Memristive Architecture for SHA-256

Manikandeshwar Sasidhar
Dept. of Electrical and Computer Engineering
Portland State University
Portland, OR, USA
manikandeshwar@gmail.com

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

Fig. 1. From [1], Symbol for a memristor. Resistance decreases when current flows into the device and vice versa[2].

***Abstract*—SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that produces a fixed-size, 256-bit (32-byte) hash value from an arbitrary amount of input data. This paper describes a way to implement the SHA-256 algorithm on a memristor crossbar using SystemVerilog.**

***Index Terms*—SHA-256, hash function, memristor, crossbar, SystemVerilog**

I. Introduction

Memristors [2] (fig. 1) and memristive devices [3] are innovative structures with a wide range of applications. Essentially, these devices are resistors whose resistance varies based on their usage history, allowing them to store data as resistance values.[1] While memory storage is the most common application, memristive devices are also valuable as functional components in analog circuits, neuromorphic systems, and logic circuits. The approach to using memristors for logic involves interpreting resistance levels as logical states, where high resistance represents logical zero and low resistance represents logical one. In this method, memristors serve as the fundamental components of logic gates, acting as inputs, outputs, computational logic elements, and latches at various stages of computation. This technique is well-suited for crossbar array [4,5,6] architectures and can be seamlessly integrated into standard memristor-based crossbars commonly used for memory storage. This approach is particularly attractive because it allows for the exploration of advanced computer architectures that differ from the traditional Von Neumann architecture. In these architectures, the same devices used for data storage can also perform logical operations, effectively enabling computation within the memory. This paper focuses on this innovative approach. The National Institute of Standards and Technology (NIST) [13] specifies using secure hash algorithms like SHA-1 [14], SHA-224 [15], SHA-256[11], SHA-384, and SHA-512[16]. Hash functions generate message digests during data transmission, essential for secure applications like email and internet banking. They convert messages of any length into fixed-length outputs and are one-way functions, making it hard to reverse or find collisions. This paper will focus on the memristive implementation of SHA-256 using SystemVerilog.

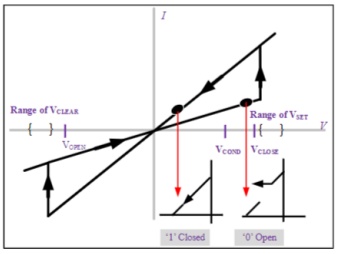

Fig. 2. Voltage vs Current characteristics for a Memristor

II. MEMRISTORS

A. Behaviour and Construction

A memristor's voltage (V) vs. current (I) graph is typically characterized by a pinched hysteresis loop. This distinct loop behavior is a result of the memristor's resistance changing based on the history of the voltage and current applied to it. The loop shape varies with different frequencies of the input signal, generally becoming narrower as the frequency increases.(Fig. 2)

Memristors are nonvolatile and compatible with standard CMOS technologies. These devices are fabricated in the metal layers of an integrated circuit, where memristive effects occur in the oxide between the metal layers (e.g., TiO2 and TaOx)[8] or within the metal layers (e.g., in STT-MRAM[9]). The physical model of a TiO2 memristor is depicted in Fig. 3.

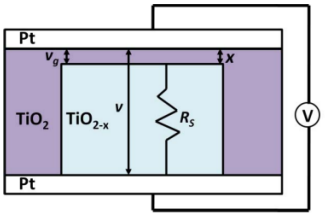

Fig. 3. First observed by HP in 2008 [17], this device consists of a TiO2 layer sandwiched between two titanium electrodes/plates. The conductivity of the structure is attributed to ion transport. The crucial aspect of the switching phenomenon was the formation of a bi-layer comprising two different titanium dioxide species. While TiO2 is electrically insulating (essentially a semiconductor), TiO2-x is conductive due to its oxygen vacancies, which act as electron donors, thereby imparting a positive charge to the vacancies themselves[7].

Case	p	q	p→q
1	0	0	1
2	0	1	1
3	1	0	0
4	1	1	1

Fig. 4. Truth table of the IMPLY function. Mathematically, we can represent this as X = A ¡= B ? 1 : 0, where X = P -¿ Q [5]

Typically, memristors are relatively small, as their fabrication process is akin to processing cross-layer vias between metal layers. Consequently, memristors offer high density and good scalability. The read and write times for these devices can be as fast as 120 picoseconds (fig. 3).

B. IMPLY Logic

The logic function ”p implies q” (fig. 4) [5] also known as ”material implication,” or ”if p then q,” with a corresponding truth table provided in fig. When combined with FALSE (a function that consistently outputs zero), the IMPLY logic function forms a computationally complete logic structure. Because the IMPLY function can be incorporated into a memristor-based crossbar, it serves as a fundamental logic element for circuits based on memristors. By adjusting the values of Vset and Vcond [4], we can obtain the IMPLY logic from just 2 memristors in a medium (Fig. 5.)

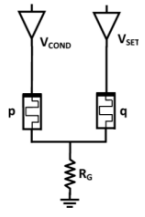

Fig. 5. IMPLY logic gate. The initial state of memristors p and q is the input of the logic gate and the output is the final state of the memristor q after applying the voltages VSET and VCOND. A load resistor RG is connected to both memristors[1].

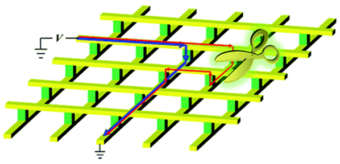

Fig. 6. Crossbar array with individual switches for each of the memristors

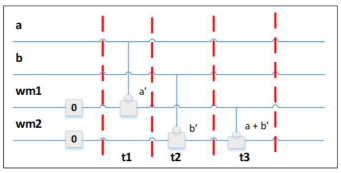

Fig. 7. Simple example showing the ISD notation of an IMPLY gate. In a similar fashion. one can represent all the basic and derived gates

C. Crossbar Array

The fundamental design of a memristor-based crossbar (fig. 6) comprises two sets of parallel conductive (metal) lines. These lines intersect perpendicularly and function as top and bottom electrodes for the memristive material situated between them. To write data to a cell within the crossbar, a specific voltage is applied to the junction, with voltage applied to both lines simultaneously[1]. In this paper, we constructed a synthesiazble model of the crossbar array from [4] and use it to perform an iteration of the SHA-256 loop.

III. ISD NOTATION

To easier understand the command vs time properties of memristive logic, we will use the “Imply Sequence Diagram” (ISD) notation (fig. 7) for the application of analyzing and synthesizing the memristor circuits. In this notation, horizontal lines represent physical memristors, while the box with a circle symbol represents a pulse applied to it. The top side of this symbol is the negated input. The left side is the non-negated input and the value of memristor before the pulse. The right side is the value of the memristor after the pulse. A 0 in the square indicates an additional pulse required to reset the input state of the memristor to 0. The horizontal lines are memristors and the vertical dash lines represent times.

IV. SHA-256

SHA-2is the acronym for the Secure Hash Algorithm-2 [](fig. 8). It is a combination of cryptographic hash functions (mathematical operations used for security purposes). It was designed by NSA (National Security Agency)[13]. The integrity of the data can be determined by computing the hash from the execution of the cryptographic algorithm and comparing it with a known value of the hash function. If the comparison doesn’t match, then we know that the file has been tampered with. The main advantage of the cryptographic hash function is its collision resistance. It is computed with the used of 32-bit words. The application of SHA-2[15] includes security protocols, cryptocurrencies and cryptographic algorithms. SHA-256[11] is one of the hash

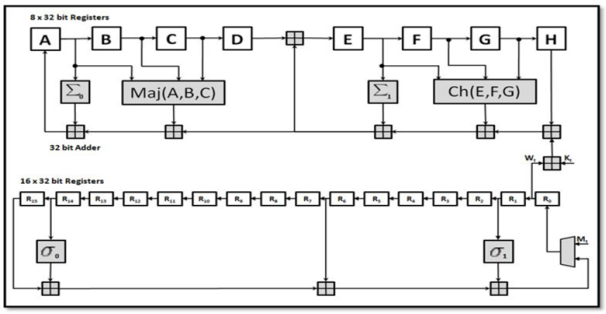

Fig. 8. SHA-256

Main logic blocks of SHA-256 Architecture

1. Modulo Full Adder
2. CH operation: The SHA256 formal specification defines the XOR gate, but it can be replaced with OR: Ch (a, b, c) = a.b + !a.c. (same as mux).
3. aj operation: The SHA256 formal specification defines the XOR gate, but it can be replaced with OR: Maj (a, b, c) = ab + bc + ac.
4. $\Sigma 0$ operation: $\Sigma 0$ (x) = ROTR 2(x) $\oplus$ ROTR 13(x) $\oplus$ ROTR 22(x). ROTRn means Bitwise Rotate Right by n bits. A combinational implementation of rotation operation is using concatenation. $\Sigma 0$ = ({x[1:0], x[31:2]}$\oplus${x[12:0], x[31:13]}$\oplus$ {x[21:0], x[31:22]}).
5. $\Sigma 1$ operation: $\Sigma 1$ (x) = ROTR 6(x) $\oplus$ ROTR 11(x) $\oplus$ ROTR 25(x). $\Sigma 1$ = ({x[5:0], x[31:6]} $\oplus$ {x[10:0], x[31:11]} $\oplus$ {x[24:0], x[31:25]}).
6. $\sigma 0$ operation: $\sigma 0$(x) = ROTR7(x) $\oplus$ ROTR 18(x) $\oplus$ SHR3(x). SHRn is Bitwise Shift Right n bit. A combinational implementation of shift operation is using concatenation and a temporary register with 0. temp= 32'h0, $\sigma 0$= ({x[6:0], x[31:7]} $\oplus$ {x[17:0], x[31:18]} $\oplus$ {temp[2:0], x[31:3]}).
7. $\sigma 1$ operation: $\sigma 1$(x) = ROTR 17(x) $\oplus$ ROTR 19(x) $\oplus$ SHR 10(x). temp= 32'h0, $\sigma 1$ = ({x[16:0], x[31:17]} $\oplus$ {x[18:0], x[31:19]} $\oplus$ {temp[9:0], x[31:10]}).

Fig. 9. The components in SHA-256

functions in the SHA-2 family. The main thing to note is that SHA256 is irreversible in a sense that one can't obtain the real text starting from the cipher text.

A. SHA-256 Symbols

Fig. 9. shows the bit-wise logical implementation of the different components of a loop of SHA-256. To see how they are connected, refer fig. 8

B. Compression Unit

The compression unit (fig. 10) in SHA-256 refers to the algorithmic component responsible for compressing input data

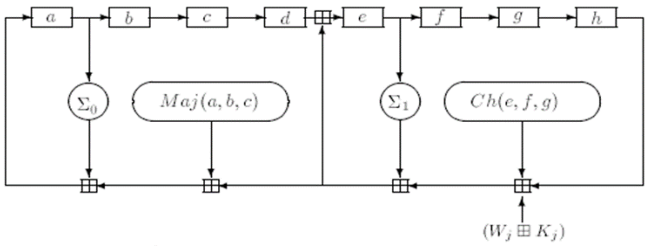

Fig. 10. Block diagram of the compression unit

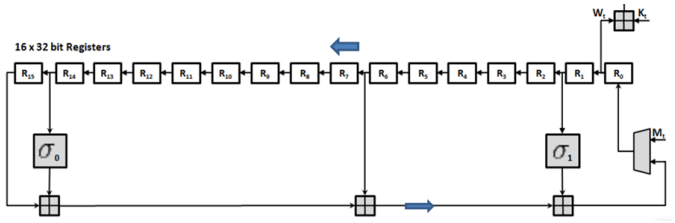

Fig. 11. The message scheduler

blocks into a fixed-size output, which is a critical step in the SHA-256 hashing process. This unit operates on 512-bit blocks of data and uses a series of logical operations, including bitwise operations, modular addition, and Boolean functions, to transform the input data into a 256-bit hash value. The compression unit iteratively processes each block of input data, incorporating it into the hash computation along with the previous hash value. This iterative process ensures that each block of input data contributes to the final hash output in a secure and deterministic manner, making SHA-256 a widely used cryptographic hash function for securing digital information.

C. Message Scheduler

The message scheduler (fig. 11) in SHA-256 is a crucial component responsible for processing the input message before it undergoes compression in the compression function. Its primary role is to expand the input message into a set of 64 32-bit words that serve as input to the compression function. The message scheduler takes the initial message, which may be of any length, and preprocesses it into a fixed-size format suitable for processing by the compression function. It pads the input message to ensure it's a multiple of 512 bits, as required by the SHA-256 algorithm. Additionally, it incorporates the length of the original message to ensure message integrity.

V. SYSTEMVERILOG SIMULATION OF MEMRISTIVE ARCHITECTURE OF SHA-256

The basic structure of the SHA-256 is built using 2 different types of components - The memristor crossbar array and the CPU to control the select signals to the crossbar. The compression unit and the message scheduler are built using said crossbar structure, while the CPU is a basic CMOS circuit. Since the memristor array can't exactly "copy" values supplied to it, it needs the CPU to generate the appropriate select signals to "write" them into the memristors.

A. Crossbar Array on SystemVerilog

The strategy to implement a single row of 3 memristors is shown in Fig[]. For example, if we want to do a simple 2 input NAND gate, we do: IMPLY (2,0) where sel = (11,11,01) and then IMPLY (1,0) where sel = (11,11,00). This is assuming that FF2 was 0 before the start of the operation. Otherwise, we must perform a FALSE operation (basically a clear) for which the signal is not shown in the diagram. For the construction of a 2D array of memristors, I simply replicated a single flip flop, N times horizontally and vertically.

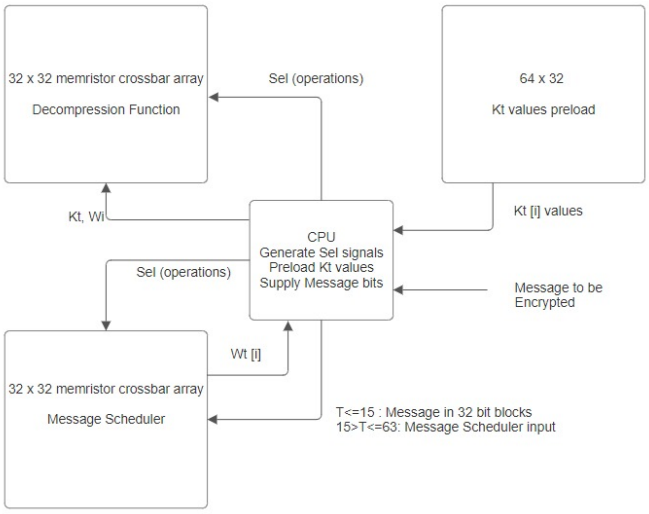

Fig. 12. Block diagram of the different components of the SHA-256

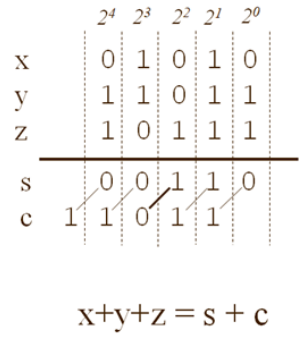

Fig. 13. Example of a carry save adder (CSA)

B. Optimization of SHA-256 components

For our basic SHA256 design, the longest data path, or critical path, is present in the calculation of the variable A for each round in the compression unit. This involves a total of 7 operands, H, Wt, Kt, $\sum 0$, $\sum 1$, Ch(E, F, G) and Maj(A, B, C) with modulo additions. This means that we see the most delay during this process. So, the first optimization we employ is a method of using Carry Save Adders (CSA) (fig. 13) which minimizes the delay caused by the carry propagation in normal adders. CSA's take in three inputs and separate the sum and carry paths as the two outputs. A Carry Save Adder is essentially just a full adder with repurposed inputs and outputs as shown in the Figure below. For a series of adders, instead of the carry bit being propagated through to the next adder, the carry operation is saved to the very end. In our design, carry save adders compute the bit-wise value of each of the operands parallelly and a ripple carry adder is used to calculate the final value afterwards (CPA). (fig. 14)

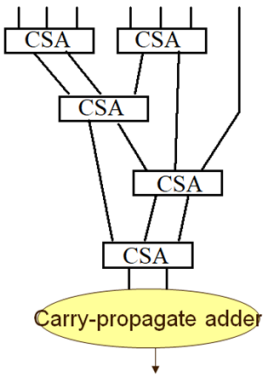

Fig. 14. Using multiple CSAs and a ripple carry adder to generate a 32 bit value

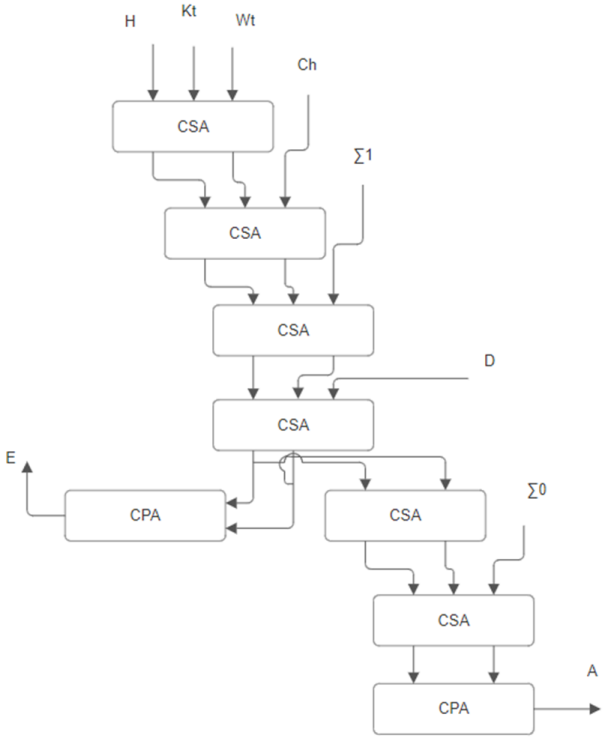

Fig. 15. A very unoptimized instruction flow diagram of the compression unit

C. Compression Unit

The compression unit, is responsible for generating the values A to H. The following list shows the calculation of A through H in the compression unit.

- A= W + K + Ch(E,F,G) + $\sum 1$ + Maj(A,B,C) + $\sum 0$
- B= A
- C= B
- D= C
- E= D + W + K + Ch(E,F,G) + $\sum 1$
- F= E
- G= F
- H= G

Note that here,(fig. 15) The values of the ith loop of the unit are used to calculate the i+1 th values of A and E, but the other values are equal to the ith iteration. For Ch(choose), Maj(majority), $\sum 1$ and $\sum 1$, refer []. For optimizing the Compression unit, we can carry out several of these processes simultaneously. It only takes 5 time units as opposed to the previous 8.

- 1,2,3: CSA(h,wt,kt) CPA(maj,$\sum 0$) CPA(ch,$\sum 1$) Assume these are labelled 1,2 and 3. 1,2 and 3 are executed in parallel, and so on for the remaining steps.
- 4: CSA(3,1)
- 5,6: CSA(4,2) CSA(4,D)
- 7: CPA(A) CPA(E)
- 8: SHIFT (includes the shifting operations as shown in []).

D. Message Scheduler

The Message scheduler in SHA-256 creates a message schedule from the padded message which is later used to iteratively generate a series of hash values. The SHA-256 algorithm uses a message schedule of sixty-four 32-bit words. After pre-processing is completed, each message block (512 bits long) is processed in order. (Where M represents blocks generated from input message). Each message block has 512 bits represented as sixteen 32-bit words. In our design, we use 16 registers to store these values. Look to Figure 3 to find these 16 registers denoted by R0 to R15 from right to left on top. As we see, they are connected to create kind of a cyclic shift register of registers with serial input from M on the bottom right and a serial output wt on top right. The output of the adder on top right goes to the Compression part. For the first 16 cycles, wt = mt (mt represents 32-bit words from message block (M)). The multiplexer, shown at bottom right of Figure 3, is used to either select word directly from the message block (for the first 16 rounds) or to make computations inside the scheduler (for the last 48 rounds). In our particular case, since the time bottleneck occurs in the compression unit, there is no need to optimize any part of the message scheduler. (fig. 16)

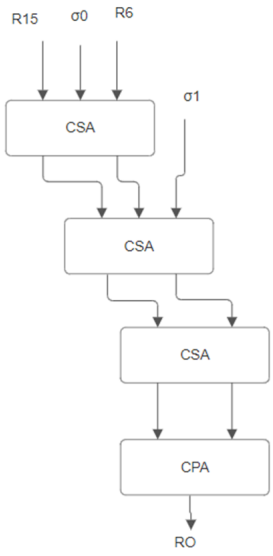

Fig. 16. Instruction flow of the message scheduler

E. CPU and misc.

Theoretically, a real hardware model would consist of a ”brain” that supplies the appropriate select signals and synchronizes the entire circuit. There would also be an array of memory elements that hold the kt and Wt values as well as a mechanism to convert the message into ASCII bits, break up the message into 32 bit parts as well as store the final encrypted message. In our case, for the sake of simplicity, we used a test bench to simulate the working of the CPU with all the pre-loaded values fed to the appropriate systems at the right times.

REFERENCES

[1] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny and U. C. Weiser, ”Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies,” in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 22, no. 10, pp. 2054-2066, Oct. 2014, doi: 10.1109/TVLSI.2013.2282132.

[2] L. Chua, ”Memristor-The missing circuit element,” in IEEE Transactions on Circuit Theory, vol. 18, no. 5, pp. 507-519, September 1971, doi: 10.1109/TCT.1971.1083337.

[3] L. O. Chua and Sung Mo Kang, ”Memristive devices and systems,” in Proceedings of the IEEE, vol. 64, no. 2, pp. 209-223, Feb. 1976, doi: 10.1109/PROC.1976.10092.

[4] K. C. Rahman, M. R. Khan and M. A. Perkowski, ”Memristor based 8-bit iterative full adder with space-time notation and sneak-path protection,” 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS), Boston, MA, USA, 2017, pp. 695-698, doi: 10.1109/MWSCAS.2017.8053018.

[5] Borghetti J, Snider GS, Kuekes PJ, Yang JJ, Stewart DR, Williams RS. ’Memristive’ switches enable ’stateful’ logic operations via material implication. Nature. 2010 Apr 8;464(7290):873-6. doi: 10.1038/nature08940. PMID: 20376145.

[6] Siemon A, Drabinski R, Schultis MJ, Hu X, Linn E, Heittmann A, Waser R, Querlioz D, Menzel S, Friedman JS. Stateful Three-Input Logic with Memristive Switches. Sci Rep. 2019 Oct 10;9(1):14618. doi: 10.1038/s41598-019-51039-6. PMID: 31602003; PMCID: PMC6787102.

[7] M.D. Pickett, D.B. Strukov, J.L. Borghetti, J.J. Yang, G.S. Snider, D.R. Stewart, and R.S. Williams, “Switching dynamics in titanium dioxide memristive devices,” J. Appl. Phys., vol. 106, no. 7, pp. 1–6, Oct. 2009.

[8] Z. Biolek, D. Biolek, and V. Biolkova, “SPICE model of memristor with nonlinear dopant drift,” Radioengineering, vol. 18, no. 2, pp. 210–214, Jun. 2009.

[9] J. J. Yang, M. D. Pickett, X. Li, D. A. A. Ohlberg, D. R. Stewart, and R. S. Williams, “Memristive switching mechanism for metal/oxide/metal nanodevices,” Nature Nanotechnol., vol. 3, pp. 429–433, Jul. 2008.

[10] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, “Memristive switches enable ‘stateful’ logic operations via material implication,” Nature, vol. 464, pp. 873–876, Apr. 2010. IMPLY

[11] C. Jeong, et al., ”Implementation of Efficient SHA-256 Hash Algorithm for Secure Vehicle Communication using FPGA”, 2014 International SoC Design Conference (ISOCC), South Korea, 3-6 Nov. 2014. (SHA)

[12] M. Kammoun, M. Elleuchi, M. Abid and M. S. BenSaleh, ”FPGA-based implementation of the SHA-256 hash algorithm,” 2020 IEEE International Conference on Design and Test of Integrated Micro and Nano-Systems (DTS), Hammamet, Tunisia, 2020, pp. 1-6, doi: 10.1109/DTS48731.2020.9196134.