Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Maestría en Diseño Electrónico

Implementation of a Hash Security Algorithm using an ARM Cortex M3 Soft Core.

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: **OMAR EDUARDO CASTRO GONZÁLEZ**

Director **CUAUHTÉMOC RAFAEL AGUILERA-GALICIA**

Tlaquepaque, Jalisco. Julio 14 2025.

Acknoledgements

I would like to express my deepest gratitude to my advisor, Cuauhtémoc Aguilera, for his guidence, expertise and practice throughtout this research journey.

I am also grateful to my peer and colleagues at Intel Corporation, for fostering a collaborative and inspiring environment.

Finally I am thankful to my family, their endless support and love

To René, Alex, Valeria, Tota, Sofia, Isabella, Leonardo, and especially Adriana, for her unwavering encouragement that inspired me to complete this journey.

Abstract

This work presents the implementation of the SHA-3-512 hash algorithm on an ARM Cortex-M3 soft core deployed on an Xilinx Artix-7 FPGA. Initially, we introduce hash algorithms, outlining their role in ensuring data integrity, authentication, and security in various applications, including cryptography, digital signatures, and blockchain technology. We then delve into the inner workings of SHA-3-512, a member of the Keccak family, emphasizing its sponge construction and robust resistance to collision attacks, making it one of the most secure hash algorithms. The study details the acquisition of the ARM Cortex-M3 softcore from ARM's DesignStart program, including modifications to streamline the softcore by removing unneeded peripherals and optimizing it for our specific application. Two implementations are developed: a pure software implementation, executed entirely on the ARM Cortex-M3, and a mixed hardware-software co-processor design that leverages the FPGA's parallel processing capabilities to enhance performance. We compare the performance and resource utilization of both implementations, analyzing metrics such as execution time, power consumption, and FPGA resource usage. The results demonstrate that the mixed hardware-software co-processor achieves a performance improvement of 161 times that of the pure software implementation while maintaining efficient resource utilization. This case study highlights the advantages of hardware acceleration in cryptographic applications and provides insights into optimizing hash algorithm implementations on resource-constrained embedded systems.

***Tags**: Hash, ARM, SHA, FPGA, Cryptography*

Contents

Introduction

“Emerging applications such as the Internet of Things (IoT), smart cities, and autonomous vehicles demand more efficient and smaller electronic systems [Aguilera-Galicia - 19].” These devices must be capable of engaging with real-world information by gathering, processing, and responding to data or presenting it to users in an accessible and beneficial way.

“The term "Internet of Things" (IoT) denotes a trend where a large number of embedded devices employ communication services offered by Internet protocols. Many of these devices, often called "smart objects," are not directly operated by humans but exist as components in buildings or vehicles or are spread out in the environment [Tschofenig et al. -15]."

All these devices require a level of security that must be matched to the level of computing power that the IoT will possess. Some guidance on the level of protection can be found from NIST (National Institute of Standards and Technology), a federal agency under the U.S. Department of Commerce that plays a crucial role in developing technology and security standards.

Under the IoT Device Cybersecurity for the Federal Government, developed by NIST in the Special Publication 800-213 [Fagan et al. - 21], there are two main categories of requirements: technical cybersecurity capabilities and non-technical supporting capabilities. Within the Technical Cybersecurity Capabilities, it lists device-specific functions, including Device Identification, Device Configuration, Data Protection, Logical and Physical Interface Control, Software and Firmware Updates, and Cybersecurity Event Logging. According to the internal report from NIST 8259A [Fagan et al., 2020], the technical measures for IoT devices to achieve data protection include encryption, which utilizes cryptographic mechanisms to protect data at rest and in transit. Data Integrity – Implement cryptographic hashing or digital signatures to detect and prevent data modification. Access Control – Restrict access to data based on authentication, ensuring only authorized entities can access sensitive data. Secure Storage – Protect sensitive data stored on the device. Secure Communication – Use secure communication protocols and ensure end-to-end encryption where feasible. In this work report, we focus on the data integrity mechanism.

“One of the most widely used IoT cryptographic primitives is the cryptographic hash function, which plays an essential role in various cyber and information security applications. This function maps an arbitrary-length input to a fixed-length output, producing a hash value or message digest fingerprint [Aumasson -18].

In this work, we aim to implement a cryptographic hash function in an Internet of Things (IoT) device.

The evaluation of hash functions for IoT solutions can be measured using several key performance metrics proposed by [Windarta et al. - 22]; these metrics are divided into software and hardware implementation categories as follows:

Software Implementation Metrics

Read-Only Memory (ROM) or Code Size: Measures the fixed amount of memory required to store the hash function’s code, independent of input size. Lower ROM usage is crucial for IoT devices with limited storage capacity.

Random Access Memory (RAM): Evaluates the memory needed for temporary data during hash computation. Lower RAM usage is preferred for resource-constrained devices.

Hardware Implementation Metrics

Gate Area: Quantifies the physical area (in gate equivalents) required for hardware implementation on devices like FPGAs or ASICs. Smaller gate areas reduce hardware costs and power consumption.

General Metrics

Latency: Measured in cycles per block or cycles per byte; this indicates the time taken to process a block or byte of data. Lower latency is better for performance in constrained environments.

Throughput: Represents the rate at which data is processed. Higher throughput is desirable for efficient data handling in IoT systems.

Power and Energy Consumption: Measures the power and energy required for hash computation. Low power and energy usage are critical for battery-powered IoT devices.

All these metrics will be required at the end of our work to evaluate the implemented solutions.

To implement a hash function tailored for IoT devices, two proposed approaches are discussed by [Rodríguez-Henríquez et al. - 06] "The first methodology is to use microprocessors and the second

is to use reprogrammable logic devices." The primary advantage of using standalone microprocessors is that they are optimized for sequential operations, achieve higher clock frequencies, and, in some instances, the instruction set may include special instructions that optimize hash operations. However, the main disadvantage is that most IoT microprocessors are not optimized for parallel operation. Thus, performance may be affected. The use of reprogrammable logic devices has the advantage that a microprocessor may be instantiated in the programmable logic; these devices are known as softcore. Additionally, parallel logic can be added to the softcore to speed up certain operations that can be handled in parallel.

In this work, we choose to use a reprogrammable logic device and implement a softcore that provides hash functionality, both with and without parallel logic, to enhance parallel operations.

The work is divided into five chapters, which we now describe.

Chapter 1 presents the theory of how cryptographic hash primitives work, the various types that exist, and the metrics and security requirements associated with them. It introduces the reader to the hash security language that will be used throughout the document. Finally, it will showcase some applications where SHA-3 implementations can be utilized on IoT devices.

Chapter 2 delves into the specifics of the SHA-3 algorithms, focusing on their implementations within a 64-bit computing environment for pedagogical reasons. This chapter provides a detailed examination of these widely used hash functions and their operational nuances in the context of 64-bit systems. Although our primary aim is not to build an SHA-3 algorithm in a 64-bit environment, we create one that can provide us with reference outputs and partial internal results to validate our design against.

Chapter 3 outlines the process required to integrate a microprocessor softcore into an FPGA. It specifically outlines the process of using the Design Start ARM program to instantiate a Cortex-M3 softcore into an Artix-A7 FPGA.

Chapter 4 describes the modifications made to simplify the default Design Start Project by removing debug features and capabilities for GPIO and SPI. It also illustrates how to add AXI accelerator logic, which can be used to implement parallel logic that supports the standalone softcore.

Chapter 5 details the implementation of a 32-bit SHA3-512 software running on the standalone softcore. We then implement a mixed software-hardware solution that leverages the advantages of

programmable logic to enhance performance compared to the first solution. Both solutions will be functionally validated, and their metrics will be compared and evaluated.

Throughout these chapters, the reader will gain a comprehensive understanding of hash functions, from their theoretical underpinnings to practical implementations in low-power computing environments. This work aims to equip readers with the knowledge to evaluate, implement, and optimize cryptographic schemes in various hardware contexts using the Design Start ARM Cortex-M3 softcore. For readers who are not interested in hash security functions, this work will also serve as a detailed description of how to use the Design Start ARM Cortex-M3 softcore to implement various algorithms.

1. Introduction to Hash Algorithms

1.1. What is a Hash function?

"The concept of security for hash functions is centered around protecting data integrity, ensuring that data—whether transmitted openly or encrypted—remains unaltered. Consider the most common application of hash functions: digital signatures. If even a single bit of the message is altered, the resulting hash will be entirely different [Aumasson -18]." See, for example, **Error! R eference source not found.**, a message passes through a Hash Function, and the output is a Hash value.

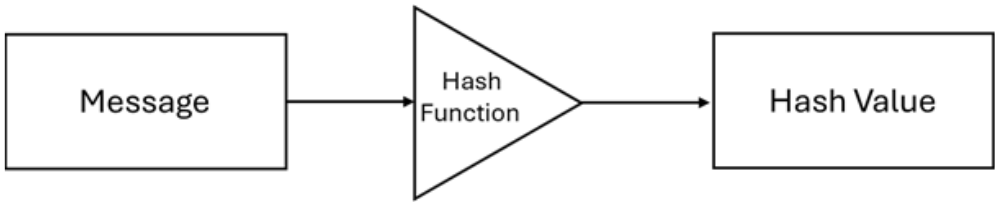

Figure 1-1- Displays a message being processed by the hashing block (triangle) to produce a singular block known as the hash value.

To verify the message's integrity, the receiver must apply the same hashing operation to the message and compare it with the transmitted hash. The cryptographic strength of hash functions stems from the unpredictability of their outputs. For example, the ASCII values of the letters *'a'* and *'c'* differ by only one bit. The ASCII value for *a* is 011000012, and the value for *c* is 01100011, but the SHA-3 hash value of these two letters is entirely different.

SHA3-512 ("a") = 697f2d856172cb8309d6b8b97dac4de344b549d4dee61edfb4962d8698b7f
a803f4f93ff24393586e28b5b957ac3d1d369420ce53332712f997bd336d09ab02a
SHA3-512 ("c") = bfe4d7f7377116dc15f794d902621797b72b32396382de2b6e49d4f1d7eAb
dfddcfc3bc127bb67f92f9458a5733bb21804e7ccd56b4b6f81049339f477cd279d

A secure hash function should behave like a black box, producing a unique and unpredictable string each time it receives input.

1.2. Strategy to build Hash functions.

According to [Aumasson -18], the overall strategy for processing messages is to break them into pieces of the same size and process them one at a time. This strategy, known as iterative hashing, can be implemented in two ways. The first method uses a Merkle-Damgård construction, which uses a compression function to transform an input into a smaller output. In Figure *1-2*, M_1 and M_2 are identical-size pieces of the original input message, H_0 is an initial seed value, and H_1 is the output of the first compression block that is fed into the second block.

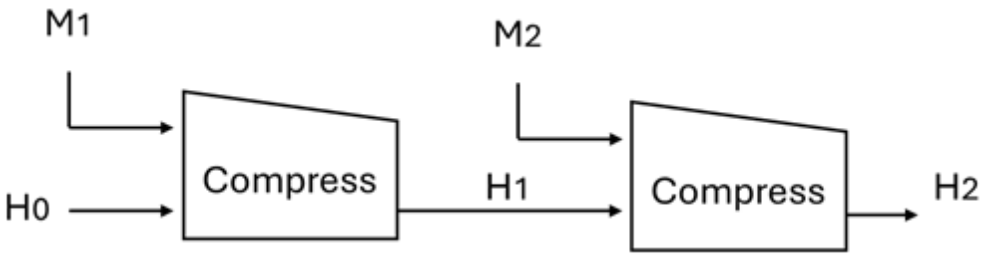

Figure 1-2- Merkle-Damgård construction using a compression function called Compress. The figure is taken from [Aumasson -18].

One example of this type of construction is the MD5 (Message Digest Algorithm 5). It was popular at the end of the last millennium and in the early 2000s. The MD5 hash algorithm uses four distinct compression functions to map parts of the input message into scrambled outputs. One of these compression functions can be seen in Figure *1-3*.

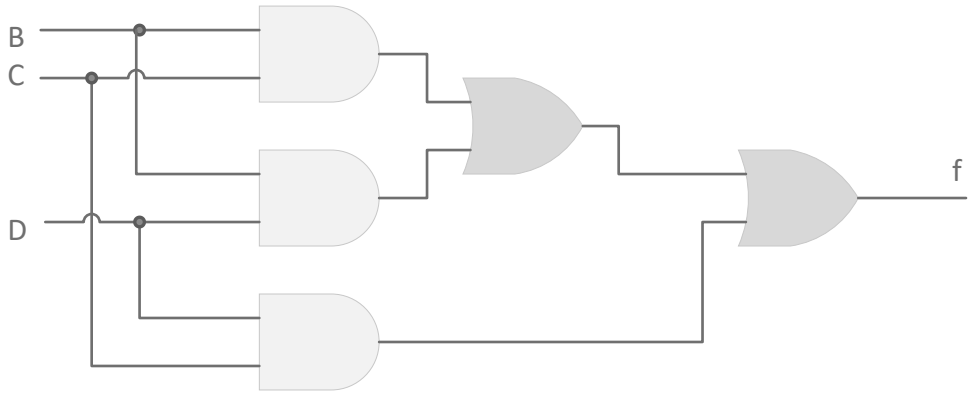

Figure 1-3- Example of one out of four compression functions used in MD5, the inputs correspond to a part of Message *B* and *D* and Initial seeds *C*. *f*(*B, C, D*) = (*B* & *C*) | (*B* & *D*) | (*C* & *D*). The figure is taken from [Rivest – 92]

Another way to implement the hash function is through iterative hashing; this utilizes a function that transforms an input into an output of the same size. Such functions are called sponge functions.

In *Figure 1-4*, M_1, M_2, and M_3 are fragments of the initial message, all the same size, while H_0 represents the initial sponge values. Each P block includes binary operations that shuffle and reorder the inputs. The message fragments first get "*absorbed*" into the process, not simultaneously, but after some prior permutation. The output is later "*squeezed*" out of the process.

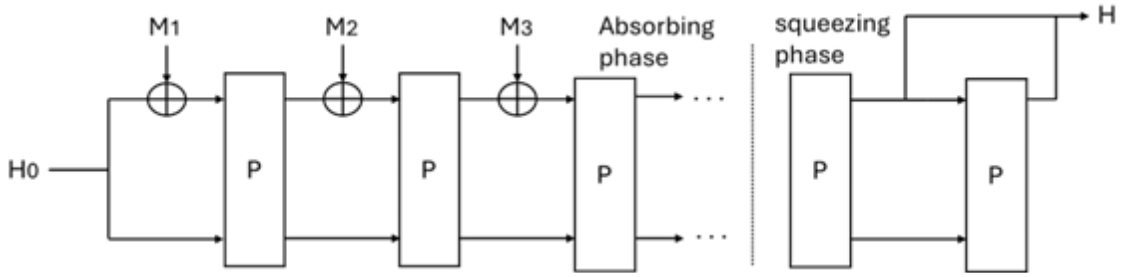

Figure 1-4- The sponge functions take the message parts at the absorbing phase, permuting the input with some previous result and getting an output at the squeezing phase. Taken from [Aumasson -18].

One example of this construction is the Keccak family of hash functions, which we will explore later in this document.

1.3. Security Requirements.

According to [Dang – 12], hash functions need to have the following properties:

Collision Resistance – It should be computationally infeasible to find two inputs with the same hash output. "Collision Resistance is measured by the amount of work that would be needed to find a collision for a cryptographic hash function with high probability"[Dang – 12]. For example, an attacker creates two contracts (*A* and *B*) with the same hash, gets one (*A*) signed, and then swaps it with *B*, which has the same hash and a valid signature. Collision resistance prevents this by making it hard to find such pairs.

Preimage Resistance - It should be challenging to find an input that produces a given hash. "Preimage resistance is measured by the amount of work that would be needed to find a preimage for a cryptographic hash function with high probability. The estimated strength for the preimage resistance provided by a hash function is the length of the hash value"[Dang – 12]. For example, a website saves password hashes. A user password *A*

produces a hash *H[A].* If an attacker hacks the database and obtains the *H[A],* they will try to find an input that, when hashed, produces the same *H[A].* Preimage resistance prevents this by making it challenging to reverse the hash.

Second Preimage Resistance - It is challenging to find a different input matching a known input's hash. "Second preimage resistance is measured by the amount of work that would be needed to find a second preimage for a cryptographic hash function with high probability. The estimated strength for the second preimage resistance provided by a hash function is the length of the hash value, and for some specific algorithms, it may depend on the size of the input message length"[Dang – 12]. An example would be if you have a signed document (*A*) that produces *H(A)* once signed, an attacker wants to forge a different document (*B*) that has the same *H(A)* to trick the signature verification. The second preimage makes this challenging.

The NIST (National Institute of Standards and Technology) has summarized the security strengths of the main Hash algorithms that this federal agency has recommended.

Significant efforts are underway to develop algorithms that possess the three security properties previously described. To implement a hash algorithm, it is advisable to utilize one of the algorithms

TABLE I - STRENGTHS OF THE SECURITY PROPERTIES OF THE APPROVED HASH ALGORITHMS. TAKEN FROM [NIST – 15A] AND [DANG – 12].

	Collision Resistance Strength in bits	**Preimage Resistance Strength in bits**	**Second Preimage Resistance Strength in bits**
SHA-1	< 80	160	≤ 160
SHA-224	112	224	min (224, ≤256)
SHA-256	128	256	≤256
SHA-384	192	384	384
SHA-512	256	512	≤ 512
SHA-512/224	112	224	224
SHA-512/256	128	256	256
SHA3-224	112	224	224
SHA3-256	128	256	256
SHA3-384	192	384	384
SHA3-512	256	512	512

listed in TABLE I. For this work, we have decided to use SHA3-512 as it offers the highest security strength.

1.4. Hash Applications.

As mentioned earlier, hash functions serve a crucial purpose: generating signatures for known messages. If an intercepted message is modified, the altered signature will not match the original. This verification mechanism is used to secure emails, software distribution, online transactions, and legal documents. There is, however, another application, which is proof of work. In recent years, there has been a surge in cryptocurrencies; in all these schemes, the hash function plays a central role, not only in the send/receive signing operations but also in the main proof-of-work, which validates every transaction.

The proof of work applications function in the following way:

When a cryptocurrency transaction occurs, the transaction data is stored in a block header. The block header also includes the software version, the previous block hash, a timestamp, the Merkle root, and the difficulty target. This block header is shared with all the miners in the system. The miners begin with this initial information and combine it with a nonce, a 32-bit number. They then start performing the hash function on the combined data. The first miner to find a hash value that meets the difficulty level (the number of zeros in the hash) will create a valid block. This information will be shared with the other miners. The difficulty for the next block will be adjusted, the original transaction will be honored, and the miner will receive payment for their discovery. The mining process involves constant hash calculations, making it an energy-intensive activity. Implementations with low power requirements are necessary to reduce this energy consumption. Examples include Bitcoin, which uses SHA3-256, and Ethereum, which utilizes a variant of SHA3-256 called Keccak-256.

1.5. Conclusions

In this chapter, we introduce hash functions, their construction, and the security requirements they must meet. The key takeaway is that hash functions serve as both tools and potential targets. They are continually analyzed and assessed by both ethical and malicious actors, with real financial incentives for breaking specific security schemes. For non-expert cryptologists, it is crucial to use official and vetted hash implementations. These are the advantages of implementing algorithms with the highest collision resistance and primary and secondary preimage resistance, such as SHA3-512.

2. SHA3-512 Algorithm

2.1. The SHA family of Hash Functions.

SHA hash functions are standards set by NIST for use by non-military federal agencies in the US. They're considered worldwide standards, and while some governments might choose other hash functions for reasons of sovereignty, it's mainly for non-security reasons.

Message Digest 5 (MD5) was the go-to hash function from 1992 to 2005 when it was cracked. The SHA-1 was created and released by the National Security Agency (NSA) in 1995 and uses a Merkle-Damgård function. Following several attacks on SHA-1, NIST suggested switching to SHA-2, also developed by the NSA. SHA-2 is a family of four hash functions: SHA-256, SHA-384, SHA-512, and SHA-224 [Aumasson -18].

"In 2007, NIST organized the Hash Function Competition to select the successor to SHA-2. NIST, along with selected cryptanalysts, analyzed and verified a total of 51 entries. After 15 months, NIST selected five finalists: Blake, Grøstl, JH, Keccak, and Skein. Following additional months of analysis, the Keccak hash algorithm was declared the winner, and it became SHA3." [Aumasson -18].

2.2. Keccak (SHA-3) algorithm details.

Here, we summarize the algorithm descriptions from [Bertoni et al, – 11] and [NIST – 15A] to explain how the algorithm works. Readers interested in the details are advised to consult the original descriptions for further information.

At its core, this algorithm is a sponge construction that processes binary data with a variable output length. Several key parameters define the sponge's behavior:

- The sponge function (*f*), which is the generic function used in the sponge construction.
- The padding algorithm (*pad*) adds bits to reach the required input size.
- The rate (*r*), which determines how often the sponge processes the input message

- The capacity (*c*), which represents the security level of the function. A higher capacity indicates a more secure function, but it also requires more processing time to handle the entire message.
- The input (*M)*, which is the input string,
- The output size (*d*), which is the size of the output hash.
- The output string (*Z*), which is the output hash.

The sponge construction can be seen in Figure *2-1*.

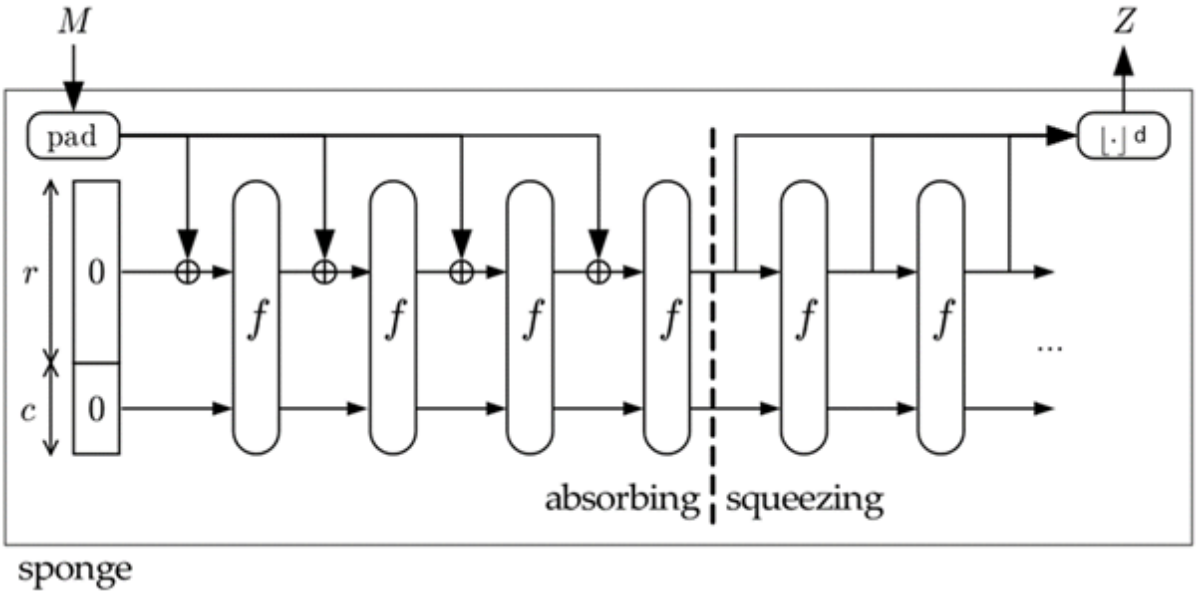

Figure 2-1- The sponge construction accepts a message *M* that gets padded. The padded messages get fragmented, and a sponge permutation function *f* processes an amount *r*. After all parts of *M* are processed, the algorithm outputs Hash[*M*] = *Z*, a value of size *d*. Taken from [Bertoni et al, – 11]

Within each permutation sponge function, we have state mappings that will scramble the input. See Figure *2-2* to see the internal components of the sponge permutation function.

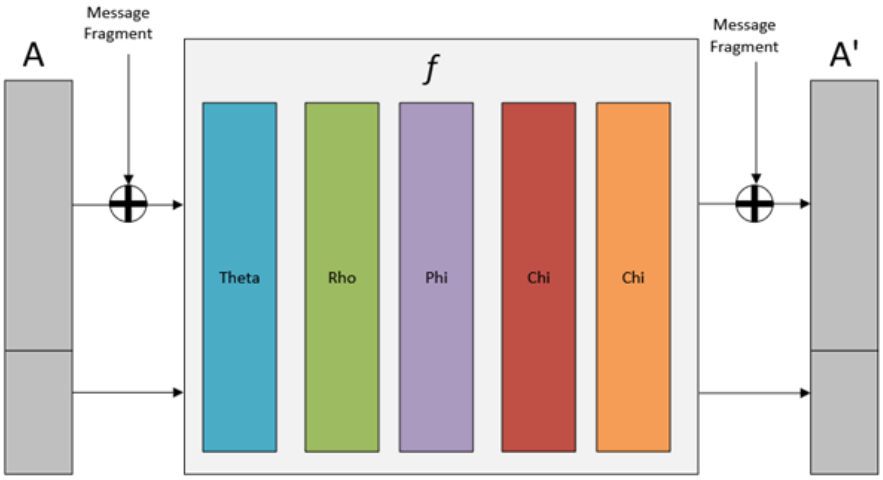

Figure 2-2- The sponge permutation function consists of five operations within each function *f* that shuffle and scramble the input array *A* into an output *A'*. The Message Fragment will be added before and after each scrambling operation.

We will now provide a brief overview of the overall algorithm.

1. The message (*M*) enters the sponge.
2. The message gets padded and fragmented into several parts. The amount that gets fragmented depends on the rate (*r*) and the capacity (*c*).
3. The initial fragment of the padded message will get absorbed into an array of bits called state *A*. This can be imagined as a 3-dimensional array of bits of fixed 5 x 5 x *w* bits, where *w* is the lane size. This array will be the input to the sponge permutation function *f*.
4. Inside the function *f*, there will be scrambling operations, called state mappings, that execute operations on the 3-dimensional array *A*.
5. After the sponge permutation function *f* finishes, the output of that function will be an updated array *A'*.
6. A new fragment of the padded message is then absorbed (XORed) into the updated array *A'*. In total, there will be a finite number of rounds.
7. When the rounds end, a finite number of bits will exit the state array to start forming the output *Z*.
8. There will be additional rounds until it extracts and completes the expected output length *d*.

A more in-depth description of the SHA3 algorithm continues. Before presenting some of the main algorithms, we need to provide some guidance on variables and operations that can be seen in TABLE *II* and TABLE .

TABLE II – ALGORITHM OPERATIONS [NIST – 15A] AND [DANG – 12].

0^s	For a positive integer s, 0^s is the string that consists of s consecutive zeros. If s = 0, then 0^s is the empty string.
len(*x*)	For a bit string *x*, len(*x*) is the length of x in bits.
x[i]	For a string x and an integer *i* such that $0 \leq i <$ len(x), *x*[i] is the bit of *x* with index i. If *x* = 001000, then *x* [2] = 1
$Trunc_s(x)$	For a positive integer s and a string x, Truncs(x) is the string comprised of bits *x*[s-1] to *x*[0]. Example: $Trunc_2(10110) = 10$
$X \oplus Y$	For a string X and Y of equal bit length, the operation results in applying the Boolean exclusive-OR operation. Example $1100 \oplus 1010 = 0110$.
X || Y	For strings X and Y, the operation results in the concatenation of both strings. For Example, 11001 || 010 = 11001010
m/n	For integers *m* and *n*, *m / n* is the integer quotient, *m* divided by *n.*
m mod *n*	For *m* and *n* integers, *m* mod *n* is the integer *r* for which $0 \leq r < n$ and *m-r* is a multiple of *n*. For example, 11 mod 5 = 1.
log2(*x*)	For a positive real number *x*, log2(*x*) is the real number y such that $2y = x$.
$X \cdot Y$	If X and Y are bit Boolean values, the operation is an integer multiplication, equivalent to a Boolean AND operation.
pad	Refers to a specific padding scheme used to ensure that the length of the input message becomes a multiple of a particular block size.

TABLE III – ALGORITHM PARAMETERS [NIST – 15A] AND [DANG – 12].

Value	Definition
A [x, y, z]	3-dimensional State Array A, the bit that corresponds to the triple (x, y, z)
b	The width of a Keccak-p permutation in bits
c	The capacity of a sponge function
w	The lane size of a Keccak-p permutation in bits *w* = (**b**/25)
f	The generic function of the sponge construction
i_r	The round index for a Keccak-p permutation
L	For a Keccak-p permutation, the binary logarithm of the lane size. log2(*w*)
Lane [i, j]	For a State Array A, a string of all the bits of the lane with x and y coordinates equal to i and j.
M	The input string to a SHA-3 hash algorithm
nr	The number of rounds for a Keccak-p permutation
r	The rate of a sponge function
RC	For a round of Keccak-p permutation, the round constant

2.2.1 Padding Algorithm – 10*1 padding rule and separation suffix

In the SHA3 (Keccak family) algorithm, the first step is to verify the input message and figure out if padding is required. The padding process, known as the 10*1 padding rule, involves the following steps:

1) Append a single bit to the message.
2) Add a sequence of zero bits
3) Append a final one-bit to the message.

The padding ensures the message length is a multiple of the rate (*r).* The algorithm for padding is shown below. Figure *2-3.*

Input:

Positive integer: r representing the rate

Positive integer: len(M) represents the Message Length

Output:

string P such that len(M) + len (P) is a positive multiple of the rate r.

Steps:

Let j = (-len(M)-2) mod r.	*// j represents the number of 0s to append*
Return P = 1 || 0^j || 1	*//P is the string started with 1, j number of 0s and a last //appended 1*

Figure 2-3- Padding Algorithm: pad10*1 (*r*, len(*M*)). The length of the original message and the rate are the inputs to the algorithm, which calculates the number of zeros needed to be added between the 1-bits. The padding string will have a length such that when summed with the length of the original message, the result will be a multiple of the rate [Bertoni et al, – 11] and [NIST-15]

Let us follow this padding algorithm and work through an example; for SHA3-512, the padding algorithm uses a fixed rate (*r*) of 72 bytes. Let's assume we have an initial message of 31 bytes.

72 bytes is equivalent to 576 bits, and 31 bytes is equivalent to 248 bits; therefore, we will execute pad10*1(576, 248). The first step is to calculate the number of zeros to append (*j*), which will ensure the total length is a multiple of the rate: the original message + the size of the 2 bits that are

one + the total number of zeros, i.e., (248 + 2 + *j*). In this case, *j* = (-248 - 2) mod 576 = 326. We need a pad with 326 zeros; the final padding will be $1||0^{326}||1$.

Let's explore some more interesting cases, assuming the rate is 576 bits. A message with 0 bits would need to have 574 zeros: $1||0^{574}||1$. A message with 577 bits would need to be padded $1||0^{573}||1$. In total, there would be 577 + 1 + 573 + 1 = 1152 bits.

Along with the pad10*1 algorithm, the SHA3-512 standard, as defined by [NIST-15], requires the use of a suffix, also known as the domain separation suffix. This fixed value is 0x06, and its primary goal is to differentiate the padding from other Keccak-based functions, such as SHAKE128 and SHAKE256, which use a different suffix of 0x1F. Thus, the final padding mechanism becomes:

1 – Append the binary sequence 00000110 (0x06) to the end of the message.

2 – Apply the 10*1 padding scheme to ensure the final padded message is a multiple of the rate.

2.2.2 The sponge construction algorithm: sponge (*f, pad, r*)

We begin with a message *M*, a string that the algorithm processes to generate an output string *Z* of length *d* bits. Next, we create a new string *P* by concatenating the original message *M* with a string of padding bits. The number of padding bits depends on the original rate and the length of message *M*. The string *P* is divided into packets, each with a size equal to the rate *r*. We initialize a state array A of 1600 bits, filled with zeros. We loop through all packets of the string *P*. For each packet, we XOR it with the state array A, and the result is passed through a permutation function f to create a new state A′. The new state A′, resulting from processing the previous packet, becomes the new A for processing the next packet. After all packets are processed, we extract *r* bits from the array A until we obtain the required number of bits *d*. If the needed number of bits *d* is not reached, we pass the state array A through the permutation function and continue extracting bits to reach the target length *d*. The algorithm can be seen in Figure *2-4*.

Inputs:

String M

nonnegative integer d

Output:

string Z such that len(Z) = d.

Steps:

$Let\ P = M\ ||pad10 * 1(r, len(M)).$	*// We generate a string P concatenating the original string and the output of the padding function. See algorithm 1*
$n = \frac{len(P)}{r}$	*//n is the number of parts the new string P will get //divided.*
$c = b - r$	*//The capacity gets calculated, width of a Keccak-p permutation in bits minus the rate*
$Let\ P_0, .., .., P_{n-1}, be\ the\ unique\ sequence\ of\ strings$	*//P gets divided into equal parts*
$A\ =\ 0^b$	*//The initial state A gets initialized with all 0s size*
$\boldsymbol{for}\ i = 0\ to\ n-1\ \boldsymbol{do}$ $A' = f(A \oplus (P_i \| 0^c))$	*//for each part in the message do the following //concatenate a string with initial fragment of //size r with a 0-value string of size c. This string //will have size b. Apply exclusive OR operation on //this string with state array A. Apply the permutation //function to the state array, to generate the new //state array. Do again for each message block.*
$Let\ Z\ be\ empty\ string$	*//Z will be the output string we initialize with empty //string.*
$\boldsymbol{while}\ len(Z) \leq d$ $Z = Z\ \|\|Trunc_r A$	*//keep doing the following operation until Z has size d //Obtain the first r number of bis from the //state array A and concatenate into Z*
$A = f(A)$	*//Apply generic function to state array.*

Figure 2-4 – The sponge construction algorithm, sponge (*f, pad, r*): The algorithm starts with a message that will be padded and then divided into *n* parts. Each part of the message is XORed with the state array, which is subsequently processed through the permutation function *f*. After processing each part of the message, we will remove a fixed number of bits from the state. array until we complete the total number of bits d that we require [Bertoni et al, – 11] and [NIST-15].

In Figure *2-4*, we describe a Keccak generic algorithm that utilizes Keccak sponge functions with various parameters. The approved SHA-3 is composed of multiple Keccak sponge functions with varying settings. These functions are defined by the rate of r and capacity c, where $r + c$ fall within a set of specific values: 25, 50, 100, 200, 400, 800, or 1600 bits. The following TABLE *IV* illustrates this relationship.

TABLE IV - KECCAK PERMUTATION WIDTHS AND RELATED QUANTITIES IN BITS. [NIST – 15A].

Permutation width b	25	50	100	200	400	800	1600
Permutation Lane size w	1	2	4	8	16	32	64
l = log₂ (w)	0	1	2	3	4	5	6

The permutation width b represents the total size of the Keccak state in bits, a key parameter in the Keccak sponge construction. Different values of b correspond to distinct Keccak permutation configurations, enabling flexibility in balancing security, performance, and resource utilization. The lane size w defines the size of each lane in the Keccak state, which is organized as a 5 x 5 x w array of bits (a 3D structure with 25 lanes, each of size w). Lastly, l, the binary logarithm of w, is used to determine how many times the permutation function needs to run. This logarithmic relationship simplifies the design and implementation of Keccak, ensuring that lane sizes align with binary arithmetic and making operations such as shifts and rotations more efficient.

When restricted to $b = 1600$, the Keccak family is denoted by Keccak[c]. Thus, we can express the Keccak[c] as the following equation (2-1) :

$$Keccak[c] = Sponge[Keccak - p[1600,12 + 2l], pad10 * 1,1600 - c](M, d) \quad (2\text{-}1)$$

Using (2-1) we can represent SHA3-512 using the following values:

Output length d: 512 bits

Capacity: $c = 2 * d = 1024$ bits.

Permutation width $b = 1600$ bits

Rate $r = b - c = 1600 - 1024 = 576$ bits

Lane size $w = 1600 / 5 \text{ x } 5 = 64$ bits

Number of Rounds $n_r = 12 + 2* l = 12 + 2 * 6 = 24$

Finally, we can represent it as

Keccak [1024] = Sponge [Keccak-p [1600, 24], pad10*1, 576] (M, 512) = SHA3-512

To compare, we can do the same for SHA3-256

Output length d: 256 bits

Capacity: $c = 2 * d = 512$ bits. This ensures a security level of 256 bits of preimage resistance and 128 bits of collision resistance.

Permutation width $b = 1600$ bits

Rate $r = b - c = 1600 - 512 = 1088$ bits

Lane size $w = 1600 / 5 \times 5 = 64$ bits

Number of Rounds $n_r = 12 + 2* l = 12 + 2 * 6 = 24$

Finally, we can represent it as

Keccak [512] = Sponge [Keccak-p [1600, 24], pad10*1, 1088] (M, 256) = SHA3-256

2.2.3 State Array A and its representation as a 3-D array

As previously stated, the state array A will be continuously used to get and mix data throughout the Keccak sponge operation. State Array A can be visualized as a 5 x 5 x w array of bits. Then, the notation A[x, y, z] represents the triples (x, y, z) that have a static position in space. In Figure *2-5*, we can see the 3-D representation of state A as a 5 x 5 x w-1 array.

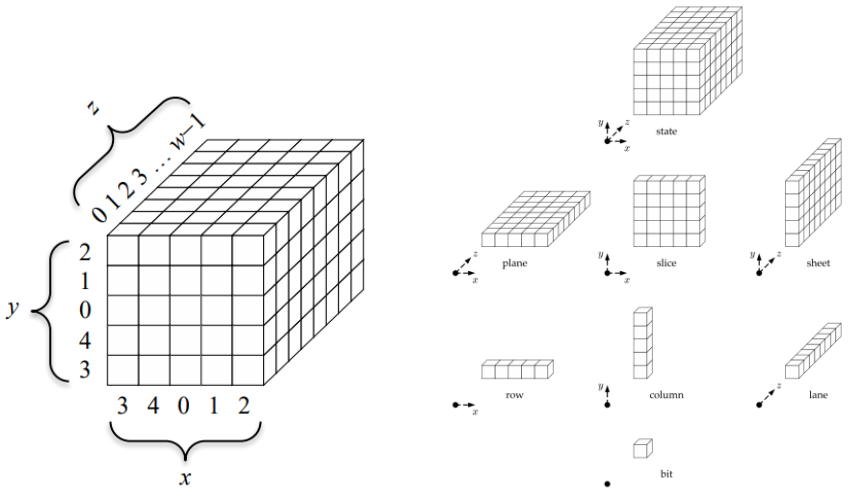

Figure 2-5 – A 3-D representation of the state array A. The x, y, and z coordinates for the diagrams of the step mapping (Left). Parts of the state array are organized by dimension (Right). Take from [NIST – 15A].

In Figure *2-5*, we can see some visual representations or subgroups in the state array *A*. These selections are required for specific operations of the permutation function.

- x-axis: 5 columns (x = 0 to 4).
- y-axis: 5 rows (y = 0 to 4).
- z-axis: *w* bits deep.
- Lane: A single line along the z-axis at a fixed (x, y).
- Row: A line along the x-axis at a fixed (y, z).
- Column: A line along the y-axis at a fixed (x, z).
- Slice: A 5 × 5 plane at a fixed z, perpendicular to the z-axis.
- Plane: A 5 × *w* plane at a fixed y, parallel to the xz-plane.
- Sheet: A 5 × *w* plane at a fixed x, parallel to the yz-plane.

We can also describe the state array as a 1-dimensional linear string S, then its bits are indexed from 0 to b-1, as shown in (2-2)

$$S = S[0]||S[1]||..||S[b-1] \tag{2-2}$$

We can switch between the linear format and the array format using an (2-3).

$$A[x, y, z] = S[w(5y + x) + z] \tag{2-3}$$

For example, for SHA-3, we have the following mappings:

For example, if b = 1600, w = 64 then

A [0,0,0] = S [64(5*0+0) + 0] = S [0]

A [0,0,1] = S [64(5*0+0) + 1] = S [1]

A [0,0,2] = S [64(5*0+0) + 2] = S [2]

..

A [0,1,0] = S [64(5*1+0) + 0] = S [320]

A [4, 4, 63] = S [64(5*4 + 4) + 63] = S [1599]

2.2.4 The Sponge Permutation Function

In the last subsection, we described the state array. In this subsection, we will examine all the permutation operations applied to the values in the state array. The permutation function comprises five-step mappings, denoted by Greek letters: Theta Θ, Rho ρ, Pi π, Chi χ, and Iota ι.

Specification for Theta[Θ] Step Mapping

The Theta step enhances diffusion (small change affects multiple positions) by XORing each bit of the state with parity values computed from adjacent columns in the array. This step disrupts local patterns, ensuring small input changes spread widely across the state. The algorithm is shown in Figure 2-6.

Input: *state array* $\boldsymbol{A}$

Output: *state array* $\boldsymbol{A'}$

for $x = 0$ *to* 4 ***do***

  for $z = 0$ *to* 63 ***do***

    $C[x,z] = \boldsymbol{A}[x,0,z] \oplus \boldsymbol{A}[x,1,z] \oplus \boldsymbol{A}[x,2,z] \oplus \boldsymbol{A}[x,3,z] \oplus \boldsymbol{A}[x,4,z]$

for $x = 0$ *to* 4 ***do***

  for $z = 0$ *to* 63 ***do***

    $D[x,z] = C[(x-1) mod 5, z] \oplus C[(x+1) mod 5, (z-1) mod 64]$

for $x = 0$ *to* 4 ***do***

  for $y = 0$ *to* 4 ***do***

    for $z = 0$ *to* 63 ***do***

      $A'[x,y,z] = A[x,y,z] \oplus D[x,z]$

Figure 2-6 – Algorithm for Theta Step Mapping: First, we compute the parity of the column. Once we have a whole plane of parity values, we XOR adjacent values. From [NIST – 15A]

We can represent the step mapping graphically in the

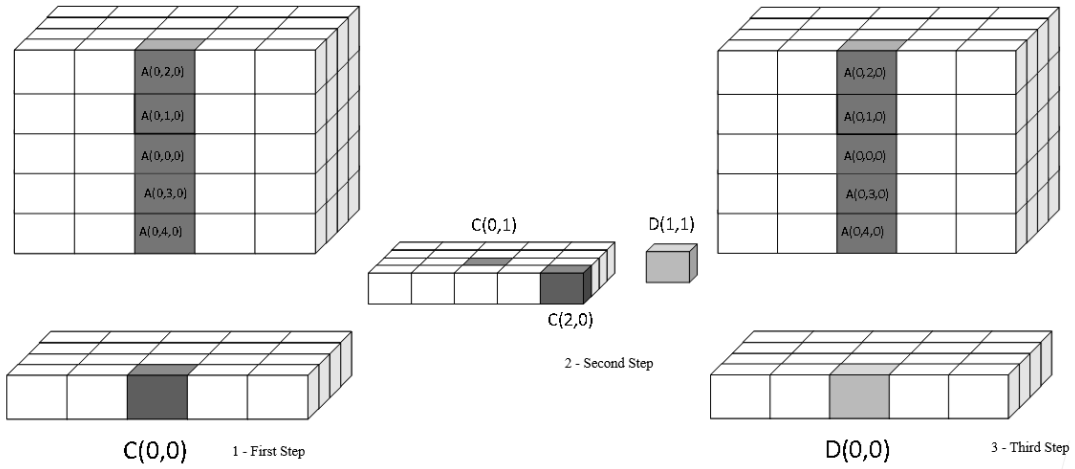

Figure *2-7*.

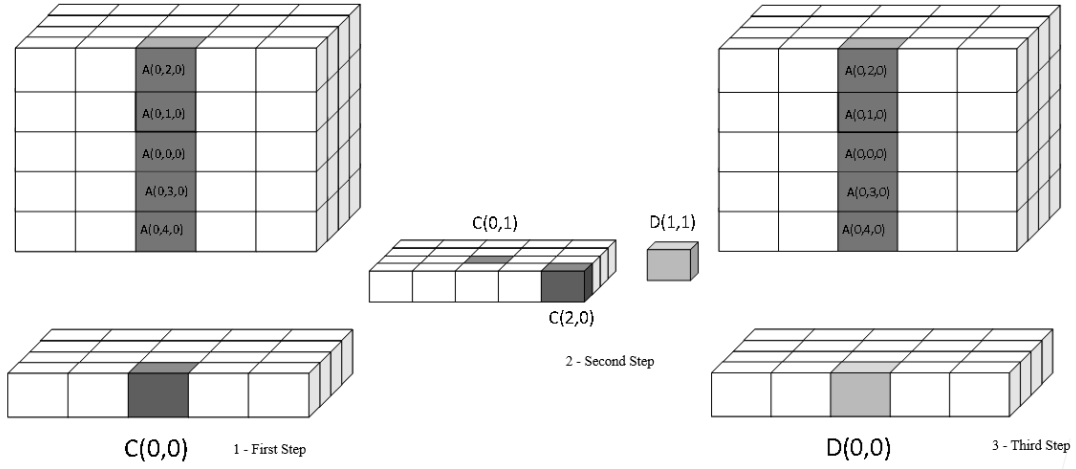

Figure 2-7- To graphically represent the Theta step mapping, we first construct the parity plane C by XORing all columns in the array. Once we have the plane C, we XOR values in adjacent rows to obtain an intermediate value D, which we use to fill a new plane D. Finally, we XOR each value in $A[x, y, z]$ with $D[x, z]$ to create the new value at $A'[x,y, z]$.

Specification for Rho [ρ] State Mapping

The Rho step in Keccak rotates each of the 25 lanes (64-bit words) in the 5 x 5 state array by a fixed offset specific to each lane's (x, y) position. This permutation disperses bits within each lane to improve diffusion and break alignment across the state. The offsets are defined in a triangular pattern to ensure diverse mixing. The following algorithm is shown in *Figure 2-8*.

Input: *state array* ***A***

Output: *state array* ***A'***

for $z = 0$ *to* 63 ***do***

$A'[0,0,z] = A[0,0,z]$

Let $(x, y) = (1, 0)$

for $t = 0$ *to* 23 ***do***

for $z = 0$ *to* 63 ***do***

$$let\ A'[x,y,z] = A\left[x, y, \left(z - \frac{(t+1)(t+2)}{2}\right) mod 63\right]$$

$$let\ (x,y) = (y, (2x + 3y) mod 5)$$

Figure 2-8 - Algorithm for Rho Step Mapping: We first copy the [0,0] lane, as this lane does not undergo rotation. After this operation, it will start rotating or moving the z-coordinate by a fixed amount for each x- and y-coordinate until all coordinates are processed, as described in [NIST – 15A].

The following Figure 2-9 Shows a graphical representation of the rotation of each bit in a column, where it is shifted to a new position in the w coordinate.

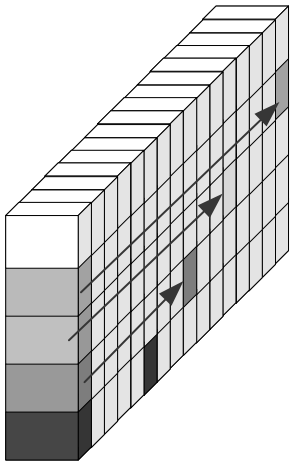

Figure 2-9 – Shifting values by a fixed amount within a sheet in the array. The image illustrates an example of how each bit in a column shifts within the *w* coordinate. The translation within the lanes aims to provide dispersion. Taken from *[NIST – 15A]*.

Specification for Pi [π] State Mapping

The Pi step permutes the 25 lanes of the 5 x 5 x 64-bit state array by rearranging the x and y coordinates while leaving the z coordinate unchanged. These reshuffles enhance diffusion,

ensuring bits from one lane influence different coordinates in the following steps. It's a fixed permutation that redistributes the state without altering individual bit values within lanes. The following is the algorithm for the π step, shown in Figure *2-10*.

Input: *state array* ***A***

Output: *state array* ***A'***

for $x = 0$ *to* 4 ***do***

  for $x = 0$ *to* 4 ***do***

    for $y = 0$ *to* 63 ***do***

$$A'[x, y, z] = A[(x + 3y) mod 5, x, z]$$

Figure 2-10 - Algorithm for Pi Step Mapping: We first copy the [0,0] lane, as this lane does not undergo rotation. After this operation, it will start rotating or moving the z-coordinate by a fixed amount for each x- and y-coordinate until all coordinates are processed, as described in [NIST – 15A].

In Figure 2-11, we can see a visual representation of the shuffling being done at the slice level.

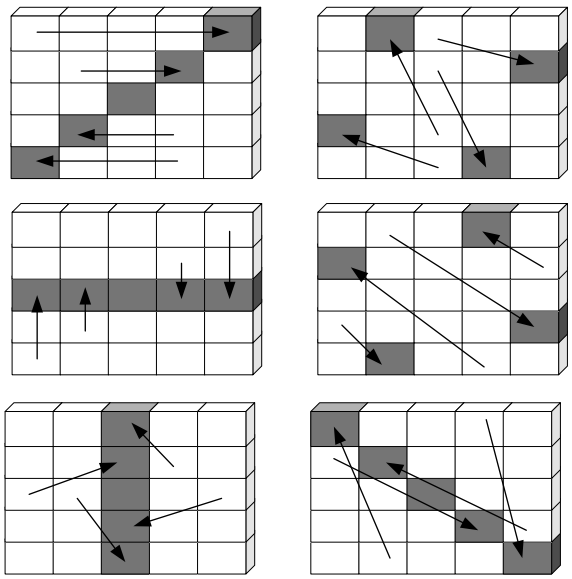

Figure 2-11 – Pi step reorders the lane positions in the 5 x 5 grid, ensuring a bit from one lane affects different coordinates in subsequent steps. Taken From [NIST – 15A].

Specification for Xi [χ] State Mapping

The Xi state mapping applies a nonlinear operation (introduces interdependencies between bits so that the output cannot be expressed as a simple sum or linear combination of inputs). This operation makes it harder to reverse-engineer the transformation. The algorithm can be seen in Figure 1-1.

Input: *state array* ***A***

Output: *state array* ***A'***

for $x = 0$ *to* 4 ***do***

  for $y = 0$ *to* 4 ***do***

    for $z = 0$ *to* 63 ***do***

$$A'[x, y, z] = A[x, y, z] \oplus ((A[(x+1) mod 5, y, z] \oplus 1) \cdot A[(x+2) mod 5, y, z])$$

Figure 2-12 - Algorithm for Xi Step Mapping: It takes the bit position at (x, y, z); it then XORs the adjacent x coordinate and ANDs the adjacent coordinate that was previously XORed with a fixed value of 1. Taken from [NIST – 15A].

In Figure 2-13, we can see a graphical representation of the transformation that each bit undergoes.

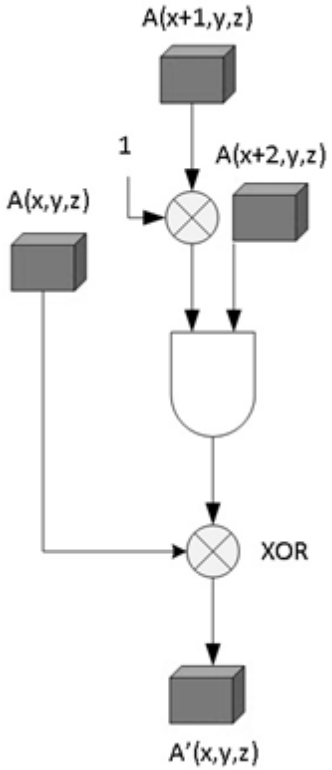

Figure 2-13 – Graphical representation of the Xi step mapping for each bit in the state Array. By mixing adjacent values from the state array, the algorithm introduces nonlinear dependencies that are hard to reverse-engineer from [NIST – 15A].

Specification for Iota [ι] State Mapping

The last step of mapping transformation is the Iota; it introduces asymmetry to the 5 x 5 x 64-bit state array to prevent symmetry-based attacks. It XORs a single lane (the lane at coordinates (x = 0, y = 0)) with a round-specific constant, leaving all other lanes unchanged. The round constants are derived from a linear feedback shift register (LFSR) and vary for each of the 24 rounds in Keccak-f[1600]. This step ensures that each round is unique, disrupting any potential symmetries or invariant properties in the state. Overall, ι enhances the cryptographic security of SHA-3 by breaking patterns that could be exploited. The algorithm to obtain the round constant is shown in Figure *2-14*.

Input: integer t

Output: bit $rc(t)$

Let $R = 1$

for i from 1 to 24

  for j from 1 to 7

    $bitposition = (1 \ll j) - 1$ // left shift operation

    if $LFSR(R)$ is 1 //LFSR with the function $x^8 + x^6 + x^5 + x^4 + 1$

      $rc(i) = rc(i) \oplus 1 \ll bitposition$

Figure 2-14 – Algorithm to calculate the round constant that will be used in the Xi state mapping. The algorithm may be implemented, or values can be precalculated and stored, as they do not change.

Once the round constants have been calculated, they are used in the Iota state mapping, shown in

Input: state array **A**

round index **rc**

Output: state array **A'**

for $z = 0$ to 63 **do**

  $A'[0,0,z] = A'[0,0,z] \oplus RC[z]$

Figure 2-15 - The Iota algorithm uses the round constants to XOR each of the values at coordinate x=0, y=0 for the complete lane.

2.3. Full step-by-step execution of a real SHA-3 – 512 Example

In the following example, we present the implementation results of SHA-3-512, which corresponds to a Keccak function with a rate of *r* = 576 bits and a capacity of *c* = 1024 bits. The delimiter is 0x06 and the output size *d* = 512 bits:

Input message = FPGA_ARM_CORTEX_M3

Input message in hexadecimal format: 46 50 47 41 5F 41 52 4D 5F 43 4F 52 54 45 58 5F 4D 33

Size of the message, m = 144 bits.

As described in Algorithm 1, we apply pad10*1(576, 144); we are required to find the smallest string P such that (144 + Len(P) + 2) mod 160 = 0.   To calculate, we obtain j = (-144-2) mod 576 = 430. The padding then would need to be concatenated with 0x06, followed by 430 zeros and then a 1; see Figure *2-16*.

```
46 50 47 41 5F 41 52 4D 5F 43 4F 52 54 45 58 5F 4D 33 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 2-16 - Padded Input Message: the first partial message gets into the state array, and the delimiter 0x06 and padding bits are added.

The same padded message can be seen in the following representation of the state array, where a 5 x 5 array with 64 bits per value is displayed. See Figure 2-17

```
4D52415F41475046 5F584554524F435F 000000000006334D 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 8000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

Figure 2-17 Padded Input message as a 5 x 5 x 64-bit array, the values are arranged as the state mapping operations operate on them.

The initial state mapping, known as the Theta state, is responsible for diffusing the 1600-bit state. Its primary function is to blend bits within and across lanes, ensuring that alterations in one section of the state influence multiple other sections. See Figure *2-18*.

```
F3E2CBF7E5D9D6F8 120A040B13047583 5F58455452497013 000000000006334D 1AA482BE828EA08C
BEB08AA8A49E86BE 4D52415F414B36DC 5F584554524F435E 800000000006334D 1AA482BE828EA08C
BEB08AA8A49E86BE 4D52415F414B36DC 5F584554524F435E 000000000006334D 1AA482BE828EA08C
BEB08AA8A49E86BE 4D52415F414B36DC 5F584554524F435E 000000000006334D 1AA482BE828EA08C
BEB08AA8A49E86BE 4D52415F414B36DC 5F584554524F435E 000000000006334D 1AA482BE828EA08C
```

Figure 2-18 - After Theta State Mapping, round 0, the main task of this operation is to blend bits across lanes.

The next state mapping role is to shift bits within each lane, enhancing the overall mixing of bits. See Figure Figure *2-19*.

F3E2CBF7E5D9D6F8 241408162608EB06 D7D6115514925C04 00006334D0000000 F414750460D52415
49E86BEBEB08AA8A B36DC4D52415F414 D611551493D0D797 A6C0000000000319 2BE828EA08C1AA48
F584554524F435F5 49057D052CDB713 7A1AF2FAC22AA292 00000C669A000000 4750460D52415F41
3D0D7D7D61155149 66DB89AA482BE829 22AA2927A1AF2FAC 000000C669A00000 A482BE828EA08C1A
2AA2927A1AFAFAC2 3549057D052CDB71 CBEB08AA8A49E86B 4D00000000000633 20AFA0A3A82306A9

Figure 2-19 - After Rho State Mapping, round 0, the main job of this state mapping is to mix bits within the same lane.

The following state mapping Pi is designed to permute the positions of the lanes. It will shuffle the lanes to prevent localized patterns. See Figure *2-20*

F3E2CBF7E5D9D6F8 B36DC4D52415F414 7A1AF2FAC22AA292 000000C669A00000 20AFA0A3A82306A9
00006334D0000000 2BE828EA08C1AA48 F584554524F435F5 66DB89AA482BE829 CBEB08AA8A49E86B
241408162608EB06 D611551493D0D797 00000C669A000000 A482BE828EA08C1A 2AA2927A1AFAFAC2
F414750460D52415 49E86BEBEB08AA8A 49057D052CDB7135 22AA2927A1AF2FAC 4D00000000000633
D7D6115514925C04 A6C0000000000319 4750460D52415F41 3D0D7D7D61155149 3549057D052CDB71

Figure 2-20 – After Pi State Mapping, round 0, the main task of this state is to distribute bits to other regions to prevent fixed patterns.

The following state mapping is the Chi operation, which provides non-linear mixing. Each lane is XORed with a non-linear combination of two neighboring bits in the duplicate row, thus preventing linear cryptoanalysis attacks. See Figure *2-21*.

F3E2CBF7E5D9D6F8 B36DC4D52415F414 7A1AF2FAC22AA292 000000C669A00000 20AFA0A3A82306A9
00006334D0000000 2BE828EA08C1AA48 F584554524F435F5 66DB89AA482BE829 CBEB08AA8A49E86B
241408162608EB06 D611551493D0D797 00000C669A000000 A482BE828EA08C1A 2AA2927A1AFAFAC2
F414750460D52415 49E86BEBEB08AA8A 49057D052CDB7135 22AA2927A1AF2FAC 4D00000000000633
D7D6115514925C04 A6C0000000000319 4750460D52415F41 3D0D7D7D61155149 3549057D052CDB71

Figure 2-21 – After Chi State Mapping, round 0, the main job of this operation is to combine lanes with neighboring values.

Finally, the Iota tasks modify lane [0,0] by XORing it with a round-specific constant (the round constant), which changes for each execution. It's meant to introduce asymmetry, preventing the formation of patterns and adding different bits for subsequent executions. See Figure *2-22*.

BBF0F9DD27F3D47B B36DC4D10D95F414 5AB552DB4229A43B D3404B922C78D050 20A2A4A3A82726AD
D4043631F43415B5 29B3A04040CA6240 7CA45545A6B435B7 66DBEABE182BE829 E003006082884223
241400742E08EB06 7293E79497705B8D 0A200C1E8A5A72C0 A096B686AAA08D1E F8A3C77A8B2AEE53
F411610064067520 6B426BC96A2CA402 04057D052CDB7126 92BE5C23C17A0FA8 44E80AEB8B088CB9
96C6575846D30044 9ECD397021140311 4710460D5669D571 FF9B6D7D7187554D 1549057D052CD868

Figure 2-22- After Iota State Mapping, round 0, the final operation that XORs only one lane [0,0] with an external fix round constant that changes for every execution, adding variability.

All these operations are carried out 24 times, after which we get the result shown in the Figure *2-23*

BEE3450C45723230 4248DA81EADEB324 **DE44ED89DE1A0AD3** 88CE744245D21B71 **808B9CCE24E52F29**
3EBC75652F7BA750 **FA232C22E5817D28** 75B30B6ABA3B1671 5093379E83BDE095 616903D4EA9C8A84
F418F4E2EEAD23B8 EB70E922E92CA304 41F1D3A04098D789 783103AD691DC631 1A154EBCA6F8E32B
299677DEAB80202F B4640B46305DEF4A E7BF142B2FE2E57F F0F05AEFC36DB423 7C5A9C4A8F7FDB5C
E710A2D74E4531DE 75A004BC4321D7B1 9321C5E942F7DAD9 23F7FB8193B12D91 8D682B98998A9AFA

Figure 2-23 – After 24 rounds, the final value of the state. The top 8 64-bit words represent the first output; if more data needs to be processed, then the new 72 bytes must be XORed with the state array after a round of state mappings.

The final value is then extracted from the array 512 bits at a time.

30 32 72 45 0C 45 E3 BE 24 B3 DE EA 81 DA 48 42 **D3 0A 1A DE 89 ED 44 DE** 71 1B D2 45 42 74 CE **88 29 2F E5 24 CE 9C 8B 80** 50 A7 7B 2F 65 75 BC 3E **28 7D 81 E5 22 2C 23 FA** 71 16 3B BA 6A 0B B3 75

We can see that it matches the value by running the following command on any Ubuntu/CentOS Linux distribution:

```
print "FPGA_ARM_CORTEX_M3" | openssl dgst -sha3-512
SHA3-512(stdin)=
303272450c45e3be24b3deea81da4842d30a1ade89ed44de711bd2454274ce88292fe524ce9c8b80
50a77b2f6575bc3e287d81e5222c23fa71163bba6a0bb375
```

2.4. Conclusions

Chapter 2 explored the SHA3-512 algorithm, a specific implementation of the Keccak sponge construction, which became the NIST SHA-3 standard due to its strong security and flexible design. SHA3-512 is derived from the Keccak algorithm by setting specific parameters, including the rate (r = 576), capacity (c = 1024), and output size (d = 512), which define the 1600-bit state size. These parameters determine how much of the input message is processed in each iteration of the Keccak-f permutation, directly affecting the algorithm's security and performance. The rate (*r*) controls the amount of message data absorbed into the state during each permutation round, while the capacity (*c*) determines the portion of the state reserved for security, ensuring resistance against attacks like preimage, second-preimage, and collision attacks. In this chapter, we also examined how the Sponge Permutation Function (*f*) operates and presented an example of how each step in the algorithm modifies the input to create the hash output. In Chapter 3, we'll focus on the ARM Cortex-M3 softcore and temporarily put SHA3-512 functionality on hold. We'll revisit it when we implement it in Chapter 5.

3. ARM Cortex Soft Cores

According to [Yiu – 19], ARM Cortex-M processors are among the most popular architectures today for the Internet of Things (IoT) and embedded applications. Furthermore, the ability to implement this architecture in an FPGA as a soft core provides the following advantages:

• Capability to handle complex tasks.

• Application programs can be developed and updated separately from the hardware design.

• Reduces the total number of components on the system board.

• Signal routing is managed automatically by FPGA synthesis tools.

• Debugging software on a well-known architecture is easier than debugging complex state machines.

• Program code can be stored on configuration flash for the FPGA, allowing firmware updates over time.

• Utilizing the Cortex-M processor is easier, as many educational resources are specifically targeted at learning the ARM Cortex Architecture as an introduction to other high-end ARM processors.

• The FPGA may be an initial easy step to prototype designs that will ultimately end up in an SoC containing Cortex-M processors.

For this and other benefits, ARM Cortex Soft Cores provide an alternative to using hard cores or state machines implemented solely in an FPGA.

3.1. Available types of ARM processors

As of 2019, three types of ARM processors may be implemented in low-end FPGAs, according to [Yiu – 19]. See TABLE V.

TABLE V – KEY CHARACTERISTICS OF DIFFERENT CORTEX PROCESSORS OFFERED BY ARM AS PART OF THEIR DESIGN START PROGRAM. TAKEN FROM [YIU – 19].

Feature	**Cortex-M**	**Cortex-R**	**Cortex-A**
Architecture Type	32-bit	32-bit or 64-bit	32-bit or 64-bit
Virtual Memory Support	No	Yes	Yes
Virtualization support	No	Yes	Yes
Interrupt handling	Nested Vectors Interrupt Controller, multi-core, non-deterministic.	Nested Vectors Interrupt Controller, multi-core, non-deterministic.	
Clock Speed	Up to 400 MHz	Up to 3 GHz	Up to 3 GHz

For beginners and educational purposes, ARM provides access to the Cortex-M series through the ARM Flexible and ARM Design Start licensing options [Yiu – 19] [ARM – 10A]

The ARM Flexible Access provides upfront, no-cost, or low-cost access to a wide range of ARM IP, support tools, and training. Experiment and design with the entire IP portfolio, calculating the cost only for the IP included in the final System-on-Chip (SoC) design. There are three tiers in this program: Standard, Entry, and Design Start [ARM – 10B] [ARM – 10C].

Standard and Entry tiers have access to the entire IP repository. Projects are reviewed quarterly, and, in some cases, tape-out fees can be waived depending on volume considerations. The Design Start entry allows you to download ARM Cortex-M3 and ARM Cortex-M1 tools for free, using them as a static project [ARM – 10B][ARM – 10C].

For our work, we will use the free option, which grants us access to the ARM Cortex-M3 but has the following limitations:

- Restricted Configurability – The ARM Cortex-M3 processor is delivered as a preconfigured, synthesizable Verilog model. It lacks the flexibility to modify specific parameters, such as the number of interrupts, which can be adjusted from 8 to 240, and the instruction and data memory sizes, which can be set from 8 KB to 1 MB and 2 KB to 2 MB, respectively [Yiu – 19] [ARM – 10D].
- Toolchain Dependency – We require specific tools, such as Xilinx Vivado for hardware design and ARM Keil for software development. Also, these tools need a free or startup license. [Yiu – 19][ARM – 10D].

- Platform Constraints for FPGA – The Cortex-M3 DesignStart FPGA model is specifically designed for the Artix-7 and Spartan-7 FPGAs. While it's possible to port to other platforms, this requires additional work [ARM – 10D].
- Performance and Resource limitation – The Cortex-M3 FPGA implementation is clocked at 100 MHz. It utilizes AXI-LITE for peripheral communication, which may introduce latency compared to a hard-wired ASIC implementation [ARM – 10D].
- Commercial Deployment – The Free DesignStart only permits prototyping and evaluation; users must transition to a full license for commercial deployment.
- No modification allowed - The license explicitly prohibits altering the Verilog Source Code of the Cortex-M3 processor. The model is delivered as a fixed, encrypted, and obfuscated RTL module [ARM – 20].

We accept the limitations and agree to the license terms to download the ARM-Cortex-M3.

3.1. Testing Example project with Arty-A7-100T

After registering and downloading the ARM Cortex-M3 FPGA package from [ARM – 10E], we will use an Arty A7-100T development board that contains the XC7A100TCSG324-1 FPGA. Therefore, we need to download the board constraint files from [DIGILENT – 23]. We also require the ARM Keil uVision 5 and the Xilinx Verilog 2019.1 software. Since we are not simulating, we do not need to download the Micron Flash Memory Model (Micron_N25Q18A13E) or the Cypress Flash Memory Model (S25fl128s).

The downloaded ARM Cortex-M3 package is designed for a specific hardware board. In this case, the Arty-A7-35 board from Digilent. We will need to specify the FPGA we are using.

We will now describe the steps that we implemented to test the example project on an Arty-A7-100T board.

After unzipping the file, we have the documentation, hardware, software, Vivado folders, and the license.txt. The documentation directory contains reference manuals, some DesignStart FPGA version instructions, and a top-level Block Design of the example project based on the Arty-A7-35T development board. The Hardware folder stores the Vivado project, the Verilog design files,

and the constraints for the Arty-A7-35T board. The Software folder contains the Keil-MDK project and SPI Flash programming files. Finally, the Vivado folder includes the encrypted Arm_IP_Repository files as well as the ARM Software stack. The documentation folder contains several documents on how to validate the example project included in the DesignStart package. The following are the instructions that we followed using the Development board Arty A7-100T and Windows 11. For all our examples, we will assume that we are installing in the C:\FPGA\ directory.

Once everything is installed, open the Vivado project by double-clicking on the file: **C:\FPGA\hardware\m3_for_arty_a7\m3_for_arty_a7\m3_for_arty_a7.xpr**

Vivado will start, and it may show some warning about the Simulation models not being found, which we can safely ignore. We first need to modify the default board. We click on:

Flow Navigation → Project Manager → Settings

A new Settings Window will then pop up. Under:

Project Settings → General

We need to modify the Project Device. We chose the Arty A7-100 development board. We exit the settings Window and click on:

Flow Navigation → Project Manager → IP Integrator -> Open Block Design

When you open the default project block, it'll be visible in the project hierarchy. This block includes the ARM Cortex-M3, Reset and Clock block, an AXI Interconnect Hub, and the AXI UART, AXI DAPLINK, AXI BRAM Generator, AXI GPIO, and AXI SPI blocks.

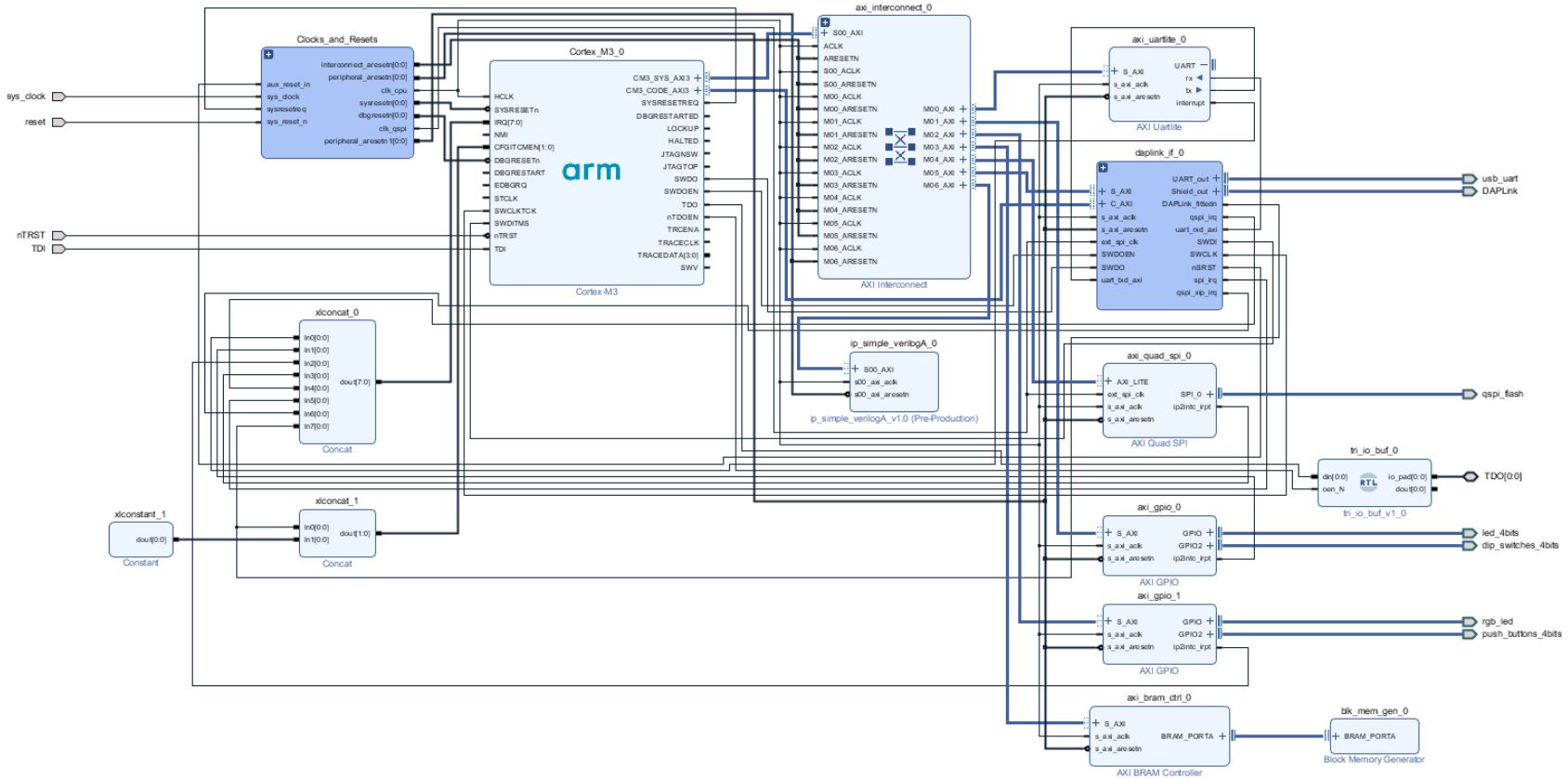

Figure 3-1- ARM DesignStart Example Soft-Core implementation block diagram. The ARM Cortex-M3 is shown in the center. Other modules include the clock and reset block, the DAPLink block used for debugging, two GPIO AXI blocks, one SPI AXI block, one AXI UART, and a BRAM controller and generator to serve as intermediate memory.

To test the default block functionality, we need to perform synthesis and implementation. We accomplish this by running:

Flow Navigation → Project Manager → Synthesis -> Run Synthesis

Flow Navigation → Project Manager → Implementation -> Run Implementation

When these steps are completed without errors, we can generate the bitstream.

Flow Navigation → Project Manager → Program and Debug → Generate Bitstream

The generated bitstream, which is the file that gets programmed into the FPGA, includes all hardware blocks and a software stack that runs directly on the softcore, validating the SPI, GPIOs, UART, and internal memory blocks. It also encompasses a previously compiled software stack embedded in the softcore. The name of this software program is *bram_a7.hex*; it uses the UART to display messages in the serial console and runs basic tests that configure the GPIO, SPI, and BRAM memory.

Before flashing our image onto the Arty A7-100 development board, we connect the board to our computer via USB and open a serial console program set to 115,200 baud, one stop bit, no parity, and Xon/Xoff flow control.

When we first connect our Arty A7-100 development board and power it on, we see in the serial console that a default program is stored in the non-volatile external memory; see Figure *3-2*.Figure *3-2* - The default serial message runs on the Arty A7-100T development board, where the GPIO and switches are operational, controlling the LED on the development board.

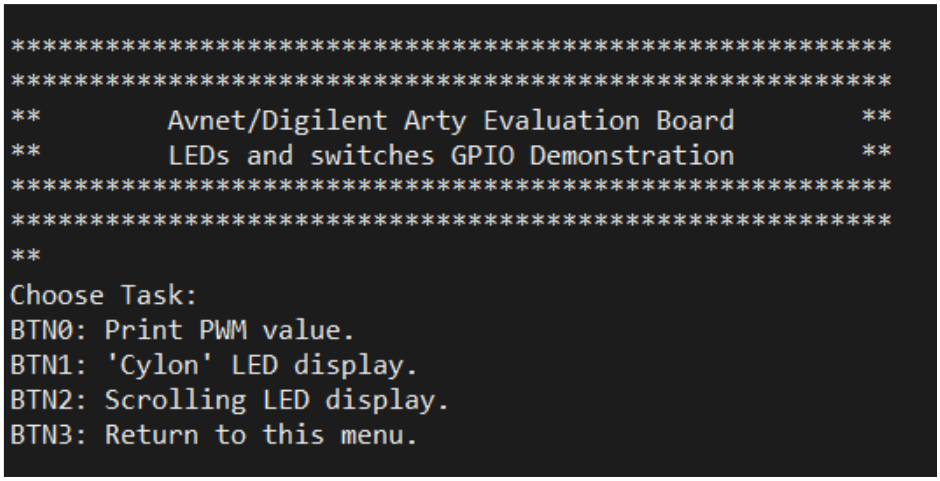

Figure 3-2 - The default serial message runs on the Arty A7-100T development board, where the GPIO and switches are operational, controlling the LED on the development board.

After validating that the connections are correct with the Arty A7-100T development board, we return to Vivado and click on:

Flow Navigation → Project Manager → Program and Debug→ Open Hardware Manager

In the center window, we click on

Open target → Auto Connect

We click on Program Device and select the file.

C:/FPGA/hardware/m3_for_arty_a7/m3_for_arty_a7/m3_for_arty_a7.runs/impl_1/m3_for_arty_a7_wrapper.bit

We click on the Program Device button and verify that we get new messages in the serial console.

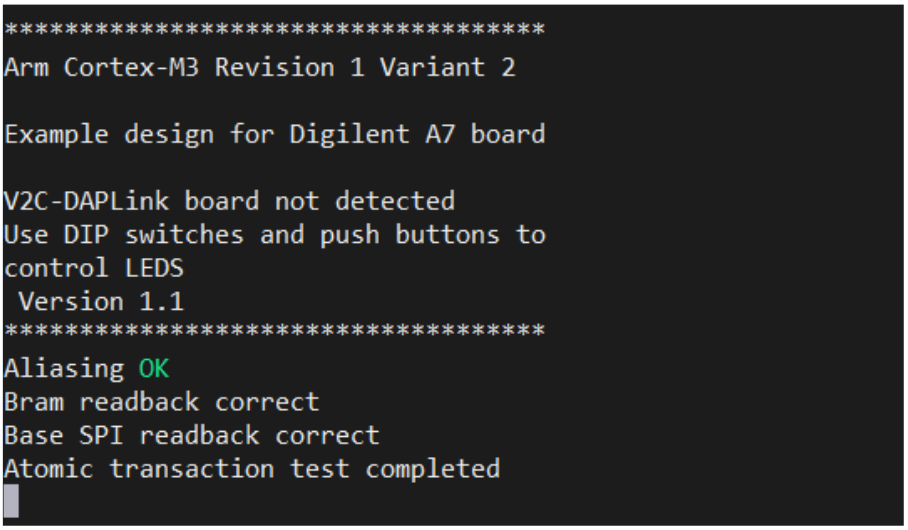

Figure 3-3 The ARM Cortex-M3 Soft Core default program, which comes with the original ARM Cortex Project, presents the user with an informative message indicating what softcore is being run, as well as running DAP, LED, GPIO, SPI, and BRAM checks.

From Figure 3-3, the default software stack is running correctly in the instantiated softcore. Once we validate that the downloaded softcore works, we can proceed to understand how to modify the software stack.

3.2. Modifying the software stack on the default hardware block

Until now, we have relied on the software stack provided with the example project. To create our software stack for the softcore, we must first develop a customized Board Support Package (BSP). The BSP consists of low-level software components, including bootloaders, device drivers, and, in some cases, an operating system specifically designed to operate on a particular hardware

platform. In the Vivado workflow we are using, the BSP is generated by the Software Development Kit (SDK) tool, which takes the hardware definition file exported from the Vivado Design Suite as input. The BSP produces a directory containing all the essential low-level components needed to compile and link the desired software stack for the softcore.

3.2.1 Generating the BSP for the default hardware

Using the default project that we have open, we need to export the hardware definition into the SDK tool. We click the following:

File → Export Hardware

The tool will prompt you to enter the location. It should save the hardware definition file. We use:

C:\FPGA\software\m3_for_arty_a7\sdk_workspace

This operation will save a file *m3_for_arty_a7_wrapper.hdf* that contains a hardware description. After this step, we can go back to Vivado and select

File → Launch SDK

When the SDK application starts, we will see the exported hardware design with its corresponding blocks, memory base, and high address.

m3_for_arty_a7_wrapper_hw_platform_0 Hardware Platform Specification

Design Information

Target FPGA Device: 7a100t
Part: xc7a100tcsg324-1
Created With: Vivado 2019.1
Created On: Mon Apr 14 13:57:24 2025

Address Map for processor Cortex_M3_0

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
axi_gpio_1	0x40120000	0x4012ffff	S_AXI	REGISTER
daplink_if_0_axi_single_spi_0	0x40030000	0x4003ffff	AXI_LITE	REGISTER
daplink_if_0_axi_xip_quad_spi_0	0x40000000	0x4000ffff	AXI_LITE	REGISTER
daplink_if_0_axi_xip_quad_spi_0	0x00000000	0x00ffffff	AXI_FULL	REGISTER
daplink_if_0_axi_gpio_0	0x40010000	0x4001ffff	S_AXI	REGISTER
axi_gpio_0	0x40110000	0x4011ffff	S_AXI	REGISTER
axi_uartlite_0	0x40100000	0x4010ffff	S_AXI	REGISTER
axi_bram_ctrl_0	0x60000000	0x60001fff	S_AXI	MEMORY
daplink_if_0_axi_quad_spi_0	0x40020000	0x4002ffff	AXI_LITE	REGISTER
axi_quad_spi_0	0x40130000	0x4013ffff	AXI_LITE	REGISTER

Figure 3-4 - The SDK Address Map for Hardware Design contains the information used to map some device drivers to their base addresses.

The memory map displayed in the SDK must match the memory map obtained during our initial block design. We encountered some issues where the memory map would not display, or some blocks were missing. In this case, we had to revisit the design to validate it and redo the synthesis and implementation.

We now specify the ARM IP software components. To include these, click on:

Xilinx → Repositories → Global Repositories → New

Specify the following route.

C:\FPGA\vivado\Arm_sw_repository

We need to create a new BSP that targets the specific hardware blocks that we have specified:

File → New → Board Support Package

We will be presented with a pop-up window where we select default values and click the Finish button. A new pop-up window will specify the OS type as standalone; we need to specify OS Version 6.7. By default, the stdin and stdout are configured to use the axi_uartlite_0 hardware block; this should be verified without changes. We have created a new BSP, and now we can exit and copy the assembler initialization and assembler routines into our BSP. Copy xpseudo_asm_rvct.c and xpseudo_asm_rvct.h from

C:\FPGA\vivado\Arm_sw_repository\CortexM\bsp\standalone_v6_7\src\arm\cortexm3\armcc

Into

C:\FPGA\software\m3_for_arty_a7\sdk_workspace\standalone_bsp_0\Cortex_M3_0\include

3.2.2 Modifying Software Stack with Newly Generated BSP

To write the software stack or program that will run on the softcore, we will use the Keil IDE. The IDE is configured to use the previously generated BSP, and it has also been configured to compile all the device driver source files during compilation and linking. Our only task is to open the main.c file and start making modifications.

Double-click on the following file.

C:\FPGA\software\m3_for_arty_a7\Build_Keil\ m3_for_arty_a7.uvprojx

Which will open the Keil project, we locate the following file

C:\FPGA\software\m3_for_arty_a7\main\main.c,

And add these lines:

```
print ("Example design for Digilent A7 board\r\n");
print ("Compiling software with our own BSP\r\n");
```

This is a minimal change, but it will demonstrate that we have effectively modified the software stack, that the BSP is generated correctly, and that we can modify the hex file that gets programmed into the memory of our softcore. We click on build all, and Keil will compile and link with the following output, shown in Figure *3-5*.

```
linking...
Program Size: Code=10664 RO-data=1048 RW-data=204 ZI-data=5436
FromELF: creating hex file...
After Build - User command #1: make_hex_a7.bat
C:\FPGA\software\m3_for_arty_a7\Build_Keil>call fromelf --vhx --32x1 --output bram_a7.hex objects\m3_for_arty_a7.axf
C:\FPGA\software\m3_for_arty_a7\Build_Keil>copy  objects\m3_for_arty_a7.axf bram_a7.elf
    1 file(s) copied.
C:\FPGA\software\m3_for_arty_a7\Build_Keil>call fromelf --vhx --8x1  --output qspi_a7.hex objects\m3_for_arty_a7.axf
C:\FPGA\software\m3_for_arty_a7\Build_Keil>call fromelf --bin      --output qspi_a7.bin objects\m3_for_arty_a7.axf
C:\FPGA\software\m3_for_arty_a7\Build_Keil>copy bram_a7.* ..\..\..\hardware\m3_for_arty_a7\m3_for_arty_a7
bram_a7.elf
bram_a7.hex
    2 file(s) copied.
C:\FPGA\software\m3_for_arty_a7\Build_Keil>copy qspi_a7.hex ..\..\..\hardware\m3_for_arty_a7\testbench
    1 file(s) copied.
".\objects\m3_for_arty_a7.axf" - 0 Error(s), 24 Warning(s).
Build Time Elapsed:  00:00:01
```

Figure 3-5 – Compiling and Linking Output using the files from BSP. The process will generate a *bram_a7.hex* file that will be copied into the hardware directory.

It can be seen from the output that the software stack was compiled and linked correctly and that two files were generated: a *bram_a7.elf* and a bram_a7.hex. The first is used in conjunction with the testbench, allowing us to simulate the program's behavior using simulation models. The second file, *bram_a7.hex*, is the file that we will use to reprogram our bitstream file.

3.2.3 Programming the softcore BRAM memory

An FPGA softcore, such as an ARM Cortex-M3 implemented in an FPGA, utilizes Block RAM (BRAM), a dedicated on-chip memory, to store both program instructions and data. Upon power-

on or reset, the softcore begins execution at a predefined reset vector, typically address 0x0 in BRAM, which points to a bootloader or the start of the program. The softcore fetches and executes instructions sequentially from BRAM via a memory controller (e.g., AXI BRAM Controller).

To program the BRAM, two methods are used:

1. Directly using the Vivado Design Suite, opening the block design, double-clicking the ARM Cortex M3 Model, and selecting the Instruction Memory, as shown in Figure *3-6*. Once the file has been specified, we would need to run the synthesis run implementation and generate the bitstream again.

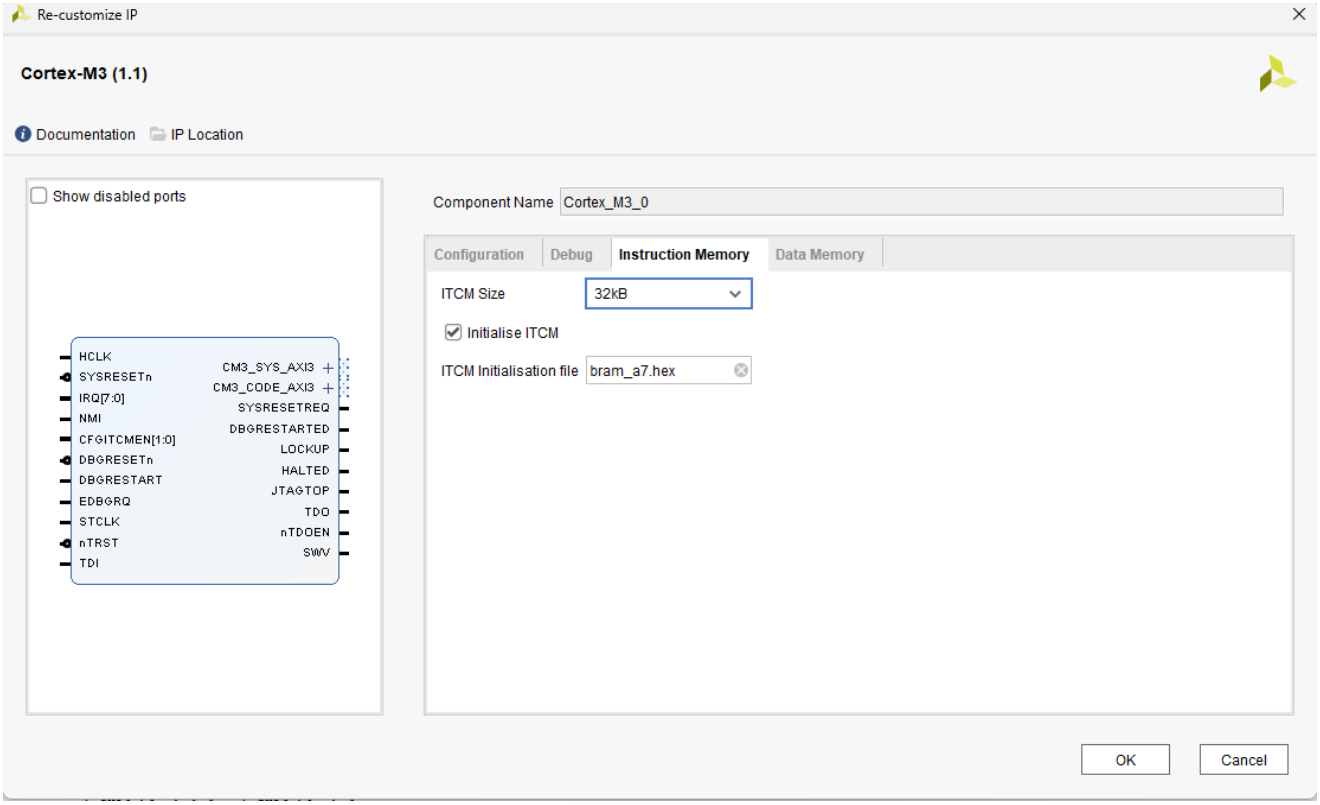

Figure 3-6 Configuring the hex file, which contains all the software stack instructions that the softcore will execute. We can configure the size of the memory stack as well as the file that we want to use.

2. A faster alternative for programming the Block RAM (BRAM) is provided by Vivado's default ARM Cortex M3 project. The method utilizes a Memory Map Information (MMI) file to update the FPGA bitstream using a TCL script that is executed from the Vivado TCL console or as a standalone batch file. For this method, after each hardware modification, we need to regenerate the MMI file. Later, after compiling a new program binary, we run

the batch file from the command prompt, and the bitstream file in the same folder will get modified with the latest binary.

3.2.4 Generating the Memory Mapped Interface (MMI) file

The first time that we want to use the MMI file to modify the bitstream, we are required to configure it for the specific FPGA that we are using. The first step is to modify the following file:

C:\FPGA\hardware\m3_for_arty_a7\m3_for_arty_a7\make_mmi_file.tcl

We look for the "set part" command and substitute it with the FPGA we are using.

```
# Set MMI output file name
set mmi_file "m3.mmi"
set part     "xc7aticsg324-1L"
```

Replace with "xc7a100tcsg324-1"

```
# Set MMI output file name
set mmi_file "m3.mmi"
set part     "xc7a100tcsg324-1"
```

Once the file is modified, we need to go back to Vivado and open the implemented design; we do the following:

Flow Navigation → Project Manager → Implementation -> Open Implemented Design

While the Implemented Design is open, we write in the TCL console and run the following command:

```
source make_mmi_file.TCL
```

The previous command will generate an MMI file that includes the hardware information.

3.2.5 Updating the Bitstream file and programming into the FPGA

Next, we need to combine this updated MMI file with bram_a7.hex to modify the bitstream. To do that, we'll open a command prompt in the same directory (Hardware) and run the batch program **make_prog_files.bat.**

The make_prog_files.bat will use the MMI file, the Hex file, and the previous bitstream to generate a new bitstream file in the exact location. After executing it, we can see an example output in *Figure 3-7*.

```
C:\FPGA\hardware\m3_for_arty_a7\m3_for_arty_a7\ make_prog_files.bat
C:\FPGA\hardware\m3_for_arty_a7\m3_for_arty_a7>call vivado -mode batch -source make_prog_files.tcl -notrace
****** Vivado v2019.1 (64-bit)
  **** SW Build 2552052 on Fri May 24 14:49:42 MDT 2019
  **** IP Build 2548770 on Fri May 24 18:01:18 MDT 2019
    ** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
source make_prog_files.tcl -notrace
  Running updatemem ...
****** updatemem v2019.1 (64-bit)
  **** SW Build 2552052 on Fri May 24 14:49:42 MDT 2019
  **** IP Build 2548770 on Fri May 24 18:01:18 MDT 2019
    ** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
source C:/Xilinx/Vivado/2019.1/scripts/updatemem/main.tcl -notrace
Command: update_mem -meminfo ./m3.mmi -data ./bram_a7.elf -proc dummy -bit
./m3_for_arty_a7.runs/impl_1/m3_for_arty_a7_wrapper.bit -out m3_for_arty_a7.bit -force
Loading bitfile ./m3_for_arty_a7.runs/impl_1/m3_for_arty_a7_wrapper.bit
Loading data files...
Updating memory content...
Creating bitstream...
Writing bitstream m3_for_arty_a7.bit...
0 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.
update_mem completed successfully
update_mem: Time (s): cpu = 00:00:03 ; elapsed = 00:00:08 . Memory (MB): peak = 522.461 ; gain = 430.941
INFO: [Common 17-206] Exiting updatemem at Tue Jun  4 17:46:55 2024...
************************************************
  m3_for_arty_a7.bit correctly generated
************************************************
Command: write_cfgmem -force -format MCS -size 16 -interface SPIx4 -loadbit {up 0 m3_for_arty_a7.bit} m3_for_arty_a7.mcs
Creating config memory files...
Creating bitstream load up from address 0x00000000
Loading bitfile m3_for_arty_a7.bit
Writing file ./m3_for_arty_a7.mcs
Writing log file ./m3_for_arty_a7.prm
==================================
Configuration Memory information
==================================
File Format        MCS
Interface          SPIX4
Size               16M
Start Address      0x00000000
End Address        0x00FFFFFF
Addr1         Addr2         Date                 File(s)
0x00000000    0x003A607B    Jun  4 17:46:55 2024    m3_for_arty_a7.bit
0 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.
write_cfgmem completed successfully
************************************************
  m3_for_arty_a7.mcs correctly generated
************************************************
```

Figure 3-7 – Generate a new Bitstream using the make_prog_files.bat batch file; this process requires the compiled and linked output of the memory-mapped interface (MMI) file, along with the previously synthesized and implemented bitstream.

After this operation, the bitstream file will get modified, and we will be ready to reprogram it into our FPGA.

C:/FPGA/hardware/m3_for_arty_a7/m3_for_arty_a7/m3_for_arty_a7/m3_for_arty_a7.bit

After we flash the new image, we get the output shown in

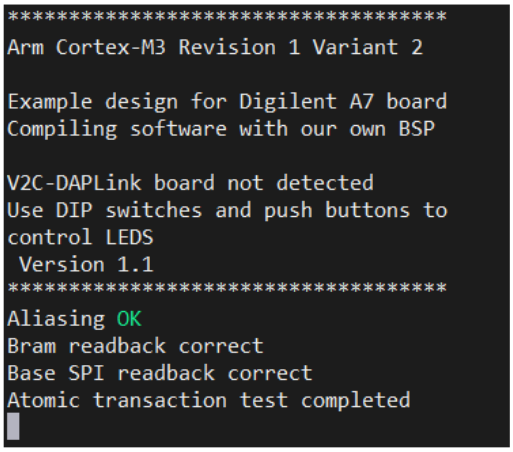

Figure 3-8 - After modifying the software stack, the FPGA serial output shows the original message from Figure 3-3, but it now includes the changes we did to the code, confirming that we have successfully modified the complete software stack.

3.3. Conclusions

In this chapter, we explore the resources provided by the ARM DesignStart program, which offers access to softcore IP cores, including the ARM Cortex-M3. We downloaded, synthesized, and implemented a Vivado project template for the Cortex-M3 softcore on an Xilinx FPGA. Using Vivado's Software Development Kit (SDK), we created a custom Board Support Package (BSP) that enables us to develop and modify the software stack. The BSP configures the Block RAM (BRAM) controller and memory map, allowing the softcore to execute programs stored in BRAM. To modify the software stack and reprogram BRAM, we compiled a new program binary using the Keil IDE. The binary is converted into a memory initialization file and integrated into the FPGA bitstream using one of two methods: (1) embedding it during bitstream generation in Vivado or (2) updating an existing bitstream with a TCL script, guided by a Memory Map Information (MMI) file. The MMI file, which maps BRAM locations, must be regenerated whenever the hardware design changes. With the modified MMI, the compilation/linking output, and the original bitstream, we were able to alter the software stack of the original project using the same hardware. In the next chapter, we will modify the hardware block to reduce it to a form that can be used to implement the hash algorithm.

4. Hardware Customization

In this chapter, we describe the hardware modifications to the default ARM Cortex-M3 softcore project, sourced from the ARM DesignStart program. As discussed in the previous chapter, the Board Support Package (BSP) and Cortex-M3 softcore impose constraints, such as memory mapping and peripheral requirements, which we must address. Our primary goal is to create a minimal system comprising the Cortex-M3 softcore, removing unnecessary components such as the AXI SPI controller, the Debug Access Port (DAP), and the general-purpose input/output (GPIO) ports. We will also show how to add a generic AXI block with read/write capabilities and how to interface this new block via the software stack.

4.1. The ARM Cortex IP Block

The ARM Cortex-M3 IP included in the default project offers the following configurable options.

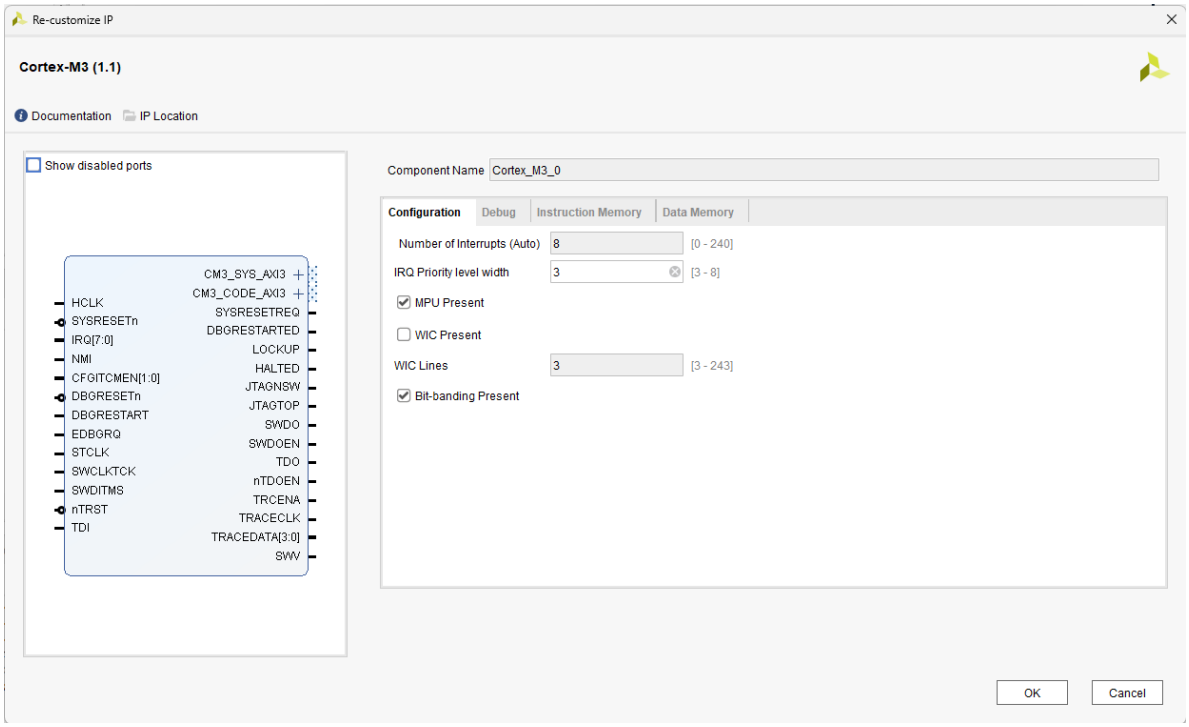

Figure 4-1 - ARM Cortex-M3 Default IP settings, Instruction, and Data Memory are set to 32 Kbytes, and Debug settings assume the DAP device; interruptions are set to a minimum of 8.

In the original Vivado design for the ARM Cortex-M3 softcore targeting the Digilent Arty-A7-100T FPGA board, debug signals are connected to a Debug Access Port (DAP) IP block, allowing connection to an external debug board via DAPLink (e.g., using JTAG or Serial Wire Debug (SWD) protocols). As we aim to implement a hash algorithm with minimal resource usage and do not have access to debugging equipment, we do not require debugging capability in the project. We thus decided to remove the DAP IP block from the Vivado block design, leaving many debug signals unconnected and reducing resource usage. We also configure the Cortex-M3 softcore IP settings in Vivado's IP Integrator to disable debug features, such as JTAG and SWD interfaces, while keeping all other settings (e.g., clock and memory interfaces) unchanged. The following TABLE *VI* lists the signals on the ARM Cortex M3 that are used:

TABLE VI – CORTEX-M3 SOC SOFTCORE SIGNALS. WHEN WE REMOVED THE DAP IP BLOCK, MOST OF THE DEBUG SIGNALS CEASED TO BE USED. ALL DESCRIPTIONS FROM [YIU – 19].

Name	**Used**	**Description**
DBGRESTARTED	No	Used in conjunction to confirm that the system has restarted after debugging operations or to execute a reset of the debug logic. It reflects the state of the Debug State Control Register
DBGRESTART	No	
DBGRESETn	No	
LOCKUP	No	Signals the following conditions: The processor is in a reset state, a Non-Maskable Interrupt (NMI) has occurred, or the processor is halted due to debug logic.
HALTED	No	When asserted, it indicates that the processor is halted (stopped). When de-asserted, it signifies that the processor is running.
JTAGNSW	No	Used for debugging and programming tasks.
JTAGTOP	No	"JTAG test output" provides test-related information during boundary scan testing
SWDO	No	Serial Wire Debug Output): SWDO serves debugging and trace purposes. The associated control signal is SWDOEN (Serial Wire Debug Output Enable), which enables or disables SWDO
TDO	No	TDO is explicitly used for boundary scan testing
nTDOEN	No	This control signal determines whether TDO is enabled or disabled.
TRCENA	No	activates the trace unit for instruction and data tracing.
TRACECLK	No	Trace the clk input
TRACEDATA	No	Trace data output

SMV	No	SMV indicates the address of the secure monitor exception handler.
EDBGRQ	No	EDBGRQ is used for debugging purposes.
STCLK	No	STCLK drives the SysTick timer.
SWCLKCTCK	No	Serial Wire Clock (SWCLK), used in Serial Wire Debug (SWD) communication.
SWDITMS	No	Serial Wire Data (SWDIO) for SWD communication.
nTRST	No	nTRST is employed during boundary scan testing.
TDI	No	TDI is part of the JTAG interface and is used for boundary scan testing.
NMI	No	Unlike regular interrupts, NMI cannot be disabled or masked. It typically handles critical events.
CM3_SYS_AXI3	Yes	The Advanced Extensible Interface (AXI) bus is compatible with many IP devices from Vivado. The interface has been added on top of the original AHB bus that all ARM Cortex devices initially had.
CM3_CODE_AXI3	No	The Advanced Extensible Interface (AXI) bus is compatible with many IP devices from Vivado. The interface has been added on top of the original AHB bus that all ARM Cortex devices initially had.
CFGITCMEN	Yes	Configuration bit for enabling the instruction Tightly Coupled Memory (ITCM). This bit configures the multiplexer that selects between external AXI blocks and internal ITCM; in our case, we choose ITCM.
SYSRESETREQ	Yes	signal stands for system reset request; this signal can be controlled by software and asserted to trigger a reset of the microcontroller
SYSRESETn	Yes	It is the global active-low system reset input. When asserted, it resets the entire system.
HCLK	Yes	High-speed clock input, which drives the core and peripheral clocks. This is the primary clock input; the original project contains a Vivado IP that generates a stable clock from the internal board clock. We will take a closer look at this block.
IRQ	Yes	Interrupt Request: Indicates an external interruption request from a peripheral or other sources. In our case, we will experience interruptions from the UART.

Two Tightly Coupled Memory (TCM) regions provide fast, low-latency storage: the Instruction Tightly Coupled Memory (ITCM) for program instructions and the Data Tightly Coupled Memory (DTCM) for data. In an FPGA design, these TCMs are typically mapped to Block RAM (BRAM), configured via Vivado's IP Integrator. The TCMs are aliased to specific address ranges in the Cortex-M3's memory map (e.g., ITCM at 0x00000000, DTCM at 0x20000000). Memory requests to addresses outside these ranges are routed through an AHB-to-AXI bridge, which converts the

Cortex-M3's AMBA AHB-Lite bus to AXI-4, enabling access to external devices such as AXI peripherals or off-chip memory. In our application, we point both the instruction and data to point internally.

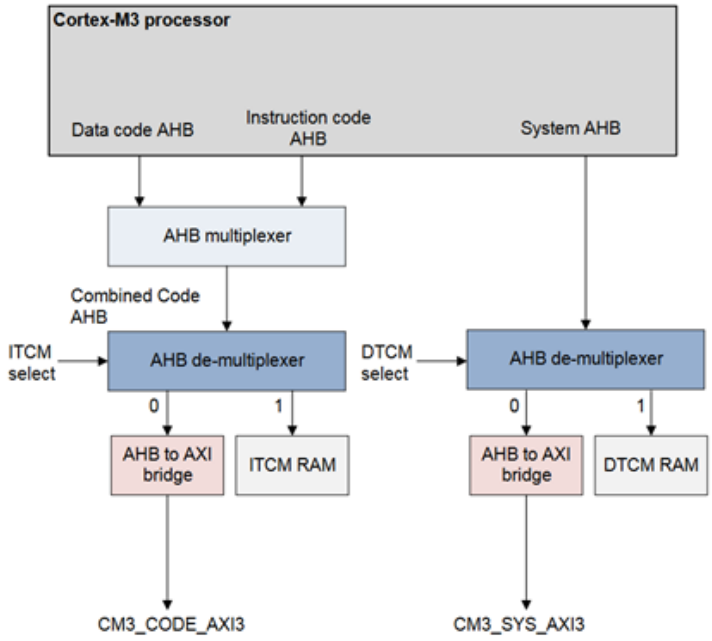

Figure 4-2 - Internal memory processing in the ARM Cortex-M3 softcore allows us to select between internal and external memory usage through the ITCM and TCM options. From [Yiu – 19].

4.2. Clocks and Resets

The Clock and Resets block manages clock and reset signals. The system clock signal (sys_clock) is routed to the Arty-A7-100T's 100 MHz onboard oscillator. The Clock and Resets block uses a Phase-Locked Loop (PLL) to generate a 100 MHz output signal (clk_cpu), which drives the Cortex-M3 softcore. The system reset signal, connected to the board's external reset pin (e.g., a reset button), is distributed to AXI peripherals and the Cortex-M3 via the Processor System Reset IP. The Cortex-M3's SYSRESETREQ signal, routed from the softcore, enables software-initiated resets, which are processed by the Clock and Resets block to reset the system.

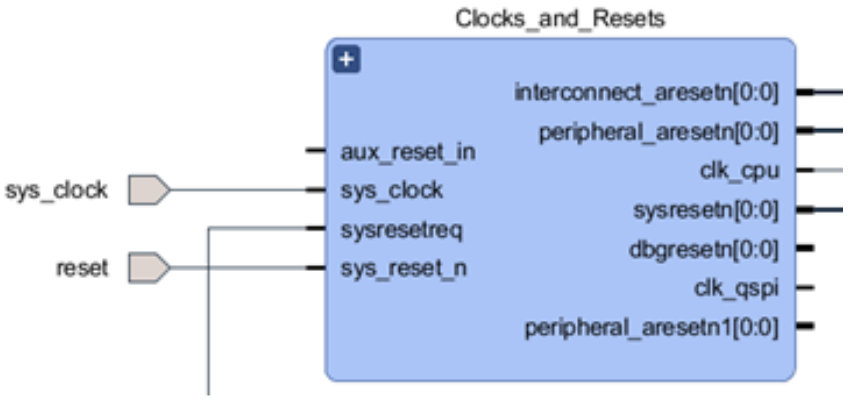

Figure 4-3 - The Clock and Reset block manages the 100 MHz clock for the ARM Cortex M3 as well as the reset signals for the ARM Cortex and all AXI peripherals. The reset can be generated externally by the development board reset pin or through the sysresetreq from the software stack.

4.3. Configuring the Instruction Throughly Coupled Memory

The CFGITCMEN[0] input signal enables the Instruction Tightly Coupled Memory (ITCM) to use internal BRAM memory when set to logic 1. In contrast, the Data Tightly Coupled Memory (DTCM) internal usage is typically enabled via CFGITCMEM[1], as shown in Figure *4-4* - Connecting the CFGITCMEN [1:0] signal to select internal memory usage via ITCM and DTCM selection.

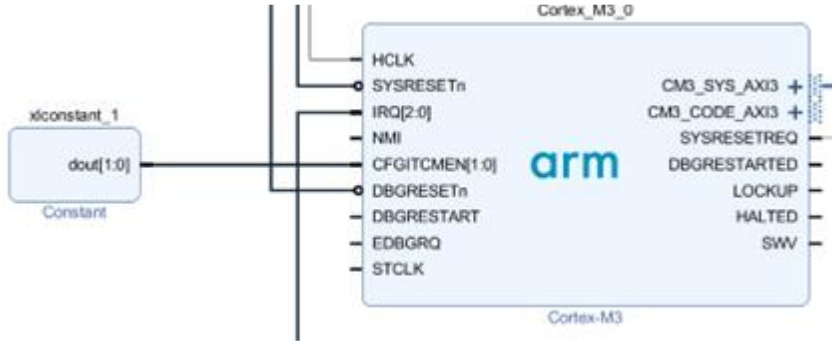

Figure 4-4 - Connecting the CFGITCMEN [1:0] signal to select internal memory usage via ITCM and DTCM selection.

4.4. Configuring Interrupts

The ARM Cortex-M3 Softcore has a minimum of 8 external interrupt inputs. After removing the GPIO, SPI, and Debug blocks, we only require one external interrupt coming from the UART block. We concatenate the single valid interrupt from the UART with a constant value to generate an 8-bit signal that we can pass to the IRQ [7:0] input.

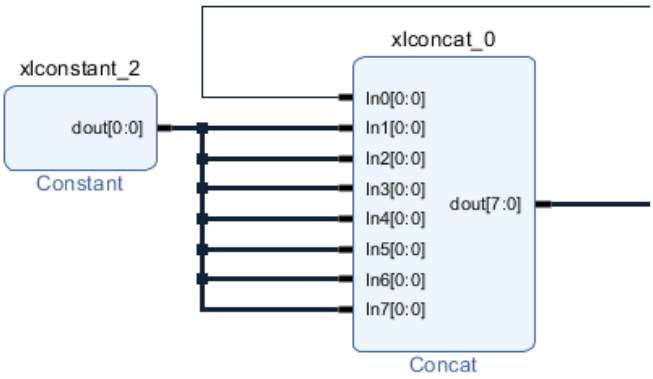

Figure 4-5 – Interruption configuration: the interruption connected to the ln0 is coming from the AXI Uart; all other interrupts are connected to a fixed value of zero. The output is routed directly to the IRQ input of the ARM Cortex-M3 IP.

4.5. Uart and Bram AXI Blocks

The Uart and Bram blocks remain unchanged, as they are directly connected to the AXI Hub, which converts between the Advanced High-Performance Bus (AHB) and the Advanced Extensible Interface (AXI).

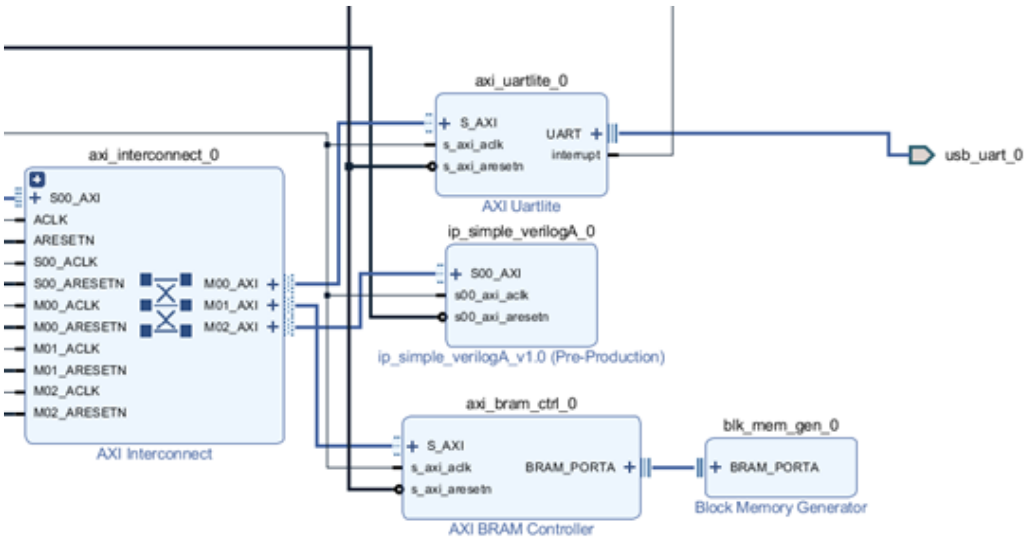

Figure 4-6 - AXI interconnect IP connects UART, BRAM, and a custom AXI block to the ARM CPU. The interrupt signal from the UART block goes into the concatenated block.

4.6. AXI Custom Block

Up to this point, we have only removed unnecessary blocks: SPI, Debug, and GPIO. We now need to create a new custom block that can assist in executing those parallel parts of the Hash algorithm to improve performance.

The custom AXI4 peripheral was created using Vivado's IP Integrator. In Vivado, we select:

Tools → Create and Package New IP

then choose:

Create a New AXI4 Peripheral.

After specifying the IP's name and location, we configure the AXI4 interface type to suit our application. For our design, we selected AXI4-Lite due to its minimal resource requirements compared to AXI4-Full and AXI4-Stream. The main disadvantage is that it lacks flexible data width, meaning the smallest register size we can use is 32 bits.

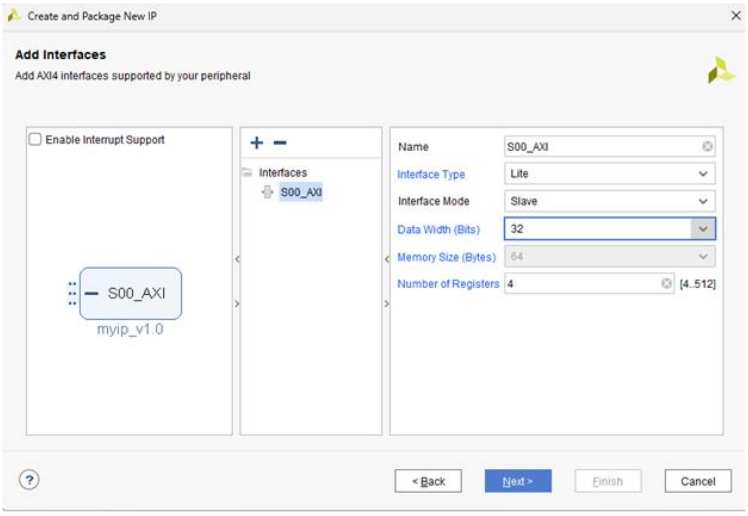

Figure 4-7 - Configuring AXI4 Custom IP, the minimum number of registers is four; we can only use 32-bit size registers.

Once the configuration of the parameters is done, we save the IP to the block design in Vivado's IP Integrator. Vivado's Create and Package New IP wizard generates a custom AXI4-Lite peripheral with minimal functionality. The peripheral's primary input/output mechanism relies on four 32-bit memory-mapped registers, which facilitate communication between the block and the softcore. In the next chapter, we enhance the functionality of these registers by modifying the AXI4-Lite peripheral's HDL code.

Once the block is generated, we validate our design by clicking on:

File → Save Block Design and Tools → Validate Design

The Validate functionality will check the block and provide us with feedback on connections that need to be fixed or assignments to the constraint board file that require attention. In our case, only the UART block needed to be connected to the USB UART port on the Arty A7 board. Other connections to the board are the reset signal and the 100 MHz clock. Once the Validation functionality finishes, we should get "Validation Successful.". We proceed with the synthesis, implementation, and generation of the Bitstream file.

Flow Navigation → Project Manager → Synthesis -> Run Synthesis

Flow Navigation → Project Manager → Implementation -> Run Implementation

Flow Navigation → Project Manager → Program and Debug → Generate Bitstream

Once these operations finish successfully, we proceed to generate the MMI file; we click in:

Flow Navigation → Project Manager → Implementation -> Open Implemented Design

In the TCL Console,

run make_mmi_file.tcl

Once we have the MMI file, we can proceed to generate the BSP with the updated memory map of the connected devices. We export the hardware definition file with the command:

File → Export Hardware

Into the known location:

C:\FPGA\software\m3_for_arty_a7\sdk_workspace

and click in

File → Launch SDK

Once the SDK tool opens, we generate the BSP per the instructions detailed in the section 3.2.1. The modified hardware block can be seen in

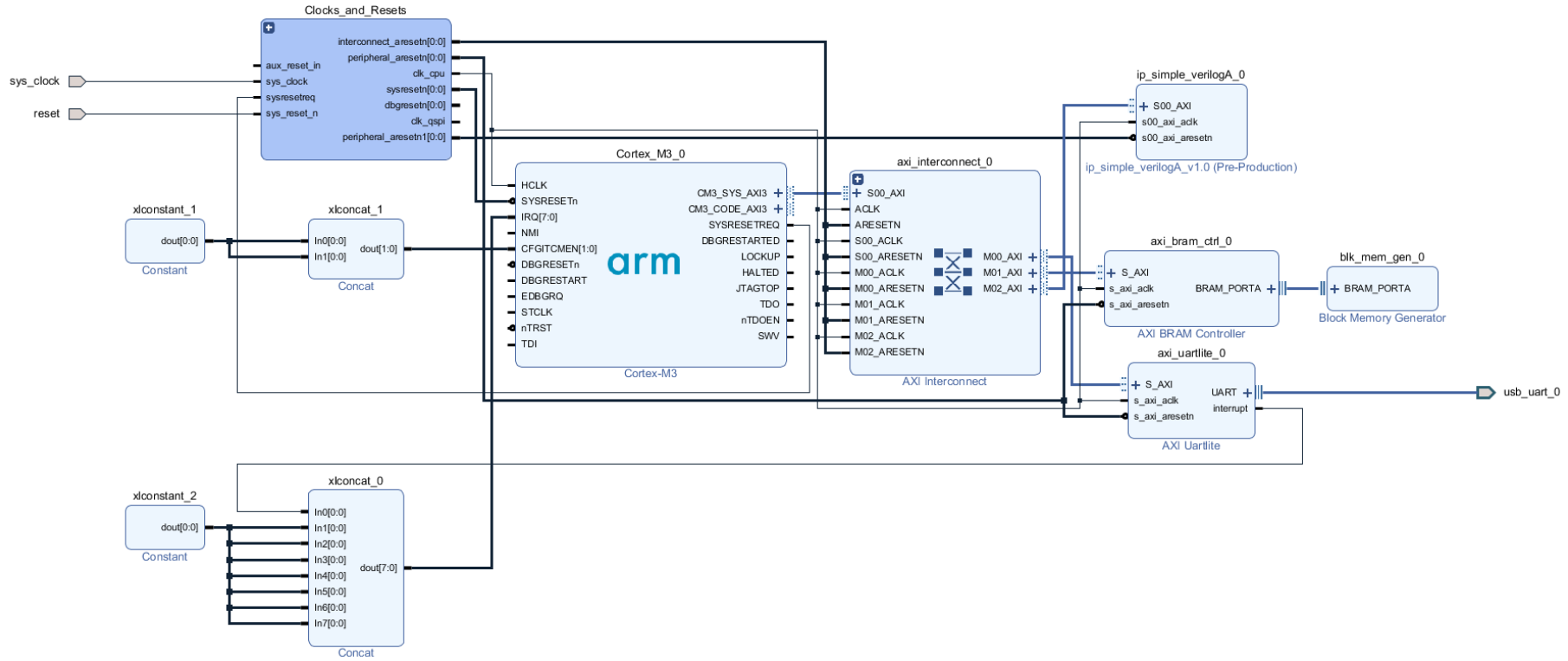

Figure 4-8 - ARM Cortex M3 project featuring a functional UART, BRAM memory, and a custom AXI block. Unnecessary blocks such as SPI, DAP Debug, and GPIO were removed.

4.7. Updating BSP for modified hardware

Once we generate the new BSP for the updated hardware, the SDK creates a device driver for our new AXI block and assigns a base address of 0x44A00000 in memory. See Figure 4-9.

Address Map for processor Cortex_M3_0

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
ip_simple_verilogA_0	0x44a00000	0x44a0ffff	S00_AXI	REGISTER

Figure 4-9 - Memory Address of the new AXI block with 4 R/W registers mapped on base address 0x44a00000.

4.8. Updating the main program to make calls to the new device driver

We now open the Keil IDE to modify the main program. Since we do not have the GPIO or SPI blocks, we can remove the code in main.c that calls these devices. Additionally, in Keil, under Project, we double-click on m3_for_arty7 to open a menu and select Manage Project Items.

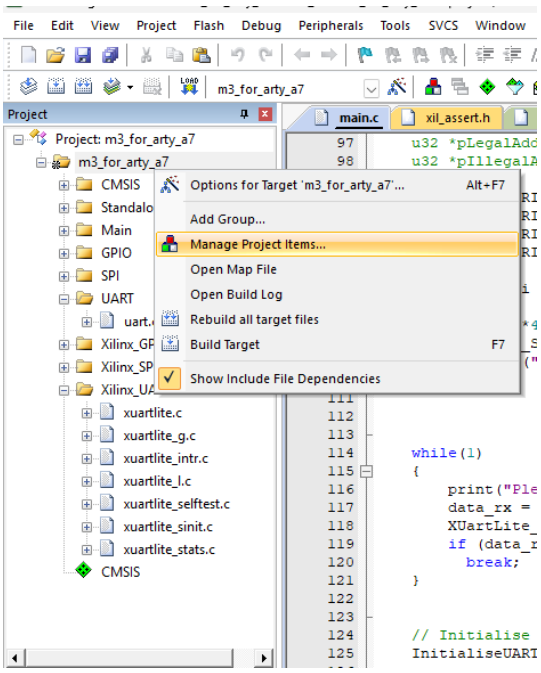

Figure 4-10 – Managing Project Items allows us to add or remove any libraries that we do not require. In this case, we need to remove the SPI and GPIO libraries so that we do not encounter any errors or warnings due to missing device drivers.

When you click this, a new window opens showing the groups. We will remove the GPIO and SPI groups from this list. This informs Keil that we don't need to compile specific libraries in the default project that were previously included. As previously mentioned, the BSP added to the Keil libraries contains the new device driver for the block that we created. The mapping can be found in the header file: xparameters.h. See Figure *4-11*

```
/******************************************************************/

/* Definitions for driver IP_SIMPLE_VERILOGA */
#define XPAR_IP_SIMPLE_VERILOGA_NUM_INSTANCES 1

/* Definitions for peripheral IP_SIMPLE_VERILOGA_0 */
#define XPAR_IP_SIMPLE_VERILOGA_0_DEVICE_ID 0
#define XPAR_IP_SIMPLE_VERILOGA_0_S00_AXI_BASEADDR 0x44A00000
#define XPAR_IP_SIMPLE_VERILOGA_0_S00_AXI_HIGHADDR 0x44A0FFFF
```

Figure 4-11 - Definitions are located in the xparameters.h file utilized by the Keil tool for compilation and linking; here, we observe that the AXI block is mapped to the 0x44A0000 address.

In the main.c file, we add the SelfTest function to validate the new block.

```
print ("FPGA compilation 2   \r\n");
flag = AXI_CUSTOM_MODULE_Reg_SelfTest((void*)0x44a00000);
if (flag == XST_SUCCESS)
{
        print("Test case passed for slave \r\n");
}
```

The function writes a predefined test value to all four registers. Then, it reads the register values to verify that they match the written values, ensuring the proper operation of the AXI4-Lite interface and peripheral.

We now compile and link the main.c file and generate a new hex file. We now follow the steps outlined in the section 3.2.2 and 3.2.3 to generate the MMI file, update the Bitstream file, and program it into the FPGA. After programming the new Bitstream file, we get the output shown in

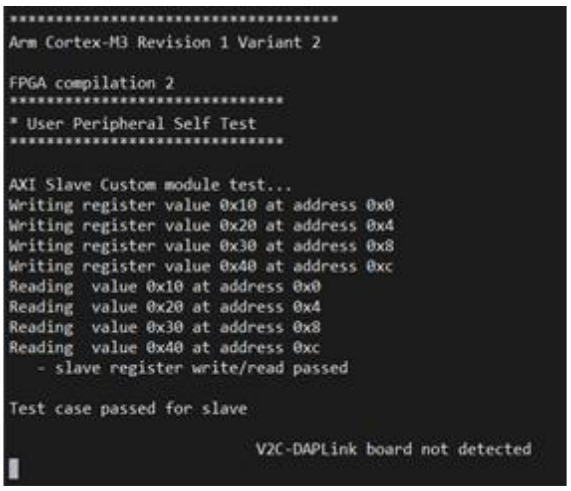

Figure 4-12 The Self-Test Program is executed on the ARTY-a7-100T development board. We observe that the ARM Cortex first writes a known value into each register and then reads it back to validate the AXI protocol and ensure minimum functionality.

4.9. General Recipe to Modify the hardware and software of the default ARM Cortex-M3 project

We now present a list of the main steps needed to modify the default ARM Cortex-M3 Project successfully.

1. Start with the default ARM Cortex-M3 default project
2. Modify Hardware
    a. Remove blocks that are not needed
    b. Add new blocks
    c. Add/Remove connections to the board constraint file.
3. Validate the design (this step will run the device mapper)
    a. If an error is found, return to step 2 and correct it.
4. Synthesize and implement the new hardware
    a. If an error is found, return to step 2 and correct it.
5. Generate Bitstream.
6. While the Implementation Windows is open, generate an MMI file.
7. Export Hardware and Open the SDK tool.
8. In the SDK, validate that all devices are included in the memory map.
9. Generate new BSP with OS Version set to 6.7

10. Validate that UART ports are mapped correctly.
11. Copy xpseudo_asm_rvct.c and xpseudo_asm_rvct.h from vivado\ Arm_sw_repository directory.
12. Open the Keil IDE Software.
13. Modify the software.
14. If new libraries are needed, include them in the project settings so that the compiler and linker have access to any new libraries.
15. Compile and Link (BUILD).
16. Hex files will be copied to the Hardware directory.
17. Run make_prog_files.bat; this will update the Bitstream file.
18. Program the new bitstream file into the FPGA and verify the results.

4.10. Conclusion

In this chapter, we modified the original ARM Cortex-M3 project by reviewing it and identifying components unnecessary for our current design. We removed the SPI, Debug, and GPIO AXI blocks, as they were not needed. We added a new AXI block to better understand the integration process into the ARM Cortex-M3 project. After testing the addition and removal of hardware and software elements from the original project, we created a list. This list can be followed by readers who may want to use the ARM Cortex-M3. We now have a working environment that can integrate the hash blocks. In the next chapter, we will apply what we learned in the last three chapters to integrate a hash algorithm into the ARM Cortex-M3 project.

5. SHA3-512 Implementation and Results

This chapter describes two implementations in the ARM Cortex M3 IP core:

1) A SHA-3 512 algorithm executing in the ARM Cortex M3 softcore, which we will call SHA-3 512 Software Implementation.
2) A SHA-3 512 algorithm implementing the permutation stage in a hardware IP module; this is a hybrid solution, which we will call SHA-3 512 Hardware Implementation.

Furthermore, both implementations are validated by running NIST-approved [NIST -15B] input vectors from 0 to 72 bytes, comparing the output products against the known correct SHA values. Additionally, for each implementation, the latency and throughput are measured. The power and FPGA resources are calculated using Vivado tools.

5.1. SHA-3 512 Software Implementation

To implement the SHA3-512 algorithm in the ARM Cortex-M3 softcore, we require the source code, which can be obtained freely from the Extended Keccak Code Package [XKCP – 23]. The advantage of using this sample source code is that designers have validated and benchmarked it since the initial NIST approval submission in 2008. Additionally, since the code is open source, multiple security experts have access to analyze its cryptography and improve it.

We worked on adapting a version that implements a 32-bit optimized version of the Keccak hash function, specifically tailored for the Keccak sponge construction.

The key components of the SHA3 source code implementation are:

a) Initialization – Sets up the 1600-bit state to all zeros and initializes round constants and rotation offsets.
b) Absorbing phase – Input data is XORed into the state, followed by a permutation if a whole block of size rate = 72 bytes has been reached.
c) Squeezing Phase – Output is extracted from the state at a rate of 72 bytes.

In the following section, we provide more details on these specific workflows. For a more precise explanation of these flows, please review Section 2.2.

5.1.1 Initialization Phase

The process begins with the Initialization Phase; the sponge state array is initialized to null, the Round Constants and Rho Offsets are calculated, and the index and some other variables are initialized. For a 32-bit architecture, a fifty-element array of 32-bit unsigned integers is employed. Figure *1-1* Provides a simplified illustration of these steps.

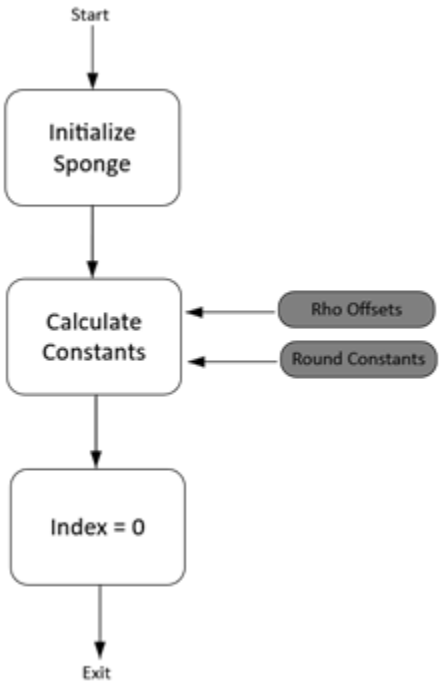

Figure 5-1 - Initialization Steps: as implemented in the SpongeInitialize function, the sponge array, constants, and Index are initialized.

5.1.2 Absorption phase implemented in the Sponge Absorb function.

The Sponge Absorb function is a critical component of the SHA-3 sponge construction. It handles the main absorbing phase of the sponge, where the data is incorporated into the 1600-bit state. This function processes the input message in blocks of size equal to the rate in bytes. In our case, 72 bytes. When a full-rate-sized block is absorbed, it will XOR the complete block into the state and apply a Keccak permutation. When the amount of initial or remaining data is insufficient to fill a complete 72-byte block, it performs a partial block absorption, where it XORs the partial block

but does not trigger the sponge permute function, as it cannot complete the whole block of size rate. **Error! Reference source not found.** shows a simplified diagram of the SpongeAbsorb f unction.

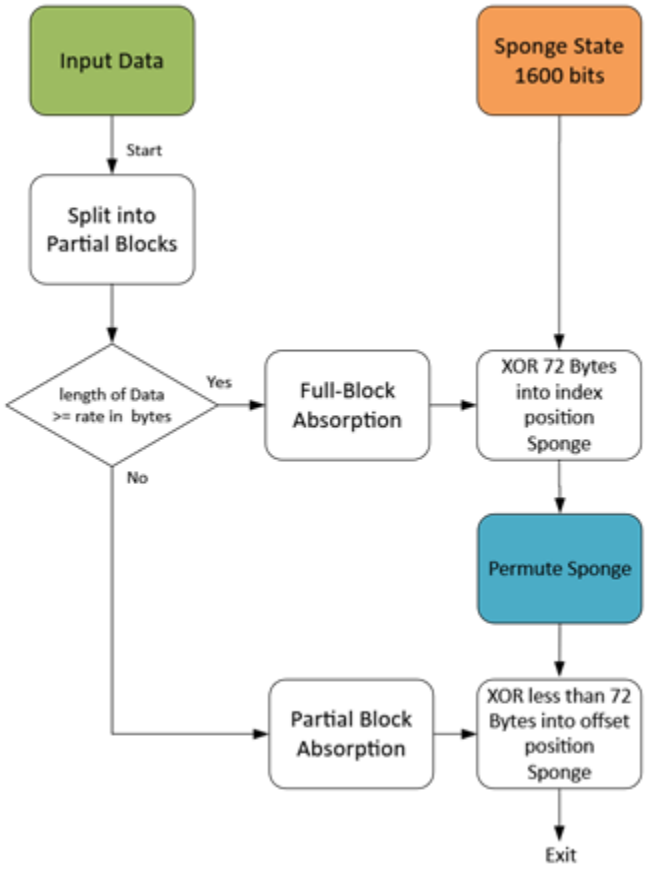

Figure 5-2 - Absorption is implemented by the SpongeAbsorb function; its main job is to break the Input message into complete and partial blocks. If they are whole blocks, they can then be passed to the sponge permutation function.

After exiting the SpongeAbsorb function, we have absorbed whole blocks of 72 bytes or partial blocks smaller than 72 bytes into the sponge. We now need to apply the suffix appending and padding rule insertion. The SpongeAbsorbLastFewBits handles the absorption of the final few bits of a message when its length (in bits) is not byte-aligned. It also applies the necessary padding to complete the absorption phase and transitions the sponge to the squeezing phase. The function operates as follows: it begins by absorbing delimitedData into the sponge's state, where delimitedData is a single byte encapsulating the remaining 1 to 7 bits of the message plus the suffix 0x06 (SHA3-512 specifies a fixed 0x06 suffix). It then checks if the first padding bit is in the most

significant bit of the byte and if the current block is almost complete. If both conditions are true, the first padding bit fills the last byte of the block, prompting a permutation to start a new block for the second padding bit. Otherwise, it absorbs the second padding byte (0x80) at position state[rateInBytes-1], with zeros implicitly present between the first and second padding bits. After these operations, the function performs a final permutation. Finally, it resets the internal state and sets the mode to squeeze. Figure *5-3* shows a simplified representation of this flow.

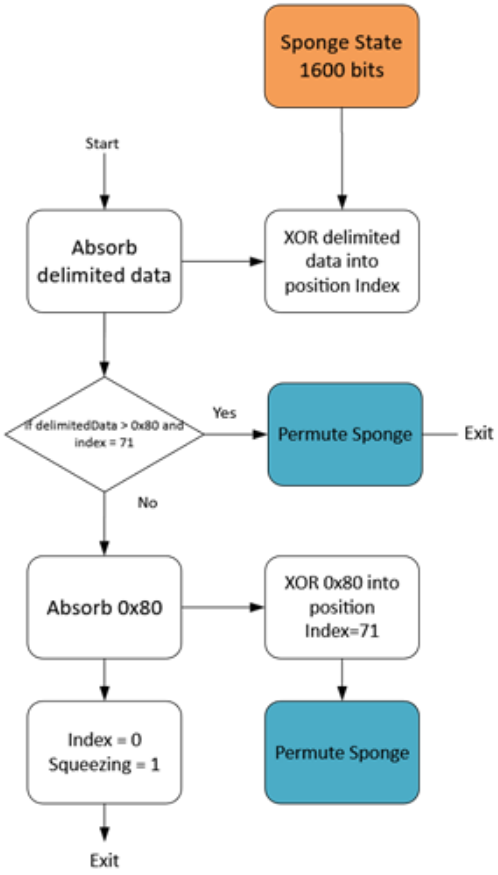

Figure 5-3 - Absorption of the last bits is implemented by the SpongeAbsorbLastFewBits function, which primarily breaks down the absorption of the previous non-aligned bits and absorbs the suffix and padding bytes.

5.1.1 Squeezing Phase

The squeezing phase is the part where the sponge extracts data from the 1600-bit state. Depending on the algorithm, additional permutations may be necessary; for the SHA-3 512-bit algorithm, this

stage is straightforward. The function will extract 64 bytes from the sponge, so the dataByteLen is always smaller than the rate (64 bits < 72 bits). If another algorithm requires additional bytes, it will need to enter another permutation and then extract the remaining bytes until it meets the required output. In Figure *5-4*, we can see the squeezing phase.

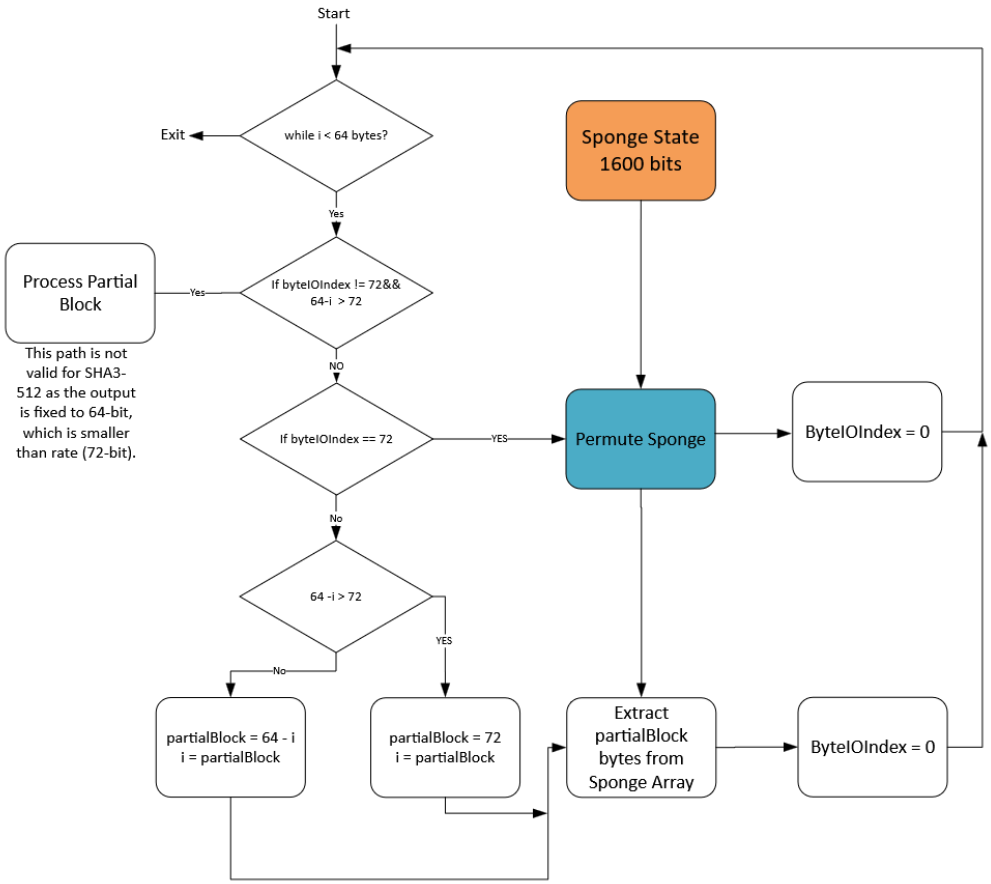

Figure 5-4 - Squeezing of 64 bits of data from the sponge array implemented by the SpongeSqueeze function.

5.1.1 The Sponge Permutation Function Operations

The core of the SHA algorithm is the data shuffle applied to the inputs during the Keccak permutation phase; the sponge is passed through different state mappings. For the SHA-3 512 algorithm, it is specified that 24 iterations of the same process. The Rho offsets and Round Constants calculated in the initialization phase are used. In Figure *5-5* we can see the full permutation flow.

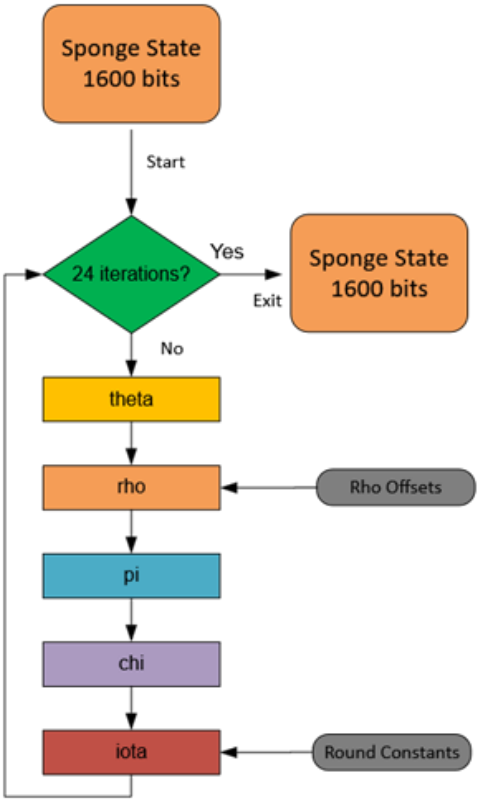

Figure 5-5 - Sponge Permutation function block: For SHA3-512, we use the Sponge State Array as input to the Sponge Permutation function block. After 24 iterations, it modifies the Sponge State Array. Rho Offsets and Round Constants are inputs to the operations.

5.1.1 Interleave and de-interleave functions for 32-bit implementation of 64-bit oriented Keccak state mappings.

Since we are implementing the algorithm in a 32-bit architecture, we require a way to represent the 1600-bit state using 32-bit-sized registers. To achieve this, we employ a technique called Bit Interleaving, a coding method that represents a w-bit word as an array of CPU words. This method is described in [Bertoni et al, - 12], it has the following parameters:

- w: the length of the lane in bits
- m: the length of the CPU word in bits
- s: the interleaving factor, calculated as s = w/m.

To make it more straightforward, let's suppose w = 64 bits and our architecture has 32-bit words. We are required to use an interleaving factor of s = 2. A 64-bit value **A** is split into two 32-bit

values: $\mathbf{A_{even}}$ (even-indexed bits) and $\mathbf{A_{odd}}$ (odd-indexed bits). Another 64-bit value, **B**, is divided into two 32-bit values, $\mathbf{B_{even}}$ and $\mathbf{B_{odd}}$. The XOR operation will then be $\mathbf{A_{even}} \oplus \mathbf{B_{even}}$ and $\mathbf{A_{odd}} \oplus \mathbf{B_{odd}}$. This produces an interleaved XOR result, which is then recombined to obtain the full 64-bit value. We take $\mathbf{A_{even}} \oplus \mathbf{B_{even}}$ for the 0, 2, 4, 6,… bits and $\mathbf{A_{odd}} \oplus \mathbf{B_{odd}}$ for the 1, 3, 5, 7,… bits.
The following is an example, let:
A = 0xDEADBEEFBAD4F00D
B =0xFEEDFACECAFEBEEF
We first extract the bits in positions 0, 2, 4, 6, and 8 to obtain $\mathbf{A_{even}}$, and then extract the bits in positions 1, 3, 5, 7, and so on to obtain $\mathbf{A_{odd}}$.
A = 1101111010101101101111101110111110111010110101001111000000001101
$\mathbf{A_{even}}$ = 11100011011010110100001111000011
$\mathbf{A_{odd}}$ = 10111110111111111111111011000010
B = 1111111011101101111110101100111011001010111111101011111011101111
$\mathbf{B_{even}}$ = 11101011110010101000111001101011
$\mathbf{B_{odd}}$= 11111110111110111011111111111111
We XOR the even parts and odd parts, and we get:
$\mathbf{A_{even}} \oplus \mathbf{B_{even}}$: 00001000101000011100000010101000
$\mathbf{A_{odd}} \oplus \mathbf{B_{odd}}$: 01000000000001000100011100111101
We now reconstruct the values into a single 64-bit value. We take bits 0, 2, 4, 6, and so on from the even result and bits 1, 3, 5, 7, and so on from the odd result.
$\mathbf{A} \oplus \mathbf{B}$ = 0010000001000000010001000010000101110000001010100100111011100010

Returning to the implementation of the sponge state, it is stored as 50 unsigned int (uint32_t) words, interleaved. Interleaving is required during the absorption (to convert the input), and de-interleaving is necessary during the squeezing (to convert the output). All permutation steps are applied to the interleaved state.

5.1.2 Debug Functions used to validate intermediate steps.

There are some debug print functions that permit us to compare internal states after each phase. This was particularly useful when the initial implementation was carried out, and some states

needed to be compared against the expected values. These functions were helpful when some of the final outputs were incorrect. In those cases, we enabled the debug messages and compared them against 64-bit implementations run on a computer. In these exceptional cases, we had to compare outputs from each state's mappings.

5.1.3 FPGA implementation of the SHA-3 512 Software Implementation

To test the SHA-3 512 Software Implementation, we used the hardware shown in Figure *4-8*We added an AXI Timer, which will be used for performance latency and throughput measurements. This timer records the start and end times of events; the latency is calculated by subtracting the start time from the end time. The FPGA is running at a 100 MHz clock frequency, and the AXI Timer counts clock cycles (10 ns). The hardware block implementation is shown in Figure *5-6*.

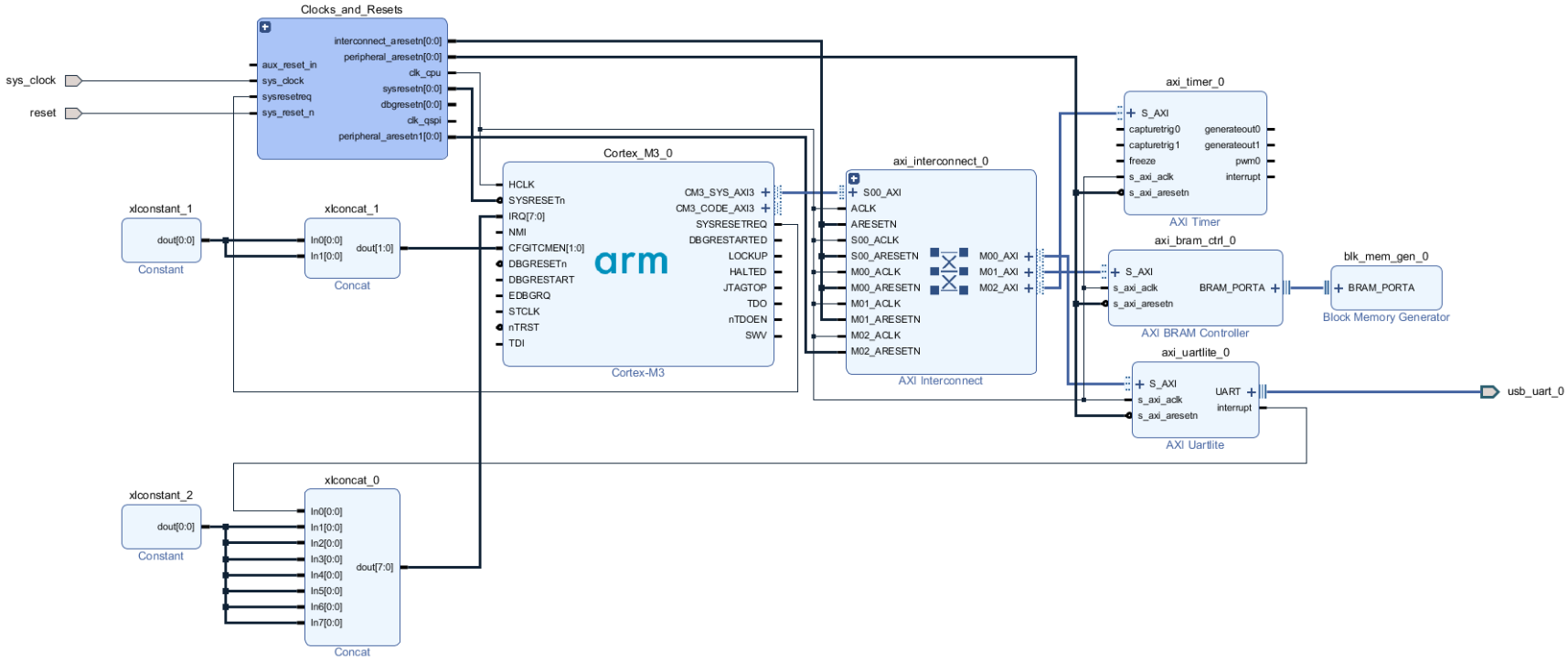

Figure 5-6 - SHA-3 512 Software Implementation: This is the implemented configuration running the software implementation. The only addition to the reduced project build in Chapter 4 is the AXI Timer, which will help characterize the solution.

The FPGA synthesis implementation is shown in Figure 5-7

Figure 5-7 - FPGA routing resources of the SHA-3 512 Software Implementation on an xc7a100tcsg324-1 FPGA.

The synthesis implementation report for the xc7a100tcsg324-1 FPGA indicates a total Slice LUTs usage of 11,418. See Figure *5-8*,

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	11418	0	63400	18.01
LUT as Logic	11363	0	63400	17.92
LUT as Memory	55	0	19000	0.29
LUT as Distributed RAM	0	0		
LUT as Shift Register	55	0		
Slice Registers	3518	0	126800	2.77
Register as Flip Flop	3518	0	126800	2.77
Register as Latch	0	0	126800	0.00
F7 Muxes	169	0	31700	0.53
F8 Muxes	42	0	15850	0.26

Figure 5-8 - Utilization report for the SHA-3 512 Software Implementation, a total utilization of 18.01%

The power report of the SHA-3 512 synthesis is shown in Figure *5-9*

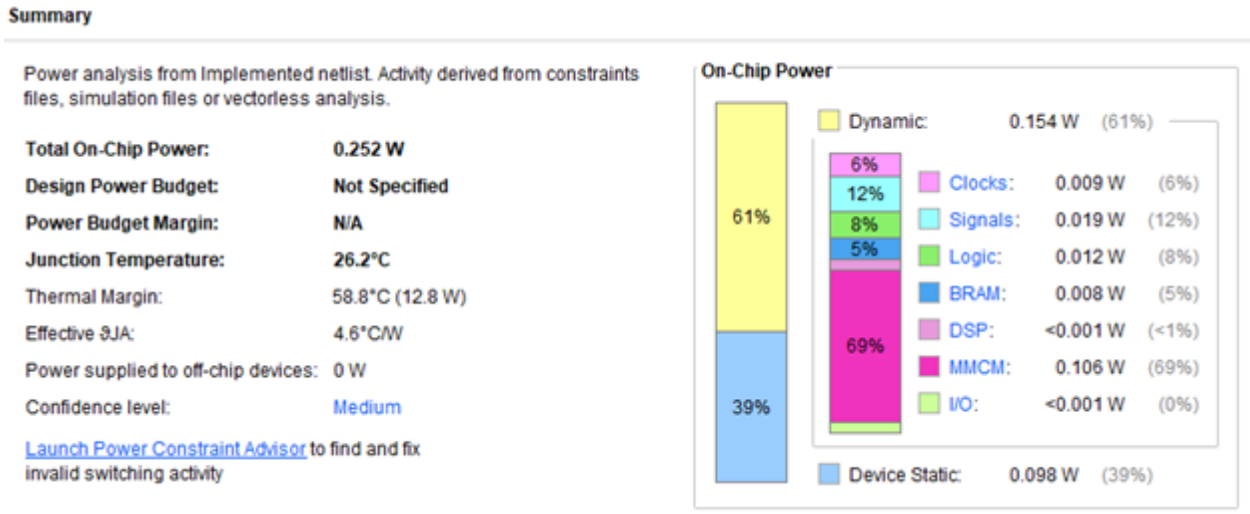

Figure 5-9 - Power Usage Estimation of the SHA-3 512 Software Implementation.

As shown in Figure *5-9*, the total on-chip power is estimated to be 0.252 watts. This estimation is based on the model's calculation of switching power. We also attempted to measure the power on the development board using a USB power meter, but the board didn't show any power spikes during software operation.

5.2. SHA-3 512 Hardware Implementation

For the SHA-3 512-bit Hardware implementation, we used a VHDL model provided by the Keccak Team as the basis [Keccak – 12] [Bertoni et al, - 12]. The provided models use a rate of 1024 and a capacity of 576, or a rate of 40 and a capacity of 160, which differ from our requirements. Additionally, three models are provided:

a. a performance-optimized model with highly parallel structures that consume more area,
b. a mid-range model implementing folded optimization.
c. Low-area coprocessor that uses internal memory as the sponge state.

For this work, we chose to implement the first option because it implements the permutation in a more straightforward and parallel manner.

5.2.1 The Keccak Team VHDL High-Speed Reference Model

The Keccak_round VHDL module implements one round of the Keccak-f[1600] permutation, transforming a 1600-bit state through the Theta, Rho, Pi, Chi, and Iota mapping functions. Each mapping function is implemented as a combinational logic block, with operations (XOR, AND, NOT, and rotations) performed in parallel across all bits of the 1600-bit state. This parallelism maximizes throughput but increases the required silicon area. The mapping stages are chained sequentially in the data flow, forming a pipeline of combinational logic blocks, where the output of one mapping feeds directly into the next, all within a single combinational path. Since there is no clock signal, great care must be taken to avoid using the output until the data has been processed. The Keccak_buffer module is a clocked buffer that manages input and output data for the Keccak sponge function. It operates in input/output modes, collecting a 1024-bit input from the Keccak Round module and XORing 1024-bit into the Sponge State. It is also capable of providing a 64-bit hash output once the process is complete. Both the Keccak round and Keccak buffer modules work in tandem with synchronous processes handling the absorb and squeeze phases. It processes 64-bit Input, buffers them based on an internal input/output mode, XORs the first 1024 bits into the Sponge State, and applies the Keccak-f[1600] permutation after 24 rounds. It then extracts 1024 bits back into the IO buffer, which will be the 64-bit output. It operates synchronously, controlled by clock, reset, and start signals. See Figure *5-10*

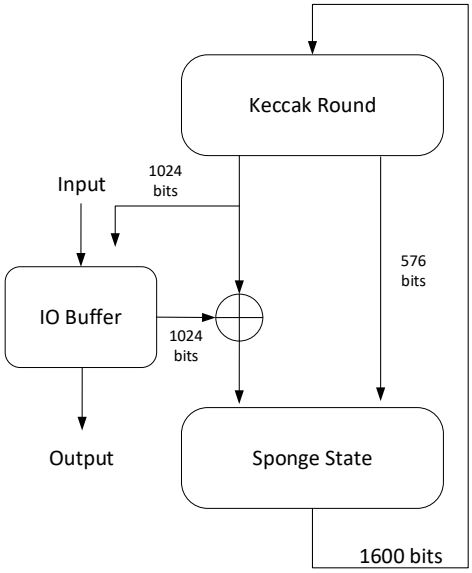

Figure 5-10 - Original SHA3-256 reference implementation block diagram. Taken from [Bertoni et al, - 12]

5.2.2 SHA3-256 reference model modifications

Our initial strategy was to modify the sizes of the input and output vectors. The idea was to modify the rate from 1024 bits to 576 bits, changing the 1024 buses to 576 bits and the 576 buses to 1024 bits. However, after several attempts, we were unable to resolve all timing issues. Since the Keccak Round is not synchronous, there were some race conditions between each external round managed by the IO Buffer model and the Keccak Round. Instead, we opted to reuse the existing Keccak_round module as is and pass the different input states directly. For SHA-3-512, which has a 576-bit rate (72 bytes), we input nine 64-bit lanes (init_0 to init_8). The module applies the Keccak-f[1600] permutation 24 times and outputs eight 64-bit lanes (out_0 to out_7), producing a 512-bit hash. The input bits are XORed with the current values in the sponge. Thus, it can also be used for non-zero permutations, as in cases with long input vectors. It operates synchronously, controlled by clock, reset, and start signals, as well as a completion signal with a done output. The main advantages of these modifications are:

- Reuse of the original Keccak_round module for the sponge permutations
- Omitting padding and sponge management (absorption and squeezing), which can be efficiently handled in software.

This simplifies the hardware design while maintaining flexibility for SHA-3-512. The next Figure *5-11* shows the simulation of the modified Hardware Implementation.

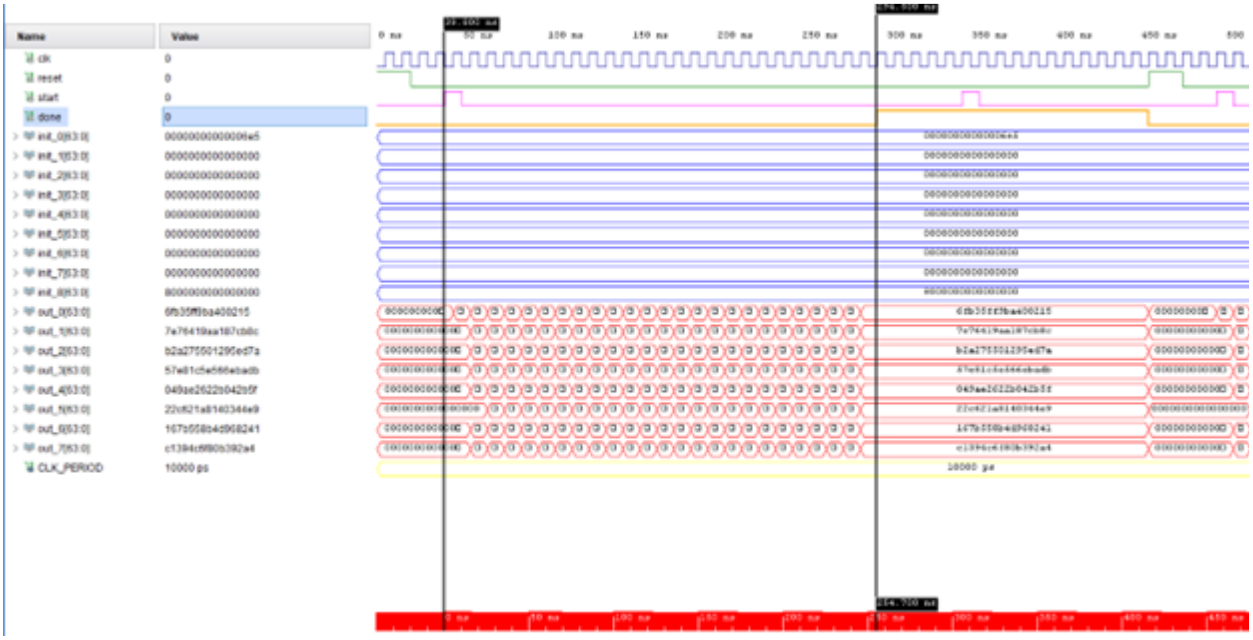

Figure 5-11 – Simulation results of the SHA3-512 Hardware Implementation: the signals in blue are the fixed input, the process starts with the start signal asserting (pink), and the signals in red show the outputs. We can see 24 partial permutation results until the Done signal is asserted (orange).

In the testbench simulation, we first de-assert the reset signal to initialize the sponge. We then populate all nine 32-bit inputs with the pre-padded input vector; after that, we assert the start signal (pink). The model will then perform 24 permutations of the Keccak Round, and some of the partial products can be observed at the output signals (red). Once we complete all permutations, the done signal will assert (orange) to indicate that the calculation is complete. At this point, we can assert the reset signal to initiate a new calculation or populate it with a new input vector and then assert the start signal if we want to calculate using the previous sponge values.

5.2.3 SHA-3 512 Hardware Implementation Block Generation

After validating the functionality of the SHA-3-512 block using a testbench, we integrate it into the original ARM Cortex M3 project by converting it into an AXI-compatible block. We used the process outlined in section 4.6 to add our model to a generic AXI-Lite interface. We designed the AXI interface to handle nine 64-bit inputs as eighteen 32-bit registers and provided sixteen 32-bit

registers for outputs. Additionally, two 32-bit registers are used for the 2-bit input signals Start and Reset, along with the single-bit output done signal, resulting in a total of 62 wasted bits. This inefficiency arises because the AXI block generator uses a minimum width of 32-bit registers. Once the AXI block was constructed, we instantiated it in the original ARM Cortex M3 project and generated the Board Support Package (BSP). The SHA-3 AXI block is memory-mapped, allowing the software to calculate the padding for a new input, pass the eighteen 32-bit input values, check the status of the done signal, and extract the sixteen 32-bit output values. In Figure 55, we can see the minimum system with the new SHA-3 Hardware Implementation SHA-3 AXI block and the new AXI_Timer block. See Figure *5-12*.

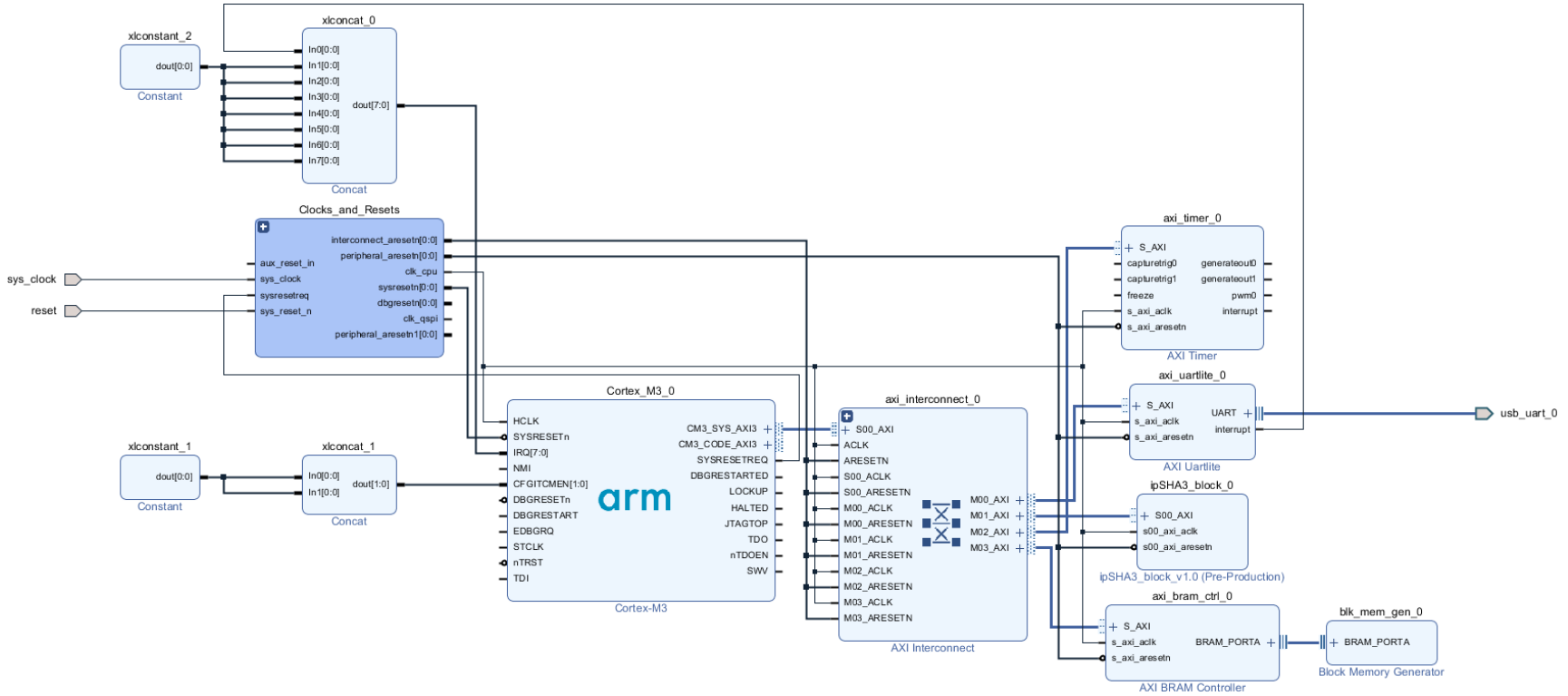

Figure 5-12 - The SHA-3 512 Hardware Implementation block is supplemented with a new SHA-3 AXI block, as well as the AXI Timer block.

After synthesizing and implementing the SHA3 512 Hardware Implementation block, we can observe the SHA-3 AXI block area, which contributes to the overall design (green), compared to the ARM Cortex-M3 (pink). See Figure *5-13*.

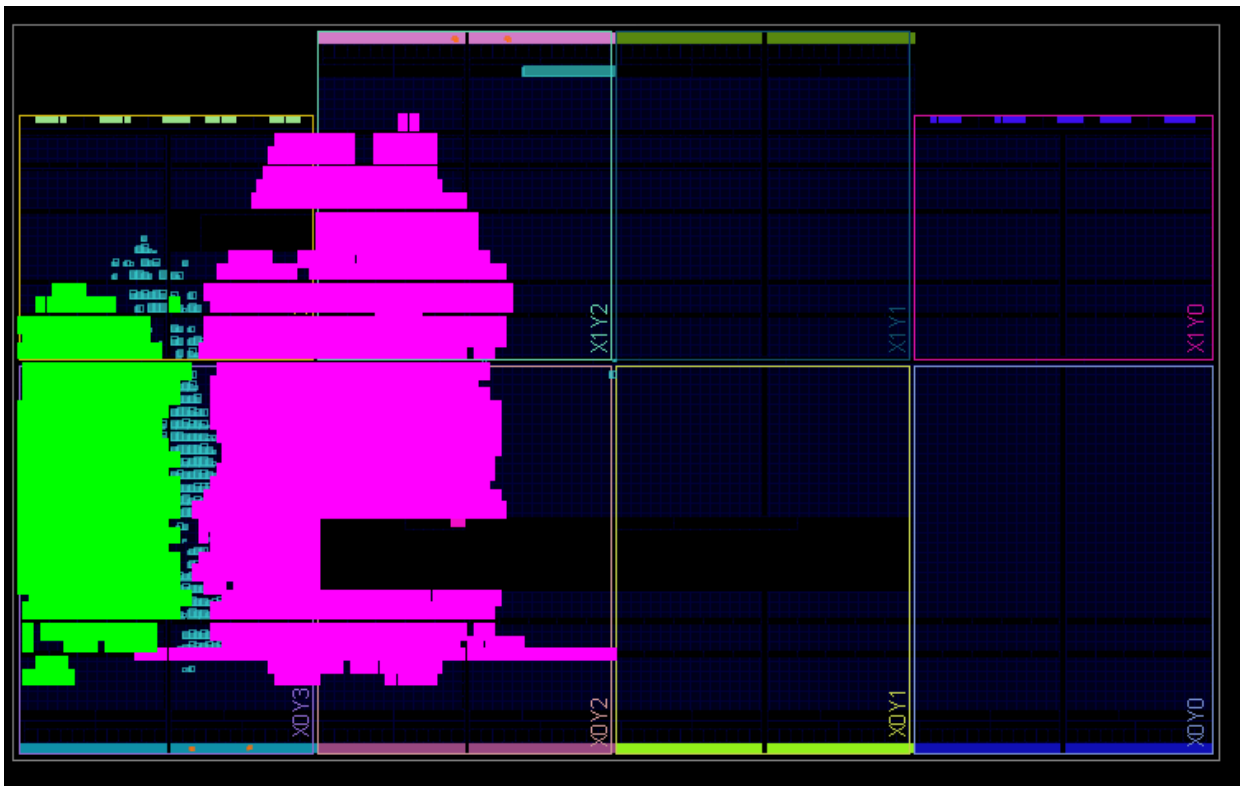

Figure 5-13 – FPGA routing resources of the SHA-3 512 Hardware Implementation on an xc7a100tcsg324-1 FPGA, the ARM Cortex M3 in pink, and the SHA-3 AXI block in green.

The synthesis report for the XC7A100TCSG324-1 FPGA is shown in Figure *5-14*. This indicates a total usage of 15,784 Slice LUTs, representing 24.90% of the total utilization of the xc7a100tcsg324-1 FPGA.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	15784	0	63400	24.90
LUT as Logic	15729	0	63400	24.81
LUT as Memory	55	0	19000	0.29
LUT as Distributed RAM	0	0		
LUT as Shift Register	55	0		
Slice Registers	5822	0	126800	4.59
Register as Flip Flop	5822	0	126800	4.59
Register as Latch	0	0	126800	0.00
F7 Muxes	298	0	31700	0.94
F8 Muxes	104	0	15850	0.66

Figure 5-14 – Utilization report for the SHA-3 512 Hardware Implementation, total utilization of 24.90%

The SHA-3-512 coprocessor utilizes 4,366 slice LUTs. We also show the power report in Figure *5-15*.

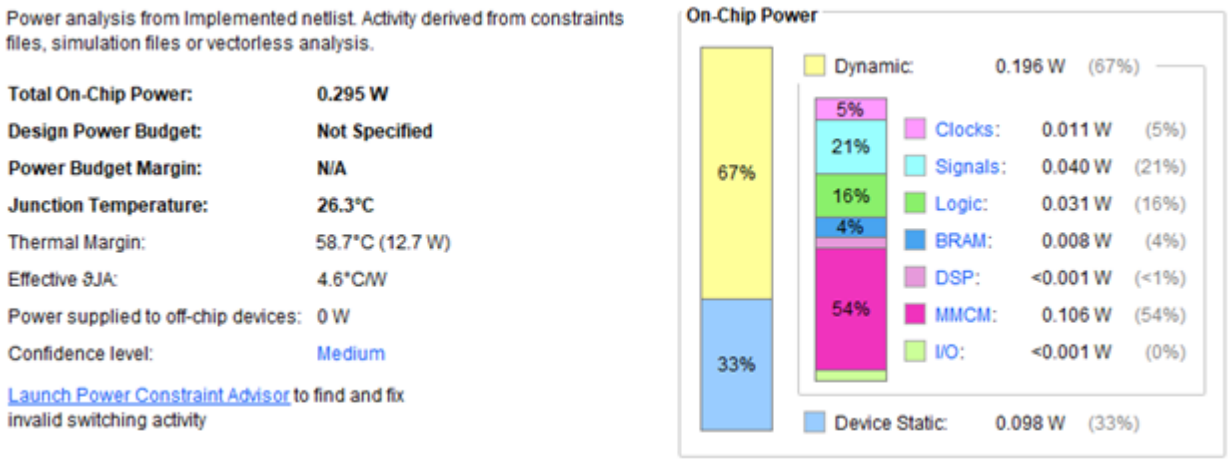

Figure 5-15 - Power Usage estimation of the SHA-3 512 SHA-3 512 Hardware Implementation,

We can also see that the total on-chip power increased to 0.295 Watts.

When we attempted to measure using a USB power meter, it was found that there were no power fluctuations above the normal current and voltage usage of the development board; these results were not included in the analysis.

5.3. The validation test framework is used to validate the Software and Hardware implementations.

To validate both the software and hardware implementations of SHA-3-512, we developed a test framework that performs the following tasks: Using standard test vectors [NIST 15B], the client, running on a computer, communicates with the FPGA via a serial UART interface. The ARM Cortex-M3 softcore executes a server that listens for specific commands. The client needs to read a file with the test vector and then transmit the size and value of each input vector. The client then needs to send the command for either the SHA-3 Software Implementation or the SHA-3 Hardware Implementation. It then needs to send a command to read back the result and another command to read the latency of executing the test. The framework is then compared against the expected value. See *Figure 5-16*.

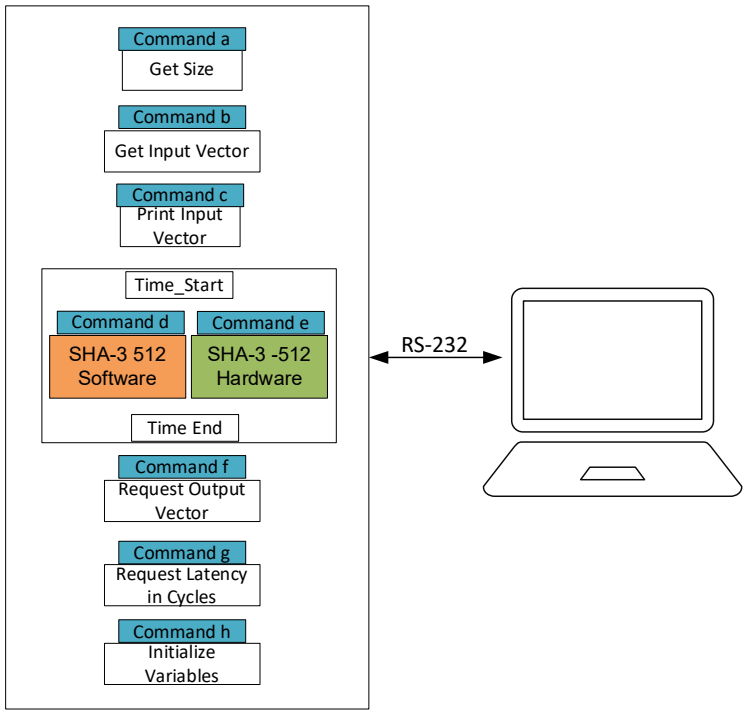

Figure 5-16 – Client-Server Test Framework Architecture: The server runs on the FPGA's ARM Cortex-M3 Softcore, and it supports eight commands. The computer client first needs to send the size, test vector, and either the Software or Hardware implementation. It then reads the Hash result, reads the timer value, or initializes the variables.

For functional validation, we instantiated the hardware shown in Figure *5-12*. We then included both the software and hardware implementations of the software stack. This allows us to call either one without re-programming the FPGA. The functional validation enables us to validate that both

the hardware and software implementations are calculating the expected Hash correctly. It also permits us to gather the timing values to calculate the latency and throughput. In Figure *5-17*, we can see an example run.

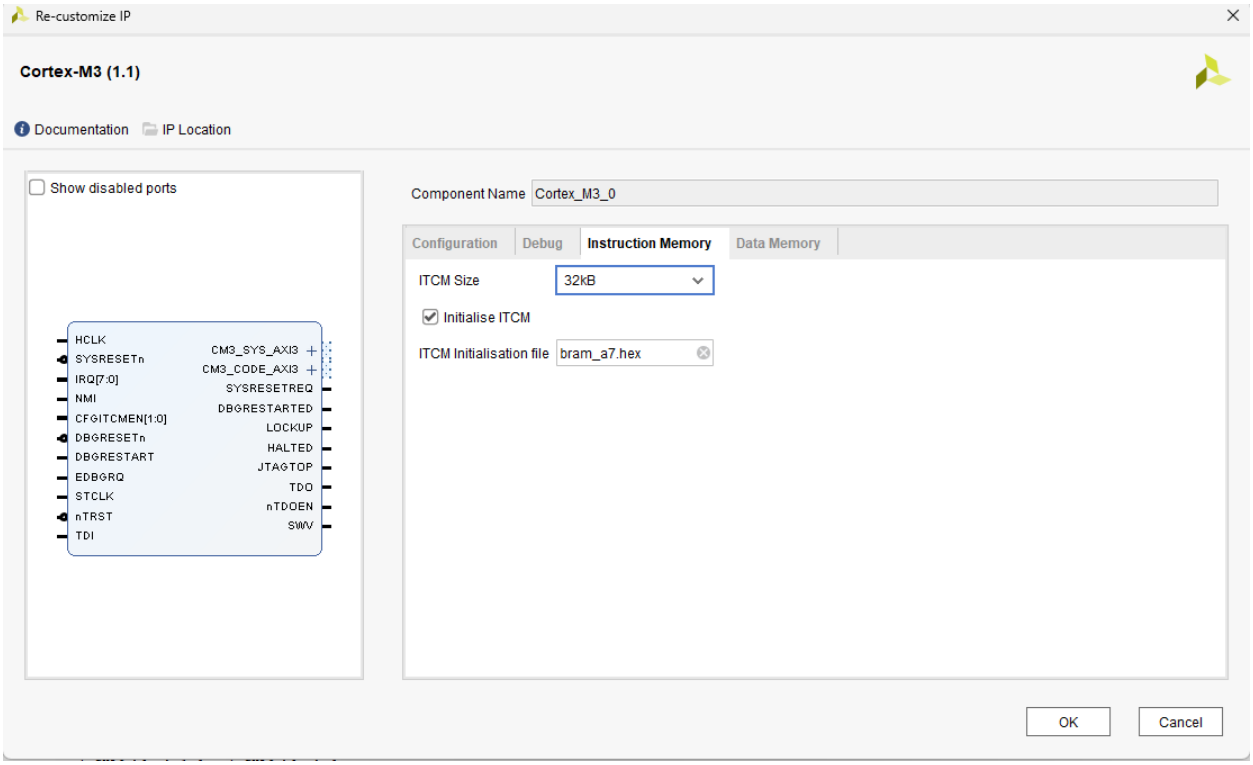

Figure 5-17 – Functional validation example showing two test vectors, each with a precise size of 360 bits. The output first runs the test vector in the hardware implementation, sends the size and vector (input), and then prints the output from the FPGA, followed by the expected output from the file. The same process is shown for the software implementation.

Once we finish the functional validation, we proceed to measure the amount of RAM and ROM required. We begin with the complete software stack, which encompasses both hardware and software implementations. To calculate the RAM and ROM usage of the software implementation, we remove the hardware implementation, resulting in the following usage:

Program Size: Code=17540 RO-data=1320 RW-data=208 ZI-data=5728

Thus, the software implementation has a RAM requirement of RW-data + ZI-data = 5,936 Kbytes and a ROM requirement of Code + RO-data = 18,860 Kbytes.

Next, we measure the RAM and ROM utilized by the hardware implementation. We return to our complete software stack and remove the software implementation code, yielding the following usage:

Program Size: Code=14904 RO-data=1320 RW-data=208 ZI-data=5432

Thus, the hardware variant has a RAM requirement of RW-data + ZI-data = 5,640 Kbytes and a ROM requirement of Code + RO-data = 16,224 Kbytes. See TABLE For RAM/ROM usage.

5.4. Results for both implementations of the SHA3-512 algorithm

After testing the implementations using standard test vectors from NIST FIPS PUB 202 [NIST 15B] and comparing the results against standard generated outputs, we evaluated the performance of the software and hardware SHA-3-512 implementations. Figure *5-18* Displays the latency measurements for different input vector sizes. The left vertical axis represents hardware latency, measured in microseconds, and the right vertical axis represents software latency. The horizontal axis represents different input vector sizes.

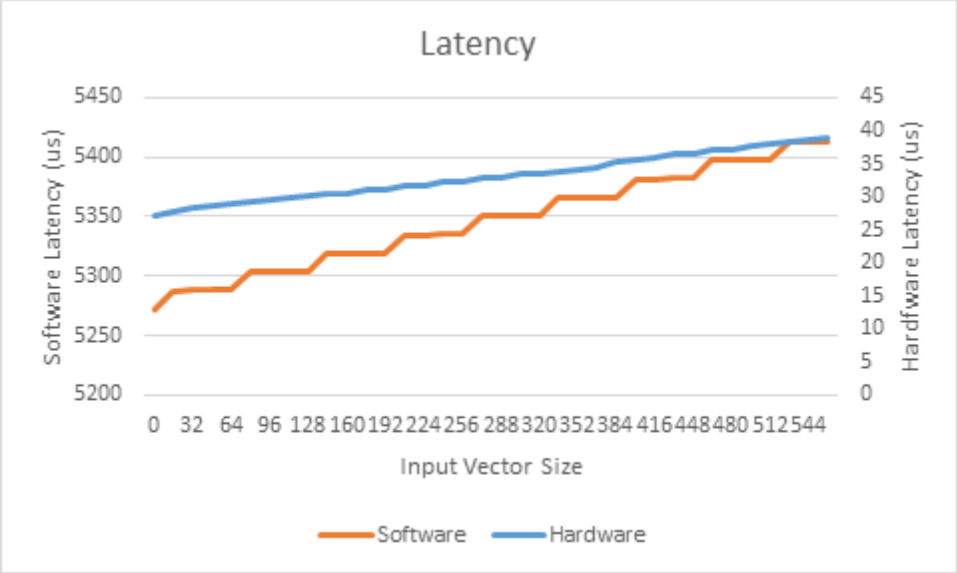

Figure 5-18 – Latency Results using input vectors from 0 to 560 bits. The Left y-axis represents the Software implementation Latency, and the Right y-axis shows the Hardware Implementation Latency.

The software implementation exhibits latencies ranging from 5,271 µs to 5,413 µs, while the hardware implementation is approximately 163 times faster, with latencies between 27.08 µs and 38.93 µs. We see a stepwise behavior in the software implementation due to processing overhead for every 8 bytes of input. This occurs because the software absorbs input into 64-bit (8-byte)

lanes, requiring the bit interleaving function to map bytes to lanes. For example, an 8-byte input fits into one lane, but a 9-byte input spans two lanes, resulting in additional overhead before permutation. In contrast, the hardware implementation processes a fixed 576-bit rate per permutation, with latency variability attributed to external padding operations. Despite this variability, the hardware consistently operates on a fixed block size, contributing to its significant performance advantage.

The SHA-3-512 algorithm generates a constant 512-bit output for both implementations. Using the latency measurements, we can calculate the throughput in two ways:

a. Throughput as operations per second. - The hardware implementation achieves a mean throughput of 30,219 operations per second, while the software implementation achieves a mean throughput of 187 operations per second. See Appendix 1 for more information.

b. Throughput as fixed data divided by latency - The hardware implementation achieves a maximum throughput of 18.9 Mbps at 100 MHz; however, at a maximum input size, it downgrades to 13.15 Mbps. In contrast, the software implementation has a maximum throughput of 97 Kbps, which decreases to 94 Kbps at 100 MHz.

In TABLE VII we present the complete results for each implementation.

TABLE VII – COMPARISON BETWEEN THE SHA-3 SOFTWARE IMPLEMENTATION VERSUS THE SHA-3 HARDWARE IMPLEMENTATION. THE HARDWARE IMPLEMENTATION IS 163 TIMES FASTER, BUT IT USES 17% MORE POWER AND 38% MORE RESOURCES; WE PRESENT AVERAGE VALUES CALCULATED FROM Appendix A – Results

	Hardware Implementation	**Software Implementation**
RAM	5.6 Kbytes	5.9 Kbytes
ROM	16.2 Kbytes	18.8 Kbytes
Slice LUTs	15,784	11,418
Total On-Chip Power	295 mW	252 mW
Latency (average)	33.09083 µs	5.34 ms
Throughput per second	30,219 operations/sec	187 operations/sec

The Hardware implementation is 163 times faster than the software implementation, as expected, mainly due to the parallel execution of all state mapping operations. There is also no need to use

interleaving to preprocess the data that is sent to and from the state array. We can also see that the Hardware implementation uses 17% more power and occupies 38% more area.

5.5. Conclusions and Future Work

In this work, we utilized an ARM Cortex-M3 IP softcore to implement the SHA3-512 hash algorithm using two approaches that we name:

1. Software implementation
2. Hardware implementation

Although both solutions have hardware and software elements, this distinction is merely a means to differentiate them. Both solutions comply with the SHA3-512 parameters and, as such, represent the most robust hash implementation.

We found that the software implementation is more straightforward, but on a 32-bit architecture, it was considerably slower than the hardware implementation. The hardware implementation was more complex because the default implementation was designed as a SHA3-256. Thus, we were required to make modifications.

For this project, we utilized an ARM Cortex-M3 IP softcore and found that most resources and tools have not been updated. We had to use older versions of Vivado and Keil software to avoid errors and warnings. As a result, there was a significant learning curve in adapting the original ARM Cortex-M3 project. After conducting multiple experiments, we developed a general methodology to implement changes to the original project, which we presented in Chapter 4.

In a production environment with limited-resource devices such as the ARM Cortex-M3, a less secure SHA3 implementation might be necessary to achieve higher speeds with lower resource usage. Nevertheless, this work validated that the most secure algorithm in NIST's arsenal, SHA3-512, can be implemented on a small softcore, demonstrating that less complex algorithms should also be feasible. Each implementation resulted in different hardware consumption and performance characteristics.

The Hardware implementation uses 5% less RAM and 16% less ROM, as expected, since it executes the state mapping permutations in hardware. Thus, less memory is in use. However, this is offset by the increased power usage, which is 17% higher, and the number of slices, which increases by 38%. Overall, the performance is 163 times better in the hardware implementation.

Finally, there is still work to be done. The following lists future enhancements that could be implemented:

1. For the software implementation on a 32-bit architecture, implementing a SHA3-512 solution that assumes 64-bit lanes poses significant challenges. We tested and validated the interleaving technique; however, we could also have split each 64-bit lane into its low- and high-32-bit parts. The state mappings (θ, ρ, π, χ, ι) do not inherently suffer in terms of correctness, as all operations can be adapted to work with 32-bit parts. The primary challenge lies in implementing rotations in the Rho state mapping to correctly handle bit crossings between the low and high parts of the state. Testing this enhancement could potentially eliminate the step behavior observed in the latency results (see Figure *5-18* For the software implementation).

2. The hardware implementation was tested only for input sizes up to 72 bytes, complying with NIST's small vector testing. To support larger input sizes, the hardware implementation would not require significant changes; however, the software control module would need to XOR the output of the first partial results with the incoming partial data and initiate another cycle in the hardware implementation.

3. Our hardware implementation, described as fully unrolled, processes the state mappings directly in a sequential manner. The Keccak Team proposes a mid-range core that uses folding, where "one of the known techniques for trading silicon area at the cost of performance is the so-called folding." [Bertoni et al, - 12]. Future work could implement this variant and evaluate its performance and resource usage.

6. Appendix A – Results

TABLE VIII – PERFORMANCE MEASUREMENT TABLE, SHOWING THE SPECIFIC LATENCY AND THROUGHPUT FOR A GIVEN INPUT SIZE.

bits	Hardware	Software	Hardware	Software	Hardware	Software			Hardware	Software	Hardware	Software	Hardware	Software	Hardware	Software
	Latency In Cycles		Latency (s)		Latency (us)			Output Data	Throughput (bps)		Throughput (Kbps)		Throughput (Mbps)		Operations per sec	
0	2708	527154	2.708E-05	0.00527154	27.08	5271.54	194.66544	512	18906942	97125.3	18906.942	97.1253	18.9069	0.09713	36927.6	189.698
16	2787	528770	2.787E-05	0.0052877	27.87	5287.7	189.72731	512	18371008	96828.5	18371.008	96.8285	18.371	0.09683	35880.9	189.118
32	2829	528798	2.829E-05	0.00528798	28.29	5287.98	186.92047	512	18098268	96823.4	18098.268	96.8234	18.0983	0.09682	35348.2	189.108
48	2868	528820	2.868E-05	0.0052882	28.68	5288.2	184.38633	512	17852162	96819.3	17852.162	96.8193	17.8522	0.09682	34867.5	189.1
64	2888	528804	2.888E-05	0.00528804	28.88	5288.04	183.10388	512	17728532	96822.3	17728.532	96.8223	17.7285	0.09682	34626	189.106
80	2926	530355	2.926E-05	0.00530355	29.26	5303.55	181.25598	512	17498291	96539.1	17498.291	96.5391	17.4983	0.09654	34176.3	188.553
96	2945	530360	2.945E-05	0.0053036	29.45	5303.6	180.08829	512	17385399	96538.2	17385.399	96.5382	17.3854	0.09654	33955.9	188.551
112	2984	530382	2.984E-05	0.00530382	29.84	5303.82	177.74196	512	17158177	96534.2	17158.177	96.5342	17.1582	0.09653	33512.1	188.543
128	3004	530366	3.004E-05	0.00530366	30.04	5303.66	176.55326	512	17043941	96537.1	17043.941	96.5371	17.0439	0.09654	33288.9	188.549
144	3042	531917	3.042E-05	0.00531917	30.42	5319.17	174.85766	512	16831032	96255.6	16831.032	96.2556	16.831	0.09626	32873.1	187.999
160	3061	531922	3.061E-05	0.00531922	30.61	5319.22	173.77393	512	16726560	96254.7	16726.56	96.2547	16.7266	0.09625	32669.1	187.997
176	3100	531944	0.000031	0.00531944	31	5319.44	171.59484	512	16516129	96250.7	16516.129	96.2507	16.5161	0.09625	32258.1	187.99
192	3120	531928	0.0000312	0.00531928	31.2	5319.28	170.48974	512	16410256	96253.6	16410.256	96.2536	16.4103	0.09625	32051.3	187.995
208	3158	533479	3.158E-05	0.00533479	31.58	5334.79	168.92939	512	16212793	95973.8	16212.793	95.9738	16.2128	0.09597	31665.6	187.449
224	3177	533484	3.177E-05	0.00533484	31.77	5334.84	167.92068	512	16115833	95972.9	16115.833	95.9729	16.1158	0.09597	31476.2	187.447
240	3216	533506	3.216E-05	0.00533506	32.16	5335.06	165.89117	512	15920398	95968.9	15920.398	95.9689	15.9204	0.09597	31094.5	187.439
256	3241	533495	3.241E-05	0.00533495	32.41	5334.95	164.60815	512	15797593	95970.9	15797.593	95.9709	15.7976	0.09597	30854.7	187.443
272	3279	535046	3.279E-05	0.00535046	32.79	5350.46	163.17353	512	15614517	95692.7	15614.517	95.6927	15.6145	0.09569	30497.1	186.9
288	3298	535051	3.298E-05	0.00535051	32.98	5350.51	162.23499	512	15524560	95691.8	15524.56	95.6918	15.5246	0.09569	30321.4	186.898
304	3337	535073	3.337E-05	0.00535073	33.37	5350.73	160.34552	512	15343123	95687.9	15343.123	95.6879	15.3431	0.09569	29967	186.89
320	3357	535057	3.357E-05	0.00535057	33.57	5350.57	159.38546	512	15251713	95690.7	15251.713	95.6907	15.2517	0.09569	29788.5	186.896
336	3395	536608	3.395E-05	0.00536608	33.95	5366.08	158.05832	512	15081001	95414.2	15081.001	95.4142	15.081	0.09541	29455.1	186.356
352	3414	536613	3.414E-05	0.00536613	34.14	5366.13	157.18014	512	14997071	95413.3	14997.071	95.4133	14.9971	0.09541	29291.2	186.354
368	3453	536635	3.453E-05	0.00536635	34.53	5366.35	155.41124	512	14827686	95409.4	14827.686	95.4094	14.8277	0.09541	28960.3	186.346
384	3534	536637	3.534E-05	0.00536637	35.34	5366.37	151.84975	512	14487832	95409	14487.832	95.409	14.4878	0.09541	28296.5	186.346
400	3574	538183	3.574E-05	0.00538183	35.74	5381.83	150.58282	512	14325686	95134.9	14325.686	95.1349	14.3257	0.09513	27979.9	185.81
416	3595	538188	3.595E-05	0.00538188	35.95	5381.88	149.70459	512	14242003	95134	14242.003	95.134	14.242	0.09513	27816.4	185.809
432	3637	538211	3.637E-05	0.00538211	36.37	5382.11	147.98213	512	14077536	95130	14077.536	95.13	14.0775	0.09513	27495.2	185.801
448	3658	538199	3.658E-05	0.00538199	36.58	5381.99	147.12931	512	13996720	95132.1	13996.72	95.1321	13.9967	0.09513	27337.3	185.805
464	3698	539745	3.698E-05	0.00539745	36.98	5397.45	145.95592	512	13845322	94859.6	13845.322	94.8596	13.8453	0.09486	27041.6	185.273
480	3719	539755	3.719E-05	0.00539755	37.19	5397.55	145.13444	512	13767142	94857.9	13767.142	94.8579	13.7671	0.09486	26888.9	185.269
496	3761	539773	3.761E-05	0.00539773	37.61	5397.73	143.51848	512	13613401	94854.7	13613.401	94.8547	13.6134	0.09485	26588.7	185.263
512	3790	539769	0.0000379	0.00539769	37.9	5397.69	142.41926	512	13509235	94855.4	13509.235	94.8554	13.5092	0.09486	26385.2	185.264
528	3830	541315	0.0000383	0.00541315	38.3	5413.15	141.33551	512	13368146	94584.5	13368.146	94.5845	13.3681	0.09458	26109.7	184.735
544	3851	541320	3.851E-05	0.0054132	38.51	5413.2	140.56609	512	13295248	94583.6	13295.248	94.5836	13.2952	0.09458	25967.3	184.734
560	3893	541343	3.893E-05	0.00541343	38.93	5413.43	139.05548	512	13151811	94579.6	13151.811	94.5796	13.1518	0.09458	25687.1	184.726
	3309.0833	**534666.81**	**3.309E-05**	**0.005346668**	**33.0908**	**5346.67**	**163.15365**	**512**	**15635919**	**95766.2**	**15635.919**	**95.7662**	**15.6359**	**0.09577**	**30219.8**	**187.032**

7. Appendix B – Source Code

All hardware Verilog models, as well as the software used in this work, are available in the following GitHub project: https://github.com/omarecg77/ItesoARMCortex.

TABLE IX – ORGANIZATION OF THE VERILOG AND C CODE USED TO TEST AND VERIFY THE SHA3-512 IMPLEMENTATIONS.

Path	Description
ItesoARMCortex/Hardware/AXI_model/	AXI Lite Model with the Keccak SHA3-512 Hardware Block.
ItesoARMCortex/Hardware/Keccak_wrapper/	SHA3-512 Hardware Block for simulation
ItesoARMCortex/Software/ArmCortex/	The server software running in the softcore includes code for the SHA3-512 Software and Hardware implementation.
ItesoARMCortex/Software/TestBench/	Client Software running on a PC includes the Test Vectors used to validate the implementations.

8. Bibliography

[Aguilera-Galicia-19] Aguilera-Galicia, C. R. (2019). Design and Implementation of Reciprocal Square Root Units on Digital ASIC Technology For Low Power Embedded Applications. Tesis de doctorado, Doctorado en Ciencias de la Ingeniería. Tlaquepaque, Jalisco: ITESO.

[Tschofenig et al. - 15] Tschofenig, H., et al. (2015) Architectural Considerations in Smart Object Networking. Internet Architecture Board. https://www.rfc-editor.org/rfc/rfc7452.txt

[Fagan et al. – 20] Fagan, M., Megas, K. N., Scarfone, K., & Smith, M. (2020). IoT device cybersecurity capability core baseline (NISTIR 8259A). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.8259A

[Fagan et al. -21] Fagan, M., Megas, K. N., Scarfone, K., & Smith, M. (2021). IoT device cybersecurity guidance for the federal government: Establishing IoT device cybersecurity requirements (NIST Special Publication 800-213). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-213

[Aumasson -18] Aumasson, J.-P. (2018). Serious cryptography. No Starch Press.

[Windarta et al, - 22] Windarta, S., Suryadi, S., Ramli, K., Pranggono, B., & Gunawan, T. S. (2022). Lightweight cryptographic hash functions: Design trends, comparative study, and future directions. IEEE Access, 10, 82272–82294. https://doi.org/10.1109/ACCESS.2022.3195471

[Rodriguez et al, - 06] Rodríguez-Henríquez, F., Saqib, N. A., Díaz-Pérez, A., & Koç, Ç. K. (2006). Cryptographic algorithms on reconfigurable hardware. Springer. https://doi.org/10.1007/978-0-387-36682-1

[Rivest – 92] Rivest, R. (1992). The MD5 message-digest algorithm (RFC 1321). Internet Engineering Task Force. https://doi.org/10.17487/RFC1321

[Dang – 12] Dang, Q. (2012). Recommendation for applications using approved hash algorithms (NIST Special Publication 800-107, Revision 1). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-107r1

[NIST – 15A] National Institute of Standards and Technology. (2015). SHA-3 standard: Permutation-based hash and extendable-output functions (FIPS Publication 202). U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.202

[NIST -15B] National Institute of Standards and Technology. (n.d.). Cryptographic Algorithm Validation Program: Secure hashing - SHA-3 vs SHA-3VS. U.S. Department of Commerce. Retrieved June 15, 2025, from https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/Secure-Hashing#sha3vsha3vss

[Bertoni et al, – 11] Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G. (2011). Cryptographic sponge functions. https://keccak.team/files/CSF-0.1.pdf

[Yiu – 19] Yiu, J. (2019). System-on-chip design with Arm® Cortex®-M processors: Reference book. Arm Education Media.

[ARM – 10A] Arm. (n.d.). DesignStart: Fast, low-risk access to Arm IP. Arm. https://www.arm.com/resources/designstart. Retrieve June 2025.

[ARM - 10B] Arm. (n.d.). Cortex-M3: High-performance, energy-efficient 32-bit processor. Arm. https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m3

[ARM – 10C] Arm. (n.d.). Arm Flexible Access. Arm https://www.arm.com/products/flexible-access

[ARM – 10D] Arm. (n.d.). Cortex-M3 DesignStart™ Eval RTL and FPGA quick start guide. Arm. https://www.arm.com/resources/designstart

[ARM – 10E] Arm (n.d.). Cortex-M3 DesignStart FPGA Xilinx Edition (AT426). Arm. https://developer.arm.com/downloads/view/AT426

[ARM – 20] Arm. (n.d.). End-User License Agreement for Arm DesignStart Evaluation Arm. Included in the ARM Cortex-M3 package.

[DIGILENT – 23] Digilent. (n.d.). Vivado-boards [Source code]. GitHub. https://github.com/Digilent/vivado-boards/archive/master.zip

[XKCP – 23] Keccak Team. (2023). XKCP: eXtendable Keccak Code Package Computer software. GitHub. https://github.com/XKCP/XKCP

[Keccak – 12] Keccak Team. (n.d.). Hardware implementations. Retrieved June 15, 2025, from https://keccak.team/hardware.html

[Bertoni et al, - 12] Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G. Keer (2012). Keccak Implementation Overview. https://keccak.team/files/Keccak-implementation-3.2.pdf