: SP605 - Evaluation Kit key components.

ML605	
Virtex-6 FPGA	• XC6VLX240T-1FFG1156 device
Configuration	• Onboard JTAG configuration circuitry
Memory	• 512 MB DDR3 • 32 MB parallel BPI flash memory
Communications	• USB To UART bridge • PCI Express® Gen1 x8 and Gen2 x4 edge connectors
Clocking	• 200 MHz oscillator (differential) • 66 MHz socketed oscillator (single-ended)
Control	• 4 Pushbuttons • 4 DIP switches
Power	• 12V wall adapter or ATX power supply • Voltage and current measurement capability of 12V, 2.5V, 1.5V, 1.2V, and 1.0V supplies

Table 3.4: ML605 - Evaluation Kit key components.

3.8 Hashcat

Hashcat is an open-source password recovery tool released in 2009 that claims to be the fastest in the world [17]. It supports over 300 optimized hashing algorithms, has an active community, and has a very well-documented process of how to implement support for new algorithms. The user provides the tool with a list of hashes and Hashcat will run the specified hashing algorithm on a number of password candidates in the hope of matching the resulting hashes with the user-supplied ones. This could be done in different attack modes, such as brute-force and dictionary attack mode. A brute-force attack means generating all possible combinations given a defined character space and a password length. In a dictionary attack, on the other hand, the attacker provides a list of pre-generated password candidates such as a collection of passwords from a password leak [17].

Implementing hashing algorithms in Hashcat requires two files: a **module** and a **kernel**. These together form what is referred to as a Hashcat plugin. The module is a C file that contains configurations of the formatting, decoding, encoding, etc. and is executed on the CPU. The kernel file contains the logic for the actual hashing algorithm to be executed on the backend device. Although there is support for CPU hashing, Hashcat has historically been focused on utilizing the parallel computing power of the GPU with support for OpenCL, CUDA, and HIP [16].

The kernel can be of two different types: **fast** or **slow**. In slow kernel mode, there are three kernels that make up the hashing process: a message-generating kernel, the developer-defined hashing kernel, and a comparison kernel. Depending on the attack mode and parameters, the message-generating kernel will generate all messages to be hashed and load these into a buffer that the hashing kernel will read from. Once the hashing kernel runs it reads messages from this buffer, performs the developer-defined operations, and loads the resulting hash into another buffer that the comparison kernel will read from and do the developer-defined comparison. It does this until all messages produced by the first kernel have been hashed and compared in the latter kernels.

It is referred to as slow because unless the hashing algorithm is slow, the bottleneck will be reading from GPU memory. The principle with a fast kernel is to instead load a basic form of the messages to the kernel executing the hashing and manipulate them from there, as defined by the developer [16].

3.9 UART

Universal Asynchronous Receiver-Transmitter (UART) is a peripheral device used for asynchronous serial communication often used in unison with Recommended Standard 232 [32] and 485 [33]. As it is also one of the earliest communication devices [38], it has found widespread use throughout the years [36]. Since it only requires a single wire (two for full duplex capabilities), it can be a good choice when resources are limited [36]. Several UARTs can often be found in microcontroller chips, and specialized ones can be used for devices such as automobiles and smart cards [38].

A UART data packet contains several components. Firstly, a *start bit* signals to the receiver that a new message is incoming. Secondly, it contains anywhere from five to nine *data bits* (usually eight) that form the actual message. Thirdly there is an optional *parity bit* that is used to verify the integrity of the message. Lastly, there are one or two *stop bits* that tell the receiver that the message is complete [38].

3.10 PCI Express

Introduced in 2002, *Peripheral Component Interconnect Express* (PCI Express, or simply PCIe) is a common interface in computers used for connecting peripherals. Examples of devices that

utilize the PCIe bus are graphic cards, memory units, and others that have high requirements for communication speed [12].

Currently, on generation 6, released in 2021, the PCIe is capable of communication speeds of up to 64GT/s when using the 16x-channel configuration, or 256GB/s bi-directionally. In 2025 it is estimated to be able to reach speeds of 512GB/s, or 128 GT/s, based on the planned specifications by PCI-SIG and the trend of doubling transmission speeds every three years [2, 29].

4 Method

This chapter presents the methodology that was used to obtain the results. The biggest part of the project was to establish an implementation of the Argon2 algorithm for VHDL which could be programmed on a FPGA. The main tool used for this purpose was *Xilinx ISE Design Suite* [20], which was useful for programming, debugging, synthesizing, implementing the synthesis, and finally programming the FPGA via JTAG. *ISIM* was used for simulation [21].

A Spartan6 - SP605 evaluation kit was used during the development. It provided several useful components in addition to the actual FPGA built-in on the board. The most important additions to this project were the external DDR3-RAM for matrix allocation and LED lights used for debugging [14].

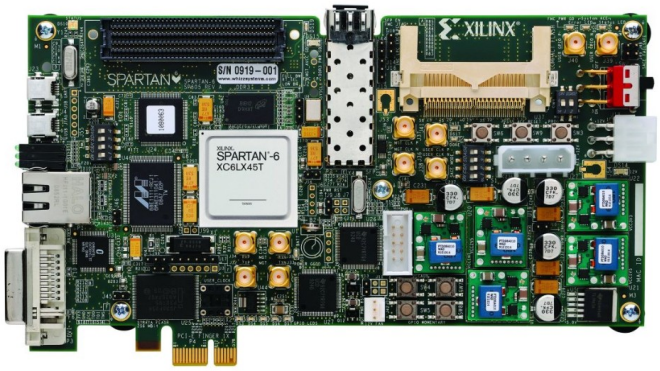

Figure 4.1: Spartan 6 - SP605 Evaluation Kit.

4.1 Existing Argon2 and BLAKE2b VHDL Implementations

There seems to be only one publicly available FPGA implementation of Argon2, which is an unfinished VHDL design made by Christian Krieg. According to the description of the project, it is a work in progress and therefore not synthesizable. It contained an outline of the main Argon2 state machine. There was also an implementation of H' but the author admits that it hadn't been tested, and after some investigation turned out to contain a multitude of bugs that needed to be addressed. The project however has fully working BLAKE2 and compression components [40].

While unfinished and prone to bugs as described above, it served as a starting point for this thesis. Although some components were ready for use, they would need some minor modifications in the end as well. His project can be found in a public GitHub repository [40].

4.2 System Design

The chosen design was inspired by the previously mentioned attempt and opted to implement all of the Argon2 logic on the Spartan 6. A couple of necessary components were identified, some directly integral to the Argon2 method of operation and others to the infrastructure of the system such as communication and memory components. The components were contained within the entity `Cracker`. The design is illustrated in Figure 4.2.

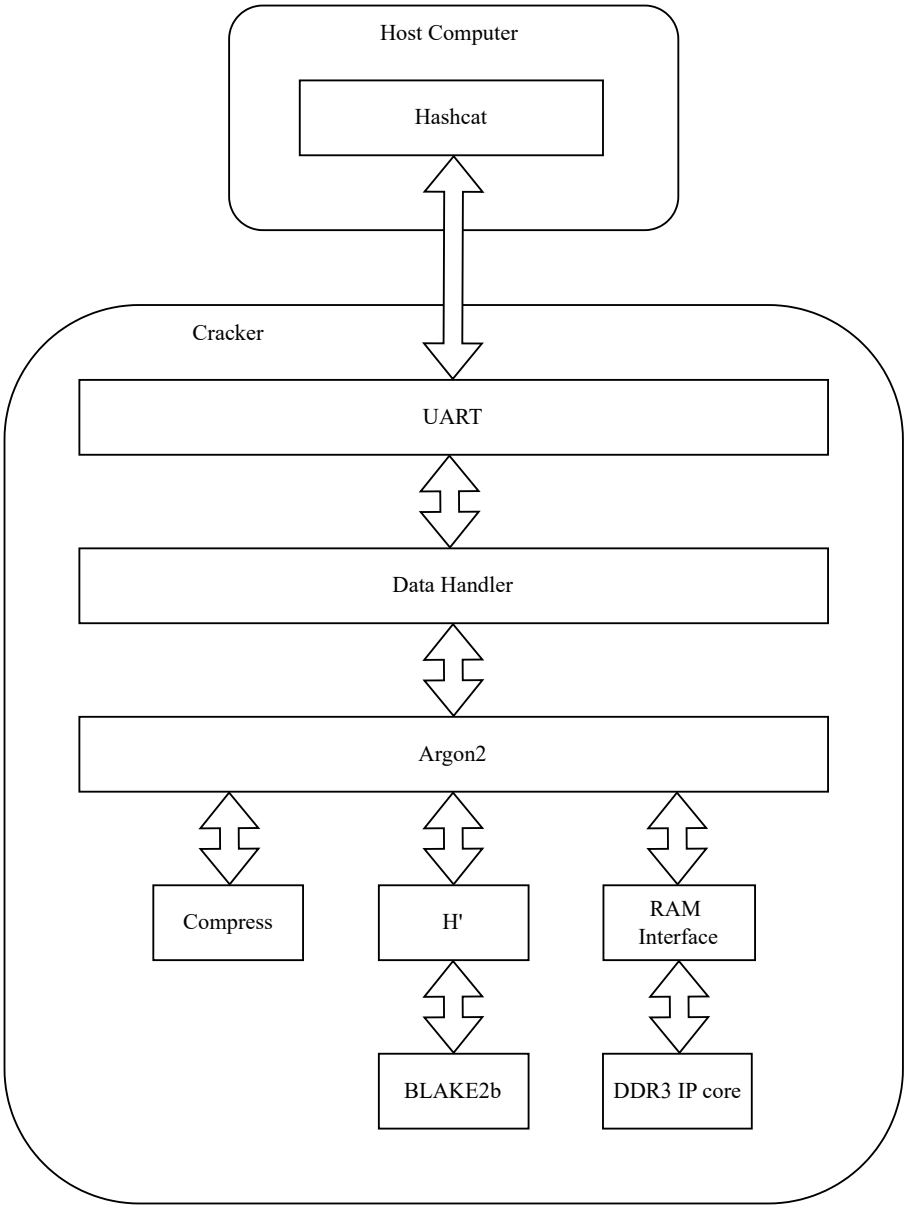

Figure 4.2: Overview of the intended cracking system.

The components will be shown in-depth in the following sections. The architectural diagrams will not necessarily contain every signal defined in the code. Every mention of `Clk` refers to the User Clock (66 MHz oscillator). Thin arrows with a filled tip represent binary signals, thick arrows represent logic vectors, and thin arrows without a filled tip represent integers.

4.3 Argon2

The Argon2 component was responsible for the main execution loop of the Argon2 hashing algorithm. It was implemented as a *Finite State Machine* (FSM) and communicates with the RAM, Compress, and H' components. A diagram of the Argon2 component is presented in Figure 4.3. Out of the input parameters presented in Table 3.2, only the password and salt are communicated through the Data Handler component. The other parameters are hard-coded.

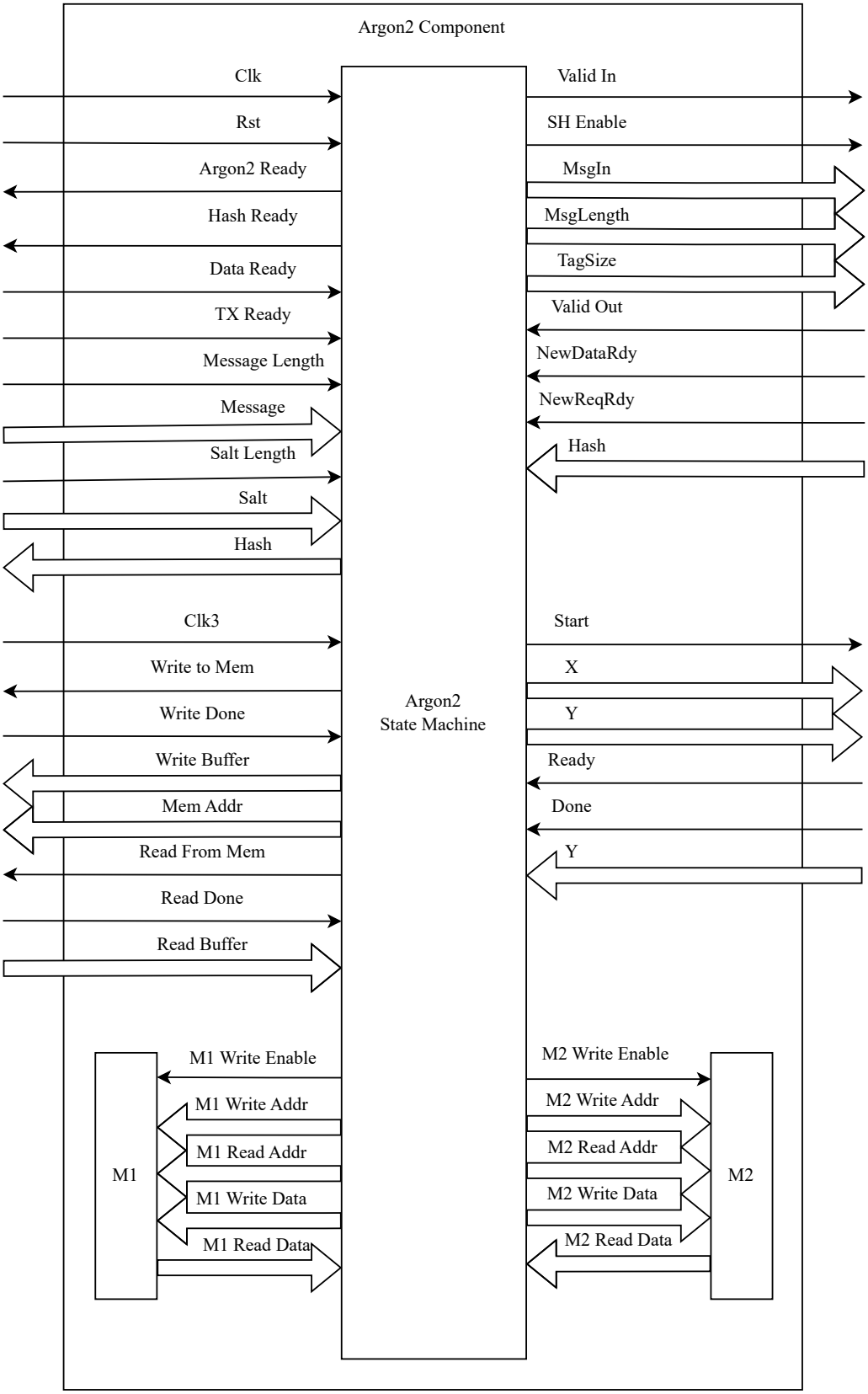

Figure 4.3: Diagram of the Argon2 component.

The Argon2 component contains two *Block RAM* (BRAM) components scaled for one block each. The FSM contains ten different states and is mostly sequential. The function of the states corresponds heavily with the distinct steps of the algorithm presented in Section 3.5. A state diagram can be found in Figure 4.4.

- **Idle**: The default state. Wait for all input parameters to be set, then jump to `Init`.
- **Init**: Concatenate the given Argon2 parameters into a message that is passed to `H'`, 8 bytes at a time. The output of `H'` will be the initial hash H_0. Jump to `Col0`.
- **Col0**: Calculate every block in the first column using `H'`, one block at a time. After block output from `H'` is received, write the block to its appropriate index in the DDR3 RAM by jumping to `Mem Write`. Jump to `Col1` when the first column is filled.
- **Col1**: Calculate every block in the second column using `H'`, one block at a time. After block output from `H'` is received, write the block to its appropriate index in the DDR3 RAM by jumping to `Mem Write`. Jump to `Comp` when the second column is filled.
- **Comp**: During the first iteration, fetch two blocks through state shifts to `Mem Read` and use them as input to the `Compress` component. State shift to `Mem Write` and write to the appropriate slot. During any additional iterations, fetch two blocks and perform the same action as before but before the jump to `Mem Write`, fetch the block which is in the position where the write target is and xor these blocks. Finally, write the resulting block into the same position. Jump to `Final block state` when all iterations are completed.
- **Final block**: Fetch every block in the last column using state shifts to `Mem Read` and xor them with each other to create the final block `C`. Pass this block into `H'` and jump to `Output Hash`.
- **Output Hash**: Wait for the output result from `H'` from the previous state and assign the resulting hash to it. Jump to `Idle`.
- **Mem Write**: Write what is in one of the BRAMs to the onboard DDR3 RAM. Performs cross-clock domain communication with the RAM controller. Jump to `Mem Wait`.
- **Mem Read**: Read from the onboard DDR3 RAM into one of the BRAMs. Performs cross-clock domain communication with the RAM controller. Jump to `Mem Wait`.
- **Mem Wait**: Wait for the reading/writing operation to complete before jumping back to the calling state, i.e. whichever state requested the write/read operation.

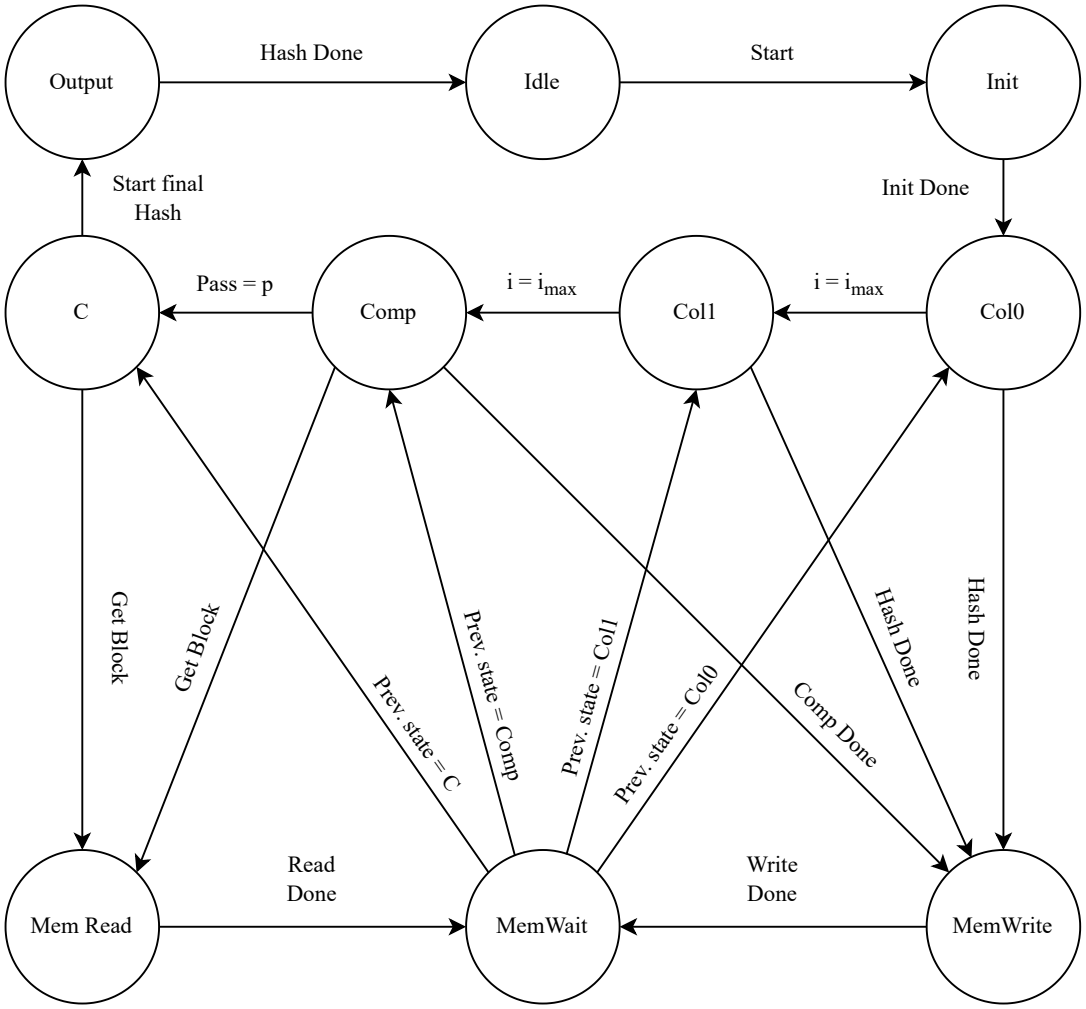

Figure 4.4: State diagram for Argon2 component implementation.

The Argon2 implementation can be split into four distinct parts: `BLAKE2b`, `H'`, `Compress`, and the overarching state machine that utilizes the three previously mentioned components. It is as previously mentioned based on the implementation made by Christian Krieg. Much of the state machine in `H'` was preserved but there were a couple of things that needed to be adjusted. The messages passed into `BLAKE2b` needed to be converted to little endian, the message passing from `Argon2` needed to be rewritten to take up less FPGA space, and a couple of bug fixes were made. In the end, only a few lines from the FG-function were kept intact from the repo while the rest was rewritten in a more sequential manner.

4.4 H′

H′ was implemented as an FSM communicating with an internal BLAKE2b component. This component was written by Christian Krieg as well, found in the public GitHub repository [8]. A diagram of the H′ component is presented in Figure 4.5

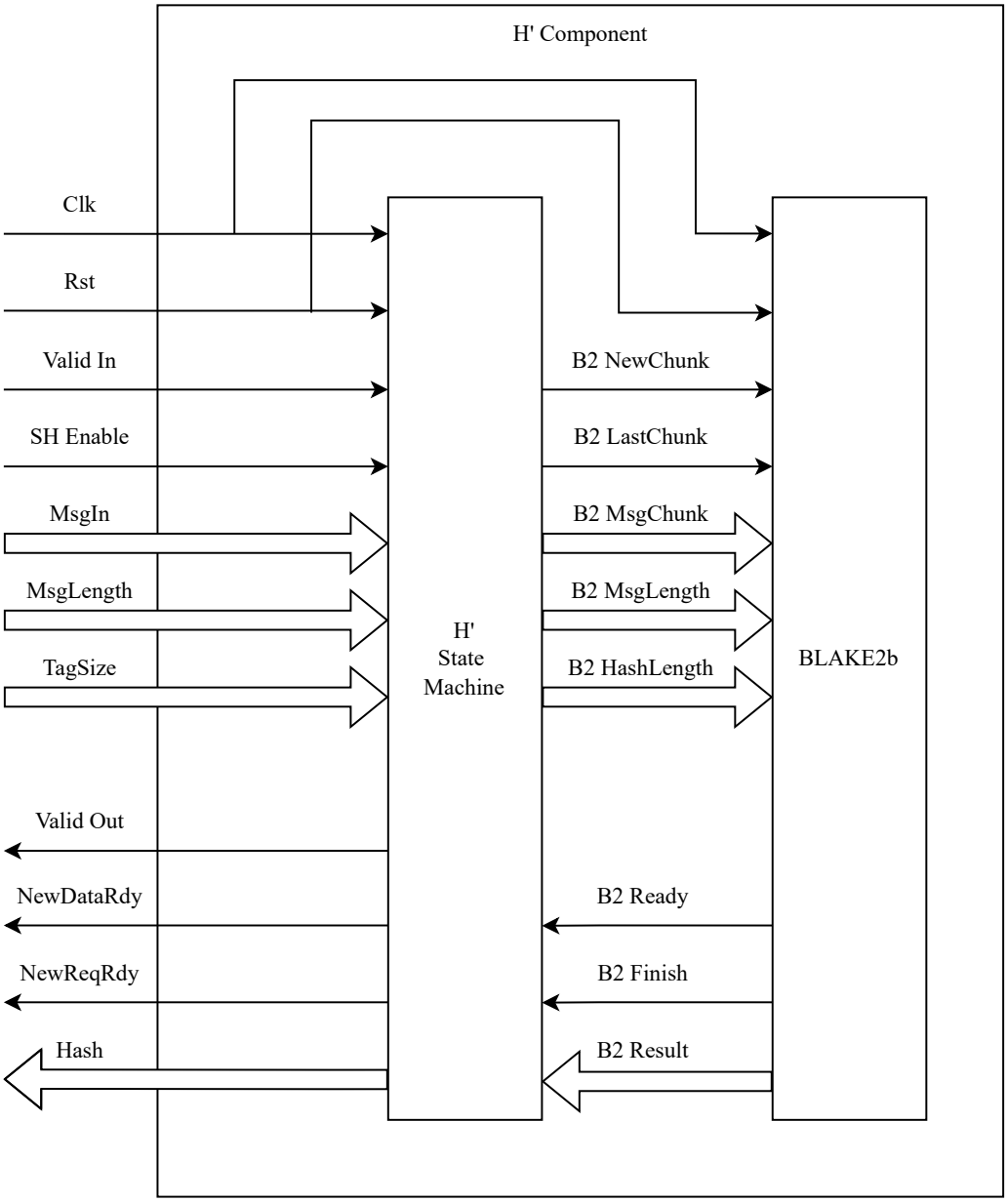

Figure 4.5: Diagram of the internal Hash function H′ component.

The main FSM is presented in Figure 4.6 and contains the following states.

- **Idle**: The default state. When H_0 is to be calculated, prepare a `BLAKE2b` message chunk, `B2 MsgChunk`, by masking 8 bits at a time until the appropriate length is reached. Once completed, `B2 NewChunk` and `B2 LastChunk` are driven high in order to signal `BLAKE2b` to begin hashing. Jump to `B2 Wait Init` to anticipate a response. In case of any message not related to H_0 and the length of the message is shorter than 128 (the

maximum size of a chunk), then `B2 MsgChunk` gets assigned directly to the message and the same process takes place as in the previous case to start the hashing. If the message is longer than 128 bytes, the first 128 bytes is used as the chunk and only `B2 NewChunk` is driven high while jumping to `Init`.

- **Init**: Wait for `BLAKE2b` to be ready to accept more chunks. Take the next 128 bytes of the message and repeat this process until the entire message has been loaded into `BLAKE2b`, except for the last 8 bytes. When this is completed, jump to `Init End`.
- **Init End**: Load the last 8 bytes into a chunk, assert `B2 NewChunk` and `B2 LastChunk` and jump to `B2 Wait Init`.
- **B2 Wait Init**: Wait for `BLAKE2b` to finish hashing. If the requested hash size, `TagSize`, is lower or equal to 64, then `Hash` is directly assigned to `B2 Result` before jumping back to `Idle`. Otherwise, jump to `Cont` or `Cont End` depending on how many more iterations are required to produce the requested hash size.
- **Cont**: Take the 64 lower bytes from `B2 Result` and use it as input to the new `BLAKE2b` chunk. Also concatenate the lower 32 bytes of every piece of `B2 Result` to form the final hash with an assertion of `ValidOut` every time the hash buffer is full to allow `Argon2` to save the block into BRAM. Jump to `Cont End` when finished.
- **Cont End**: Perform the final `Cont`-state-operation with conditions dependent on the desired size of the hash. Jump to `Idle` or `Out Wait`.
- **Out Wait**: Add the last 32 bytes left over from `BLAKE2b` to `Hash`, then assert `ValidOut` and jump to `Idle`.

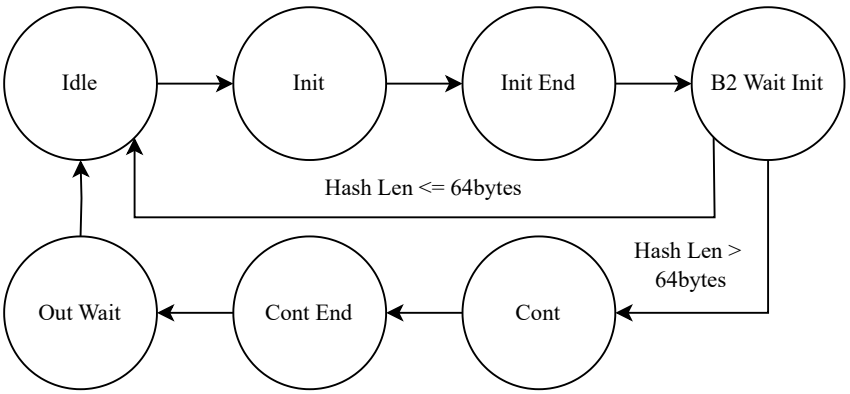

Figure 4.6: State diagram for the H′ component implementation.

4.5 Compress

The compression function was implemented as a separate component. It contains a finite state machine and uses two BRAM components, `M1` and `M2`, for storing the intermediate states of the block. It receives the two input blocks in parallel, read in chunks of 1024 bits over 8 clock cycles. The output block is similarly sent in chunks of 1024 bits over 8 clock cycles. A diagram of the implementation is presented in Figure 4.7.

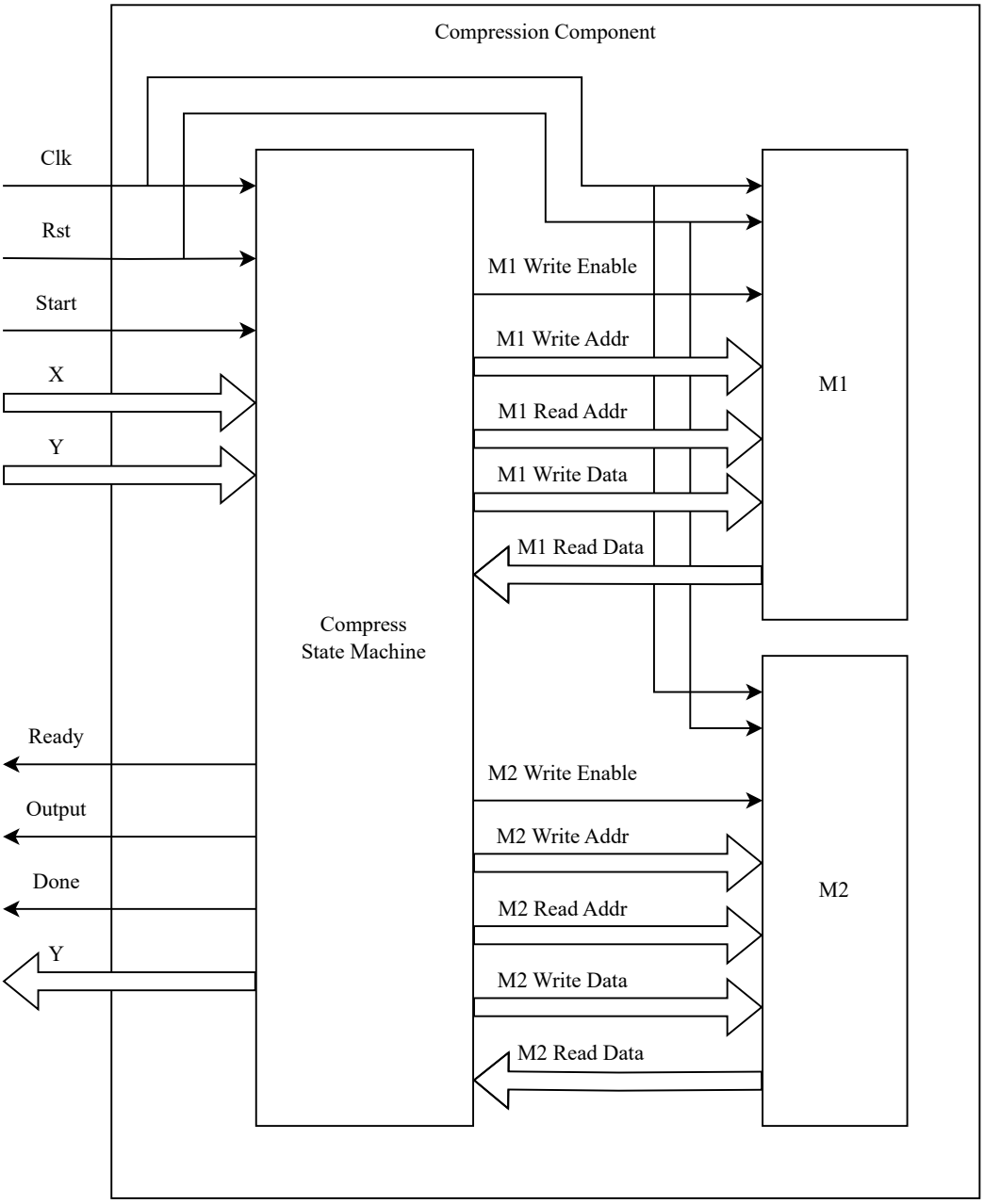

Figure 4.7: Diagram of the Compression component.

The state machine is presented in Figure 4.10 and contains the following states.

- **Idle**: The default state. Drives the output signal `Ready` high, and the rest of the output signals low. If the start signal `Start` is set to high, jump to `Receive`.
- **Receive**: Receive the two input signals `X` and `Y` by reading them for eight consecutive clock cycles, and store the xor products in the block RAM `M1` on addresses incremented in parallel to the counter. The products are stored as rows, as demonstrated in Figure 4.8. When the counter has reached 8, reset and jump to `Rows`.

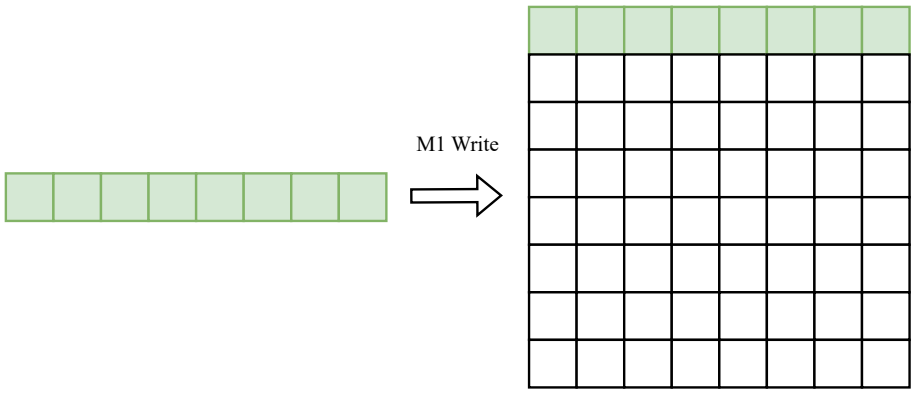

Figure 4.8: Writing layout in Block Ram M1.

- **Rows**: Read the data on addresses 0-7 of `M1` and apply the GB function on them. After it has been applied to a chunk, it is written to the second block RAM `M2` column-wise while the next chunk is being processed. The writing procedure can be seen in Figure 4.9. After the last chunk has been written to `M2`, jump to `Columns`.

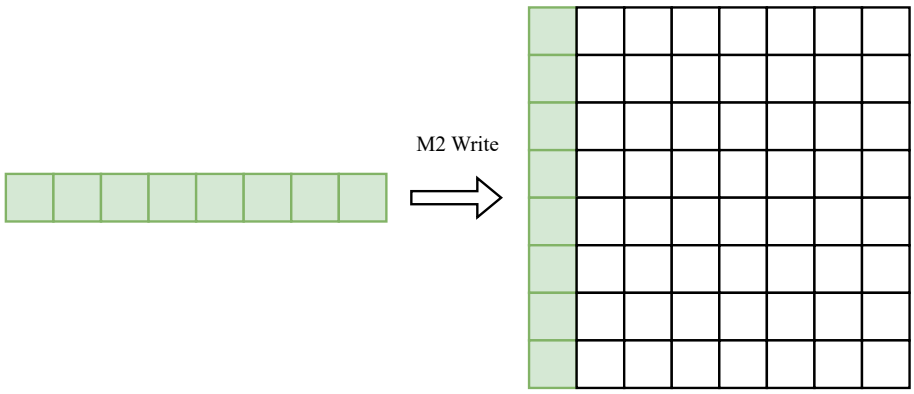

Figure 4.9: Writing layout in Block Ram M2.

- **Columns**: Read the data on addresses 0-7 of `M2` and perform the GB function on them. After it has been applied to a chunk, the result is xored with its corresponding chunk in `M1` and subsequently overwrites it as the next chunk is being processed. After the last chunk has been written to `M1`, jump to `Send`.
- **Send**: Drive the output signal `Output` high for eight clock cycles while simultaneously updating the value on signal `Y` to the corresponding chunk of the result. After eight chunks have been sent, the output signal `Done` is driven high which is followed by a jump to `Idle`.

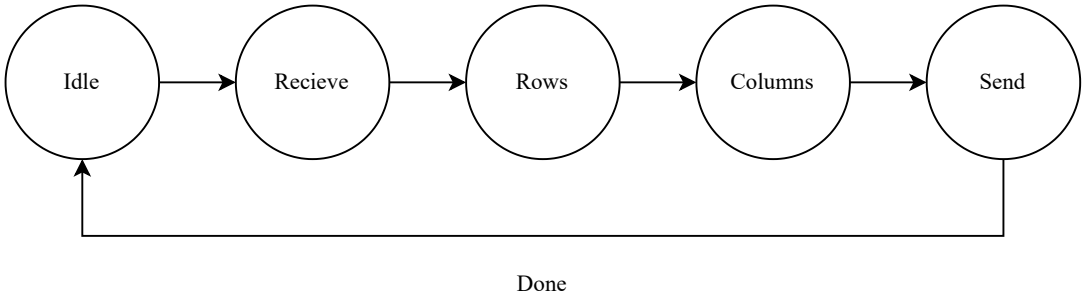

Figure 4.10: State diagram for the compression component implementation.

The column-wise writing in `Rows` is an optimization attempt. It was done in order to prepare the data for being interpreted as columns when reading from an address in state columns. This way it is cheaper in terms of clock cycles than just reading and shifting one address at a time for 8 clock cycles, and cheaper in terms of hardware resources than implementing a transpose function.

4.6 RAM Interface

Since Argon2 is a memory-hard function, more than the available block RAM is needed. Although a DDR2 RAM component was included with Krieg's implementation, it was unusable since the Spartan 6 and Virtex 6 only support DDR3. Thus, a new memory interface had to be implemented.

The SP605 has access to 128 MB DDR3 memory and the ML605 has access to 512 MB. These were interfaced through pre-built VHDL components which Xilinx refers to as IP cores [41, 42]. Therefore, the only thing that needed to be implemented was a component that utilized the supplied interface signals of the IP core. Due to differences between the Spartan and Virtex models, this component needed to be implemented twice. A description of the Virtex implementation can be found in Figure 4.11.

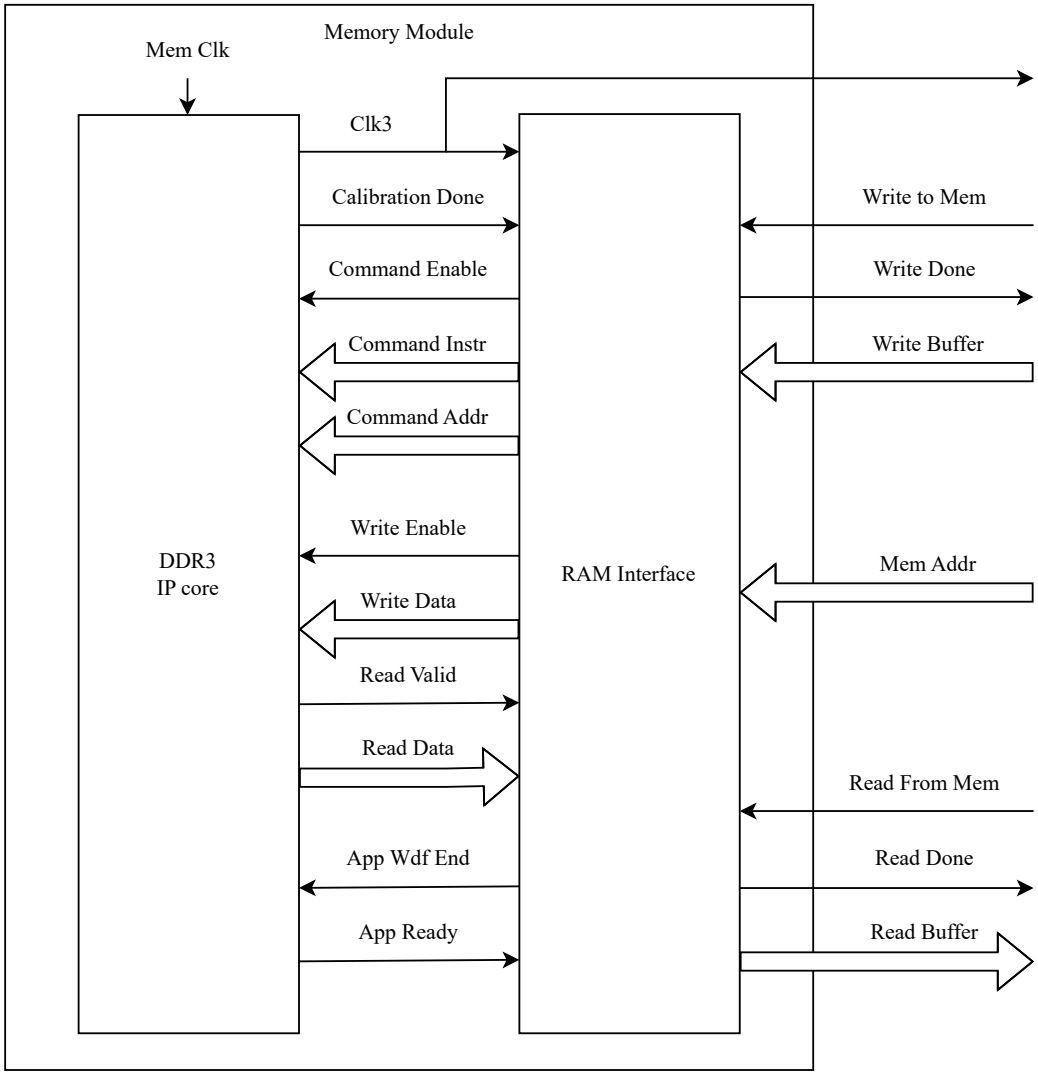

Figure 4.11: Diagram of the RAM module.

In Figure 4.11, the clock signal `Clk3` is introduced. `Clk3` is generated through a PLL from `Mem Clk` which will be closer in magnitude to `User Clk`, and is used to interface with the IP core. `Argon2` handles the clock domain crosses between this clock and `User Clk`. There are many more signals that the DDR3 IP core uses which are directly connected to pins on the board to enable this component to write to the physical RAM component. These are all generated by the tool and have been left out for the sake of simplicity.

4.6.1 RAM Interface Flow

The communication between the `RAM Interface` and the DDR3 IP core follows the protocol from the Virtex 6 Memory Interface Solutions User Guide in regard to issuing read and write commands [42].

The RAM Interface starts in an initialization state where it stays until the DDR3 IP core has finished calibration and asserted `Calibration Done`. Then it enters a waiting state where an assertion of `Write to Mem` by `Argon2` initiates a write of the 128 bytes that are located in `Write Buffer` to RAM, and `Command Instr` is set to 0. An assertion of `Read from Mem` initiates a read from RAM where the data will be put on `Read Buffer`, and `Command`

`Instr` is set to 1. The desired read/write address is put on `Mem Addr` by `Argon2`. These reads/writes are done in chunks of 16 bytes. Each 16-byte chunk occupies 8 addresses which means that a 128-byte chunk would occupy 64 addresses. `RAM Interface` keeps track of this and assigns `Command Addr` to `Mem Addr` summed up with multiples of 8. Once `App Ready` is asserted, `Command Enable`, `Read Enable/Write Enable` will be asserted to issue a read/write request. These 128-byte read/write operations are declared finished with the respective `Write Finished` and `Read Finished` signals asserted.

If `Argon2` wants to write a block (1024 bytes), it splits this into 128-byte chunks on `Write Buffer`, asserts `Write to Mem`, and then waits for `Write Finished` to be asserted. This is repeated 7 more times. `Argon2` assigns `Mem Addr` for every chunk to the appropriate address according to the pattern above. E.g. with block[0][1], `Mem Addr` would be 512, block[0][2] 1024 etc..

`App Wdf End` is used to signify that the current value on `Write Data` is the last part of the data to be written to a specific address. In this implementation, this signal will always be asserted at the same time as `Write Enable` and `Command Enable`. In Figure 4.12 the write path is visualized and below that in Figure 4.13, a simulation of the implementation is shown.

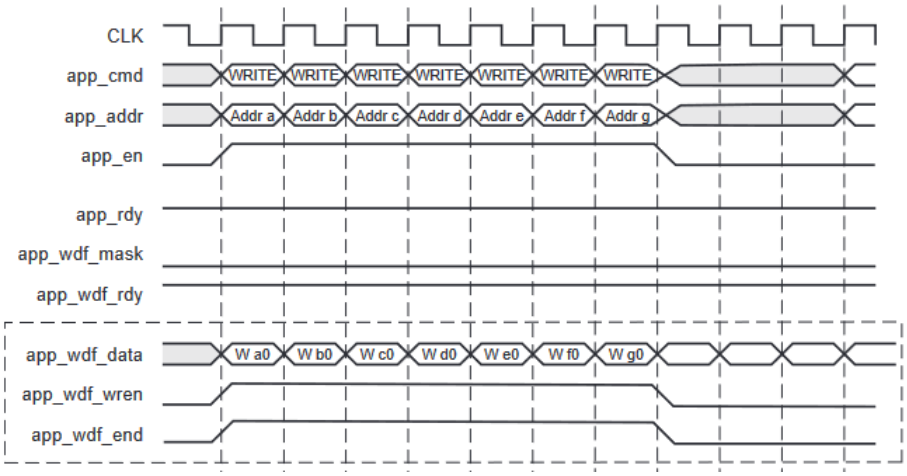

Figure 4.12: Example of back-to-back write commands from the user guide. *app_cmd* is equivalent to `Command Instr`, *app_addr* to `Command Addr`, *app_en* to `Command Enable` and *app_wdf_wren* is equivalent to `Write Enable`.

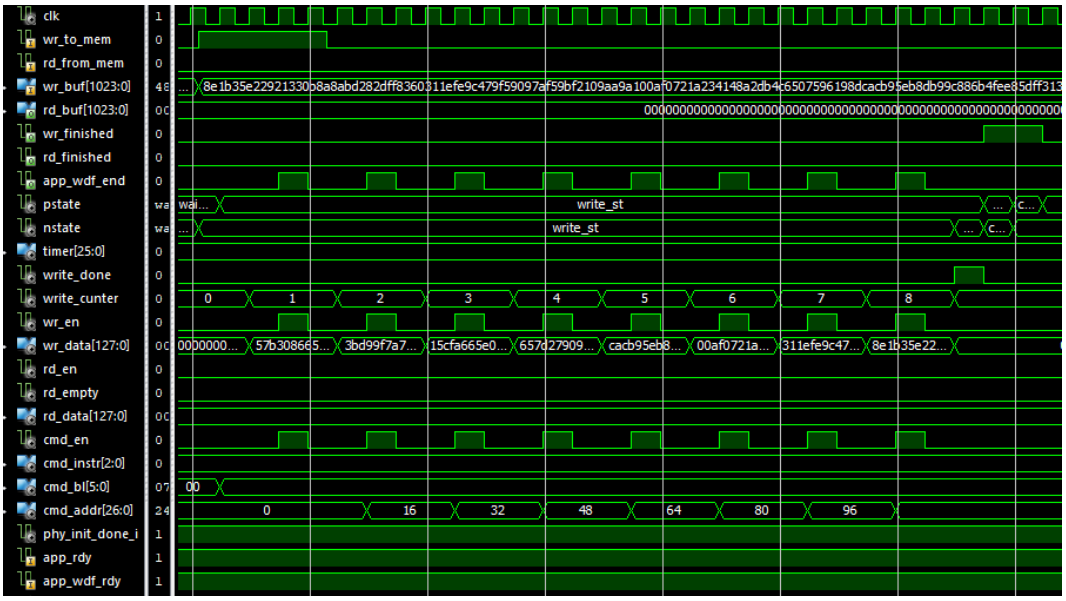

Figure 4.13: Simulation of writing 128 bytes into RAM.

The process in Figure 4.13 is repeated 8 times to write the full 1024 bytes in Figure 4.14.

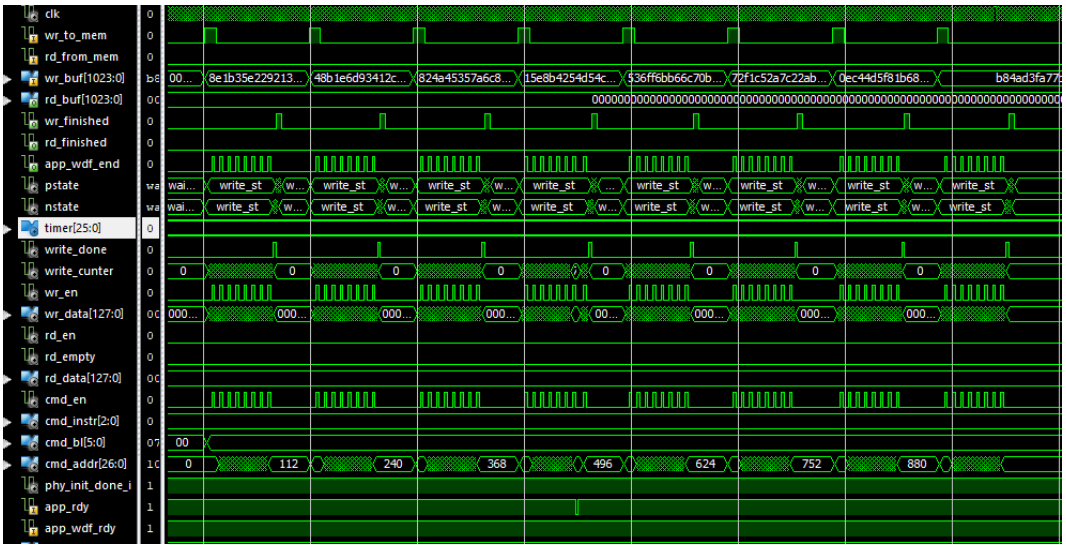

Figure 4.14: Simulation of writing 1024 bytes into RAM.

The read path follows the same command principle as the write path. When *rd_data_valid* is asserted, the data in the `Read Data` buffer is valid and the next read command is sent.

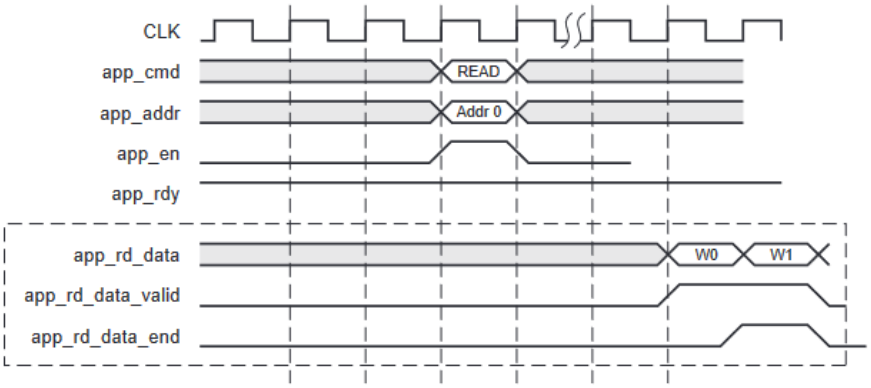

Figure 4.15: Read path from the user guide. *app_cmd* is equivalent to `Command Instr`, *app_addr* to `Command Addr` and *app_en* is equivalent to `Command Enable`.

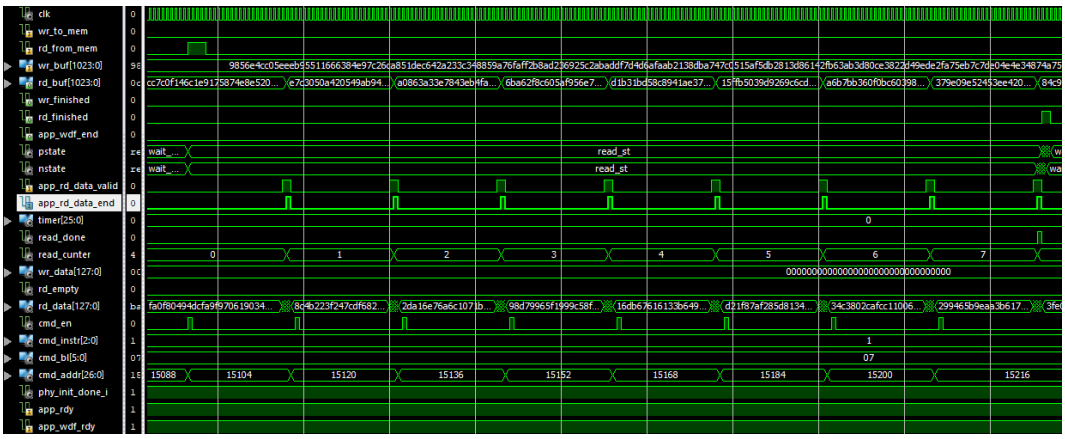

Figure 4.16: Simulation of reading 128 bytes from RAM.

4.7 UART

The UART component receives and transfers one bit at a time from and to the host using the flow control signals `RTS` and `CTS`. Each byte transfer is initiated with a start bit at the `Data RXD` signal with the rest of the data following one clock cycle behind. Once 8 bits have been received, `Data Valid` is asserted and it is now clear for the Data Handler component to read from the `Data In` logic vector. Once one byte is ready to be transmitted from `Data Handler`, `Data Send` is asserted and a transmission of `Data Out` on the `Data TXD` signal begins while keeping the `Busy` signal high. Note that while the direction of signals `Data In` and `Data Out` appears to be interchanged, this makes sense in the scope of the whole Cracker system. `Data In` is the result of received data from the external host, and `Data Out` contains the data that the UART component is supposed to send. This UART implementation is downloaded from [26].

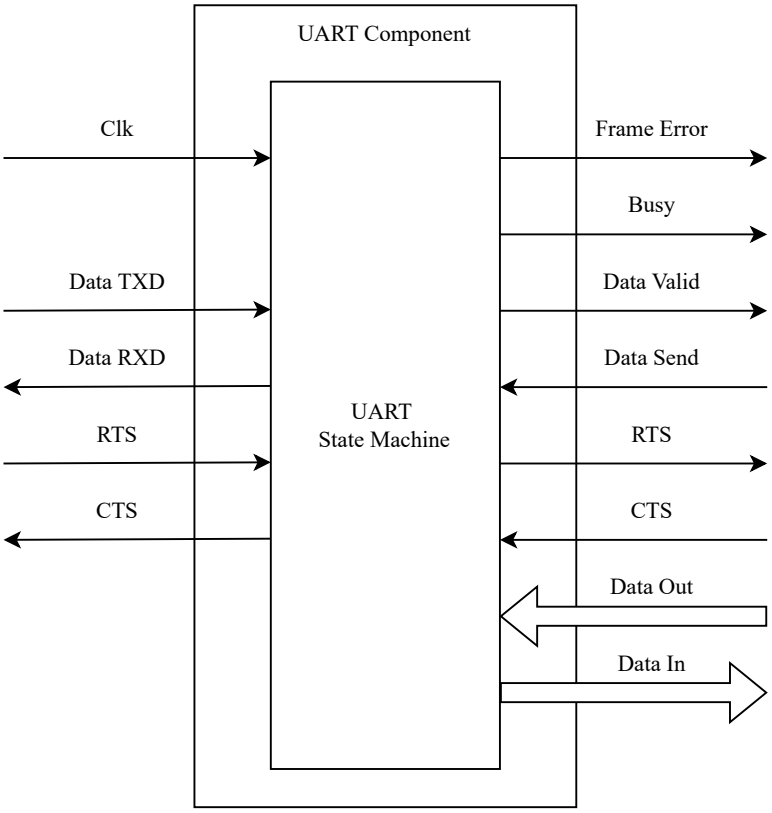

Figure 4.17: Diagram of UART communication component.

4.8 Data Handler

The Data Handler component accumulates the bytes received from the UART component that are to be used as parameters for the Argon2 component. Once the whole `Message`, `Message Length`, `Salt`, and `Salt Length` have been received from `UART`, `Data Ready` is asserted and `Argon2` is free to use the data that is present on the four signals. Once `Data Ready` and `Argon2 Ready` are asserted, `Argon2` will begin the hashing process. `Data Handler` will now be available to receive the next message to be hashed while waiting for `Argon2` to finish.

Once `Argon2` has generated the desired digest and asserted `Hash Ready`, `Data Handler` will begin transferring the data located on `Hash` to `UART` in single-byte chunks.

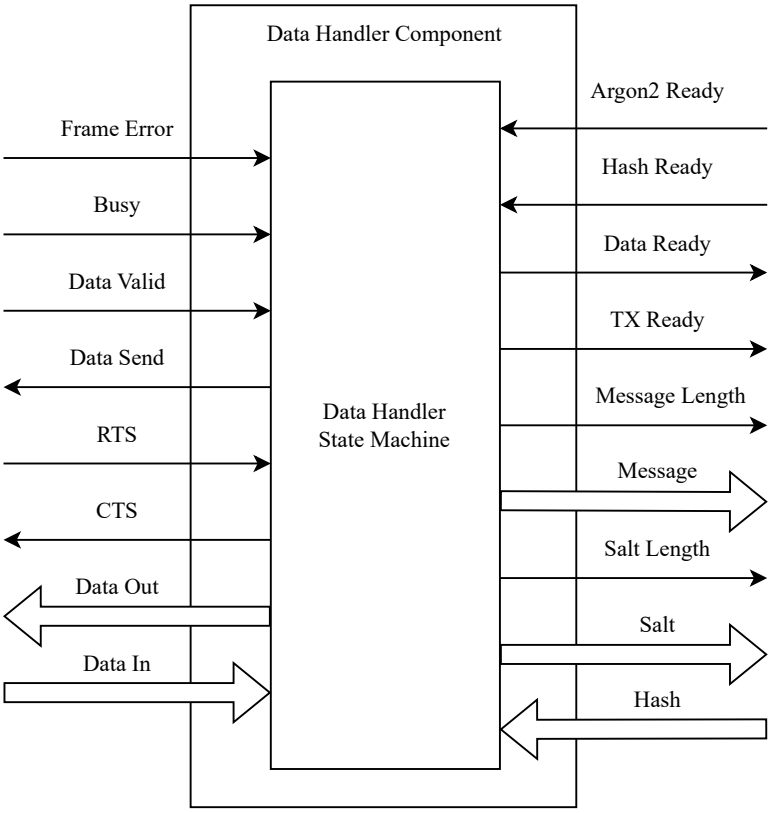

Figure 4.18: Diagram of the DataHandler component.

4.9 Integrating FPGA Support Into Hashcat

The desired goal was to get access to the messages that Hashcat generated, write them to the FPGA, perform the calculations and transmit the hash back to the host machine, and let Hashcat perform the final comparison of the hash list. The most straightforward way of doing this was to create a hashing kernel that doesn't do any form of hashing as well as a basic comparison kernel. The messages to be hashed would therefore be passed from one piece of GPU memory to another. This buffer was then accessible right before the comparison kernel was to be run, which is where the FPGA communication took place. The dictionary attack mode was used to send the passwords that were defined in the text file. However, since Argon2 has multiple configurable parameters, there was a need to be able to send more data than only passwords. The message-dictionary file was used for this cause, where every parameter was followed by a line break. To ease the synthesis time, every parameter was defined on the FPGA side except message, salt, and their respective lengths.

4.9.1 Communication with Host

All data was transferred through UART with one start bit, eight data bits, and one stop bit. The flow control signals RTS and CTS were also used. Using the Winbase library from the Win32 API, one sequence of bytes was written and one digest was fetched at a time. The input sequence consisted of four parts concatenated: a one-byte message length, a 32-byte message, a one-byte salt length, and 16 bytes corresponding to the salt.

4.10 Performance Tests

As the implementation of Argon2 was completed, the next phase of the project was to test how fast the implementation managed to produce one hash compared to the CPU. The varied parameters were parallelism, iterations, and memory usage. Although there are more adjustable options in Argon2, such as hash- and salt size, these were set to fixed. The resulting execution time for each set of parameters is the average of 10 consecutive runs. All the values from which the averages were calculated can be found in Appendix A and B.

4.10.1 CPU

The reference implementation of Argon2 has built-in support for the measurement of execution time. However, the measured time is presented in CPU time. This means that when increasing the parallelism parameter, measured execution times increase, possibly due to increased overhead. However, the algorithm should theoretically perform better when p is increased as long as there are CPU cores available. Therefore, another way of measuring was needed.

For this thesis, actual wall time is more interesting since parallelism is the strength of the FPGA. In order to get measurements of the wall time, the Linux command *time* was used together with some minor alterations to the source code. In order to show the difference, some results from the original measurement method are listed alongside the proposed measurement method in Table 4.1. These tests were carried out on an Intel(R) Xeon(R) Silver 4108 CPU @ 1.80GHz × 32 with 8 GiB of memory. The parameters were arbitrarily chosen as $m = 262144$ and $t = 1$.

Method / P	Reference	Altered
1	0.220s	0.213s
2	0.236s	0.135s
4	0.288s	0.103s
8	0.480s	0.094s

Table 4.1: Results from the built-in reference implementation and the modified measurement method (wall time).

4.10.2 FPGA Performance Estimations

The simulated FPGA performance was measured from the beginning of the initialization state to the end of the output state. Since simulations take hours to simulate just a few milliseconds, estimations were made based on a few real simulations. Although the simulation time is deterministic, the memory write and read operations will take a variable amount of time depending on how the different clocks synchronize for the current action; usually a difference

within a few clock cycles. In this case, an average was used based upon 5 measured values. The clock frequency used will be the maximum achievable clock frequency achieved by the synthesis tool.

In order to calculate the time of the algorithm, a formula was conceived that took the modifiable parameters into account. The variables below follow the state diagram in Figure 4.4 and are used to define the total computation time in Equation 4.1. The state `Compress` has been divided into two different variables, one for the first pass and one for the subsequent passes. This was done in order to accommodate for the fact that the first two columns are pre-calculated in the previous state for the first pass.

The first state `Init` runs in constant time. The last state `Output` switches between two constant values that have a slight time difference of a few μs at 46 MHz depending on if the current iteration number is larger or equal to zero. For simplicity, the slower constant value will be used in the calculations.

- **Init**$_t$: The time of `Init`. Constant.
- **Col**$_t$: The time for the states `Col0` and `Col1`.
- **FirstPass**$_t$: The time of the first iteration in state `Compress`.
- **NextPasses**$_t$: The time for the remaining passes, i.e. the rest of the `Compress` iterations.
- **FinalBlock**$_t$: The time of the state in which the last column is combined into the final block to be hashed.
- **Output**$_t$: The time of `Output`. Constant.

The total execution time can be written as:

$$Total\ time = Init_t + Col_t + FirstPass_t + NextPasses_t + FinalBlock_t + Output_t \tag{4.1}$$

The variables used for calculating the non-constant times Col_t, $FirstPass_t$, $NextPasses_t$ and $FinalBlock_t$ are presented below.

- **H'**$_t$: The time to calculate one block with `H'`.
- **Compress**$_t$: The time it takes for state `Compress` to process two blocks and output a new block. For $FirstPass_t$, this time will be one clock cycle slower, but this fact is ignored for brevity.
- **MemWR**$_t$: The time it takes for one block to be written into RAM.
- **MemRD**$_t^x$: The time it takes to read x blocks from RAM, where $x \in \{1, 2, 3\}$. $MemRD_t^2$ includes the time it takes to copy a fetched block from one BRAM to the other. $MemRD_t^3$ in addition to this, include the xor operation with a third fetched block into BRAM. Hence, the following relationship applies:

$$MemRD_t^2 \neq 2 \cdot MemRD_t^1$$

$$MemRD_t^3 \neq 3 \cdot MemRD_t^1$$

The formulas for estimating the variable state times are presented in Equation 4.2 - 4.5.

$$Col_t = 2p \cdot (H'_t + MemWR_t) \tag{4.2}$$

$$FirstPass_t = (m' - 2p) \cdot (MemRD_t^2 + Compress_t + MemWR_t) \tag{4.3}$$

$$NextPasses_t = m' \cdot (MemRD_t^3 + Compress_t + MemWR_t) \cdot (t-1) \tag{4.4}$$

$$FinalBlock_t = MemRD_t^1 \cdot p \tag{4.5}$$

5 Results

This chapter presents the acquired results of the project. They are split into two main sections, **Resource usage** and **Performance tests**. The former contains the basis for the answer to research question 1 and the latter for research question 2.

5.1 Resource Usage and Constraints

The design proved to be quite large with respect to the Spartan 6 resources. Although several efforts to reduce area usage were made after establishing a working implementation, in the end, it did not fit on the Spartan 6 as it could not route the signals within the available area. The resource usage can be seen in Table 5.1. A switch of target hardware from the Spartan 6 to the Virtex 6 was therefore needed in order to acquire some performance estimations, as described in the method chapter. In Table 5.2 the required amount of resources is presented together with the amount available on the Virtex 6. The maximum achievable clock frequency for the design was 46 MHz.

Resource	Utilized	Available	Ratio
Slice register	19037	54576	34%
Slice LUTs	22939	27288	84%
Fully used LUT-FF pairs	11145	30831	36%
Bonded IOBs	61	296	20%
Block RAM/FIFO	44	116	37%
BUFG/BUFGCTRLs	4	16	25%
DSP48E1s	20	58	25%

Table 5.1: FPGA resource usage for the cracker design on a Spartan 6.

Resource	Utilized	Available	Ratio
Slice register	22522	301440	7%
Slice LUTs	28934	150720	19%
Fully used LUT-FF pairs	12964	38492	33%
Bonded IOBs	82	600	13%
Block RAM/FIFO	44	416	10%
BUFG/BUFGCTRLs	5	32	15%
DSP48E1s	20	768	2%

Table 5.2: FPGA resource usage for the cracker design on a Virtex 6.

5.2 Performance Tests

In this chapter, the performance of the CPU and FPGA are presented. As no physical cracking system was produced, no real-time measurements could be acquired for FPGA. However, as the implementation could be synthesized for a Virtex 6, this was used to simulate the performance of the design instead. The simulations were run with a 46 Mhz system clock, the highest achievable clock frequency for the design in order to get an upper bound. The performance results are presented in two tables with accompanying graphs for each piece of hardware. In the first table, iteration t is fixed to 1 where the parallelism p and memory cost m vary. In the second table, p is fixed to 1 where t and m vary.

5.2.1 Virtex 6 Simulation Estimations

The performance of each step of the algorithm is presented below in simulated time and required clock cycles. The number of required clock cycles for the different actions was estimated from the simulation time divided by the clock frequency, rounded to the nearest integer.

$$\text{Clock Cycles} = [\text{Simulation Time} \cdot \text{Clock Frequency}]. \tag{5.1}$$

Action	Time (μs)	Clock cycles
$Init_t$	19.4772	896
$Output_t$	153.4485	7059
H'_t	528.429042	24308
$Compress_t$	12.7167	585
$MemWR_t$	1.7354	80
$MemRD^1_t$	8.8908	409
$MemRD^2_t$	17.6165	810
$MemRD^3_t$	28.2637	1300

Table 5.3: Time and clock cycles for an action of the implemented algorithm.

A variability of the memory access time during different states was noted, which can be viewed in Table 5.4.

$MemWR_t$ (μs)	$MemRD^2_t$ (μs)	$MemRD^3_t$ (μs)
1.717302	17.629518	28.21592
1.804254	17.629518	28.302876
1.84773	17.586042	28.259400
1.695564	17.629518	28.411566
1.717302	17.607780	28.128972
1.7564304	**17.6164752**	**28.2637476**

Table 5.4: RAM-related action times and respective averages in the last row.

5.2.2 FPGA Performance Results

The FPGA performance was estimated by the sum of time estimations of each state as described in Algorithm 4.1.

p \ *m(KiB)*	32768	65536	131072	262144	524288	1048576	2097152
1	1052	2103	4204	8408	16814	33628	67254
2	1053	2104	4205	8409	16815	33629	67255
4	1055	2106	4207	8411	16817	33631	67257
8	1059	2110	4212	8415	16821	33635	67261
16	1067	2118	4220	8423	16829	33643	67269
32	1083	2134	4236	8439	16846	33659	67285

Table 5.5: Simulated performance in *m*s for Virtex 6 where $t = 1$, p, and m vary.

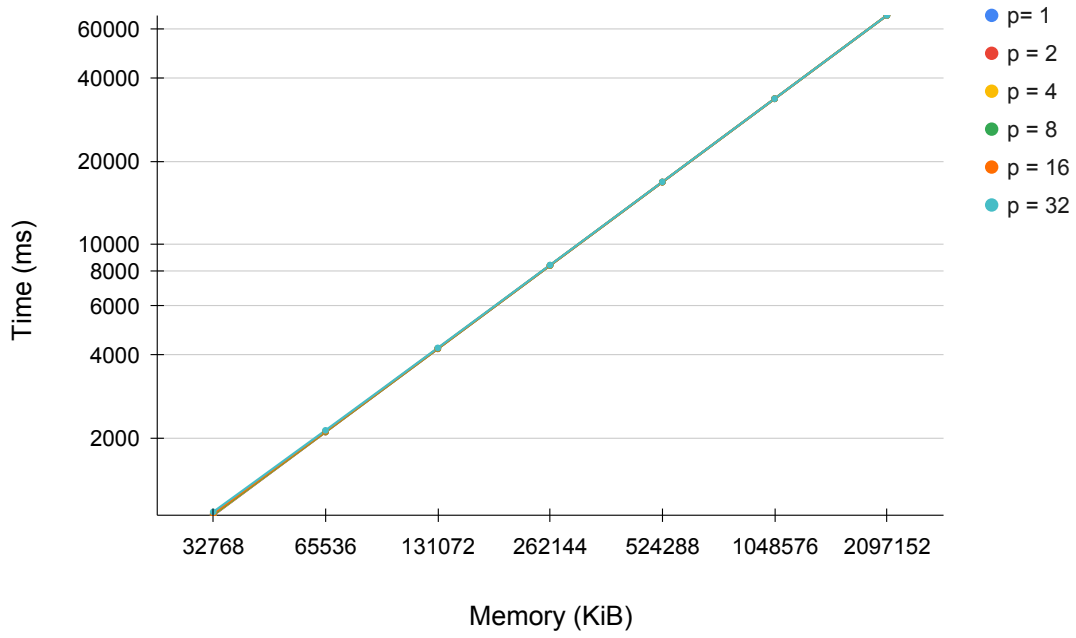

Figure 5.1: Plotted results from Table 5.5. **Note:** the y-axis is logarithmic.

t \ *m(KiB)*	32768	65536	131072	262144	524288	1048576	2097152
1	1052	2103	4204	8408	16814	33628	67254
2	2452	4902	9803	19605	39210	78418	156836
3	3851	7702	15402	30803	61605	123209	246417
4	5251	10501	21001	42001	84001	168000	335999
5	6651	13301	26600	53199	106396	212791	425581

Table 5.6: Simulated performance in *m*s for Virtex 6 where $p = 1$, t, and m vary.

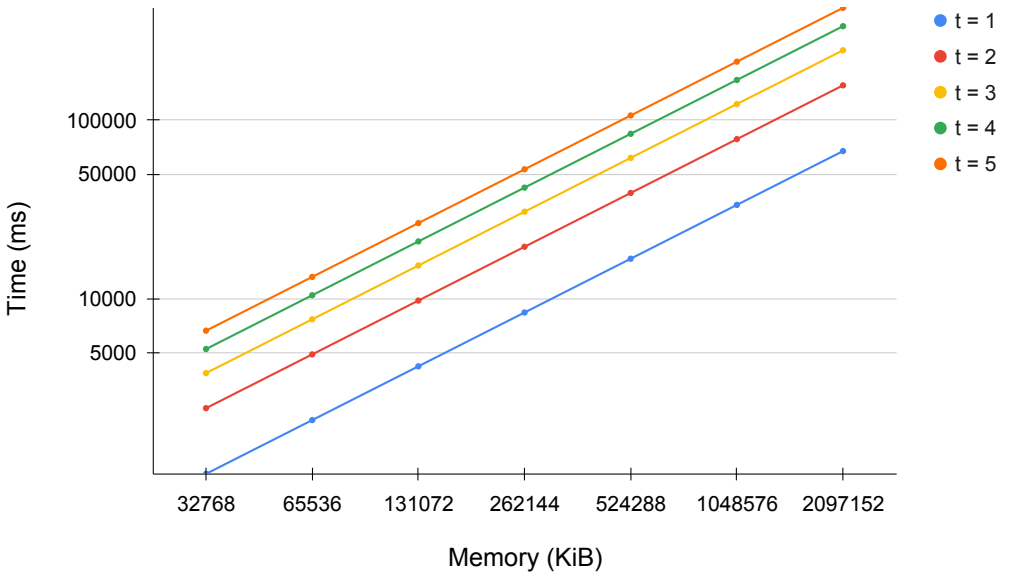

Figure 5.2: Plotted results from Table 5.6. **Note:** the y-axis is logarithmic.

5.2.3 CPU Performance Results

The CPU performance was acquired by running the benchmark test for the reference implementation Argon2 according to the altered method measuring wall time as mentioned in Section 4.10.1.

p \ *m(KiB)*	32768	65536	131072	262144	524288	1048576	2097152
1	58	101	182	350	690	1380	2796
2	66	109	174	273	490	911	1772
4	47	82	131	216	373	688	1326
8	39	64	101	176	306	556	1067
16	38	53	80	131	235	450	871
32	44	62	102	144	239	441	841

Table 5.7: CPU results in *m*s where $t = 1$, p and m vary.

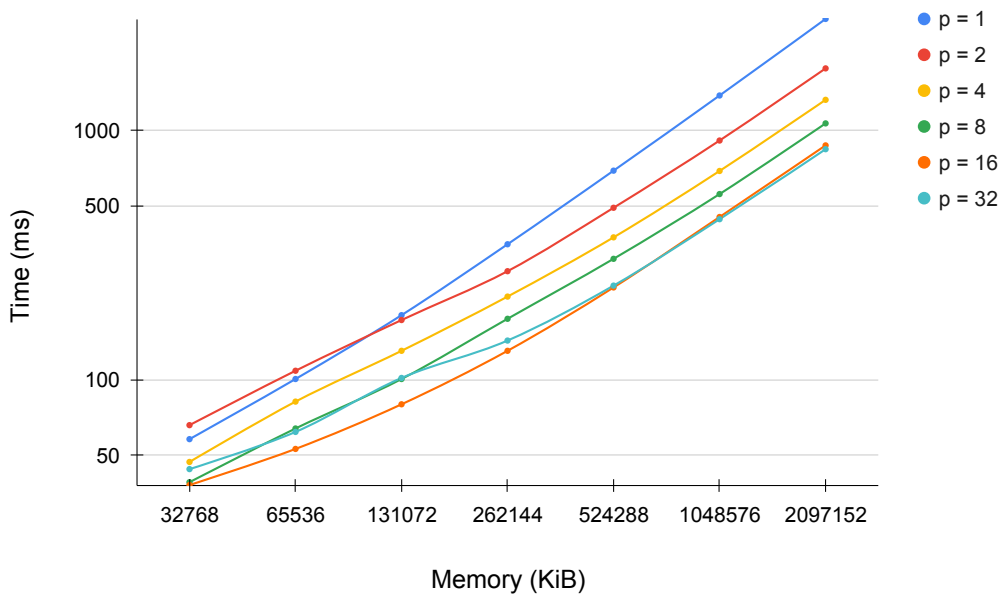

Figure 5.3: Plotted results from Table 5.7. **Note:** the y-axis is logarithmic.

t \ *m(KiB)*	32768	65536	131072	262144	524288	1048576	2097152
1	58	100	183	348	685	1377	2768
2	71	128	237	459	923	1863	3763
3	86	152	290	567	1160	2354	4761
4	98	178	344	690	1381	2896	5795
5	108	198	398	798	1619	3369	6833

Table 5.8: CPU results in *m*s where $p = 1$, t and m vary.

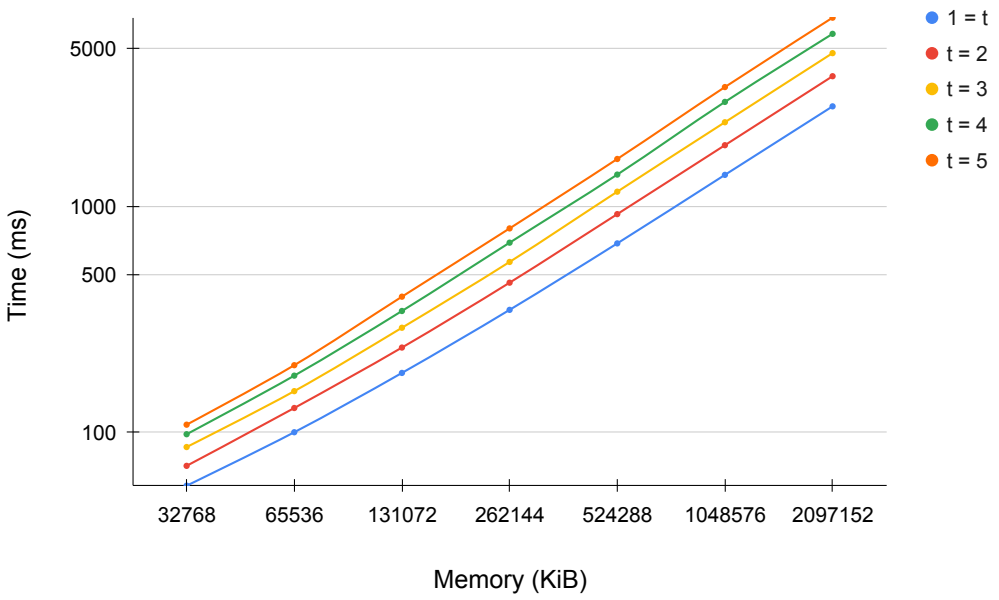

Figure 5.4: Plotted results from Table 5.8. **Note:** the y-axis is logarithmic.

5.2.4 Graph Comparisons

In order to see the difference in asymptotic performance as the memory parameter increases, the respective performance measurements for $t = 1$, $p = 1$, and $t = 1, p = 32$, are plotted together in Figure 5.5 and Figure 5.6.

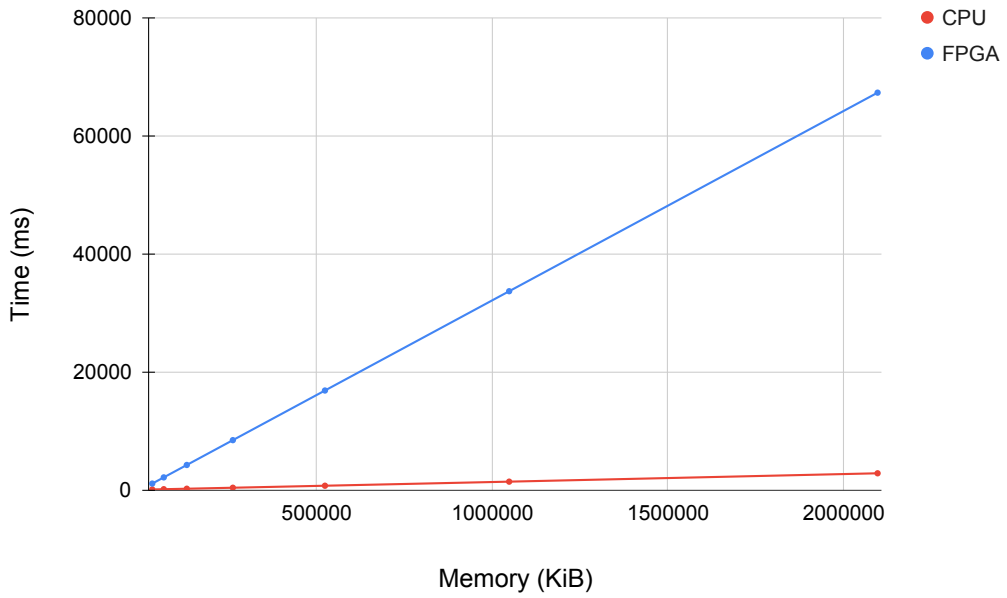

Figure 5.5: Comparison between FPGA and CPU with respect to memory cost when p = 1 and t = 1.

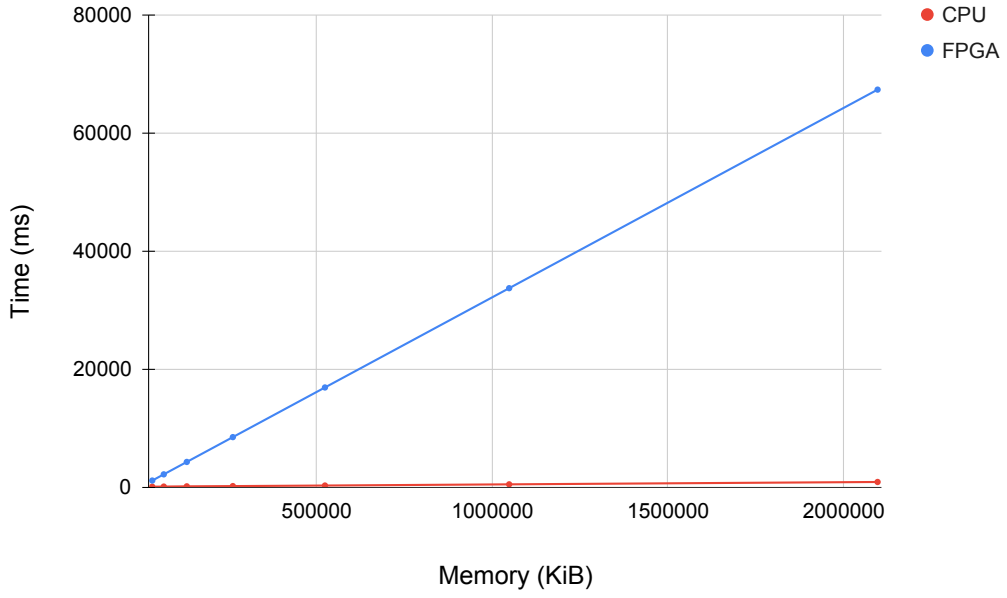

Figure 5.6: Comparison between FPGA and CPU with respect to memory cost when p = 32 and t = 1.

6 Discussion

This chapter presents an analysis of the acquired results based on the information presented in the theory chapter. The discussion includes both the direct results such as the estimated performance relative to a modern CPU, and also the chosen method and project delimitations. Alternative ways in which the experiment could have been executed as well as proposed continuations on the topic in general are also presented. The work raises several questions on how different approaches would compare, such as a combined design with a GPU where only a part of the algorithm executes on the FPGA. The ethical implications of this work are also discussed briefly at the end.

6.1 Results

6.1.1 Resource Usage and Constraints

The resulting design could not be fitted on the Spartan 6. As mentioned previously, several unsuccessful efforts were made in order to make the design use less area. Compared to the scrypt implementation by Duong Le et. al, the Argon2 implementation for the Virtex 6 requires roughly three times the amount of LUTs and just over double the amount of FFs for the single PE implementation [35]. This implementation uses 57012 clock cycles per hash compared to the ~243009 clock cycles for the Argon2 implementation with $p = 1$, $t = 1$, and $m = 128$. The former could be synthetized with a maximum achievable clock frequency of 156 MHz while the latter reached 46 MHz.

6.1.2 Test Results

Constant t

As can be seen in Table 5.5 and Figure 5.1, there is barely any difference when t stays constant; all scenarios have the same linear growth. This is to be expected since the implementation works mostly sequentially.

There is also a slight time increase as p grows. This can be explained by looking at the two ways blocks can be calculated when $t = 1$ in the order of operations of Argon2 in Section 3.5.4. In the first method, the two-column calculations using H' in steps 3 and 4 are used. In the second method, the compress function G in step 5 is used. By increasing p, the number of

blocks needed to be calculated by the former method increase and the latter method decrease. Looking at Table 5.3, the time per block is $H'_t + MemWR_t$ (24388 clock cycles) in the former case and $MemRD^2_t + Compress_t + MemWR_t$ (1475 clock cycles) in the latter.

The $FinalBlock_t$ calculation time, as described by Equation 4.5, is also increased by raising p. In contrast, the CPU results, presented in Figure 5.4, show a decrease in execution time owing to some form of parallel computation of the blocks. The total execution time is a fraction of the FPGA implementation.

Constant p

There is quite a large difference between $t = 1$ and $t = 5$; roughly a 6 times slowdown between those two configurations, as seen in Figure 5.2. This slowdown seems to be linear, which is as expected. For $m = 32768$, the slowdown is around 1.4 seconds and around 90 seconds for $m = 2097152$ for every increment of t. This pattern is to be expected since t in effect acts as a multiple of step 5 as described in the Argon2 order of operations in Section 3.5.4. This fifth step also grows in execution time as the amount of memory blocks needed to be processed increases, indicated by parameter m. The total execution time of the CPU is a fraction of the FPGA implementation.

6.2 Method

As no complete VHDL implementation of Argon2 could be found during the thesis work, the most straightforward way to gauge its potential was to design an implementation and subsequently evaluate it as was done in this thesis. This approach proved to result in a lot of difficulties however. These problems are discussed in the following chapters along with possible improvements.

As the chosen approach focused primarily on functionality, the optimization attempts with regard to the area were initiated as soon as correct hashes were produced. In other words, the resulting implementation is a product of a fast but theoretically unattainable solution being split up and sequentialized to the point of being able to fit on the target platform. This process was very crude in hindsight. An alternative method would have been to start at the other end with a CPU performance measurement complemented by a theoretical analysis of the individual Argon2 parts. This way, a more methodical design process could have been executed with clear goals of performance as well as pre-calculated areal constraints. A possible alternative outcome of this approach could have been the realization that the compression stage, which carries the most impact on the asymptotic performance, could not reach the desired speeds much earlier which would be sufficient to answer the second research question for the relevant memory parameters (i.e. those recommended for password storage).

6.2.1 Targeted FPGA

The entire *Cracker* structure was originally intended to be implemented on the Spartan 6 SP605 board. Without proper knowledge of the space required for the final design, the design included variability of all Argon2 parameters. While some of these later on were deemed too difficult and time-consuming to add, such as parallel lane computing, it became increasingly clear that space was becoming a significant problem as the project progressed. Many areal optimization attempts such as fixing the parameter sizes were made to accommodate this and in the end, it reached about 90% usage of slice LUTs. Unfortunately, ISE was still unable to place and route the design within the constraints of the Spartan 6.

By switching the target to a Virtex 6 ML605 board this was not a problem anymore as shown in Table 5.2 where there is potential to fit much more than on a Spartan 6. A lot of the efforts to minimize the area requirements in order to fit the design on a Spartan 6 depended

on sequentialization. An improvement in performance could possibly be observed on the Virtex 6 if those same decisions were reversed. However, as bigger combinatorial circuits entail higher propagation delays [37], it is impossible to predict if these potential performance enhancements are even synthesizable for the same clock frequency without actually performing the redesign.

6.2.2 Reference CPU

In order to get some indication of how well the performance of the FPGA turned out to be, it needed to be compared to some other hardware. The hardware was chosen mostly because of its proximity to the available FPGA board. It would have been interesting to also compare the performance with a GPU solution. However, due to time constraints and a lack of verified GPU implementations it was decided that it would be compared to the CPU only. It is however difficult to choose a model that provides a fair comparison. The generation, purchase price, and power usage per hash are all factors that need to be considered for a thorough comparison of the feasibility of hashing Argon2 on CPUs vs FPGAs.

6.2.3 Hashcat

Integrating FPGA support into Hashcat provided information about the feasibility of utilizing the FPGA-Argon2 implementation in the real world. If the Spartan 6 design was synthesizeable it would have set an upper limit on how much data could be funneled and hence how many hashes could be created within a given time constraint.

6.2.4 Design Optimizations

There are a number of optimizations that could be done to decrease computation time without design overhauls. What follows are general design optimizations on a per-component basis.

Compression Function

The compression function is the most repeated routine of the entire Argon2 algorithm, especially for high values of memory usage. In order for the program to run at a desirable pace, it is crucial for this particular component to be as effective as possible. As it stands, the implementation presented in this report takes 585 clock cycles to complete and only uses pipelining to a small extent when writing to the block RAM components. The design is in other words very crude and can be optimized.

When analyzing the results from the CPU, it can be estimated that the reference implementation of the compression function takes $0,247$us to complete including reading 3 blocks and writing one. This means that with the highest possible frequency for this design (46MHz), the whole compression stage must take less than

$$\frac{0,247e^{-6}}{\frac{1}{46e^{6}}} = 0,247 \cdot 46 \approx 11$$

clock cycles for it to be beneficial. Due to the interfacing of the RAM, it takes approximately 880 clock cycles to read the two blocks necessary for `Compress` to compute. This means that even if the compression component would be optimized down to 11 clock cycles per run, the interface would have to be close to one hundred times faster before there is any point in trying to achieve this.

Memory Interface

As seen in Figure 4.13, it takes a total of 3 clock cycles to write the 128-bit chunks to memory, and 24 clock cycles to write the 128 bytes. However, Figure 4.12 makes it clear that it is possible to write these 128-bit chunks back-to-back in a total of 8 clock cycles, which is 3 times faster. There is although no guarantee that this would be possible for the entire duration of the block since *app_rdy* still needs to be asserted. But for the sake of argument, the whole operation $MemWR_t$ would get reduced from 80 clock cycles to 27.

The read operation does however not have any significant optimizations that could be done since it is the IP core consuming the most amount of time retrieving the data.

General Design Optimizations

In Table 5.2 we can see that there is a significant amount of space left to be utilized. Therefore, there is a possibility that multiple instances of `Compress` and possibly the `H'` component could fit, which would enable parallel execution on several lanes. Fitting three more `Compress` and `H'` is not inconceivable given the current 19% Slice LUT usage on the Virtex 6. Any more than that doesn't seem feasible, but for the sake of argument let's assume p number of `Compress` and `H'` components. Equations 4.2 - 4.4 can then be rewritten as:

$$Col_t = (H'_t + MemWR_t \cdot p) + (H'_t + MemWR_t) \tag{6.1}$$

$$FirstPass_t = (m' - 2p)/p \cdot (MemRD_t^2 \cdot p + Compress_t + MemWR_t) \tag{6.2}$$

$$NextPasses_t = m'/p \cdot (MemRD_t^3 \cdot p + Compress_t + MemWR_t) \cdot (t-1) \tag{6.3}$$

Since the current design does not allow for multiple reads and writes at the same time, these need to be multiplied by p. In the case of Equation 6.1, once the first column has been computed with sequential RAM writes, the computation of the second column will already be out of sync with the memory writes, thus no overlapping writes will occur. There should not be any overlap in the write operation occurring in Equation 4.3 and Equation 4.4. This is because there would be at least one $MemRD_t^2$ delay between processes, which is bigger than a $MemWR_t$ delay, hence the p factor only for $MemRD_t^2$ in Equation 6.2 and Equation 6.3. Assuming the same clock frequency of 46 MHz, the results of these changes optimized for $p = 4$ with the memory optimization are presented in Table 6.1.

Hardware \ *m(KiB)*	32768	65536	131072	262144	524288	1048576	2097152
FPGA (unoptimized)	1055	2106	4207	8411	16817	33631	67257
FPGA (optimized)	687	1374	2746	5491	10981	21961	43921
CPU	47	82	131	216	373	688	1326

Table 6.1: Results in *m*s from an optimized FPGA implementation when $p = 4$ and $t = 1$

With this optimization there is a decrease in computation time of roughly 65%, which is still far off matching the performance of the CPU. By looking at Table 6.1 and calculating the quotients between the CPU and the FPGA, it is apparent that in the most beneficial case for the FPGA (32768 KiB), it is still slower by roughly a factor of 15 and 22 for the optimized and unoptimized versions respectively. In the worst-case scenario (2GiB), it is slower by roughly a factor of 33 and 51. In other words, a Virtex 6 that would be able to synthesize this design

with a clock frequency of at least 690 MHz could compete with this CPU when the memory parameter is set to 32768 KiB or less. For the unoptimized version, a clock frequency of just over one GHz would be needed.

6.2.5 Performance Tests

It would have been preferable to have run Argon2 on actual FPGA hardware since Hashcat has a built-in cracking timer. It would have also been possible to take into account the FPGA-Hashcat communication through UART.

6.3 Future Work

As the implementation presented in this project is only a proof of concept, there are most certainly several improvements to be made. Using a modern FPGA could be among those improvements.

In order to properly utilize it, it would also have to be connected to a password generator. A good choice for reasons previously presented in the theory chapter would be Hashcat. The initial plan for this project included a hashcat backend solution that would allow it to run the hashing sequence on the FPGA, but it was never fully utilized due to the solution failing to synthesize for the Spartan 6. An important aspect of this is how the connection between the FPGA and the host computer, as the communication speed requirements for that would depend on the achieved performance of the hashing. A full analysis of these requirements for the chosen design would have to be made in that case. A few ideas born from the results of this design are explored in the following sections.

6.3.1 Parallel Compression Components

The algorithmic flow of the solution produced in this project is strictly sequential and as the results show, this is a rather slow solution. The first step to properly use the strength of the FPGA, parallelism, this would of course have to be modified. Even in the case of $p = 1$, parallelism could be utilized to calculate multiple passwords at the same time.

When looking at the size results from Table 5.2, it seems that there is room for more instantiations of the compression component. However, due to routing and the need for more overhead mechanisms, it is difficult to say exactly how many. Based on the numbers though, a rough estimation would be that two more compression components could be instantiated. This would result in triple the performance of the compression step.

However, an important point to make is that in order for these components to run in parallel, they need to share the available memory. This means that this optimization technique only has a maximal effect if the memory parameter is equal to or less than a third of the available memory. In this specific case with the ML605 board, the highest value the memory parameter could have with the integrated memory is

$$m \leqslant \lfloor \frac{512MB}{3} \rfloor = 170MB.$$

If the memory parameter exceeds this value, the solution could still run twice as fast as long as the memory parameter does not exceed half of the available memory. If that is the case, the added overhead would likely slow the solution down. This means that this approach should only be used when the memory parameter is big enough. Of course, if the parallelism parameter is equal to or higher than the amount of instantiated compression components, then these can operate on the same memory and provide an upgrade in performance regardless of the memory cost.

6.3.2 Stand-alone Compression

As stated in Section 4, the entire algorithmic flow was implemented on the FPGA in this study. As there are several different variable parameters and signals with a size of 1KiB, this solution takes up a lot of space on the FPGA. As the results show, smaller FPGAs like the Spartan-6 can not fit the design at all. Although the Virtex 6 did, less room for performance optimizations was given since these tend to take up more space on the FPGA.

An idea to combat this would be to use a mixed-hardware solution that only performs a part of the algorithm on the FPGA. Note that while the compression component is used in the example presented owing to its asymptotic impact on the performance, this would only be beneficial if an optimized version capable of outperforming the CPU was reached in conjunction with a fast enough RAM interface as previously described.

The difference between the current and the proposed design can be viewed in Figure 6.1 and Figure 6.2. While the compression is a part of the Argon2 algorithm, in these figures it is illustrated as a separate component to achieve a clear comparison.

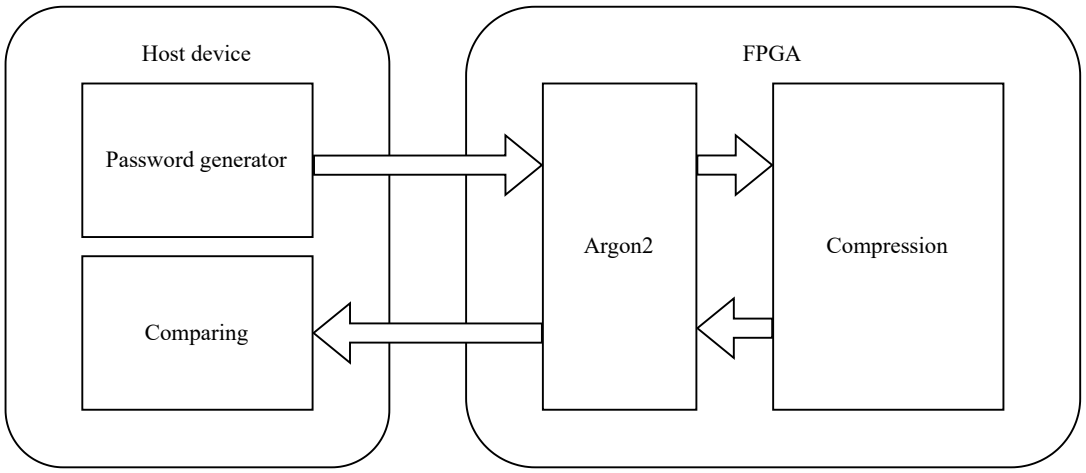

Figure 6.1: Current design.

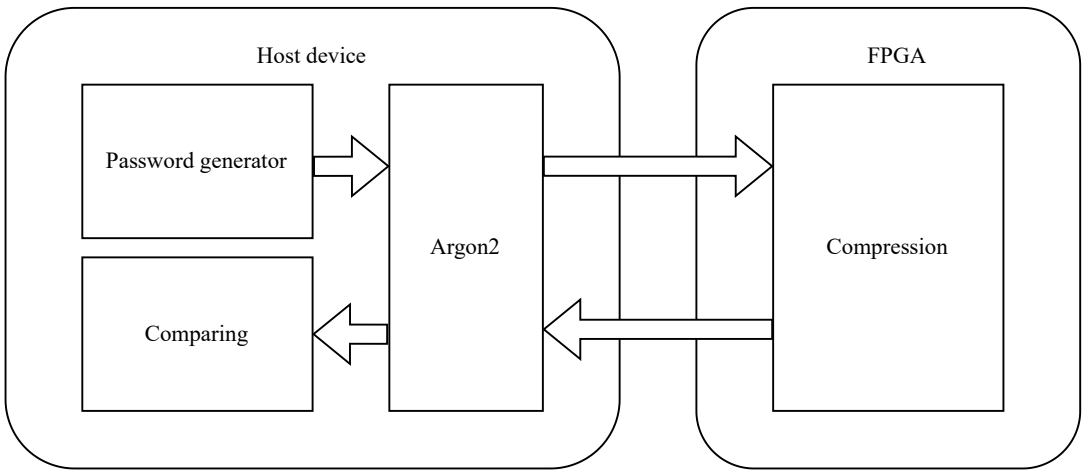

Figure 6.2: Alternative design.

6.4 The Work In A Wider Context

As password hashing is central to modern solutions of integrity standards, it is crucial for the developers of these solutions to understand which flaws exist in the many available KDFs or hash functions to choose from. As Argon2 is presented as a hardware-resistant hashing function, it is important that it continues to be scrutinized as hardware technology progresses.

Another perhaps small but important aspect is the power consumption of large-scale computing clusters, as FPGAs can be configured to a significant advantage when it comes to power consumption apparent in the report by Duong Le et. al. [25]. As the incentive to crack hashes seemingly will not disappear in the near future, people will naturally be seeking the fastest ways to do so. Would the FPGA prove to be superior performance-wise even in some cases, large clusters comprised of hardware with a higher power consumption per hash could get replaced with FPGA clusters. In a time where every bit of power saving counts, this would certainly have a positive impact, no matter how small. As a reference, it can be compared with the increasingly large carbon footprint of Bitcoin mining [9]. While specifically Argon2 currently lacks the obvious monetary incentive of Bitcoin, it still shows the importance of energy-effective solutions in cryptography where computing power is the most important resource.

Although this project failed to produce an FPGA implementation that managed to achieve better performance in terms of speed, or even a working system, it could be used as a starting point for continued work on the problem. It should also act as a deterrent for other projects about evaluating Argon2 in a similar approach without a different and well-thought-out design.

7 Conclusion

The purpose of this project was to create a basic FPGA implementation of Argon2 to be used in conjunction with the password generator Hashcat and evaluate the performance. Although the ultimate goal of producing a complete cracking system failed due to limited resources and time, enough work has been done to answer the research questions below.

7.1 1. How much area would be required for a basic implementation of Argon2 on a Spartan 6 FPGA?

While a conclusion on how much resources Argon2 would need in the general case is impossible to determine from this study, it is obvious that the chosen design route does not suit the Spartan 6 well. The design was heavily optimized in terms of area after the simulation could produce correct results, but these optimizations proved to be insufficient as the design was not able to be synthesized for the Spartan 6 in the end. While there may be a chance that further optimizations in terms of area are possible, it is quite clear that the Spartan 6 is not fit for this design approach. With this implementation, an upper bound for the area has been presented for the Virtex 6.

7.2 What performance could be expected of a basic implementation of Argon2 on a Spartan 6 FPGA?

The estimated performance of the solution simulated for the Virtex 6 was approximately 24 times slower than the reference CPU with a parameter configuration that best suited the FPGA. This makes it very clear that even in the unlikely case of a successfully optimized version of this design targeted for the Spartan 6 being produced, it would be incredibly slow and subsequently not very useful. There is some potential for alternative composite designs that utilize other types of hardware where the Spartan 6 could still be of use as a single component, but whether or not those propositions are practical or even possible remains to be seen.

7.3 Closing Remarks

If another attempt to implement the algorithm on an FPGA were to be made, heavy design alterations such as those presented in the discussion chapter would have to be considered; one of which being only having the compression component on the FPGA. This in combination with a larger FPGA more in line with modern standards would probably be the best strategy going forward.

Bibliography

[1] *A short history of the computer password*. URL: https://www.welivesecurity.com/2017/05/04/short-history-computer-password/ (visited on 04/18/2023).

[2] *Announcing the PCIe® 7.0 Specification: Doubling the Data Rate to 128 GT/s for the Next Generation of Computing*. URL: https://pcisig.com/blog/announcing-pcie%C2%AE-70-specification-doubling-data-rate-128-gts-next-generation-computing (visited on 03/05/2024).

[3] *Argon2 github*. URL: https://github.com/christian-krieg/argon2 (visited on 07/13/2022).

[4] Sumaia Atiwa, Yunus Dawji, Ahmed Refaey, and Sebastian Magierowski. "Accelerated Hardware Implementation of BLAKE2 Cryptographic Hash for Blockchain". In: *2020 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. 2020, pp. 1–6. DOI: 10.1109/CCECE47787.2020.9255709.

[5] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications". In: *2016 IEEE European Symposium on Security and Privacy (EuroSP)*. 2016, pp. 292–302. DOI: 10.1109/EuroSP.2016.31.

[6] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*. RFC 9106. Sept. 2021. DOI: 10.17487/RFC9106. URL: https://www.rfc-editor.org/info/rfc9106.

[7] *BLAKE2 — fast secure hashing*. URL: https://www.BLAKE2.net/#sp (visited on 02/20/2024).

[8] *BLAKE2 github*. URL: https://github.com/christian-krieg/BLAKE2 (visited on 03/20/2024).

[9] Mikhail Bondarev. "Energy consumption of bitcoin mining". In: *International Journal of Energy Economics and Policy* (2020).

[10] *Computational complexity*. URL: https://en.wikipedia.org/wiki/Computational_complexity (visited on 02/26/2024).

[11] *Cryptography Hash functions*. URL: https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.html (visited on 06/05/2022).

[12] *Definition of PCI Express*. URL: https://www.pcmag.com/encyclopedia/term/pci-express (visited on 01/17/2024).

[13] *Field-programmable gate array*. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array (visited on 05/31/2024).

[14] *Getting Started with the Xilinx Spartan-6 FPGA SP605 Evaluation Kit*. URL: https://docs.xilinx.com/v/u/en-US/ug525 (visited on 03/13/2024).

[15] *Getting Started with the Xilinx Virtex-6 FPGA ML605 Evaluation Kit*. URL: https://docs.rs-online.com/7603/0900766b80f2e0f0.pdf (visited on 03/27/2024).

[16] *Hashcat Plugin Development Guide*. URL: https://github.com/hashcat/hashcat/blob/master/docs/hashcat-plugin-development-guide.md (visited on 11/07/2022).

[17] *Hashcat Wiki*. URL: https://hashcat.net/wiki/ (visited on 07/13/2022).

[18] *HEKS: A family of Key Stretching Algorithms*. URL: https://theworld.com/~reinhold/HEKSproposal.html (visited on 03/05/2024).

[19] *https://www.cs.utexas.edu/ dwu4/courses/sp22/static/projects/Kornerup.pdf*. URL: https://www.cs.utexas.edu/~dwu4/courses/sp22/static/projects/Kornerup.pdf (visited on 03/05/2024).

[20] *ISE Design Suite*. URL: https://www.xilinx.com/products/design-tools/ise-design-suite.html (visited on 03/13/2024).

[21] *ISE Simulator (ISim)*. URL: https://www.xilinx.com/products/design-tools/isim.html (visited on 03/13/2024).

[22] *Key derivation function*. URL: https://en.wikipedia.org/wiki/Key_derivation_function (visited on 05/16/2024).

[23] *Key derivation functions*. URL: https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/ (visited on 05/16/2024).

[24] *LUKS2*. URL: https://gitlab.com/cryptsetup/LUKS2-docs/blob/main/luks2_doc_wip.pdf (visited on 06/04/2024).

[25] *MRSA: A High-Efficiency Multi ROMix Scrypt Accelerator for Cryptocurrency Mining and Data Security*. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9631222 (visited on 03/27/2024).

[26] *opencores*. URL: https://opencores.org/projects/simple_uart_for_fpga (visited on 08/10/2022).

[27] *OWASP storage*. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (visited on 04/14/2024).

[28] *Password Storage Cheat Sheet*. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (visited on 06/05/2022).

[29] *PCI-SIG® Announces Upcoming PCI Express® 6.0 Specification to Reach 64 GT/s*. URL: https://www.businesswire.com/news/home/20190618005945/en/PCI-SIG%C2%AE-Announces-Upcoming-PCI-Express%C2%AE-6.0-Specification (visited on 01/17/2024).

[30] *PHC winner Argon2*. URL: https://github.com/P-H-C/phc-winner-argon2 (visited on 06/08/2022).

[31] *PKCS 5: Password-Based Cryptography Specification Version 2.0*. URL: https://en.wikipedia.org/wiki/Key_derivation_function (visited on 05/16/2024).

[32] *Recommended Standard 232*. URL: https://circuitdigest.com/article/rs232-serial-communication-protocol-basics-specifications (visited on 03/03/2024).

[33] *RS-485 Reference Guide*. URL: https://web.archive.org/web/20180517101401/http://www.ti.com/lit/sg/slyt484a/slyt484a.pdf (visited on 03/03/2024).

[34] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. RFC 7693. Nov. 2015. DOI: 10.17487/RFC7693. URL: https://www.rfc-editor.org/info/rfc7693.

[35] *Scrypt*. URL: https://www.tarsnap.com/scrypt/scrypt.pdf (visited on 01/08/2024).

[36] *UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter*. URL: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html (visited on 03/03/2024).

[37] *Understanding Timing Considerations for FPGA VIs*. URL: https://www.ni.com/docs/en-US/bundle/labview-fpga-module/page/understanding-timing-considerations-for-fpga-vis-fpga-module.html (visited on 05/26/2024).

[38] *Universal asynchronous receiver-transmitter*. URL: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter (visited on 03/03/2024).

[39] *Versal Adaptive SoC Clocking Resources Architecture Manual (AM003)*. URL: https://docs.amd.com/r/en-US/am003-versal-clocking-resources/BUFGCTRL (visited on 04/02/2024).

[40] *VHDL implementation of Argon2 memory-hard function for password hashing and other applications*. URL: https://github.com/christian-krieg/argon2 (visited on 03/13/2024).

[41] *Virtex-6 Memory Resource Guide*. URL: https://docs.xilinx.com/v/u/en-US/ug363 (visited on 07/13/2022).

[42] *Virtex-6 Memory solutions*. URL: https://docs.xilinx.com/v/u/en-US/ug406 (visited on 07/13/2022).

[43] *Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide (UG953)*. URL: https://docs.amd.com/r/en-US/ug953-vivado-7series-libraries/BUFGCE_1 (visited on 04/02/2024).

A CPU Benchmark - Variable: Parallelism

```
--- Memory: 32768 KiB ---
### Parallellism: 1
Hash: c636fc371f651de1fe5de8aead7d9a10c63788c50a87f949b4eef969c3d1b743
0.058
0.058
0.059
0.058
0.058
0.045
0.059
0.060
0.044
0.046
Median: 0.058
Average: 0.054
### Parallellism: 2
Hash: 7e9c3d2a7363042b810e0b47a4c99fa9b0f86396d2959e34e56138bb2bbd69f2
0.065
0.066
0.066
0.065
0.066
0.066
0.066
0.067
0.068
0.067
Median: 0.066
Average: 0.066
### Parallellism: 4
Hash: 8ea7362f321c782a0b7cea5e1a908db5fa5f1cd27c53751ae88add8940b8c58b
0.046
```

```
0.046
0.047
0.047
0.048
0.048
0.048
0.048
0.046
0.047
Median: 0.047
Average: 0.047
### Parallellism: 8
Hash: 7cc03322219834f02f31392521d8d5d3c2686324a32bf2802be1867bd636970b
0.039
0.040
0.039
0.039
0.038
0.039
0.038
0.039
0.038
0.036
Median: 0.039
Average: 0.038
### Parallellism: 16
Hash: 8a9ca54b24bab7a1b487fe8899e08cd9e23a76fa5f8064c6fdd8063ff6fdafb9
0.040
0.030
0.033
0.035
0.038
0.038
0.041
0.039
0.038
0.036
Median: 0.038
Average: 0.037
### Parallellism: 32
Hash: aed14d70241956e429a193afc7eb4fd5c62cc4a41869f551eb2cc5b5dad570c9
0.045
0.046
0.044
0.046
0.046
0.044
0.045
0.038
0.041
0.044
Median: 0.044
Average: 0.044
```

```
--- Memory: 65536 KiB ---
### Parallellism: 1
Hash: bea906ecd5e6981ae9532155cda8d22b2935d21f4a8c20160a47a17e964bfedc
0.089
0.099
0.101
0.102
0.100
0.101
0.087
0.100
0.103
0.101
Median: 0.101
Average: 0.098
### Parallellism: 2
Hash: 8bd7e037ba7f1efa5b42e77fa3fc66825d2775a3ce32c5dd9efd255aa9571e52
0.114
0.110
0.107
0.109
0.107
0.109
0.110
0.107
0.108
0.112
Median: 0.109
Average: 0.109
### Parallellism: 4
Hash: fb441b560854c2f65cb6d5d5fdcdba65db3c2f6bc4e8b6feb46cd7605920cfe1
0.083
0.082
0.081
0.083
0.081
0.080
0.081
0.082
0.081
0.083
Median: 0.082
Average: 0.082
### Parallellism: 8
Hash: cf493827c85f69f3ec0ed7d4802ce7a93c8c9d2428cd9411ab3bda8aa12e6cb1
0.064
0.065
0.064
0.065
0.062
0.065
0.063
0.061
```

```
0.063
0.065
Median: 0.064
Average: 0.064
### Parallellism: 16
Hash: bc79359fe3acc76ea58c48eac5f580ac74df6f96dbf4be3c313ee47962b9e635
0.059
0.057
0.058
0.053
0.051
0.048
0.053
0.050
0.053
0.058
Median: 0.053
Average: 0.054
### Parallellism: 32
Hash: 557afe65316563dc9369cb960049d4c5fc9b1cd29a4b9c3f5f8a6fb54e504140
0.068
0.067
0.057
0.060
0.060
0.063
0.062
0.062
0.061
0.062
Median: 0.062
Average: 0.062
--- Memory: 131072 KiB ---
### Parallellism: 1
Hash: d0678c00b97f8eb9537ee4feea5f5877675cb3597c13348535a6bff64184e73e
0.185
0.182
0.186
0.171
0.186
0.181
0.182
0.188
0.183
0.181
Median: 0.182
Average: 0.182
### Parallellism: 2
Hash: 268a11142b69b8925777df6b6d588d9cc50ea1403cf1f451e20dd2a1fce70512
0.176
0.177
0.182
0.178
```

```
0.168
0.170
0.172
0.180
0.164
0.171
Median: 0.174
Average: 0.174
### Parallellism: 4
Hash: 8ae1e86b6b3f0a034f30bf237fc51d8aff549f6c20c7bc42a589ce659e133259
0.134
0.130
0.131
0.131
0.132
0.128
0.128
0.129
0.131
0.127
Median: 0.131
Average: 0.130
### Parallellism: 8
Hash: ba555f82d253ffeca5cb4e83097aedff48d1033ae614a39355d9820b22615116
0.101
0.103
0.101
0.100
0.101
0.103
0.099
0.100
0.099
0.099
Median: 0.101
Average: 0.101
### Parallellism: 16
Hash: 341930fc081c26dc03a85b7d3e1cd769b949aae9604492e8b376a96f3c357261
0.081
0.079
0.077
0.080
0.076
0.080
0.083
0.080
0.080
0.078
Median: 0.080
Average: 0.079
### Parallellism: 32
Hash: 5449707f8862844489c96b745a3963a36623fdacb49d92caf1dacb91394eb89f
0.100
```

```
0.103
0.110
0.110
0.101
0.094
0.103
0.101
0.106
0.086
Median: 0.102
Average: 0.101
--- Memory: 262144 KiB ---
### Parallellism: 1
Hash: 81dd8ee3366bc391277e264d193fec0fcf0a5a73802e1256a2174b67e983cf3c
0.348
0.350
0.345
0.355
0.333
0.353
0.350
0.353
0.349
0.350
Median: 0.350
Average: 0.349
### Parallellism: 2
Hash: 1ff2ebb6d3931cfe7dc1bc51a1554aa7e6be37844bfeeda8951e968a272ea21d
0.274
0.274
0.270
0.272
0.277
0.270
0.268
0.274
0.274
0.271
Median: 0.273
Average: 0.272
### Parallellism: 4
Hash: bf96eb2c3f4cf299ef5e3417b150a0c5b1f8383c5da41051d66b9698c5d76a82
0.216
0.211
0.214
0.216
0.219
0.214
0.216
0.218
0.217
0.217
Median: 0.216
```

```
Average: 0.216
### Parallellism: 8
Hash: 223e2cd6f2e6b8691ccbd0d4acae923bf52badad485830888cd337762900f877
0.176
0.181
0.173
0.176
0.175
0.173
0.177
0.176
0.177
0.175
Median: 0.176
Average: 0.176
### Parallellism: 16
Hash: 9aa2f16ab4945ee9fd82eda5a38f568ba5795166daac0fd4fda92f9085a64ad7
0.130
0.135
0.130
0.131
0.132
0.129
0.132
0.130
0.132
0.131
Median: 0.131
Average: 0.131
### Parallellism: 32
Hash: 043a06e6bc495872542b67cad40520e3c2fe15596dc18b35fa58db908c43cb8b
0.145
0.148
0.142
0.145
0.144
0.144
0.147
0.144
0.136
0.140
Median: 0.144
Average: 0.144
--- Memory: 524288 KiB ---
### Parallellism: 1
Hash: f22a6090ee805d7a3dc01875a8c4113193ed8e57739c7e92dcbff5e154b542e9
0.693
0.693
0.690
0.675
0.687
0.692
0.691
```

```
0.692
0.684
0.677
Median: 0.690
Average: 0.687
### Parallellism: 2
Hash: 13b710ecbfb948cbb7703567c32fcf6b68bdc7e77b6d5ab7b3e2ef0fb6f12592
0.495
0.490
0.490
0.489
0.484
0.491
0.489
0.491
0.503
0.490
Median: 0.490
Average: 0.491
### Parallellism: 4
Hash: 97126422c05955a0766b3f4f7d0f6444f062f8118c6b3cc608e9dba528e6daeb
0.373
0.377
0.371
0.373
0.375
0.374
0.364
0.371
0.376
0.371
Median: 0.373
Average: 0.372
### Parallellism: 8
Hash: 7db1de3350aee10d7fcceb63af934552f8edcb3022670bc5f6151175aff6879d
0.304
0.305
0.305
0.307
0.310
0.304
0.307
0.307
0.308
0.305
Median: 0.306
Average: 0.306
### Parallellism: 16
Hash: f820d4e32ef43908a1ebbe03f51dc0a515574818a2159cac55c4f3ba36203dfd
0.233
0.231
0.234
0.237
```

```
0.249
0.246
0.235
0.234
0.243
0.235
Median: 0.235
Average: 0.238
### Parallellism: 32
Hash: 0f034e43449caecbe1b6bd3201cea05f250471a886e02def5849bb99e9856a67
0.240
0.237
0.239
0.238
0.240
0.240
0.236
0.242
0.238
0.239
Median: 0.239
Average: 0.239
--- Memory: 1048576 KiB ---
### Parallellism: 1
Hash: eca700db0e881d738166019509f73d5e056d8d7b090020bf5977333254c0ea02
1.379
1.365
1.383
1.381
1.373
1.381
1.376
1.375
1.383
1.389
Median: 1.380
Average: 1.379
### Parallellism: 2
Hash: 774fc354b3aff4c03c2b3d839e3d364201d6a6ed33a18fedcf54063c9235309b
0.911
0.914
0.910
0.916
0.913
0.893
0.912
0.907
0.905
0.916
Median: 0.911
Average: 0.910
### Parallellism: 4
Hash: a6d85a7dfb89888b531e3c6dedd393e22c3e199f4876eaf3956665b2ab20c7dc
```

```
0.683
0.688
0.688
0.691
0.682
0.688
0.690
0.687
0.687
0.690
Median: 0.688
Average: 0.687
### Parallellism: 8
Hash: a85446e56b744eb9dbdb5f0b49faddf25e7e7fc664af93a2d2c9a96f16695eec
0.560
0.560
0.553
0.559
0.548
0.557
0.558
0.555
0.548
0.554
Median: 0.556
Average: 0.555
### Parallellism: 16
Hash: 65703e6c261cd4a8d9aa552951a41212f2651d69343ce62aaa2a7b78bb7126b3
0.446
0.444
0.460
0.444
0.454
0.459
0.462
0.442
0.465
0.445
Median: 0.450
Average: 0.452
### Parallellism: 32
Hash: df7cb33149e8cc6997e6283fe568a3485c74586efd2a832c6a10638ca3a793f3
0.438
0.444
0.439
0.439
0.441
0.437
0.447
0.448
0.440
0.443
Median: 0.441
```

```
Average: 0.442
--- Memory: 2097152 KiB ---
### Parallellism: 1
Hash: 4ea57de91e5dd4283a53ca2be669c637586953d704ce52c2c45be2c94a90efa9
2.830
2.772
2.767
2.815
2.796
2.793
2.802
2.816
2.797
2.762
Median: 2.796
Average: 2.795
### Parallellism: 2
Hash: 12e55f0831c82a1e532b5d81845327f2a2dffea72fc40efd4e3258e1bae6fbd6
1.777
1.767
1.779
1.770
1.768
1.771
1.774
1.774
1.767
1.772
Median: 1.772
Average: 1.772
### Parallellism: 4
Hash: 1f3e17b304568a4f49ef4e2e6fa4826e8d4f46cceddf157adf9b7b153cf17eb0
1.328
1.331
1.328
1.327
1.332
1.325
1.321
1.321
1.320
1.324
Median: 1.326
Average: 1.326
### Parallellism: 8
Hash: b88baf017311a8212087fd32f095807d19978d61309fa4eefa5197fe2c824d1a
1.068
1.067
1.065
1.072
1.059
1.073
1.093
```

```
1.066
1.066
1.071
Median: 1.067
Average: 1.070
### Parallellism: 16
Hash: ce2b97110556451e7b1f39877fcc9f5349978ef0784e4c06138eb84587b1ff10
0.871
0.910
0.864
0.866
0.886
0.875
0.873
0.871
0.866
0.868
Median: 0.871
Average: 0.875
### Parallellism: 32
Hash: f9883f0c593681c31b3de840c0a6330ad2f0f68e1490886e9bdaa45c36bc0775
0.842
0.839
0.845
0.845
0.840
0.848
0.834
0.839
0.845
0.836
Median: 0.841
Average: 0.841
```

B CPU Benchmark - Variable: Iterations

```
--- Memory: 32768 KiB ---
### Iterations: 1
Hash: c636fc371f651de1fe5de8aead7d9a10c63788c50a87f949b4eef969c3d1b743
0.051
0.061
0.058
0.060
0.059
0.059
0.058
0.058
0.058
0.058
Median: 0.058
Average: 0.058
### Iterations: 2
Hash: feb2e45a207884fc8eea1e4c7b4406516a2803bdfc79db8bfffb2c7e10c4c966
0.063
0.073
0.071
0.070
0.072
0.073
0.070
0.073
0.072
0.059
Median: 0.071
Average: 0.070
### Iterations: 3
Hash: 1cad4772a777920c14c9c5fbf1f41af9783458ce358fb343d8e469ec43b364d9
0.085
```

```
0.086
0.087
0.087
0.087
0.070
0.085
0.083
0.087
0.086
Median: 0.086
Average: 0.084
### Iterations: 4
Hash: 9e1177007073bb5fb899c2abf0a2a809a31139f6cfc764334cbe9d2a71c0a6d0
0.096
0.096
0.100
0.097
0.099
0.099
0.099
0.082
0.098
0.083
Median: 0.098
Average: 0.095
### Iterations: 5
Hash: 816bc8eacc467b6a69f4c46571160515a55de35e3fcd644af2bd14cec312d939
0.110
0.110
0.108
0.110
0.108
0.107
0.111
0.108
0.109
0.097
Median: 0.108
Average: 0.108
--- Memory: 65536 KiB ---
### Iterations: 1
Hash: bea906ecd5e6981ae9532155cda8d22b2935d21f4a8c20160a47a17e964bfedc
0.087
0.087
0.099
0.104
0.100
0.102
0.104
0.099
0.099
0.101
Median: 0.100
```

```
Average: 0.098
### Iterations: 2
Hash: 90b33ccd036c4b015c51f5cb6e9dbe4b35bf62769d3554c0b52d4939e9ab70a3
0.128
0.129
0.127
0.129
0.113
0.128
0.125
0.128
0.129
0.129
Median: 0.128
Average: 0.127
### Iterations: 3
Hash: 595a75fa52539e458497b751188e7143f16850bd8618cc3fa4edb774684d298b
0.155
0.150
0.153
0.152
0.152
0.140
0.155
0.152
0.141
0.141
Median: 0.152
Average: 0.149
### Iterations: 4
Hash: 41fc4aa5787918c9d86a9500d39029c4ebc690b438f94545d04aa6798540b8f3
0.176
0.181
0.163
0.177
0.175
0.181
0.178
0.179
0.180
0.178
Median: 0.178
Average: 0.177
### Iterations: 5
Hash: e9c743e710890087812523363922 93b375ae2be68c3b7db43589fdfdccf1044a
0.203
0.208
0.203
0.208
0.209
0.191
0.193
0.192
```

```
0.192
0.189
Median: 0.198
Average: 0.199
--- Memory: 131072 KiB ---
### Iterations: 1
Hash: d0678c00b97f8eb9537ee4feea5f5877675cb3597c13348535a6bff64184e73e
0.186
0.184
0.179
0.182
0.183
0.183
0.180
0.185
0.184
0.185
Median: 0.183
Average: 0.183
### Iterations: 2
Hash: 7582e457d5f8e5dc3315e7e096d6b0fcccf448c34f9bc7f5da69ccad618ac126
0.239
0.238
0.222
0.233
0.238
0.243
0.234
0.237
0.233
0.237
Median: 0.237
Average: 0.235
### Iterations: 3
Hash: 4288663db389ce6627c3b39d458e68b89a4c1570126cfb602642bac86250863d
0.291
0.291
0.279
0.289
0.296
0.289
0.293
0.284
0.294
0.285
Median: 0.290
Average: 0.289
### Iterations: 4
Hash: 8de0fd2e81be2766aed6045524d9f018192bedb5db4890cb19562ae01d41c047
0.350
0.347
0.345
0.344
```

```
0.344
0.347
0.332
0.330
0.342
0.349
Median: 0.344
Average: 0.343
### Iterations: 5
Hash: 3b6c949dc666fc3b048bdaa866752c5bf7cf631c4cfd5a07e04f20361cbfc9a2
0.398
0.394
0.405
0.399
0.389
0.400
0.397
0.385
0.401
0.388
Median: 0.398
Average: 0.396
--- Memory: 262144 KiB ---
### Iterations: 1
Hash: 81dd8ee3366bc391277e264d193fec0fcf0a5a73802e1256a2174b67e983cf3c
0.351
0.336
0.351
0.333
0.349
0.348
0.333
0.346
0.348
0.351
Median: 0.348
Average: 0.345
### Iterations: 2
Hash: 1194e41250d38fd74ad1f063e872a82e0ccbf0b9a1cdef10907699b8b52861e4
0.462
0.463
0.447
0.458
0.462
0.450
0.460
0.449
0.450
0.463
Median: 0.459
Average: 0.456
### Iterations: 3
Hash: 2f11e76eb7a2fbcd606fec2602de1f36f905a04c6024dc6a0b310e4c9fba58e5
```

```
0.566
0.575
0.569
0.576
0.559
0.559
0.560
0.573
0.557
0.573
Median: 0.567
Average: 0.567
### Iterations: 4
Hash: 45835da1ee703897fc0e13b6b1677921112be10bccbf94c9de41e3800fdee22b
0.695
0.693
0.689
0.672
0.685
0.691
0.693
0.685
0.682
0.691
Median: 0.690
Average: 0.688
### Iterations: 5
Hash: e2e62215308574f180157b35c987cd045c3400a51bea622c575524f406e735d4
0.805
0.803
0.794
0.800
0.802
0.795
0.790
0.794
0.775
0.801
Median: 0.798
Average: 0.796
--- Memory: 524288 KiB ---
### Iterations: 1
Hash: f22a6090ee805d7a3dc01875a8c4113193ed8e57739c7e92dcbff5e154b542e9
0.685
0.683
0.696
0.688
0.690
0.686
0.672
0.671
0.696
0.680
```

```
Median: 0.685
Average: 0.685
### Iterations: 2
Hash: 860bacc607e1a10dfb2b1a6bbcd2c0805cb20d1dd593d45fcb42b9eb44a21eb6
0.935
0.926
0.916
0.913
0.921
0.923
0.931
0.923
0.926
0.920
Median: 0.923
Average: 0.923
### Iterations: 3
Hash: a993d170302d9045acbe444007de7d6a59725615f05a94a3c474d3cbeb47419f
1.160
1.173
1.172
1.158
1.155
1.151
1.174
1.161
1.159
1.160
Median: 1.160
Average: 1.162
### Iterations: 4
Hash: 9e2ebcc7a89dc063d00140ef1513d2250c8c519f46f6579c586f329ddc3d3b8c
1.386
1.383
1.369
1.366
1.374
1.400
1.378
1.375
1.390
1.394
Median: 1.381
Average: 1.382
### Iterations: 5
Hash: b3f7e6770d3c3a7191d317092057671211e9cc00e0463a4b1525d33ed61d2803
1.610
1.605
1.640
1.619
1.626
1.619
1.645
```

```
1.627
1.605
1.610
Median: 1.619
Average: 1.621
--- Memory: 1048576 KiB ---
### Iterations: 1
Hash: eca700db0e881d738166019509f73d5e056d8d7b090020bf5977333254c0ea02
1.394
1.373
1.380
1.379
1.376
1.392
1.366
1.379
1.371
1.362
Median: 1.377
Average: 1.377
### Iterations: 2
Hash: f93d13e2018d09b8a31456e109c98906602797ab40f1e9bc3662f5179ddfa267
1.861
1.840
1.854
1.860
1.868
1.864
1.873
1.869
1.867
1.847
Median: 1.863
Average: 1.860
### Iterations: 3
Hash: b626ccfe0a33351b9ea19f9213af30b3cb51c601f8ba2aa783fd7f36c0ce76b5
2.335
2.377
2.335
2.362
2.333
2.376
2.355
2.346
2.365
2.354
Median: 2.354
Average: 2.354
### Iterations: 4
Hash: 04408a7d1571aa78372a382c7e8699aff54acd6f0c551e0abf488825bfcf04e5
2.845
2.906
2.935
```

```
2.830
2.894
2.889
2.932
2.898
2.875
2.902
Median: 2.896
Average: 2.891
### Iterations: 5
Hash: d8edd9406613c84d49d68187f69ee3203838236f2a99a79c04b3f64adf607fcb
3.391
3.440
3.350
3.430
3.357
3.377
3.338
3.352
3.400
3.360
Median: 3.369
Average: 3.380
--- Memory: 2097152 KiB ---
### Iterations: 1
Hash: 4ea57de91e5dd4283a53ca2be669c637586953d704ce52c2c45be2c94a90efa9
2.796
2.751
2.807
2.777
2.757
2.821
2.759
2.802
2.755
2.760
Median: 2.768
Average: 2.778
### Iterations: 2
Hash: 3d39430c1c488e4d5957e60e28b08266e634d67c3986ff1c5da1648a0afd77de
3.785
3.736
3.806
3.757
3.762
3.767
3.785
3.750
3.762
3.764
Median: 3.763
Average: 3.767
### Iterations: 3
```

```
Hash: 61111896b04925062862a011e82ffb5b447e33e431afd55a2dfff5d1adb4f88e
4.757
4.832
4.761
4.829
4.760
4.738
4.797
4.761
4.763
4.749
Median: 4.761
Average: 4.775
### Iterations: 4
Hash: bfad8b5338590a1a1d36cab70ff3d4936a6d82c8ee4abbbd69de39499bb9d708
5.838
5.809
5.833
5.781
5.877
5.767
5.749
5.775
5.823
5.758
Median: 5.795
Average: 5.801
### Iterations: 5
Hash: 6e37703bf2559c16106a5dc21ac5d07db099817b66a133dcb4a398792599748a
6.852
6.752
6.894
6.784
6.887
6.831
7.183
6.810
6.835
6.816
Median: 6.833
Average: 6.864
```