FPGA implementation of MD6 hash algorithm

Team students: Aviel Birduaker, Ofek Sharabi

Supervisor: Mr. Uri Stroh

University name: JCT – Jerusalem College of Technology

June 21, 2023

Abstract

MD6 is a cryptographic hash function - designed by Ronald Linn Rivest - to provide highly secure message digest functionality. It is a modern hash function designed as a candidate for the SHA-3 competition [1] that builds upon the previous MD4, MD5, and SHA-1 hash functions, while addressing many of their weaknesses. MD6 employs a Merkle-Damgård construction [2], which means that it processes input data in blocks, compressing each block into a fixed-size output. The MD6 algorithm features a flexible block size, allowing it to efficiently process data of varying sizes. It also includes several innovative features, such as a parallel compression function and a tree-based message update algorithm. In our project, we have implemented a single compression function of the MD6 algorithm. The use of Vivado, a software suite developed by Xilinx Inc., proved to be crucial in achieving our goal of implementing this algorithm in hardware on the Basys3 FPGA board by Digilent Inc. that is based on the Xilinx Artix-7 FPGA. We started with a prototype design that did not meet the specified requirements of the Basys3 where its resources are still limited compared to those on professional-grade FPGAs. Therefore, we had to be mindful of resource constraints and find ways to optimize our MD6 design to meet spec.

1. Introduction [2]

1.1 MD6 IO

The inputs to the MD6 are as follows:

- M – the message to be hashed (mandatory). Its length is up to 4096 bits.
- d – message digest length desired, in bits (mandatory). SHA-3 provides four different output lengths, denoted as SHA3-224, SHA3-256, SHA3-384, and SHA3-512. The numbers in their names indicate the output lengths in bits, so the respective output lengths are 224 bits (28 bytes), 256 bits (32 bytes), 384 bits (48 bytes), and 512 bits (64 bytes) [3].
- K – key value (optional). Its length is up to 512bits.
- L – mode control (optional) determines the number of levels in the hash tree that is used to compute the hash.
- r – number of rounds (optional) determines the number of iterations of the basic MD6 compression.

The output of MD6 is a bit string of d bits length.

1.2 compression function

The MD6 hash function uses a recursive construction to generate a binary tree of hash values, where each level of the tree is generated by applying the compression function to a block of data. The compression function takes a block of 89 64-bit words, referred to as the "N" parameter. This block is divided into two parts: the first 64 words are used as data, while the remaining 25 words are used for additional purposes, such as message padding or metadata. It operates by using a combination of logical operations such as AND, XOR, along with "shift amounts", to mix the input data block with the current state of the hash function and to apply a permutation function to the mixed data, resulting in a new hash state.

As mentioned, the 64 words input are fed in, and 16 hashed words are generated. Each round of the MD6 compression function corresponds to one clock in hardware, and each round is composed of 16 steps. Each step corresponds to one calculation that operates on one word of the input block, using logical operations to mix the data and the current state of the hash function. The result of each step is then used as input to the next step of the next round, resulting in a sequence of 16 steps that comprise one round of the MD6 compression function. If there are a lot of rounds in the MD6 compression function, it can increase the security of the hash function by making it more difficult for attackers to find collisions or other weaknesses [4]. However, using many rounds can also increase the computational cost of generating the hash, which can impact performance.

Back to the compression function inputs [2], the mode of operation of MD6 involves formatting the input to the compression function f in a specific way. This involves using a total of 89 words, which are categorized into five parts. The first four items, Q, K, U, and V, are known as auxiliary inputs, while the last item, B, contains the actual data payload.

The constant vector Q has a length of q = 15 words and approximates the fractional part of the square root of 6. The "key" K, which has a length of $k = 8$ words, serves various purposes such as a salt, or secret key. It contains a supplied key of $keylen$ bytes.

The "unique node ID" U is a one-word auxiliary input that serves as an identifier for the particular compression function operation being performed. It consists of two parts: the level number and the index within the level. l is the level number, represented by one byte, while i is the position within the level, represented by seven bytes. The first compression function operation is labelled as i = 0.

The "control word" V is an additional one-word auxiliary input that plays a crucial role in providing essential parameters relevant to the computation process. It consists of several components, including:

- The number of rounds in the compression function, represented by r, which uses 12 bits.
- The mode parameter, L, which determines the maximum level and uses 8 bits.
- The value of z, which is set to 1 if this is the final compression operation and 0 otherwise. It uses 4 bits.
- The number of padding data bits, p, in the current input block B, which uses 16 bits and indicates the number of zero bits appended to the end of B.
- The original length of the supplied key K, represented by $keylen$, which uses 8 bits and is measured in bytes.
- The desired length of the digest output, represented by d, which uses 12 bits and is measured in bits.

Internally, the compression function consists of a main loop of r rounds, each consisting of c = 16 steps, followed by a truncation operation that reduces the result to c = 16 words. The main loop involves a total of $t = rc$ steps, with each step computing a one-word value. This loop can be implemented as a nonlinear feedback shift register with 89 words or by loading the input into the first n words of an array A of length $n + t$ and computing each of the remaining t words in turn. The truncation operation returns the last 16 words of A as the compression function output, also known as the "chaining variable." The compression function always outputs c = 16 words (1024 bits) for this chaining variable. The compression function takes the "feedback tap positions" t_0, t_1, t_2, t_3, t_4, each in the range of 1 to n-1 = 88, as parameters (those taps are not related to the number of steps t).

The round constants, denoted by S_j, introduce variability between rounds. Each round j has a unique constant S_j, which remains constant throughout the steps within that round.

The main computation loop is represented in equation 1. It is also can be viewed as a nonlinear feedback shift register as shown in figure 1.

```
$for\ i = n\ to\ (t + n - 1)$
{
    $x = S_{i-n} \oplus A_{i-n} \oplus A_{i-t_0}$
    $x = x \oplus (A_{i-t_1} \wedge A_{i-t_2}) \oplus (A_{i-t_3} \wedge A_{i-t_4})$
    $x = x \oplus (x >> r_{i-n})$
    $A_i = x \oplus (x << l_{i-n})$
}
```

Equation 1 Main computation loop

There is no dependence of one step on the output of another for at least c steps since $c \le min(t_0, t_1, t_2, t_3, t_4)$. As the steps within a round are independent, they can be executed simultaneously, allowing all steps within a round to be performed in a single clock tick. Consequently, a complete compression function can be computed in r clock cycles.

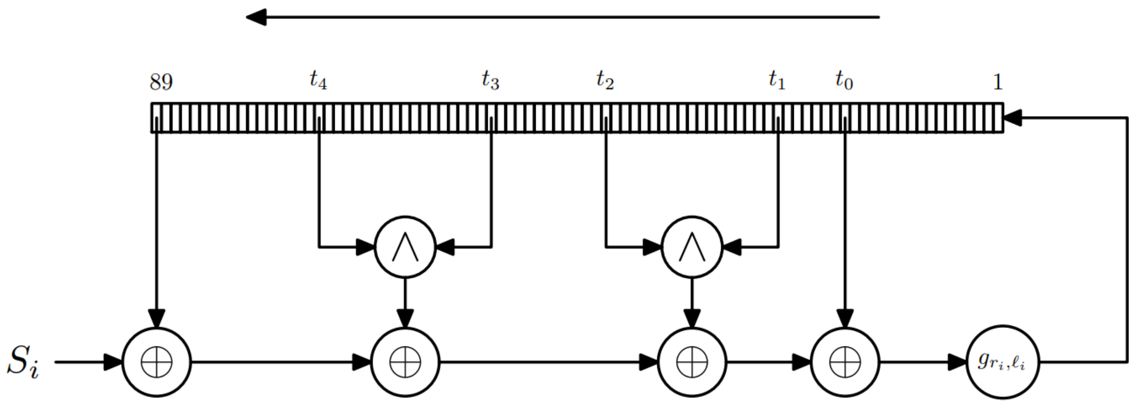


Figure 1 The main computation loop of the compression function viewed as a nonlinear feedback shift register [2]

The project's primary contributions can be summarized as follows:

- Speed: FPGAs are hardware devices that can perform multiple operations simultaneously, making them highly parallelizable. By implementing the MD6 algorithm on an FPGA, we took advantage of its parallel processing capabilities, resulting in faster execution times than software implementations running on traditional processors.
- Customizability: FPGAs are programmable hardware devices, which means we could customize the MD6 implementation to meet our specific requirements. By adjusting the iterative implementation, data path width, and other parameters, we could optimize the design to reduce resource utilization.
- Lower power consumption: FPGAs are designed to be power-efficient and can provide better performance per watt compared to general-purpose processors. By implementing MD6 on an FPGA, it's possible to reduce power consumption, making it suitable for energy-constrained environments or mobile devices.

Our responses of Xilinx Open Hardware as follows:

- **Technical Complexity:** Some of the challenges that we faced when implementing a single compression function of MD6 hash function in hardware included the managing of 4096-bits input data and 512-bits output data flow where they had to be carefully managed to ensure that the correct data is fed into the compression function and that the correct output data is generated. Moreover, we commenced our endeavor with a prototype design, only to discover that it did not align with the specified requirements of the Basys3. The Basys3, being equipped with limited resources in contrast to professional-grade FPGAs, presented us with significant constraints. As a result, we had to diligently address these resource limitations and implement optimization strategies for our MD6 design to meet the required specifications.
- **Re-usability and Implementation:** We successfully implemented a complex project on the Basys3 FPGA board by Digilent Inc. that is based on the Xilinx Artix-7 FPGA using Python programming techniques to manage the data. This project required a deep understanding of hardware design and software development, and we ensure that every component functioned seamlessly. Additionally, we went above and beyond by adding a GUI to the project, which greatly improved its usability and reusability and with the GUI, users can easily interact with the project.
- **Marketability/innovation:** By efficiently implementing the complex MD6 compression function on the hardware platform of the Basys3 board and adding a GUI for reusability, this innovative solution offers companies a competitive advantage in the marketplace and drives technology development forward.

2. Design

In the following section, we will provide a detailed description of our FPGA-based MD6 implementation, including the design methodology and HW/SW architecture. A blocks diagram is shown in figure 2.

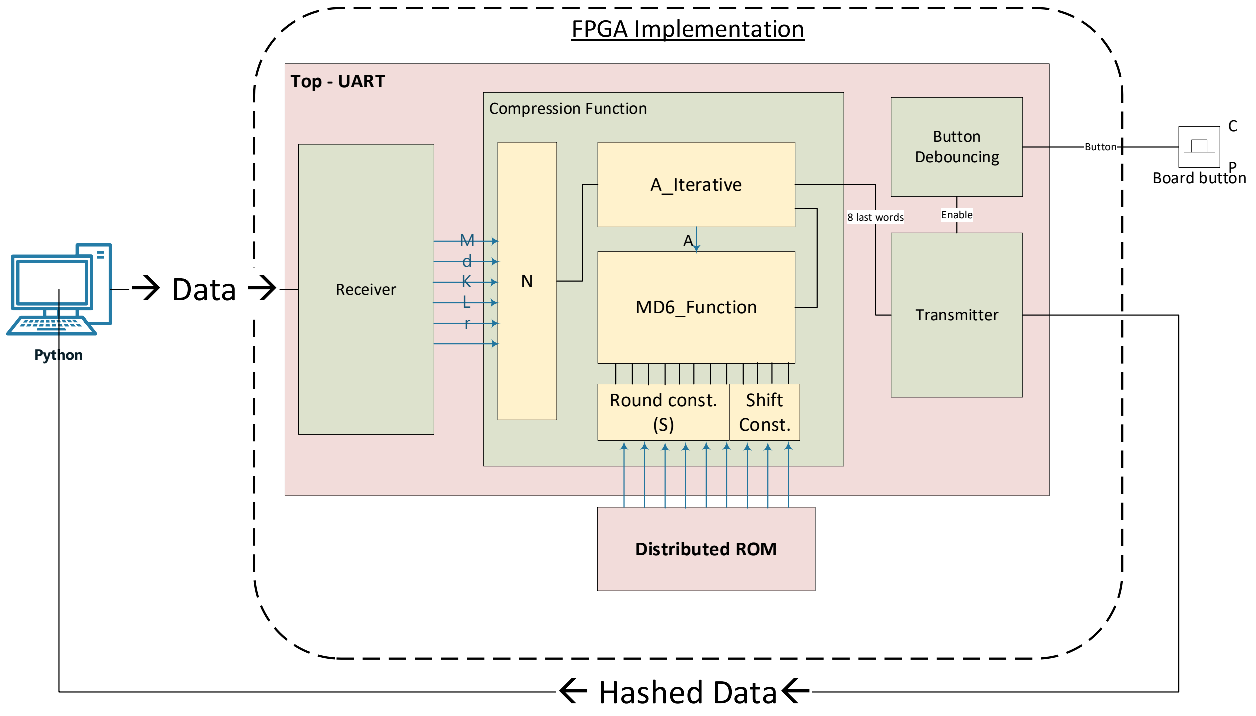


Figure 2 The MD6 compression function design fully represented in blocks diagram. Each color corresponds to a unique hierarchy.

2.1 Hardware

2.1.1 Compression Function

2.1.1.1 Motivation

Initially, the compression function was designed without considering the limitations of the board's resources. The calculations (rotations and padding logics) were implemented using RTL code, operating on excessively long vectors. Moreover, the main computation loop shown in equation 1 was implemented based on sequential logic model for each step out of 16 steps per round. Consequently, these approaches resulted in excessive usage of both Look-Up Tables (LUTs) and Flip-Flops (FFs) as shown in figure 3 exceeding the maximum available resources on the board. Therefore, we needed to change our approach.

In the first stage, we understood that we needed to minimize the number of non-algorithmic calculations, such as calculating zero padding, rotating the bit order of the vector (to receive a big-endian order) or some value of a signal. Alternatively, we utilized the data received from the input of the UART communication protocol and leveraged the processing power of the computer we used to perform these calculations in software. In addition, we used the built-in distributed ROM of the FPGA to avoid overloading the LUT resources.

In the second stage, the main computation loop has been implemented using a combinational logic model. we also noticed that there is redundancy in the algorithm. The calculations are based only on the last 89 words of the *A* vector. Therefore, instead of storing all the unused values and instead of keeping *A* as a large vector of all the words which were calculated, we found a way to implement an iterative approach to save area, by performing a shift operation to insert the 16 new words into a fixed length vector of 89 words.

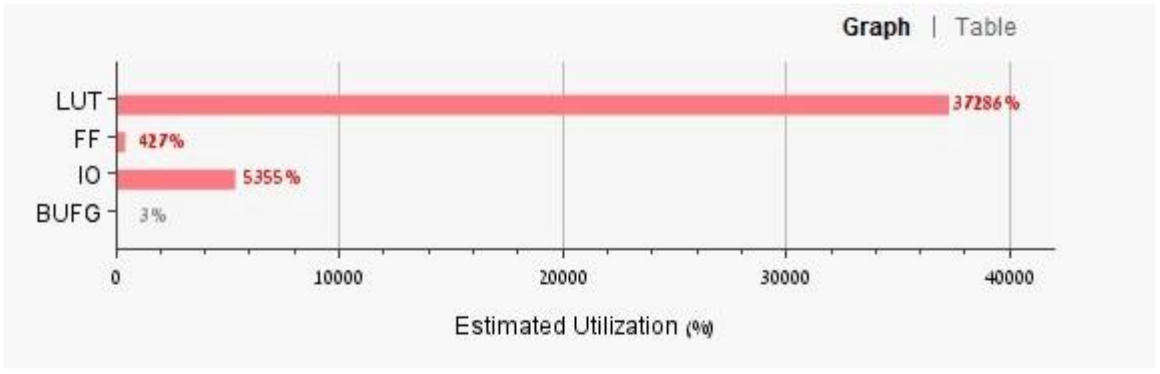


Figure 3 Area utilization graph doesn't meet the requirements. The "IO" resource should be disregarded as the UART architecture is not included in this section of the design.

2.1.1.2 Modules

- CF (compression function) module: serves as the top-level module of the MD6 design. It likely connects the other modules and handles the overall control flow of the design. It also incorporates an FSM - shown in figure 4 - in the A_computation_loop module to control transitions and facilitate iterative operations in the A_iterative module.
- N module: responsible for arranging the first 89 words of the vector A.
- Round Constants module: provides the round constants required for the computations.
- Shift Amounts module: provides the shift amounts required for shifting operations.
- A_computation_loop module: performs a series of calculations for generating the new 16 steps of vector A. It likely utilizes the round constants and shift amounts provided by the respective modules mentioned earlier.
- A_iterative module: handles the iterative operation of updating the vector A. It receives the new 16 steps from the A_computation_loop module and places them at the most significant bit (MSB) part of the A vector. Before placing the new steps, the existing vector A is right-shifted by 16 words.

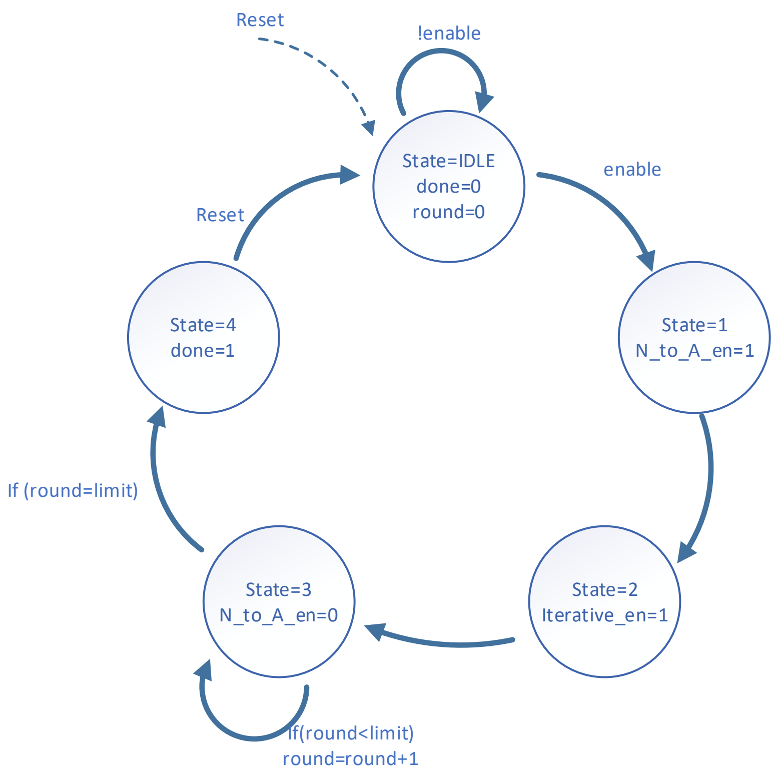


Figure 4 FSM of the compression function

2.1.2 UART

The UART (Universal Asynchronous Receiver Transmitter) provides a simple and reliable way to transmit and receive the serial data between the Basys3 board and the computer through a USB cable. Our Verilog code includes button debouncing, receiver, MD6 top, and transmitter modules arranged in a hierarchical structure.

- Our design incorporates the use of a button on the FPGA board, which enables the transmission of the hashed message. Hence, button debouncing is used in hardware to ensure that the button press is detected only once, even if the button is pressed multiple times or bounces. When a button is pressed, it can cause the electrical contacts inside the button to bounce rapidly, causing the signal to oscillate between high and low states. This bouncing can result in multiple signals being detected for a single button press, which can cause problems for the system that is receiving the signal.

- The receiver is quite sophisticated, and that's because besides being able to receive bytes of data, it can also manage them in a way that helps with preparing for the hashing process. The first block of data that is received, which is 4096 bits in size, is the message, and since the MD6 is defined in a big-endian way where the high-order byte of a word is defined to be the leftmost byte, instead of performing calculations that involve rotation and so on, we made sure that the bytes of the message are inserted in the desired order, and it's just a matter of what position in the vector you place the message. The key is subject to the same rule.

These are the types of data that arrive in this order: Message (4096 bits),d (16 bits) Key (512 bits), L (8 bits), r (16 bits), keylen (8 bits), padding (16 bits) and message index (8bits). All of them are accompanied by signals that blink a light indicating the end of receiving each type of data. Additional details can be found in 2.2 section.

After all the data is received, it enables the compression function to start the hash process.

- The transmitter is designed to send back to the computer the $224 \parallel 256 \parallel 384 \parallel 512$ bits hashed data, with proper byte alignment to ensure correct display of the hashed data on the computer.

2.1.3 Distributed ROM

In the algorithm, there are “round constant” S and “shift amounts” L-shift and R-shift for performing calculations within the compression function to provide some variability between rounds. These S values consist of 168 entries, with each entry occupying 64 bits of memory, and the shift amounts are pre-determined and repeat every 16 steps. To optimize the memory utilization of the FPGA component, we have chosen to store these S values in the distributed ROM memory available within the FPGA.

The decision to use distributed ROM is driven by the need for asynchronous memory that aligns with the timing requirements of our design along with saving the use of the FPGA LUTs resources. Each S value has a depth of 64 bits, resulting in a total width requirement of 168 * 64 bits. However, the maximum width supported by scattered ROM blocks is 1024 bits, equivalent to 16 * 64 bits. To accommodate our specific requirement, we have employed 11 ROM memories. The first ten ROM blocks are fully utilized with dimensions of 64 * 160 bits, while the final ROM block is configured as 64 * 8 bits, with the S values occupying the first 64 * 8 bits and the remaining 64 * 8 bits filled with zeros.

2.2 Software

We have developed a Python code as a wrapper for the hardware implementation. The Python file fulfills several crucial roles:

- Acts as a receiver, accepting input from the user.
- Generates auxiliary data that supports the algorithm's execution.
- Performs necessary adjustments to the received data to align it with the algorithm's requirements.
- Utilizes UART serial communication to transmit the processed data to the board.
- Leverages the TKinter framework to create a user-friendly GUI (Graphical User Interface), allowing users to conveniently access and interact with the algorithm through an intuitive graphical interface.

Below are the descriptions of the functions provided by the Python file:

1. Receiving data:

Prior to entering the required data for the algorithm, the user is prompted to answer "definition questions" aimed at categorizing the data they intend to input in the most appropriate manner. Specifically, the initial questions posed to the user determine the type of data they wish to provide. According to the algorithm's specifications, the user has the flexibility to choose between three options: the key (K), mode control (L), and the number of rounds (r). Once this selection is made, the user proceeds to specify the format in which they prefer to input both the message and the key. The available format options include hexadecimal, binary, and ASCII.

Following the "setting questions," the user will proceed to input M and the K. Subsequently, they will have the opportunity to choose from the available options the desired d, L, and r. It is important to note that the optional parts of data will only be allowed if the user has indicated their interest in providing them during the "setting questions" phase.

2. Production of auxiliary data:

To ensure the algorithm functions accurately, the Python file generates auxiliary data that instructs the algorithm on how to process the received data appropriately. This auxiliary data includes the following:

- Padding Length of M: In order to meet the algorithm's requirements, a padding length for M is determined. This length specifies the amount of padding with zeros required to be added to a portion of the message. This step ensures the algorithm operates correctly and handles the message in the expected manner.
- Message Index: The Python file generates the message index, a crucial piece of information that indicates the number of compression functions to be utilized based on the algorithm's requirements. This information guides the algorithm in applying the appropriate number of

compression functions to the input. Another significant application of this is to determine the appropriate timing for zero-padding the data.

- Length of the Key (K) in Bytes: The algorithm mandates the input K to be provided in a specific byte length. Therefore, the Python file determines the length of the key in bytes, adhering to the algorithm's prerequisite. This data serves as an input requirement for the algorithm to utilize the key effectively.

3. Data Matching:

We employ serial communication using UART, to data transmission from the computer to the board. As a result, the data needs to be converted into byte format since UART exclusively transfers data in this format. Subsequently, prior to transmitting the data to the board, it is necessary to zero-pad each part of the data based on fixed sizes. This padding ensures that the algorithm implemented on the board can accurately categorize the received data into its respective segments. Given that the hardware has predetermined inputs for each part of the data (with fixed maximum values), this approach enables proper classification.

Furthermore, following the padding process, we perform a reversal of each data segment, except for the message (M) and the key (K). The purpose of this reversal is to arrange the data in a more convenient manner for the design. Notably, the message and key are excluded from this reversal since their reception on the board is done in a different way.

4. Transferring data by UART:

To transfer the data, we employ UART serial communication. The user will be prompted to specify the communication port to which the board is connected for data transfer. The relevant COM port can be identified by referring to the computer device manager.

To enable the necessary commands for communication, we leverage the "serial" library. This library provides a comprehensive set of functions. These functions collectively ensure data preparation, communication with the board, and a user-friendly approach, making the Python file a versatile tool for both managing data and providing an accessible interface to the algorithm.

5. User friendly interface:

To ensure a comfortable and user-friendly experience while interacting with the algorithm, we have developed a graphical interface using the TKinter library. Upon entering the interface, users are greeted with explanatory windows and instructions that provide clear guidance on how to proceed with the algorithm.

As outlined in the "Receiving data" subsection, dedicated windows are presented to the user for inputting or selecting the required data. Once the user has entered the data, a prompt will be displayed to select the appropriate COM port for connecting the board. Then, the data will be transmitted to the Basys3 board, where the message will be hashed. Subsequently, by pressing the designated button on the board, a window will appear, presenting the user with the hashed message.

6. Error Handling and User Guidance:

In order to handle situations where the user enters incorrect data, such as an improperly formatted message or key, or selects the wrong COM port, our program incorporates robust error handling mechanisms. When such errors occur, a dedicated error window is displayed, alerting the user of the mistake. To assist the user in rectifying the error, the program is designed to automatically present the same window in which the mistake was made. This allows the user to conveniently correct the data and proceed with the algorithm without having to navigate through multiple windows again.

3. IMPLEMENTATION

The Implementation on the Basys3 board involved utilizing the Vivado software tool. The role of implementation in Vivado encompasses several key tasks such as synthesis, placement, routing and bitstream generation.

3.1 Hardware setup

Our project employs the Digilent Basys3 board, an entry-level FPGA board exclusively designed for the Vivado Design Suite. This board features the Xilinx Artix 7-FPGA architecture, providing exceptional performance and flexibility [5]. Table 1 showcases the key features of the product.

Table 1 The Basys3 key features [5] [6]

Feature	Description
FPGA	XC7A35T-1CPG236C
I/O Interfaces	• USB-UART for programming and serial communication • USB-UART Bridge • 12-bit VGA output • USB HID Host for mice, keyboards and memory sticks
Memory	32 Mbit Serial Flash
Displays	One 4-digit 7-Segment displays
Switches and LEDs	• 16 Slide switches • 16 LEDs • 5 Push-buttons
Clocks	One 100 MHz crystal oscillator
Expansion ports	• Pmod for XADC signals • 3 Pmod ports
Resources	• 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops) • 1,800 Kbits of fast block RAM

3.2 Synthesis

Initially, the RTL (Registration Transfer Level) description of the design is synthesized using the Vivado Design suite, 2022,1 by Xilinx (AMD). This process converts the high-level behavioral code, often written in VHDL or Verilog, into a gate-level representation shown in figure 5 utilizing predefined component libraries. The synthesis optimizes the design for factors like space, performance, and power consumption.

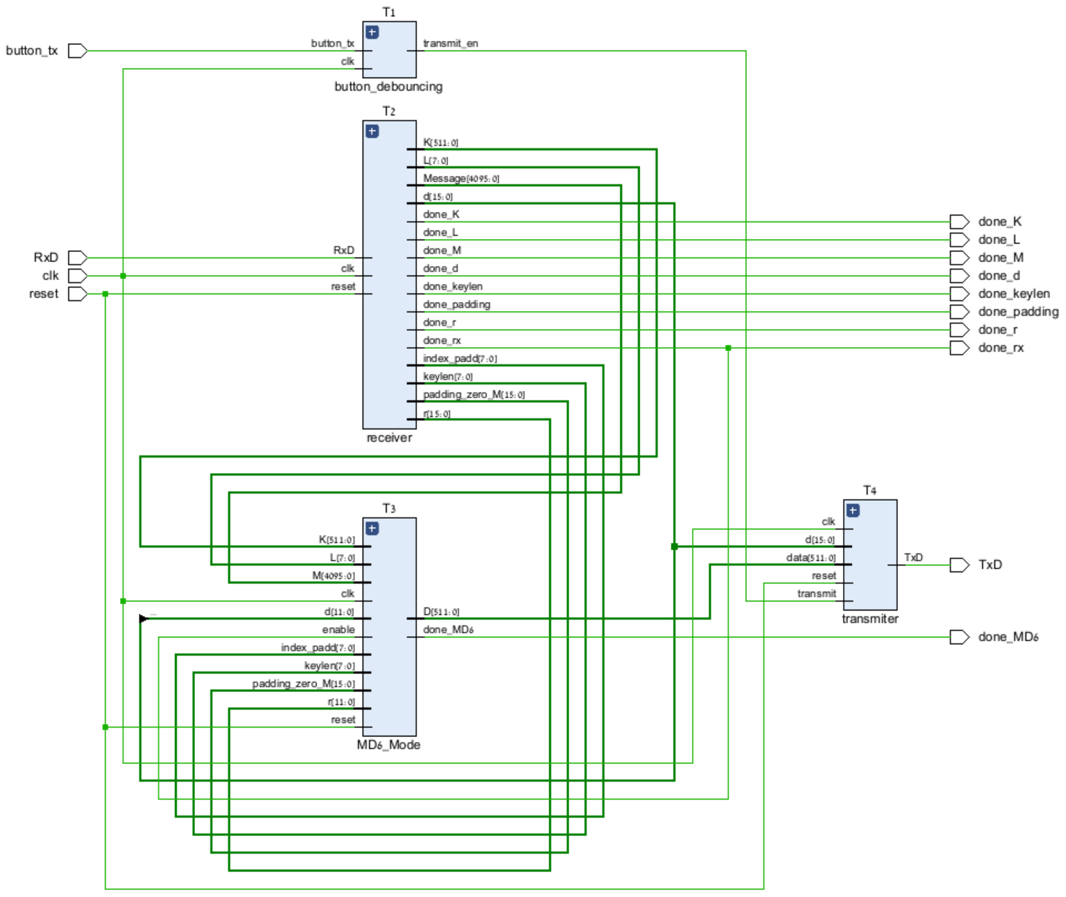


Figure 5 Detailed block diagram of the design

3.3 Placement

Following synthesis, as shown in figure 6, the design is placed onto the physical resources of the target device, including LUTs, flip-flops, and other programmable components. Placement aims to identify an optimal location for each component within the FPGA, considering timing constraints, power optimization, and physical limitations. IO planning within the package is shown in figure 7.

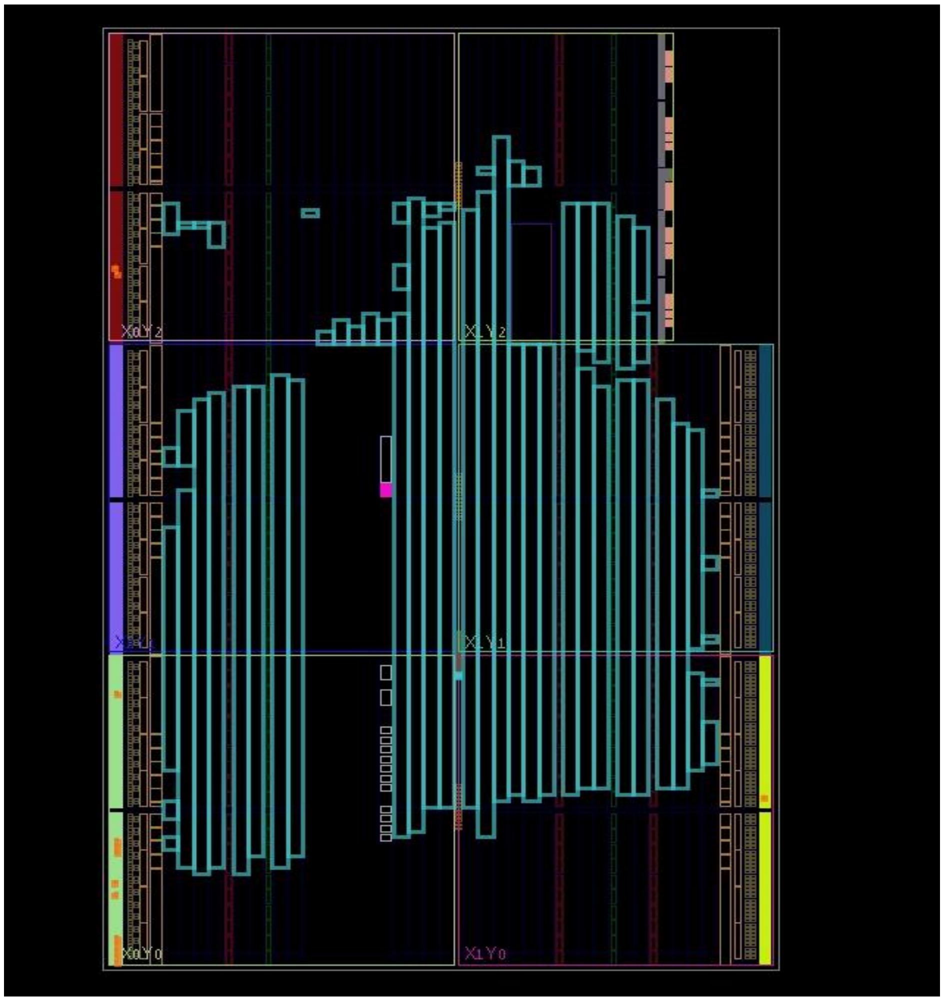


Figure 6 Design placement on the chip

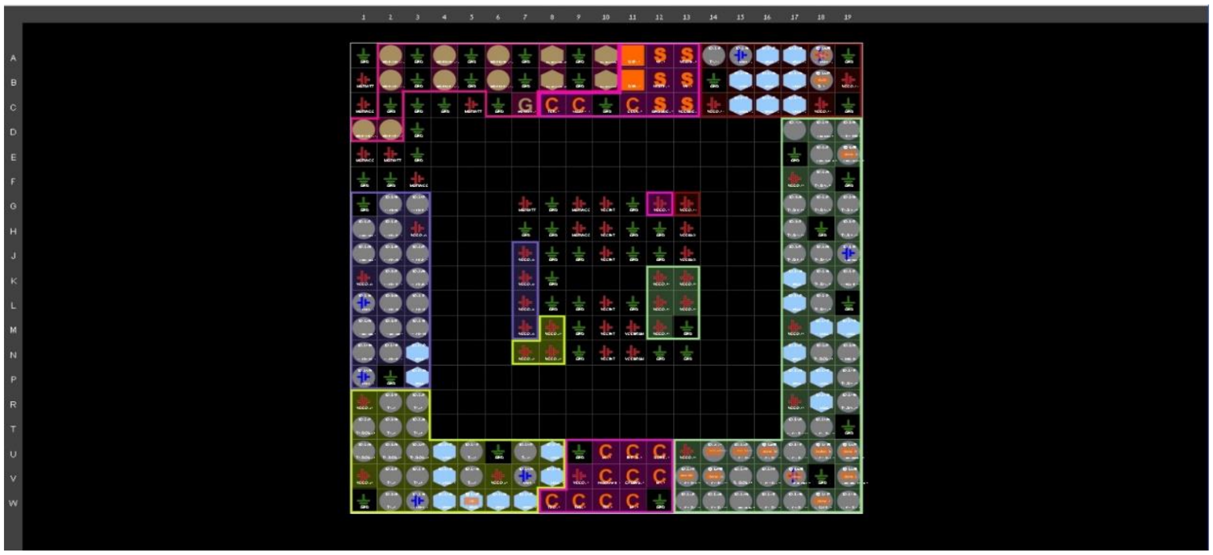
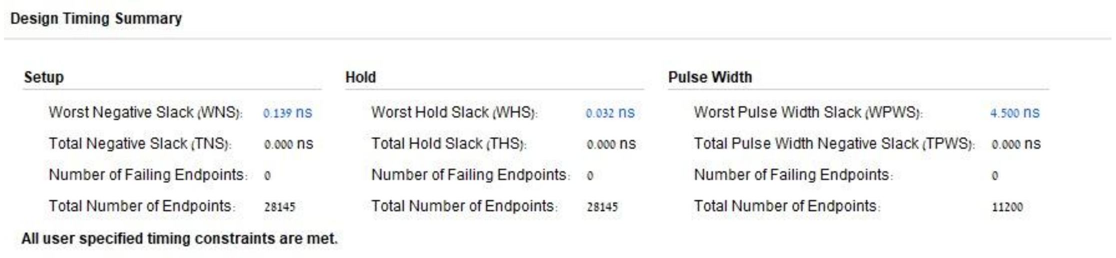


Figure 7 IO planning within the package

3.4 Routing

Once the placement is complete, the routing stage determines how to establish connections among various components and their interconnects within the FPGA. Routing is responsible for establishing physical paths, ensuring signal integrity, timing compliance, and efficient utilization of resources. Those are shown in figure 7 and figure 8.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.139 ns	Worst Hold Slack (WHS):	0.032 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	28145	Total Number of Endpoints:	28145	Total Number of Endpoints:	11200

All user specified timing constraints are met.

Figure 8 Timing analysis results

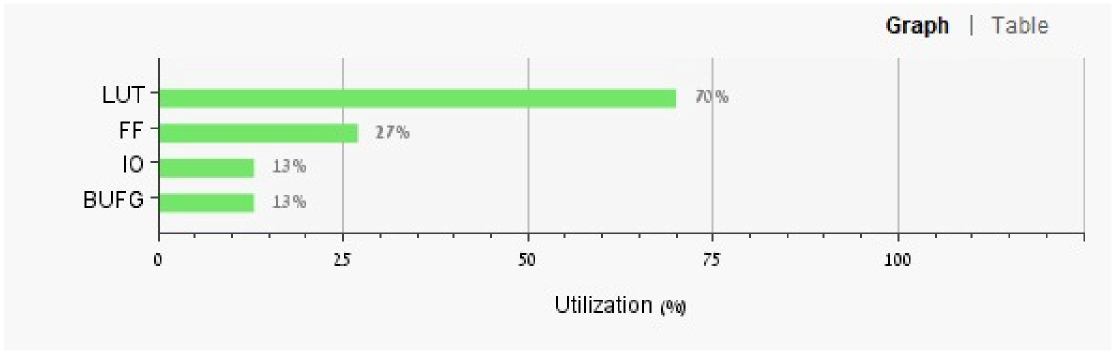


Figure 9 Area utilization graph meets the requirements

3.5 Bitstream Generation

After routing, the implementation process generates a bitstream. This binary file contains the configuration data required to program the FPGA or CPLD, representing the final implementation of the design. The bitstream is loaded onto the target device to make it function as intended.

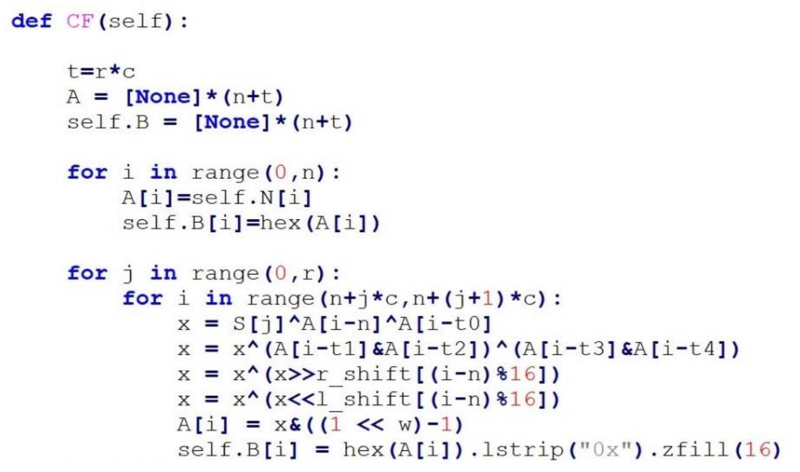
4. Test vectors

The project involved conducting experiments to evaluate the performance of the MD6 algorithm implementation. Test vectors were carefully designed to provide comprehensive coverage of different input scenarios, allowing thorough testing of the hardware implementation. To establish a reference for comparison, the MD6 algorithm was also implemented in software using Python. This software version served as a benchmark and enabled a quantitative evaluation of the hardware implementation's accuracy and efficiency.

In order to ensure the accuracy of the hardware implementation of the MD6 algorithm, a corresponding software implementation was developed for comparison. This was achieved by creating a program that generates random test vectors based on user-defined parameters, which serve as inputs for both the hardware and software applications. By comparing the outputs of both implementations, the correctness of the hardware implementation can be verified.

The software implementation differs significantly from the hardware implementation in terms of the compression function's execution shown in figure 10. In the software application, the compression function is applied to each vector A sequentially, using a loop that iterates a predetermined number of times (specifically, 16 times the number of rounds). On the other hand, the hardware application performs computations in an iterative and parallel manner. In each round, it carries out 16 calculations concurrently, and in the subsequent round, it only utilizes the last 89 words of vector A.

By employing this comparative approach, any discrepancies between the outputs of the hardware and software implementations can be detected, indicating a potential error in the hardware implementation. This methodology allows for thorough testing and validation of the hardware implementation's correctness, providing confidence in its reliability and adherence to the MD6 algorithm specifications.



```
def CF(self):

    t=r*c
    A = [None]*(n+t)
    self.B = [None]*(n+t)

    for i in range(0,n):
        A[i]=self.N[i]
        self.B[i]=hex(A[i])

    for j in range(0,r):
        for i in range(n+j*c,n+(j+1)*c):
            x = S[j]^A[i-n]^A[i-t0]
            x = x^(A[i-t1]&A[i-t2])^(A[i-t3]&A[i-t4])
            x = x^(x>>r_shift[(i-n)%16])
            x = x^(x<<l_shift[(i-n)%16])
            A[i] = x&((1 << w)-1)
            self.B[i] = hex(A[i]).lstrip("0x").zfill(16)
```

Figure 10 The SW implementation based on sequential model

The results obtained from the experiments provided valuable insights into the efficiency of the hardware implementation. These findings helped in identifying potential optimizations and areas for improvement, ultimately leading to a better understanding of the strengths and limitations of the MD6 algorithm when implemented in both hardware and software contexts.

5. Conclusion and Future Work

The successful implementation of MD6 in hardware using the Vivado software suite and the Basys3 FPGA board, based on the Xilinx Artix-7 FPGA, improved performance, and reduced processing time compared to software implementations. Despite the resource constraints of the Basys3 board, careful consideration and optimization techniques were applied to meet the specifications, showcasing the feasibility of FPGA-based designs for cryptographic algorithms.

Future work that can be developed with the design:

- Further optimize the MD6 hardware design to improve performance and resource utilization, taking advantage of advanced FPGA features and optimization techniques.
- Explore the scalability of the MD6 implementation to handle larger data sizes and increased rounds, ensuring its applicability to more demanding cryptographic scenarios.
- Investigate the feasibility of integrating the MD6 hardware implementation into embedded systems or IoT devices to provide efficient and secure message digest functionality.

References

[1] N. I. o. S. a. Technology, *SHA-3 Standard: Permutation-Based Hash and,* Information Technology Laboratory, 2015, p. 2.

[2] R. L. Rivest, "The MD6 hash function A proposal to NIST for SHA-3," Massachusetts Institute of Technology, Cambridge, October 2008.

[3] C. F. M. G. Ewan Fleischmann, *Classification of the SHA-3 Candidates,* Weimar: Bauhaus-University Weimar, Sirrix AG security technologies, 2009.

[4] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, Chapter 9: Hash Functions and Data Integrity: CRC Press, 1996.

[5] X. inc., "Digilent Basys3 Board," Xilinx, [Online]. Available: https://www.xilinx.com/support/university/xup-boards/DigilentBasys3Board.html#overview.

[6] X. inc., "Basys 3 Reference Manual," [Online]. Available: https://digilent.com/reference/programmable-logic/basys-3/reference-manual.