ECE 111: Advanced Digital Design Project
Prof. Yatish Turakhia

SHA-256 and Bitcoin Hashing Report

June 10, 2023

Conner Hsu (A16665092)
Haozhang Chu (A16484292)
Kirtan Shah (A16227067)

Contents

1 Simplified SHA-256

1.1 Introduction

Hash functions are powerful tools that play a vital role in various areas of computer science and information security. At their core, hash functions are mathematical algorithms that transform data of any size into a fixed-size string of characters, known as a hash value or hash code. This transformation is designed to have a few important properties:

- compression; the output hash value is fixed in size regardless of the size of the input.
- avalanche effect: a small change in the input results in a huge change in the output.
- determinism: the same input must always generate the same output.
- pre-image resistant: an inverse hash function should not exist and it should be very difficult to determine the input that generated a given output.
- collision resistance: the hash function should be nearly completely injective and thus should almost never have two inputs map to the same output.

For cryptographic applications, avalanche effect, noninvertibility and injectiveness are very important properties. These properties all together make hash functions invaluable for tasks such as data retrieval, data integrity verification, password storage, and digital signatures. By producing unique and irreversible hash values, hash functions enable efficient data indexing, quick data comparison, and secure authentication. Whether it's ensuring data integrity, enhancing search performance, or providing robust security measures, hash functions are essential components in modern computing systems.

In this report, we explore the implementation of a particular hash function called Secure Hashing Algorithm - 256 or SHA-256 for short. This hash function takes in any length input and outputs a 256-bit digest/hash value. We will build this using System Verilog and use the Arria-II FPGA to realize the implementation on hardware.

1.2 Algorithm

Before running the SHA-256 algorithm, the input message must be broken up into blocks of 512 bits. The final block must contain at least 66 extra bits; a 1 followed by at least one 0, and 64 bits equal to the size of the original input message. The number of zeros padding between the first 1 and the message size bits depends on how many bits are needed to make the last block 512 bits long.

For example, suppose the message is 640 bits long. This means that the first block will contain the first 512 bits of the message, and the second block will contain the last 128 bits of the message. Next, a 1 will be appended to the last block making it 129 bits long. Then, 319 zeros will be appended to make the last block now 448 bits long. Last, 64 bits will be appended to store the message size and the last block will not be 512 bits long.

As another example, suppose that the input message divides evenly into 512-bit blocks meaning that the message size is some multiple of 512. In this case, the final block will have a 1 in the beginning, then 448 zeros, then end off 64 bits for the message size. The number of zeros is increased to 448 to ensure that this final block is 512 bits long.

Throughout the SHA-256 algorithm, a set of hash values $H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7$ are updated after each block in the input message is processed. These variables are each 32 bits long and thus when put together form the 256-bit output digest.

From here, the states of the SHA-256 are described as follows. The algorithm begins in the `IDLE` state. In this state, the algorithm waits for a start signal to begin computation. Once a start signal is received, it initializes iterative variables, the address to read in the input message, and the hash values, H_i for $0 \leq i \leq 7$. Once this is done, the algorithm will move into the `READ` state in the next clock cycle.

In the `READ` state, the SHA-256 algorithm will read data from the memory read in input of the module. Since the memory module being used can only read out 32 bits (or one word) at a time, it takes multiple clock cycles before the entire input message will be read in. In our implementation, we designed the SHA-256

to take a fixed input message length of 640 bits. This means that the `READ` state will use at least 20 clock cycles to read in $20 \cdot 32$ bits. Once the entire input message is read in, the algorithm moves to the `BLOCK` state in the next clock cycle.

In the `BLOCK` state, the algorithm will separate the input message into 512-bit blocks in the same process as described earlier. For the 640-bit input message, 512 of the bits will be placed in the first block, and the last 128 bits will be placed in the second block. Some logic is needed to fill the remaining 384 bits of the second block with the zeros padding and input message size. A for-loop is used to initialize the first 10 words with $100\ldots00$ and then the last two words are set to the input message size. Additionally, before moving into the next state, another set of variables, a, b, c, d, e, f, g, h are initialized to the hash values from earlier. Once this is completed, the algorithm moves to the `COMPUTE` state in the next clock cycle. When the algorithm is in the `BLOCK` state for the first time, it will send the first 512 bits of the input message to the `COMPUTE` state. When the algorithm is in the `BLOCK` state of the second time, it will send the last 128 bits of the input message plus the padding and message size to the `COMPUTE` state.

In the `COMPUTE` state, the algorithm processes the 512-bit block. The algorithm will break these 512 bits into 16 words and then use a process called word expansion to expand this into 64 words. Each of these blocks of 32 bits are processed sequentially over 64 clock cycles. A 32-bit block along with the a, b, c, d, e, f, g, h values are passed to a SHA-256 hash round function each clock cycle.

```
// 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);
    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
```

While it is difficult to explain everything that is going on in the function, some important properties should be noted.

- The AND operation is a noninvertible function if the function is not outputting 1. It is not possible to determine the inputs if the output of an AND operation is 0.
- The XOR operation is always noninvertible.
- While the function is complicated it can be realized that there is no nondeterministic processes in this function.

These properties help SHA-256 to realize properties such as determinism and pre-image resistance. After the SHA-256 hash round is complete, the output of the function will be assigned to a, b, c, d, e, f, g, h and will be used in the next clock cycle. The process of updating the a, b, c, d, e, f, g, h values for each word and then using the same values for the next 32-bit block helps to realize the avalanche effect. A small change in the input message will affect the output of one word, which then affects the input/output of the next 32-bit block, and will then change the inputs/outputs for all of the 32-bit blocks down the line.

Once all 64 words blocks are processed in the `COMPUTE` state, the algorithm will move back to the `BLOCK` state. If there are more blocks of 512 bits to be processed, then it will take the next block and move back to the `COMPUTE` state. However, if there are no more blocks to be processed, then the algorithm will move to the `WRITE` state.

In the `WRITE` state, the algorithm will write the hash values, $H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7$ to memory. Since the memory module being used can only write 32 bits to memory per clock cycle, it will take at least 8 clock cycles to write the hash values to memory.

1.3 Optimization

As we will see in the next section, optimizing the SHA-256 algorithm for speed is very important. In this section, we will briefly discuss some of the changes that helped improve the maximum speed we can clock the algorithm at.

We optimized the `simplified_sha256` by changing the way we assign message words and perform word expansion. First, instead of using an if statement and assigning one word per cycle to `w`, we used a for-loop because it can be done in parallel. This saved about 15 cycles.

```
                    for(int t = 0; t < 16; t = t + 1) begin
                            if (t+16*block_idx < NUM_OF_WORDS)
                                w[t] <= message[t+16*block_idx];
                            else if (t+16*block_idx < num_blocks*16) begin //add
                                ↪ buffer
                                if (t+16*block_idx == NUM_OF_WORDS)
                                    w[t] <= 32'h80000000;
                                else if (t+16*block_idx < num_blocks*16-2)
                                    w[t] <= 32'h00000000;
                                else if (t+16*block_idx == num_blocks*16-2)
                                    w[t] <= message_size[63:32];
                                else
                                    w[t] <= message_size[31:0];
                            end
                    end
```

We also computed the word expansion in parallel with the `COMPUTE` state, which allowed us to reduce the `w` register from length 64 to length 16. This saved thousands of ALUTS.

```
            COMPUTE: begin
                // 64 processing rounds steps for 512-bit block
                if (i < 65) begin
                    if (i < 16) wt <= w[i];
                    else begin
                        wt <= wtnew();
                        for(int n = 0; n < 15; n++) w[n] <= w[n+1];
                        w[15] <= wtnew();
                    end
                    {a, b, c, d, e, f, g, h} <= sha256_op(a, b, c, d, e, f, g, h,
                        ↪ wt, tstep);
                    i <= i + 1;
                end
```

2 Bitcoin Hashing

2.1 Introduction

Bitcoin hashing is a critical aspect of the cryptocurrency's underlying technology, blockchain. In the context of Bitcoin, hashing refers to the process of converting transaction data into a fixed-length alphanumeric string, i.e. a hash value. In order for a block to be accepted as part of the block chain, its hash must be meet a specific criteria. Bitcoin miners must repeatedly hash different combinations of the original transaction data plus a random string of numbers called the "nonce". By changing the nonce, different hash values can be generated and Bitcoin miners will keep trying different nonces until the hash meets the specific criteria. The process of finding the nonce that meets the criteria requires a lot of computational power and since miners compete with each other to see who meets the criteria, speed is also very essential.

In this part of the report, we will investigate the implementation of a Bitcoin hashing algorithm that will attempt to generate multiple hash values at a time for different nonce values. We are not trying to meet any specific criteria for the final hash value, but instead are trying to design a module that will compute many hash values with different nonces very quickly.

2.2 Algorithm

The goal of this algorithm is to compute 16 hash values for 16 different nonce values. Since speed is highly important in the context of Bitcoin mining, we have designed our implementation to computes these hashes as fast as possible.

This algorithm is designed to take in a 640-bit input message (read in from an external memory module similar to the SHA-256. The first 608 bits are reserved for the transaction data and the last 32 bits of the input message are reserved for the nonce. The hash function for the Bitcoin hashing is realized with the SHA-256 function from earlier. The first block of 512 bits contains the first 16 words of the transaction data. The second block contains the last 3 words of the transaction data, the word containing the nonce, and then padding and input message size. These two blocks are then processed by the SHA-256 algorithm again.

We have designed one module to process the first 512-bit block that only contains transaction data. The output hash of this module then helps to initialize the initial hash values of the SHA-256 module that processes the second block. Since we want to compute the hash values for inputs with 16 different nonce values, we ended up using 16 SHA-256 instances running in parallel to compute all 16 nonce values at once.

After output hashes are computed for both blocks of all 16 nonce values, these output hashes are then used to form a third 512-bit block for each nonce value. The first 256 bits of this block contains the hash value of the respective hash output from the first and second blocks. The last 256 bits contain padding and input message size (256). This third block is then fed into one last SHA-256 `COMPUTE` state and uses the same initial hash as the one used to process the first 512-bit block.

To implement this algorithm we used multiple modules. We created a module called `onephase_sha256.sv` that implements a SHA-256 algorithm that only goes through one `COMPUTE` state, and another module called `twophase_sha256.sv` that implements a SHA-256 algorithm that goes through the `COMPUTE` state twice. Finally, a higher level module called `bitcoin_hash.sv` is created to instantiate 1 one phase SHA-256 and 16 two phase SHA-256. Wires are used to connect the output hash of the one phase SHA-256 instance to the 16 instances of the two phase SHA-256.

The top-level module, `bitcoin_hash.sv` handles the reading in and writing out data. This makes it so the read and write states of the one and two phase SHA-256 modules aren't necessarily required.

2.3 Optimization

As seen in the previous section, our proposed algorithm for Bitcoin hashing is fast since it computes all 16 hash value in parallel. Theoretically speaking, this requires the roughly the same computation time as a SHA-256 module that goes through 3 `COMPUTE` states, i.e. processes 3 blocks. However, our algorithm requires implementing 17 SHA-256 instances onto the FPGA, 1 for processing the first 512-bit block and 16 for processing the second and third blocks for each nonce value. Practically speaking, the Arria-II FPGA has a difficult time fitting all 17 SHA-256 instances, so we optimized the ALUT usage.

Since we need to feed the values for `h0-h7` from phase 1, we added a parameter `inh`. The input message

and outputs were passed as registers as well.

```
module twophase_sha256 (
    input logic  clk, reset_n, start,
    input logic[31:0] inh[8],
    input logic[31:0] message[4],
    output logic[31:0] outs[8],
    output logic done);
```

We were able to remove the h0-h7 registers since a-h and inh hold the final values of phase 2. This saved several thousand ALUTs.

```
            BLOCK2: begin
                w[0] <= a + inh[0];
                w[1] <= b + inh[1];
                w[2] <= c + inh[2];
                w[3] <= d + inh[3];
                w[4] <= e + inh[4];
                w[5] <= f + inh[5];
                w[6] <= g + inh[6];
                w[7] <= h + inh[7];
```

We also removed the BLOCK1 state and moved it to the IDLE state because we can hard code the initial behavior. This eliminates the need for an if statement to decide which data to populate depending on which block we are in. This saves several thousand ALUTs. This also had the added benefit of saving a cycle.

```
            IDLE: begin
                if(start) begin
                    block_idx <= 0;
                    // BLOCK1
                    for(int t = 0; t < 16; t = t + 1) begin
                        if(t < 4) w[t] <= message[t];
                        else if(t == 4) w[t] <= 32'h80000000;
                        else if(t < 15) w[t] <= 32'h00000000;
                        else w[t] <= 32'd640;
                    end
                    a <= inh[0];
                    b <= inh[1];
                    c <= inh[2];
                    d <= inh[3];
                    e <= inh[4];
                    f <= inh[5];
                    g <= inh[6];
                    h <= inh[7];
                    wt <= message[0]; // wt = w[0]
                    i <= 1;
                    state <= COMPUTE;
                end
            end
```

Since we are dealing with registers only, we can go directly from COMPUTE to DONE and output the final hash values. This saved a cycle.

```
            DONE: begin
                outs = '{a + 32'h6a09e667, b + 32'h6a09e667, c + 32'h3c6ef372, d +
                    ↪  32'ha54ff53a,
                    e + 32'h510e527f, f + 32'h9b05688c, g + 32'h1f83d9ab, h + 32'
                        ↪ h5be0cd19};
                state <= DONE;
            end
```

2.4 Resource Usage

Resource	Usage
Estimated ALUTs Used	21074
-- Combinational ALUTs	21074
-- Memory ALUTs	0
-- LUT_REGs	0
Dedicated logic registers	15361
Estimated ALUTs Unavailable	2908
-- Due to unpartnered combinational logic	2908
-- Due to Memory ALUTs	0
Total combinational functions	21074
Combinational ALUT usage by number of inputs	
-- 7 input functions	1
-- 6 input functions	3484
-- 5 input functions	8
-- 4 input functions	19
-- <=3 input functions	17562
Combinational ALUTs by mode	
-- normal mode	12871
-- extended LUT mode	1
-- arithmetic mode	6026
-- shared arithmetic mode	2176

Resource	Usage
Estimated ALUT/register pairs used	29422
Total registers	15361
-- Dedicated logic registers	15361
-- I/O registers	0
-- LUT_REGs	0
I/O pins	118
DSP block 18-bit elements	0
Maximum fan-out node	clk~input
Maximum fan-out	15362
Total fan-out	145836
Average fan-out	3.98

Figure 1: Resource usage summary for Bitcoin Hashing Module

Fitter Summary	
Fitter Status	Successful - Sat Jun 10 17:28:21 2023
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	bitcoin_hash
Top-level Entity Name	bitcoin_hash
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	94 %
Combinational ALUTs	21,098 / 36,100 (58 %)
Memory ALUTs	0 / 18,050 (0 %)
Dedicated logic registers	15,370 / 36,100 (43 %)
Total registers	15370
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 %)

Figure 2: Fitter report snapshot.

Fmax	Restricted Fmax	Clock Name	Note
144.13 MHz	144.13 MHz	clk	

Figure 3: $f_{\max}$ report snapshot.

3 Simulation Transcripts

3.1 SHA-256

```
# --------
# 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    : .../simplified_sha256/tb_simplified_sha256.sv(262)
#    Time: 3410 ps  Iteration: 2  Instance: /tb_simplified_sha256
```

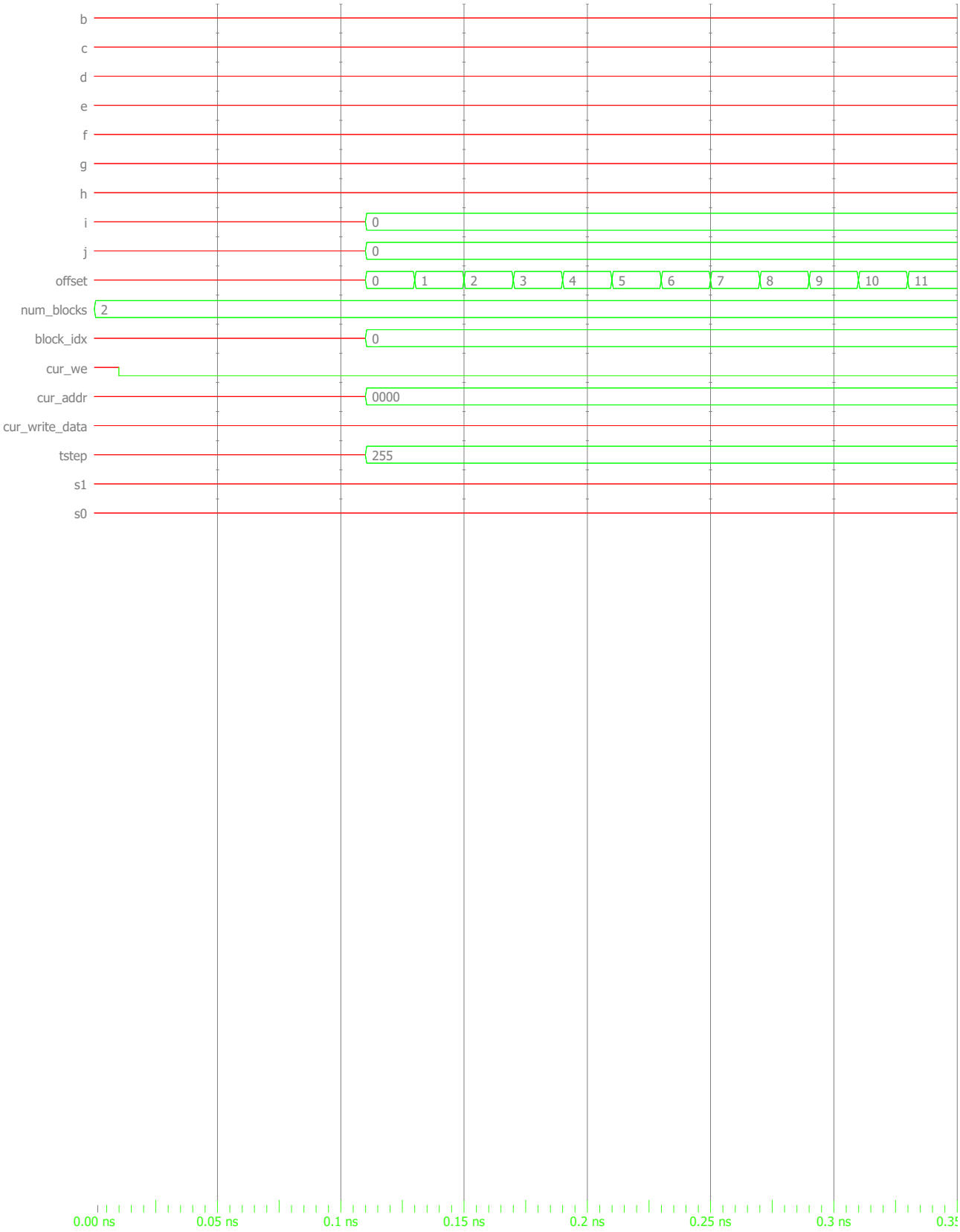
3.2 Bitcoin Hashing

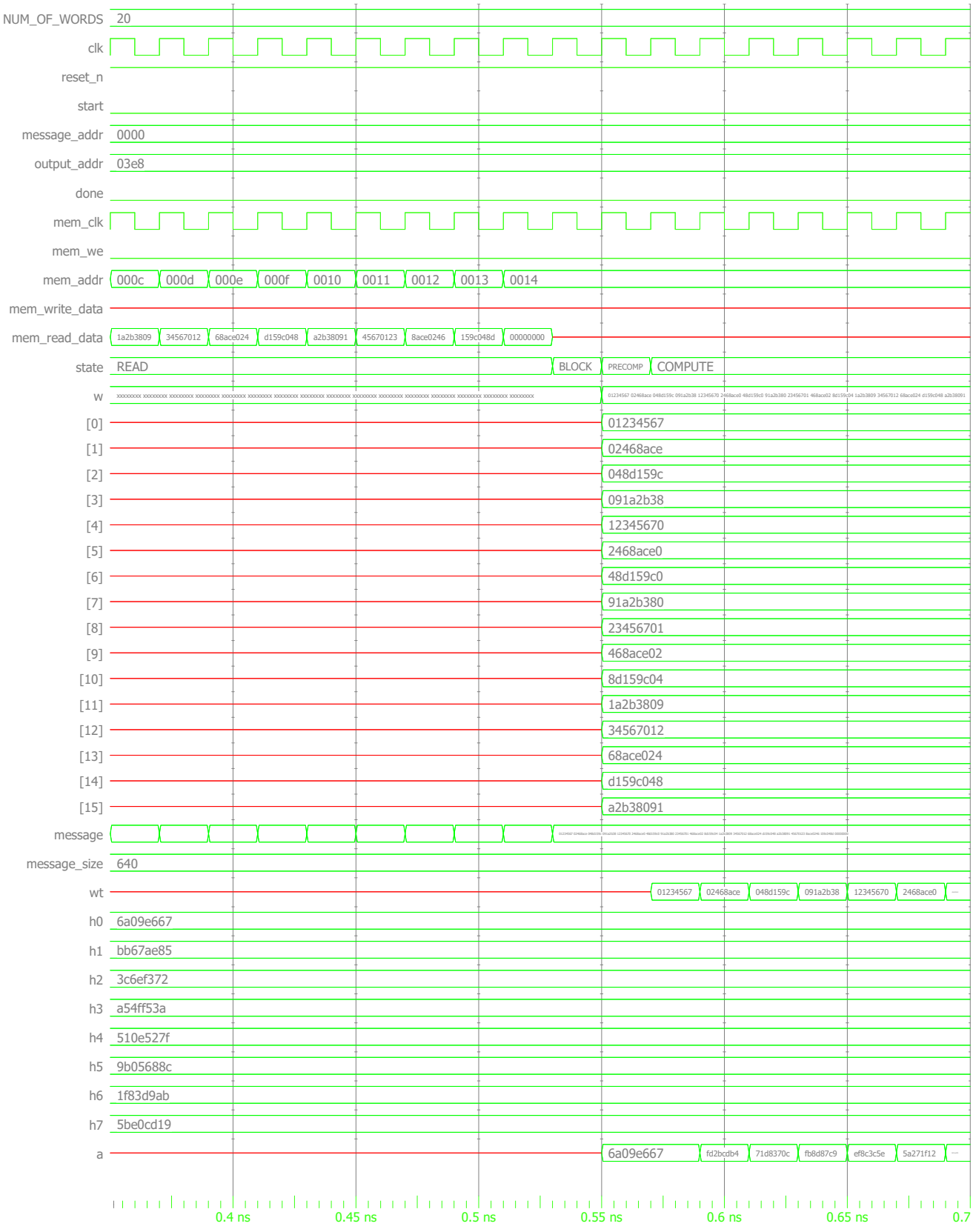
```
# ---------------
# 19 WORD HEADER:
# ---------------
# 01234567
# 02468ace
# 048d159c
# 091a2b38
# 12345670
# 2468ace0
# 48d159c0
# 91a2b380
# 23456701
# 468ace02
# 8d159c04
# 1a2b3809
# 34567012
# 68ace024
# d159c048
# a2b38091
# 45670123
# 8ace0246
# 159c048d
# ***************************
#
# ---------------------
# 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:         244
#
#
# ***************************
#
# ** Note: $stop    : .../bitcoin_hash/tb_bitcoin_hash.sv(338)
#    Time: 5010 ps  Iteration: 2  Instance: /tb_bitcoin_hash
```

4 Simulation Waveforms

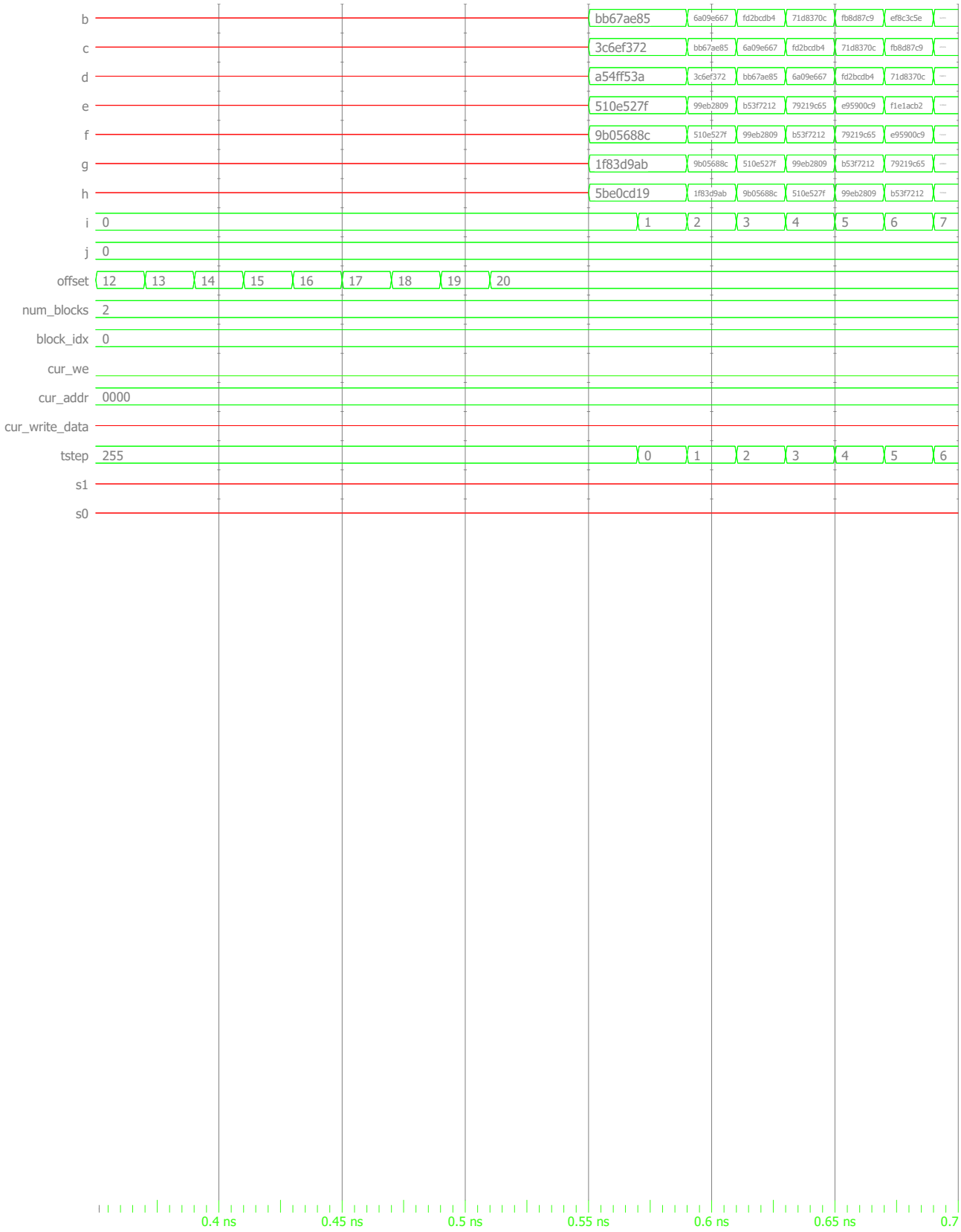
4.1 SHA-256

Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 1 Page: 1

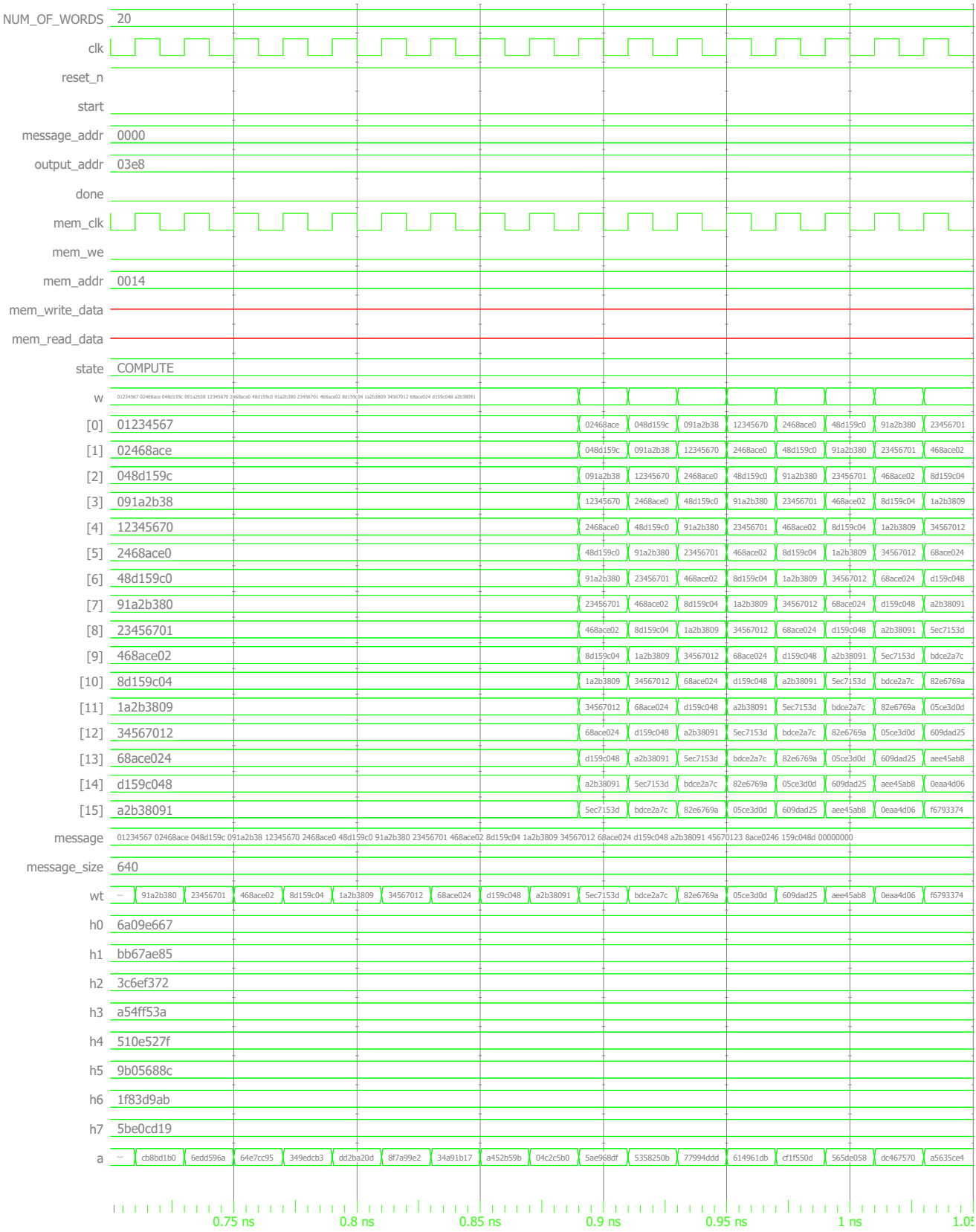

b
c
d
e
f
g
h
i
j
offset
num_blocks
block_idx
cur_we
cur_addr
cur_write_data
tstep
s1
s0
0
0
0 1 2 3 4 5 6 7 8 9 10 11
2
0
0000
255
0.00 ns
0.05 ns
0.1 ns
0.15 ns
0.2 ns
0.25 ns
0.3 ns
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 1 Page: 2



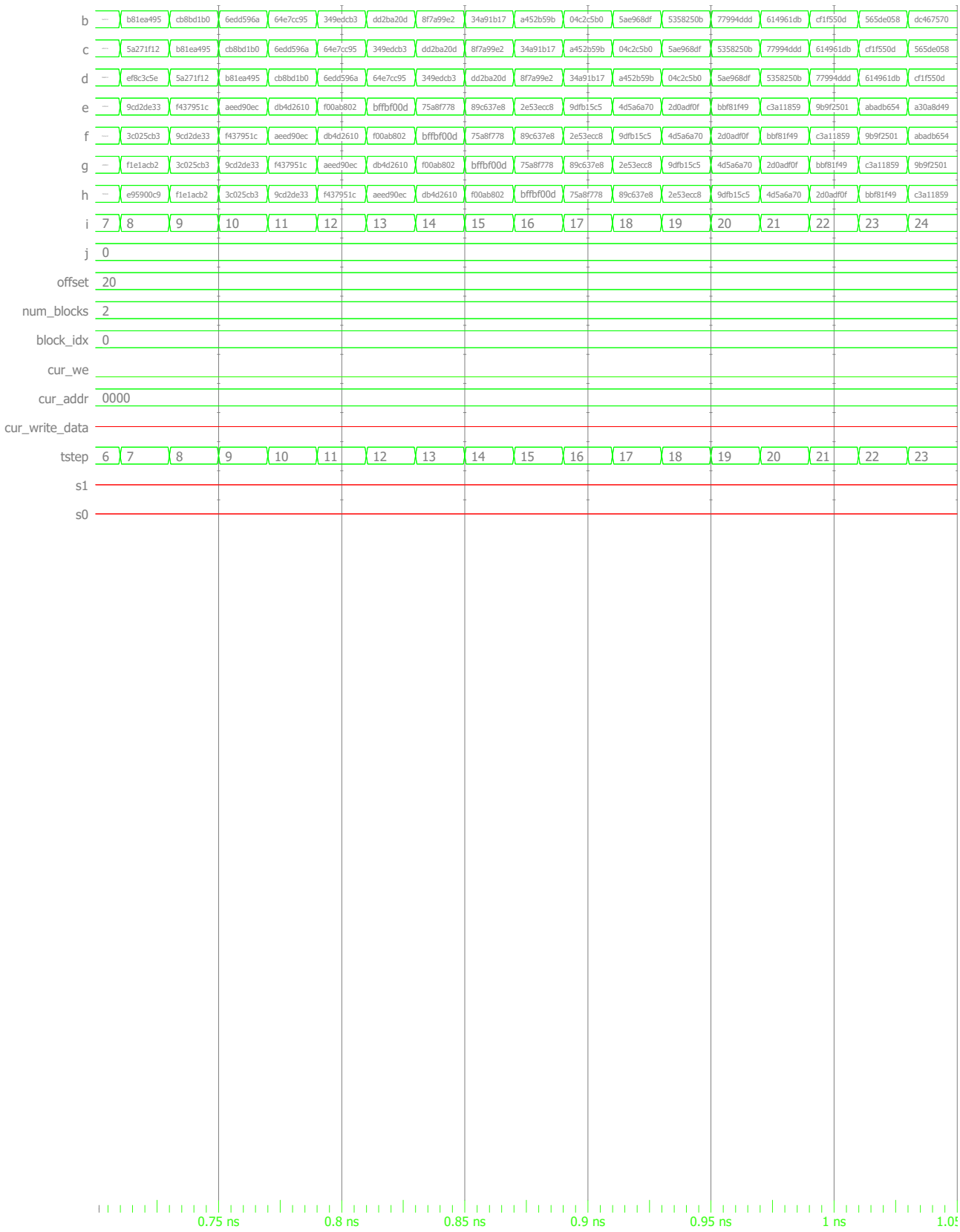
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 2 Page: 3



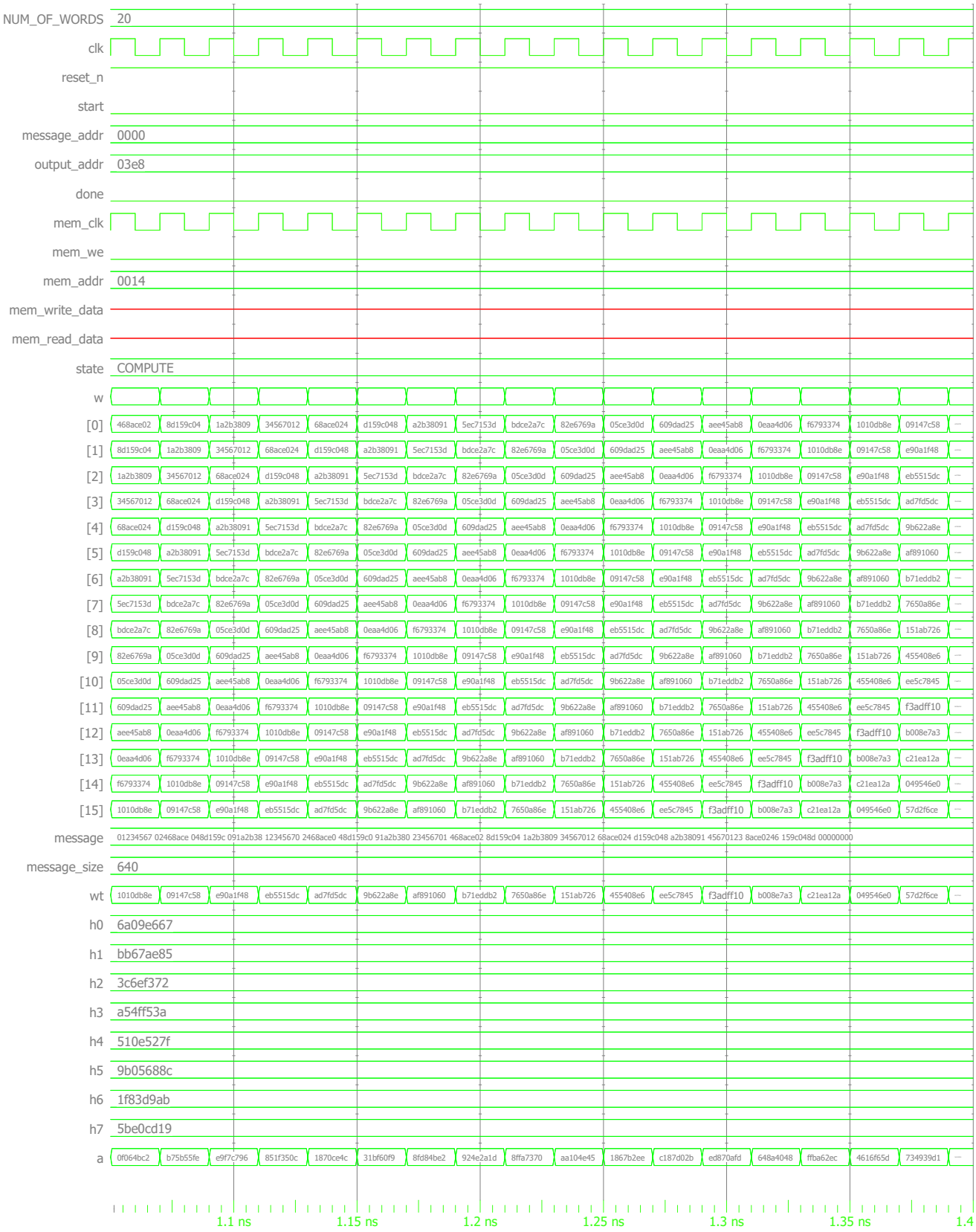
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 2 Page: 4



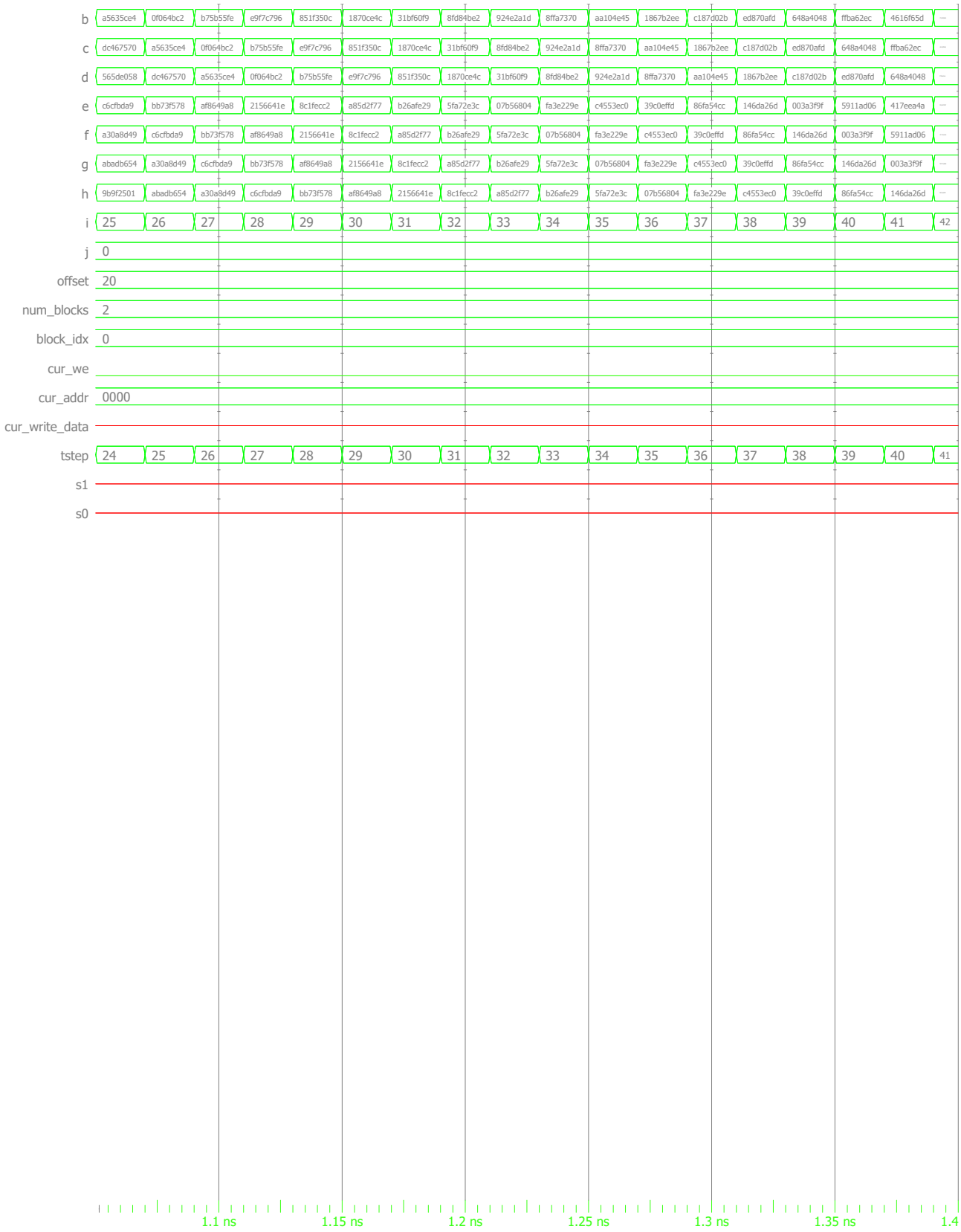
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 3 Page: 5



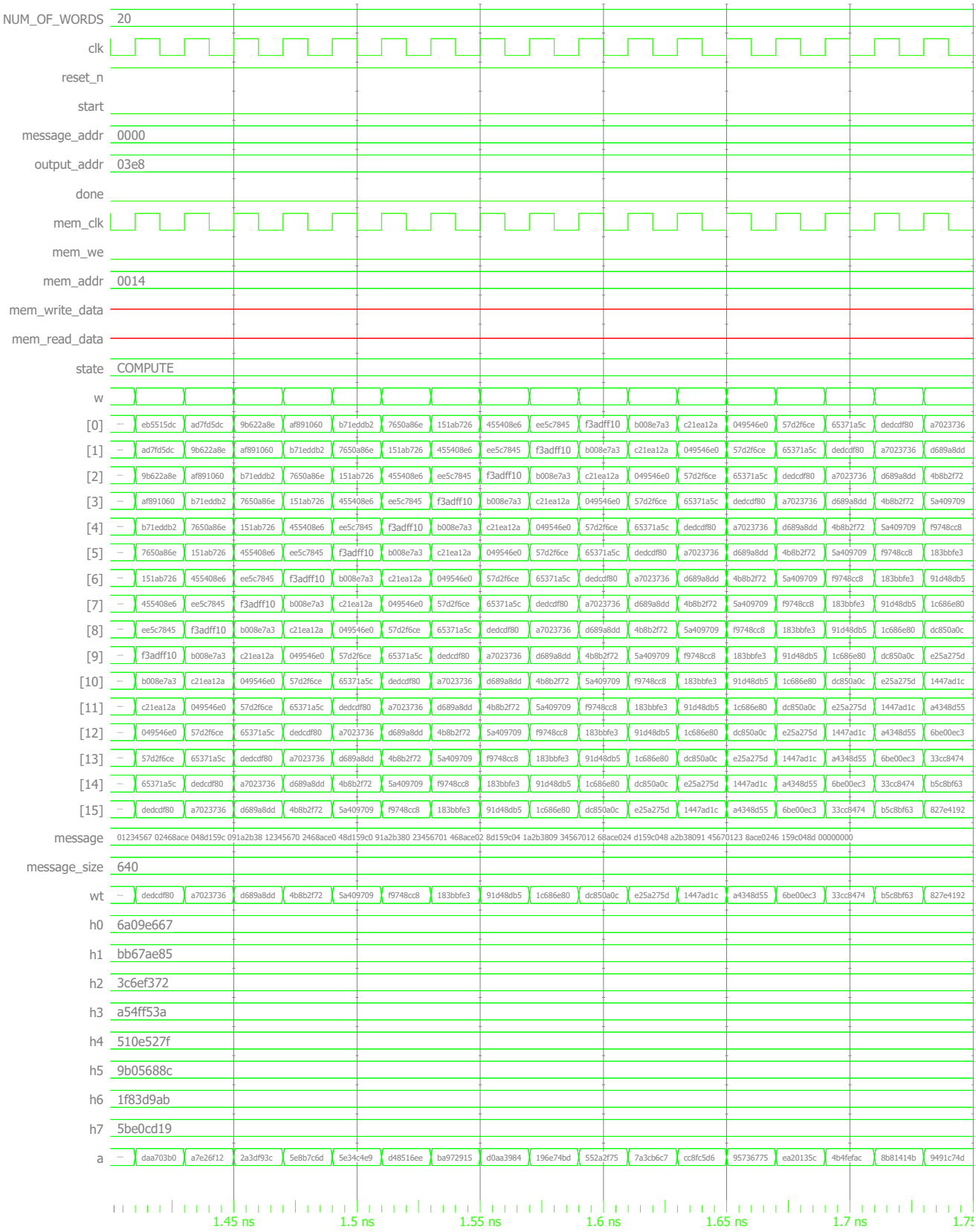
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 3 Page: 6



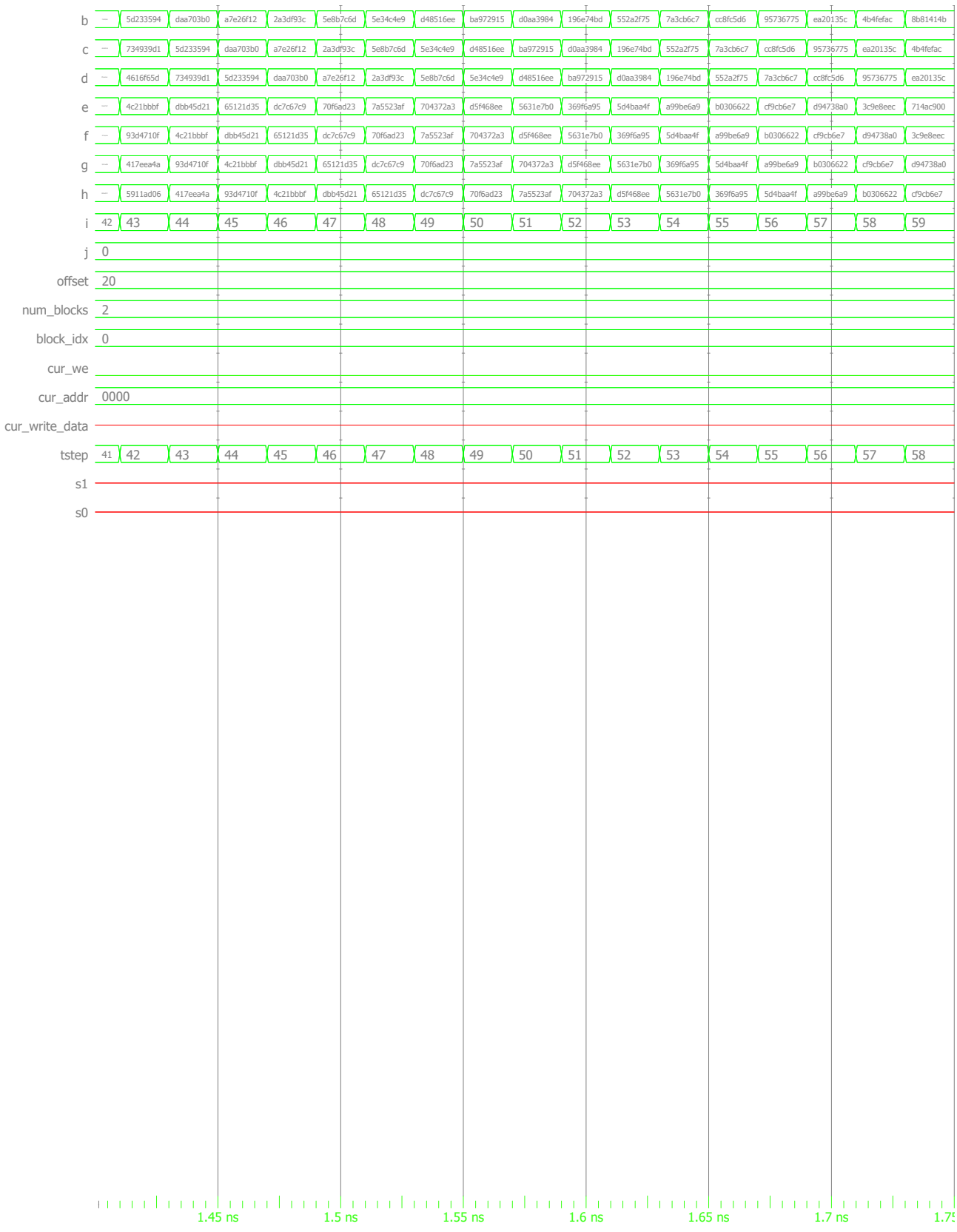
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 4 Page: 7



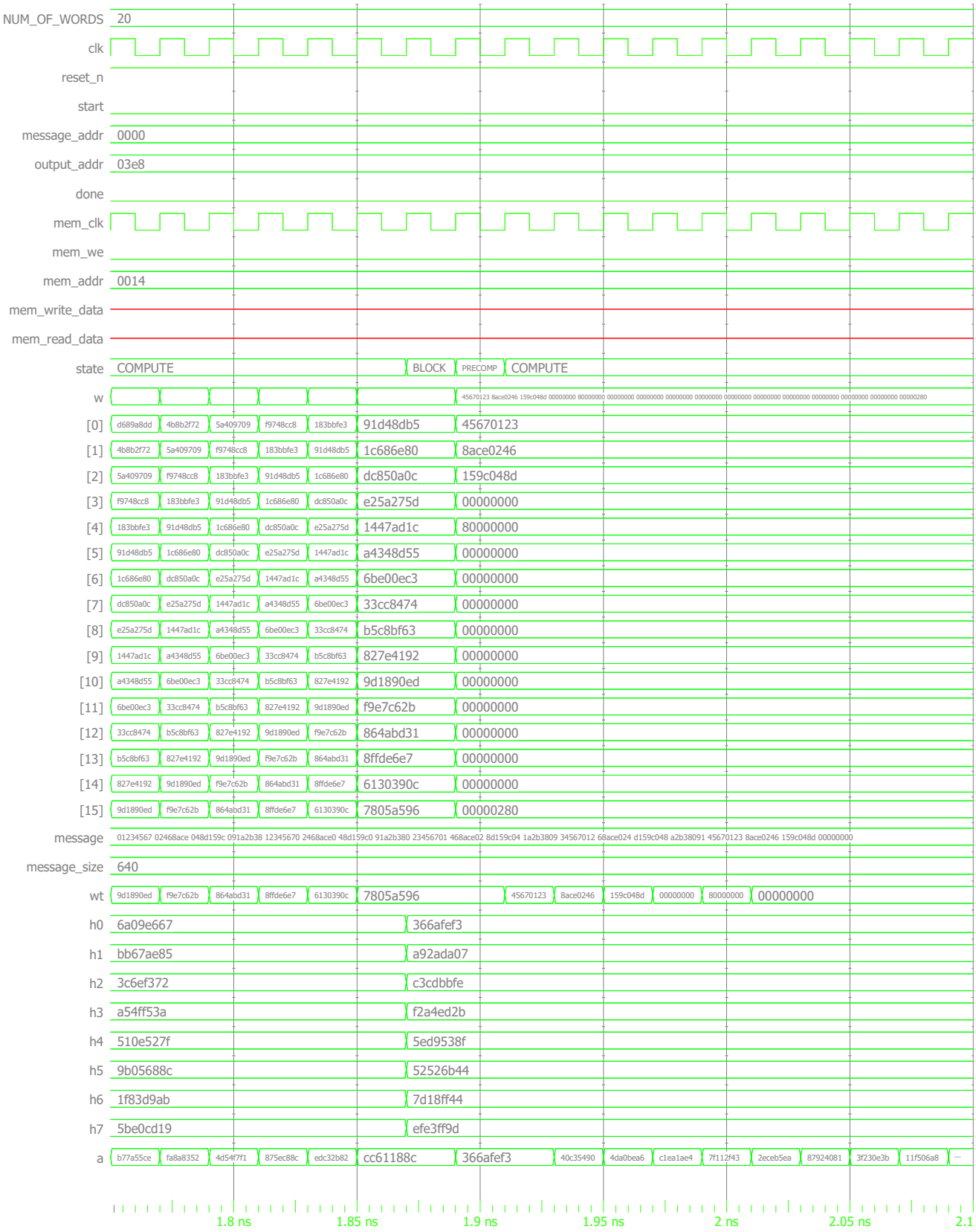
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 4 Page: 8



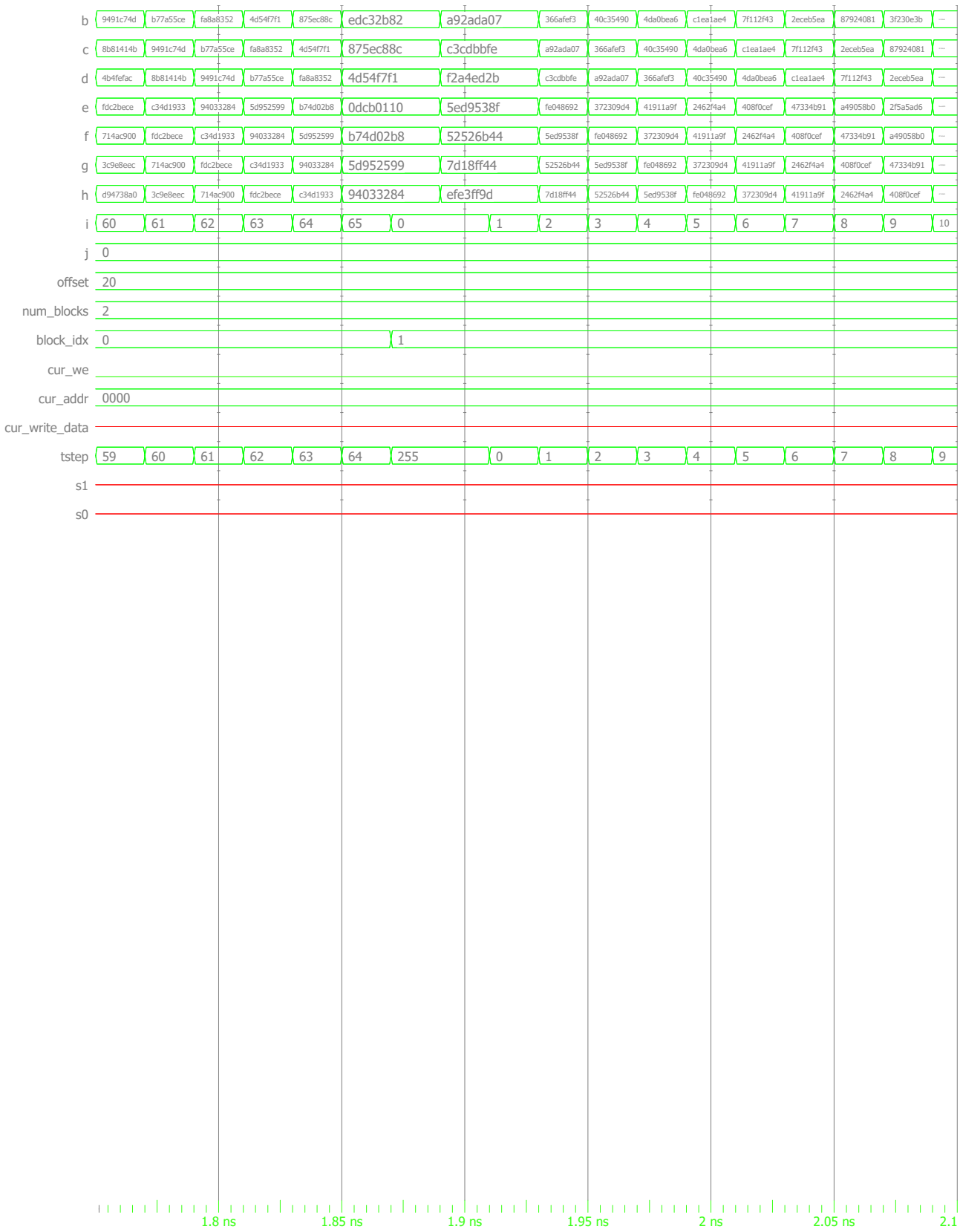
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 5 Page: 9



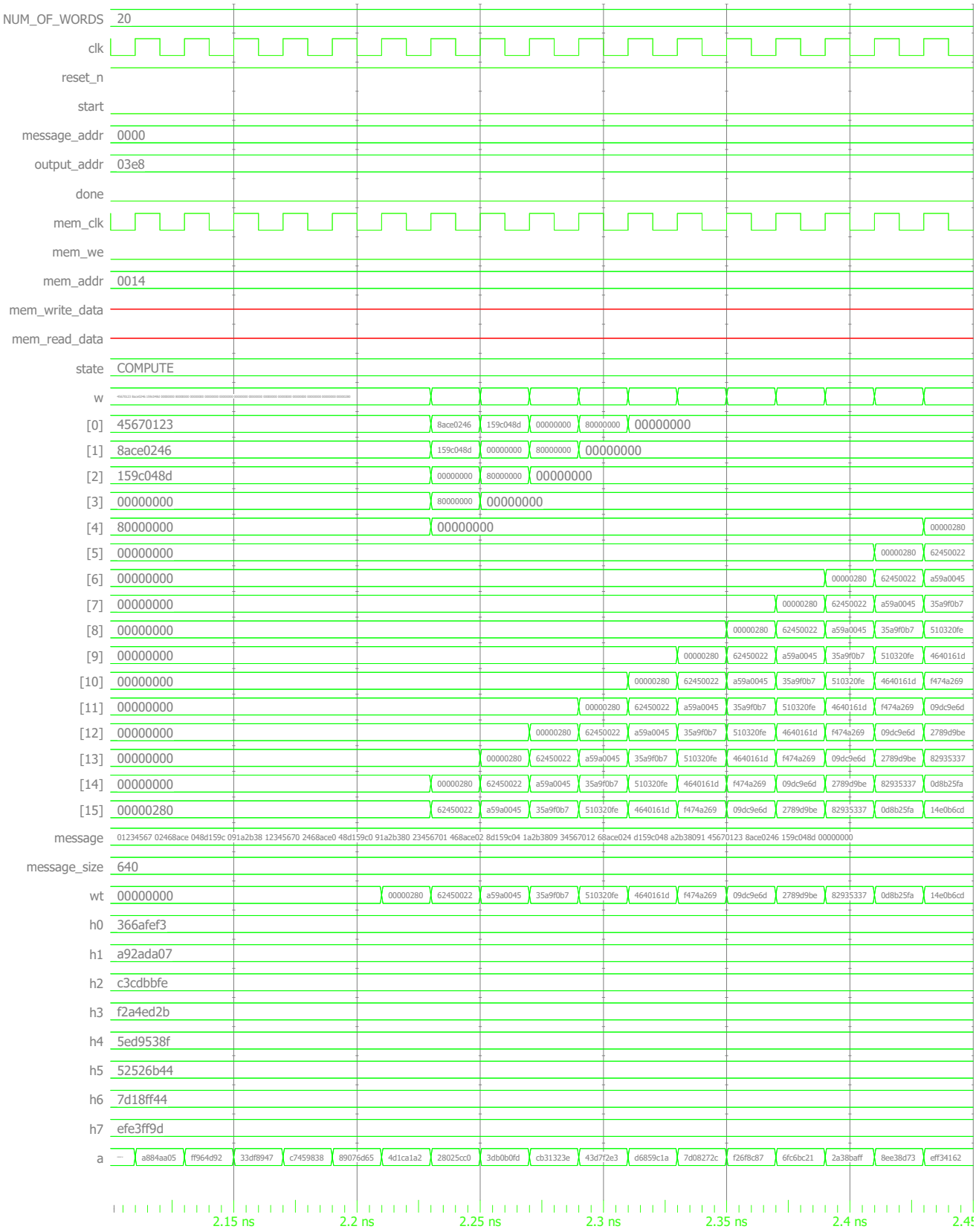
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 5 Page: 10



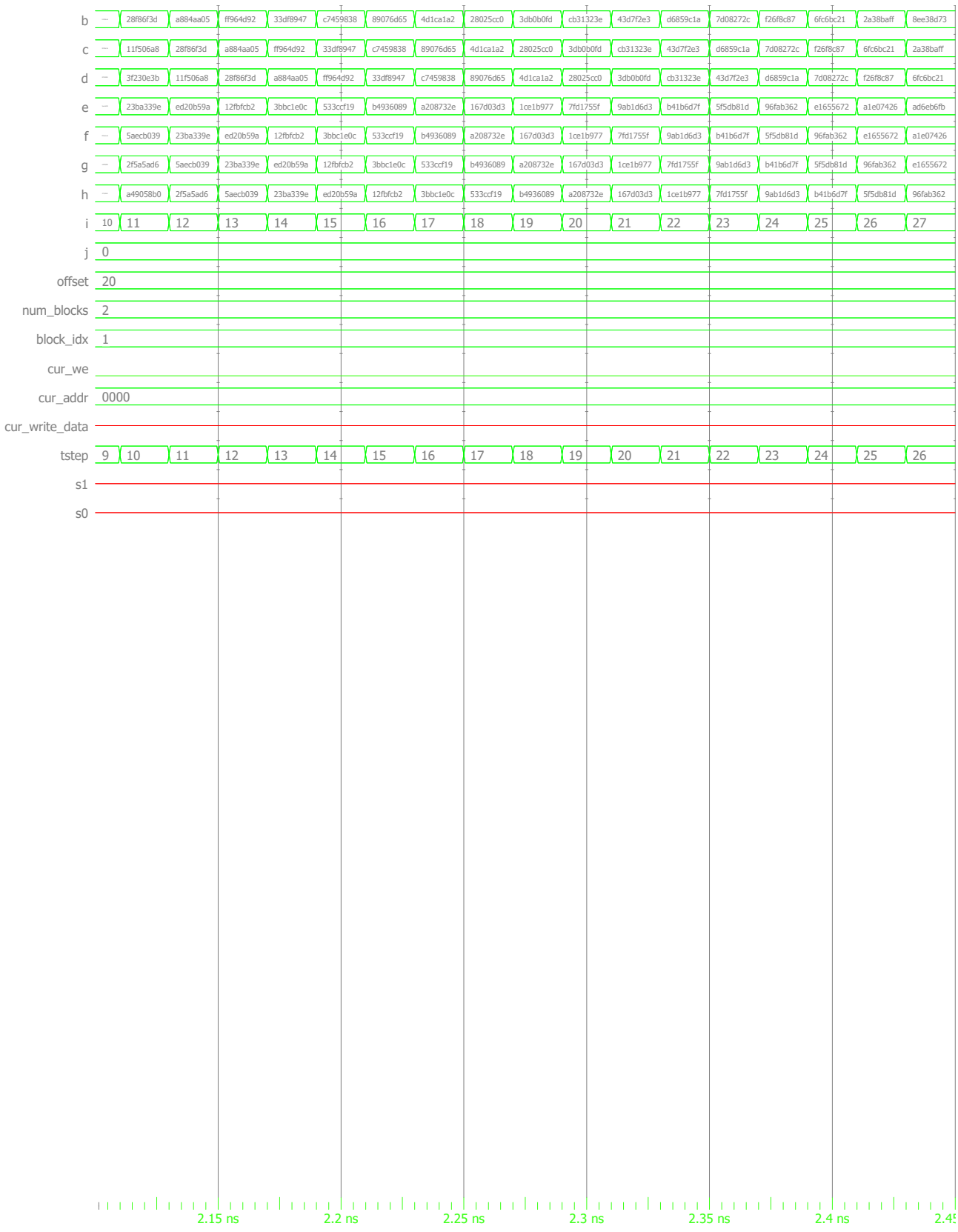
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 6 Page: 11



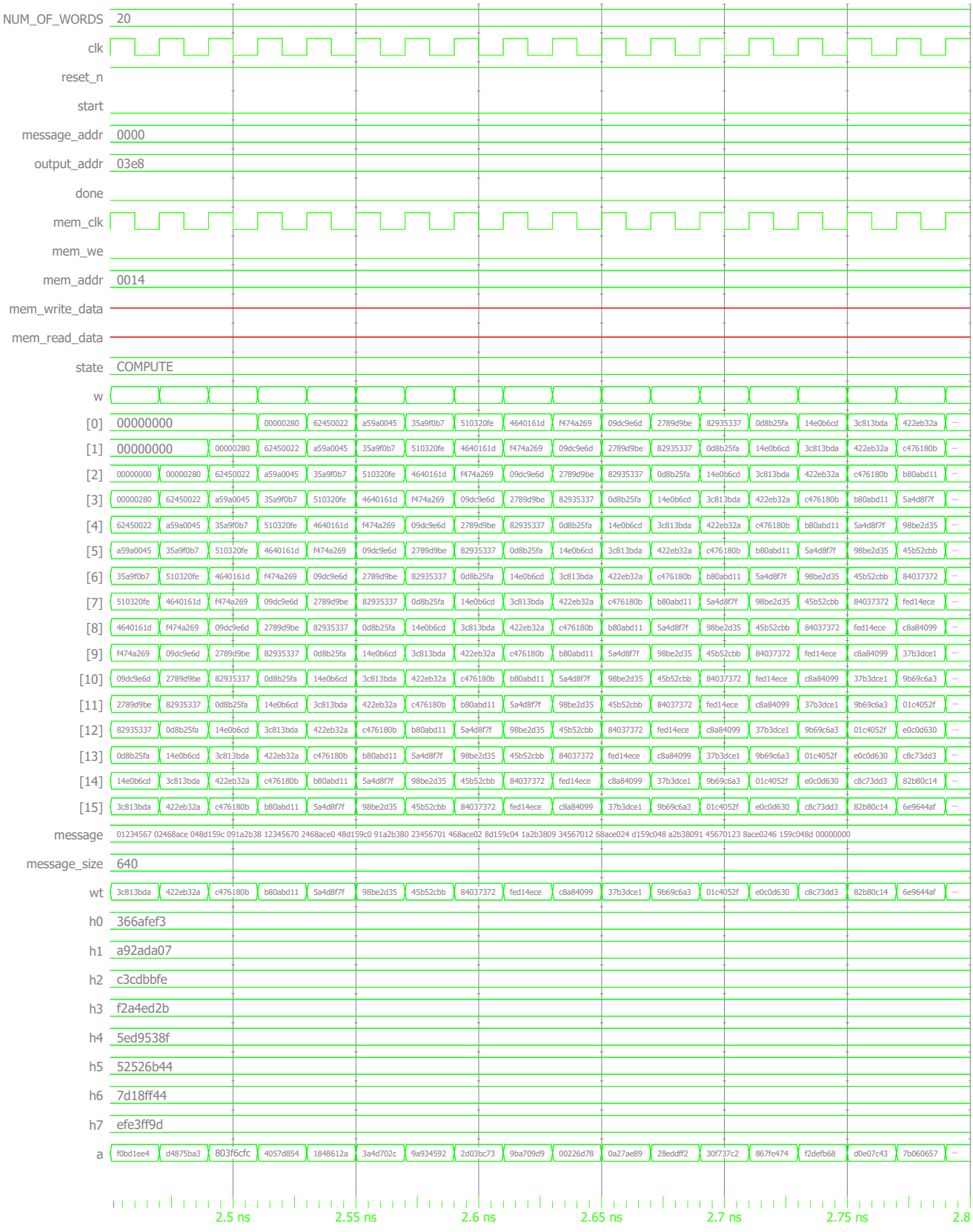
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 6 Page: 12



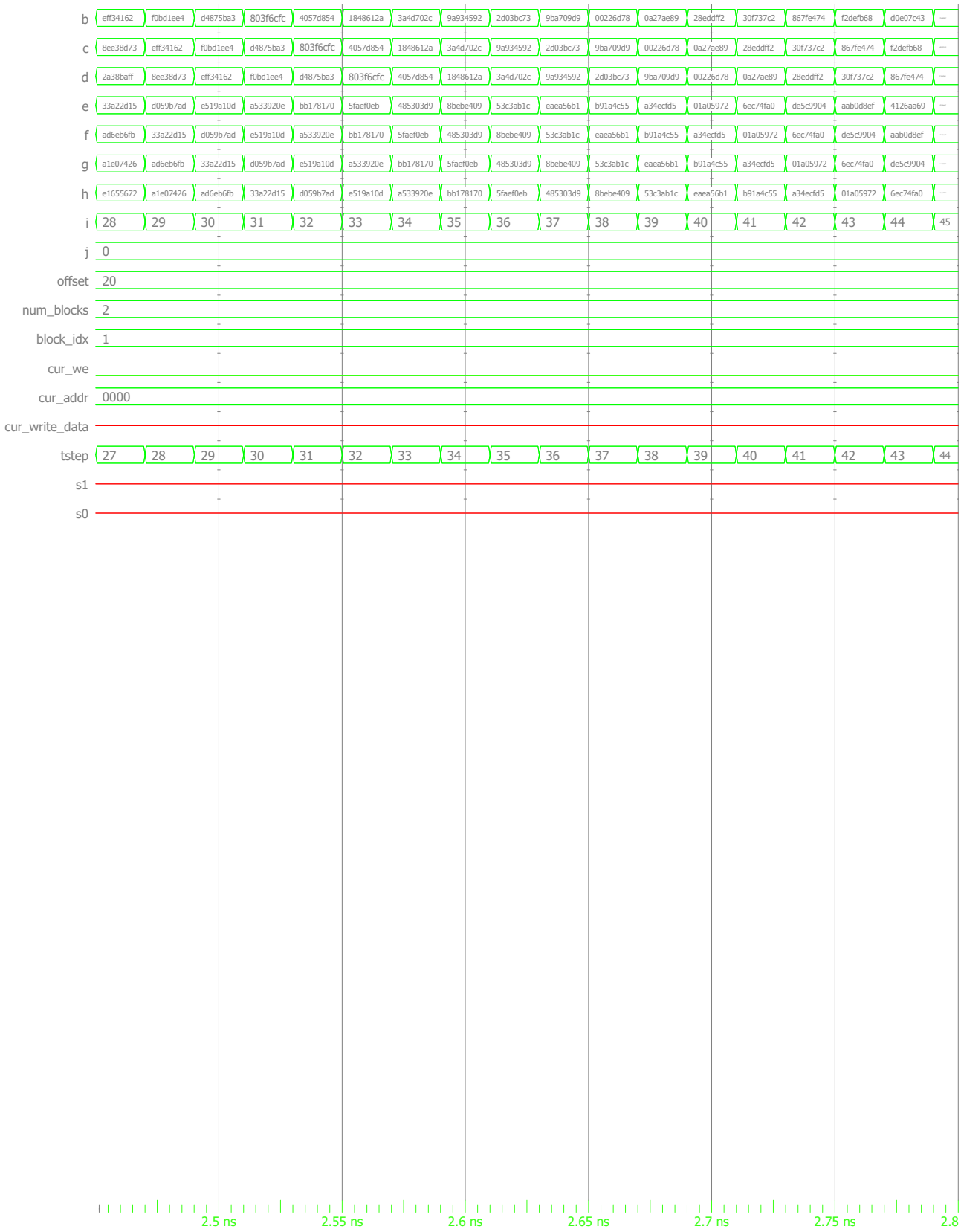
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 7 Page: 13



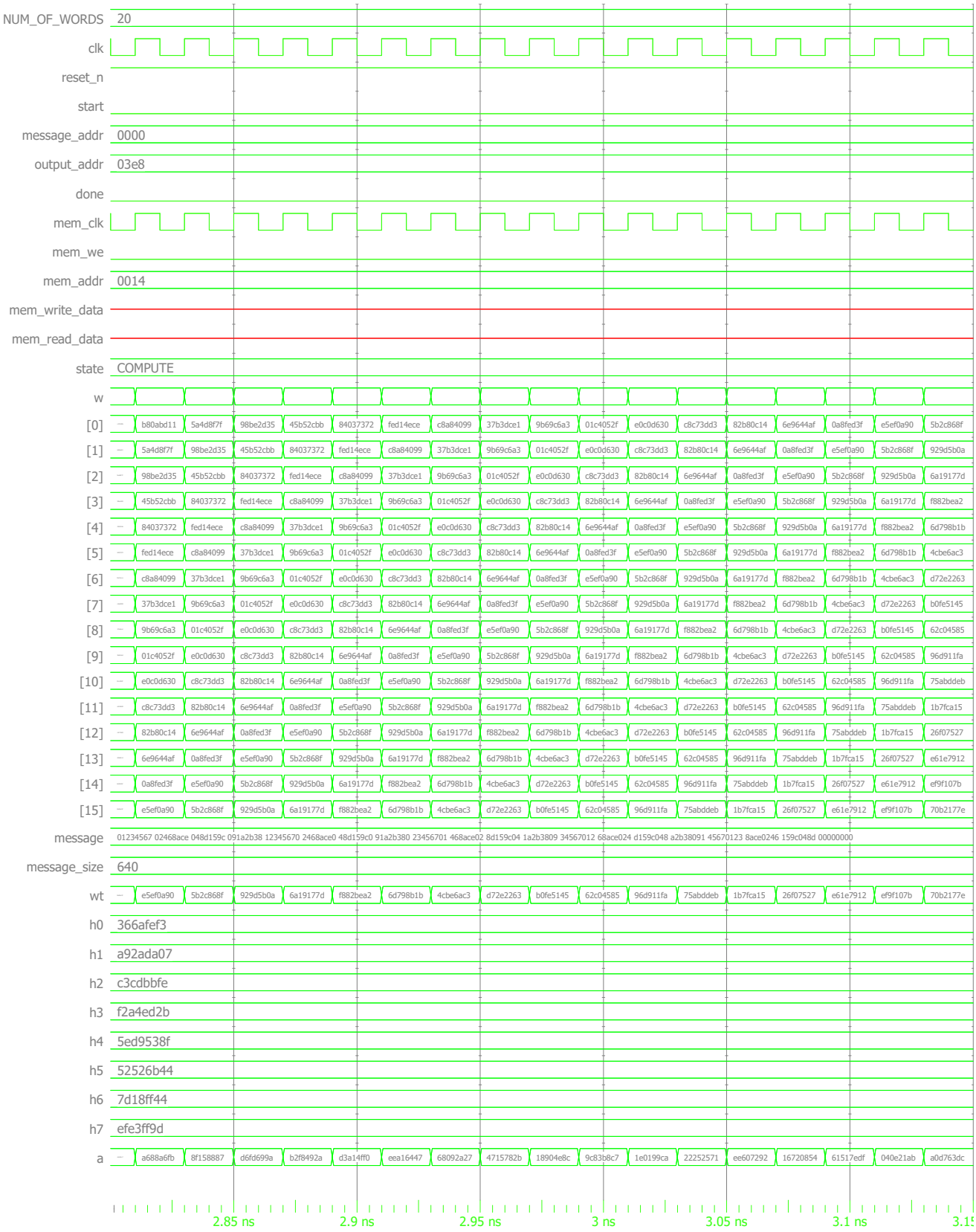
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 7 Page: 14



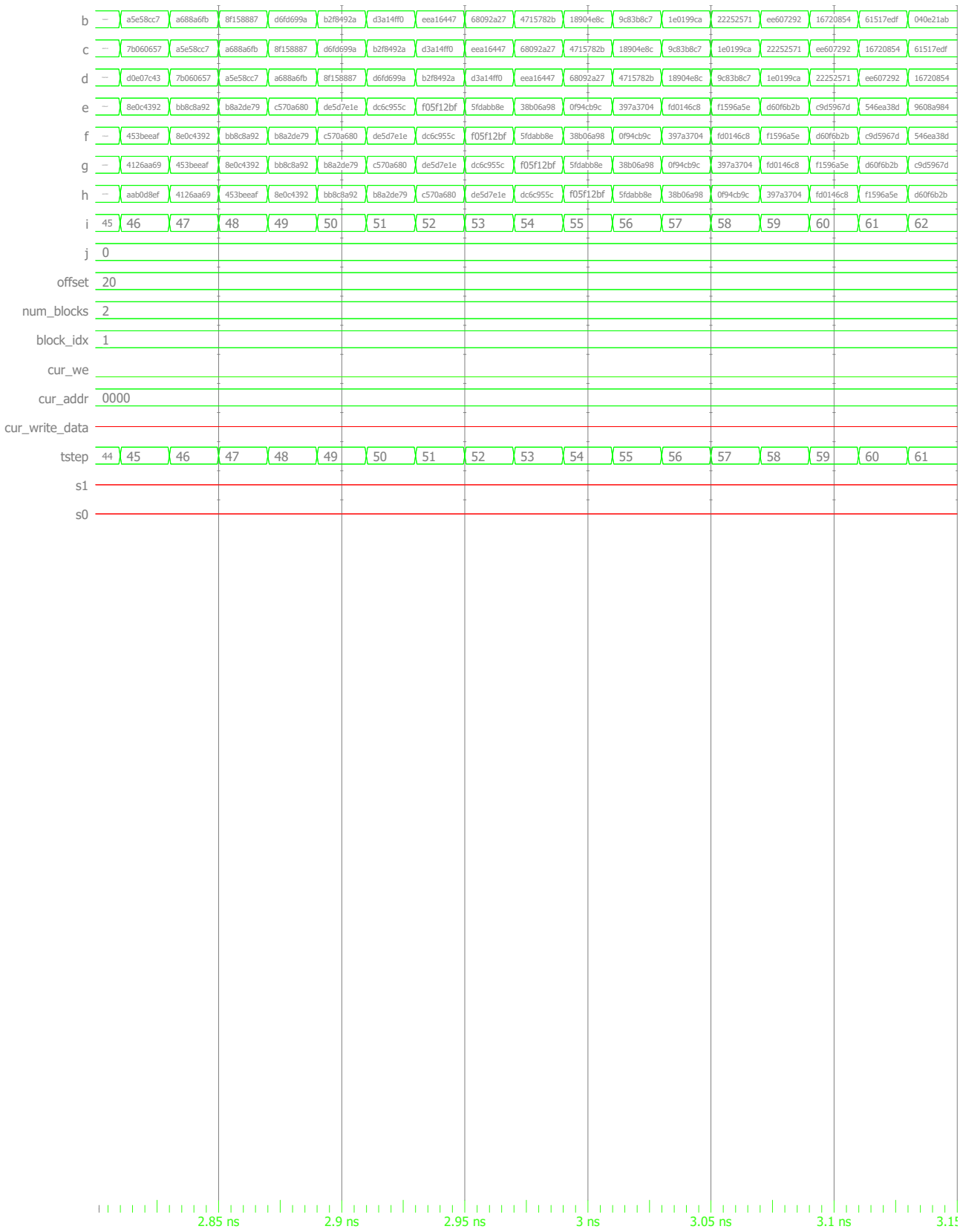
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 8 Page: 15



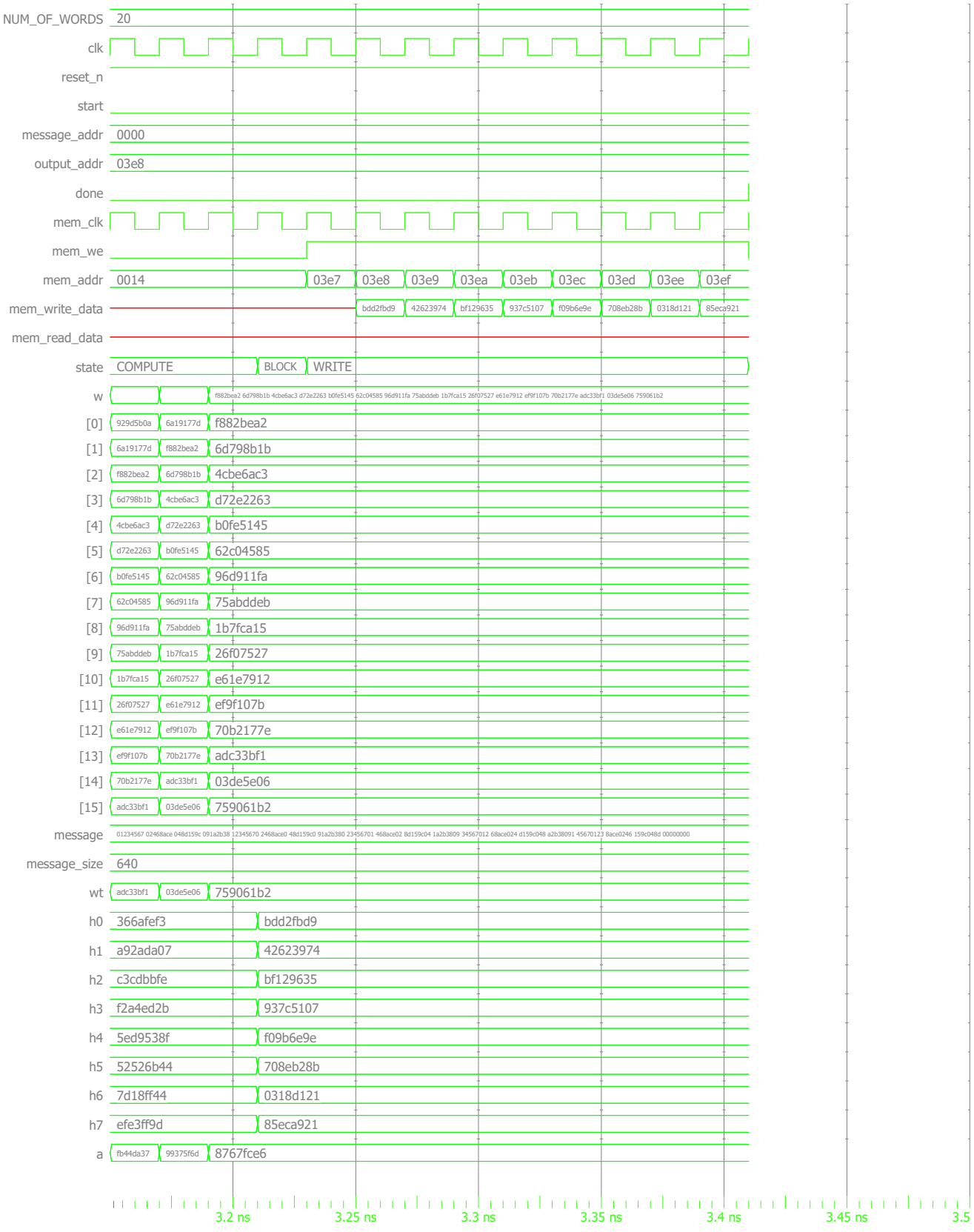
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 8 Page: 16



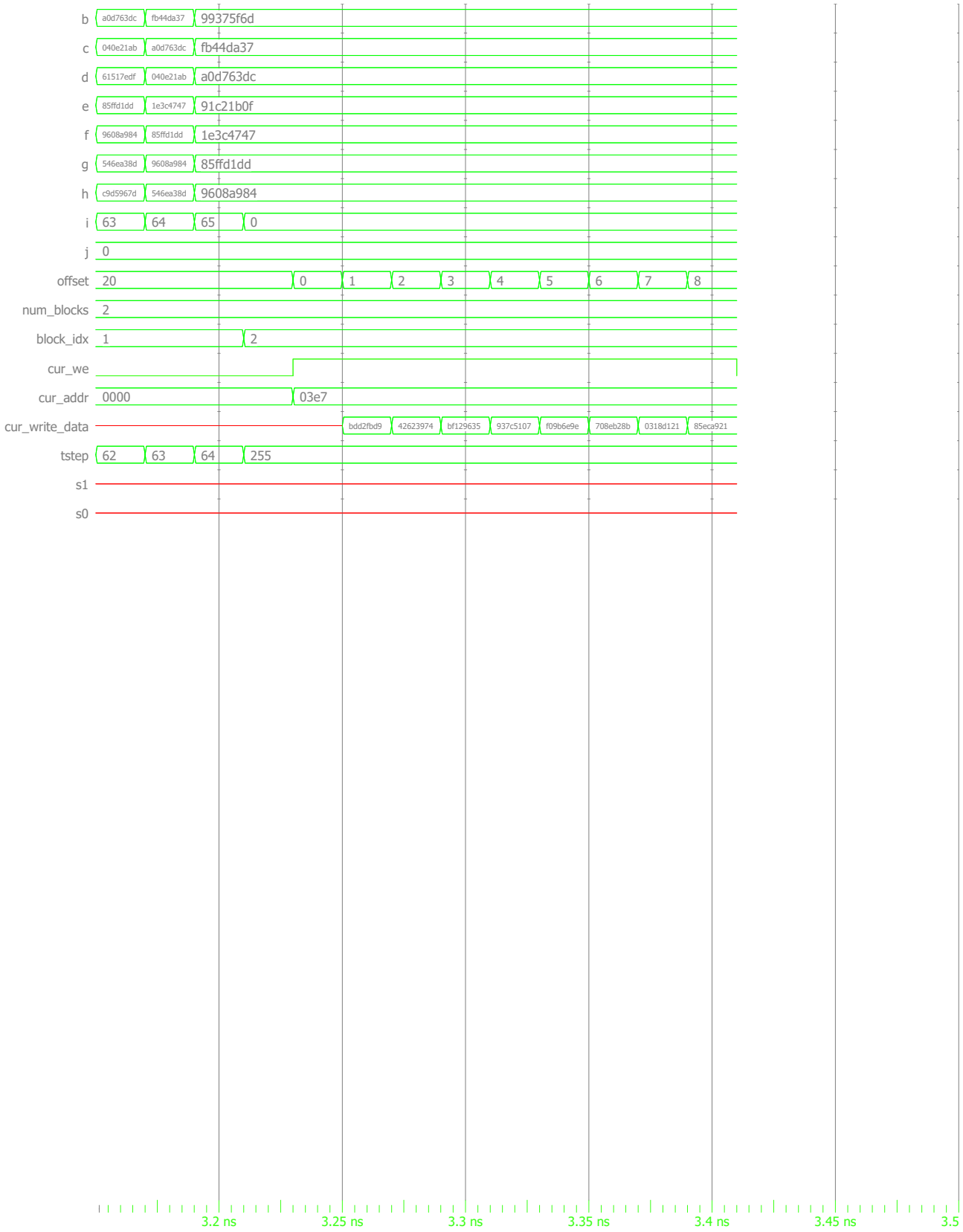
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 9 Page: 17



Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 9 Page: 18

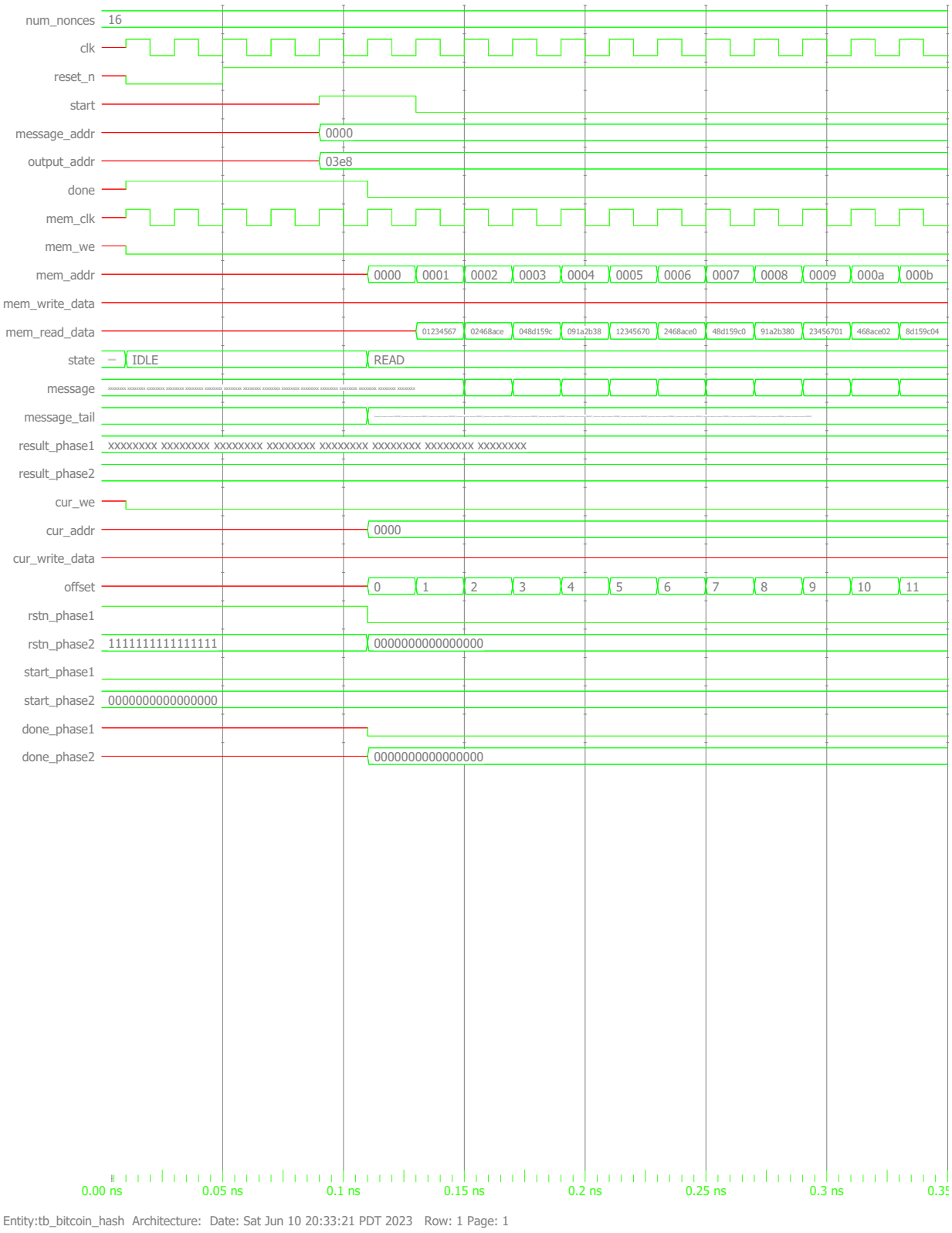


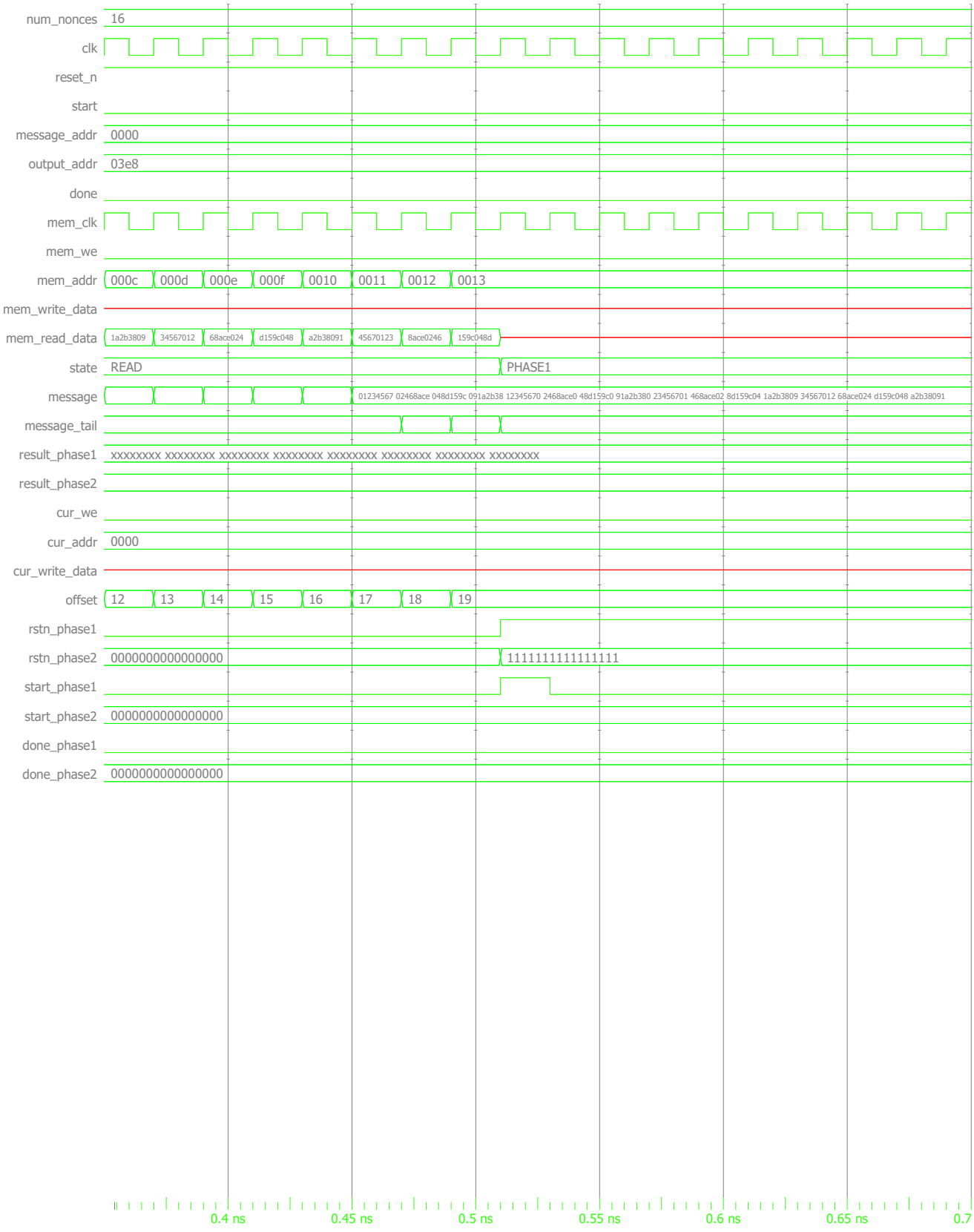
Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 10 Page: 19



Entity:tb_simplified_sha256 Architecture: Date: Sat Jun 10 19:54:26 PDT 2023 Row: 10 Page: 20

4.2 Bitcoin Hashing





Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 2 Page: 2

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 3 Page: 3

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 4 Page: 4

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 5 Page: 5

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 6 Page: 6

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 7 Page: 7

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 8 Page: 8

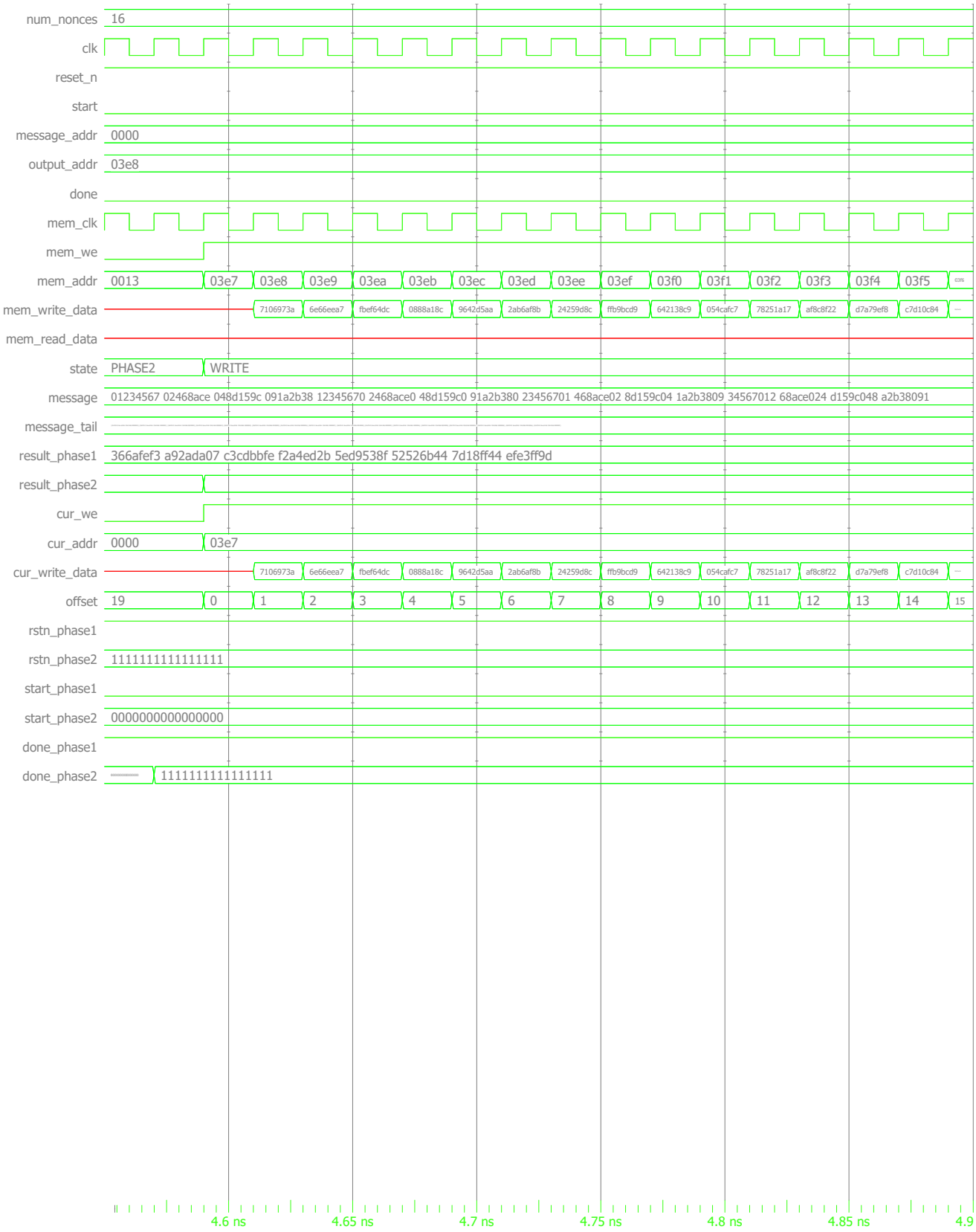
Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 9 Page: 9

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 10 Page: 10

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 11 Page: 11

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 12 Page: 12

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 13 Page: 13



Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 14 Page: 14

Entity:tb_bitcoin_hash Architecture: Date: Sat Jun 10 20:33:21 PDT 2023 Row: 15 Page: 15