iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 11 **Issue:** XII **Month of publication:** December 2023

DOI: https://doi.org/10.22214/ijraset.2023.57521

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue XII Dec 2023- Available at www.ijraset.com

SHA-256 Hash Function on Intel DE10 Lite FPGA

Michael DiNardi[1], Damu Radhakrishnan[2]
[1]*Ericsson, Austin, Texas, USA-78703*
[2]*Division of Engineering Programs, State University of New York, New Paltz, USA-12561*

Abstract: ***The SHA-256 hash function is a standardized and trusted algorithm that takes a set of data and produces a unique, deterministic, and irreversible representation called a hash or digest. A component of other protocols, SHA-256 protects password storage, secures and verifies Bitcoin transactions, and authenticates internet communication. We did a thorough analysis of the hash function and a preexisting Verilog implementation at the algorithmic, architectural, and circuit levels to identify and address the bottlenecks. We propose a new SHA-256 hardware architecture that utilizes binary tree structured adder trees to speed up hash computation. The proposed design targets Intel DE10-Lite FPGA and achieves 23% increase in computation speed. In applications, this can offer faster online communication or a more secure Bitcoin network.***
Keywords: Hash function, Verilog, Field Programmable Gate Array, Binary Tree, Modular addition.

I. INTRODUCTION

Hash functions are extremely useful and are used in almost all information security applications [1]. It is a mathematical function that converts a numerical input value into another compressed numerical value. Secure Hash Algorithms, also known as SHA, are a family of cryptographic functions designed to keep data secured. It works by transforming the data using a hash function: an algorithm that consists of bitwise operations, modular additions, and compression functions. The SHA-256 algorithm is a secure and trusted industry standard used in e-transactions, Bitcoin, and certain United States governmental applications to protect information from adversaries. The Secure Hash Signature Standard (SHS) was proposed by the US National Institute of Standards and Technology (NIST) in 2002 [2]. The standard describes four secure hash algorithms (SHA) and the version which outputs a 256-bit message digest is referred to as SHA-256. Technology leaders and public-sector agencies widely use and safely rely on SHA-256 due to the algorithm not having any known vulnerabilities that make it insecure and not being "broken" unlike other popular algorithms.

Hash functions take a message and produce a digest, a fixed-length representation of the message [1]-[3]. Key properties of hash functions are that the same message always yields the same digest, no two messages share the same digest, and a message cannot be decrypted from a given digest. Hash functions are used to identify data without revealing it, to identify whether a piece of data changed, or to confirm whether two pieces of data are the same.

Hardware implementations of hash functions are advantageous over software implementations for better security and faster speed. Field Programmable Gate Array (FPGA) devices provide an excellent technology for the implementation of general purpose cryptographic algorithms [4]–[12]. They are used as coprocessors for microprocessor based systems or in high performance embedded applications as they are more physically secure by nature and are physically separated from the main processor. They can also perform computation more efficiently due to specialized logic. Moreover, FPGAs are well-suited for implementing hash functions as they are flexible and easily upgradable.

This research proposes a new SHA-256 hardware architecture targeting Intel DE10-Lite FPGA that utilizes binary tree structured adder trees to reduce computation time. The study modifies a preexisting design in Verilog [5]. Quartus Prime tools are utilized to examine the longest delay and to calculate the maximum operating speed.

II. THE SHA-256 ALGORITHM

SHA-2 is a set of hash functions – SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 - standardized by NIST [2], [13]. They are one-way algorithms that process a set of binary data, called a message, to produce a condensed representation called a hash, message digest or simply digest. For each algorithm, no two messages are mapped to the same digest - every digest is unique to its original message. The numerous algorithms available have different security strengths, dependent on the digest size. The digest size ranges from 224 to 512 bits, as denoted by the name of the algorithm. In addition, the algorithms differ by the message size, the word size, and the constants.

©IJRASET: All Rights are Reserved

The SHA-256 algorithm takes a message of length less than 2^{64} bits and produces a message digest of 256 bits. It has a security of 128 bits, which means that a birthday attack can produce a collision in $O(2^{128})$ time [14]. All operations in SHA-256 are done on 32-bit words, and all additions are done modulo 232. The algorithm uses six logical operations as:

$Ch(x, y, z) = (x \wedge y) \oplus (\sim x \wedge z)$

$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$

$\Sigma^0(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$

$\Sigma^1(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$

$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$

$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

where $\wedge$, $\sim$ and $\oplus$ are the bitwise AND, NOT and XOR operations; and $ROTR^m(x)$ denotes a rotate right function of x m times and $SHR^m(x)$ denotes a shift right function of x m times.

The following subsections describe the process for SHA-256, though all algorithms follow the same two-stage process: preprocessing and hash computation. Information on the other algorithms can be found in the official NIST standard [13].

A. *Preprocessing*

1) Message Padding: Padding is performed before hash computation to ensure the message is a multiple of 512 bits. Suppose a message M is of length l bits. First, append a "1" to the end of the message, followed by k "0" bits, where k satisfies the equation $l + 1 + k = 448 \bmod 512$. Then, the binary representation of l as a 64-bit number is appended so that the length of the padded message is a multiple of 512 bits.
2) Message Parsing: The padded message is then parsed into N 512-bit blocks: $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. A block $M^{(i)}$ is represented as 16 32-bit words: $M0^{(i)}, M1^{(i)}, \ldots, M15^{(i)}$.

B. *Hash Computation*

The 256-bit hash output is denoted as H, and is divided into eight 32-bit words $H_0, H_1, \ldots, H_7$. The hash computation is performed in two stages.

1) Message Expansion: Message expansion, also known as message schedule or block decomposition, expands the message block from 512 bits to 2048 bits, specifically from 16 32-bit words to 64 32-bit words, denoted as Wt. The 64 words are constructed as follows:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases} \quad (1)$$

such that

$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$

$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

2) Message Compression: Message compression is an iterative process that condenses the expanded block down to eight 32-bit words ($H_0, H_1, \ldots, H_7$), the digest size of 256 bits. First, $H_0^{(0)}$ through $H_1^{(0)}$ are initialized to the first 32-bits of the fractional parts of the square roots of the first eight prime numbers. Next, eight working variables are set to the $(i-1)^{st}$ hash value:

$a = H_0^{(i-1)}$ $b = H_1^{(i-1)}$ $c = H_2^{(i-1)}$ $d = H_3^{(i-1)}$ $e = H_4^{(i-1)}$

$f = H_5^{(i-1)}$ $g = H_6^{(i-1)}$ $h = H_7^{(i-1)}$

For t = 0 to 63:

$a = T_1 + T_2$ $b = a$ $c = b$ $d = c$ $e = d + T_1$

$f = e$ $g = f$ $h = g$

where $T_1 = h + \Sigma_1(e) + Ch(e,f,g) + K_t + W_t$, and $T_2 = \Sigma_0(a) + Maj(a,b,c)$

such that

$\Sigma_0(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$

$\Sigma_1(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$

$Ch(x,y,z) = (xy) \oplus (\sim xz)$

$Maj(x,y,z) = (xy) \oplus (xz) \oplus (yz)$

K_t are the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue XII Dec 2023- Available at www.ijraset.com

After the 64 rounds of processing for H_i, the new value of $H_j^{(t)}$ is computed:

$$H_0^{(t)} = H_0^{(i-1)} + a \qquad H_1^{(t)} = H_1^{(i-1)} + b \qquad H_2^{(t)} = H_2^{(i-1)} + c \qquad H_3^{(t)} = H_3^{(i-1)} + d$$
$$H_4^{(t)} = H_4^{(i-1)} + e \qquad H_5^{(t)} = H_5^{(i-1)} + f \qquad H_6^{(t)} = H_6^{(i-1)} + g \qquad H_7^{(t)} = H_7^{(i-1)} + h$$

A direct translation of the SHA-256 algorithm into hardware is shown in Figure 1, consisting of the message compressor (left) and the message expander (right). A straightforward implementation leads to a scheme whose critical path is rather long, allowing a clock rate not sufficiently high enough for many applications. In such cases, it becomes necessary to implement the corresponding algorithms in hardware so that the corresponding delays are not noticeable.

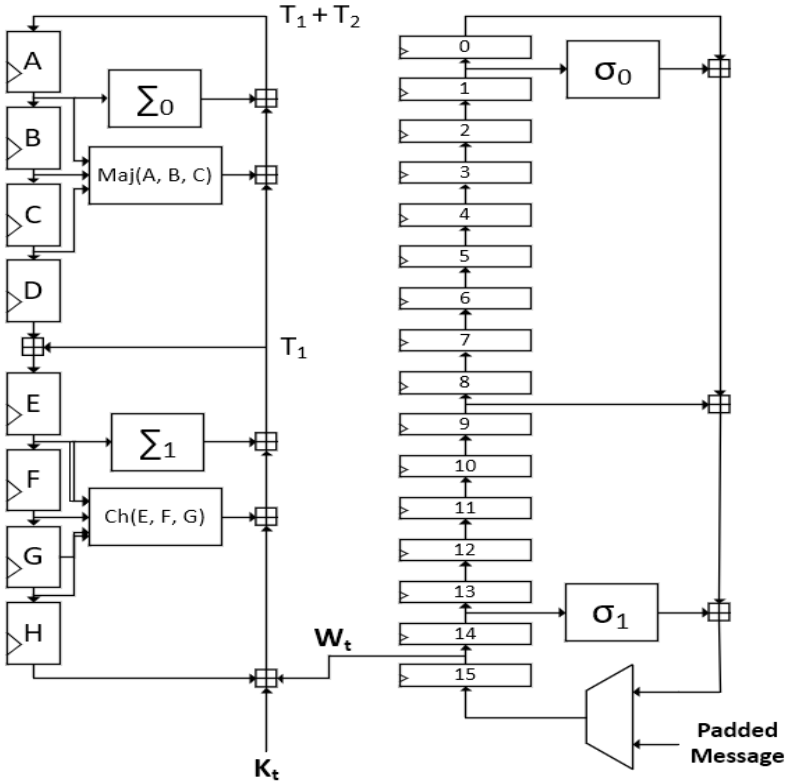


Fig. 1. Block diagram of the SHA-256 algorithm comprised of message compressor (left), and message expander (right)

Custom hardware implementations of the hash functions SHA-256, 384 and 512 are available, obtaining high clock rates through a reduction of the critical path length, both in the Expander and in the Compressor of the hash scheme [15]. This is obtained by applying suitable transformations to the simplest scheme shown in Figure 1. The transformations are called delay balancing and quasi-pipelining. Their VHDL implementation using Synopsys Design Compiler on a Sun workstation resulted in a clock rate well exceeding 1 GHz using 0.13µm technology.

III. SHA-256 IMPLEMENTATION ON DE-10 LITE FPGA

Our research goal here was to speedup the simple scheme of SHA-256 implementation shown in Figure 1 and implement it on a DE10-Lite FPGA using Verilog coding. In this regard, first a Verilog implementation of SHA-256 by GitHub user Secworks was first analyzed for its structure and functionality, while simultaneously comparing it to the SHA-256 algorithm (Secworks). Individual testbenches were provided for each module to verify its correctness. The modules were examined in RTL viewer in Quartus Prime, and the critical paths were analysed.

A. Sha256_k_constants.v

Sha256_k_constants.v module holds the 64 constants, the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. It functions as read-only memory, comprised of a 6-bit address line for 64 rounds and a 32-bit output.

©IJRASET: All Rights are Reserved

B. Sha256_w_mem.v

Sha256_w_mem.v is the message expander module, expanding the 16 words of the padded block to the 64 words. This module executes Equation 1. For rounds 0 to 15, each of the 16 words of the block are directly output. For rounds 16 to 64, new words are generated using logical and addition operations.

C. Sha256_core.v

Sha256_core.v is the main core, comprised of the constants module, the message expander module, and the message compressor logic. It iteratively computes the values for registers *A* through *H*.

D. Sha256.v

Sha256.v is a top-level wrapper with a 32-bit interface for simple integration within other systems.

IV. PROPOSED ARCHITECTURE

Figure 2 shows the data paths for registers *A* and *E* in Figure 1, according to the RTL viewer. A total of 10 operand additions are needed for register *A* and 9 operands for register *E*. This results in a critical path delay of six 32-bit adder stages for both *A* and *E*. To minimize the critical path delay, we rearranged the operands and added them in a binary tree like fashion. This results in logarithmic reduction in delay, thereby addition time can be reduced to 4 adder stages. The modified data path is shown in Figure 3. In this manner the critical path delay is reduced from six adder stages to four adder stages.

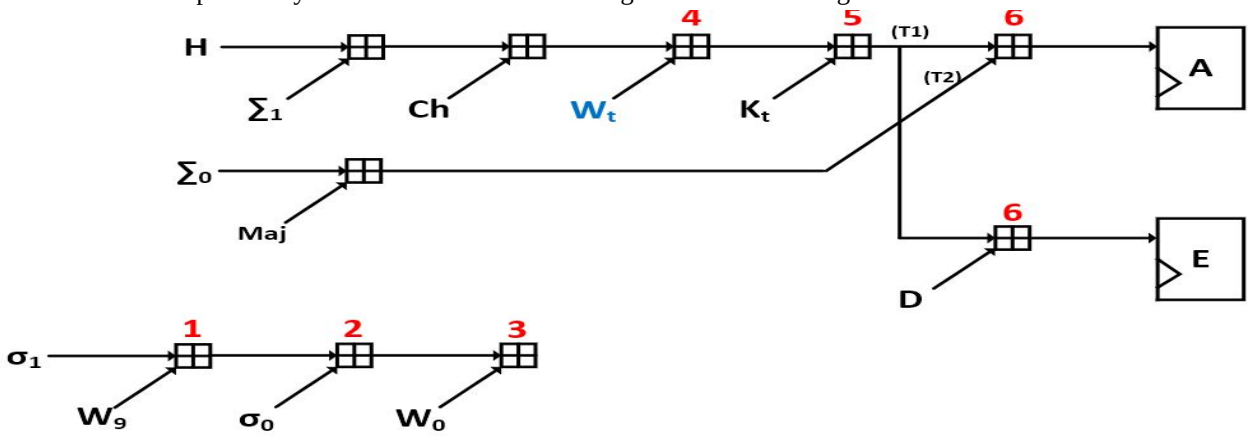


Fig. 2. Data Paths of A, E, and W_t from Figure 1.

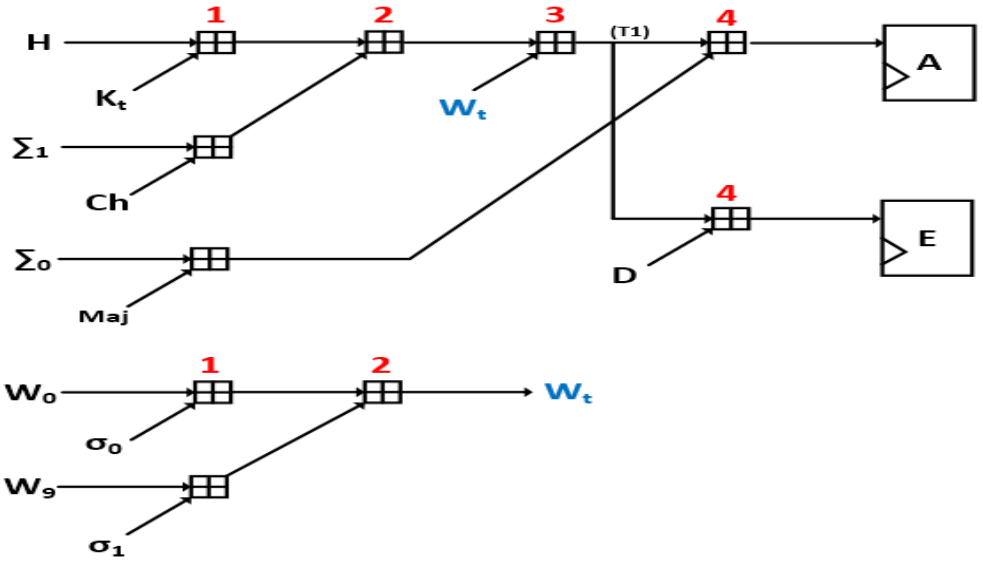


Fig. 3. Optimized Data paths of registers A, E, and W_t

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue XII Dec 2023- Available at www.ijraset.com

V. MODIFICATIONS IN VERILOG CODE

The modifications according to our proposed design were introduced in Verilog. This was done in three stages. First, an adder tree was implemented for W_t. Next, an adder tree was implemented for *a_reg*. Lastly, an adder tree was implemented for *e_reg*. Simulations were then carried out.

Figure 4 shows the excerpts of the original unmodified sha256_core.v, involving T_1 computation logic, and state updates for registers *A* through *H*. The code uses blocking statements, so they are executed sequentially. First, Σ_1 is computed, then Ch, and finally T_1 is computed. T_1 involves a five-operand addition of *h*, Σ_1, *Ch*, W_i, and K_i. Lastly, *a* is computed by adding T_1 and T_2. This is according to the data path shown in Figure 2. Registers *A* through *H* are then updated for 64 rounds.

```
always @*
  begin: t1_logic
   reg [31:0] sum1;
   reg [31:0] ch;

   sum1 = {e_reg[5:0], e_reg[31:6]}^
         {e_reg[10:0], e_reg[31:11]}^
         {e_reg[24:0], e_reg[31:25]};
   ch = (e_reg & f_reg)^((~e_reg) & g_reg);
   t1 = h_reg+sum1+ch+w_data+k_data;
  end // t1_logic
  .
  .
  .
   if (state_update)
     begin
       a_new  = t1 + t2;
       b_new  = a_reg;
       c_new  = b_reg;
       d_new  = c_reg;
       e_new  = d_reg + t1;
       f_new  = e_reg;
       g_new  = f_reg;
       h_new  = g_reg;
       a_h_we = 1;
     end
```

Fig. 4. Unmodified T_1 logic and state update.

Figure 5 shows the first modification that was made, which implemented the addition using a binary tree of adder stages for register *a* and W_t as shown in Figure 3. Once again blocking statements are used, so they are executed sequentially. First, Σ_1 is computed. Then, *h* and K_t are added together in *adder1*. Next, Σ_1 and *Ch* are added together as *adder2*, Next, *adder1* and *adder2* are added together as *adder3*. W_t and *adder3* are then added together, resulting in *T1*. Lastly, *T1* and *T2* are added together, resulting in *a*.

```
reg [31:0] sum1;
 reg [31:0] adder1;
 reg [31:0] adder2;
 reg [31:0] adder3;
 always @*
  begin : t1_logic
         sum1 = ({e_reg[5:0], e_reg[31:6]}^
         {e_reg[10:0],e_reg[31:11]}^{e_reg[24:0], e_reg[31:25]});
   adder1 = h_reg + k_data;    //h + kt
         adder2 = sum1 + ((e_reg & f_reg)^((~e_reg) & g_reg)); //sum1 + ch
```

```
adder3 = adder1 + adder2; //h + kt + ch + sum1
      t1 = w_data + adder3;
end // t1_logic
      .
      .
      if (state_update)
 begin
   a_new  = t1 + t2;
   b_new  = a_reg;
   c_new  = b_reg;
   d_new  = c_reg;
   e_new  = d_reg + t1;
   f_new  = e_reg;
   g_new  = f_reg;
   h_new  = g_reg;
   a_h_we = 1;
 end
```

Fig. 5. Modified T_1 logic using adder tree and state update.

The W_t modification was made on module sha256_w_mem.v. The logic is part of the message expander, that computes W_t, from rounds 16 through 64, shown in Equation 1. W_t is computed by adding four operands, σ_1, *w9*, σ_0, and *w0*. Figure 6 shows an excerpt of the unmodified module.

```
w_0  = w_mem[0];
  w_1  = w_mem[1];
  w_9  = w_mem[9];
  w_14 = w_mem[14];
  d0 = {w_1[6:0], w_1[31:7]}^
     {w_1[17:0], w_1[31:18]}^
     {3'b0, w_1[31:3]};
  d1 = {w_14[16:0], w_14[31:17]}^
     {w_14[18:0], w_14[31:19]}^
     {10'b0, w_14[31:10]};
  w_new = d1 + w_9 + d0 + w_0;
```

Fig. 6. Unmodified W_t logic.

Figure 7 shows the modified logic for computing W_t from rounds 16 through 64 by utilizing an adder tree. First, *w0* and σ_0 are added together as *adder1*. Then, *w9* and σ_1 are added together as *adder2*. Lastly, *adder1* and *adder2* are added together to result in W_t.

```
w_1  = w_mem[1];
  w_14 = w_mem[14];
  d0 = {w_1[6:0], w_1[31:7]}^
     {w_1[17:0], w_1[31:18]}^
     {3'b0, w_1[31:3]};
  d1 = {w_14[16:0], w_14[31:17]}^
     {w_14[18:0], w_14[31:19]}^
     {10'b0, w_14[31:10]};
  adder1 = w_mem[0] + d0;
  adder2 = w_mem[9] + d1;
  w_new = adder1 + adder2;
```

Fig. 7. Modified W_t logic using adder tree.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue XII Dec 2023- Available at www.ijraset.com

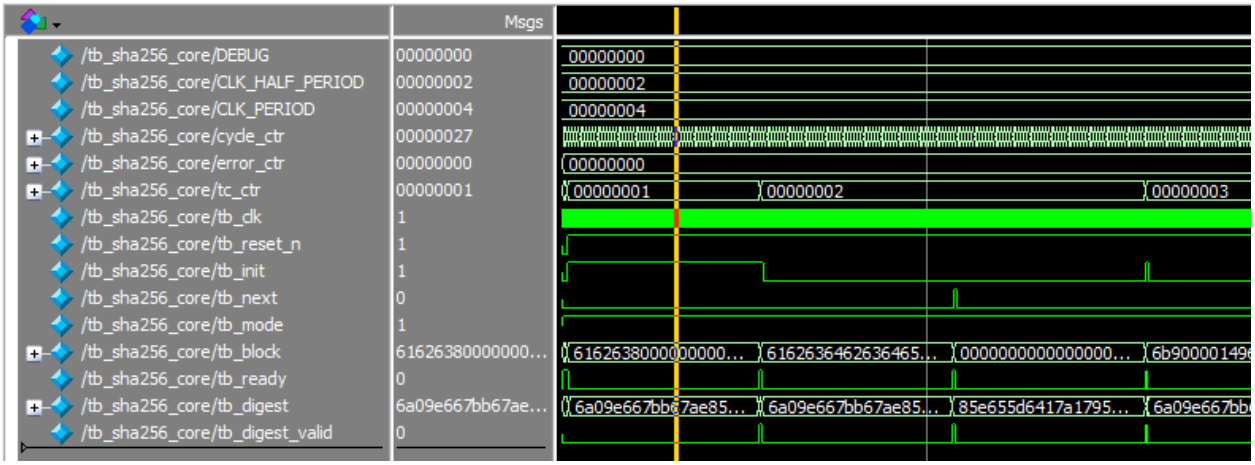
VI. VERIFICATION

The Verilog implementation of SHA-256 algorithm was verified by running the testbench simulation, tb_sha256_core.v, that was provided in the open-source code [16]. Modelsim 21.1 was used to run the simulation. Figure 8 shows the three test case inputs and the corresponding results: test cases (a), testbench waveforms (b), and testbench output (c). Three test cases were performed. Test case 1 inputs one block, the string "*abc*", test case 2 inputs two blocks, the string "*abcdbcdecdefdefgefghfghighijhijkijkljk lmklmnlmnomnopnopq*", and test case 3 inputs nine blocks. The output hash is *tb_digest*, and the completed flag is *tb_digest_ready*. Simulating the testbench for each modification verified that the design was functioning properly, and that implementing the adder trees was functionally possible.

```
begin : sha256_core_test
    // TC1: Single block message: "abc".
    tc1 =
512'h6162638000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000018;
    res1 = 256'hBA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD;
    single_block_test(1, tc1, res1);
    // TC2: Double block message.
    // "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
    tc2_1 =
512'h6162636462636465636465666465666765666768666768696768696A68696A6B696A6B6C6A6B6C6D6B6C6D6E6C6D6E6F6D6E6F706E6F70718000000000000000;
    res2_1 = 256'h85E655D6417A17953363376A624CDE5C76E09589CAC5F811CC4B32C1F20E533A;
    tc2_2 =
512'h00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001C0;
    res2_2 = 256'h248D6A61D20638B8E5C026930C3E6039A33CE45964FF2167F6ECEDD419DB06C1;
    double_block_test(2, tc2_1, res2_1, tc2_2, res2_2);
  end
begin : issue_test;
    block0 =
512'h6b900001_496e2074_68652061_72656120_6f662049_6f542028_496e7465_726e6574_206f6620_5468696e_6773292c_206d6f72_6520616e_64206d6f_7265626f_6f6d2c20;
    block1 =
512'h69742068_61732062_65656e20_6120756e_69766572_73616c20_636f6e73_656e7375_73207468_61742064_61746120_69732074_69732061_206e6577_20746563_686e6f6c;
    block2 =
512'h6f677920_74686174_20696e74_65677261_74657320_64656365_6e747261_6c697a61_74696f6e_2c496e20_74686520_61726561_206f6620_496f5420_28496e74_65726e65;
    block3 =
512'h74206f66_20546869_6e677329_2c206d6f_72652061_6e64206d_6f726562_6f6f6d2c_20697420_68617320_6265656e_20612075_6e697665_7273616c_20636f6e_73656e73;
    block4 =
512'h75732074_68617420_64617461_20697320_74697320_61206e65_77207465_63686e6f_6c6f6779_20746861_7420696e_74656772_61746573_20646563_656e7472_616c697a;
    block5 =
512'h6174696f_6e2c496e_20746865_20617265_61206f66_20496f54_2028496e_7465726e_6574206f_66205468_696e6773_292c206d_6f726520_616e6420_6d6f7265_626f6f6d;
    block6 =
512'h2c206974_20686173_20626565_6e206120_756e6976_65727361_6c20636f_6e73656e_73757320_74686174_20646174_612069
```

```
    block7 =
512'h6f6c6f67_79207468_61742069_6e746567_72617465_73206465_63656e74_72616c69_7a617469_6f6e2c49_6e207468_6520
6172_6561206f_6620496f_54202849_6e746572;
    block8 =
512'h6e657420_6f662054_68696e67_73292c20_6d6f7265_20616e64_206d6f72_65800000_00000000_00000000_00000000_0000
0000_00000000_00000000_00000000_000010e8;
    expected = 256'h7758a30bbdfc9cd92b284b05e9be9ca3d269d3d149e7e82ab4a9ed5e81fbcf9d;
    $display("Running test for 9 block issue.");
```

(a)



(b)

```
VSIM 4> run -all
#     -- Testbench for sha256 core started --
# *** Toggle reset.
# *** TC 1 single block test case started.
# *** TC 1 successful.
#
# *** TC 2 double block test case started.
# *** TC 2 first block started.
# *** TC 2 first block done.
# *** TC 2 second block started.
# *** TC 2 second block done.
# *** TC 2 first block successful
#
# *** TC 2 second block successful
#
# Running test for 9 block issue.
# Digest ok.
# *** All  3 test cases completed successfully
# *** Simulation done.
```

(c)

Fig. 8. tb_sha256_core test cases (a), waveforms (b), and output (c).

VII. RESULTS

The original design and the four modifications were compared for their maximum operating frequency, area, and hashrate. The *hashrate* is the reciprocal of the hash time. If a new hash is generated every N cycles (in our case every 64 cycles),

$$Hashrate = \frac{1}{CLK\ period * N} = \frac{CLK\ freq}{64}$$

Table 1 shows the comparison. A0 is the original design, B1 implements an adder tree on register *a*, B2 implements an adder tree on registers *a* and *e*, B3 implements an adder tree on registers *a*, *e*, and W_t, and C1 switches Wt and Σ_1. The original design, A0, operates at a maximum clock rate of 67.18 MHz, with a hash rate of 1049.69 KH/s. B1, B2, B3, and C1 are 7.67%, 8.89%, 10.00%, and 23.00% faster, respectively. A0 uses 3572 lookup tables (LEs), B1, B2, B3 uses 4.26%, 4.81%, and 5.21% more LEs, and C1 uses 0.7% more LEs. A0, B1, B2, and B3 have practically the same hash rate/area of 300 H/s/LE. C1 has the best performance per area at 359 H/s/LE.

TABLE I
SHA-256 MODIFICATION COMPARISON

Ver.	Modification	Fmax (Slow 1200mV 85C) (MHz)	Hashrate (KH/s)	Area (LEs)	Hashrate/Area (H/s/LE)
A0	Original	67.18	1049.69	3572	294
B1	Adder tree - a_reg	72.33	1130.16	3724	303
B2	Adder tree: a_reg, e_reg	73.15	1142.97	3744	305
B3	Adder tree: a_reg, e_reg, Wt	73.87	1154.22	3758	307
C1	Adder tree: a_reg, e_reg, Wt. Swaped Wt and Σ_1	82.69	1292.03	3600	359

VIII. CONCLUSIONS

Hash functions are naturally well-suited to be implemented in hardware as they involve the logical operations, the manipulation of bits, and iterative rounds. Furthermore, hardware acceleration is advantageous over software since specialized logic serves one purpose or operation; where as software is executed on a general-purpose processor.

SHA-256 hardware implementation was studied at the algorithmic, architectural, and circuit levels to find areas for improvement specifically aimed at speeding up the computation time. As the critical path was identified as the addition of seven operands in sequence, it was decided to parallelize these additions using adder trees. This resulted in a design with four adder stages in sequence, resulting in a hashrate 23% faster than the original design, while achieving the best performance per area of 359 H/s/LE.

We also noticed many avenues for further research. One is to compare the performance of this SHA-256 core with software counterparts. Ideally, a software program needs to be written for a soft-core processor for the same FPGA, the DE10-Lite. Additionally, the SHA core could be physically tested and implemented in a system within the DE10-Lite and other FPGAs such as the DE-2.

IX. ACKNOWLEDGMENT

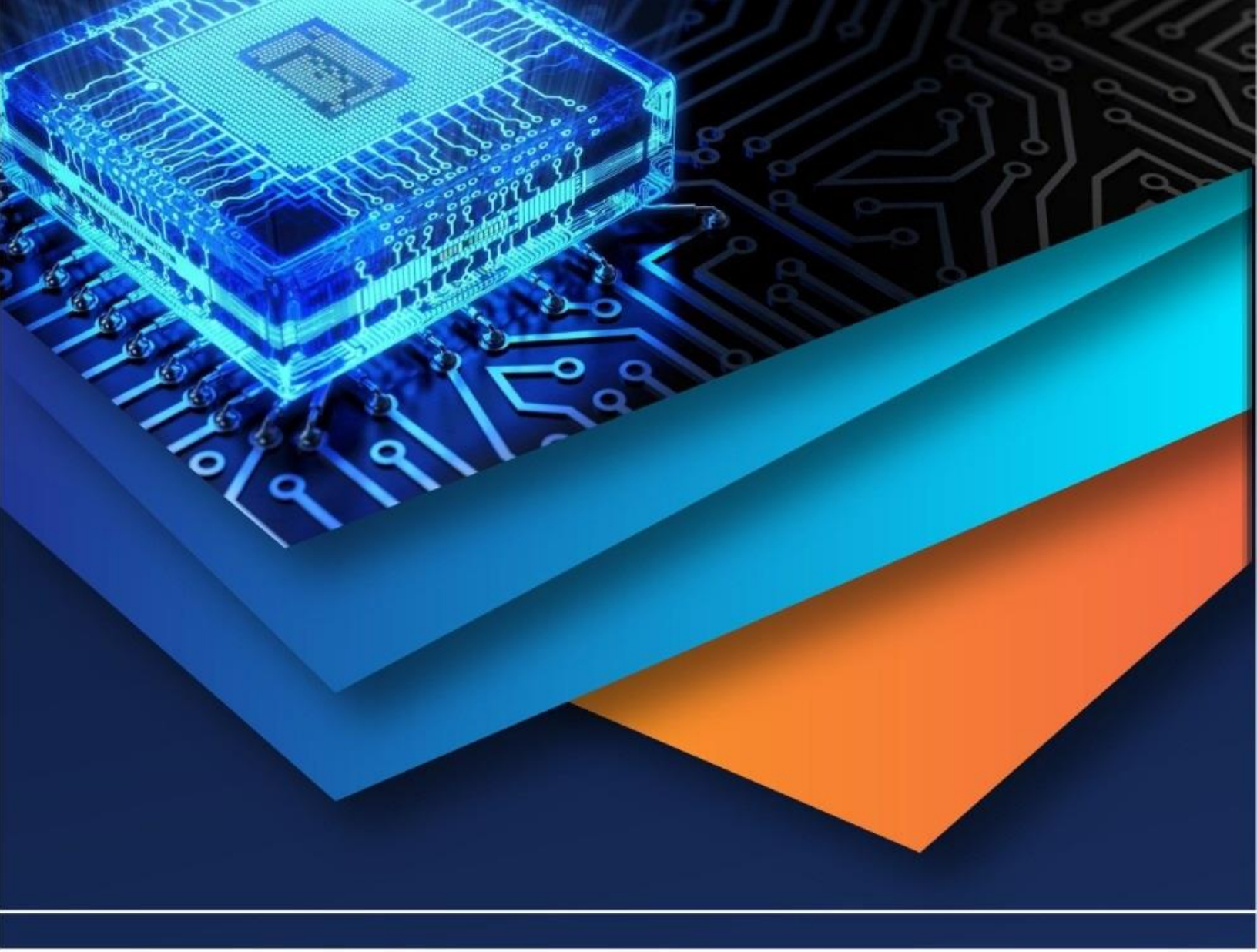
The authors acknowledge the support provided by the State University of New York, New Paltz, USA in completing this study.

REFERENCES

[1] B. Schneier, Applied Cryptography: Protocols, Algorithms and Source Code in C, Wiley, 1996.
[2] Secure Hash Standard. Federal Information Processing Standards Publication 180-2. NIST Maryland, USA., 2002.
[3] J. Docherty and A. Koelmans, “Hardware implementation of SHA-1 and SHA-2 hash functions,” Microelectronics System Design Research Group, EECE, Newcastle University, UK, Tech. Rep., 2011.
[4] K. K. Ting, S. C. L. Yuen, K. H. Lee, and P. H. W. Leong, “An FPGA based SHA-256 processor,” in Proc. Field-programmable Logic and Applications: Reconfigurable Computing is Going Mainstram Conference, 2000, pp. 577-585.
[5] I. Ahmad, and A.S. Das, “Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs,” Computers and Electrical engineering, vol. 31(6), pp. 345-360, 2005.
[6] I. Algredo-Badillo, C. Feregrino-Uribe, R. Cumplido, and M. Morales-Sandoval, “FPGA-based implementation alternatives for the inner loop of the secure hash algorithm SHA-256,” Microprocessors and Microsystems, vol. 37, pp. 750-757, 2013.
[7] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, Improving SHA-2 hardware implementations, in Proc. of 8th International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2006, 2006, Yokohama, Japan, pp. 298-310.
[8] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W.P. Marnane, “Optimization of the SHA-2 family of hash functions on FPGAs,” in IEEE Computer Society Annual Symp. On Emerging VLSI Technologies and Architectures, 2006, pp. 317-322.
[9] H. Mestiri, F. Kahri, B. Bouallegue, and M. Machhout, “Efficient FPGA hardware implementation of secure hash function SHA-2,” International Journal of Computer Network and Information Security, vol. 1, pp. 9-15, 2015.

[10] M. Togan, A. Floarea, and G. Budariu, Design and implementation of cryptographic modules on FPGA, in Proc. Applied Mathematics and Informatics, 2010, pp. 149-154.
[11] A. H. Gad, S. E. E. Abdalazeem, O. A. Abdelmegid, and H. Mostafa, "Low power and area SHA-256 hardware accelerator on Virtex-7 FPGA," in Proc. 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES), 2020, pp. 181-185.
[12] C. Jeong and Y. Kim, Implementation of efficient SHA-256 hash algorithm for secure vehicle communication using FPGA, in Proc. International SOC Design Conference (ISOCC), 2014, pp. 224-225.
[13] Secure Hash Standard (SHS), FIPS PUB 180-4, Maryland, USA, 2015.
[14] C. Paar and J. Pelzl, Understanding Cryptography, New York: Springer, 2010.
[15] L. Dadda, M. Macchetti, and J. Owen, "An ASIC design for a high-speed implementation of the hash function SHA-256 (384,512)," in Proc. ACM Great Lakes Symposium on VLSI, 2004, pp. 421-425.
[16] (2001) Secworks, sha256. Available: https://github.com/secworks/sha256

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)