ECE 111 Final Report

Nicholas Cronin (A16659831) & Henry Liu (A16239901)

Professor Farinaz Koushanfar - A01

12/16/2023

SHA256:

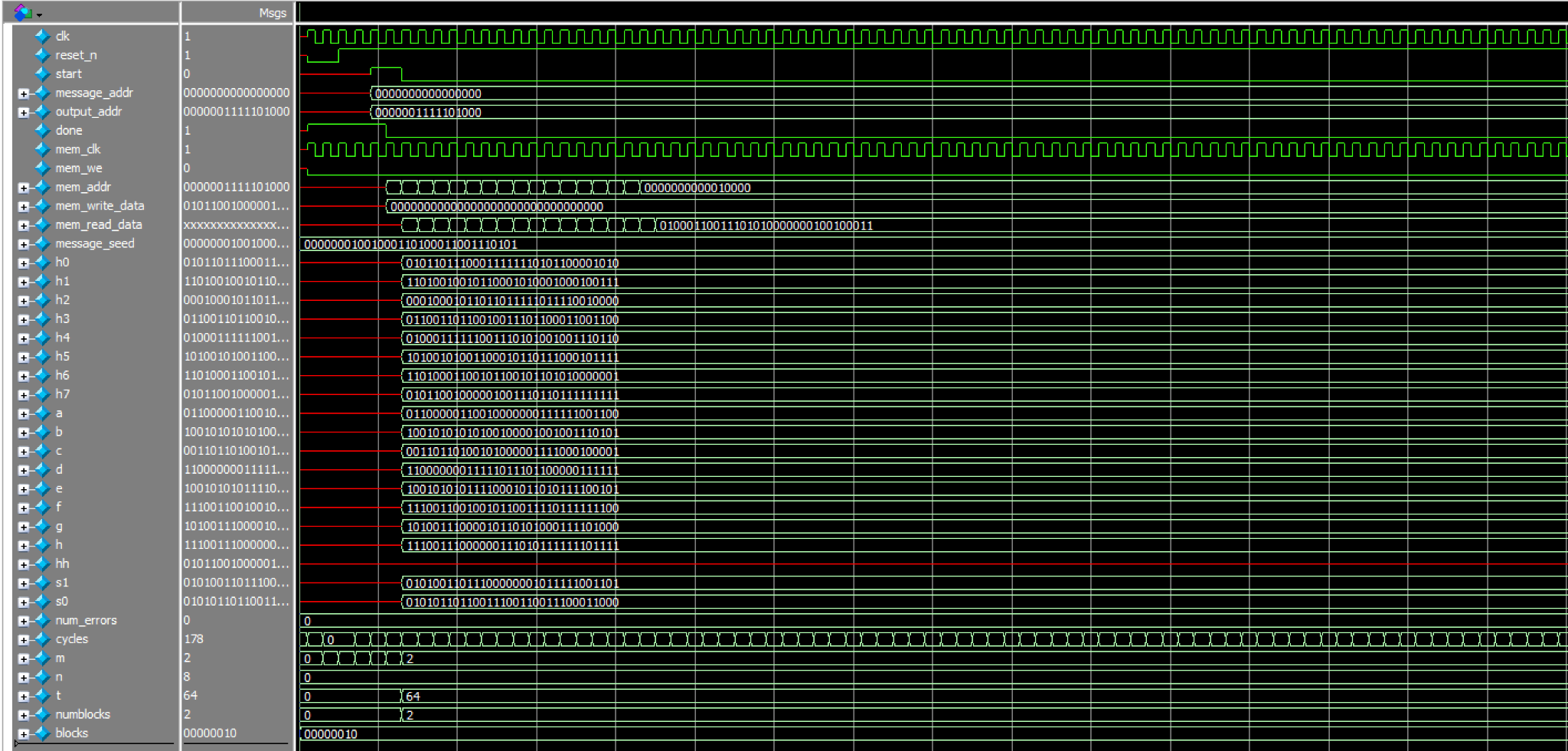
*Explanation:*The SHA-256 hash algorithm is a hashing algorithm with 256-bit outputs. It is a deterministic, one-way, fixed-output, collision-resistant algorithm that is used in bitcoin blockchaining. The hashing algorithm functions by reading in an input message 512 bits at a time and partitioning those bits into 16, 32-bit blocks called words. It then runs a hashing algorithm on those words 64 times. After running the hashing algorithm on the words, it saves the current hashed output and uses that to compute the hashed output of the next 16 words. Once the message ends, the SHA algorithm will append the necessary padding and the length (in bytes) of the message to the last 512-bit block, hash that last block, and output the final hash.

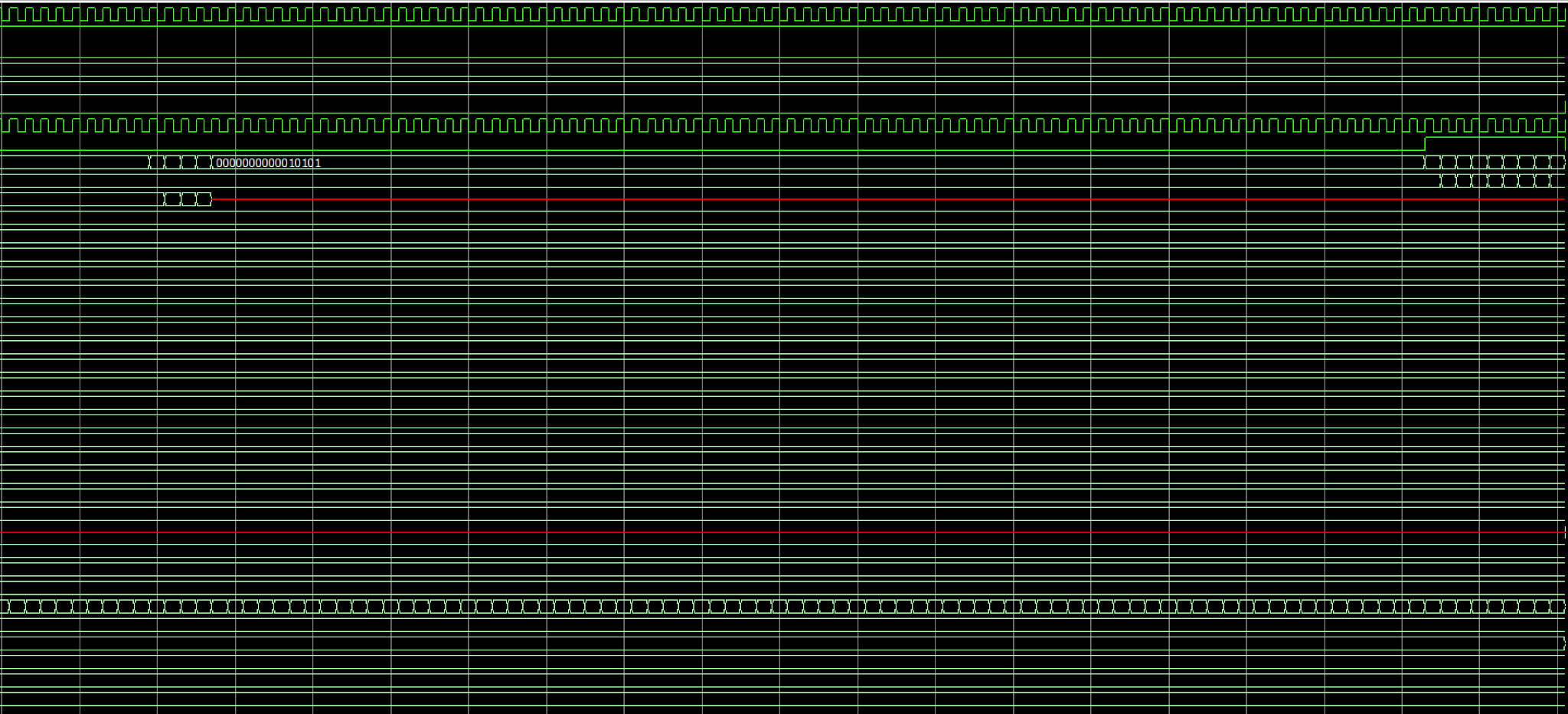
*Algorithm:*

We implemented our algorithm using a finite-state machine that splits up the process described above into many different states of execution. First, we have the IDLE state, where we set up any necessary variables to their initial states (initial hashing constants, counters, and tracking variables) and wait for the start flag. Once we receive the start flag, we move to the READ state, where we read in the first 512-bit block from our input and partition that block into 16- and 32-bit words. If there are less than 512 bits left to be read, we add the necessary padding and the length of the input to the end of the block and mark that we are finished reading. After this, we enter the block state, where we set up our algorithm for hashing, and proceed to compute, where we hash our current block using our current hashing variables 64 times, complete with optimized word expansion for a more efficient algorithm. After COMPUTE, we save the new hashing outputs and go back to READ to continue the process. If we have already read all input bits, then READ will transfer to the WRITE state, where we write the hashed outputs from the COMPUTE state to memory and mark that we have finished execution.

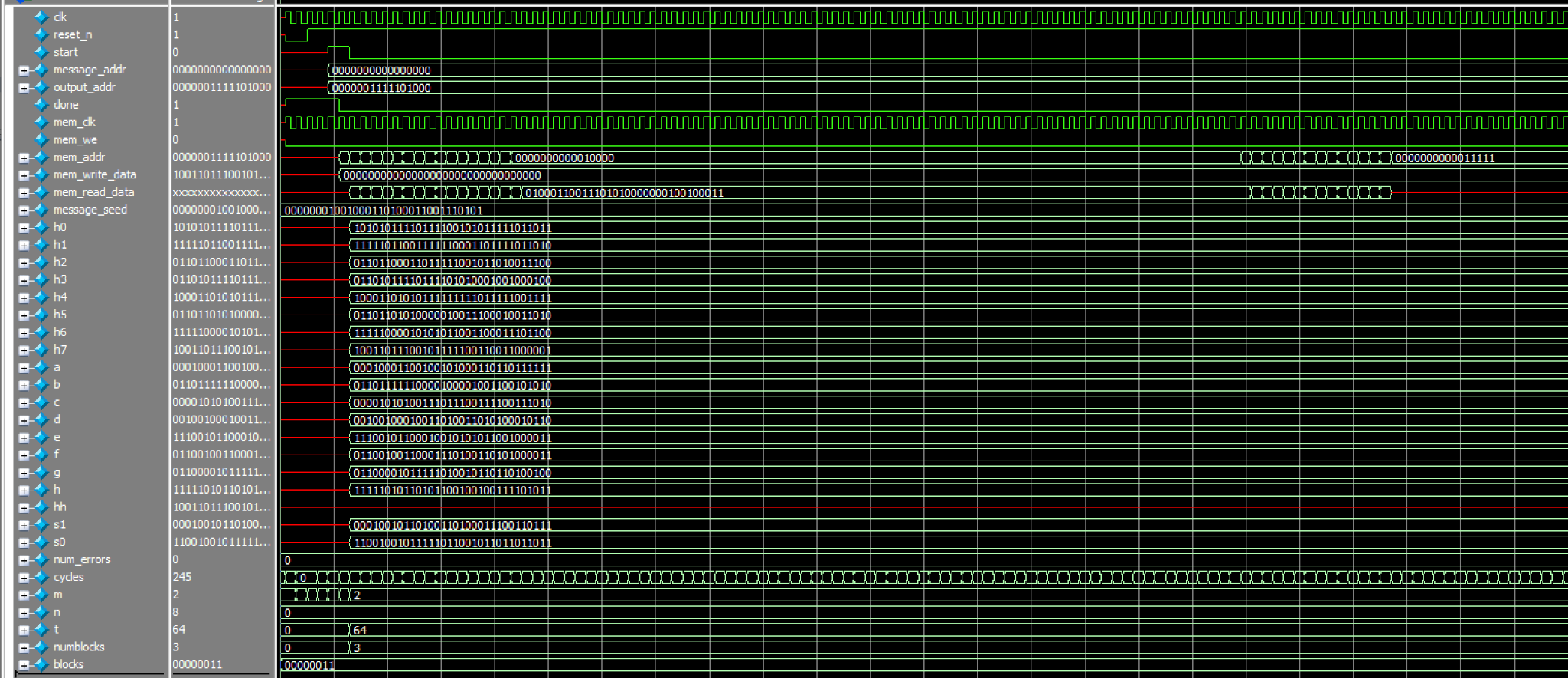
*Simulation Waveforms:*

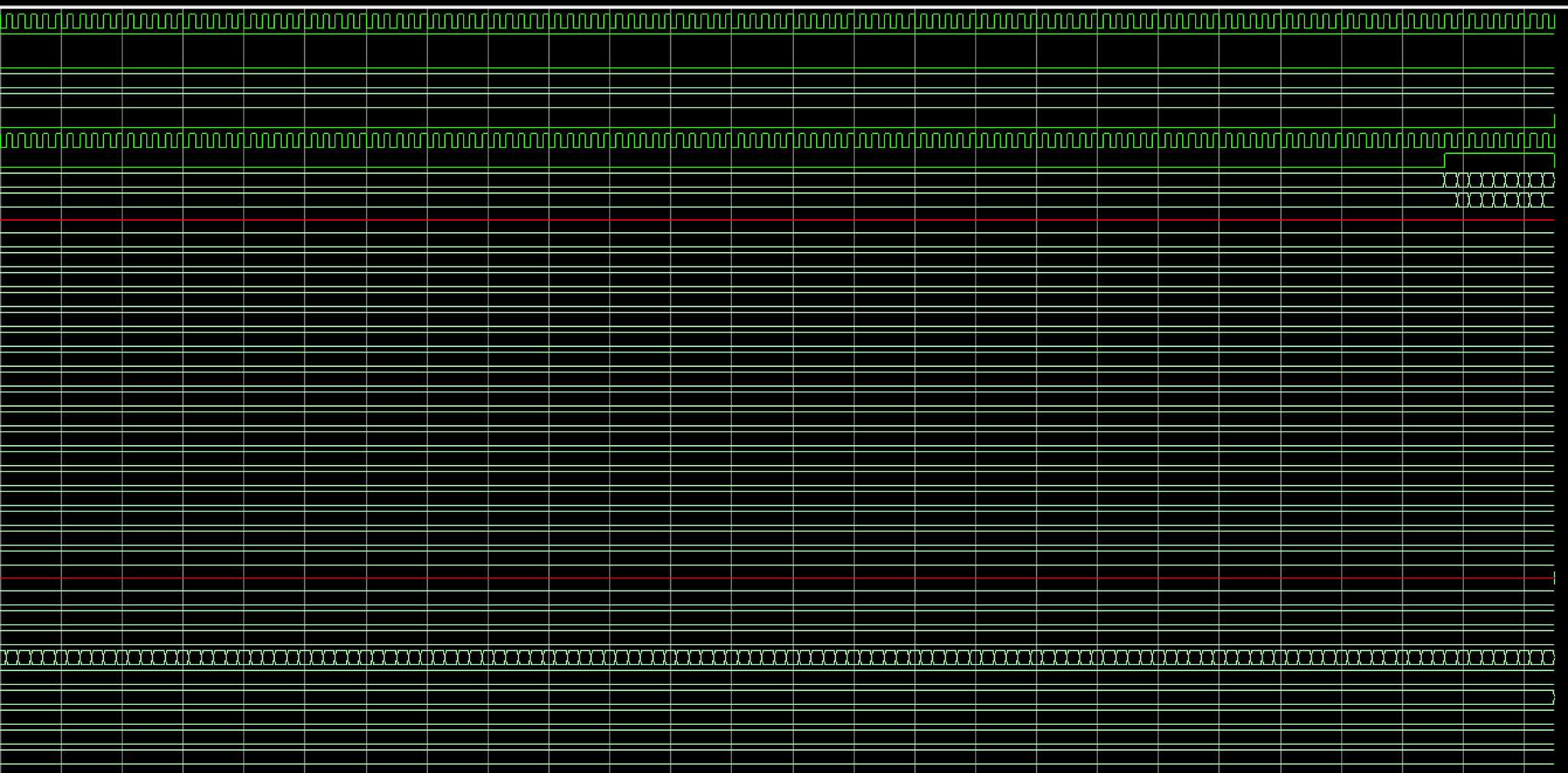
20 Word Testbench



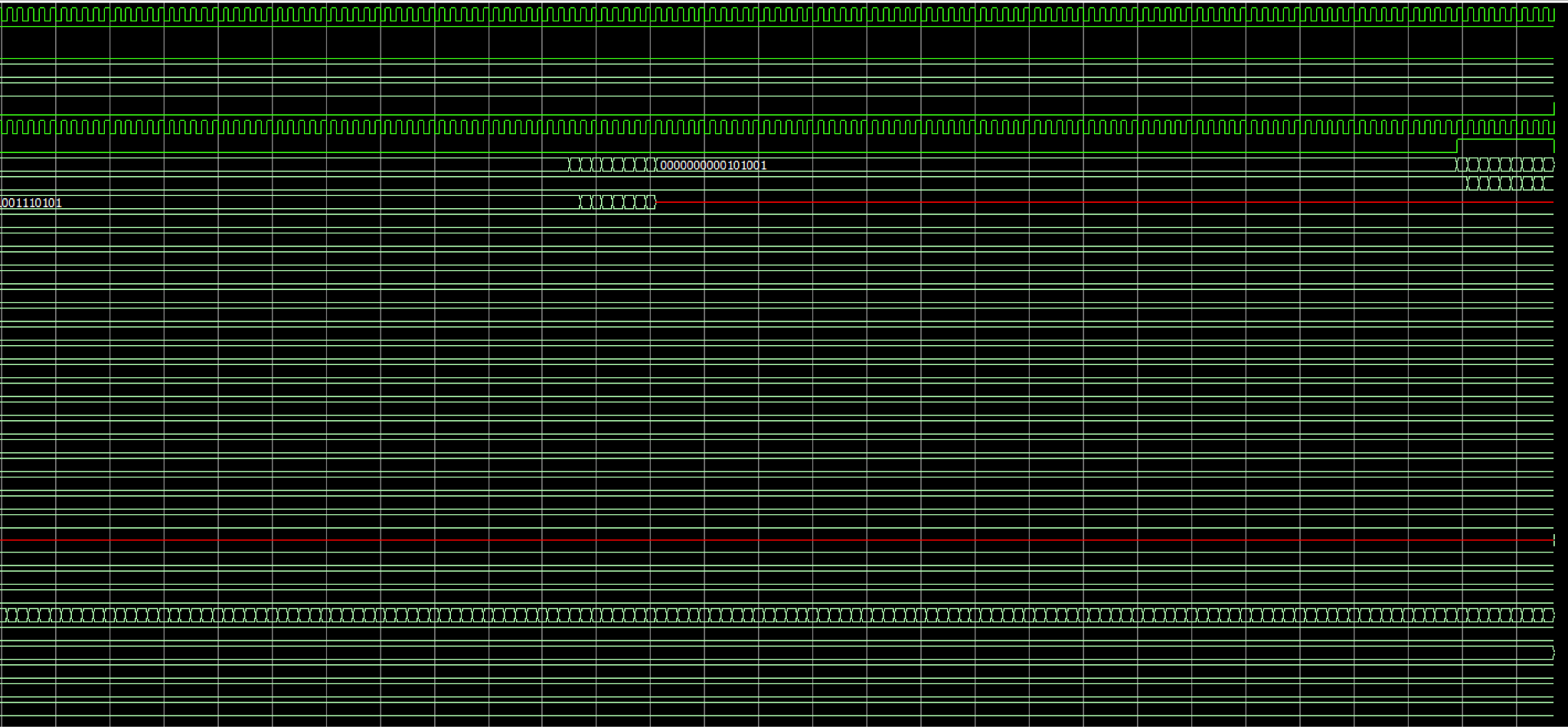
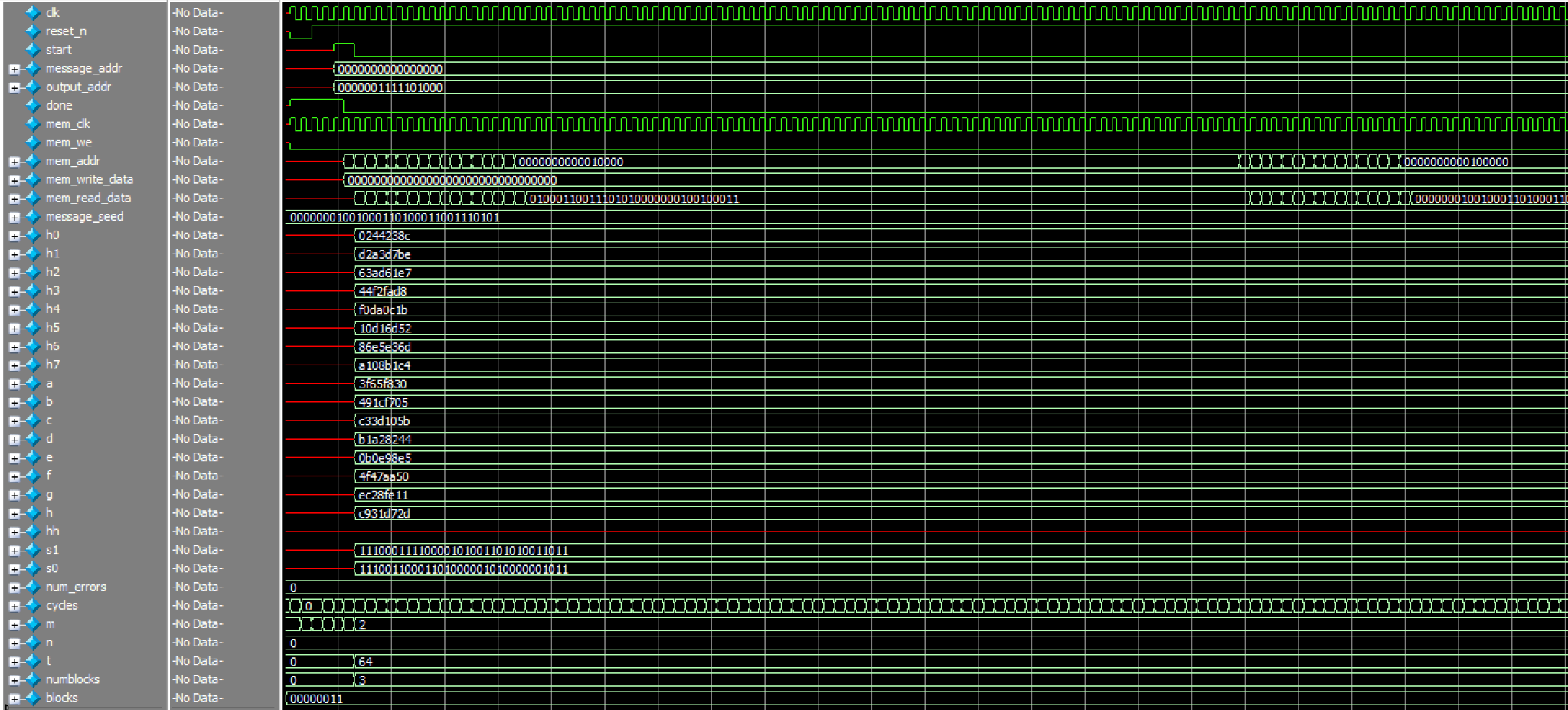


30 Word Testbench



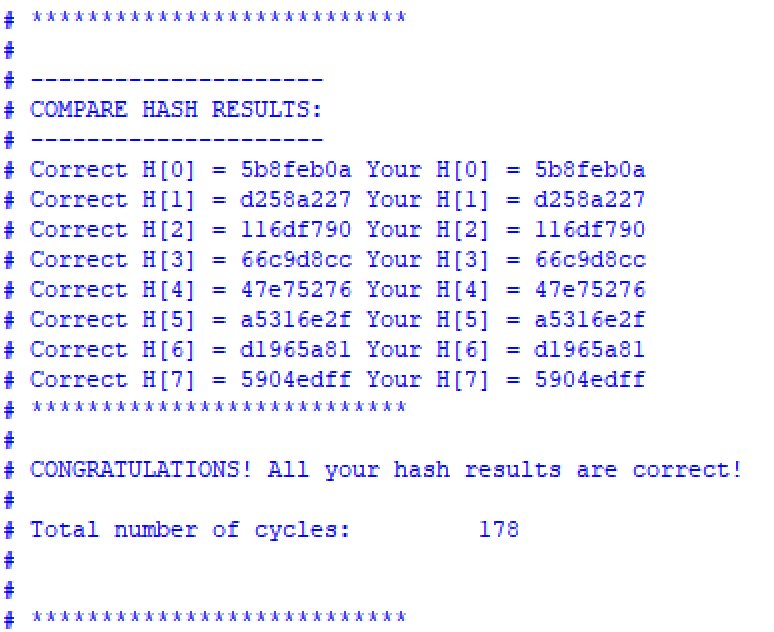


40 Word Testbench

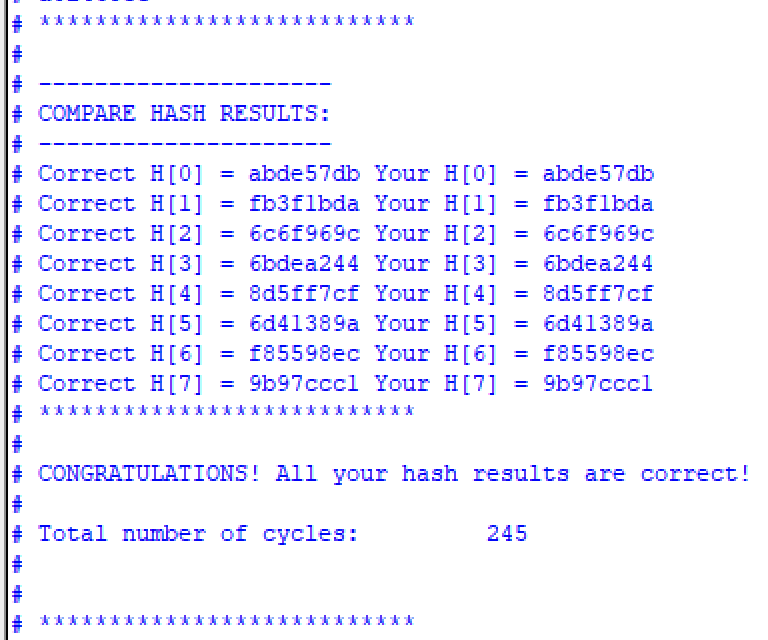
**

*ModelSim Transcripts:*

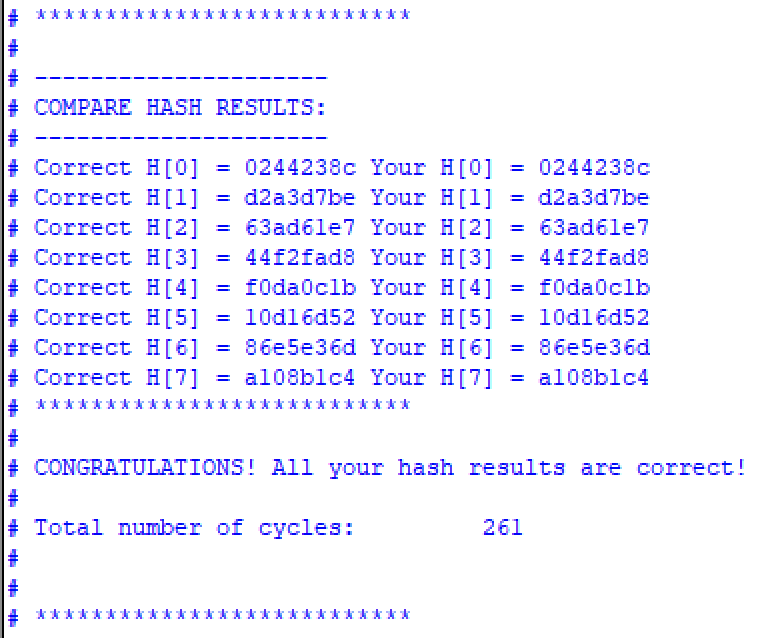
20 Word Testbench



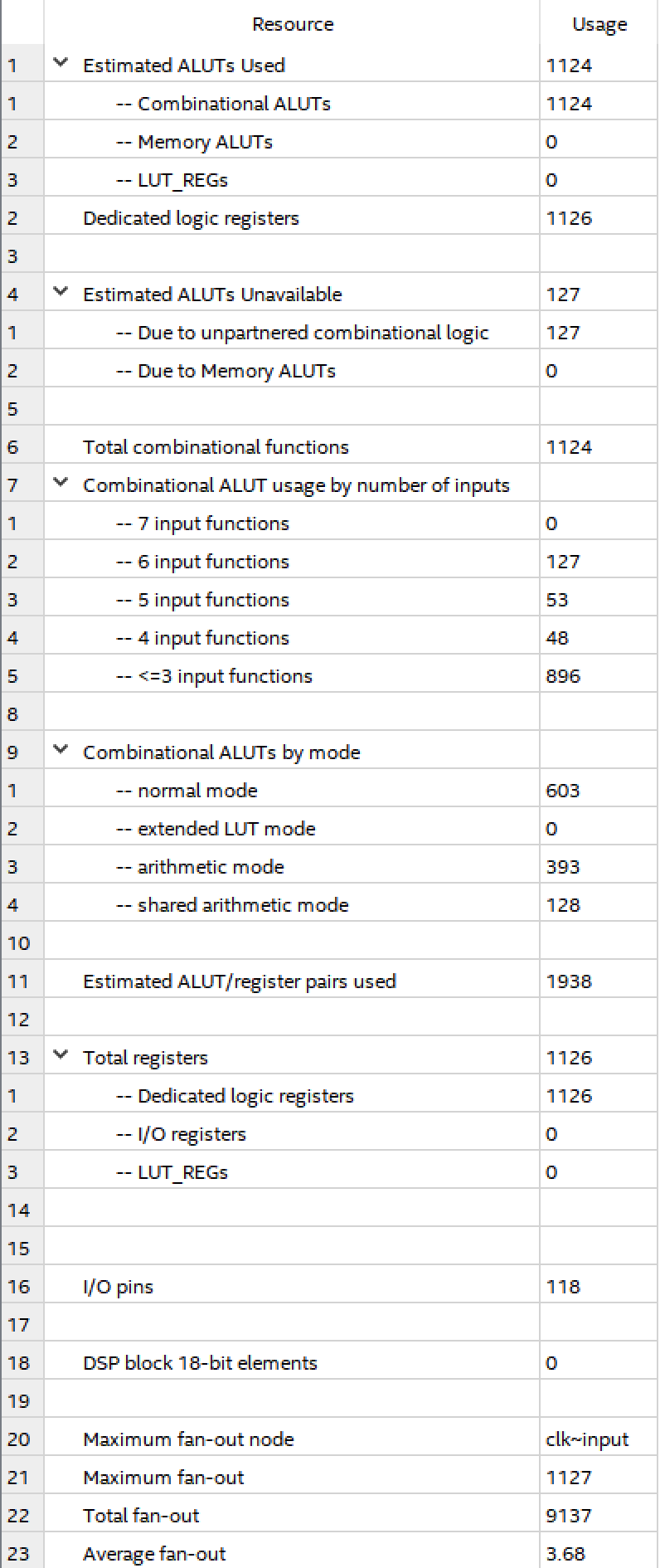
30 Word Testbench



40 Word Testbench

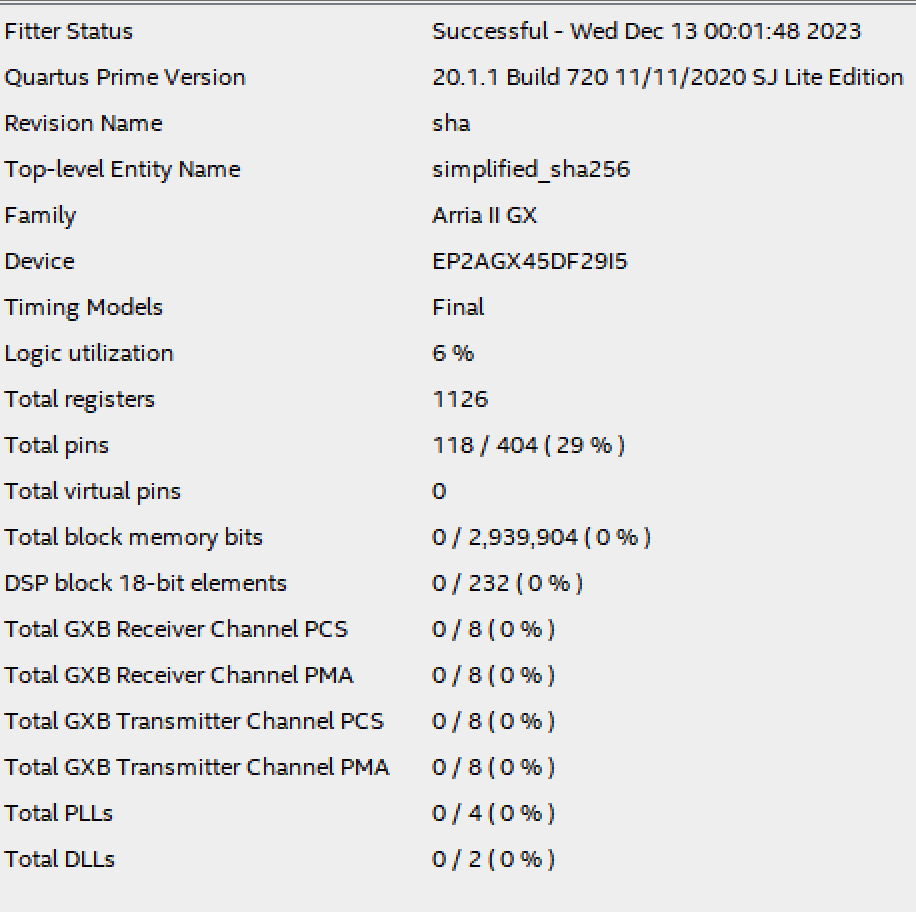


*Resource Usage:*

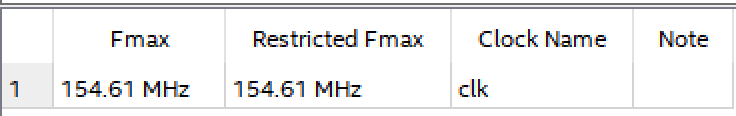
**

*Area:*Number of ALUTs + Registers = 1124 + 1126 = 2250

*Fitter Report:*

**

*Timing Fmax Report:*



Bitcoin\_hash

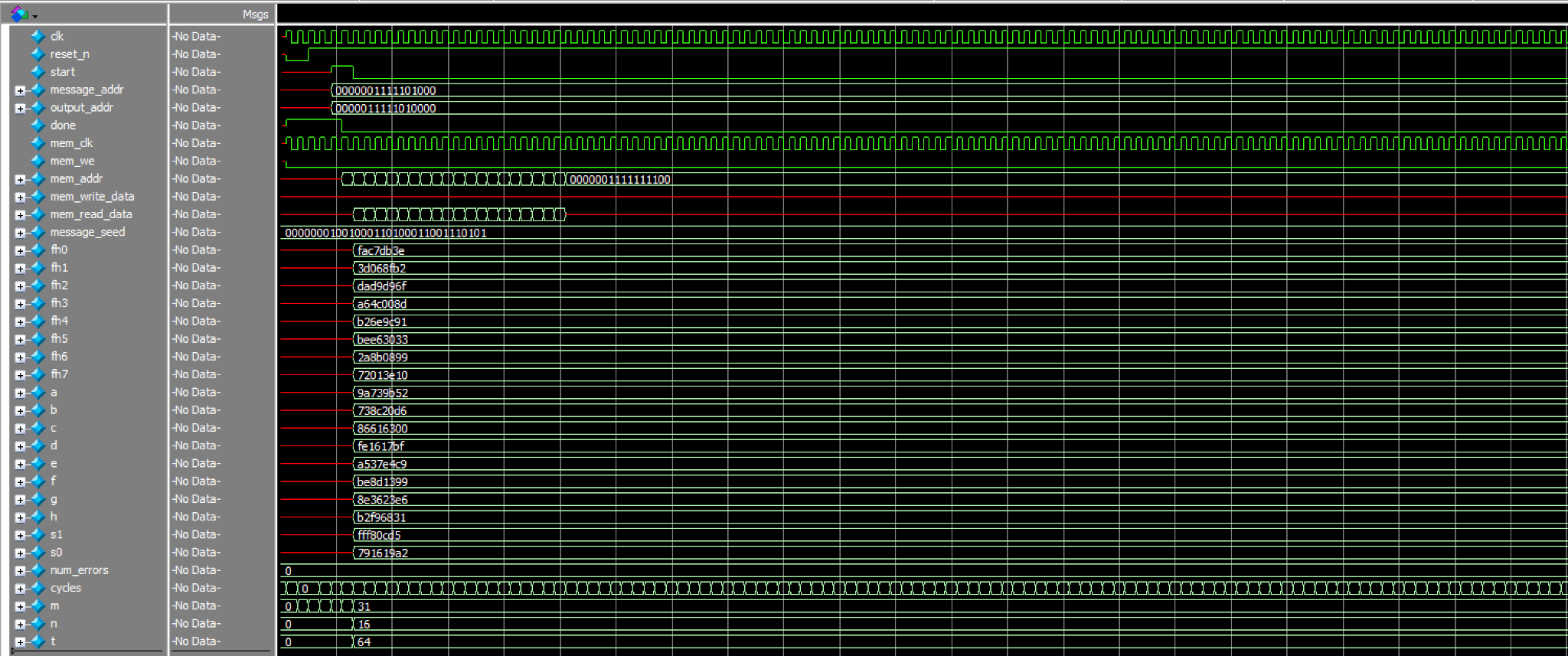
*Explanation:*

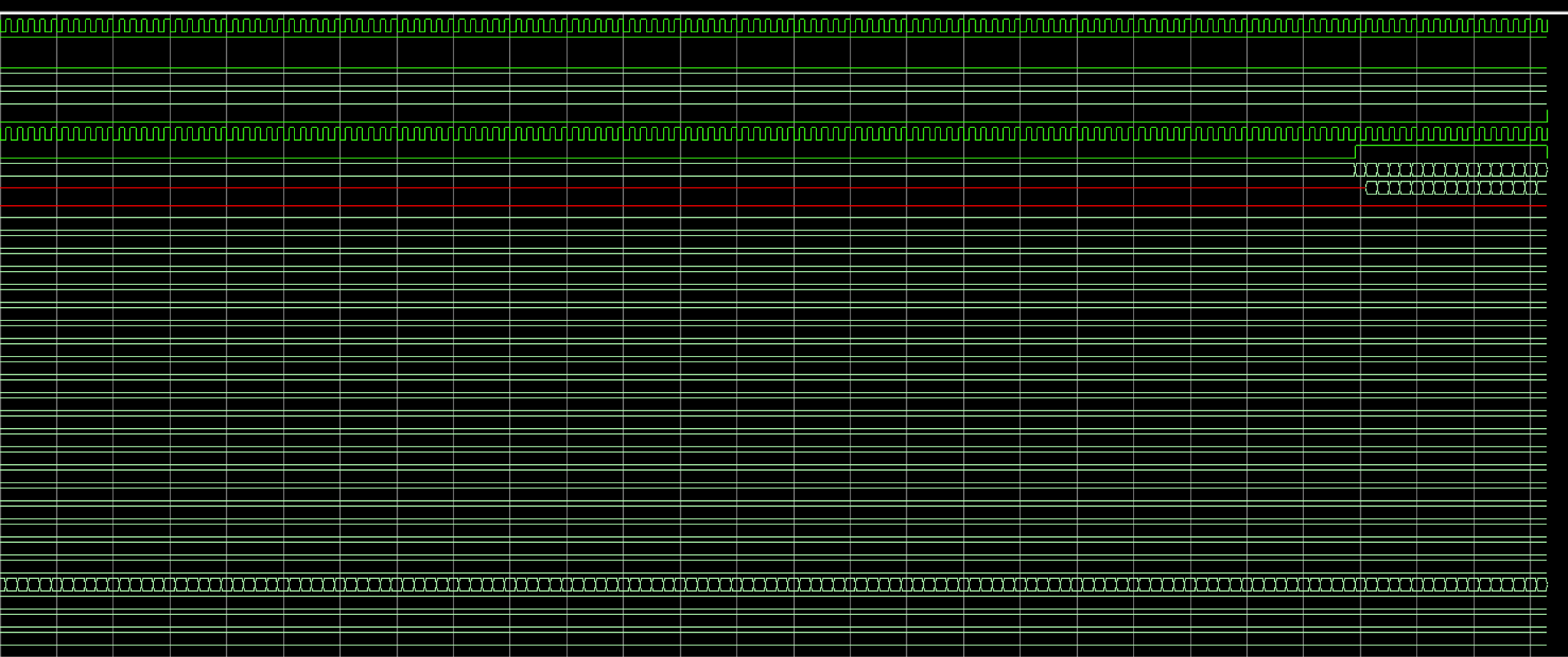
Bitcoin\_hash is a simulated implementation of the hashing performed on bitcoin blockchains when adding a new transaction to the blockchain. Essentially, the new addition to the blockchain needs to satisfy some arbitrary security requirements in order to be added; often, this requirement is in the form of some X number of zeros leading the hash. So in order to get this arbitrary number of zeroes, a 32-bit “nonce” → nonsense value is added to the transaction information and manipulated over and over again until the security requirements are met. So in Bitcoin\_hash, we are given the transaction information as well as some nonce values 0 to 15 and calculate a new output based on each nonce value, simulating this real-life process of satisfying the arbitrary security requirement.

*Algorithm:*

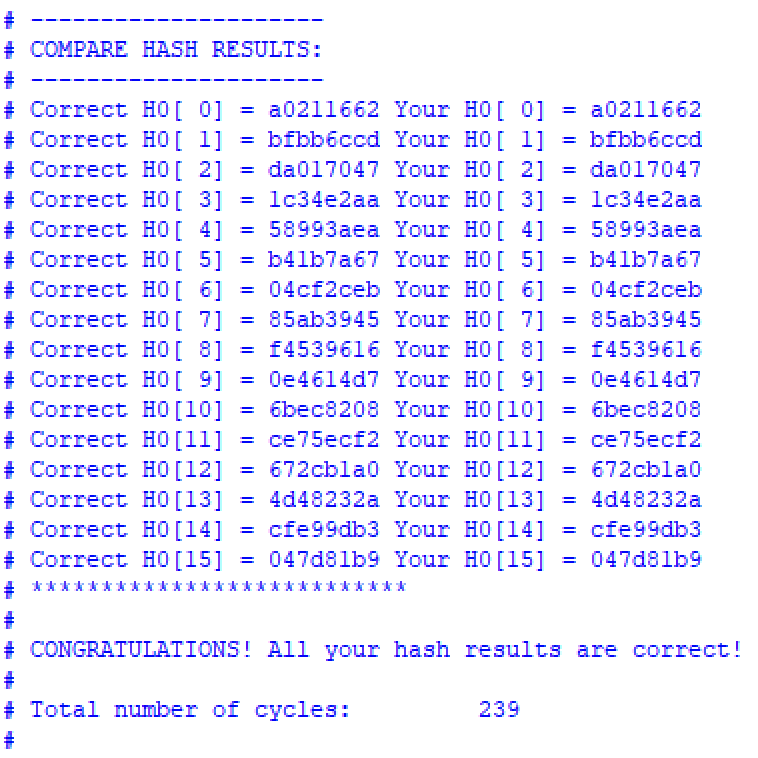
We implemented our algorithm for bitcoin\_hash in a similar, though more complex, way as compared to SHA-256. We used another finite state machine in order to transition between stages of execution as we executed our bitcoin hash. We started again in the IDLE state, where we initialized variables we would use throughout execution while waiting for a start flag. Once the start flag is received, we will transition to the READ stage, where we will read in our entire 19-block input (plus one block for the nonce that we set ourselves) into two arrays, W1 and W2. We would then compute the hash for W1 in the BLOCK1 and COMPUTE1 steps and save those outputted hash values to be used as inputs to W2 since we are manipulating the nonce in W2 and W1 will stay the same each time. Then we would cycle through the BLOCK2, COMPUTE2, PROCESS, and COMPUTE3 states, each time hashing the W2 array with a new nonce value and saving the h0 hash output into a different array to be used as output. Once we had cycled through all 16 potential nonce values (done in parallel), we would transition to the WRITE state, where we would write all our saved h0 values for each nonce to an address in memory.

*Simulation Waveforms:*

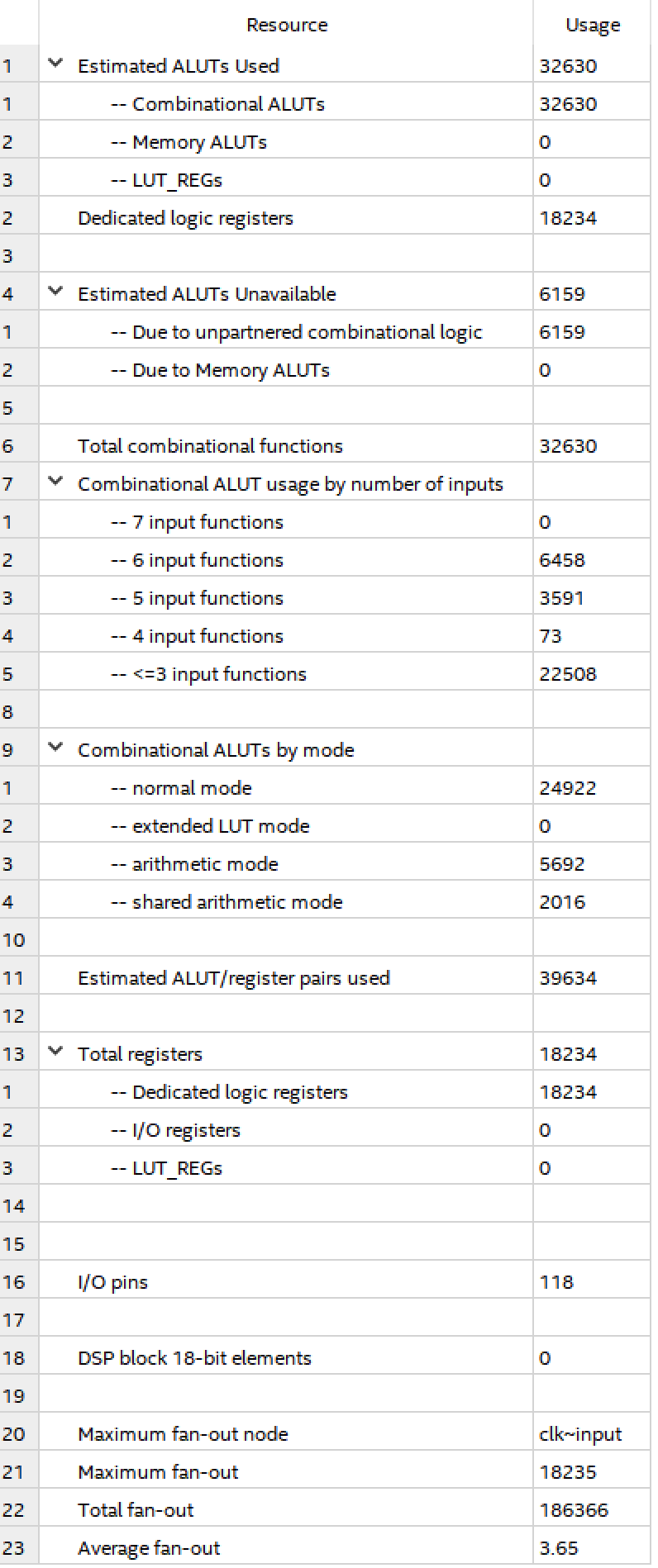
**

**

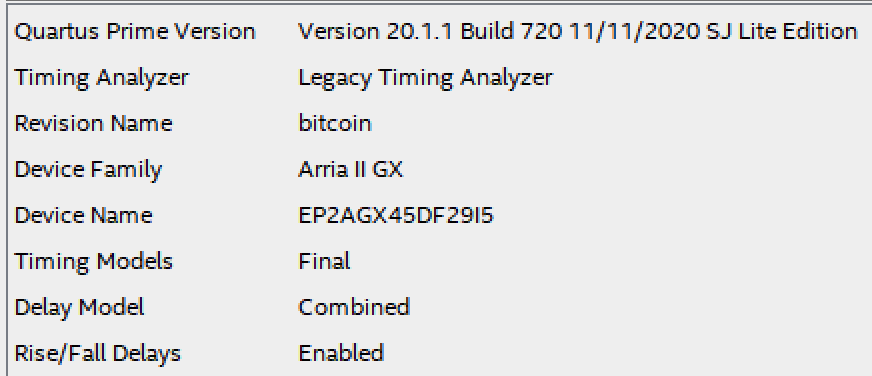
*ModelSim Transcripts:*



*Resource Usage:*



*Timing:*

**

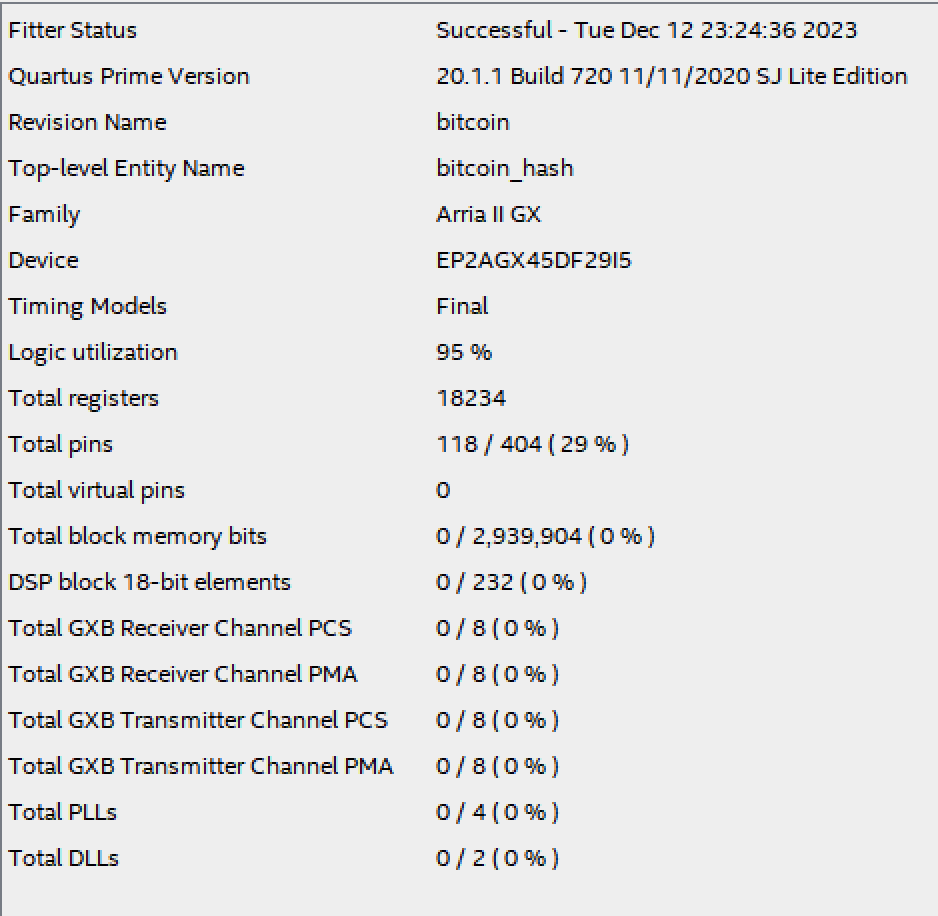
*Delay:*

2.06 microseconds

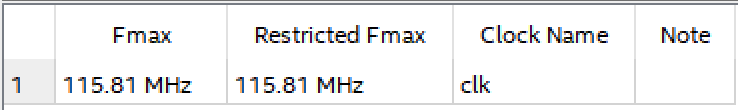
*Area:*

Number of ALUTs + Registers = 32630 + 18234 = 50,864

*Fitter Report:*



*Timing Fmax Report:*

**