ECE 111: Final Project

Final Project Part-1: SHA-256

1. SHA-256 Explanation

SHA-256 is a “secure hash algorithm” developed by the National Security Agency in 2001. It is a cryptographic method that takes input data of any size and converts it into a fixed 256-bit output, hence the name. For SHA-256 to work properly, it must satisfy these main properties: compression, the avalanche effect, determinism, preimage resistance, collision resistance, and efficiency.

Firstly, SHA-256 must be able to compress data of any size into a 256-bit output value or digest, ensuring that the output must always be 256 bits regardless of the input. This includes smaller sized inputs which must always result in a 256-bit output.

Secondly, SHA-256 must be able to satisfy the avalanche effect, meaning that a minute change in the input data must result in a drastic change in the output. This property prevents malicious attacks by making brute-force methods mute.

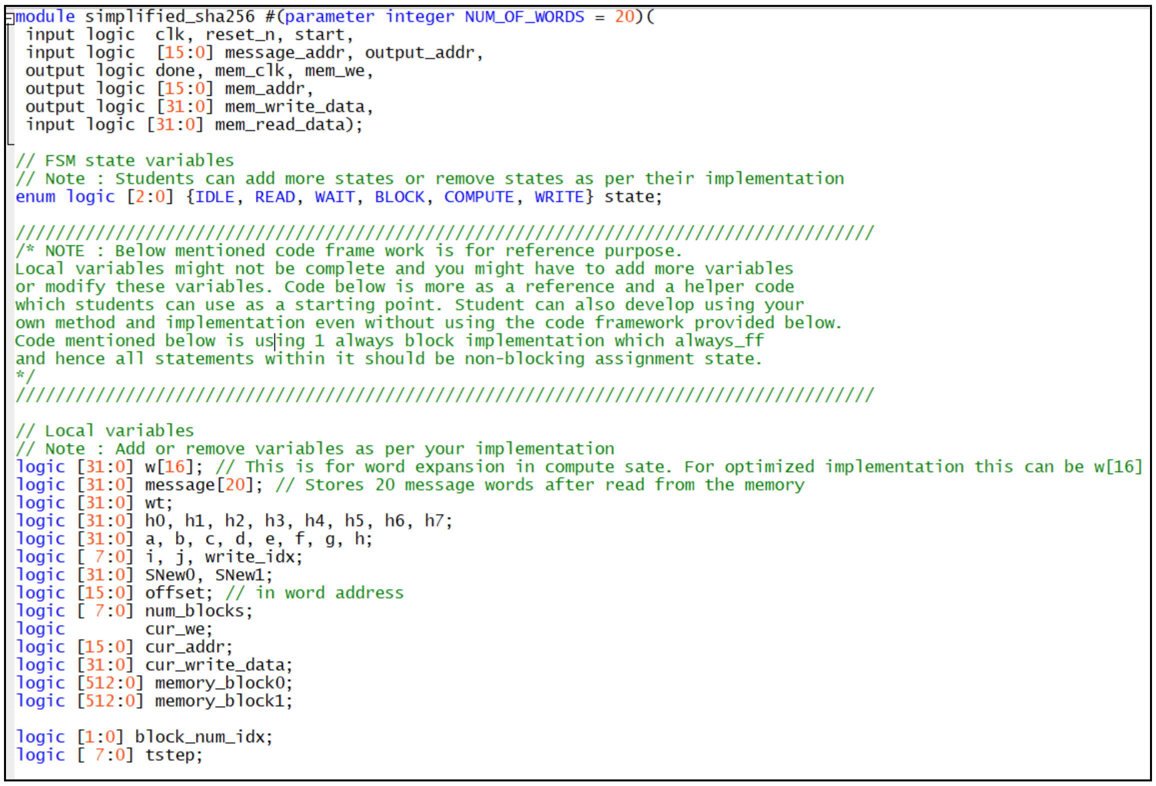
Thirdly, SHA-256 must be deterministic meaning that the same algorithm executed on any machine must result in the same output, given the same output.

Fourthly, SHA-256 must be preimage resistant meaning that it is a one-way function where no such inverse is possible. I.e. there is no practical way to retrieve the input given the output data.

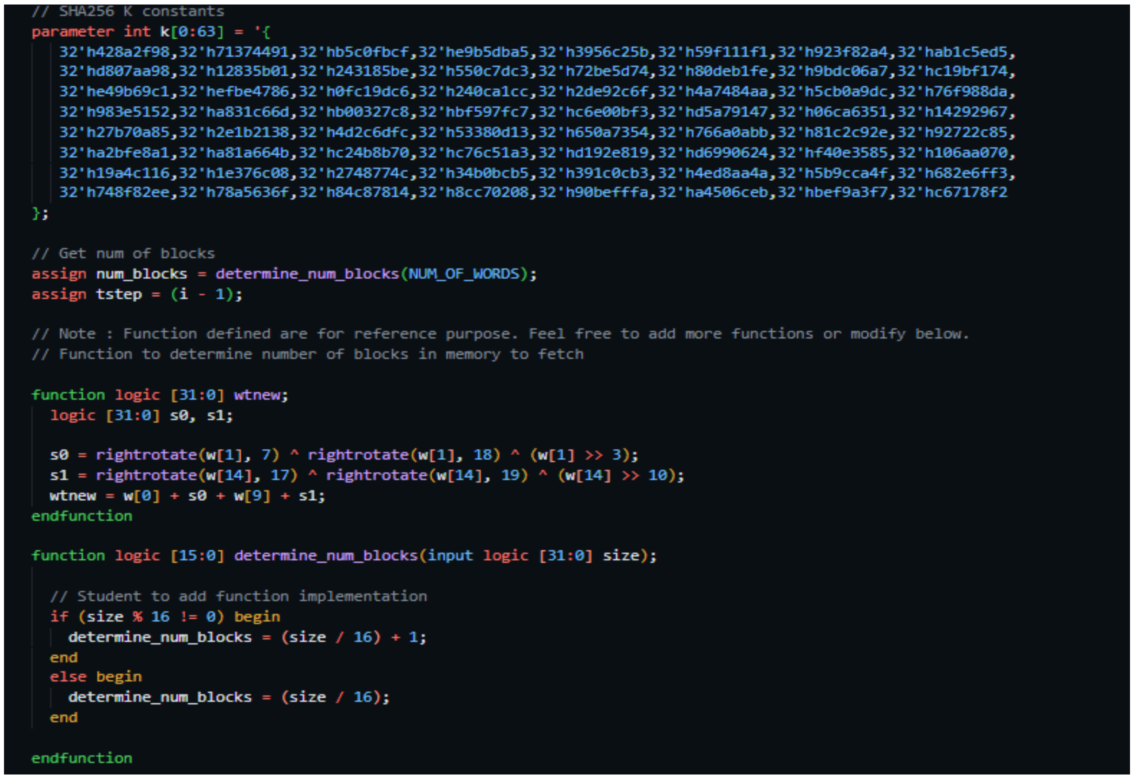
Fifth, SHA-256 must be collision resistant. With SHA algorithms, it is mathematically possible for two completely different inputs to result in the same output digest, due to the fact that compression is occurring. However, the designer must ensure that the output is large enough so that it is not practically possible to manipulate this fact.

Finally, the SHA-256 algorithm must be efficient since this algorithm is often performed many times in the background for integrity verification.

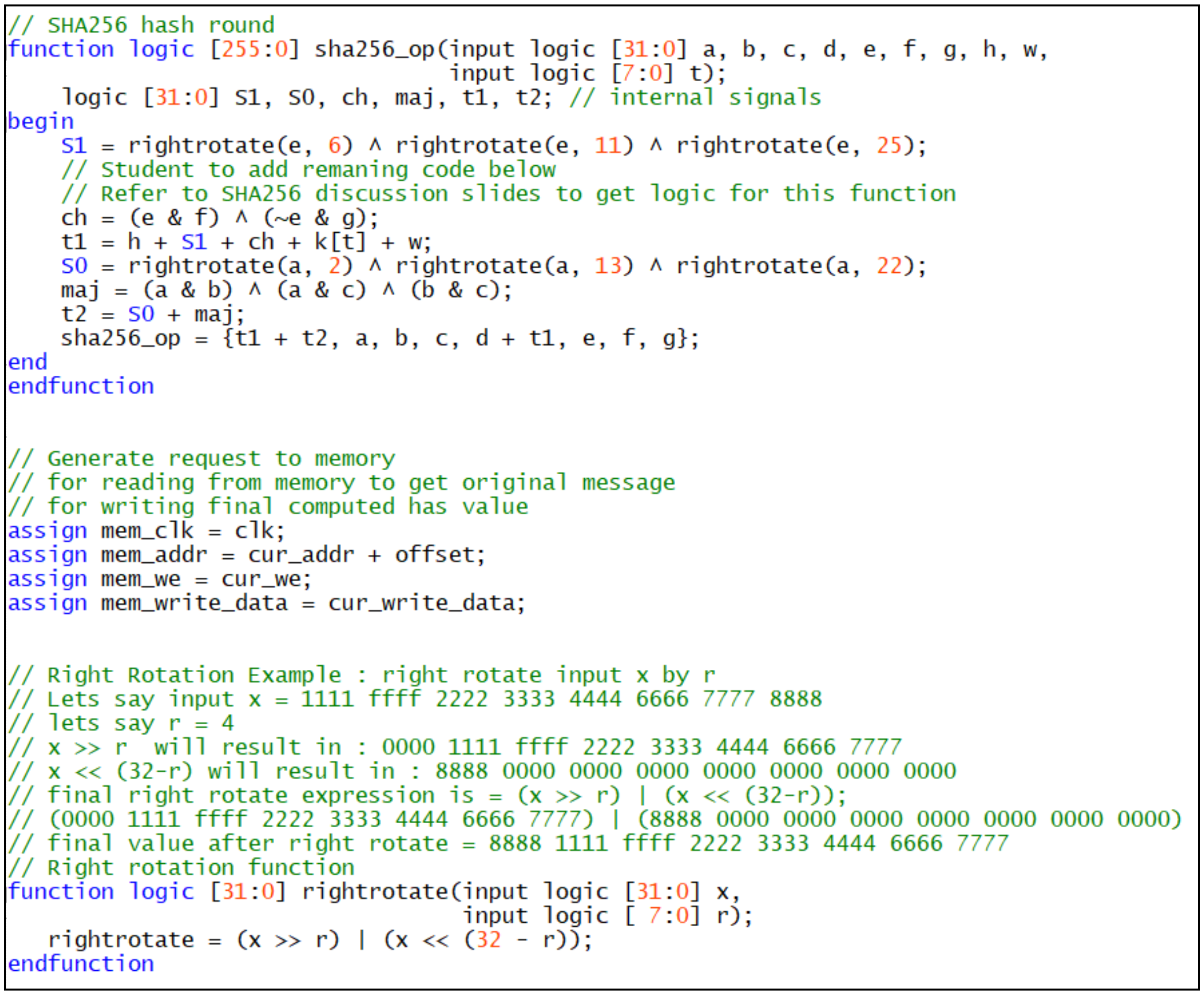
2. SHA-256 code implementation and description



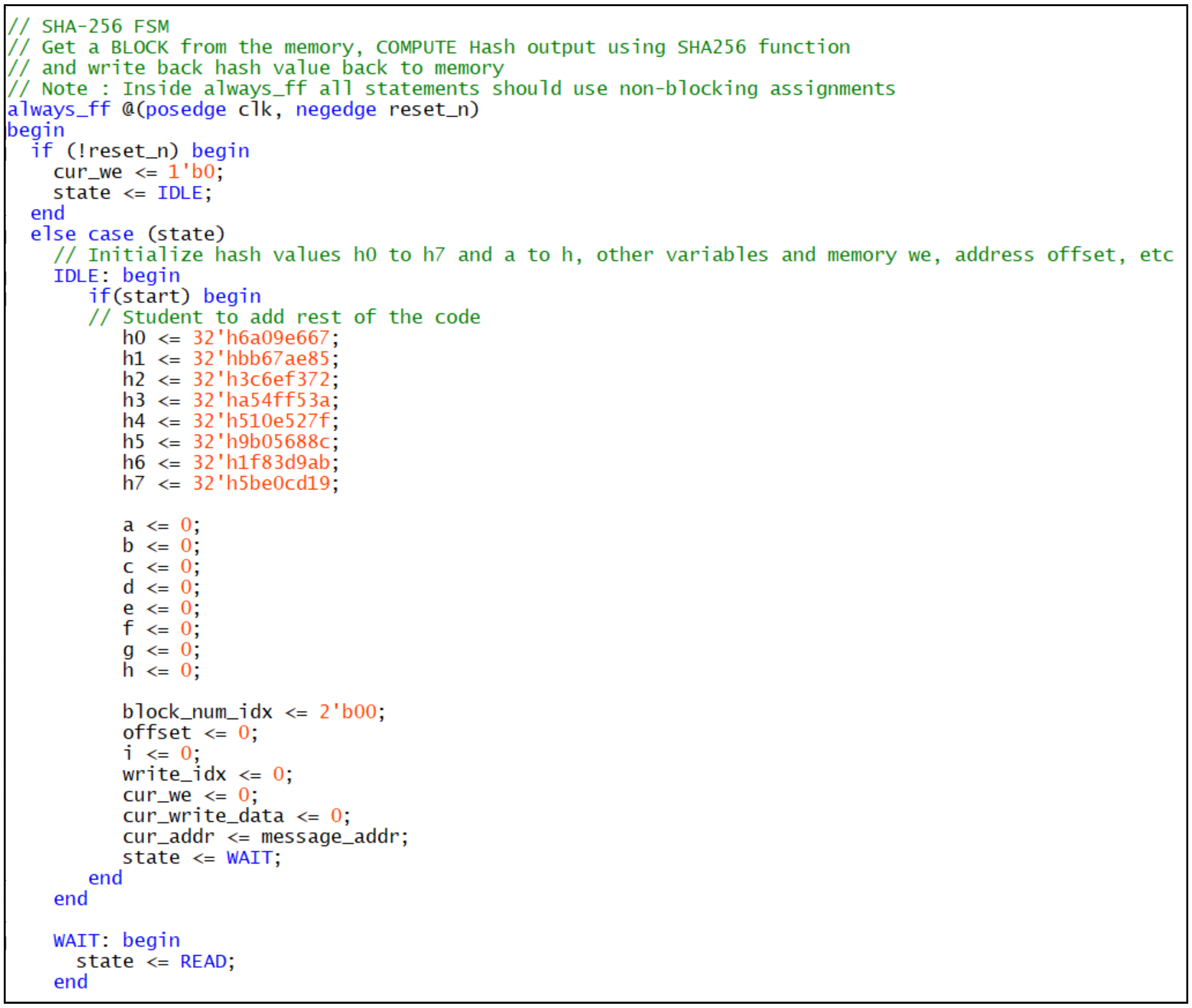
```
module simplified_sha256 #(parameter integer NUM_OF_WORDS = 20)(
 input logic  clk, reset_n, start,
 input logic  [15:0] message_addr, output_addr,
 output logic done, mem_clk, mem_we,
 output logic [15:0] mem_addr,
 output logic [31:0] mem_write_data,
 input logic [31:0] mem_read_data);

// FSM state variables
// Note : Students can add more states or remove states as per their implementation
enum logic [2:0] {IDLE, READ, WAIT, BLOCK, COMPUTE, WRITE} state;

/////////////////////////////////////////////////////////////////////////////////////
/* NOTE : Below mentioned code frame work is for reference purpose.
Local variables might not be complete and you might have to add more variables
or modify these variables. Code below is more as a reference and a helper code
which students can use as a starting point. Student can also develop using your
own method and implementation even without using the code framework provided below.
Code mentioned below is using 1 always block implementation which always_ff
and hence all statements within it should be non-blocking assignment state.
*/
/////////////////////////////////////////////////////////////////////////////////////

// Local variables
// Note : Add or remove variables as per your implementation
logic [31:0] w[16]; // This is for word expansion in compute sate. For optimized implementation this can be w[16]
logic [31:0] message[20]; // Stores 20 message words after read from the memory
logic [31:0] wt;
logic [31:0] h0, h1, h2, h3, h4, h5, h6, h7;
logic [31:0] a, b, c, d, e, f, g, h;
logic [ 7:0] i, j, write_idx;
logic [31:0] SNew0, SNew1;
logic [15:0] offset; // in word address
logic [ 7:0] num_blocks;
logic        cur_we;
logic [15:0] cur_addr;
logic [31:0] cur_write_data;
logic [512:0] memory_block0;
logic [512:0] memory_block1;

logic [1:0] block_num_idx;
logic [ 7:0] tstep;
```



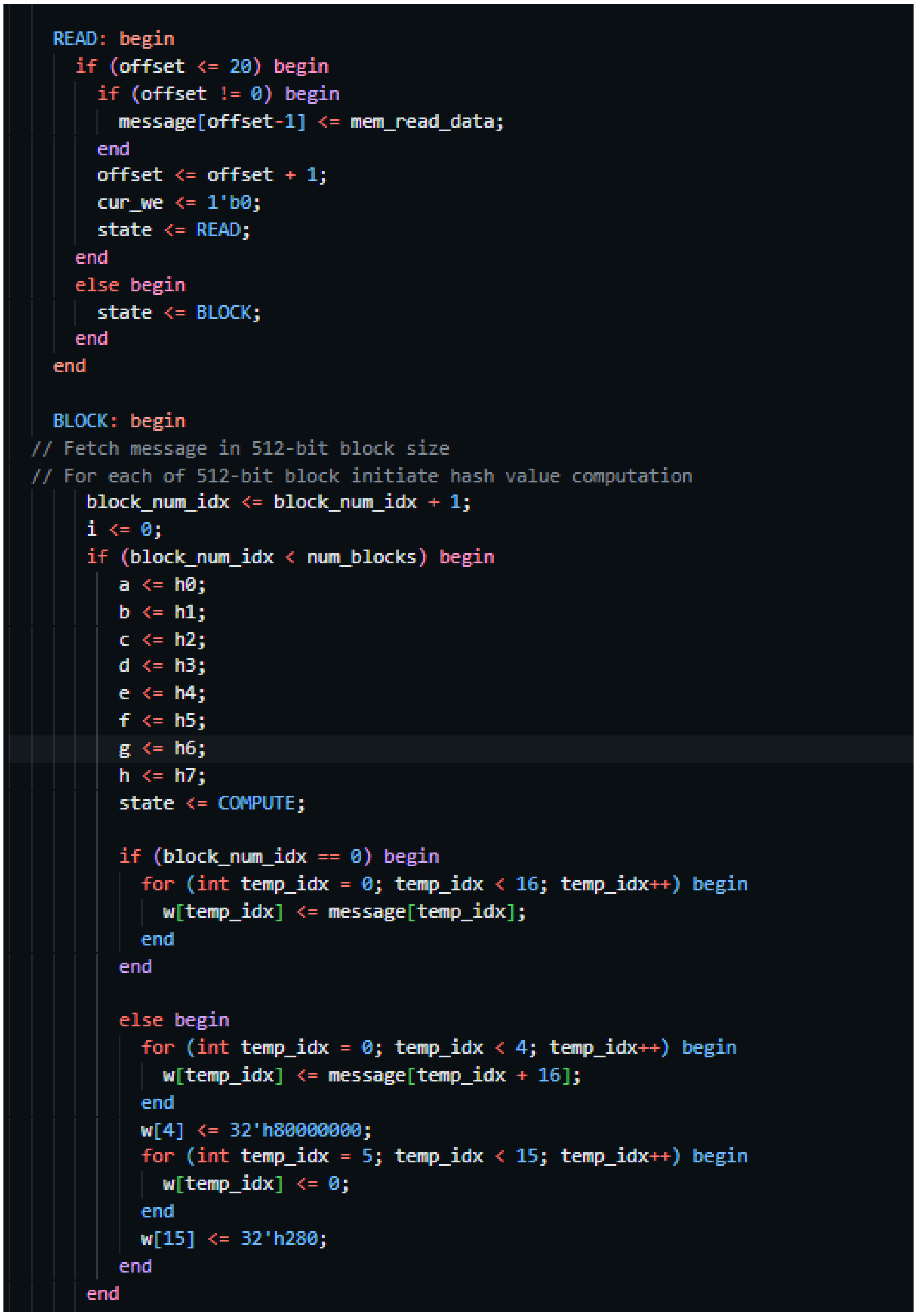
```
// SHA256 K constants
parameter int k[0:63] = '{
    32'h428a2f98,32'h71374491,32'hb5c0fbcf,32'he9b5dba5,32'h3956c25b,32'h59f111f1,32'h923f82a4,32'hab1c5ed5,
    32'hd807aa98,32'h12835b01,32'h243185be,32'h550c7dc3,32'h72be5d74,32'h80deb1fe,32'h9bdc06a7,32'hc19bf174,
    32'he49b69c1,32'hefbe4786,32'h0fc19dc6,32'h240ca1cc,32'h2de92c6f,32'h4a7484aa,32'h5cb0a9dc,32'h76f988da,
    32'h983e5152,32'ha831c66d,32'hb00327c8,32'hbf597fc7,32'hc6e00bf3,32'hd5a79147,32'h06ca6351,32'h14292967,
    32'h27b70a85,32'h2e1b2138,32'h4d2c6dfc,32'h53380d13,32'h650a7354,32'h766a0abb,32'h81c2c92e,32'h92722c85,
    32'ha2bfe8a1,32'ha81a664b,32'hc24b8b70,32'hc76c51a3,32'hd192e819,32'hd6990624,32'hf40e3585,32'h106aa070,
    32'h19a4c116,32'h1e376c08,32'h2748774c,32'h34b0bcb5,32'h391c0cb3,32'h4ed8aa4a,32'h5b9cca4f,32'h682e6ff3,
    32'h748f82ee,32'h78a5636f,32'h84c87814,32'h8cc70208,32'h90befffa,32'ha4506ceb,32'hbef9a3f7,32'hc67178f2
};

// Get num of blocks
assign num_blocks = determine_num_blocks(NUM_OF_WORDS);
assign tstep = (i - 1);

// Note : Function defined are for reference purpose. Feel free to add more functions or modify below.
// Function to determine number of blocks in memory to fetch

function logic [31:0] wtnew;
  logic [31:0] s0, s1;

  s0 = rightrotate(w[1], 7) ^ rightrotate(w[1], 18) ^ (w[1] >> 3);
  s1 = rightrotate(w[14], 17) ^ rightrotate(w[14], 19) ^ (w[14] >> 10);
  wtnew = w[0] + s0 + w[9] + s1;
endfunction

function logic [15:0] determine_num_blocks(input logic [31:0] size);

  // Student to add function implementation
  if (size % 16 != 0) begin
    determine_num_blocks = (size / 16) + 1;
  end
  else begin
    determine_num_blocks = (size / 16);
  end

endfunction
```



```
// SHA256 hash round
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w,
                                 input logic [7:0] t);
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
begin
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
    // Student to add remaning code below
    // Refer to SHA256 discussion slides to get logic for this function
    ch = (e & f) ^ (~e & g);
    t1 = h + S1 + ch + k[t] + w;
    S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
    maj = (a & b) ^ (a & c) ^ (b & c);
    t2 = S0 + maj;
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
end
endfunction


// Generate request to memory
// for reading from memory to get original message
// for writing final computed has value
assign mem_clk = clk;
assign mem_addr = cur_addr + offset;
assign mem_we = cur_we;
assign mem_write_data = cur_write_data;


// Right Rotation Example : right rotate input x by r
// Lets say input x = 1111 ffff 2222 3333 4444 6666 7777 8888
// lets say r = 4
// x >> r  will result in : 0000 1111 ffff 2222 3333 4444 6666 7777
// x << (32-r) will result in : 8888 0000 0000 0000 0000 0000 0000 0000
// final right rotate expression is = (x >> r) | (x << (32-r));
// (0000 1111 ffff 2222 3333 4444 6666 7777) | (8888 0000 0000 0000 0000 0000 0000 0000)
// final value after right rotate = 8888 1111 ffff 2222 3333 4444 6666 7777
// Right rotation function
function logic [31:0] rightrotate(input logic [31:0] x,
                                  input logic [ 7:0] r);
   rightrotate = (x >> r) | (x << (32 - r));
endfunction
```



```
// SHA-256 FSM
// Get a BLOCK from the memory, COMPUTE Hash output using SHA256 function
// and write back hash value back to memory
// Note : Inside always_ff all statements should use non-blocking assignments
always_ff @(posedge clk, negedge reset_n)
begin
  if (!reset_n) begin
    cur_we <= 1'b0;
    state <= IDLE;
  end
  else case (state)
    // Initialize hash values h0 to h7 and a to h, other variables and memory we, address offset, etc
    IDLE: begin
       if(start) begin
       // Student to add rest of the code
          h0 <= 32'h6a09e667;
          h1 <= 32'hbb67ae85;
          h2 <= 32'h3c6ef372;
          h3 <= 32'ha54ff53a;
          h4 <= 32'h510e527f;
          h5 <= 32'h9b05688c;
          h6 <= 32'h1f83d9ab;
          h7 <= 32'h5be0cd19;

          a <= 0;
          b <= 0;
          c <= 0;
          d <= 0;
          e <= 0;
          f <= 0;
          g <= 0;
          h <= 0;

          block_num_idx <= 2'b00;
          offset <= 0;
          i <= 0;
          write_idx <= 0;
          cur_we <= 0;
          cur_write_data <= 0;
          cur_addr <= message_addr;
          state <= WAIT;
       end
    end

    WAIT: begin
      state <= READ;
    end
```



```
  READ: begin
    if (offset <= 20) begin
      if (offset != 0) begin
        message[offset-1] <= mem_read_data;
      end
      offset <= offset + 1;
      cur_we <= 1'b0;
      state <= READ;
    end
    else begin
      state <= BLOCK;
    end
  end

  BLOCK: begin
// Fetch message in 512-bit block size
// For each of 512-bit block initiate hash value computation
     block_num_idx <= block_num_idx + 1;
     i <= 0;
     if (block_num_idx < num_blocks) begin
        a <= h0;
        b <= h1;
        c <= h2;
        d <= h3;
        e <= h4;
        f <= h5;
        g <= h6;
        h <= h7;
        state <= COMPUTE;

        if (block_num_idx == 0) begin
          for (int temp_idx = 0; temp_idx < 16; temp_idx++) begin
            w[temp_idx] <= message[temp_idx];
          end
        end

        else begin
          for (int temp_idx = 0; temp_idx < 4; temp_idx++) begin
            w[temp_idx] <= message[temp_idx + 16];
          end
          w[4] <= 32'h80000000;
          for (int temp_idx = 5; temp_idx < 15; temp_idx++) begin
            w[temp_idx] <= 0;
          end
          w[15] <= 32'h280;
        end
     end
```

```
      else begin
       state <= WRITE;
      end
   end

   COMPUTE: begin
      // 64 processing rounds steps for 512-bit block
       {a, b, c, d, e, f, g, h} <= {h0, h1, h2, h3, h4, h5, h6, h7};

      if (i < 64) begin
         i <= i + 1;
         for (int n = 0; n < 15; n++) begin
           w[n] <= w[n+1];
         end
         w[15] <= wtnew;
         {a, b, c, d, e, f, g, h} <= sha256_op(a, b, c, d, e, f, g, h, w[0], i);
         state <= COMPUTE;
       end

       else begin
           h0 <= h0 + a;
           h1 <= h1 + b;
           h2 <= h2 + c;
           h3 <= h3 + d;
           h4 <= h4 + e;
           h5 <= h5 + f;
           h6 <= h6 + g;
           h7 <= h7 + h;
           state <= BLOCK;
       end
     end
```

```
   WRITE: begin
      if (write_idx<= 7) begin
        state <= WRITE;
        write_idx <= write_idx + 1;
        offset <= write_idx;
        cur_we <= 1'b1;
        cur_addr <= output_addr;
        case (write_idx)
          0: cur_write_data <= h0;
          1: cur_write_data <= h1;
          2: cur_write_data <= h2;
          3: cur_write_data <= h3;
          4: cur_write_data <= h4;
          5: cur_write_data <= h5;
          6: cur_write_data <= h6;
          7: cur_write_data <= h7;
        endcase
      end
      else begin
        state <= IDLE;
      end
    end

  endcase
end
// Generate done when SHA256 hash computation has finished and moved to IDLE state
assign done = (state == IDLE);

endmodule
```

Description:

Overall, the SHA-256 algorithm we implemented takes in an input message from memory and performs various processes to produce the final hash which is written back to memory. The input message is processed in blocks of 512 bits, and the corresponding hashes for each round are produced according to the given initial hash-inputs. All parts of the SHA-256 algorithm are written within the simplified_sha256 module above and the major parts of the algorithm will be described below.

Firstly, the rightrotate function, given to us, takes in a 32-bit input and rotates it to the right by a given number of bits. This function is crucial to the SHA-256 algorithm as it defines the compression process, ensuring that small changes in the input lead to drastic changes in the output digest.

This function is used by the wtnew function which generates additional words for the initial 16 32-bit input words. We need additional words because SHA-256 is designed to take in 64 32-bit words so that 64 rounds of processing for each 512-bit block can be performed. Therefore, the wtnew function addresses this issue by performing word expansion.

The new message created by wtnew and the rightrotate function are then used in the sha256_op function. The overall functionality of this function was given to us; however, the main takeaway is that after calling this function 64 times a new hash is created for the next 512-bit block.

Apart from these primary functions, the FSM is the main part of the code determining proper functionality. The FSM moves through the following states: IDLE, WAIT, READ, BlOCK, COMPUTE, and WRITE.

To begin, the IDLE state waits for the state signal to be triggered. Once triggered, IDLE initiates the predetermined hash values and other local variables to be used in the other states. After the initialization is completed, the FSM transitions to the WAIT state.

The WAIT state is a temporary buffer state which quickly transitions to the READ state. The READ states reads and stores the message stored in the memory to message. After the message is stored, the FSM transitions to the BLOCK state.

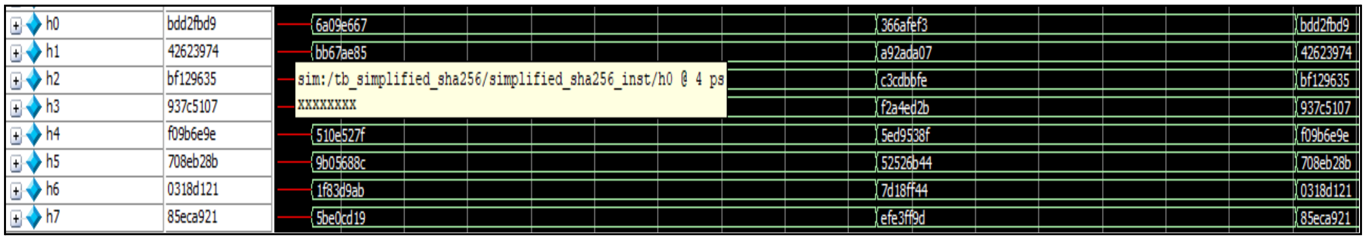
The BLOCK state slices the message into 512-bit blocks. If the message is not evenly divisible by 512-bits, padding is added accordingly. After the blocks are created, the FSM transitions to the COMPUTE state.

The COMPUTE state runs each block through 64 rounds of compression, performed by the sha256 function. Once the compression is completed, the hash values are updated, and the FSM moves back to the BLOCK state to continue processing blocks. However, if there are no more blocks to process, the FSM will move to the WRITE state.

The WRITE state stores the final hash to memory. Once the hash is set in memory, the FSM moves to the IDLE state and the done signal is triggered.

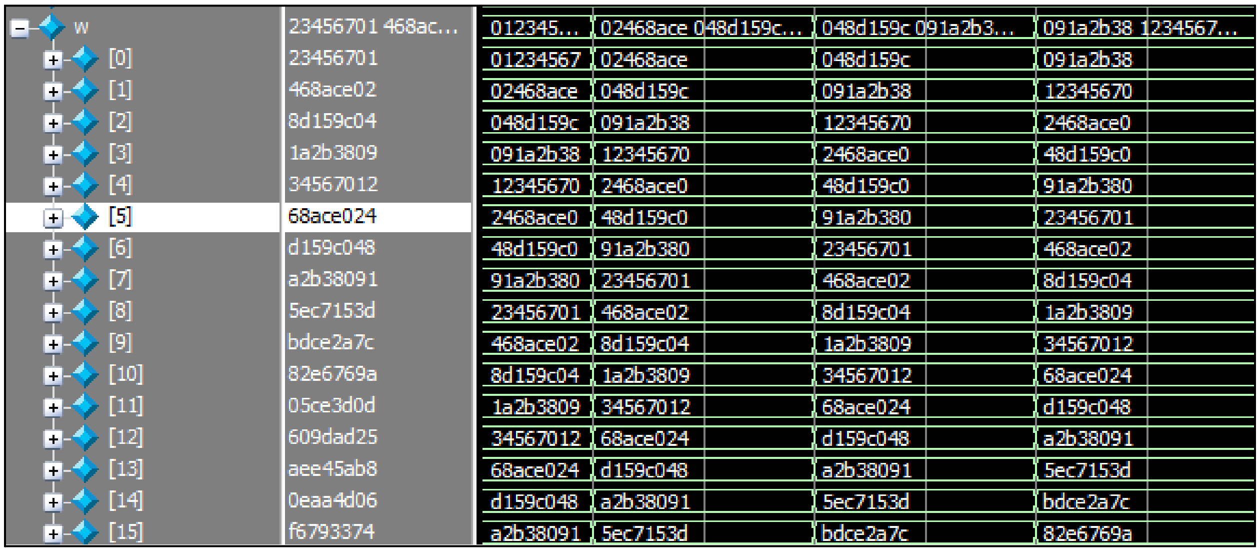
3. ModelSim Simulation Waveform

h0 - h7



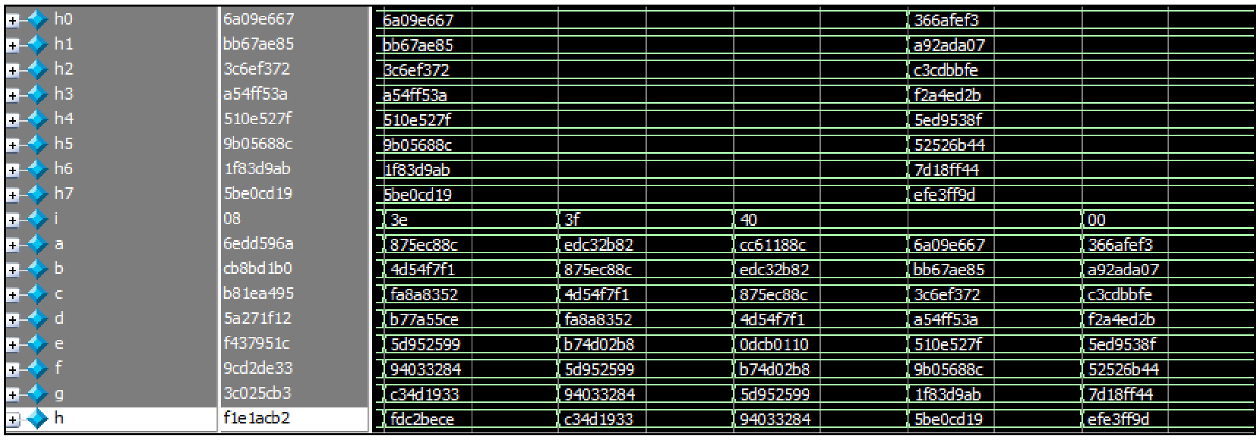
```
h0  bdd2fbd9   6a09e667                                      366afef3   bdd2fbd9
h1  42623974   bb67ae85                                      a92ada07   42623974
h2  bf129635   sim:/tb_simplified_sha256/simplified_sha256_inst/h0 @ 4 ps   c3cdbbfe   bf129635
h3  937c5107   xxxxxxxx                                      f2a4ed2b   937c5107
h4  f09b6e9e   510e527f                                      5ed9538f   f09b6e9e
h5  708eb28b   9b05688c                                      52526b44   708eb28b
h6  0318d121   1f83d9ab                                      7d18ff44   0318d121
h7  85eca921   5be0cd19                                      efe3ff9d   85eca921
```

w0 - w15 optimization (w[15] at the next timestep contains the new wt computed for w[16]-w[64].



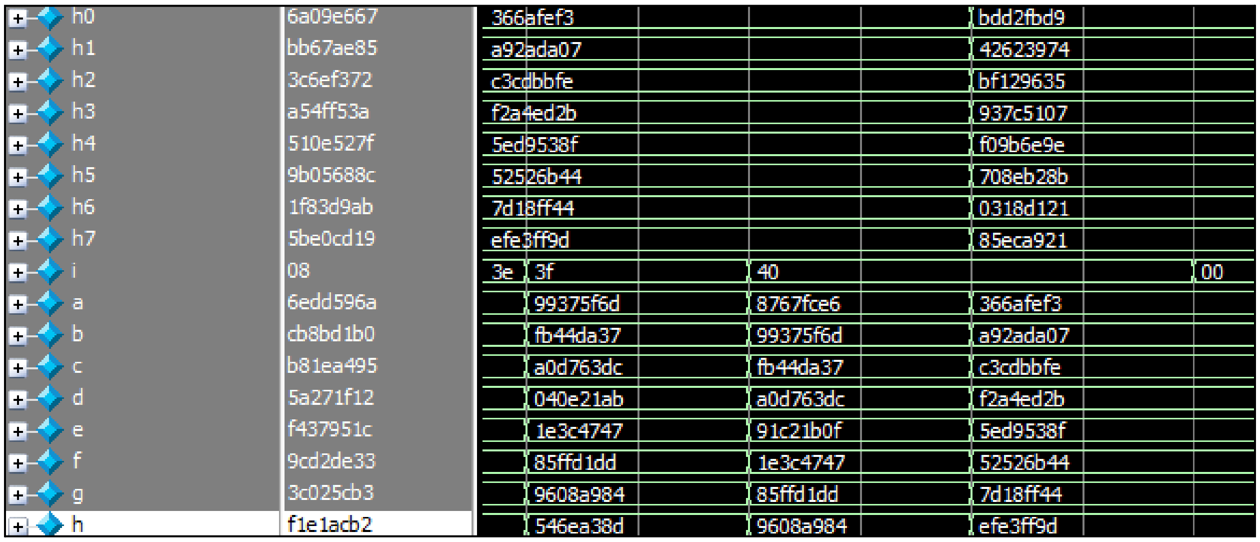
```
w      23456701 468ac...   012345...  02468ace 048d159c...  048d159c 091a2b3...  091a2b38 1234567...
[0]    23456701            01234567   02468ace              048d159c             091a2b38
[1]    468ace02            02468ace   048d159c              091a2b38             12345670
[2]    8d159c04            048d159c   091a2b38              12345670             2468ace0
[3]    1a2b3809            091a2b38   12345670              2468ace0             48d159c0
[4]    34567012            12345670   2468ace0              48d159c0             91a2b380
[5]    68ace024            2468ace0   48d159c0              91a2b380             23456701
[6]    d159c048            48d159c0   91a2b380              23456701             468ace02
[7]    a2b38091            91a2b380   23456701              468ace02             8d159c04
[8]    5ec7153d            23456701   468ace02              8d159c04             1a2b3809
[9]    bdce2a7c            468ace02   8d159c04              1a2b3809             34567012
[10]   82e6769a            8d159c04   1a2b3809              34567012             68ace024
[11]   05ce3d0d            1a2b3809   34567012              68ace024             d159c048
[12]   609dad25            34567012   68ace024              d159c048             a2b38091
[13]   aee45ab8            68ace024   d159c048              a2b38091             5ec7153d
[14]   0eaa4d06            d159c048   a2b38091              5ec7153d             bdce2a7c
[15]   f6793374            a2b38091   5ec7153d              bdce2a7c             82e6769a
```

After i reaches 64 in phase 2 (41 in hexadecimal), the new set of a, b, c, d, e, f, g, and h are stored in h0-h7.



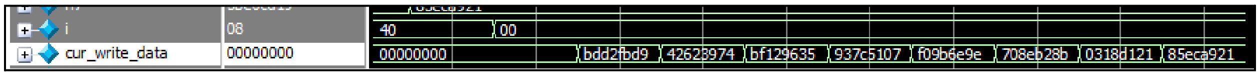
```
h0   6a09e667   6a09e667                                   366afef3
h1   bb67ae85   bb67ae85                                   a92ada07
h2   3c6ef372   3c6ef372                                   c3cdbbfe
h3   a54ff53a   a54ff53a                                   f2a4ed2b
h4   510e527f   510e527f                                   5ed9538f
h5   9b05688c   9b05688c                                   52526b44
h6   1f83d9ab   1f83d9ab                                   7d18ff44
h7   5be0cd19   5be0cd19                                   efe3ff9d
i    08         3e         3f         40                               00
a    6edd596a   875ec88c   edc32b82   cc61188c   6a09e667   366afef3
b    cb8bd1b0   4d54f7f1   875ec88c   edc32b82   bb67ae85   a92ada07
c    b81ea495   fa8a8352   4d54f7f1   875ec88c   3c6ef372   c3cdbbfe
d    5a271f12   b77a55ce   fa8a8352   4d54f7f1   a54ff53a   f2a4ed2b
e    f437951c   5d952599   b74d02b8   0dcb0110   510e527f   5ed9538f
f    9cd2de33   94033284   5d952599   b74d02b8   9b05688c   52526b44
g    3c025cb3   c34d1933   94033284   5d952599   1f83d9ab   7d18ff44
h    f1e1acb2   fdc2bece   c34d1933   94033284   5be0cd19   efe3ff9d
```

Correct hashes are also stored after i reaches 64 in the phase 3 state.



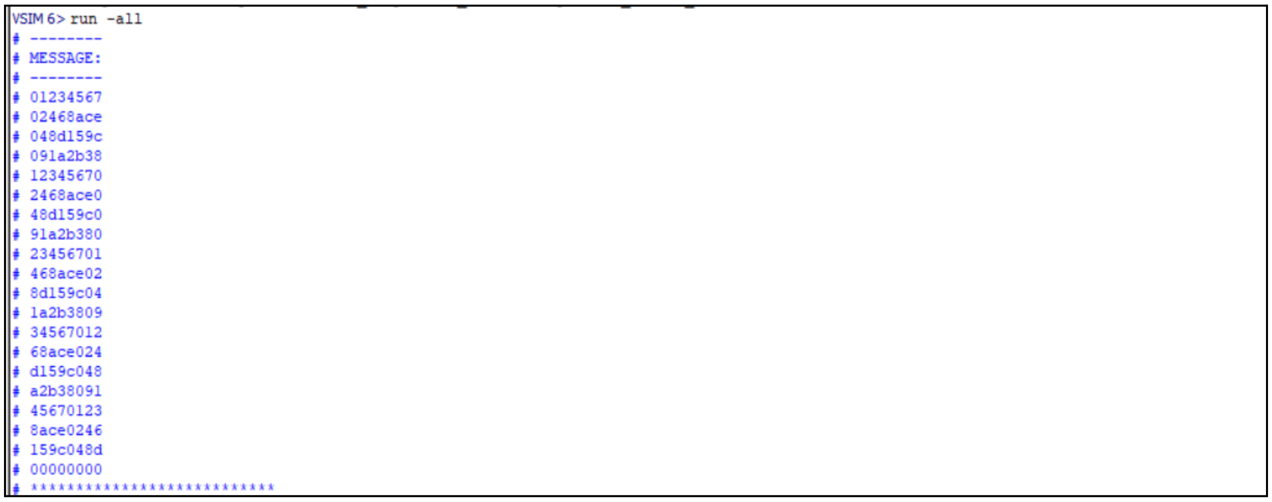
```
h0  6a09e667  366afef3                          bdd2fbd9
h1  bb67ae85  a92ada07                          42623974
h2  3c6ef372  c3cdbbfe                          bf129635
h3  a54ff53a  f2a4ed2b                          937c5107
h4  510e527f  5ed9538f                          f09b6e9e
h5  9b05688c  52526b44                          708eb28b
h6  1f83d9ab  7d18ff44                          0318d121
h7  5be0cd19  efe3ff9d                          85eca921
i   08        3e  3f         40                           00
a   6edd596a      99375f6d   8767fce6           366afef3
b   cb8bd1b0      fb44da37   99375f6d           a92ada07
c   b81ea495      a0d763dc   fb44da37           c3cdbbfe
d   5a271f12      040e21ab   a0d763dc           f2a4ed2b
e   f437951c      1e3c4747   91c21b0f           5ed9538f
f   9cd2de33      85ffd1dd   1e3c4747           52526b44
g   3c025cb3      9608a984   85ffd1dd           7d18ff44
h   f1e1acb2      546ea38d   9608a984           efe3ff9d
```

Correct outputs stored after reaching write state (occurs after i = 64 when in phase3 compute for the 2nd time).

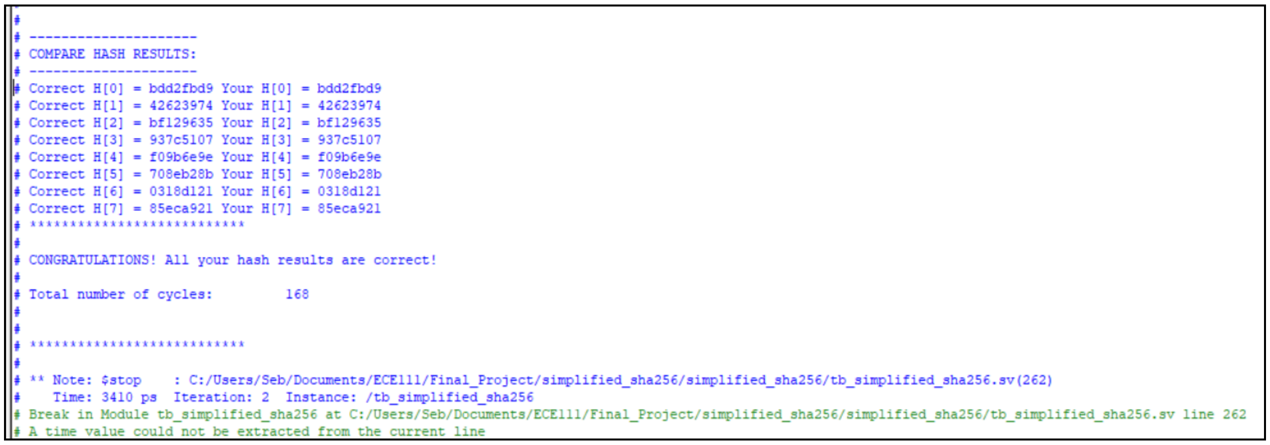


```
i              08        40    00
cur_write_data 00000000  00000000   bdd2fbd9  42623974  bf129635  937c5107  f09b6e9e  708eb28b  0318d121  85eca921
```

4. ModelSim Simulation Output transcript



```
VSIM 6> run -all
# --------
# MESSAGE:
# --------
# 01234567
# 02468ace
# 048d159c
# 091a2b38
# 12345670
# 2468ace0
# 48d159c0
# 91a2b380
# 23456701
# 468ace02
# 8d159c04
# 1a2b3809
# 34567012
# 68ace024
# d159c048
# a2b38091
# 45670123
# 8ace0246
# 159c048d
# 00000000
# ***************************
```



```
#
# ---------------------
# COMPARE HASH RESULTS:
# ---------------------
# Correct H[0] = bdd2fbd9 Your H[0] = bdd2fbd9
# Correct H[1] = 42623974 Your H[1] = 42623974
# Correct H[2] = bf129635 Your H[2] = bf129635
# Correct H[3] = 937c5107 Your H[3] = 937c5107
# Correct H[4] = f09b6e9e Your H[4] = f09b6e9e
# Correct H[5] = 708eb28b Your H[5] = 708eb28b
# Correct H[6] = 0318d121 Your H[6] = 0318d121
# Correct H[7] = 85eca921 Your H[7] = 85eca921
# ***************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:         168
#
#
# ***************************
#
# ** Note: $stop    : C:/Users/Seb/Documents/ECE111/Final_Project/simplified_sha256/simplified_sha256/tb_simplified_sha256.sv(262)
#    Time: 3410 ps  Iteration: 2  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at C:/Users/Seb/Documents/ECE111/Final_Project/simplified_sha256/simplified_sha256/tb_simplified_sha256.sv line 262
# A time value could not be extracted from the current line
```

Explanation:

As shown above, our SHA-256 implementation works as expected as it produces the correct results after the 3 phases of compute, where phase2 compute and phase3 compute occur twice. This indicates that the internal code and all steps of generating blocks and computation is sound and works as intended. Furthermore, our code produces the final hash after 168 cycles, showing the effectiveness of using a w[16] shift register implementation rather than w[64] with multiple multiplexers.

Final Project Part-2: Blockchain and Bitcoin Hashing

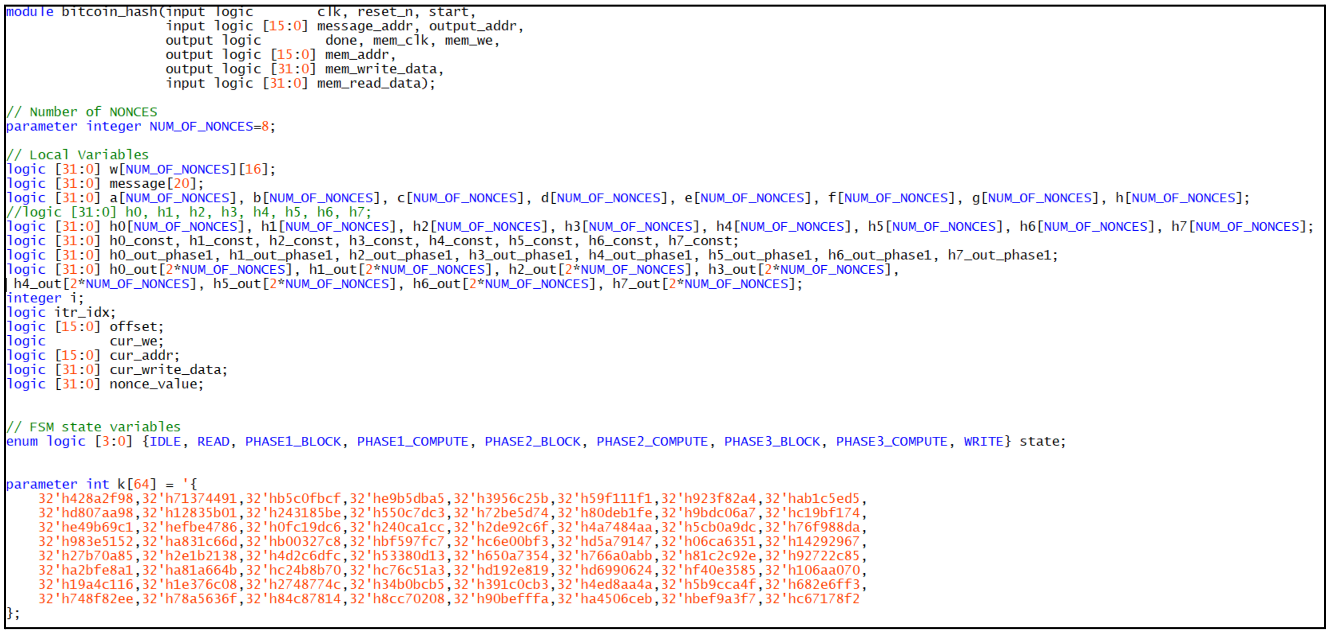
1. Bitcoin Hashing Explanation

A blockchain, which is a chain of digital blocks, uses data blocks of size 512. In the blockchain, the output signature of say, block1, is the same as the input signature of block2 as they are linked. Blocks that are linked to each other share the same signature. This signature is generated by the previously mentioned hashing algorithm, SHA-256. What makes this concept powerful in terms of security is that the data within the blocks in the blockchain are immutable, not capable of being altered. Bitcoin hashing must satisfy/be: deterministic, compression, pre-image resistant, avalanche effect, collision resistant, and efficient. These properties are all satisfied by SHA 256.

Consider a scenario where a hacker alters the transaction that occurs in block-1, where the signature linking block-1 and block-2 is X32. When the hacker modifies the transaction details, the signature of the block also changes, say to W10. However, block-2's signature is X32, and since they do not match, they are no longer chained, and the blockchain shifts to where previously all blocks were chained, in this case, block-2 and block-3. The block-1 transaction is effectively void.

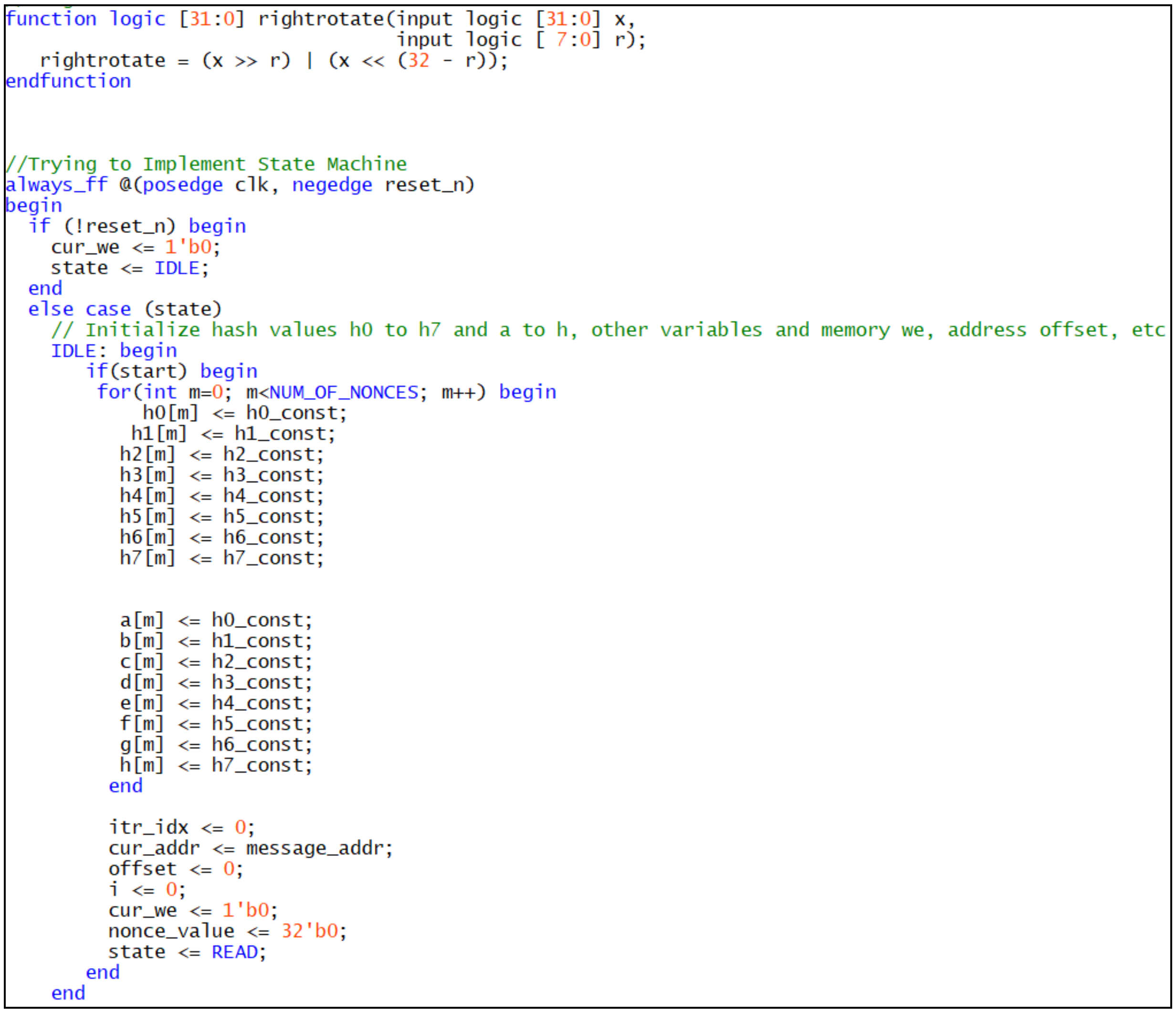
To strengthen the security of the signatures generated, a nonce can be used to satisfy the specific security requirement of a blockchain. A nonce is a random string of numbers that is appended somewhere alongside the transaction data and metadata while preserving the two. Nonce values are generated multiple times in attempts to satisfy a specific security requirement, say, the signature/hash of a block starting with a specific sequence of characters. This makes it difficult for hackers to modify a transaction in the blockchain as they have to compute signatures for not only the altered block but new blocks in the blockchain.

2. Bitcoin hashing implementation and description



```
module bitcoin_hash(input logic        clk, reset_n, start,
                    input logic [15:0] message_addr, output_addr,
                    output logic        done, mem_clk, mem_we,
                    output logic [15:0] mem_addr,
                    output logic [31:0] mem_write_data,
                    input logic [31:0] mem_read_data);

// Number of NONCES
parameter integer NUM_OF_NONCES=8;

// Local Variables
logic [31:0] w[NUM_OF_NONCES][16];
logic [31:0] message[20];
logic [31:0] a[NUM_OF_NONCES], b[NUM_OF_NONCES], c[NUM_OF_NONCES], d[NUM_OF_NONCES], e[NUM_OF_NONCES], f[NUM_OF_NONCES], g[NUM_OF_NONCES], h[NUM_OF_NONCES];
//logic [31:0] h0, h1, h2, h3, h4, h5, h6, h7;
logic [31:0] h0[NUM_OF_NONCES], h1[NUM_OF_NONCES], h2[NUM_OF_NONCES], h3[NUM_OF_NONCES], h4[NUM_OF_NONCES], h5[NUM_OF_NONCES], h6[NUM_OF_NONCES], h7[NUM_OF_NONCES];
logic [31:0] h0_const, h1_const, h2_const, h3_const, h4_const, h5_const, h6_const, h7_const;
logic [31:0] h0_out_phase1, h1_out_phase1, h2_out_phase1, h3_out_phase1, h4_out_phase1, h5_out_phase1, h6_out_phase1, h7_out_phase1;
logic [31:0] h0_out[2*NUM_OF_NONCES], h1_out[2*NUM_OF_NONCES], h2_out[2*NUM_OF_NONCES], h3_out[2*NUM_OF_NONCES],
 h4_out[2*NUM_OF_NONCES], h5_out[2*NUM_OF_NONCES], h6_out[2*NUM_OF_NONCES], h7_out[2*NUM_OF_NONCES];
integer i;
logic itr_idx;
logic [15:0] offset;
logic        cur_we;
logic [15:0] cur_addr;
logic [31:0] cur_write_data;
logic [31:0] nonce_value;


// FSM state variables
enum logic [3:0] {IDLE, READ, PHASE1_BLOCK, PHASE1_COMPUTE, PHASE2_BLOCK, PHASE2_COMPUTE, PHASE3_BLOCK, PHASE3_COMPUTE, WRITE} state;


parameter int k[64] = '{
    32'h428a2f98,32'h71374491,32'hb5c0fbcf,32'he9b5dba5,32'h3956c25b,32'h59f111f1,32'h923f82a4,32'hab1c5ed5,
    32'hd807aa98,32'h12835b01,32'h243185be,32'h550c7dc3,32'h72be5d74,32'h80deb1fe,32'h9bdc06a7,32'hc19bf174,
    32'he49b69c1,32'hefbe4786,32'h0fc19dc6,32'h240ca1cc,32'h2de92c6f,32'h4a7484aa,32'h5cb0a9dc,32'h76f988da,
    32'h983e5152,32'ha831c66d,32'hb00327c8,32'hbf597fc7,32'hc6e00bf3,32'hd5a79147,32'h06ca6351,32'h14292967,
    32'h27b70a85,32'h2e1b2138,32'h4d2c6dfc,32'h53380d13,32'h650a7354,32'h766a0abb,32'h81c2c92e,32'h92722c85,
    32'ha2bfe8a1,32'ha81a664b,32'hc24b8b70,32'hc76c51a3,32'hd192e819,32'hd6990624,32'hf40e3585,32'h106aa070,
    32'h19a4c116,32'h1e376c08,32'h2748774c,32'h34b0bcb5,32'h391c0cb3,32'h4ed8aa4a,32'h5b9cca4f,32'h682e6ff3,
    32'h748f82ee,32'h78a5636f,32'h84c87814,32'h8cc70208,32'h90befffa,32'ha4506ceb,32'hbef9a3f7,32'hc67178f2
};
```

```
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w,
                                 input logic [7:0] t);
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
begin
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
    ch = (e & f) ^ ((~e) & g);
    t1 = h + S1 + ch + k[t] + w;
    S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
    maj = (a & b) ^ (a & c) ^ (b & c);
    t2 = S0 + maj;
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
end
endfunction

// Modified word expansion for bitcoin parallel implementation to have "n" parameter below
function logic [31:0] wt_expansion(input logic [15:0] n);
 logic [31:0] s1, s0;
  s0 = rightrotate(w[n][1], 7) ^ rightrotate(w[n][1], 18) ^ (w[n][1] >> 3);
  s1 = rightrotate(w[n][14], 17) ^ rightrotate(w[n][14], 19) ^ (w[n][14] >> 10);
  wt_expansion = w[n][0] + s0 + w[n][9] + s1;
endfunction

// Generate request to memory
// for reading from memory to get original message
// for writing final computed has value
assign mem_clk = clk;
assign mem_addr = cur_addr + offset;
assign mem_we = cur_we;
assign mem_write_data = cur_write_data;

// Setting Initial Hash Values
assign h0_const = 32'h6a09e667;
assign h1_const = 32'hbb67ae85;
assign h2_const = 32'h3c6ef372;
assign h3_const = 32'ha54ff53a;
assign h4_const = 32'h510e527f;
assign h5_const = 32'h9b05688c;
assign h6_const = 32'h1f83d9ab;
assign h7_const = 32'h5be0cd19;

// Right Rotation Example : right rotate input x by r
// Lets say input x = 1111 ffff 2222 3333 4444 6666 7777 8888
// lets say r = 4
// x >> r  will result in : 0000 1111 ffff 2222 3333 4444 6666 7777
// x << (32-r) will result in : 8888 0000 0000 0000 0000 0000 0000 0000
// final right rotate expression is = (x >> r) | (x << (32-r));
// (0000 1111 ffff 2222 3333 4444 6666 7777) | (8888 0000 0000 0000 0000 0000 0000 0000)
// final value after right rotate = 8888 1111 ffff 2222 3333 4444 6666 7777
// Right rotation function
```



```
function logic [31:0] rightrotate(input logic [31:0] x,
                                  input logic [ 7:0] r);
   rightrotate = (x >> r) | (x << (32 - r));
endfunction



//Trying to Implement State Machine
always_ff @(posedge clk, negedge reset_n)
begin
  if (!reset_n) begin
    cur_we <= 1'b0;
    state <= IDLE;
  end
  else case (state)
    // Initialize hash values h0 to h7 and a to h, other variables and memory we, address offset, etc
    IDLE: begin
       if(start) begin
        for(int m=0; m<NUM_OF_NONCES; m++) begin
            h0[m] <= h0_const;
           h1[m] <= h1_const;
          h2[m] <= h2_const;
          h3[m] <= h3_const;
          h4[m] <= h4_const;
          h5[m] <= h5_const;
          h6[m] <= h6_const;
          h7[m] <= h7_const;


          a[m] <= h0_const;
          b[m] <= h1_const;
          c[m] <= h2_const;
          d[m] <= h3_const;
          e[m] <= h4_const;
          f[m] <= h5_const;
          g[m] <= h6_const;
          h[m] <= h7_const;
         end

         itr_idx <= 0;
         cur_addr <= message_addr;
         offset <= 0;
         i <= 0;
         cur_we <= 1'b0;
         nonce_value <= 32'b0;
         state <= READ;
       end
    end
```

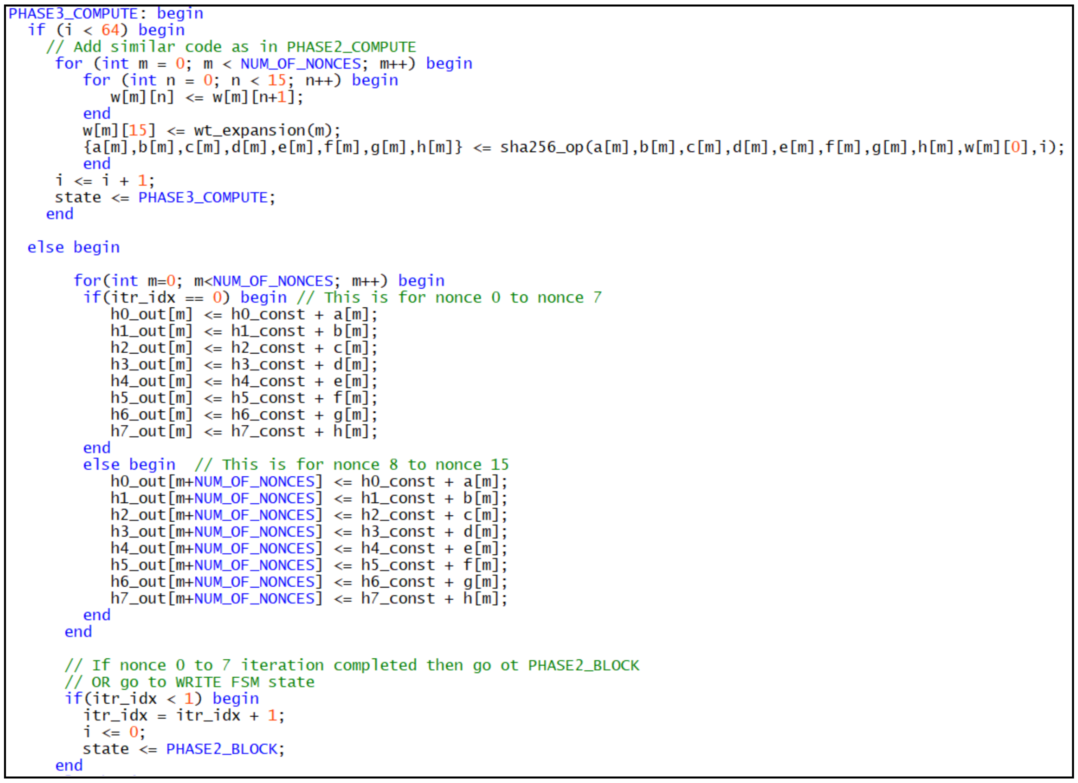
```
READ: begin
  // Add code to READ all 20 input message words from memory same logic
  // as in simplified_sha256 FSM or done in bitcoin hash serial implementation
  // end of this read message[0] to message[19] will have all 20 inpute message words
  if (offset <= 20) begin
    if (offset != 0) begin
      message[offset-1] <= mem_read_data;
    end
    offset <= offset + 1;
    cur_we <= 1'b0;
    state <= READ;
  end
  else begin
    state <= PHASE1_BLOCK;
  end
end

PHASE1_BLOCK: begin
   // Add code to Fill w[0][n] to w[15] with message[0] to message[15]
  // Since first block in PHASE1 does not have nonce, only file w[0] element
  // So fille w[0][0], w[0][1] to w[0][15] words with message[0] to message[15]
  for (int temp_idx = 0; temp_idx < 16; temp_idx++) begin
     w[0][temp_idx] <= message[temp_idx];
  end

  // Initial hash constants
  // Since first block there is no NONCE, so only use a[0] to h[0]
  // hence only fill a[o] to h[0] with hash constants
  a[0] <= h0_const;
  b[0] <= h1_const;
  c[0] <= h2_const;
  d[0] <= h3_const;
  e[0] <= h4_const;
  f[0] <= h5_const;
  g[0] <= h6_const;
  h[0] <= h7_const;

  i <= 0;
  state <= PHASE1_COMPUTE;
end
```

```
PHASE1_COMPUTE: begin
   if (i < 64) begin
     // Add similar code as in COMPUTE FSM state in simplified_sha256 FSM
    // Ensure w[16] optimized implementation is used
    // And make sure use a[0],b[0],c[0],d[0],e[0],f[0],g[0],h[0] insead of
    // a,b,c,d,e,f,g,h for bitcoin parallel implementation
      i <= i + 1;
      for (int n = 0; n < 15; n++) begin
        w[0][n] <= w[0][n+1];
      end
      w[0][15] <= wt_expansion(0);
      {a[0], b[0], c[0], d[0], e[0], f[0], g[0], h[0]} <= sha256_op(a[0], b[0], c[0], d[0], e[0], f[0], g[0], h[0], w[0][0], i);
      i <= i + 1;
     state <= PHASE1_COMPUTE;
   end
   else begin
       h0[0] <= h0[0] + a[0];
      h1[0] <= h1[0] + b[0];
      h2[0] <= h2[0] + c[0];
      h3[0] <= h3[0] + d[0];
      h4[0] <= h4[0] + e[0];
      h5[0] <= h5[0] + f[0];
      h6[0] <= h6[0] + g[0];
      h7[0] <= h7[0] + h[0];

      // Store phase1 output hash for later use at end of in PHASE3_COMPUTE before PHASE2_COMPUTE starts again for next nonce iteration
      h0_out_phase1 <= h0[0] + a[0];
      h1_out_phase1 <= h1[0] + b[0];
      h2_out_phase1 <= h2[0] + c[0];
      h3_out_phase1 <= h3[0] + d[0];
      h4_out_phase1 <= h4[0] + e[0];
      h5_out_phase1 <= h5[0] + f[0];
      h6_out_phase1 <= h6[0] + g[0];
      h7_out_phase1 <= h7[0] + h[0];

      state <= PHASE2_BLOCK;
   end
end
```

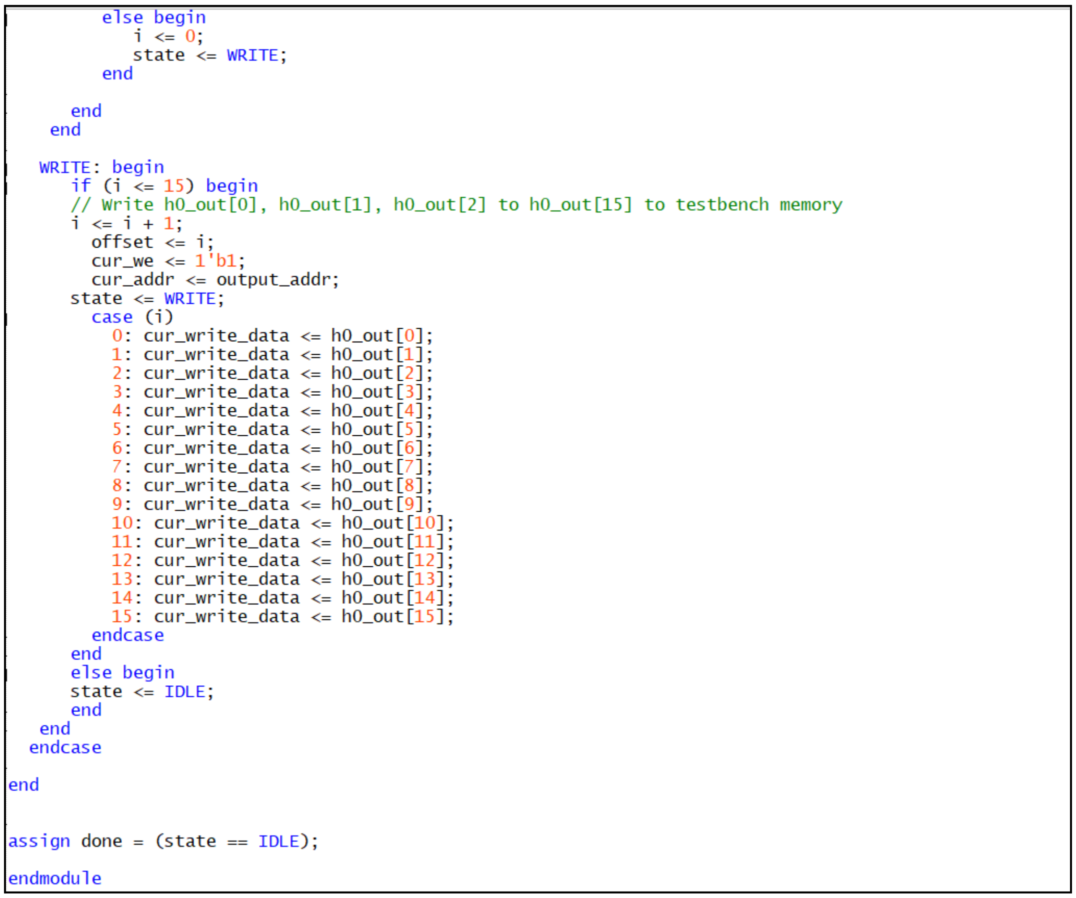
```
PHASE2_BLOCK: begin
   // Within for loop with m=0 and m<NUM_OF_NONCES Add code to Fill in w[m][0] to w[m][16] with message words, nonce and padding bits, message, size
  for(int m=0; m<NUM_OF_NONCES; m++) begin
  //w[m][0] to w[m][2] using message[16] to message[18]
   //Add code to check if itr_idx == 0 then w[m][3] <= m else  w[m][3] <= m + NUM_OF_NONCES;
  //w[m][4] <= 32'h80000000;
  //w[m][5] to w[m][15] to 0
  //w[m][15] = 32'd640;
     for (int temp_idx = 0; temp_idx < 3; temp_idx++) begin
        w[m][temp_idx] <= message[temp_idx + 16];
     end
     if (itr_idx == 0) begin
        w[m][3] <= m;
     end
     else begin
        w[m][3] <= m + NUM_OF_NONCES;
     end

     w[m][4] <= 32'h80000000;
     for (int idx = 5; idx < 15; idx++) begin
        w[m][idx] <= 0;
     end
     w[m][15] <= 32'd640;
  end

    for(int m=0; m<NUM_OF_NONCES; m++) begin
     // Add code to initialize a through h using h0 to h7 which was generated from PHASE1_COMPUTE
     a[m] <= h0_out_phase1;
     b[m] <= h1_out_phase1;
     c[m] <= h2_out_phase1;
     d[m] <= h3_out_phase1;
     e[m] <= h4_out_phase1;
     f[m] <= h5_out_phase1;
     g[m] <= h6_out_phase1;
     h[m] <= h7_out_phase1;

     // Initialize h0 to h7 for each nonce with first block hash output i.e. hash output from PHASE1_COMPUTE
     h0[m] <= h0_out_phase1;
     h1[m] <= h1_out_phase1;
     h2[m] <= h2_out_phase1;
     h3[m] <= h3_out_phase1;
     h4[m] <= h4_out_phase1;
     h5[m] <= h5_out_phase1;
     h6[m] <= h6_out_phase1;
     h7[m] <= h7_out_phase1;
  end
```

```
    i <= 0;
    state <= PHASE2_COMPUTE;
  end

  PHASE2_COMPUTE: begin

    // Note : Code below is not complete. Just few guidelines how to support multiple
    // nonce implementation using for loop
     if (i < 64) begin
        // similar code as PHASE1_COMPUTE however use {a[m],b[m],c[m],d[m],e[m],f[m],g[m],h[m]}
      //insead of a[0],b[0],c[0],d[0],e[0],f[0],g[0],h[0] and w[m][i] and put it under for loop
      //if() ....
       for (int m = 0; m < NUM_OF_NONCES; m++) begin
          for (int n = 0; n < 15; n++) begin
             w[m][n] <= w[m][n+1];
          end
          w[m][15] <= wt_expansion(m);
          {a[m],b[m],c[m],d[m],e[m],f[m],g[m],h[m]} <= sha256_op(a[m],b[m],c[m],d[m],e[m],f[m],g[m],h[m],w[m][0],i);
          end
      //else ...
      // use below mention way for wt_expansion
        i <= i + 1;
       state <= PHASE2_COMPUTE;
     end
     else begin
          // Add below mentioned within for loop with m=0 and m<NUM_OF_NONCES
        for (int m = 0; m < NUM_OF_NONCES; m++) begin
          h0[m] <= h0[m] + a[m];
          h1[m] <= h1[m] + b[m];
          h2[m] <= h2[m] + c[m];
          h3[m] <= h3[m] + d[m];
          h4[m] <= h4[m] + e[m];
          h5[m] <= h5[m] + f[m];
          h6[m] <= h6[m] + g[m];
          h7[m] <= h7[m] + h[m];
        end
        state <= PHASE3_BLOCK;
     end
  end
```

```
PHASE3_BLOCK: begin

   // Within for loop with m=0 and m<NUM_OF_NONCES : Add code to fill in w[m][0] <= h0[m] to h7[m]
  // Fill in w[m][8] = 32'h80000000;
  // w[m][9] to w[m][14] to 0
  // w[m][15] = 32'd256;
  for (int m = 0; m < NUM_OF_NONCES; m++) begin
     w[m][0] <= h0[m];
     w[m][1] <= h1[m];
     w[m][2] <= h2[m];
     w[m][3] <= h3[m];
     w[m][4] <= h4[m];
     w[m][5] <= h5[m];
     w[m][6] <= h6[m];
     w[m][7] <= h7[m];
     w[m][8] <= 32'h80000000;
     for (int temp_idx = 9; temp_idx < 15; temp_idx++) begin
        w[m][temp_idx] <= 0;
     end
     w[m][15] <= 32'd256;

  // Add within forloop code to initiatlize a through h with initial hash constants h0_const to h7_const
  // a[m] <= h0_const;
      a[m] <= h0_const;
     b[m] <= h1_const;
     c[m] <= h2_const;
     d[m] <= h3_const;
     e[m] <= h4_const;
     f[m] <= h5_const;
     g[m] <= h6_const;
     h[m] <= h7_const;
  end
  // h[m] <= h7_const;

  i <= 0;
  state <= PHASE3_COMPUTE;
end
```



```
PHASE3_COMPUTE: begin
  if (i < 64) begin
    // Add similar code as in PHASE2_COMPUTE
     for (int m = 0; m < NUM_OF_NONCES; m++) begin
        for (int n = 0; n < 15; n++) begin
           w[m][n] <= w[m][n+1];
        end
        w[m][15] <= wt_expansion(m);
        {a[m],b[m],c[m],d[m],e[m],f[m],g[m],h[m]} <= sha256_op(a[m],b[m],c[m],d[m],e[m],f[m],g[m],h[m],w[m][0],i);
        end
     i <= i + 1;
     state <= PHASE3_COMPUTE;
    end

  else begin

      for(int m=0; m<NUM_OF_NONCES; m++) begin
        if(itr_idx == 0) begin // This is for nonce 0 to nonce 7
           h0_out[m] <= h0_const + a[m];
           h1_out[m] <= h1_const + b[m];
           h2_out[m] <= h2_const + c[m];
           h3_out[m] <= h3_const + d[m];
           h4_out[m] <= h4_const + e[m];
           h5_out[m] <= h5_const + f[m];
           h6_out[m] <= h6_const + g[m];
           h7_out[m] <= h7_const + h[m];
        end
        else begin  // This is for nonce 8 to nonce 15
           h0_out[m+NUM_OF_NONCES] <= h0_const + a[m];
           h1_out[m+NUM_OF_NONCES] <= h1_const + b[m];
           h2_out[m+NUM_OF_NONCES] <= h2_const + c[m];
           h3_out[m+NUM_OF_NONCES] <= h3_const + d[m];
           h4_out[m+NUM_OF_NONCES] <= h4_const + e[m];
           h5_out[m+NUM_OF_NONCES] <= h5_const + f[m];
           h6_out[m+NUM_OF_NONCES] <= h6_const + g[m];
           h7_out[m+NUM_OF_NONCES] <= h7_const + h[m];
        end
      end

      // If nonce 0 to 7 iteration completed then go ot PHASE2_BLOCK
      // OR go to WRITE FSM state
      if(itr_idx < 1) begin
        itr_idx = itr_idx + 1;
        i <= 0;
        state <= PHASE2_BLOCK;
     end
```



```
        else begin
           i <= 0;
           state <= WRITE;
        end

     end
   end

  WRITE: begin
     if (i <= 15) begin
     // Write h0_out[0], h0_out[1], h0_out[2] to h0_out[15] to testbench memory
     i <= i + 1;
       offset <= i;
       cur_we <= 1'b1;
       cur_addr <= output_addr;
     state <= WRITE;
       case (i)
         0: cur_write_data <= h0_out[0];
         1: cur_write_data <= h0_out[1];
         2: cur_write_data <= h0_out[2];
         3: cur_write_data <= h0_out[3];
         4: cur_write_data <= h0_out[4];
         5: cur_write_data <= h0_out[5];
         6: cur_write_data <= h0_out[6];
         7: cur_write_data <= h0_out[7];
         8: cur_write_data <= h0_out[8];
         9: cur_write_data <= h0_out[9];
         10: cur_write_data <= h0_out[10];
         11: cur_write_data <= h0_out[11];
         12: cur_write_data <= h0_out[12];
         13: cur_write_data <= h0_out[13];
         14: cur_write_data <= h0_out[14];
         15: cur_write_data <= h0_out[15];
       endcase
     end
     else begin
     state <= IDLE;
     end
  end
 endcase

end


assign done = (state == IDLE);

endmodule
```

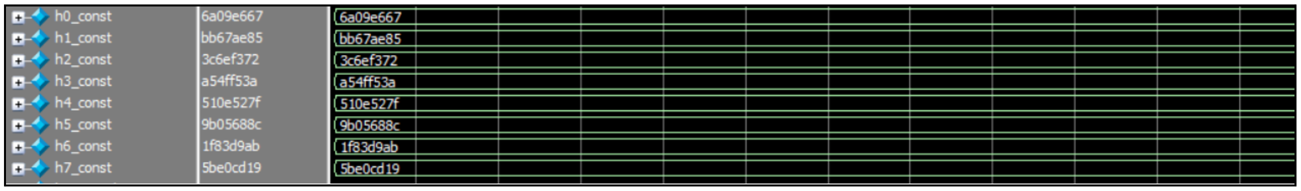
Description:

The bitcoin hashing algorithm takes in 19 words from testbench and reads them to memory (unlike SHA-256, the last word in the bitcoin testbench is the nonce value). We process the memory so that the first block has the first 16 words read to memory, and the second block has the last 3 words, the nonce value, and padding bits. However, in our parallel implementation, we make sure that w[3] is different for every nonce iteration (w[3] for nonce 0 has nonce value 0, while w[3] for nonce 15 has nonce value 15). This is so that we can streamline the computation process as the input for the second phase block for every nonce value are available (derived from the output of phase1 compute). When we go into phase 2 compute, we perform the SHA-256 algorithm for 8 different nonce values at a time. In the for loop iterating over the nonce values, we perform the wt expansion, and then the SHA-256 operation. After iterating over an ith value, we then increment i and cycle back to the start of phase 2 compute. When i = 64, there's another nonce value for loop for updating the values of h0 to h7.

In the phase-3 block, we update the values of w[0] to w[7] for each nonce value with h0 to h7 values according to the respective nonce, with w[8] to w[15] being padding bits. This time the padding bit size is 32'd256 rather than 32'd640 as the output message size is 256 bits. We then update A to H with the hash constants. In phase 3 compute, we perform the same computation in phase 2 compute, except we also check the value of itr_idx. If itr_idx is 1 (we have already done phase3-compute previously) we transition to the write state and end the program. If it is 0 (this is our first time in phase-3 compute), we increment itr_idx and go back to phase 2-block to perform the parallel computation for nonce values 8 to 15.

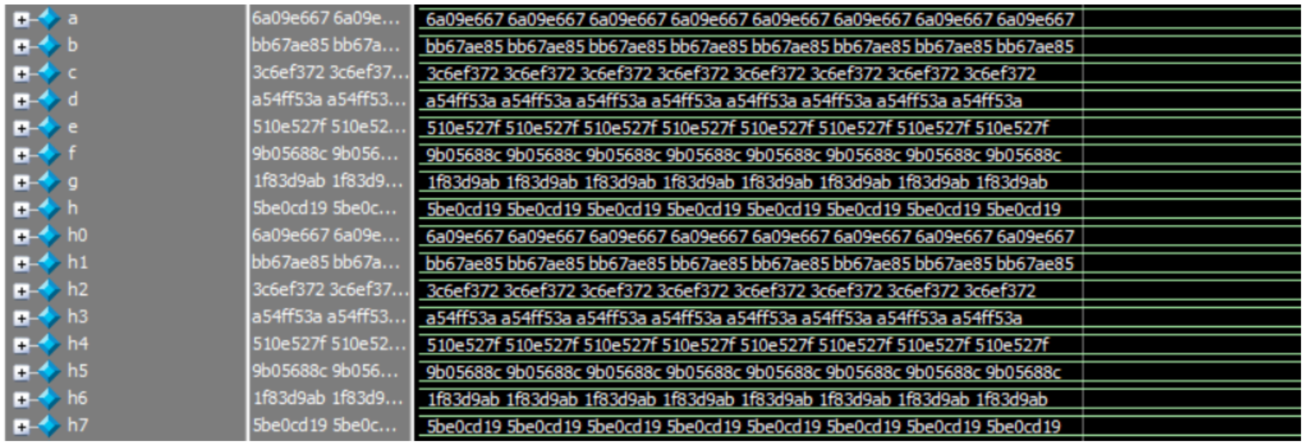
3. ModelSim Simulation Waveform

h0 - h7 constants are correctly displayed



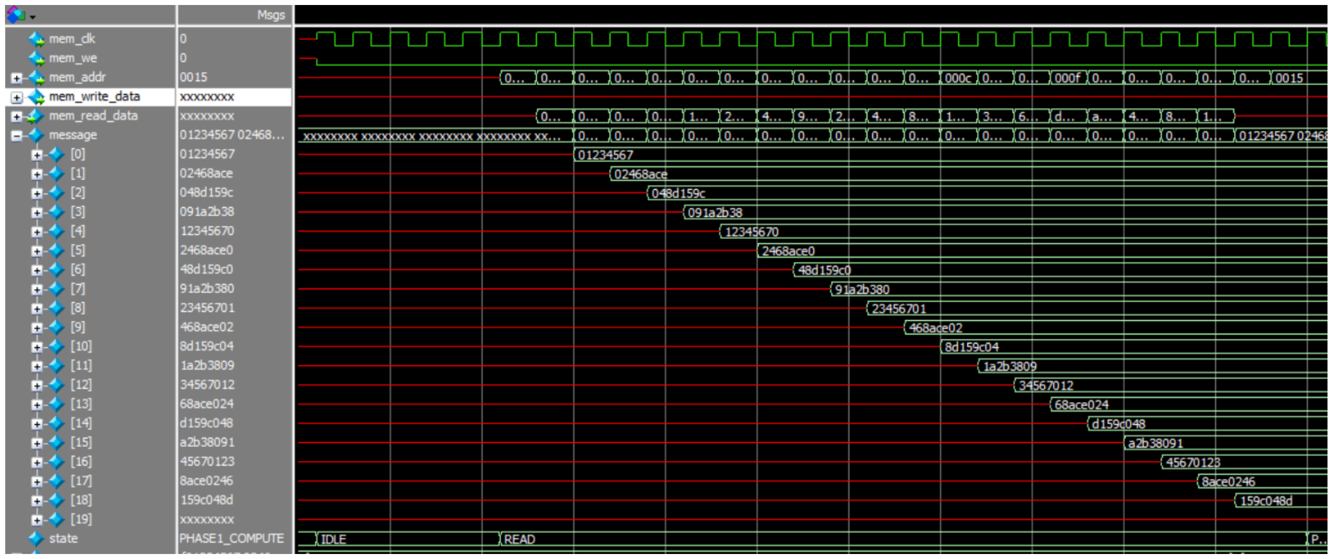
```
h0_const  6a09e667
h1_const  bb67ae85
h2_const  3c6ef372
h3_const  a54ff53a
h4_const  510e527f
h5_const  9b05688c
h6_const  1f83d9ab
h7_const  5be0cd19
```

a - b and h0 - h7 are updated after IDLE state



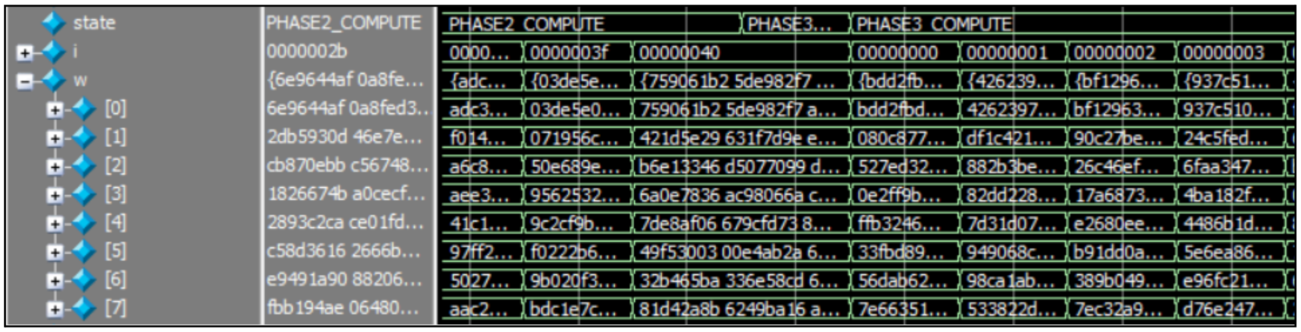
```
a   6a09e667 6a09e...  6a09e667 6a09e667 6a09e667 6a09e667 6a09e667 6a09e667 6a09e667 6a09e667
b   bb67ae85 bb67a...  bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85
c   3c6ef372 3c6ef37...  3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372
d   a54ff53a a54ff53...  a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a
e   510e527f 510e52...  510e527f 510e527f 510e527f 510e527f 510e527f 510e527f 510e527f 510e527f
f   9b05688c 9b056...  9b05688c 9b05688c 9b05688c 9b05688c 9b05688c 9b05688c 9b05688c 9b05688c
g   1f83d9ab 1f83d9...  1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab
h   5be0cd19 5be0c...  5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19
h0  6a09e667 6a09e...  6a09e667 6a09e667 6a09e667 6a09e667 6a09e667 6a09e667 6a09e667 6a09e667
h1  bb67ae85 bb67a...  bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85
h2  3c6ef372 3c6ef37...  3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372
h3  a54ff53a a54ff53...  a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a
h4  510e527f 510e52...  510e527f 510e527f 510e527f 510e527f 510e527f 510e527f 510e527f 510e527f
h5  9b05688c 9b056...  9b05688c 9b05688c 9b05688c 9b05688c 9b05688c 9b05688c 9b05688c 9b05688c
h6  1f83d9ab 1f83d9...  1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab
h7  5be0cd19 5be0c...  5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19
```

During READ, message words are being corrected stored



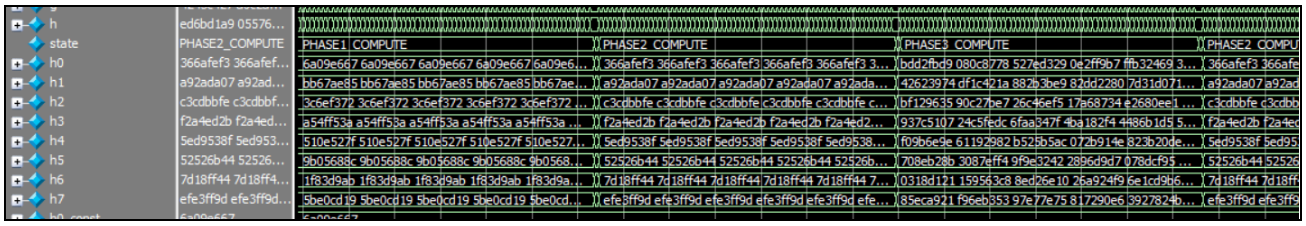
```
                 Msgs
mem_clk          0
mem_we           0
mem_addr         0015
mem_write_data   xxxxxxxx
mem_read_data    xxxxxxxx
message          01234567 02468...
  [0]            01234567
  [1]            02468ace
  [2]            048d159c
  [3]            091a2b38
  [4]            12345670
  [5]            2468ace0
  [6]            48d159c0
  [7]            91a2b380
  [8]            23456701
  [9]            468ace02
  [10]           8d159c04
  [11]           1a2b3809
  [12]           34567012
  [13]           68ace024
  [14]           d159c048
  [15]           a2b38091
  [16]           45670123
  [17]           8ace0246
  [18]           159c048d
  [19]           xxxxxxxx
state            PHASE1_COMPUTE
```

w0 - w7 is properly updating when moving to the next time step, shown when i = 00000040



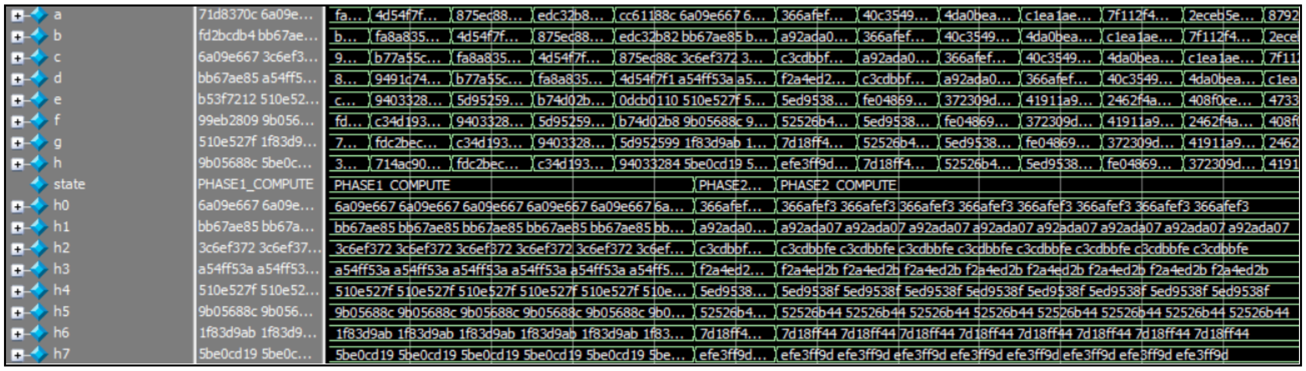
```
state   PHASE2_COMPUTE       PHASE2_COMPUTE | PHASE3... | PHASE3_COMPUTE
i       0000002b             0000... | 0000003f | 00000040 | 00000000 | 00000001 | 00000002 | 00000003
w       {6e9644af 0a8fe...   {adc... | {03de5e... | {759061b2 5de982f7 ... | {bdd2fb... | {426239... | {bf1296... | {937c51...
  [0]   6e9644af 0a8fed3...  adc3... | 03de5e0... | 759061b2 5de982f7 a... | bdd2fbd... | 4262397... | bf12963... | 937c510...
  [1]   2db5930d 46e7e...    f014... | 071956c... | 421d5e29 631f7d9e e... | 080c877... | df1c421... | 90c27be... | 24c5fed...
  [2]   cb870ebb c56748...   a6c8... | 50e689e... | b6e13346 d5077099 d... | 527ed32... | 882b3be... | 26c46ef... | 6faa347...
  [3]   1826674b a0cecf...   aee3... | 9562532... | 6a0e7836 ac98066a c... | 0e2ff9b... | 82dd228... | 17a6873... | 4ba182f...
  [4]   2893c2ca ce01fd...   41c1... | 9c2cf9b... | 7de8af06 679cfd73 8... | ffb3246... | 7d31d07... | e2680ee... | 4486b1d...
  [5]   c58d3616 2666b...    97ff2... | f0222b6... | 49f53003 00e4ab2a 6... | 33fbd89... | 949068c... | b91dd0a... | 5e6ea86...
  [6]   e9491a90 88206...    5027... | 9b020f3... | 32b465ba 336e58cd 6... | 56dab62... | 98ca1ab... | 389b049... | e96fc21...
  [7]   fbb194ae 06480...    aac2... | bdc1e7c... | 81d42a8b 6249ba16 a... | 7e66351... | 533822d... | 7ec32a9... | d76e247...
```

During each PHASE, h0 - h7 is being corrected updated



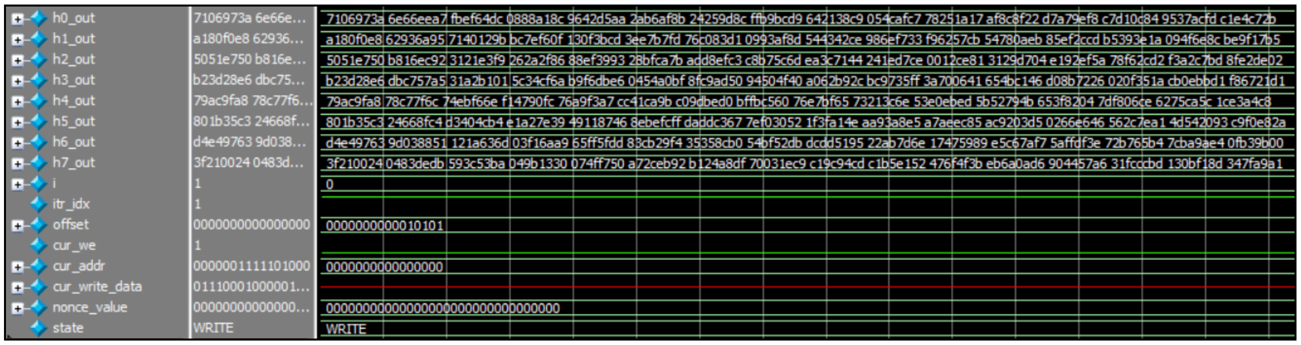
```
h       ed6bd1a9 05576...
state   PHASE2_COMPUTE     PHASE1_COMPUTE | PHASE2_COMPUTE | PHASE3_COMPUTE | PHASE2_COMPU
h0      366afef3 366afef...  6a09e667 6a09e667 6a09e667 6a09e667 6a09e6... | 366afef3 366afef3 366afef3 366afef3 366afef3 3... | bdd2fbd9 080c8778 527ed329 0e2ff9b7 ffb32469 3... | 366afef3 366afe
h1      a92ada07 a92ad...    bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae... | a92ada07 a92ada07 a92ada07 a92ada07 a92ada... | 42623974 df1c421a 882b3be9 82dd2280 7d31d071... | a92ada07 a92ad
h2      c3cdbbfe c3cdbbf...  3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372 ... | c3cdbbfe c3cdbbfe c3cdbbfe c3cdbbfe c3cdbbfe c... | bf129635 90c27be7 26c46ef5 17a68734 e2680ee1 ... | c3cdbbfe c3cdbb
h3      f2a4ed2b f2a4ed...   a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a ... | f2a4ed2b f2a4ed2b f2a4ed2b f2a4ed2b f2a4ed2... | 937c5107 24c5fedc 6faa347f 4ba182f4 4486b1d5 5... | f2a4ed2b f2a4ed
h4      5ed9538f 5ed953...   510e527f 510e527f 510e527f 510e527f 510e527... | 5ed9538f 5ed9538f 5ed9538f 5ed9538f 5ed9538... | f09b6e9e 61192982 b525b5ac 072b914e 823b20de... | 5ed9538f 5ed95
h5      52526b44 52526...    9b05688c 9b05688c 9b05688c 9b05688c 9b0568... | 52526b44 52526b44 52526b44 52526b44 52526b... | 708eb28b 3087eff4 9f9e3242 2896d9d7 078dcf95 ... | 52526b44 52526
h6      7d18ff44 7d18ff4...  1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9a... | 7d18ff44 7d18ff44 7d18ff44 7d18ff44 7d18ff44 7... | 0318d121 159563c8 8ed26e10 26a924f9 6e1cd9b6... | 7d18ff44 7d18ff
h7      efe3ff9d efe3ff9d... 5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd... | efe3ff9d efe3ff9d efe3ff9d efe3ff9d efe3ff9d efe... | 85eca921 f96eb353 97e77e75 817290e6 3927824b... | efe3ff9d efe3ff9
```

… a - h values are also working and updating as intended



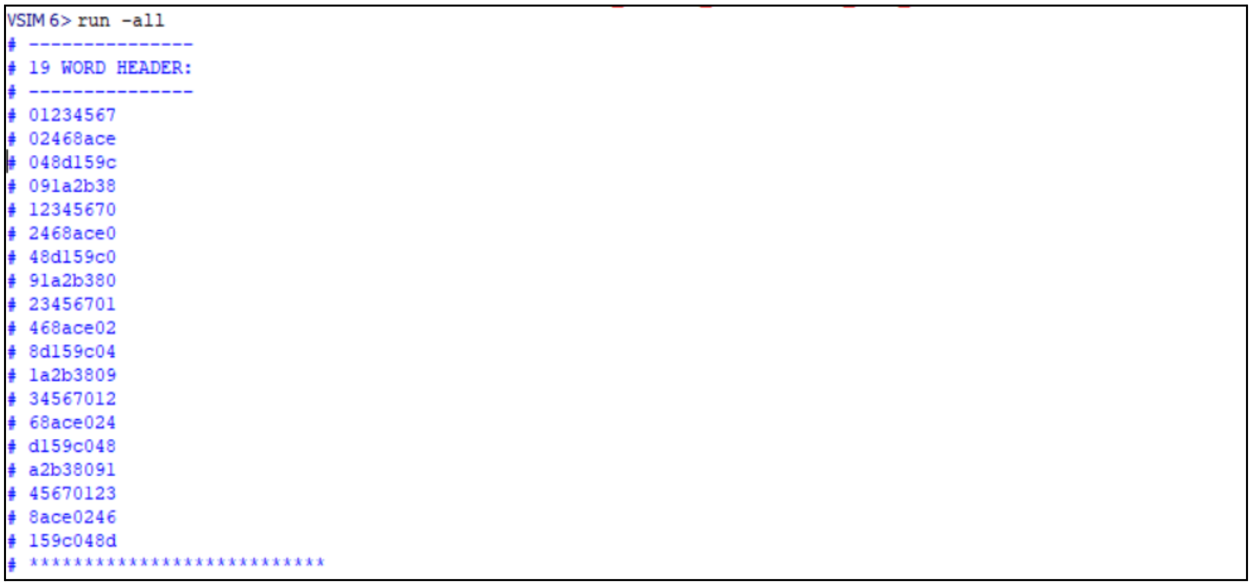
```
a       71d8370c 6a09e...    fa... | 4d54f7f... | 875ec88... | edc32b8... | cc61188c 6a09e667 6... | 366afef... | 40c3549... | 4da0bea... | c1ea1ae... | 7f112f4... | 2eceb5e... | 8792
b       fd2bcdb4 bb67ae...   b... | fa8a835... | 4d54f7f... | 875ec88... | edc32b82 bb67ae85 b... | a92ada0... | 366afef... | 40c3549... | 4da0bea... | c1ea1ae... | 7f112f4... | 2ece
c       6a09e667 3c6ef3...   9... | b77a55c... | fa8a835... | 4d54f7f... | 875ec88c 3c6ef372 3... | c3cdbbf... | a92ada0... | 366afef... | 40c3549... | 4da0bea... | c1ea1ae... | 7f11
d       bb67ae85 a54ff5...   8... | 9491c74... | b77a55c... | fa8a835... | 4d54f7f1 a54ff53a a5... | f2a4ed2... | c3cdbbf... | a92ada0... | 366afef... | 40c3549... | 4da0bea... | c1ea
e       b53f7212 510e52...   c... | 9403328... | 5d95259... | b74d02b... | 0dcb0110 510e527f 5... | 5ed9538... | fe04869... | 372309d... | 41911a9... | 2462f4a... | 408f0ce... | 4733
f       99eb2809 9b056...    fd... | c34d193... | 9403328... | 5d95259... | b74d02b8 9b05688c 9... | 52526b4... | 5ed9538... | fe04869... | 372309d... | 41911a9... | 2462f4a... | 408f
g       510e527f 1f83d9...   7... | fdc2bec... | c34d193... | 9403328... | 5d952599 1f83d9ab 1... | 7d18ff4... | 52526b4... | 5ed9538... | fe04869... | 372309d... | 41911a9... | 2462
h       9b05688c 5be0c...    3... | 714ac90... | fdc2bec... | c34d193... | 94033284 5be0cd19 5... | efe3ff9d... | 7d18ff4... | 52526b4... | 5ed9538... | fe04869... | 372309d... | 4191
state   PHASE1_COMPUTE       PHASE1_COMPUTE | PHASE2... | PHASE2_COMPUTE
h0      6a09e667 6a09e...    6a09e667 6a09e667 6a09e667 6a09e667 6a09e667 6a... | 366afef... | 366afef3 366afef3 366afef3 366afef3 366afef3 366afef3 366afef3 366afef3
h1      bb67ae85 bb67a...    bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb67ae85 bb... | a92ada0... | a92ada07 a92ada07 a92ada07 a92ada07 a92ada07 a92ada07 a92ada07 a92ada07
h2      3c6ef372 3c6ef37...  3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef372 3c6ef... | c3cdbbf... | c3cdbbfe c3cdbbfe c3cdbbfe c3cdbbfe c3cdbbfe c3cdbbfe c3cdbbfe c3cdbbfe
h3      a54ff53a a54ff53...  a54ff53a a54ff53a a54ff53a a54ff53a a54ff53a a54ff5... | f2a4ed2... | f2a4ed2b f2a4ed2b f2a4ed2b f2a4ed2b f2a4ed2b f2a4ed2b f2a4ed2b f2a4ed2b
h4      510e527f 510e52...   510e527f 510e527f 510e527f 510e527f 510e527f 510e... | 5ed9538... | 5ed9538f 5ed9538f 5ed9538f 5ed9538f 5ed9538f 5ed9538f 5ed9538f 5ed9538f
h5      9b05688c 9b056...    9b05688c 9b05688c 9b05688c 9b05688c 9b05688c 9b0... | 52526b4... | 52526b44 52526b44 52526b44 52526b44 52526b44 52526b44 52526b44 52526b44
h6      1f83d9ab 1f83d9...   1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83d9ab 1f83... | 7d18ff4... | 7d18ff44 7d18ff44 7d18ff44 7d18ff44 7d18ff44 7d18ff44 7d18ff44 7d18ff44
h7      5be0cd19 5be0c...    5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be0cd19 5be... | efe3ff9d... | efe3ff9d efe3ff9d efe3ff9d efe3ff9d efe3ff9d efe3ff9d efe3ff9d efe3ff9d
```

Finally, correct hashes are stored in memory during WRITE

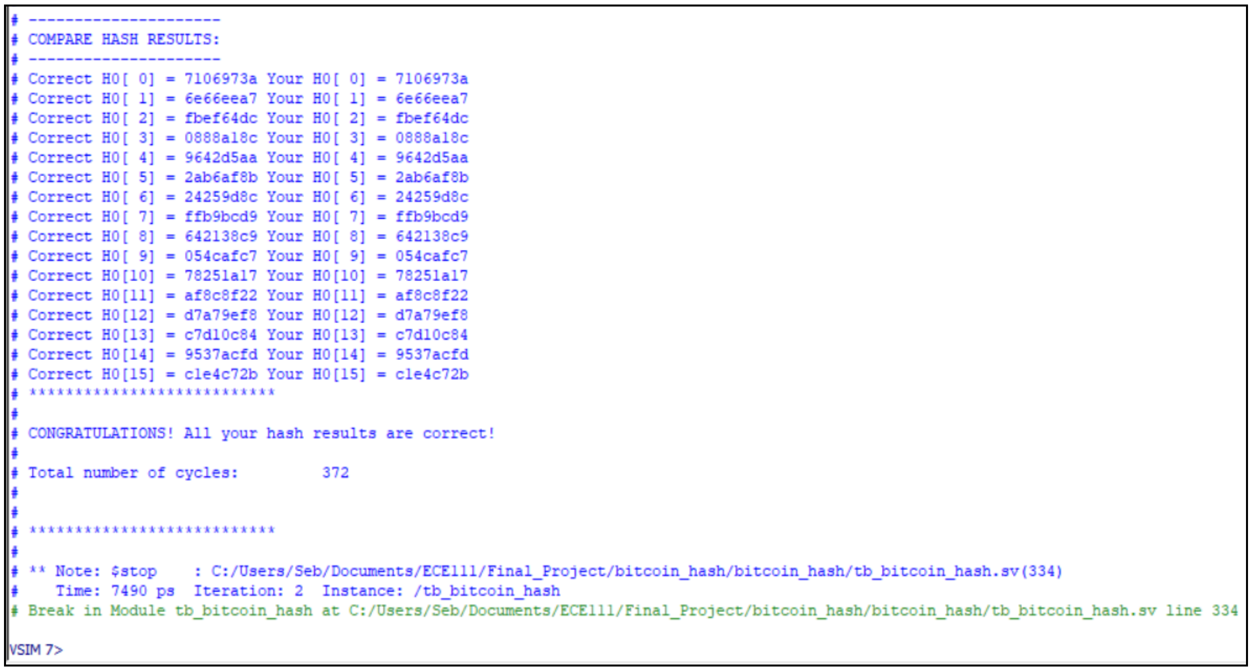


```
h0_out          7106973a 6e66e...    7106973a 6e66eea7 fbef64dc 0888a18c 9642d5aa 2ab6af8b 24259d8c ffb9bcd9 642138c9 054cafc7 78251a17 af8c8f22 d7a79ef8 c7d10c84 9537acfd c1e4c72b
h1_out          a180f0e8 62936...    a180f0e8 62936a95 7140129b bc7ef60f 130f3bcd 3ee7b7fd 76c083d1 0993af8d 544342ce 986ef733 f96257cb 54780aeb 85ef2ccd b5393e1a 094f6e8c be9f17b5
h2_out          5051e750 b816e...    5051e750 b816ec92 3121e3f9 262a2f86 88ef3993 28bfca7b add8efc3 c8b75c6d ea3c7144 241ed7ce 0012ce81 3129d704 e192ef5a 78f62cd2 f3a2c7bd 8fe2de02
h3_out          b23d28e6 dbc75...    b23d28e6 dbc757a5 31a2b101 5c34cf6a b9f6dbe6 0454a0bf 8fc9ad50 94504f40 a062b92c bc9735ff 3a700641 654bc146 d08b7226 020f351a cb0ebbd1 f86721d1
h4_out          79ac9fa8 78c77f6...  79ac9fa8 78c77f6c 74ebf66e f14790fc 76a9f3a7 cc41ca9b c09dbed0 bffbc560 76e7bf65 73213c6e 53e0ebed 5b52794b 653f8204 7df806ce 6275ca5c 1ce3a4c8
h5_out          801b35c3 24668f...   801b35c3 24668fc4 d3404cb4 e1a27e39 49118746 8ebefcff daddc367 7ef03052 1f3fa14e aa93a8e5 a7aeec85 ac9203d5 0266e646 562c7ea1 4d542093 c9f0e82a
h6_out          d4e49763 9d038...    d4e49763 9d038851 121a636d 03f16aa9 65ff5fdd 83cb29f4 35358cb0 54bf52db dcdd5195 22ab7d6e 17475989 e5c67af7 5affdf3e 72b765b4 7cba9ae4 0fb39b00
h7_out          3f210024 0483d...    3f210024 0483dedb 593c53ba 049b1330 074ff750 a72ceb92 b124a8df 70031ec9 c19c94cd c1b5e152 476f4f3b eb6a0ad6 904457a6 31fccbd 130bf18d 347fa9a1
i               1                    0
itr_idx         1
offset          0000000000000000     0000000000010101
cur_we          1
cur_addr        0000001111101000     0000000000000000
cur_write_data  01110001000001...
nonce_value     00000000000000...    00000000000000000000000000000000
state           WRITE                WRITE
```

4. ModelSim Simulation Output transcript



```
VSIM 6> run -all
# ---------------
# 19 WORD HEADER:
# ---------------
# 01234567
# 02468ace
# 048d159c
# 091a2b38
# 12345670
# 2468ace0
# 48d159c0
# 91a2b380
# 23456701
# 468ace02
# 8d159c04
# 1a2b3809
# 34567012
# 68ace024
# d159c048
# a2b38091
# 45670123
# 8ace0246
# 159c048d
# ***************************
```

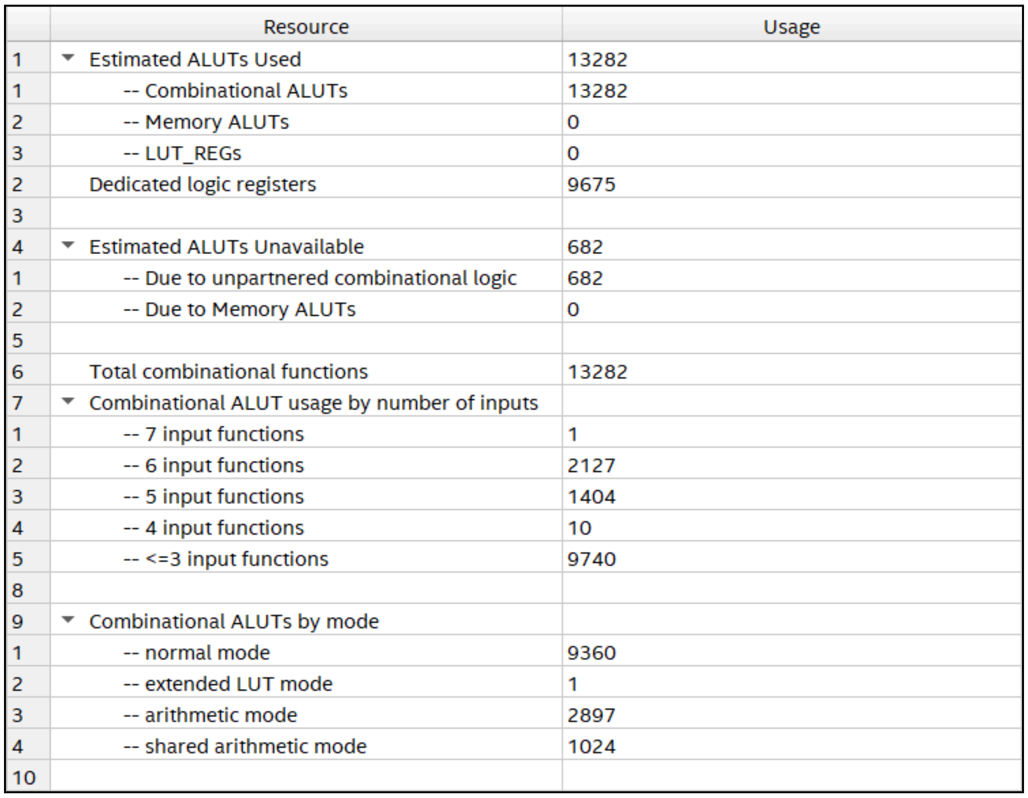


```
# --------------------
# COMPARE HASH RESULTS:
# --------------------
# Correct H0[ 0] = 7106973a Your H0[ 0] = 7106973a
# Correct H0[ 1] = 6e66eea7 Your H0[ 1] = 6e66eea7
# Correct H0[ 2] = fbef64dc Your H0[ 2] = fbef64dc
# Correct H0[ 3] = 0888a18c Your H0[ 3] = 0888a18c
# Correct H0[ 4] = 9642d5aa Your H0[ 4] = 9642d5aa
# Correct H0[ 5] = 2ab6af8b Your H0[ 5] = 2ab6af8b
# Correct H0[ 6] = 24259d8c Your H0[ 6] = 24259d8c
# Correct H0[ 7] = ffb9bcd9 Your H0[ 7] = ffb9bcd9
# Correct H0[ 8] = 642138c9 Your H0[ 8] = 642138c9
# Correct H0[ 9] = 054cafc7 Your H0[ 9] = 054cafc7
# Correct H0[10] = 78251a17 Your H0[10] = 78251a17
# Correct H0[11] = af8c8f22 Your H0[11] = af8c8f22
# Correct H0[12] = d7a79ef8 Your H0[12] = d7a79ef8
# Correct H0[13] = c7d10c84 Your H0[13] = c7d10c84
# Correct H0[14] = 9537acfd Your H0[14] = 9537acfd
# Correct H0[15] = c1e4c72b Your H0[15] = c1e4c72b
# ***************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:         372
#
#
# ***************************
#
# ** Note: $stop    : C:/Users/Seb/Documents/ECE111/Final_Project/bitcoin_hash/bitcoin_hash/tb_bitcoin_hash.sv(334)
#    Time: 7490 ps  Iteration: 2  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at C:/Users/Seb/Documents/ECE111/Final_Project/bitcoin_hash/bitcoin_hash/tb_bitcoin_hash.sv line 334

VSIM 7>
```

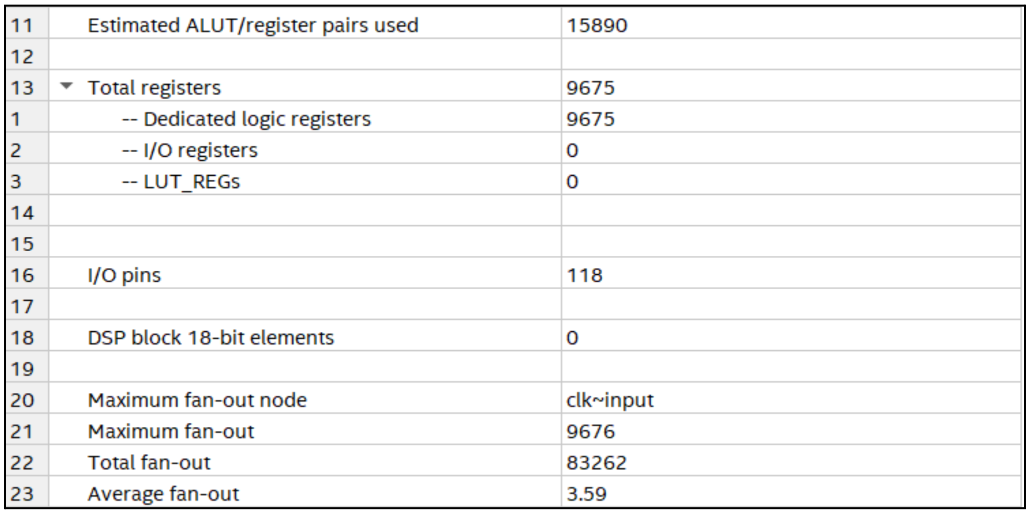
As shown in the final waveform screenshot and the simulation transcript, the final h0_out is correctly written to memory. Therefore, all internal processes and assignments must be functioning properly, and our implementation functions as a bitcoin hash. We were able to complete this operation in 372 cycles. This is highly optimized considering that our SHA-256 algorithm completes its tasks in 168 cycles.

5. Resource Usage and timing reports

Resource Usage:

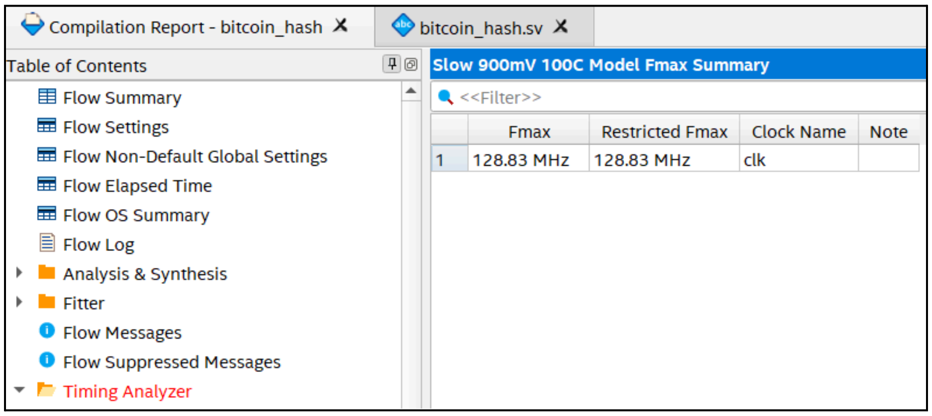


	Resource	Usage
1	Estimated ALUTs Used	13282
1	-- Combinational ALUTs	13282
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	9675
3		
4	Estimated ALUTs Unavailable	682
1	-- Due to unpartnered combinational logic	682
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	13282
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	1
2	-- 6 input functions	2127
3	-- 5 input functions	1404
4	-- 4 input functions	10
5	-- <=3 input functions	9740
8		
9	Combinational ALUTs by mode	
1	-- normal mode	9360
2	-- extended LUT mode	1
3	-- arithmetic mode	2897
4	-- shared arithmetic mode	1024
10		



11	Estimated ALUT/register pairs used	15890
12		
13	Total registers	9675
1	-- Dedicated logic registers	9675
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	118
17		
18	DSP block 18-bit elements	0
19		
20	Maximum fan-out node	clk~input
21	Maximum fan-out	9676
22	Total fan-out	83262
23	Average fan-out	3.59

Timing Report for FMAX:



Compilation Report - bitcoin_hash | bitcoin_hash.sv

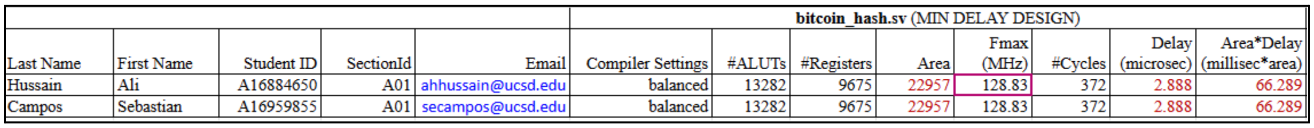
Table of Contents
- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Flow Messages
- Flow Suppressed Messages
- Timing Analyzer

Slow 900mV 100C Model Fmax Summary

<<Filter>>

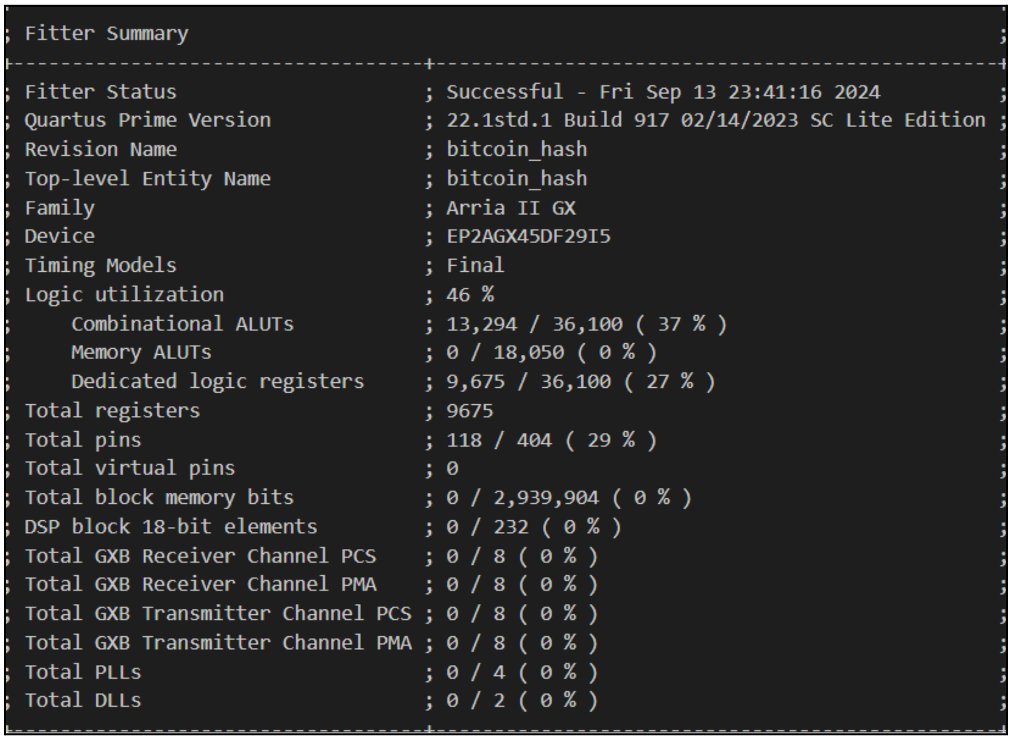
	Fmax	Restricted Fmax	Clock Name	Note
1	128.83 MHz	128.83 MHz	clk	

Area:



					bitcoin_hash.sv (MIN DELAY DESIGN)							
Last Name	First Name	Student ID	SectionId	Email	Compiler Settings	#ALUTs	#Registers	Area	Fmax (MHz)	#Cycles	Delay (microsec)	Area*Delay (millisec*area)
Hussain	Ali	A16884650	A01	ahhussain@ucsd.edu	balanced	13282	9675	22957	128.83	372	2.888	66.289
Campos	Sebastian	A16959855	A01	secampos@ucsd.edu	balanced	13282	9675	22957	128.83	372	2.888	66.289

Fitter Report:



```
; Fitter Summary                                                                    ;
+-----------------------------------+-----------------------------------------------+
; Fitter Status                     ; Successful - Fri Sep 13 23:41:16 2024         ;
; Quartus Prime Version             ; 22.1std.1 Build 917 02/14/2023 SC Lite Edition ;
; Revision Name                     ; bitcoin_hash                                  ;
; Top-level Entity Name             ; bitcoin_hash                                  ;
; Family                            ; Arria II GX                                   ;
; Device                            ; EP2AGX45DF29I5                                ;
; Timing Models                     ; Final                                         ;
; Logic utilization                 ; 46 %                                          ;
;     Combinational ALUTs           ; 13,294 / 36,100 ( 37 % )                      ;
;     Memory ALUTs                  ; 0 / 18,050 ( 0 % )                            ;
;     Dedicated logic registers     ; 9,675 / 36,100 ( 27 % )                       ;
; Total registers                   ; 9675                                          ;
; Total pins                        ; 118 / 404 ( 29 % )                            ;
; Total virtual pins                ; 0                                             ;
; Total block memory bits           ; 0 / 2,939,904 ( 0 % )                         ;
; DSP block 18-bit elements         ; 0 / 232 ( 0 % )                               ;
; Total GXB Receiver Channel PCS    ; 0 / 8 ( 0 % )                                 ;
; Total GXB Receiver Channel PMA    ; 0 / 8 ( 0 % )                                 ;
; Total GXB Transmitter Channel PCS ; 0 / 8 ( 0 % )                                 ;
; Total GXB Transmitter Channel PMA ; 0 / 8 ( 0 % )                                 ;
; Total PLLs                        ; 0 / 4 ( 0 % )                                 ;
; Total DLLs                        ; 0 / 2 ( 0 % )                                 ;
+-----------------------------------+-----------------------------------------------+
```