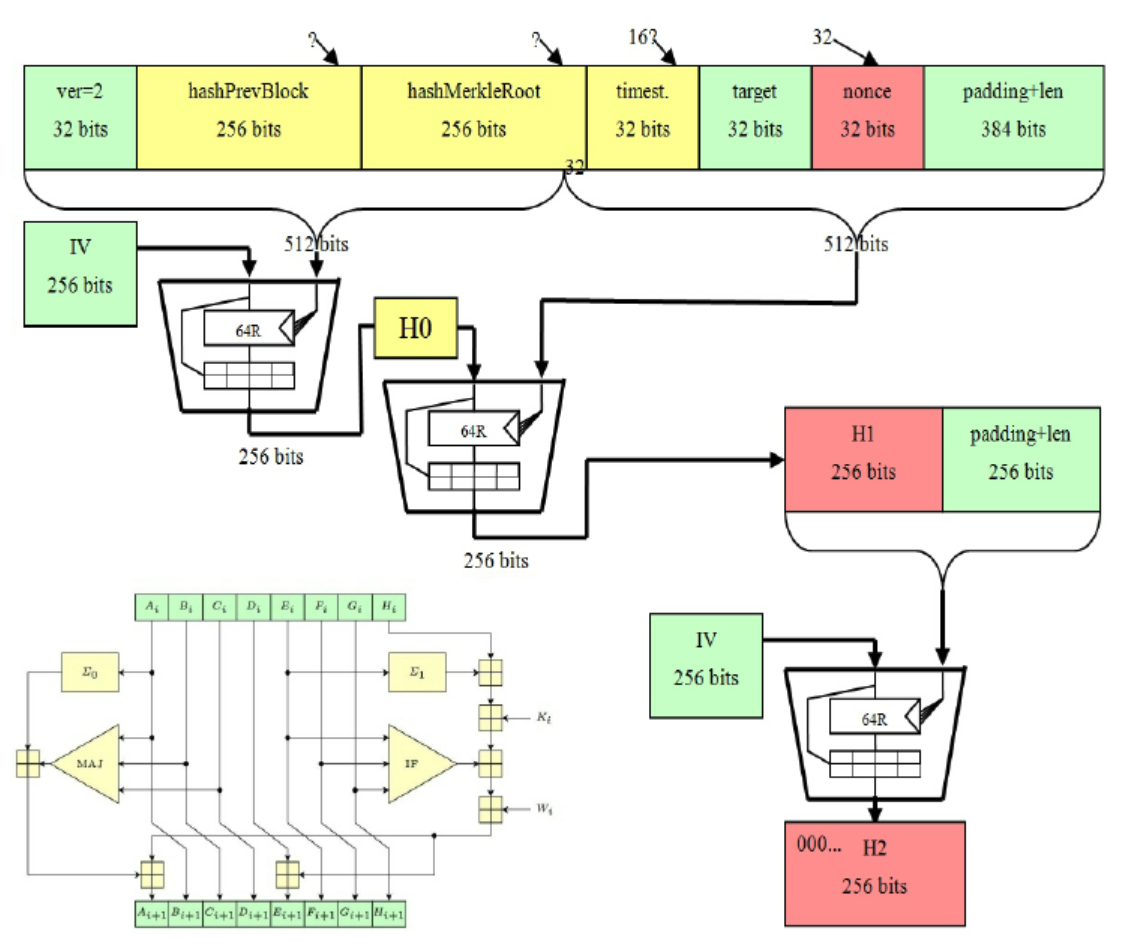
**ECE 111 Final Project**

**SHA256 and bitcoin hash introduction[[1]](#footnote-0):**

Our project utilises SHA256, a ‘secure hash algorithm,’ as a means of cryptographically converting input data--of which varies in size and type--into output data of a fixed size, as a unique hash value. The object of SHA is to produce a *unique* hash value given any proper input data. Our particular algorithm converts a 20 word message into a hash composed of eight 32-bit words. This hashing function demands that the following properties be satisfied: compression (output is static number of characters irrespective of input size), avalanche effect (a very small change in input ought to dramatically alter the output hash), determinism (identical inputs must produce identical outputs, irrespective of the system upon which it operates), pre-image resistant (the hashing algorithm should be a unidirectional function and non-reversible), collision resistance (no two inputs produce the same output), and efficiency (fast computational speed). SHA-256 has as its applications two common uses: that of authentication, in which the goal is secrecy, and that of file integrity verification, in which the goal is integrity.

The second part of our project takes the form of bitcoin hashing, a different type of hashing algorithm that, *via* a chained SHA-256 algorithm, similarly takes an input of nineteen words with one nonce value, and outputs eight 32-bit words. A blockchain, understood as a chain of digital data blocks, is chained by hashing algorithms like the aforementioned SHA-256. These immutable chained blocks contain information regarding the nature of financial transactions, and they are publicly available as a ledger, all of which are dependent on hashing.



**SHA256 algorithm explanations:**

For context, the compression function has as its first step a word expansion function, followed by a SHA-256 operation (from t=0 to t=63, for a total of 64 times). The first step in the algorithm is appending a single 1 to the end of the input message, then pad this message with 0s until the input data is 64-bits less than X, where X is a multiple of 512. After this preprocessing, it is necessary to append the message length bits to the 0-63 bit position, since SHA-256 allows an input message up until . Now, H0:H7 are initialised to outpush hash to eight 32-bit words (e.g., H7=5be0cd19), after which the algorithm may be implemented, wherein the message is divided into segments of size 512-bits, each of which are individually processed in order. We set the input as Kt, an array (constants), and Wt, as a 32-bit word from the message, along with H0:H7 as the current message digest, and the output as H0:H7 as the new message digest. Prior to processing each 512-bit message segments, we initialise A:H to H0:H7, which is followed by the 64 operations of 512 bit blocks, as outlined above. t is bounded from 0 to 63--hence the 64 rounds of processing--and if t is less than sixteen, Wt is set to the 32-bit word of that particular 512-bit message segment, while if it is between sixteen and sixty three (inclusive), then we perform right rotate (and XOR) operations: , .

We then initialise the 64 constants, provided in the starter code. For *each* t from t=0 through t=63, the following operations are performed:

After every single one of these steps--for a total of 64 steps--are finished processing, we set:

The final step involves the output. After the message that has been divided into segments of size 512-bits have each been individually processed in order as described above, H0:H7 contains the 256-bit hash of M.

**Bitcoin hash algorithm explanations:**

For context, we use SHA-256 to create each block’s output hash value, with two blocks created for Wt, the input message (of twenty words). We use a nonce value of 16.

Normally, we would try new nonce values, incrementing upwards until we finish at the point where hash is less than the target goal, but here we have fixed the number of iterations to 16, so there is no need to check against the target. The first step of three is to process block 1 of the first SHA-256 function, in which we set H0:H7 to constants and Wts correspond to the first 16 words in memory, only to move on to step two, in which we process the second block of the first SHA-256 hash function (H0:H7 from the previous step are used and Wts align with the last three words in memory, nonce, 32’h80000000, ten 32’h00000000 padding, and 32’d640). The third step is to process the second SHA-256 hash function, wherein H0:H7 align with constants, and Wts are set to H0:H7 from the previous phase, 32’h80000000, six of 32’h00000000 padding, and 32’d256. This process makes 16 final hashes, since this second and third steps are completed for 16 rounds.

In terms of the block header, the final hash is computed recursively, with SHA256[SHA256(our message)] for 16 nonces, and each message is set to {block header, nonce}. This will make 16 hashes, as mentioned above:

In terms of specifics,

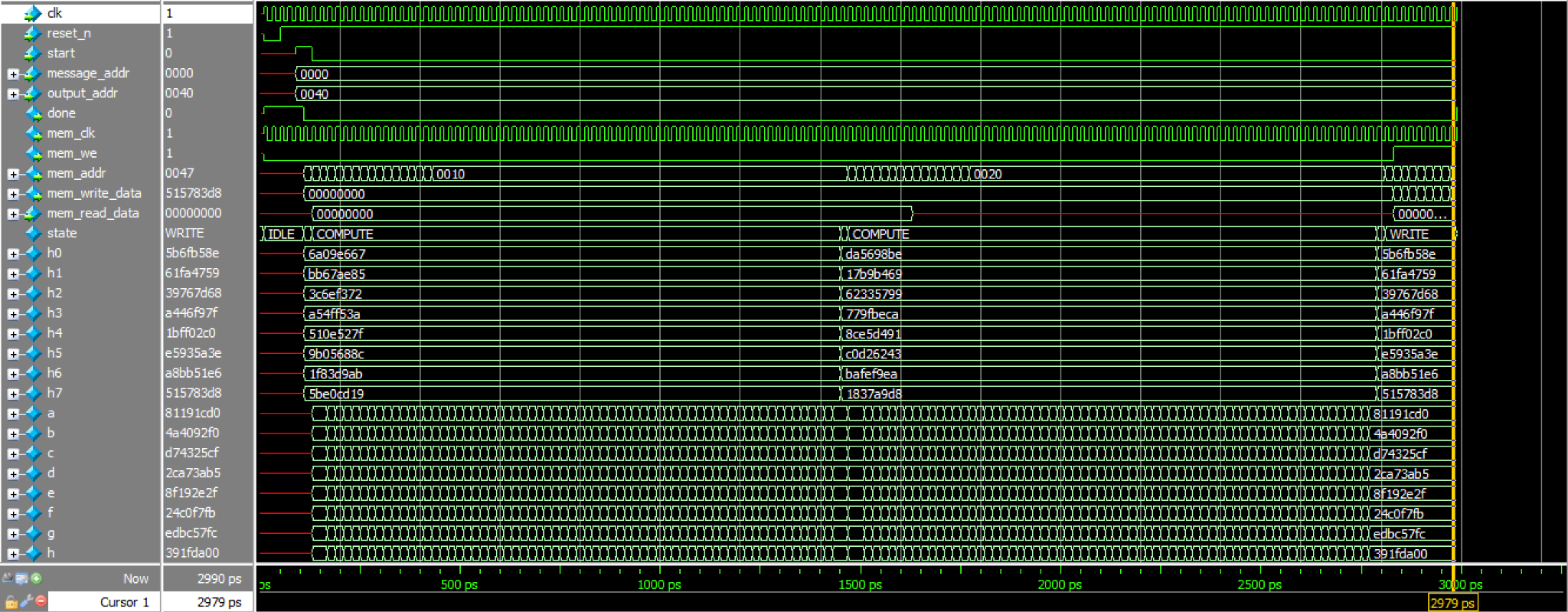
For bitcoin hashing we create 2 module called pblock that will be called in the top module called bitcoin\_hash.

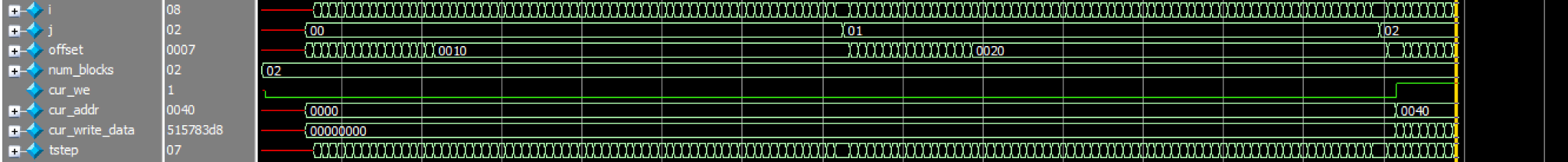
In state IDLE, we initialize the memory address and variable for incrementation(t) as well as setting the variable mem\_we to 0 in order to read from the message.

In the PREP11, PREP12, PREP13 states, the function of these states is to prepare/pull the data from memory to prepare for the COMPUTE1 state. In PREP11, we increment mem\_addr by 1 and proceed to the next state, PREP12. In PREP12 variables such as k, t, and mem\_addr are being incremented. At the end of each of these states, the incremented values are used in p block code for hashing. In COMPUTE1 state, first 16 word are used as initial hash values

Similarly the process is repeated in PREP21, PREP22, PREP23, and COMPUTE2 states. In this stage of bitcoin hash, the remaining words are taken in combining with the output hash results from the first compute. This result is passed to the file COMPUTE3 state of the computation stage. Once the computation is finished we proceed to the WRITE stage where we switch mem\_we signal to 1 for writing to memory.

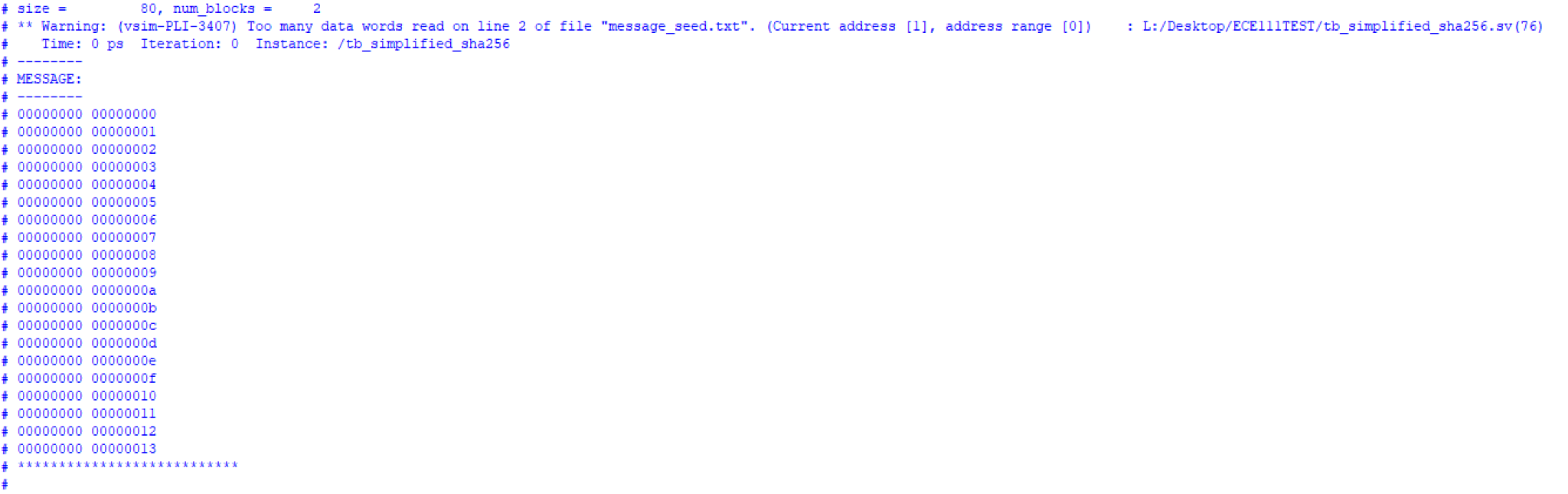
**SHA256 Simulation Result**

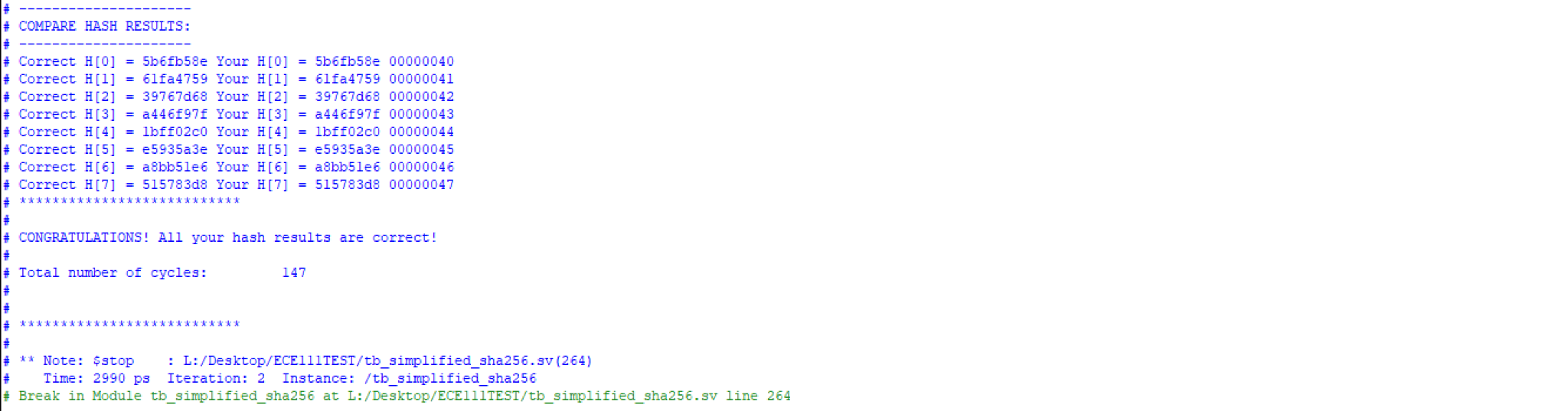
****

****

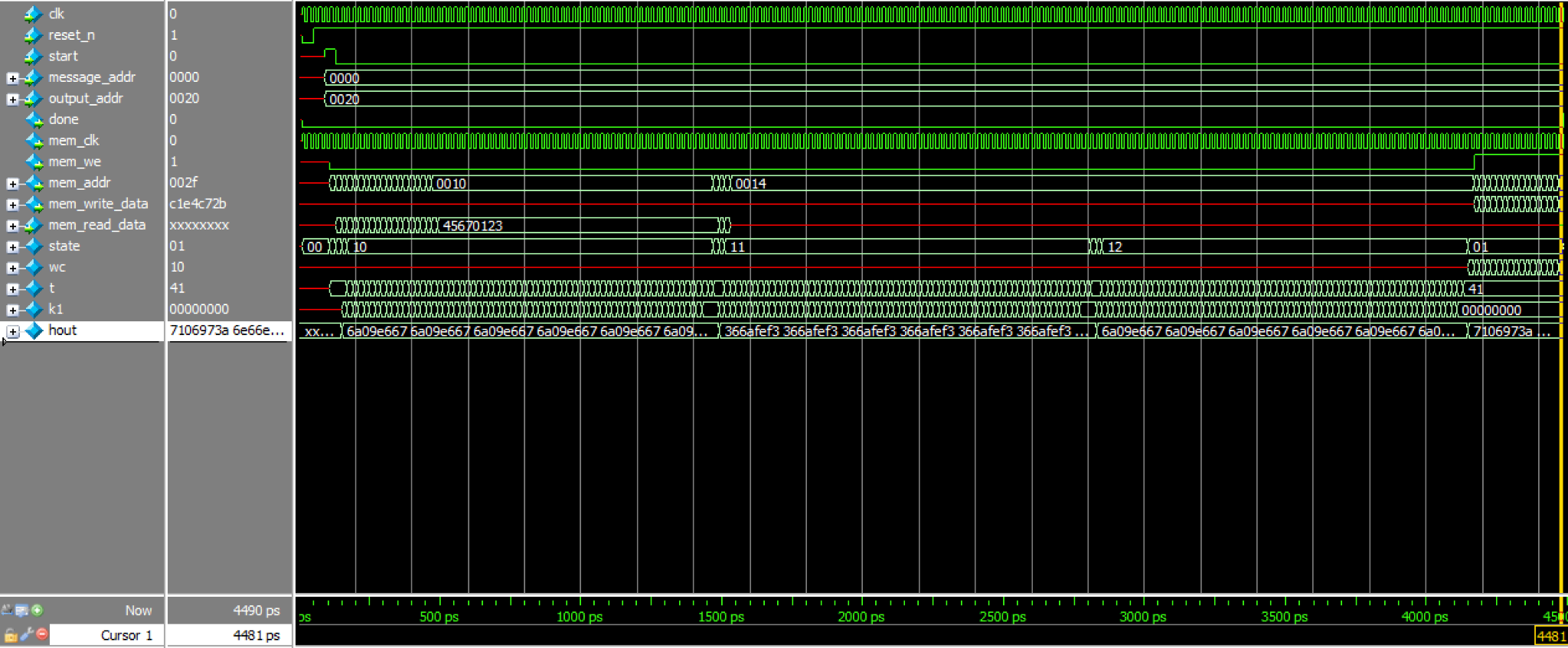
**Explanation:** the simulation result shows the correct hashes result as the transcript output from the code also confirms this fact. The output hashes in var h0:h7 during the WRITE state correspond to the expected output hash values listed in the transcript.

**SHA256 Transcript**

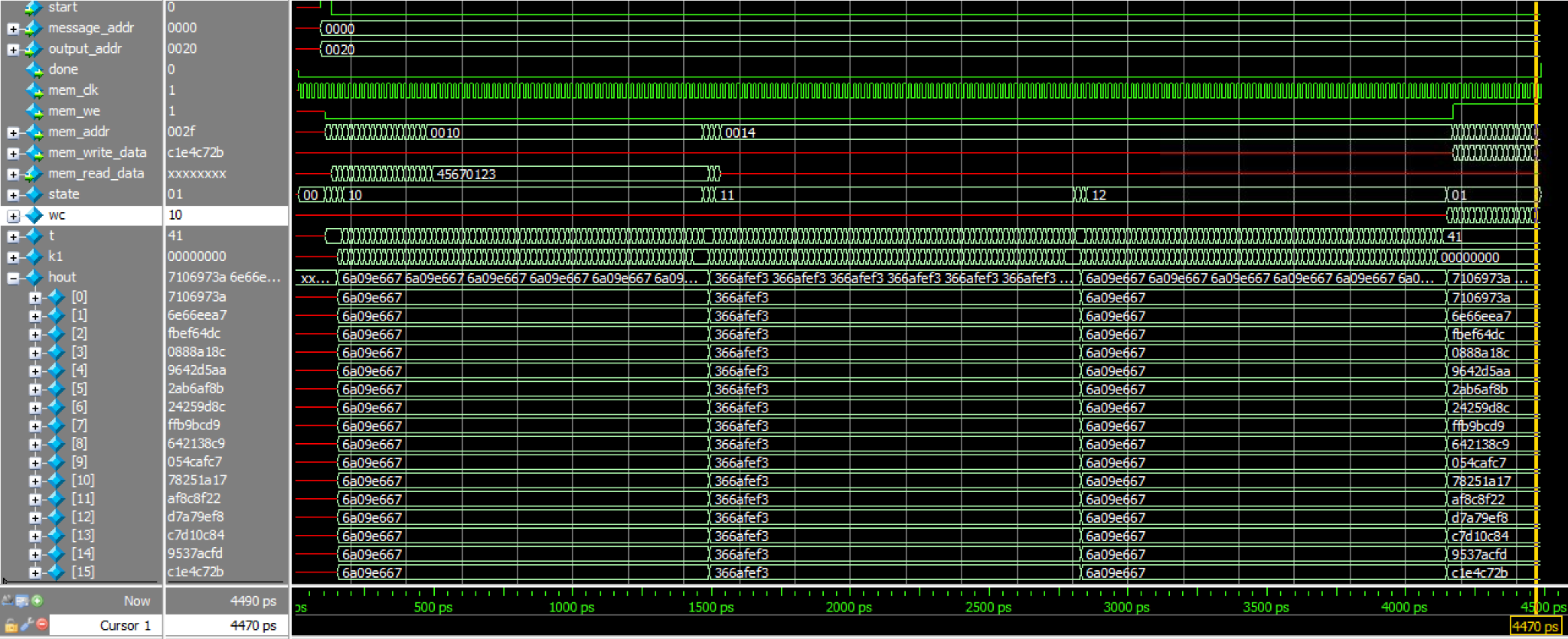
****

****

**Bitcoin Hash Simulation Result**

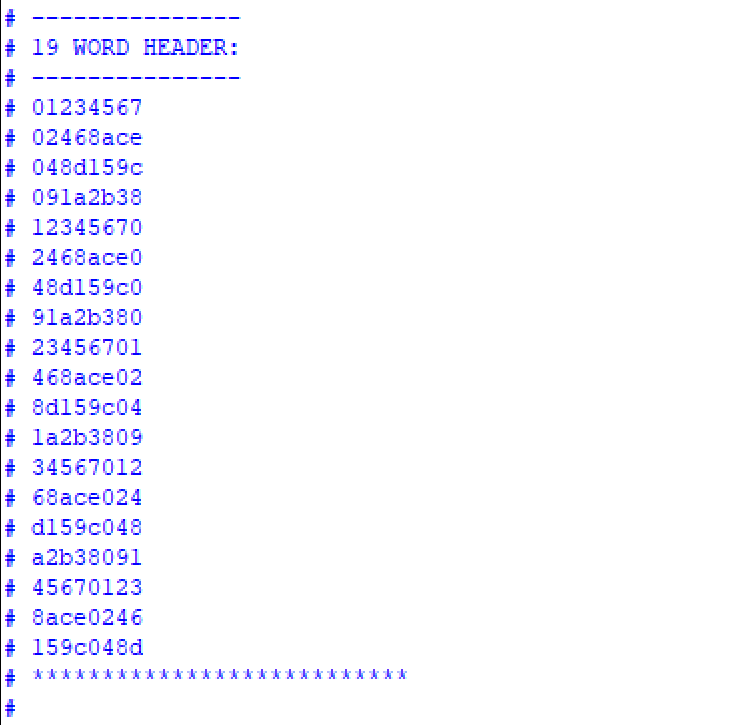
****

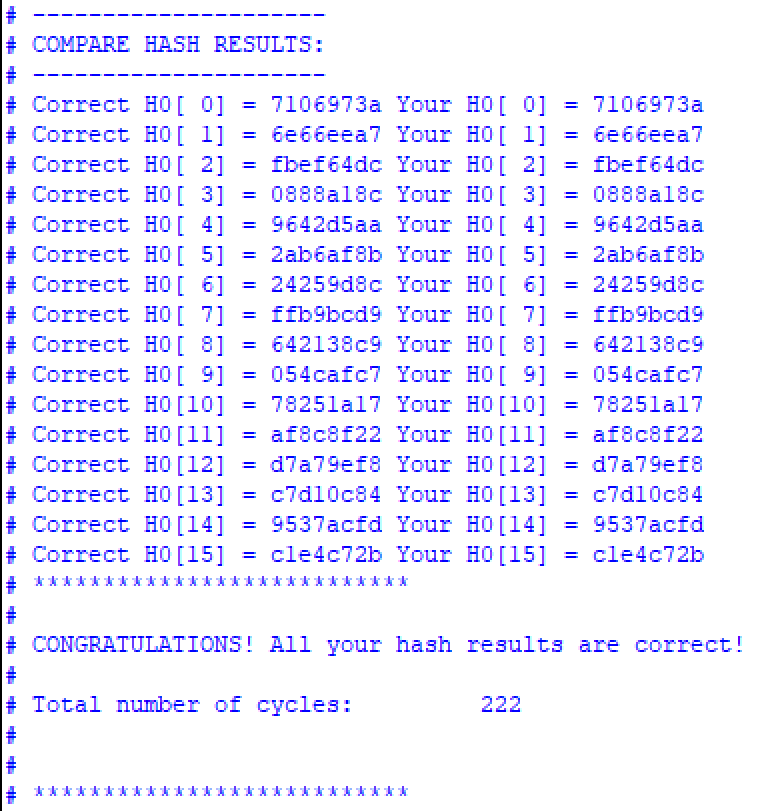
Continued below:

****

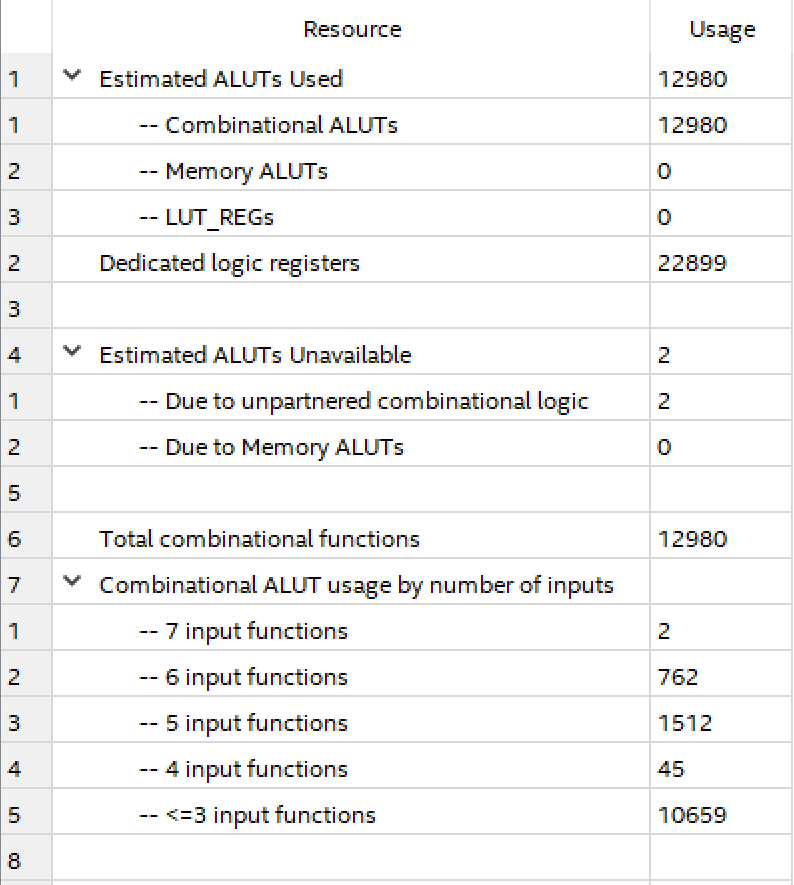
**Explanation:** As seen in the simulation result above, the final output hash results correspond to the expected hash result from the transcript. As seen in the capture, hout[0]:h[15] changes through 3 computations as in state compute1, compute2, compute3 (state: 10,11,12).

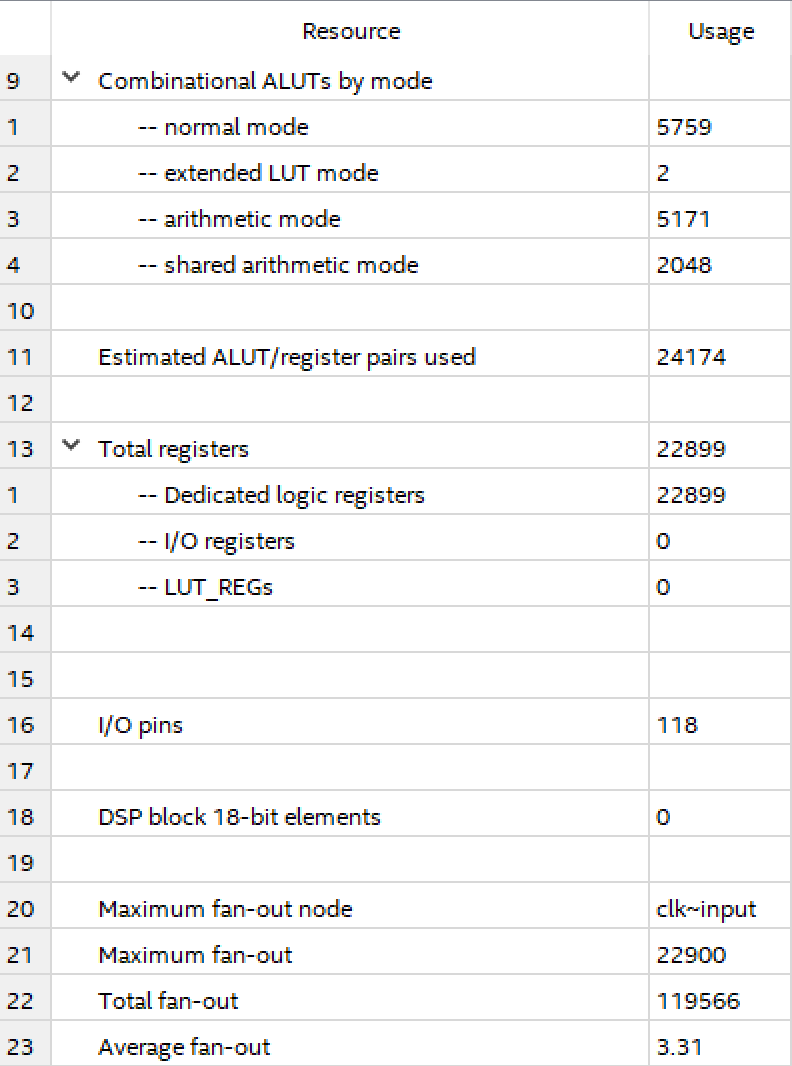
**Bitcoin Hash Transcript**

****

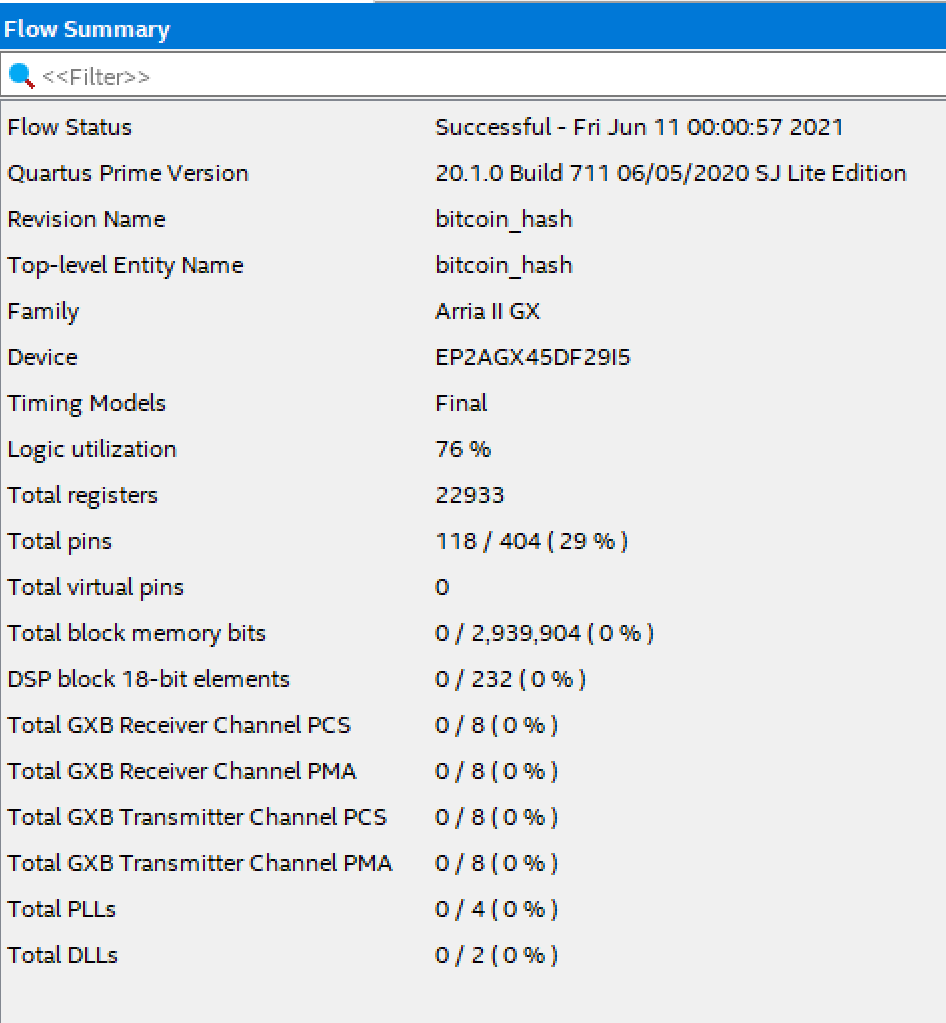
****

**Bitcoin Hash Summary Resource**

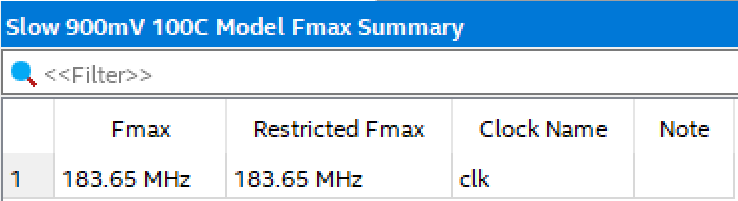
****

****

**Bitcoin Hash Flow Summary**



**Bitcoin Hash Fmax**



1. The relevant information contained within was sourced from the lecture slides. [↑](#footnote-ref-0)