SHA3-FPGA: Secure Hash Algorithm 3 Implementation on an FPGA with MicroBlaze Microprocessor

***A*bstract — With the increase in a global presence in software and hardware devices, the need for security has as well. Cryptographic solutions aim to provide an efficient and secure way of protecting and securing devices in an ever-growing field. One important cryptographical tool is hashing. However, hashing algorithms, which perform the hashing, are often outdated, and become vulnerable to cyber-attacks over time. When this happens, a new hashing algorithm standard needs to replace the old one for better security. In the last decade, the world transitioned from using Secure Hash Algorithm 1 (SHA-1) to Secure Hash Algorithm 2 (SHA-2) due to the flaws in SHA-1. However, its replacement, SHA-2, still uses a similar internal structure which has the same mathematical flaws. The reason why SHA-2 is safer is because of the length of its output compared to SHA-1. Since then, a new hash algorithm, SHA-3 has been introduced which does not share the same flaws as its other family members. However, the transition to SHA-3 has not been completely due to the lack of support for SHA-3 in software and hardware as well as issues with performance. This performance limitation, however, is only present in software implementations. This project aims to measure the performance metrics of an FPGA implementation of SHA-3 to introduce a realistic, hardware method of implementing SHA-3 to contribute to a potential global implementation to a more secure hashing algorithm standard.**

Yuta Akiya, Kyle Thomas Le, Megan Luong, **Dr. Mohamed El-Hadedy**

*Electrical and Computer Engineering Department at California State Polytechnic University, Pomona*

Contact: yakiya@cpp.edu, kylethomasle@cpp.edu, mmluong@cpp.edu, **mealy@cpp.edu**

*Index Terms*— Secure Hash Algorithm 3 (SHA-3), Field-Programmable Gate Array (FPGA), VHDL, Advanced eXtensible interface (AXI), MicroBlaze, VGA, Xilinx Vivado, Xilinx SDK

# INTRODUCTION

S

ecurity is one of the most fundamental and important part of any system. In 2020 alone, there were 3,950 data breaches confirmed [6], resulting in possibly millions of accounts and dollars lost. This is where the importance of hashing algorithms such as the Secure Hash Algorithm 3 (SHA-3) come in. Hashing algorithms provide digital signatures for applications to ensure they are safe. If any part of the application or file is altered, the hashing algorithm will produce a different value [5]. Thus, it can provide a strong sense of security if signatures of proven safe software are kept and used to compare against any possible malicious files.

SHA-3 is the latest encryption standard in the hashing family, aiming to provide a faster and more robust implementation in hardware. This performance is important for a hashing algorithm since a faster hash allows for more time for other operations. In a world where billions of devices are connected, and even more applications and files are being transferred, it is important for a hashing algorithm to be efficient and lightweight.

In addition, the other hash algorithms in the SHA family, SHA-1 and SHA-2, have been exposed to have cryptographic flaws [5]. Despite this, the world has not transitioned to using SHA-3. This is because most software and hardware do not support it [5]. Being able to implement SHA-3 on FPGA can be important in this transition because it adds more options for

Field-programmable gate arrays (FPGAs) are devices providing re-programmable hardware that are extensively efficient in processing time and precising timing [10]. The aim of this report is to design and implement the SHA-3 algorithm onto an FPGA system and analyze its performance metrics. This can then be compared to traditional and other related systems to see if the performance benefit of an FPGA is able to provide a boost to the SHA-3 algorithm.

In this report, section I covers the introduction as well as background information about SHA-3 and the MicroBlaze processor, section II covers related works, section III covers the SHA-3 algorithm in detail, section IV covers the design of the project as well as individual modules in the design, section V covers the implementation of the MicroBlaze processor, section VI covers the project results and data analysis, and section VII covers the project conclusion.

## Secure Hash Algorithm 3 Description

Secure Hash Algorithm 3 (SHA-3) was released by the National Institute of Standards and Technology (NIST) in August of 2015 as a new member of the secure hash algorithm (SHA) family [9]. SHA-3 is a hash algorithm that, for any given input of variable length, generates a unique and irreversible hash using a sponge construction approach [5]. Although SHA-3 is part of the same family as SHA-1 and SHA-2, they do not have much in common in terms of internal structure. SHA-3 is a subset of Keccak and uses sponge construction while both SHA-1 and SHA-2 use Merkle–Damgård construction.

Because SHA-1 and SHA-2 share the same base construction, they have the same flaws. Both algorithms have been exposed by public attacks and are vulnerable to certain types of cyber-attacks. For this reason, SHA-3 uses a different internal structure: Keccak.

Keccak is based on a sponge construction that permutates input data based on functions and permutations. The sponge construction can be broken down into two phases: the absorb phase and squeeze phase. Input data is taken in during the absorb phase and the permutated data is outputted during the squeeze phase. Keccak can take in and output data of any size. However, for SHA-3, the input size is a fixed 1600 bits, and the output size is either 226, 256, 384, or 512 bits depending on which version of SHA-3 is being used.

The permutation algorithm written in pseudo code from the Keccak team [3] is referenced and implemented in VHDL. The design and implementation of the algorithm is detailed in Section III.

## MicroBlaze Description

The implementation of SHA-3 is employed onto the Artix-7 based Nexys A7 FPGA and its MicroBlaze microprocessor. The MicroBlaze microprocessor is a soft microprocessor core, being fully implemented via general memory and logic in the FPGA [4]. This allows for the optional ability of embedded development without the need for dedicated hardware, saving resources and costs on the FPGA. This embedded development capability allows for C/C++ overhead while being closely connected to the FPGA fabric logic where the SHA-3 logic is held.

MicroBlaze is able to connect to the FPGA fabric logic through the implementation of memory-mapped addressing in a system called the Advanced eXtensible Interface (AXI). AXI provides for specific memory-mapped input/output designations for register control of specific operations, control signals, and input/output flow logic. This register logic allows the C/C++ embedded design of the microprocessor to connect to the FPGA fabric logic [11]. Memory and calculations can be made in the MicroBlaze microprocessor unit, then sent to the FPGA fabric logic via registers. Then, that data can be manipulated by other logic and/or sent back to the microprocessor.

# Related Works

SHA-3 implementation on an FPGA has been done in other works before.

For Xilinx, a SHA-3 512 intellectual property (IP) core is designed for commercial usage [8]. The IP core features configurable padding options of Keccak or FIPS202. In this project, however, the padding option is fixed to Keccak padding.

In addition, for Ultrascale and Altera FPGAs, the company Xiphera created a SHA3-256 IP core [12]. It is important to note that this project only supports one output size of 256 bits. It does not support any other size such as 224, 2384, and 512. In our project, however, the output size is configurable.

These implementations are important as they provide a reference to compare this system to and provide a baseline of SHA-3 on FPGAs.

# SHA-3 Algorithm

The algorithm supports many different output digests. This project aims to support some of the most common outputs, SHA3-224, SHA3-256, SHA3-384, and SHA3-512.

## Message Preparation

When a message is inputted into the SHA-3 algorithm, it does not run the Keccak permutations on it directly. SHA-3 requires that the input to the Keccak permutations is 1600 bits. For this reason, a message preparation stage is required. Since the input message can be of any length, the extent of the padding performed on the input must be based on its length. As seen in Table 1, the padded input of SHA-3 has two sections: the rate and the capacity, which have sizes that depend on the output size of the algorithm. The resulting 1600-bit message is also called the state.

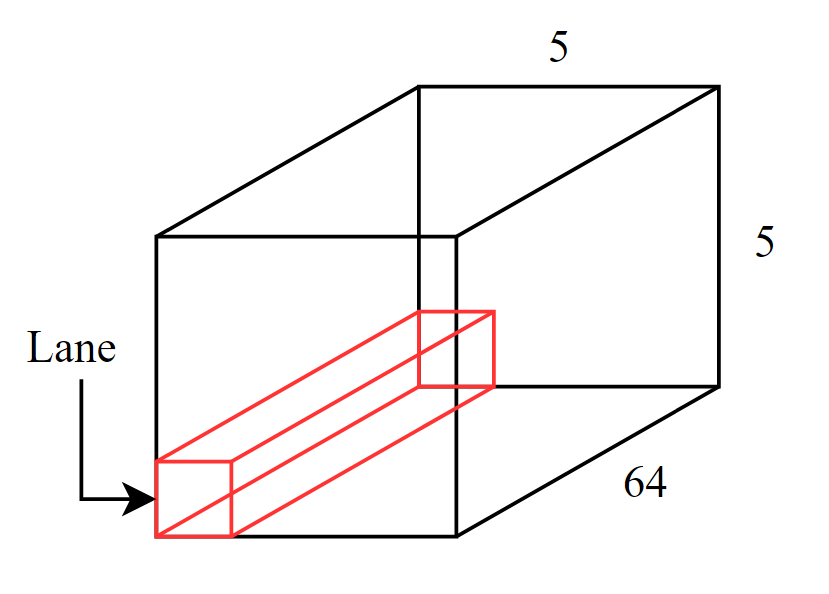
|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Output Size** | **Rate** | **Capacity** |
| SHA3-224 | 224 | 1152 | 448 |
| SHA3-256 | 256 | 1088 | 512 |
| SHA3-384 | 384 | 832 | 768 |
| SHA3-512 | 512 | 576 | 1024 |

*Table 1 – The rate and capacity of each version of SHA3.*

The message is padded so that the size is equal to the rate. Then, the capacity is filled with 0’s and a 0x80. The capacity is never affected by the input message and is used to hide the internal state from attackers. After the last bit of the input message, a 0x06 value is included.

It is important to note that if the input size is larger than the rate, the message preparation results in garbage because the input is truncated at the size of the rate.

The resulting state is now ready to be used as in input to the Keccak permutation stages. The input state can be visualized using a 3-D 5-bit by 5-bit by 64-bit box as seen in Figure 1.

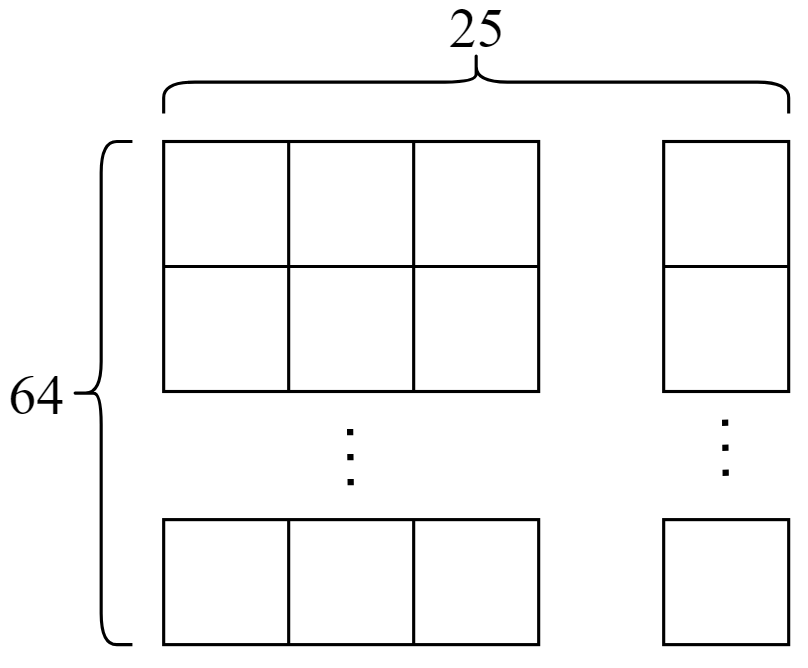


*Fig. 1 – 3-D “box” representation of 1600 bit input.*

The term for a single isle of 64 bits within the state is a lane. This is important for understanding how the permutation algorithms work.

## Keccak Permutations

### Array Setup: The Keccak permutations take in a 1600-bit input and pass it through 5 permutation stages for 24 rounds. For SHA-3, the padded input string acts as the 1600-bit input. Before going through the rounds of permutation stages, the 1600 bits must be loaded into a data structure in a specific way. This project uses a 25 by 64 bit 2-D array, MEM, as the data structure to store the input bits into. Each 64-bit vector column represents a lane from the box diagram in Figure 1.



*Fig. 2 – 2-D array representation of a 1600-bit input.*

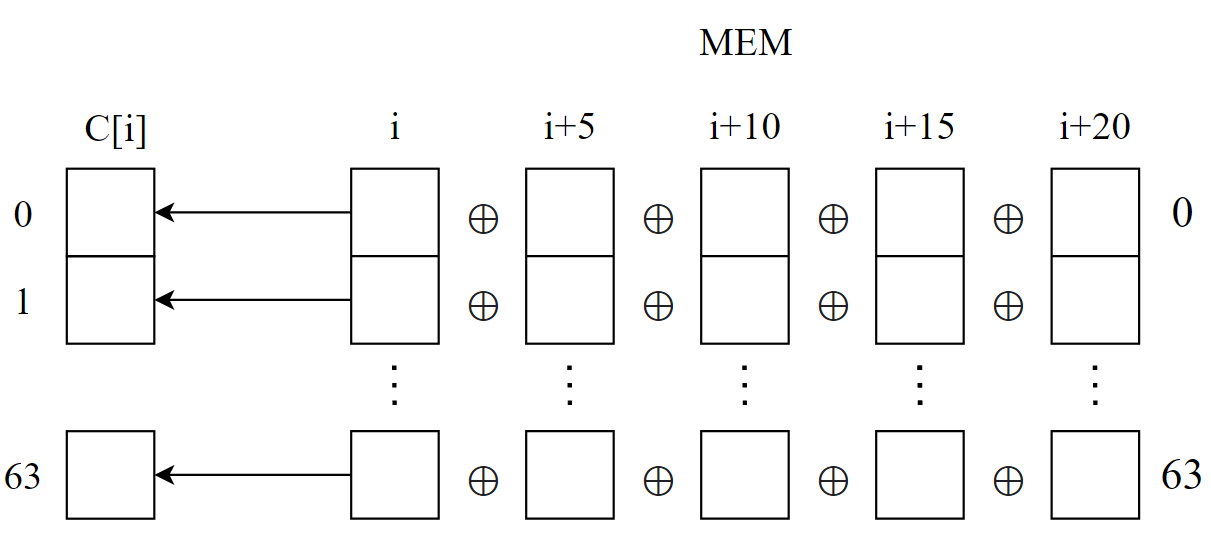
The input data is then loaded into MEM. In each iteration of a double for loop, 64 bits from the input bits are loaded into a lane of MEM.

### Theta: This step requires two temporary row arrays. A row array is essentially 5 elements of the MEM array.

It is referred to as a row because the 25 elements of the MEM array can, as seen in Figure 1, be represented as a 5x5x64 cube. Thinking of the MEM array as a box can make the naming of the row array more intuitive.

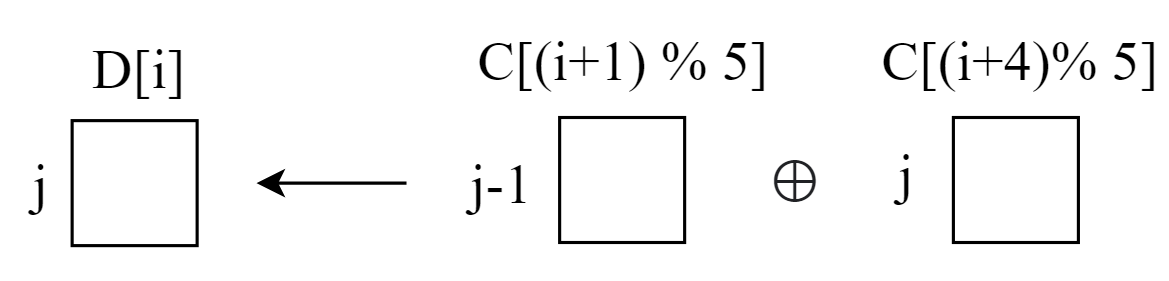
C and D are the two temporary row arrays made for the theta permutation. First, the 5 elements of C are filled using a double for loop where the outer for loop that iterates from 0 to 4 and an inner for loop that iterates through 0 to 63.

For every iteration of the outer loop, a new element of C is loaded with data derived from XORed elements of MEM as seen below.



*Fig. 3 – Demonstration on how array C is loaded in with data, where variable i represents the outer loop variable.*

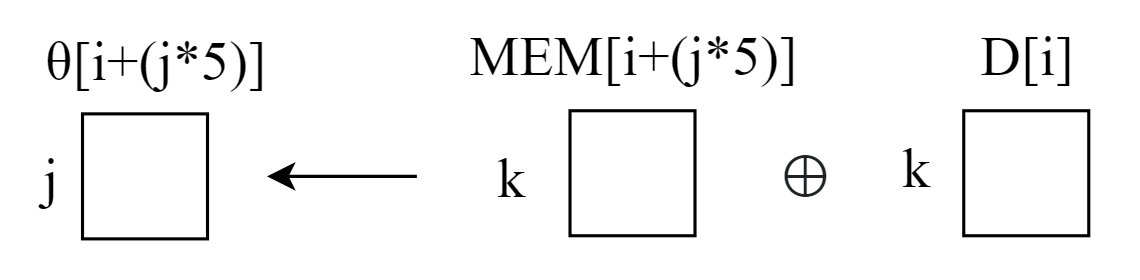
Next, array D is loaded with XORed values from array C in another double for loop setup where the outer loop iterates from 0 to 4 and the inner iterates from 0 to 63. The pattern in which D is loaded with data from C is shown in Figure 4.



*Fig. 4 – Demonstration on how array D is loaded in with data from array C, where i represents the outer loop variable and the j represents the inner loop variable.*

The j-1 seen in Figure 4 is more accurately (j-1) % 64. The modulo is left out for readability. The modulus operators used in addressing both the row and column of C allow the address to loop back to the start when the address goes over or under the range of the array. For example, if i+1 becomes 5, the modulo 5 brings back the value to 0 because 4 is the max row address for C.

The last step in the theta permutation stage is getting the theta output array which is the permutated version of the MEM array. The theta output array is derived from XORing elements from MEM and elements from array D in a triple for loop. The outer and middle loops iterate from 0 to 4 and the inner loop iterates from 0 to 63.



*Fig. 5 – Demonstration on how the theta output array is loaded in with data from MEM and array D where i represents the outer loop variable, j represents the middle loop variable, and k represents the inner loop variable.*

The theta output array, which is the updated MEM array, is now passed onto the next permutation stage, Rho. The output from each permutation is referred to as MEM instead of its actual name for consistency.

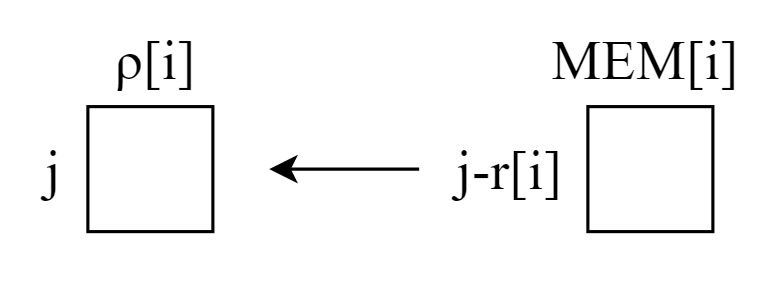
### Rho:

The rho stage consists of a bitwise rotation on the MEM array. The algorithm can be simplified by creating an array of rotation constants seen in Table 2 derived from the original algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 62 | 28 | 27 |
| 36 | 44 | 6 | 55 | 20 |
| 3 | 10 | 43 | 25 | 39 |
| 41 | 45 | 15 | 21 | 8 |
| 18 | 2 | 61 | 56 | 14 |

*Table 2 – Rho rotation constants from Chou GitHub [2]*

Using the rotation constants from Table 2 makes the rho permutation algorithm more readable in code implementation. A new array, the rho output array, is created to hold the output of the rho permutation. To load the rho array, elements from MEM are copied over to in a double for loop with an outer loop iterating from 0 to 24 and an inner loop iterating from 0 to 63 is used as shown in the diagram below.



*Fig. 6 – Demonstration on how the rho array is loaded in with elements from MEM where i is the outer loop variable, j is the inner loop variable, and r is a 25-element long array consisting of the rotation constants shown in Table 2.*

The j-r[i] address in Figure 6 is actually j-r[i] % 64. Again, the modulus operator keeps the address from exceeding the length of the array and is kept out of the diagram for readability. After the rho array is loaded, the rho array is passed onto the pi permutation stage.

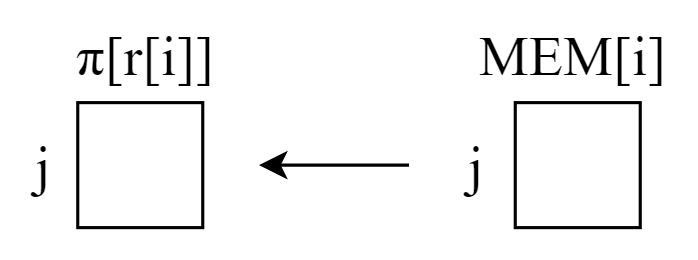
### Pi:

The pi stage, similar to the rho stage, permutates the MEM array using the pi permutation constants. The constants are shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 10 | 20 | 5 | 15 |
| 16 | 1 | 11 | 21 | 6 |
| 7 | 17 | 2 | 12 | 22 |
| 23 | 8 | 18 | 3 | 13 |
| 14 | 24 | 9 | 19 | 4 |

*Table 3 – Pi permutation constants from Keccak Team [3]*

The pi permutation is implemented using two for loops which, like in the rho stage, has an outer loop iterating from 0 to 24 and an inner loop iterating from 0 to 63. The pi output array is loaded from the MEM array as shown below.

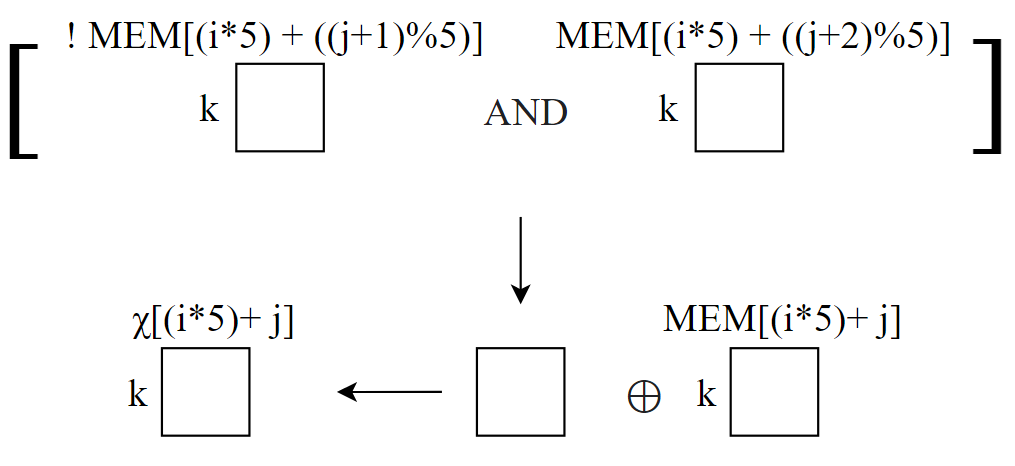


*Fig. 7 – Demonstration on how the pi array is loaded from the MEM array where i is the outer loop variable, j is the inner loop variable, and r is the 25-elemtn long array consisting of the pi permutation constants shown in Table 3.*

Once the pi output array is loaded, the array is passed onto the next permutation stage, chi.

### Chi:

The chi permutation stage performs XOR operations on elements of MEM ANDed together and stores them directly to the chi output array. A triple for loop where the outer loop iterates from 0 to 4, the middle loop iterates from 0 to 4, and the inner loop iterates from 0 to 63 is used to perform these operations. Figure 8 illustrates how this permutation is executed.

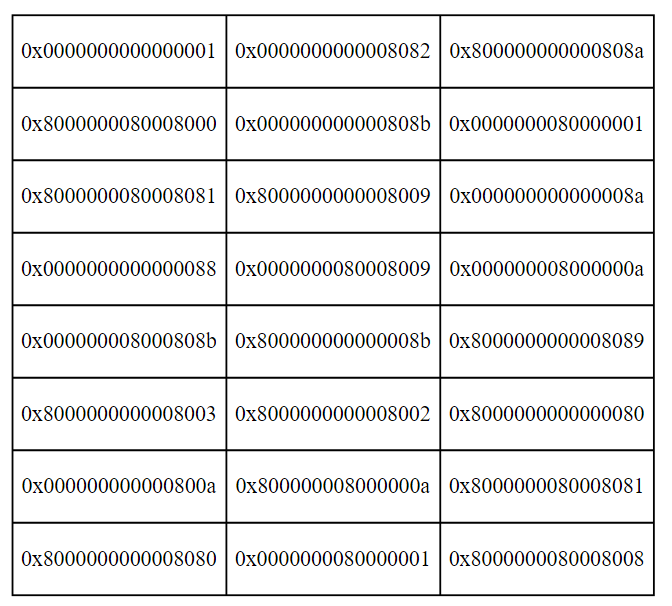


*Fig. 8 – Demonstration on how the chi output array is loaded in with data from MEM and array D where i represents the outer loop variable, j represents the middle loop variable, and k represents the inner loop variable.*

The chi output array is now loaded and is ready to be passed onto the final permutation stage, iota.

### Iota:

In the final permutation stage, iota, MEM is XORed with values from the iota constants table shown below. The iota stage, unlike the 4 other stages, takes in an integer argument that represents the current round of permutation. Based on the round count, a constant is chosen from the iota constants table shown below, and XORs the constant with the first column of MEM.



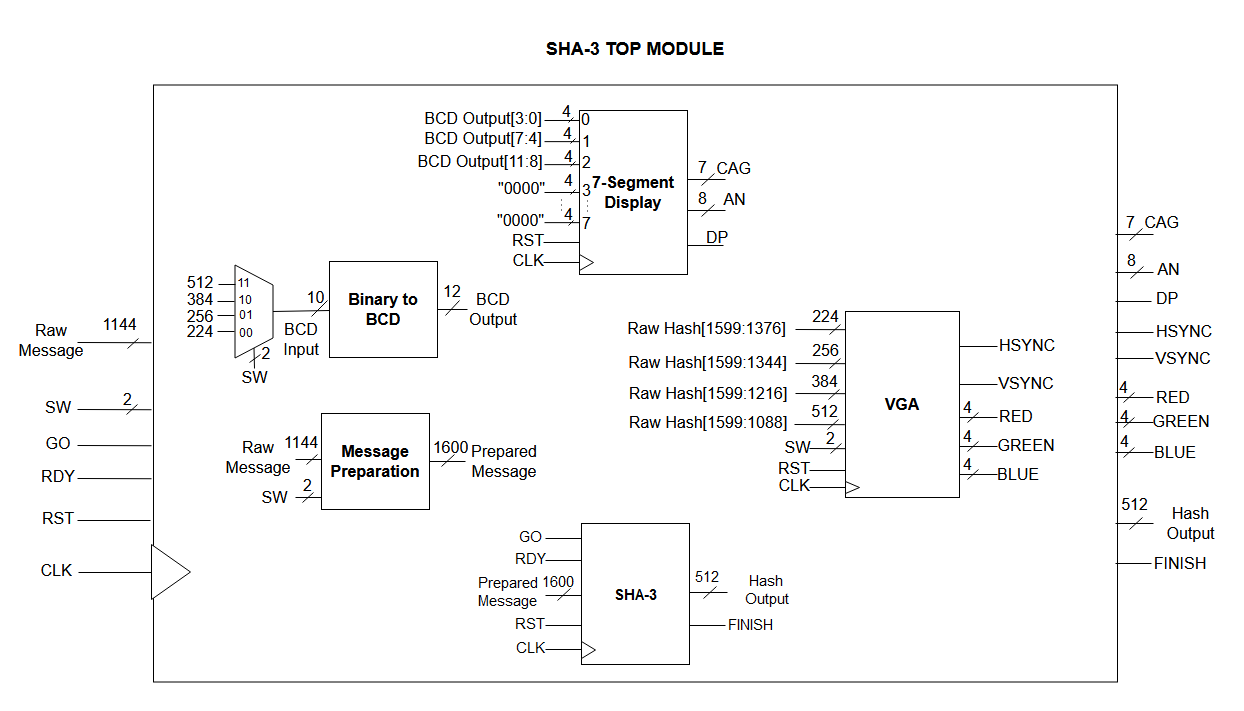
*Table 4 – Iota permutation constants from Keccak Team [3]*

As seen in table 4, there are 24 iota permutation constants: one for each of the 24 rounds of Keccak permutations. Each constant is a 16-character hex value which is 16 \* 4 or 64 bits long.

In a double for-loop, the 64 bits of the first column of MEM is XORed with one of the hex values from the iota permutation constants table. After this permutation, the MEM is converted back into a 1-D 1600-bit array, the form the input bits are in. This 1600-bit array is the output of a round of Keccak permutations. This process is repeated for 24 rounds. The implementation of the state machine to handle the 24 rounds of permutations is discussed in the design section.

# Design

To design SHA-3 onto the Nexys A7 FPGA, it was best to approach it modularly. To break down the components of the design, series of parts are seen as black boxes and diagramed in order to find what inputs and outputs are passed between.

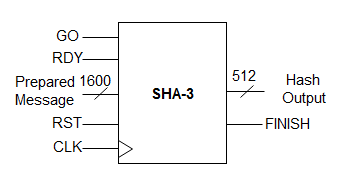


*Fig. 9 – Blackbox representation of each component combined into a top module to make up SHA-3.*

The design in Figure 9 is how the SHA-3 implementation was envisioned. Once this design is complete, each component is worked on in order to meet their functionalities.

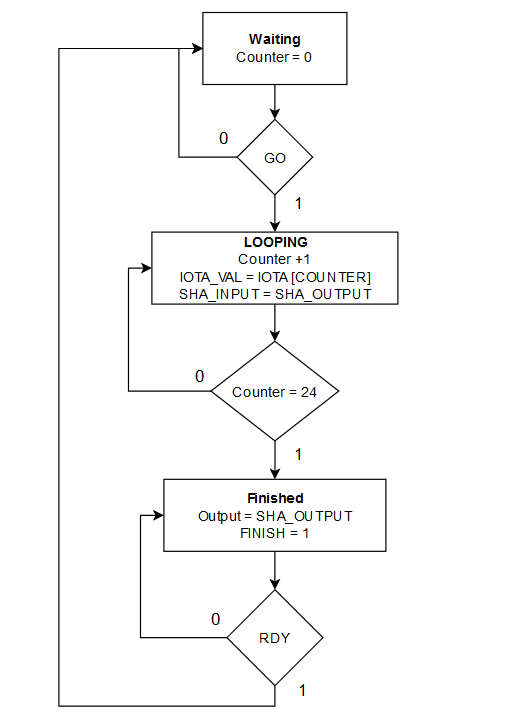
## SHA-3 State Machine / Permutation

The overall design of the permutations is illustrated in Figure 10.



*Fig. 10 – SHA-3 Permutation module design.*

A state machine is set up within this module to allow the input to be permutated by the Keccak permutation stages for 24 rounds. The state diagram is shown in Figure 11.



*Fig. 11 – SHA-3 permutation rounds state machine diagram.*

The state machine starts in a waiting state, waiting for the GO signal to start looping through the permutation rounds. On each round of permutation, the module invokes the Keccak permutations module, passing in the current value of the permutated 1600-bit input as well as the iota constant based on the round count. The output of the Keccak permutation is either an intermediate output of 1600 bits, which are passed through the permutation stages for another round, or the final output after the 24 rounds are complete. The state machine waits for 24 rounds until it moves on to the done state. When the state machine gets to the done state, the finish flag is set, and the final output is outputted as a 1-D 1600-bit array. The state machine now waits for the RDY signal in order to go back to the waiting state. The most significant 512 bits of the final hash value is outputted out of the module, since this is the largest output that the SHA-3 mode supports.

## VGA Display

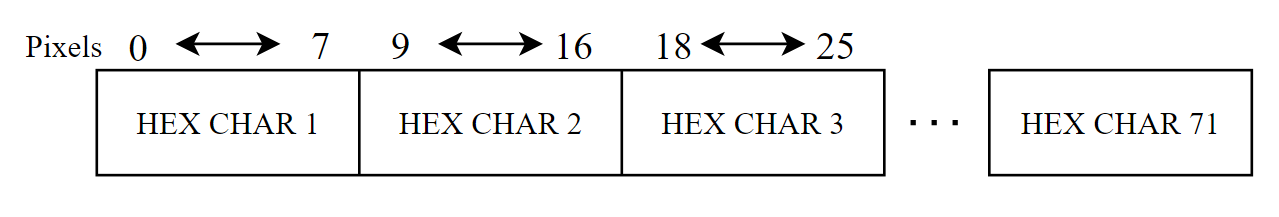
The output hash of length x-bits, where x is the SHA-3 output size, is displayed on a monitor via VGA as hex characters to visualize the project output. This is done by creating 8-bit by 8-bit sprites of hex character 0 through F in a 2-D vector. The depth of this 2-D vector is 8 bits \* 16 = 128, and the width is 8 bits. Each hex value is stored in order inside the 2-D vector, sprites, where the first vector of any hex character sprite, i, can be accessed at row i\*8.

Another 2-D array of size 31 by 640 bits array is created as a canvas for the hex character sprites to be copied onto.

A VHDL module is created that takes in a x-bit input and a 5-bit address input and outputs a single vector of 640 bits.

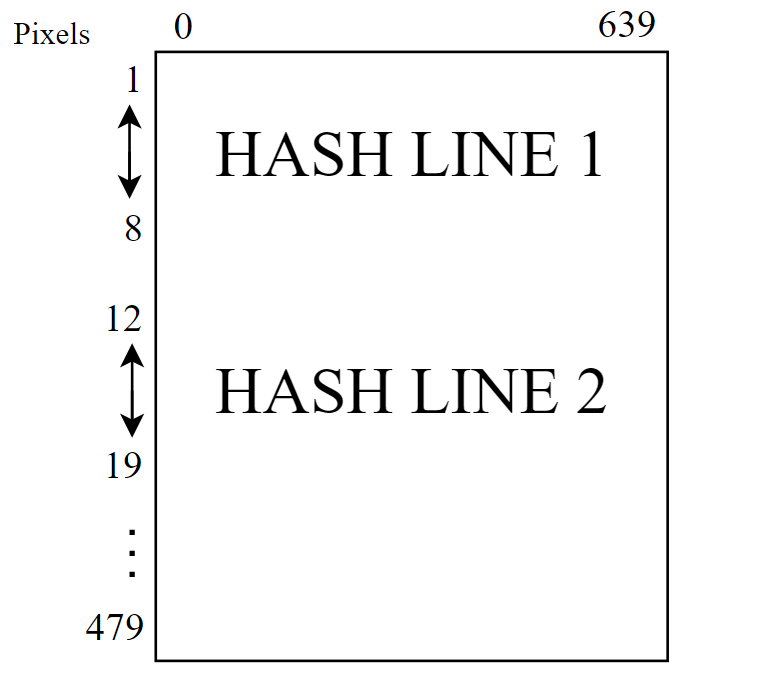
The bits of the x-bit hash input are read 4 bits at a time and converted from binary to integer. This integer is used to access the correct position of the sprites vector to get access the first vector of the hex character sprite that correlates to the 4 bits read from the hash. Another loop writes the entire 8 rows of the hex character sprite onto the canvas array. This process repeats itself n-times, where n is the number of hex characters needed to represent the x-bit input hash. On each clock cycle, the module outputs a single row of the canvas depending on the 5-bit input address to be displayed.

For our VGA resolution of 480 by 640 pixels, there is around a 70-character limit per 640-pixel row when each hex character takes up 8 pixel and there is a 1-pixel gap between each character.



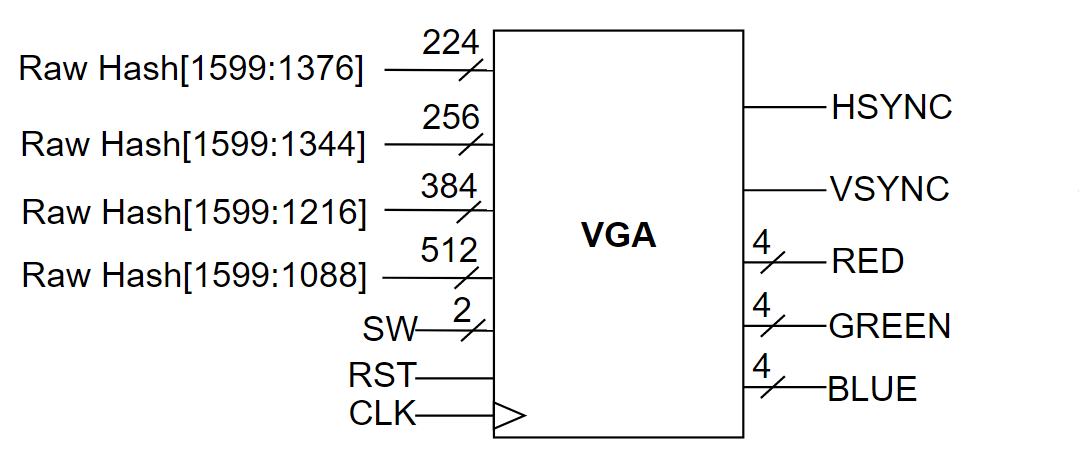
*Fig. 12 – Horizontal positioning of output hash characters on the display.*

For 384-bit and 512-bit outputs, the hex character output exceeds this character limit and is therefore split into two rows of even lines on the display. For 226-bit and 256-bit outputs, only the top line is visible because they can both be represented in less than the 70 hex characters.



*Fig. 13 – Vertical positioning of hash character outputs on the display.*

From the top module, the top VGA top module block design is examined to see how the hex char outputs are displayed.



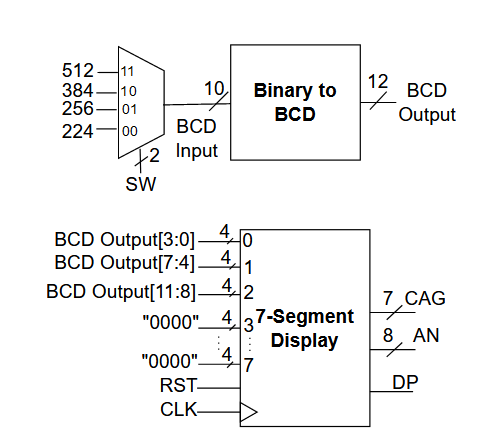
*Fig. 14 – VGA module design.*

The inputs going into the VGA module are a truncated version the 1600-bit output from the 24 rounds of Keccak permutations. Depending on the switch inputs, the size of the output hash is determined. The first x bits are taken from the 1600-bit output where x is the output size in bits. Since the hex message is always white in color, the RGB bits are fixed at 1.

There is an internal clock divider within the VGA module that takes a quarter of the speed of the original 100Mhz clock. The clock divider simply counts for 25 cycles of the original 100mhz clock. This allows the VGA module to run at 25Mhz and output the horizontal and vertical sync signals at a functional speed.

## 7-Segment Display and Switches

The system also implements the I/O switches and 8-digit 7-segment display on the FPGA. Since SHA-3 is capable of four different hashing sizes, the switches and display are used to convey to the user what mode is currently being used.



*Fig. 15 – 7-segment Display module design.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Switch Input (2-bit value) | | | |
| “00” | “01” | “10” | “11” |
| Hash Mode | SHA-3 224 | SHA-3 256 | SHA-3 384 | SHA-3 512 |
| 7-Seg Display  Value | 224 | 256 | 384 | 512 |

*Table 5 – Hash and 7-segment display settings versus switch input.*

The hash mode affects the output of the message generation component, as well as the final output size displayed to the user on the screen via VGA. The 7-segment display value only uses the lowest 3 digits, meaning that the rest are disabled.

# implementation with microblaze

With the design of the SHA-3 module, it could be implemented and wrapped with AXI in order to communicate with the MicroBlaze microprocessor. To do this, a custom intellectual property (IP) is made, which allowed our SHA-3 with AXI module to connect to the MicroBlaze microprocessor via block diagram.

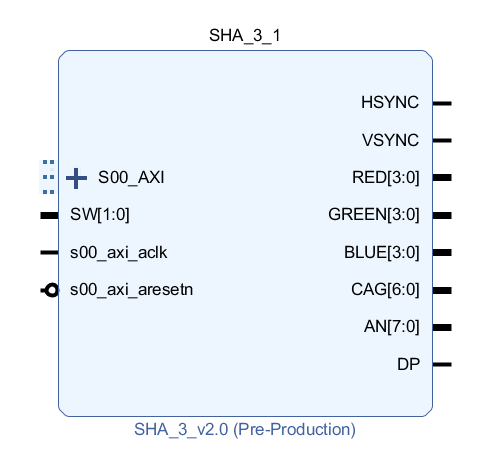
## Custom Intellectual Property (IP) with AXI

To create a custom IP that can have the microprocessor communicate with the SHA-3 algorithm, the AXI bus has to be configured to send data to our module. Since our module takes in the 1152 bits of data as our message input, as well as numerous other control signals such as GO and RDY, AXI has be configured due to its max register size of 32-bits. AXI is capable of having between 0-512 registers that are 32-bits wide. Provided the design of the SHA-3 module, the AXI registers are mapped in Table 6 in order to accommodate for all the inputs and outputs of the SHA-3 IP.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **DATA BITS** | | | |
| Bit 31-3 | Bit 2 | Bit 1 | Bit 0 |
| **REGISTER 0** | SHA3 INPUT [1151-1120] | | | |
| **REGISTER 1** | SHA3 INPUT [1119-1088] | | | |
| **…** | … | | | |
| **REGISTER 35** | SHA3 INPUT [31-0] | | | |
| **REGISTER 36** | SHA3 OUTPUT [511-480] | | | |
| **REGISTER 37** | SHA3 OUTPUT [479-448] | | | |
| **…** | … | | | |
| **REGISTER 51** | SHA3 Output [31-0] | | | |
| **REGISTER 52** |  |  | RDY | GO |
| **REGISTER 53** |  |  |  | FINISH |

*Table 6 – AXI register map for SHA-3 module.*

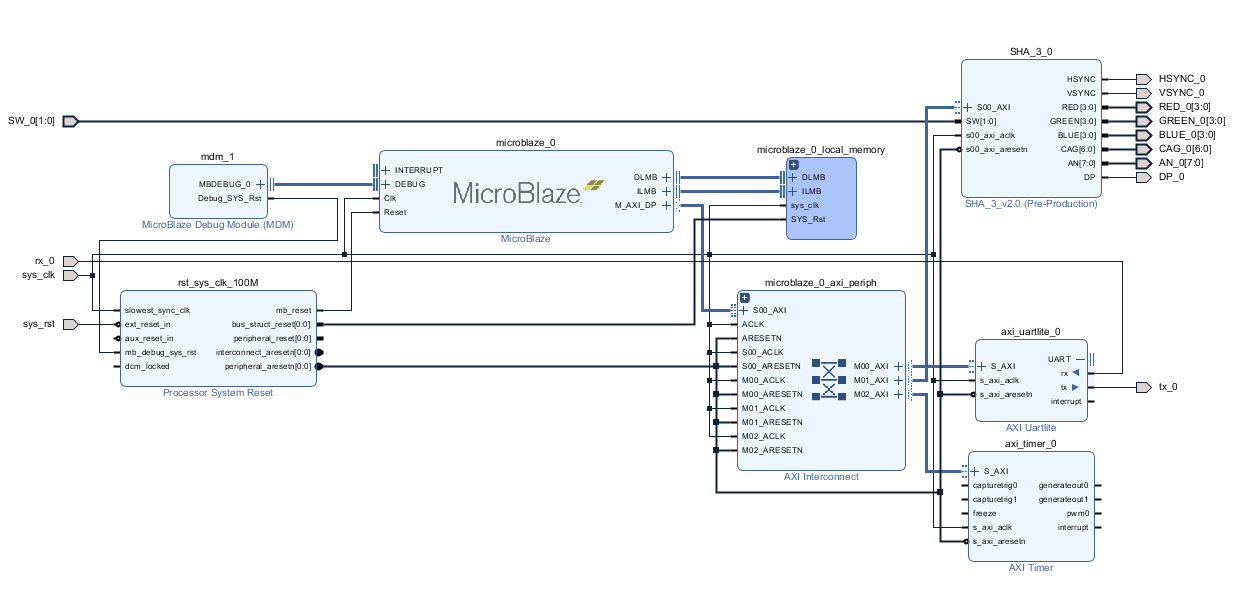
For the ports that do not require manipulation with the microprocessor, such as the switch inputs, 7-segment outputs, and VGA outputs, those are directly added to the IP block themselves. The result is the image of the IP block in Figure 16.



*Fig. 16 – Image of the AXI-wrapped SHA-3 IP block.*

## MicroBlaze Connection

Using Vivado’s Block Design feature, the SHA-3 IP block containing our SHA-3 design can be connected to the MicroBlaze processor. Using the automatic block design and automatic connection tools, Vivado also adds any subcomponents that may be required. The AXI Timer and AXI Uartlite cores are also added to provide extra functionality. The AXI Timer core is used to measure execution time via the C/C++ overhead, while the AXI Uartlite is used to pass data from the user via UART to the microprocessor.



*Fig. 17 – Image of complete system with the MicroBlaze microprocessor and SHA-3 custom IP.*

With this, the hardware is fully developed and bitstream is able to be generated. Interaction between the system can then be designed in the Xilinx SDK using C/C++.

## MicroBlaze Programming with UART

In the Xilinx SDK, the microprocessor can be programmed using C/C++ to interact with the AXI bus. In addition, the UART module added in the block design can be used to read and/or display data to the user. Using the AXI register map for the SHA-3 custom IP, data can also be read to and from it using predefined methods provided by Xilinx.

The UART module is running at a baud rate of 115200 in order to print at a faster speed and is the main interface between the user and system. On boot, UART is printrf to the terminal asking for the input message in text. Once submitted, the microprocessor processes the message and send it to the SHA-3 custom IP via registers 0-35. Once that is done, the GO flag in register 100 is written to in order to start the process.

The calculation is finished by polling register 53 and finding when the FINISH flag goes high. From then, the hash value can be viewed via VGA and is also printed to the UART terminal.

With the code finished, this executable code is compiled on top of the hardware bitstream and programmed onto the board.

# Results

## Resource Utilization

The resource utilization of the implementation is the most important part of the result. Since the goal is to aim for an efficient and low-cost system, the resource utilization must be justified for its performance. Table 7 holds the values of the look-up tables (LUTs) and flip-flops (FFs), which are the hardware resources of the system.

|  |  |  |
| --- | --- | --- |
|  | Look-Up Tables (LUTs) | Flip-Flops (FFs) |
| Standalone VHDL for SHA-3 | 4,725 | 3,248 |
| Standalone SHA-3 IP with VGA/7-segment display | 21,299 | 4,378 |
| Complete System with MicroBlaze | 22,968 | 5,867 |

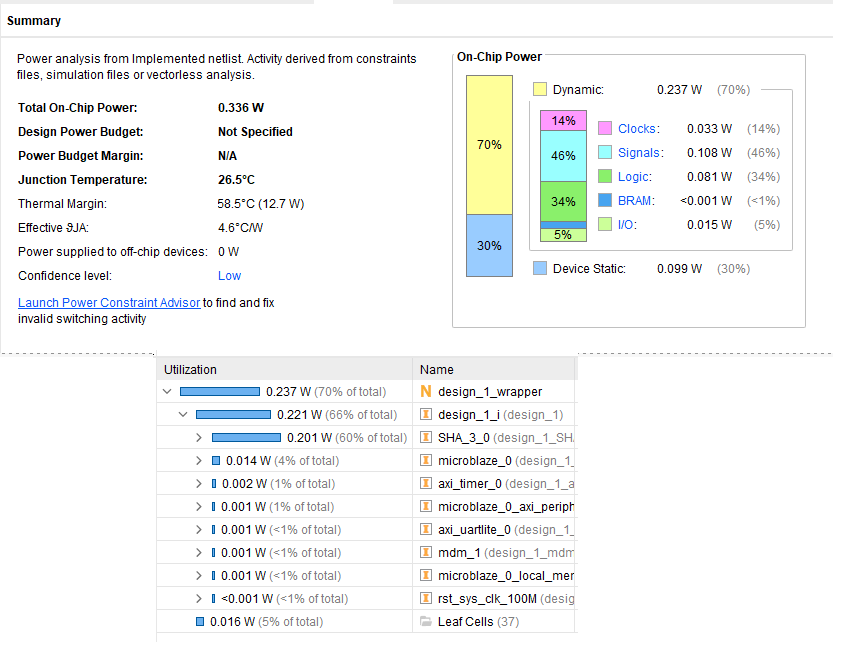
*Table 7 – Resource usage of the different components of the system.*

The standalone VHDL for the SHA-3 component holds a significantly low amount of resource usage. This is beneficial due to the fact that the permutation steps all occur in one clock cycle. This means that it is highly efficient and has a high throughput, as the latency between the SHA-3 input message and hash being generated would be around 24 clock cycles.

As an IP and in conjunction with the MicroBlaze, the system resources increase drastically. This is due to the large sprite system in the VGA core, the AXI wrapper, and the 7-segment display system. These components cause the number of resources to increase greatly, however, but have no impact on the SHA-3 component.

## Power Usage

Power usage is another important component of a system. Not only is a low-cost and efficient design is wanted, but one that requires little to no power is as well. This is one of the benefits of an FPGA, due to their low clock frequency they tend to have lower power usage. This is evident in Figure 18, which lists the power summary provided by Vivado.



*Fig. 18 – Image of power summary details of the system provided by Vivado.*

As it could be seen, the overall power usage of the system is only 0.336W. Broken down, it can be seen that the SHA-3 component uses the majority of this power, at 0.201W for the IP itself. For a system that uses the MicroBlaze microprocessor, VGA, 7-segment display, and is calculating for SHA-3, this is a remarkably low value. This supports the fact that FPGAs provide low-powered yet high throughput solutions.

## Performance Evaluation

Performance is measured using the AXI Timer IP core provided by Xilinx. Using the core, it allows a timer to be initialized, started, and stopped via the MicroBlaze microprocessor and its C/C++ interface. After sending the GO signal via AXI, the timer is started. The FINISH signal is polled via register 53 to check if the calculation is finished. Once it is finished, the timer is stopped and the amount of clock cycles that passed is measured. The result is that the SHA-3 calculation is always done within 110 clock cycles. With a 100 MHz clock with a period of 10ns, this results in the following calculation:

This execution time of 1.10μs is highly efficient, especially noting that the FPGA is running at a 100 MHz processor clock, much slower than complete systems which run in GHz clock speeds.

The throughput is also measured. For each SHA-3 iteration, up to 1144-bit messages can be processed. At 1.10μs execution time per message, the total throughput of the system can be calculated to be 1.04 Gbps.

## Comparisons

The table below shows the execution time for hashing an input message of ‘abc’ and printing the output hash in different high-level languages.

|  |  |
| --- | --- |
| Programming Language | Execution Time |
| C | 2.00ms |
| JavaScript | 2.18ms |
| VHDL System | 1.10μs |

*Table 8 – Performance comparisons based on execution time of different SHA-3 implementations. [1], [2]*

In our FPGA implementation, the execution time is measured to be 1.10μs, almost 200% better than the software implementations. The performance is significantly better in the hardware implementation while the FPGA implementation included extra features such as VGA display, variable input size, and seven-segment displays.

## Resource Evaluation

The results of this system and other related works comparable in terms of resources when compared together. Table 9 visualizes the resource usage of these, including the LUTs and FFs. However, the other systems do not provide a power usage, so the metric could not be compared together.

|  |  |  |
| --- | --- | --- |
|  | Look-Up Tables (LUTs) | Flip-Flops (FFs) |
| Standalone VHDL for SHA-3 | 4,725 | 3,248 |
| Standalone SHA-3 IP with VGA/7-segment display | 21,299 | 4,378 |
| Xilinx Vendor IP Core [8] | 7,150 | 3,200 |
| Xiphera IP Core [12] | 4,175 | 2,797 |

*Table 9 – Resource comparison of this system versus other related works.*

An important note is that the Xiphera core only supports SHA-3 with a 256-bit output, meanwhile the core developed in this design is able to accommodate for four different outputs. In addition, the IP developed in this project also holds the VGA and 7-segment system, increasing the number of resources compared to only the SHA-3 component of it.

Since the design in this project targets a 100 MHz clock system, it provides an efficient output while maintaining a comparable resource footprint.

# Conclusion

In conclusion, the performance of the SHA-3 implementation onto the Nexys A7 FPGA with the MicroBlaze microprocessor proved successful. When viewing the performance metrics of other implementations and their hardware capabilities, it is clear that SHA-3 on the FPGA is comparable, if not better. The hardware acceleration of the FPGA allows for a tremendous improvement in system performance of the execution time of the algorithm. In addition, the FPGA is able to support many other features such as VGA and 7-segment display outputs, as well as completely supporting the MicroBlaze microprocessor with UART. Ultimately, servers and systems that aim to potentially increase the performance of hashing using SHA-3 could look towards FPGA acceleration cores.

# References

[1] C. Veness, “Movable Type Scripts,” *SHA-3 Cryptographic Hash Algorithms implemented in JavaScript | Movable Type Scripts*, 2020. [Online]. Available: https://www.movable-type.co.uk/scripts/sha3.html. [Accessed: 2021].

[2] E. Chou, “euugenechou/sha3-256,” *sha3-256*, 05-Apr-2021. [Online]. Available: https://github.com/euugenechou/sha3-256. [Accessed: 2021].

[3] “Keccak Specification Summary,” *Keccak Team*. [Online]. Available: https://keccak.team/keccak\_specs\_summary.html. [Accessed: 2021].

[4] “MicroBlaze Processor Reference Guide UG984 v2020.2.” XILINX, 18-Nov-2020.

[5] R. A. Grimes, “Why aren't we using SHA3?,” CSO Online, 21-Feb-2018. [Online]. Available: https://www.csoonline.com/article/3256088/why-arent-we-using-sha3.html#:~:text=The%20number%20one%20reason%20why,device%20you%20owned%20or%20used. [Accessed: 2021].

[6] S. Jones, “What You Need to Know About SHA-3 for Embedded System Security,” *ElectronicDesign*, 20-May-2019. [Online]. Available: https://www.electronicdesign.com/technologies/embedded-revolution/article/21808025/what-you-need-to-know-about-sha3-for-embedded-system-security. [Accessed: 2021].

[7] R. S. U. 4/16/2021, “98 Must-Know Data Breach Statistics for 2021: Varonis,” *Inside Out Security*, 16-Apr-2021. [Online]. Available: https://www.varonis.com/blog/data-breach-statistics/. [Accessed: 2021].

[8] “SHA3/Keccak,” *Xilinx*. [Online]. Available: https://www.xilinx.com/products/intellectual-property/1-175rk79.html#overview. [Accessed: 2021].

[9] “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.” National Institute of Standards and Technology , Aug-2015.

[10] “Using fpgas.” [Online]. Available: https://www.sparkfun.com/fpga. [Accessed: 2021].

[11] “Vivado Design Suite AXI Reference Guide UG1037

(v4.0).” XILINX, 15-Jul-2017.

[12] “XIP3032H: SHA3-256 Secure Hash Algorithm-3 IP

(256 bit digest) IP Core” *Xiphera.* [Online]. Available: https://xiphera.com/product\_brief/XIP3032H\_PB.pdf