RTL Implementation of SHA and Bitcoin Algorithm

Suraj Sathya Prakash[†], Achyuth Mahesh Esthuri[†], Anoushka Saraswat[†]
[†]University of California San Diego

***Abstract*—**

I. Secure Hash Algorithm

A. Introduction

SHA, or Secure Hash Algorithm, is a family of cryptographic hash functions designed by the United States National Security Agency (NSA) and published by the United States National Institute of Standards and Technology (NIST). SHA algorithms convert input data of any size into a fixed-size output, known as a hash value or message digest. There are different versions of SHA, including SHA-1, SHA-2, and SHA-3, each producing hash values of varying bit lengths.

B. Properties of SHA

SHA must have several critical properties to ensure its security and effectiveness:

- Compression: The output hash value is of a fixed length, irrespective of the input message size.
- Avalanche Effect: A slight change in the input drastically changes the output hash.
- Determinism: The same input consistently produces the same hash output.
- One Way Function: It should be computationally infeasible to reverse-engineer the input from its hash output
- Collision Resistance: It should be implausible to find two different inputs that produce the same hash output.
- Efficiency: The hashing process must be fast and computationally efficient.

C. Applications of SHA

SHA is widely used in various applications requiring data integrity and security:

- Storing and Validating Passwords: Passwords are encrypted and stored in SHA256. When users log in, the system hashes the input password and compares it to the stored hash.
- Verifying File Integrity: Software providers use SHA to ensure files have not been tampered with. Users can verify the file's integrity by comparing the hash of a downloaded file with the provided hash.
- Digital Signatures: SHA is used with digital signatures to ensure the authenticity and integrity of messages and documents.
- Blockchain and Cryptocurrencies: SHA256, in particular, is fundamental in blockchain technology, including its use in Bitcoin mining and transaction verification.

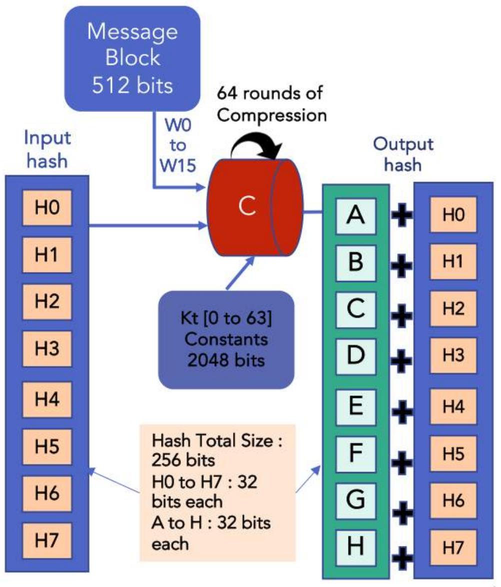


Fig. 1: Enter Caption

D. Algorithm

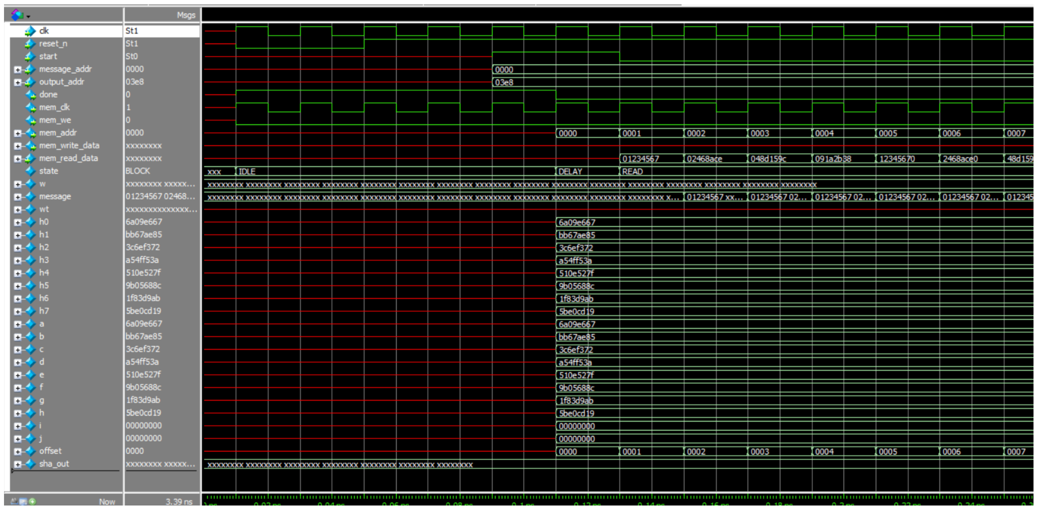
II. Bitcoin

A. Introduction

1) Blockchain Technology: Blockchain technology is the oldest and most well-known blockchain in existence. Each block in the Bitcoin blockchain typically contains about 1 MB of transaction data, and all blocks are linked together in a chain that records every Bitcoin transaction made. The Bitcoin blockchain employs the SHA256 cryptographic hashing algorithm to secure and link these blocks.

2) Data Integrity in Bitcoin Blockchain: The data integrity in the Bitcoin blockchain is maintained through its hashing mechanism. If an attempt is made to alter the data in a block, such as changing the amount of Bitcoin in a transaction, the hash for that block will change, breaking the chain link with subsequent blocks. The network detects this discrepancy, rejects the altered block, and maintains the original chain.

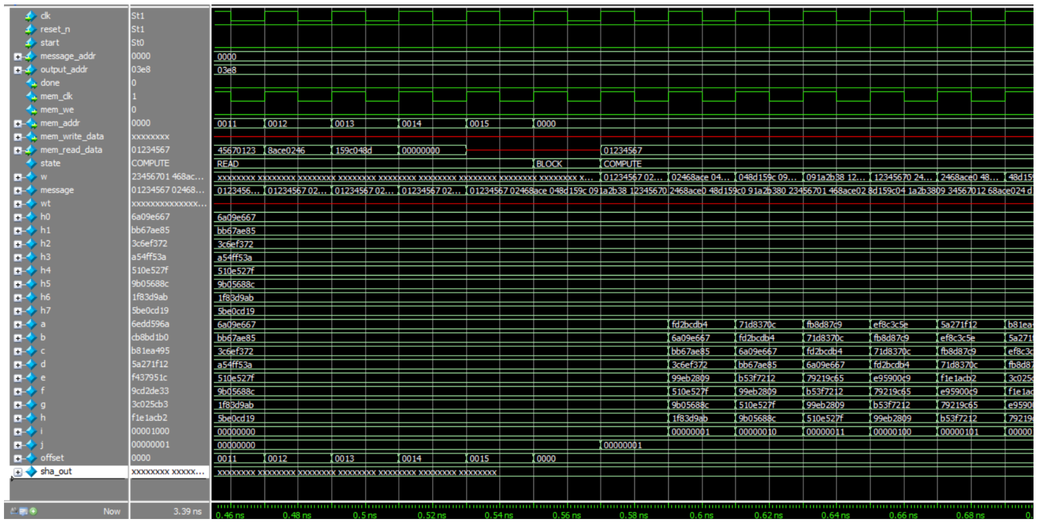
3) Acceptance of Data Blocks in the Blockchain: For a block to be accepted in the Bitcoin blockchain, its hash must meet certain criteria, such as starting with a specified number of leading zeroes. This requirement is achieved by varying the nonce, a small data piece that, when changed, produces a different hash until one with the desired properties is found.



Signal	Msgs
clk	St1
reset_n	St1
start	St0
message_addr	0000
output_addr	03e8
done	0
mem_clk	1
mem_we	0
mem_addr	0000
mem_write_data	xxxxxxxx
mem_read_data	xxxxxxxx
state	BLOCK
w	xxxxxxxx xxxxx...
message	01234567 02468...
wt	xxxxxxxxxxxxxx...
h0	6a09e667
h1	bb67ae85
h2	3c6ef372
h3	a54ff53a
h4	510e527f
h5	9b05688c
h6	1f83d9ab
h7	5be0cd19
a	6a09e667
b	bb67ae85
c	3c6ef372
d	a54ff53a
e	510e527f
f	9b05688c
g	1f83d9ab
h	5be0cd19
i	00000000
j	00000000
offset	0000
sha_out	xxxxxxxx xxxxx...

Now 3.39 ns

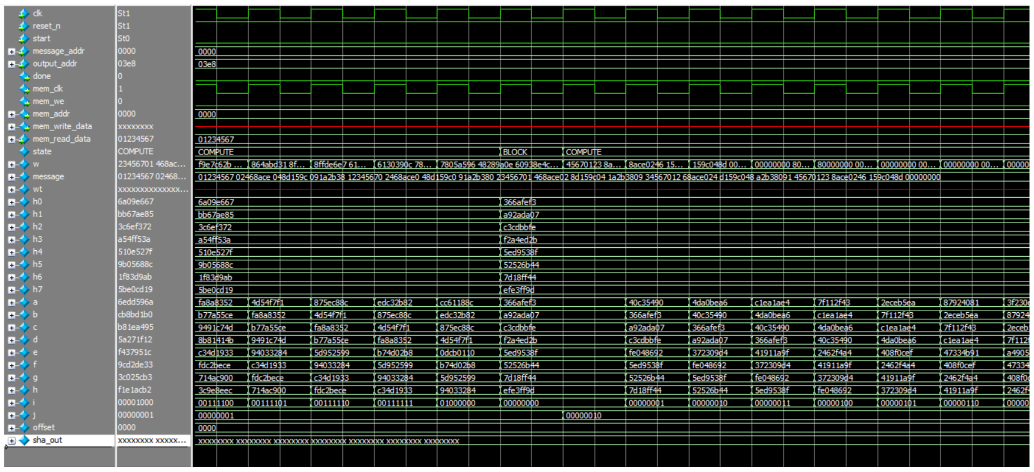
Fig. 2: Enter Caption



Signal	Msgs
clk	St1
reset_n	St1
start	St0
message_addr	0000
output_addr	03e8
done	0
mem_clk	1
mem_we	0
mem_addr	0000
mem_write_data	xxxxxxxx
mem_read_data	01234567
state	COMPUTE
w	23456701 468ac...
message	01234567 02468...
wt	xxxxxxxxxxxxxx...
h0	6a09e667
h1	bb67ae85
h2	3c6ef372
h3	a54ff53a
h4	510e527f
h5	9b05688c
h6	1f83d9ab
h7	5be0cd19
a	6edd596a
b	cb8bd1b0
c	b81ea495
d	5a271f12
e	f437951c
f	9cd2de33
g	3c025cb3
h	f1e1acb2
i	00001000
j	00000001
offset	0000
sha_out	xxxxxxxx xxxxx...

Now 3.39 ns

Fig. 3: Enter Caption



Signal	Msgs
clk	St1
reset_n	St1
start	St0
message_addr	0000
output_addr	03e8
done	0
mem_clk	1
mem_we	0
mem_addr	0000
mem_write_data	xxxxxxxx
mem_read_data	01234567
state	COMPUTE
w	23456701 468ac...
message	01234567 02468...
wt	xxxxxxxxxxxxxx...
h0	6a09e667
h1	bb67ae85
h2	3c6ef372
h3	a54ff53a
h4	510e527f
h5	9b05688c
h6	1f83d9ab
h7	5be0cd19
a	6edd596a
b	cb8bd1b0
c	b81ea495
d	5a271f12
e	f437951c
f	9cd2de33
g	3c025cb3
h	f1e1acb2
i	00001000
j	00000001
offset	0000
sha_out	xxxxxxxx xxxxx...

Fig. 4: Enter Caption

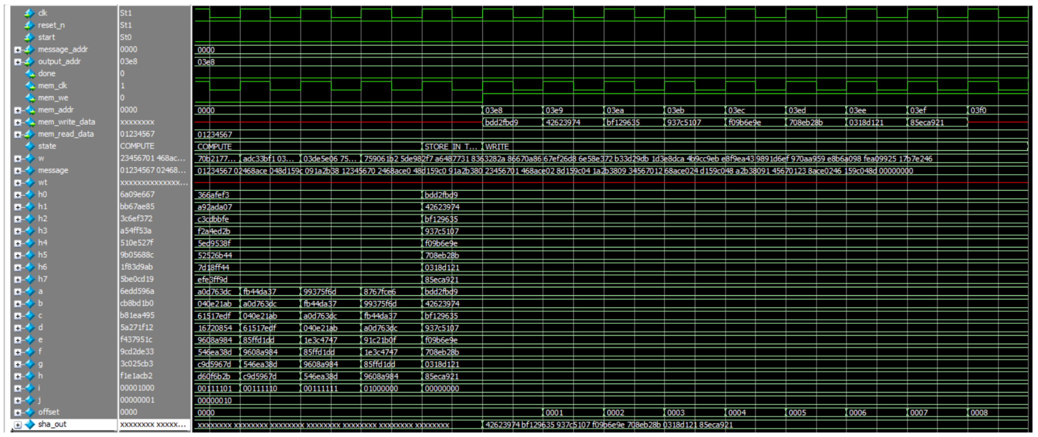


Fig. 5: Enter Caption

This process ensures that adding new blocks to the blockchain is computationally intensive, enhancing security.

4) Immutability of Blockchain: The immutability of the Bitcoin blockchain is ensured by its distributed nature. A malicious actor attempting to alter a block would need to recompute hashes for all subsequent blocks faster than the rest of the network can add new blocks. Given the computational power required, this is practically infeasible, thus securing the blockchain against tampering.

B. Algorithm

The Blockchain algorithm is explained with the help of the Finite state machine model represented in 6. We have incorporated the 9 states in the FSM. The following explanation is a walkthrough of each state's processing in detail.

- Initially, after reset is asserted, we start from the `IDLE` state.
- **IDLE**: In `IDLE` state, we wait for the start signal (`start`) to be asserted. Once `start` is asserted, we see that `h0-h7` and `a-h` values are initialized to the initial values the algorithm expects, .i.e, (`32'h6a09e667`, `32'hbb67ae85`, `32'h3c6ef372` ....). The current address (`cur_addr`) is set as the message address we get from TB and the `offset` value, `i` value (counter for counting the number of rounds for SHA256) and `sha_comp_done` (signal we are using to reuse `PHASE2` as `PHASE3` instead of creating a new state), initialized to 0. The waveform also shows the values being initialized. We also set the next state as the `DELAY` state.
- **`DELAY`**: The reason for including the `DELAY` state is that once `read` is issued, we need to wait for one clock cycle for data to be available on `mem_read_data`. Hence, this `DELAY` state helps to give the one-clock cycle delay. From the `DELAY` state, we then transition to the `READ` state.
- **`READ`**: In the `READ` state, we store the message values coming from `mem_read_data` into a local array message and keep incrementing `offset` every cycle so that we get all 19 words sent from TB into the message array. The waveform also shows the accumulated message array with values sent from TB every cycle. Once all the message values from TB are stored in the message array, we move to the state `PHASE1`. We also store the data in a word array (word array is 2-dimensional with one dimension as nonce and another dimension as a number of words, which we have set as 16 and is optimized). The `word_array` for nonce 0 is reused here for storing the values (`PHASE1` doesn't have nonce, so reusing word array nonce 0 locations to store instead of creating a separate array for `PHASE1`).
- **`PHASE1`**: In `PHASE1` state, we have optimized the compression of SHA256 where the word expansion and computation of SHA256 go hand in hand. Instead of using an expensive word array of 64 locations to do the word expansion and then computation of SHA256 initially (which will consume more cycles as well), we always just pass the first location of the word array as input to the SHA256 function. We also left shift the contents of the word array. At the last location, we use the word expansion function to compute word array values starting from the 16^{th} location (for word expansion, the first 16 iterations (`0-15`) are the values already present and from the next iteration till 64^{th} location, we need to use the word expansion function to compute the values. Hence, using a word array of 16 location does the job as by left shifting, we always give the updated value the word expansion function expects each cycle, and we are computing the new value required and storing it in the last location, which keeps shifting left). From this, we can fix the values of `w` present in the word expansion function according to 16 (instead of `t-15`, `16-15=1`, and so on). The word expansion function also expects a nonce number as input, which will help parallelize future phases. In this way, for 64 rounds, the word is computed parallel, the SHA256 operation is also executed, and the return values are stored in `a-h` (nonce 0 used). Once the 64 rounds of compressing are over, we add `h0-h7` with `a-h` values correspondingly and store it back to

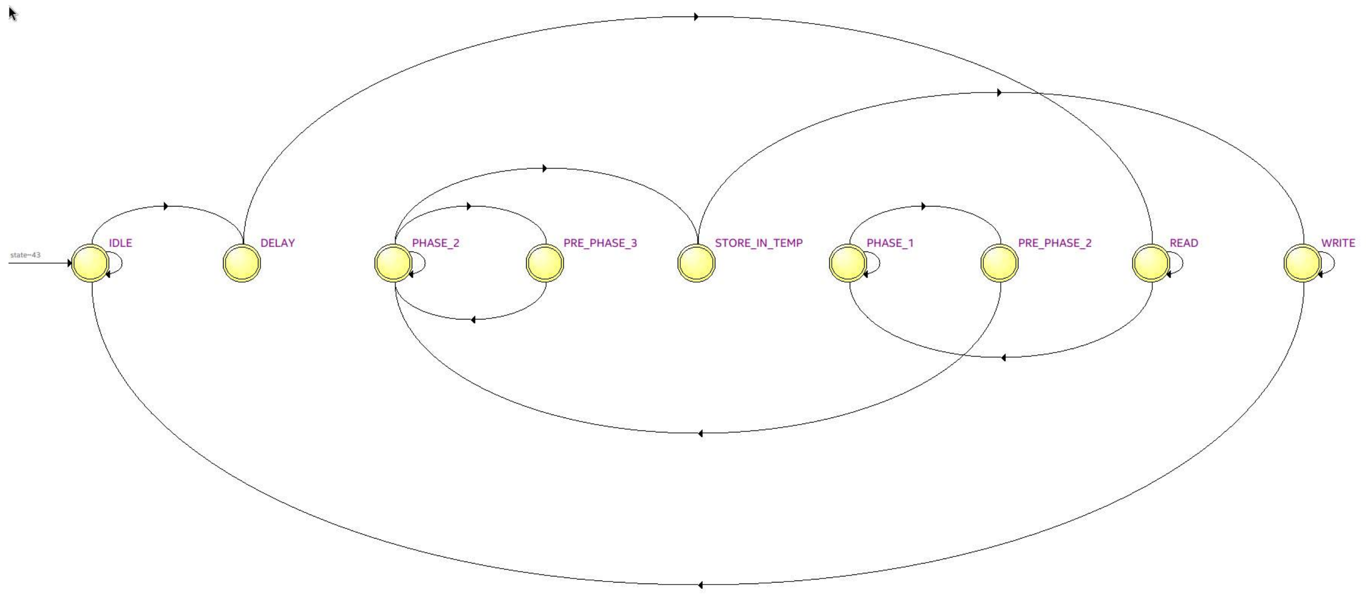


Fig. 6: State Diagram for Blockchain

h0-h7 and a-h. After that, we move to PRE_PHASE2. The waveform at the end of PHASE1 also shows the expected values which were given in the spreadsheet (H0=366afef3, H1=a92ada07, H2=c3cdbbfe and so on, all values in the waveform are matching with expected values in the spreadsheet).

- **PRE_PHASE2**: In PRE_PHASE2, for all nonce values (16 nonces, 0-15), we store the remaining message values (3 more words) in word array and the nonce value, followed by nonce value and padding as given in slides. The h0-h7 and a-h values for all nonce values are also stored with output h0-h7 and a-h values from PHASE1. We then move to PHASE2. This is visible in the waveform as well.
- **PHASE2**: In PHASE2, we do all the operations we did in PHASE1, except that now we are doing for all nonce values parallelly (this is the optimization, instead of serially doing it for each nonce one after the other, can parallelly do for all nonce values). For all nonces, we compute the word expansion values starting location 16 and the SHA256 operation simultaneously, just as in PHASE1, where we fix the word array 0^{th} location per nonce value, left shift all values of w array per nonce value and compute the word expansion value at the last location per nonce value (hence function expects as input nonce value). Once the 64 rounds of compressing are over, we add h0-h7 with a-h values for each nonce correspondingly and store it back to h0-h7 and a-h for each nonce. After that, we move to PRE_PHASE3(as the SHA computation for PHASE is not yet done, as indicated by sha_comp_done signal). The waveform at the end of PHASE2 also shows the expected values which were given in the spreadsheet (H0[0]=bdd2fbd9, H0[1]=080c8778 and so on, all values in the waveform are matching with expected values in the spreadsheet).
- **PRE_PHASE3**: In PRE_PHASE3, for all nonce values (16 nonces, $0-15$), we store the corresponding h0-h7 values from PHASE2 in the first 8 locations of word array(per nonce) and remaining words are filled with padding as given in slides. The h0-h7 and a-h values for all nonce values are also stored with init h0-h7 and a-h values. Since PHASE3 operations are the same as PHASE2, we are reusing PHASE2 state for PHASE3 computations. Hence, we next move to PHASE2 state. This is visible in the waveform as well.
- **PHASE2**: In PHASE2 now, we again perform 64 rounds compression (as we did before for PHASE2) with the newly loaded values of word array and h0-h7 and a-h values. The SHA256 operation for all nonces happens in parallel, and word expansion also happens in parallel, just as in previous phase 2 (For all nonces, we compute the word expansion values starting location 16 and the SHA256 operation simultaneously, where we fix the word array 0^{th} location per nonce value, left shift all values of w array per nonce value and compute the word expansion value at the last location per nonce value (hence function expects as input nonce value). Once the 64 rounds of compressing are over, we add h0-h7 with a-h values for each nonce correspondingly and store it back to h0-h7 and a-h for each nonce). At this stage, since we are done with all the phases (sha_comp_done is made high in the previous PRE_PHASE3 state), we move to the STORE_IN_TEMP state. The waveform at the end of PHASE2 now also shows the expected values which were given in the spreadsheet (H0[0]=7106973a, H0[1]=6e66eea7 and so on, all values in the waveform are matching with

expected values in the spreadsheet).

- **STORE_IN_TEMP**: In STORE_IN_TEMP, we set the cur_we signal as high (to tell we want to start writing), and the address we set as the output address starting at which we start writing the computed values. We are writing only the H0[0:15] values of nonce 0 back to memory (as indicated in slides). Here, we write the first location as H0[0] and then move to the WRITE state.
- **WRITE** : In the WRITE state, we increment the offset and write all the remaining values cycle by cycle. This is visible in the waveform as well, where we first write (7106973a, then 6e66eea7 and so on through mem_write_data, which is the output). Once all the values are given as output, we return to the IDLE state and wait for the start signal to carry out the next set of operations. Since we move back to IDLE state, the done signal is also asserted, saying that the bitcoin hashing is complete.
- The transcript at the end also shows that the computation was successful and matches the expected values from TB. At the intermediate stages, we also verified the values in the waveform with the expected values in the spreadsheet.

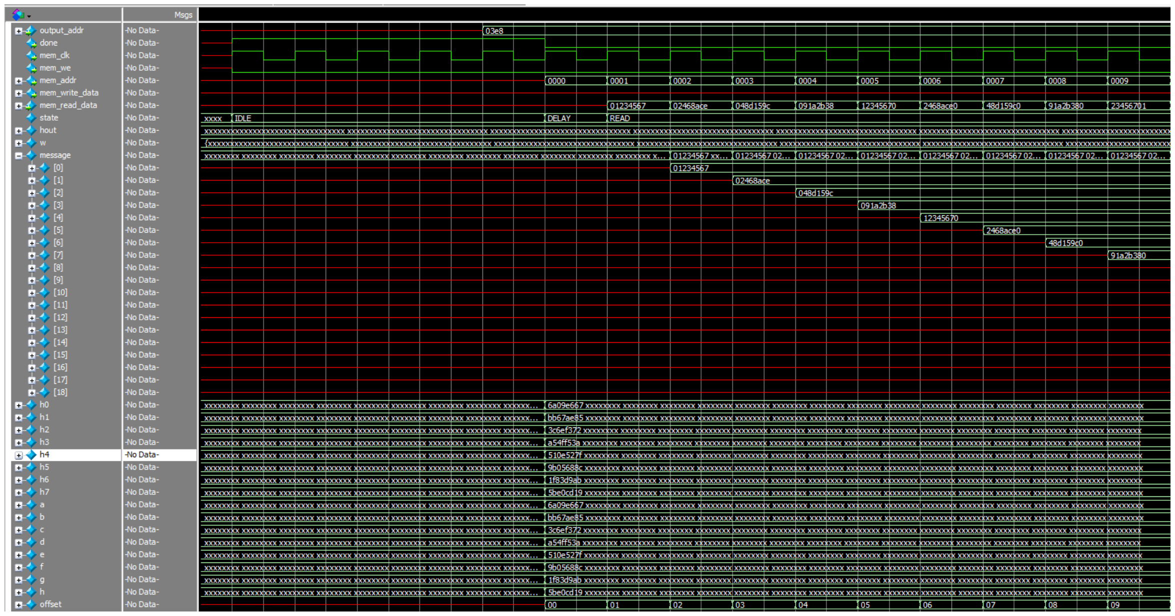


Fig. 7: Enter Caption

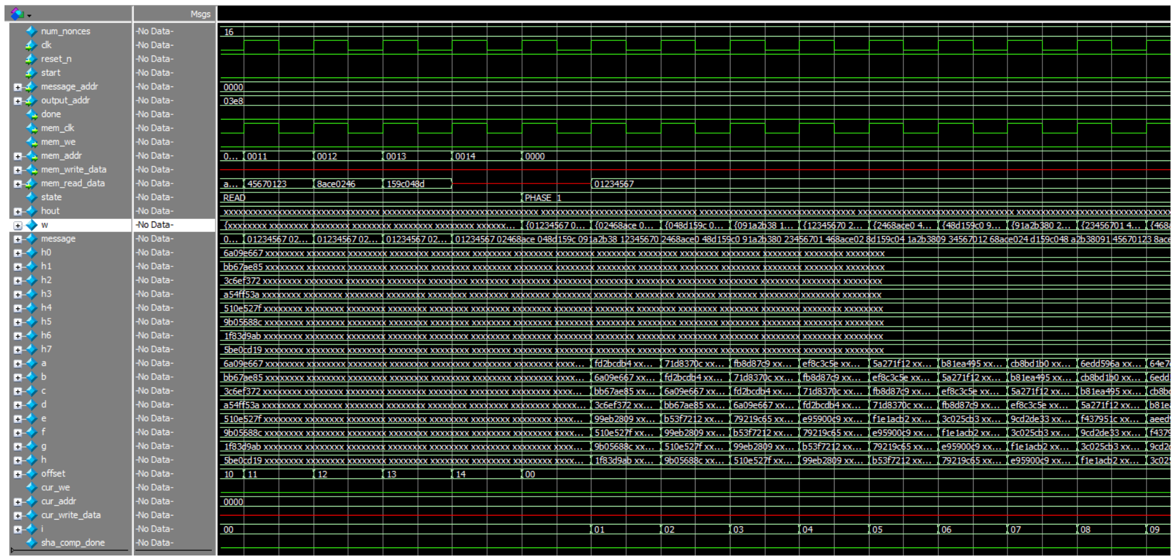


Fig. 8: Enter Caption

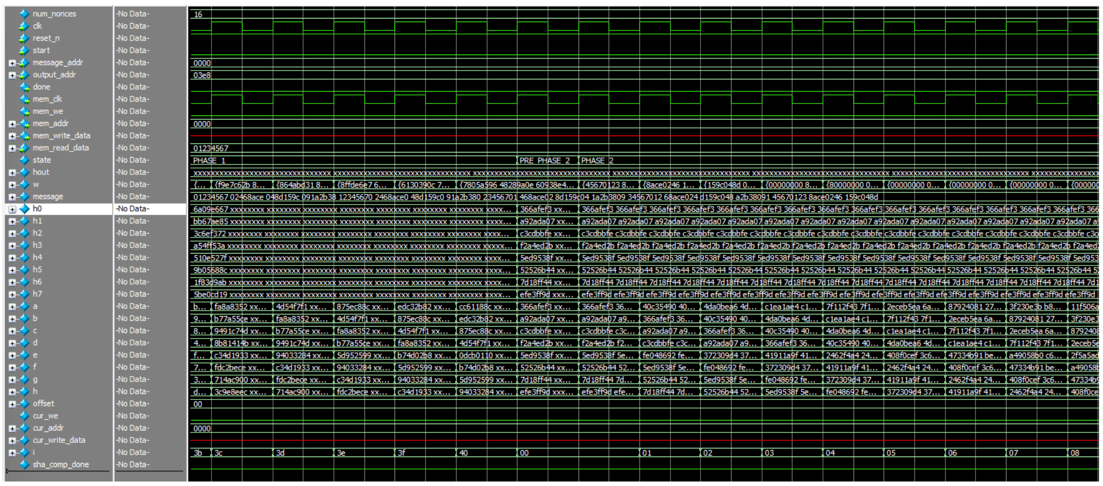


Fig. 9: Enter Caption

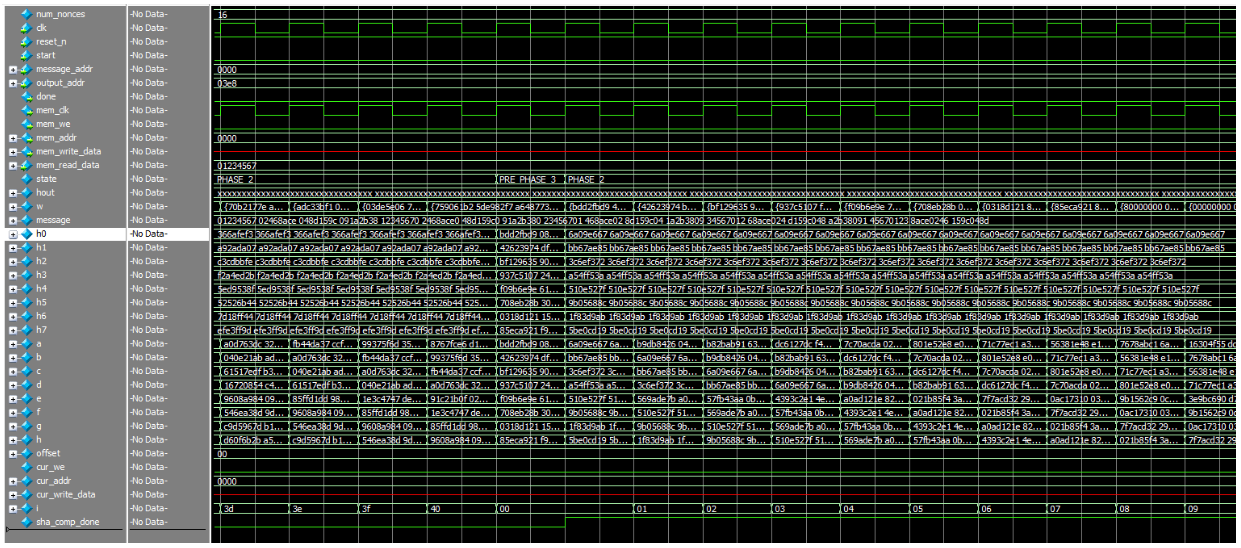


Fig. 10: Enter Caption

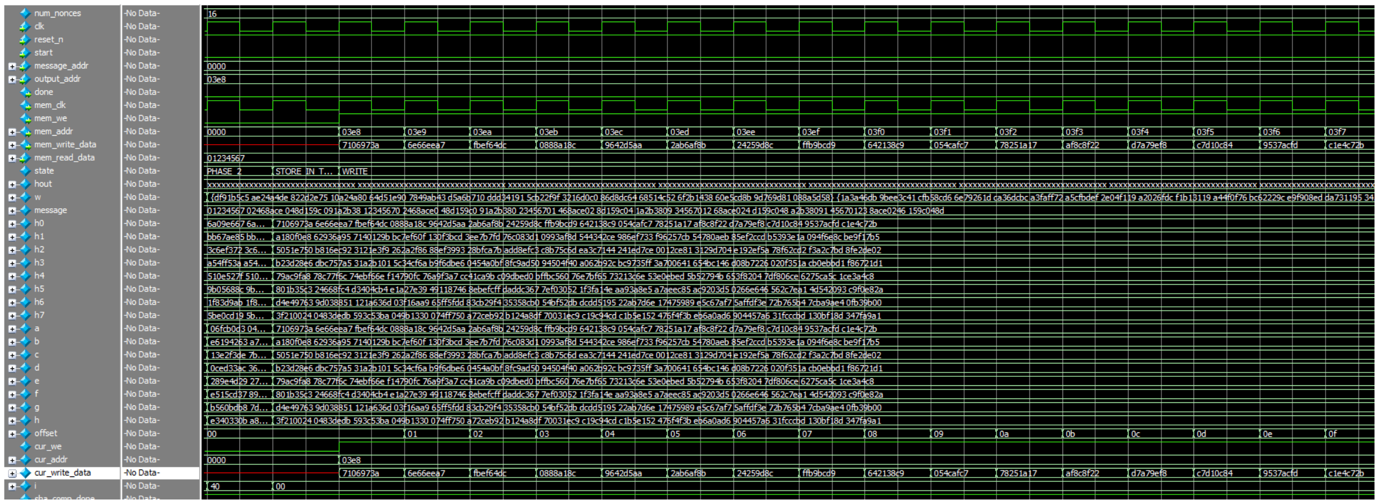


Fig. 11: Enter Caption