Ashlee Young A17050291
Andrew Onozuka A16760043
13 Dec. 2024

ECE 111 Final Project + (Presentation)

SHA-256

1. What is SHA-256?

SHA stands for Secure Hash Algorithm, and there are multiple versions of SHA, each designed for specific cryptographic requirements. The one implemented in this project is SHA-256, a widely-used cryptographic hashing algorithm. SHA-256 can process input messages up to 2^{64} bits (approximately 2.3 billion gigabytes) and produces a fixed 256-bit output, known as a message digest. Regardless of the size of the input, the output hash is always 256 bits, making it a deterministic function.

To qualify as a secure cryptographic hashing function, SHA-256 adheres to several key principles:

a. Compression: It maps an arbitrary-length input to a fixed-length output.
b. Avalanche Effect: A small change in the input causes significant, unpredictable changes in the output hash.
c. Determinism: The same input always produces the same output hash.
d. Pre-image Resistance: It is computationally infeasible to reverse-engineer the input from its hash.
e. Collision Resistance: Finding two distinct inputs that produce the same hash is computationally infeasible.
f. Efficiency: The algorithm can compute the hash quickly, even for large inputs.

These properties make SHA-256 a cornerstone of modern cryptography, ensuring data integrity and authenticity in applications such as digital signatures, password storage, and blockchain technologies. The algorithm is also integral to secure communication protocols like TLS and SSL.

The general algorithm More details on the principles and structure of SHA-256 can be found in the project writeup slides (pages 24–33).

2. Algorithm for SHA-256 Implemented in the Code

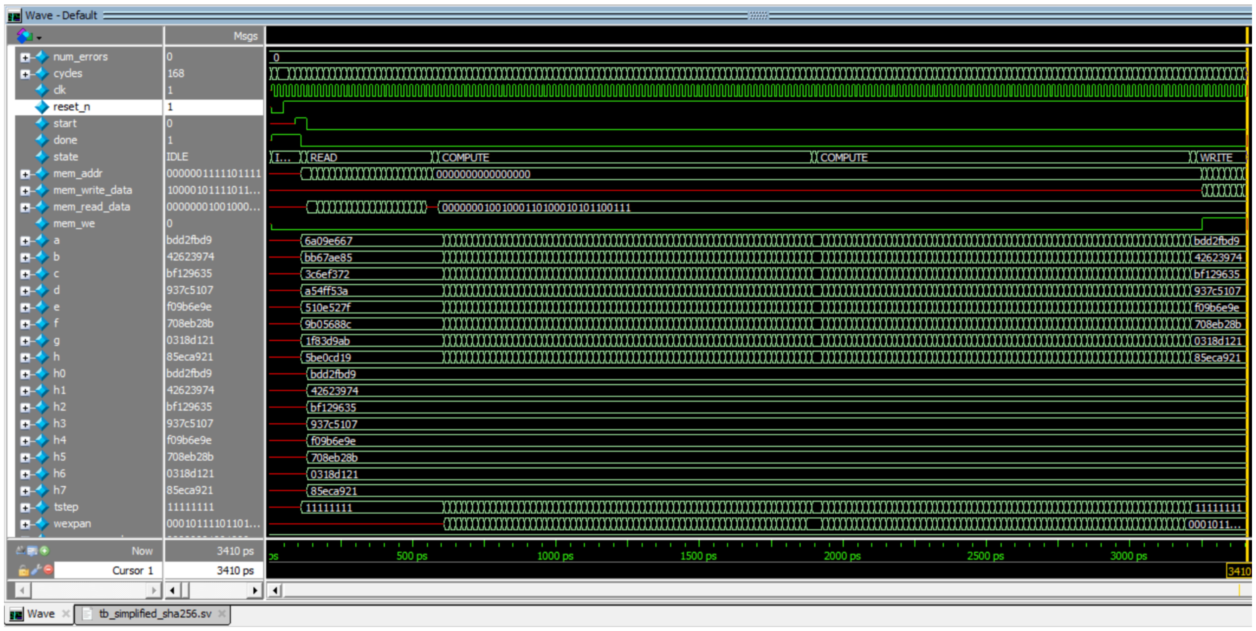
The SHA-256 implementation in our project is designed to compute a secure 256-bit hash value for a given input message, following the cryptographic principles of the algorithm. It processes input data in 512-bit blocks, using a state-based approach controlled by a finite state machine (FSM) to ensure that each step of the hashing process is executed sequentially and accurately. The implementation divides the algorithm into distinct states: IDLE, WAIT, READ, BLOCK, COMPUTE, and WRITE, with each state handling a specific stage of the process.

In the IDLE state, the algorithm initializes key variables, including the eight initial hash values (h0 through h7) defined by the SHA-256 standard and the working registers (a through h). Once the initialization is complete, the FSM transitions to the READ state, where the input message is fetched from memory into a message array. The message is then divided into 512-bit blocks in the BLOCK state. For the first block, the first 16 words of the message are directly copied, while the second block includes padding and the 64-bit representation of the message length, as required by the SHA-256 standard.

The COMPUTE state performs the core of the SHA-256 algorithm, executing 64 rounds of hashing for each block. Each round updates the working registers using the SHA-256 compression function, which includes operations like word expansion, bitwise rotations, XOR, and logical functions such as Ch and Maj. After completing 64 rounds, the hash values are updated by adding the working registers to the current hash values. If more blocks remain, the FSM returns to the BLOCK state; otherwise, it transitions to the WRITE state.

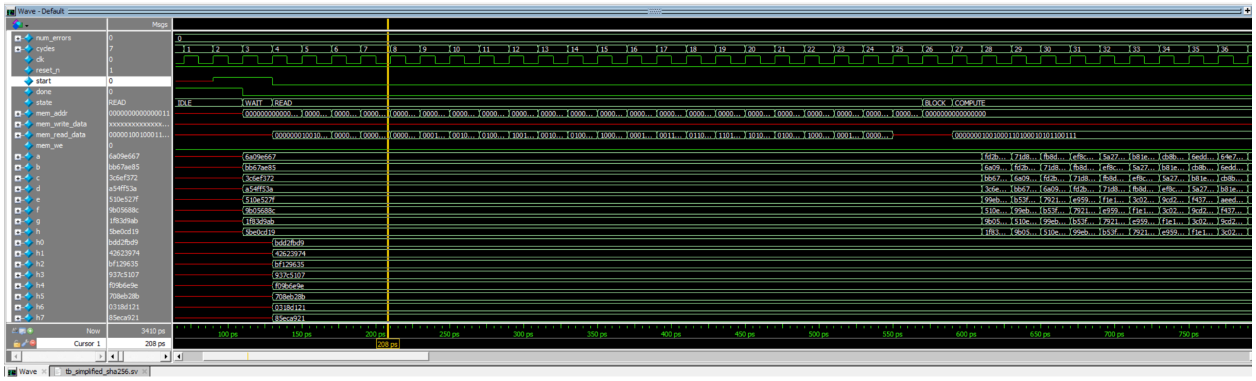
In the WRITE state, the final 256-bit hash is written back to memory, with each of the eight 32-bit hash values stored sequentially. The FSM then returns to the IDLE state, where it waits for the next input message. The implementation ensures efficient and secure hashing by adhering to the SHA-256 standard and employing a modular FSM-based design. This approach facilitates the computation of deterministic and collision-resistant hash values, making it suitable for cryptographic and data integrity applications.

3. Simulation Waveform Snapshot SHA256

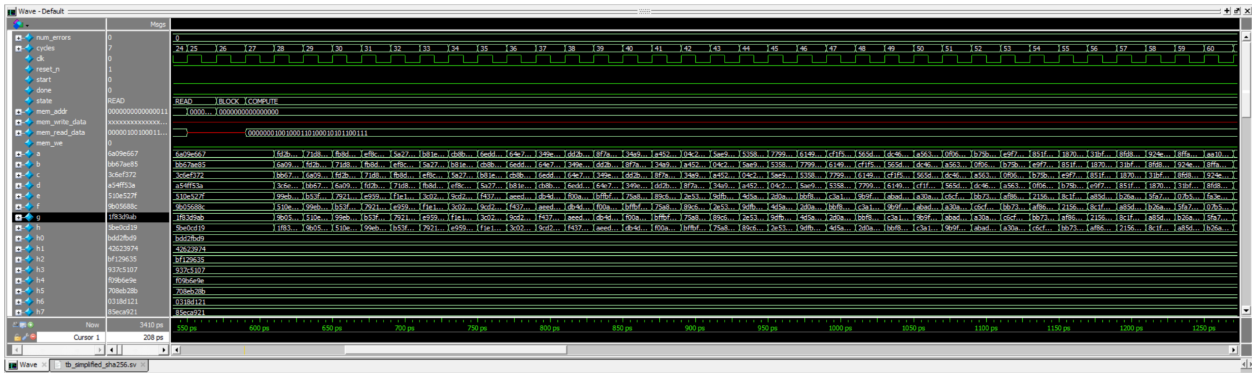


```
Wave - Default
                     Msgs
num_errors           0
cycles               168
clk                  1
reset_n              1
start                0
done                 1
state                IDLE
mem_addr             0000001111101111
mem_write_data       10000101111011...
mem_read_data        00000001001000...
mem_we               0
a                    bdd2fbd9
b                    42623974
c                    bf129635
d                    937c5107
e                    f09b6e9e
f                    708eb28b
g                    0318d121
h                    85eca921
h0                   bdd2fbd9
h1                   42623974
h2                   bf129635
h3                   937c5107
h4                   f09b6e9e
h5                   708eb28b
h6                   0318d121
h7                   85eca921
tstep                11111111
wexpan               00010111101101...
Now                  3410 ps
Cursor 1             3410 ps
```

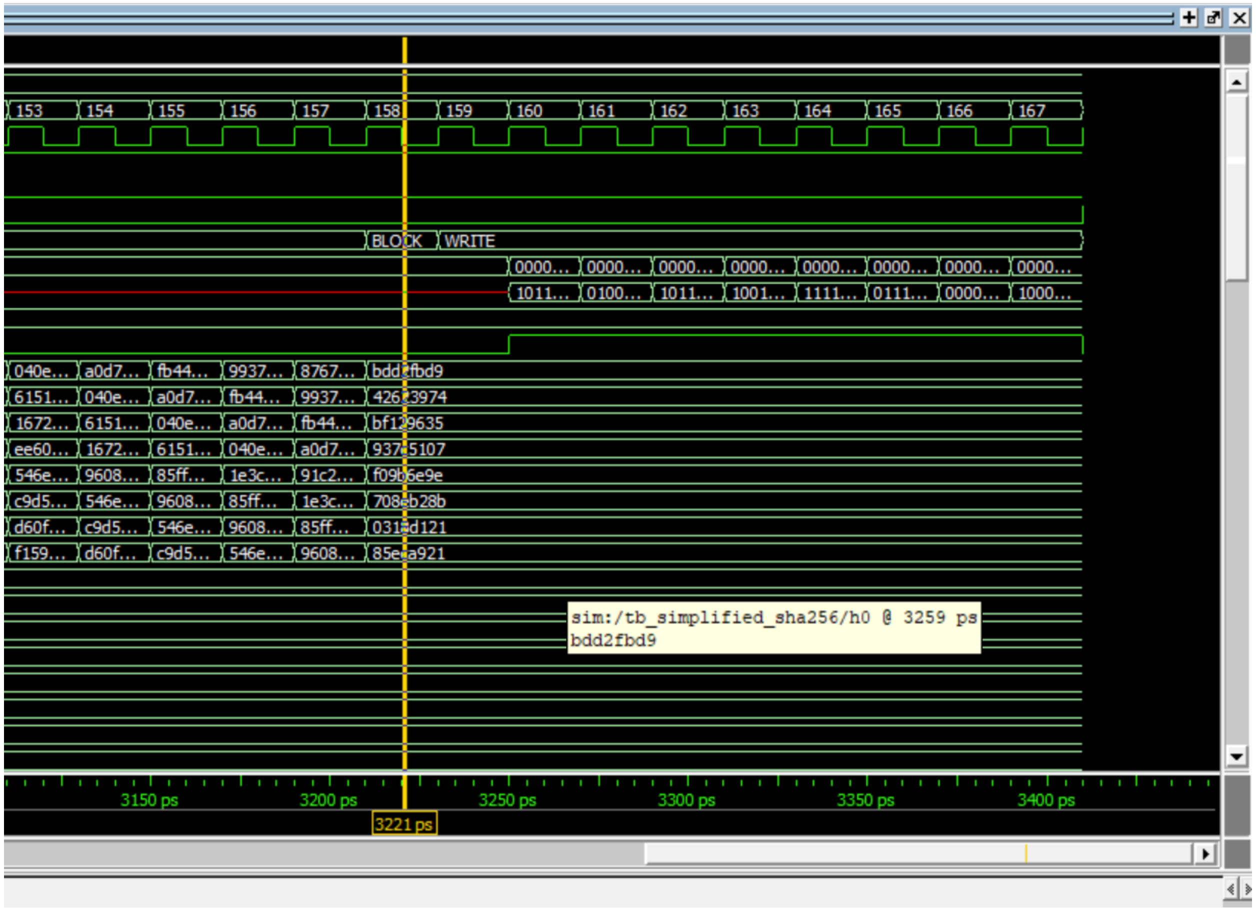
The waveform simulation of the SHA-256 testbench reflects the step-by-step operation of the hashing algorithm as implemented in the simplified_sha256 module. Each signal tracks the progression of the finite state machine (FSM), memory interactions, and computation of intermediate and final hash values. Analyzing the waveforms allows us to verify the correctness of the design and confirm that the expected results are achieved.



At the start of the simulation, the reset_n signal is asserted low to initialize the module, setting all registers and internal variables to their default states. The FSM transitions to the IDLE state, waiting for the start signal to be asserted. When start is asserted, the state signal transitions to READ, and the input message is read from memory into the message array. This is reflected in the mem_addr and mem_read_data waveforms, which show the memory addresses being accessed and the corresponding data being read. The offset variable increments with each memory read, ensuring the entire message is fetched.



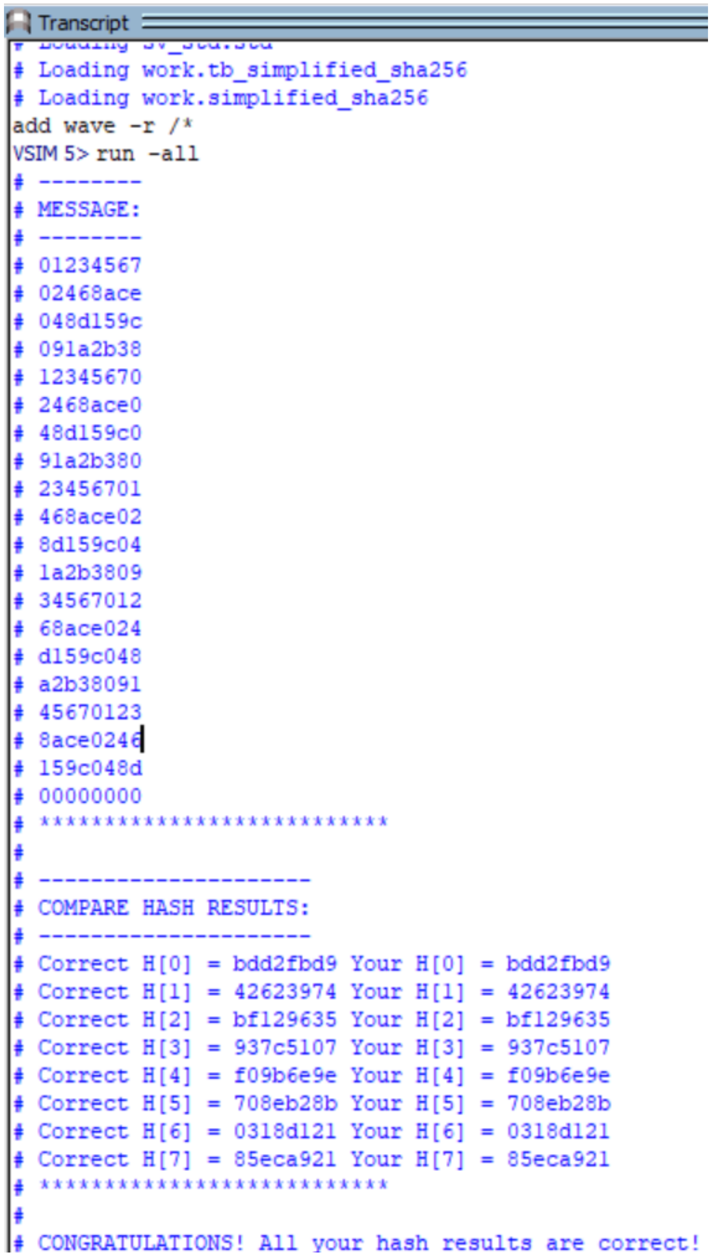
Once the input message is fully loaded, the FSM moves to the BLOCK state, where the message is divided into 512-bit blocks for processing. The state waveform transitions to COMPUTE as the hash computation begins. During this state, the waveforms for a to h display the intermediate values of the working registers, which are updated across 64 rounds of processing. The w array, responsible for word expansion, shows dynamic updates as new words are generated and processed. These operations correspond to the core SHA-256 compression function, which involves bitwise operations and additions.



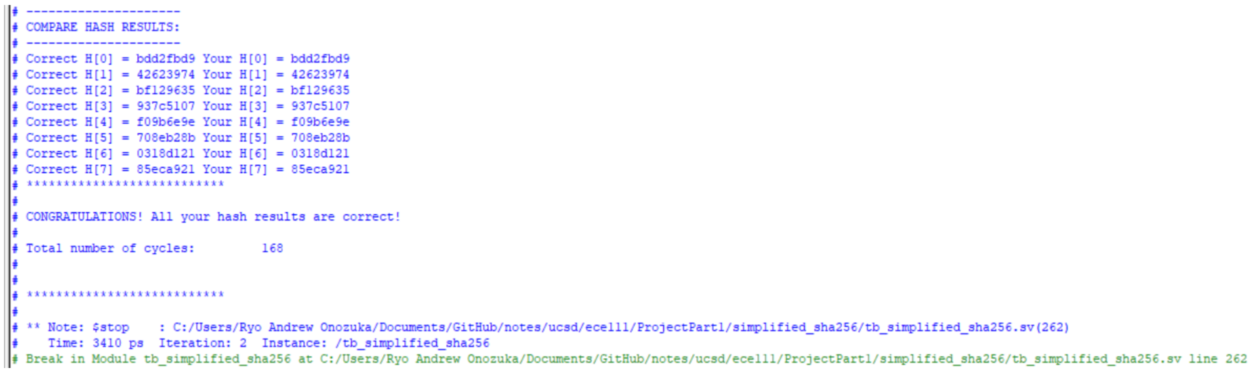
After completing the computation for a block, the FSM transitions to the WRITE state. The mem_write_data and mem_we waveforms indicate the hash values (h0 to h7) being written back to memory at the specified output_addr. The done signal is asserted when all hash computations are complete, signaling the end of the operation. At this point, the cycles waveform displays the total number of clock cycles taken, providing a performance metric for the design.

The simulation results confirm the correctness of the design, as seen in the final hash values written to memory. The testbench compares these values against the expected outputs, and the following correct results are observed:

4. ModelSim Transcript Window SHA256



```
# Loading sv_std.std
# Loading work.tb_simplified_sha256
# Loading work.simplified_sha256
add wave -r /*
VSIM 5> 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!
```



```
# ---------------------
# 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/Ryo Andrew Onozuka/Documents/GitHub/notes/ucsd/ece111/ProjectPart1/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/Ryo Andrew Onozuka/Documents/GitHub/notes/ucsd/ece111/ProjectPart1/simplified_sha256/tb_simplified_sha256.sv line 262
```

5. What is Bitcoin Hashing?

Bitcoin hashing refers to the process of applying cryptographic hash functions to secure data within the Bitcoin network. It primarily uses the hash function Secure Hash Algorithm 256-Bit (SHA-256) to generate a fixed 256-bit (32-byte) output from some input.

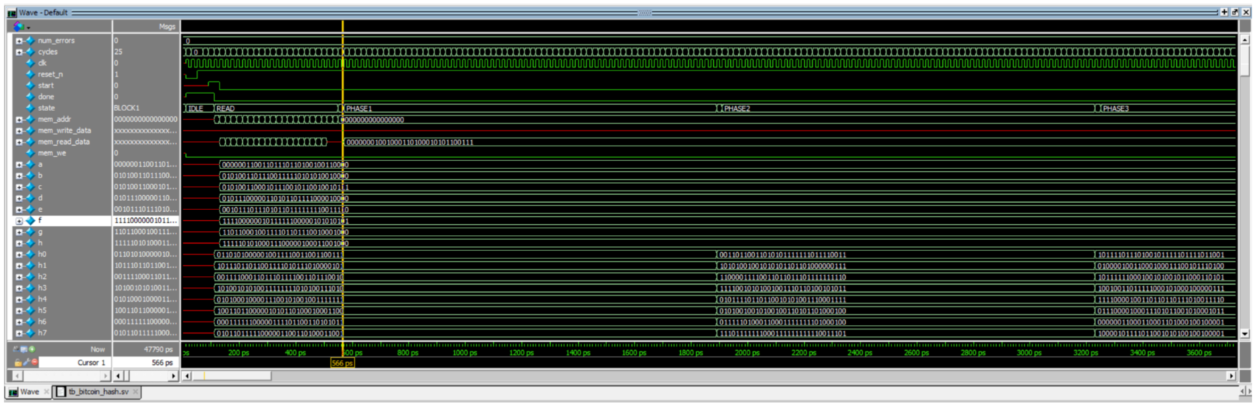
Bitcoin hashing consists of three main phases. Phase 1 processes the first block of the first SHA-256 function, where initial constants (H0…H7), the first 16 words in memory (Wt), and fixed constants (Kt[0:63]) are used. Phase 2 processes the second block of the first SHA-256 function using the outputs from Phase 1 (H0…H7). Additional memory values including a nonce, padding, and a 640-bit message size are also included. Finally, Phase 3 involves the second SHA-256 hash function where constants (H0…H7), outputs from Phase 2, padding—including a 256-bit message size—are processed. Both Phase 2 and 3 are repeated 16 times to produce 16 final hashes.

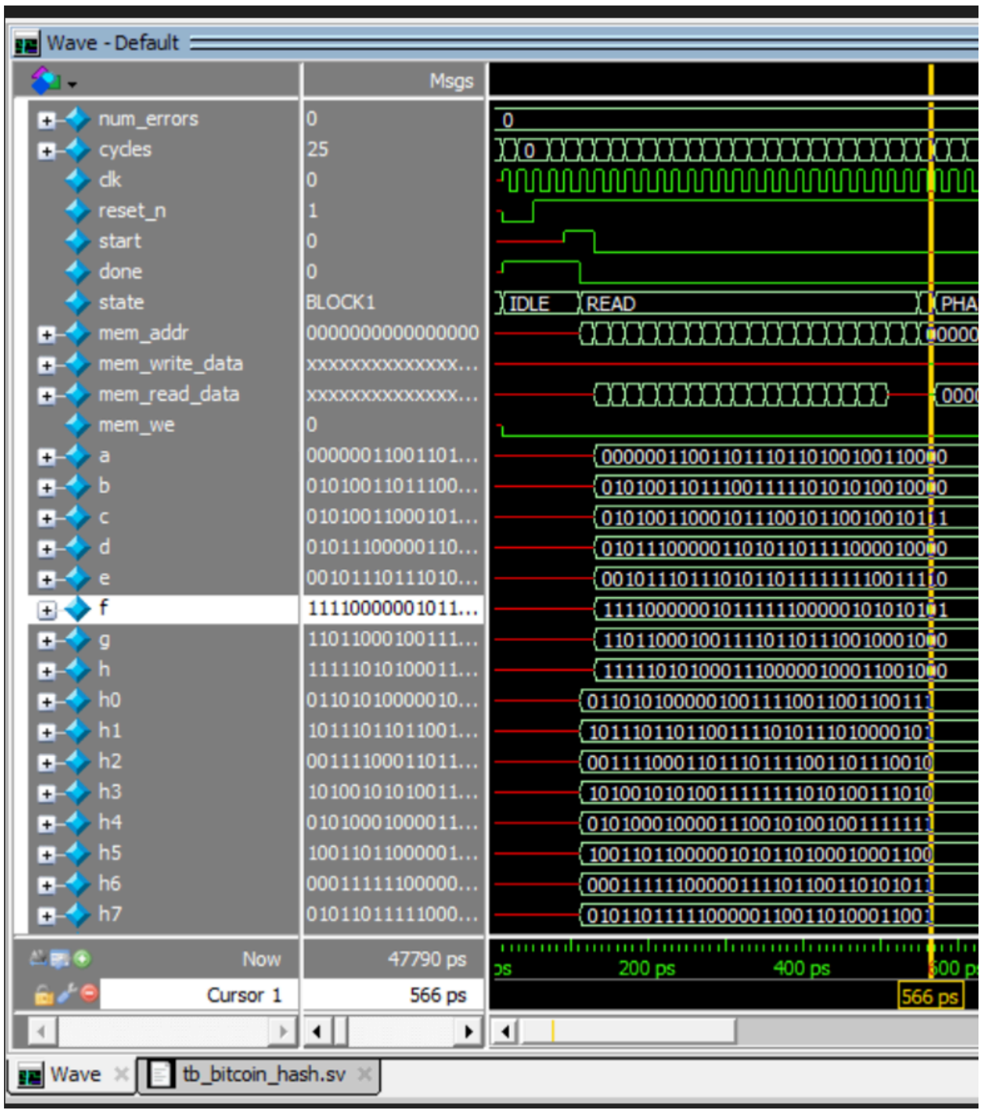
6. Algorithm for Bitcoin Hash Implemented in the Code

The provided module implements the Bitcoin hash algorithm using SHA-256, a cryptographic hash function that processes input data in 512-bit blocks. The module works with a finite state machine (FSM) that goes through the various stages of the hashing process. Initially, in the IDLE state, the hash values (h0-h7) are initialized with predefined constants, and the system waits for the start signal. Once the start signal is received, the module transitions to the READ state where it reads the input message from memory. The message, consisting of 20 32-bit words, is loaded into the message array, and once all words are read, the FSM progresses to BLOCK1, where the first message block is processed. During this state, the working variables (a-h) are initialized to the hash values h0-h7, and the first phase of SHA-256 begins. In PHASE1, PHASE2, and PHASE3, the algorithm runs 64 rounds of hashing. The first 16 rounds are directly processed using the words from the message, and for the remaining rounds, the wexpan function expands the message using bitwise operations and rotations. The actual operations for each round are carried out in the sha256_op function, which updates the working variables with the help of the round constants (k). After completing the rounds in a given block, the updated hash values are saved. The process continues with BLOCK2 and BLOCK3, where message padding is applied, and additional nonce and length information is added to the message. These padded blocks are then hashed in the same way as the first block. Finally, after all blocks are processed, the resulting hash values are written to memory in the WRITE state. The FSM returns to the IDLE state once the entire process is complete. The module thus effectively implements the Bitcoin hashing algorithm in a hardware-friendly manner, handling the reading, processing, and writing of data while adhering to the SHA-256 specification.

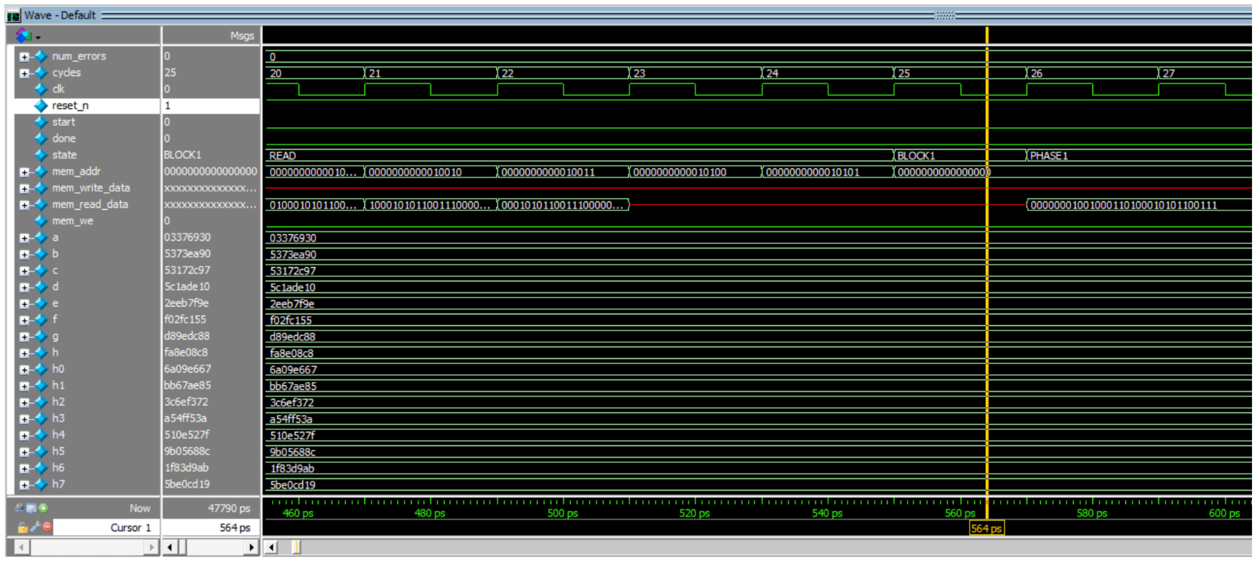
An attempt to optimize the hashing implementation was done by using parallelization techniques by executing Phase 2 in parallel so that it would be available for all nonce values. Since Phase 3 uses the output hashes from Phase 2 as its input for the same nonce values, it is executed serially. Since sixteen instances of SHA256 cannot be fit on the FPGA, an attempt to perform parallel implementation for the first 8 nonces was attempted and later for the last 8.

7. Simulation Waveform Snapshot Bitcoin Hash (unclear if explanations needed)

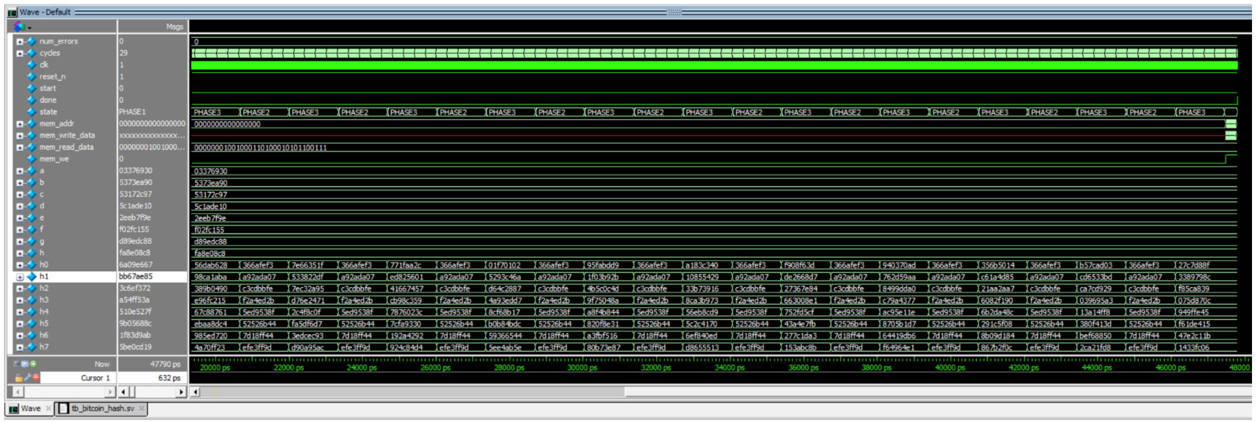




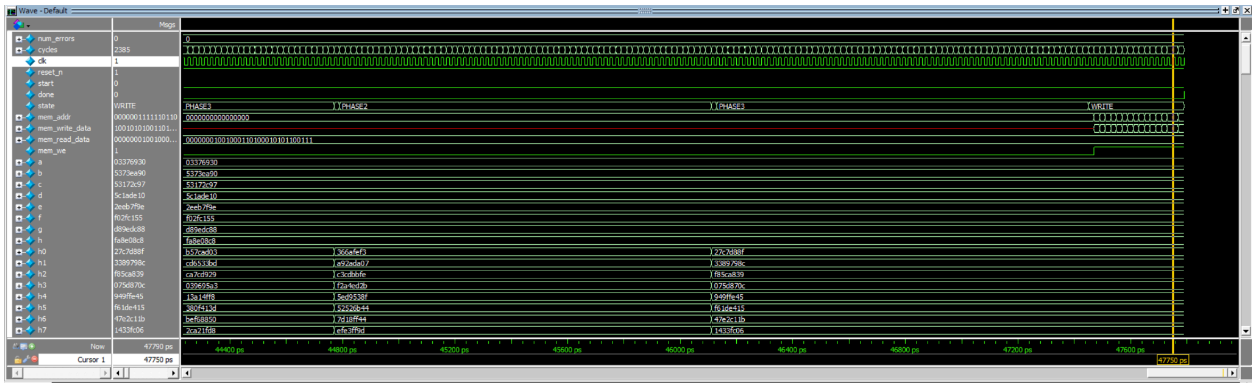
This snapshot shows the beginning of the testbench, as we can see the relevant waveforms such as a-h and h0-h7, as well as the state changes and the initial input into mem_read_data.



this snapshot shows the first block and 1st phase.

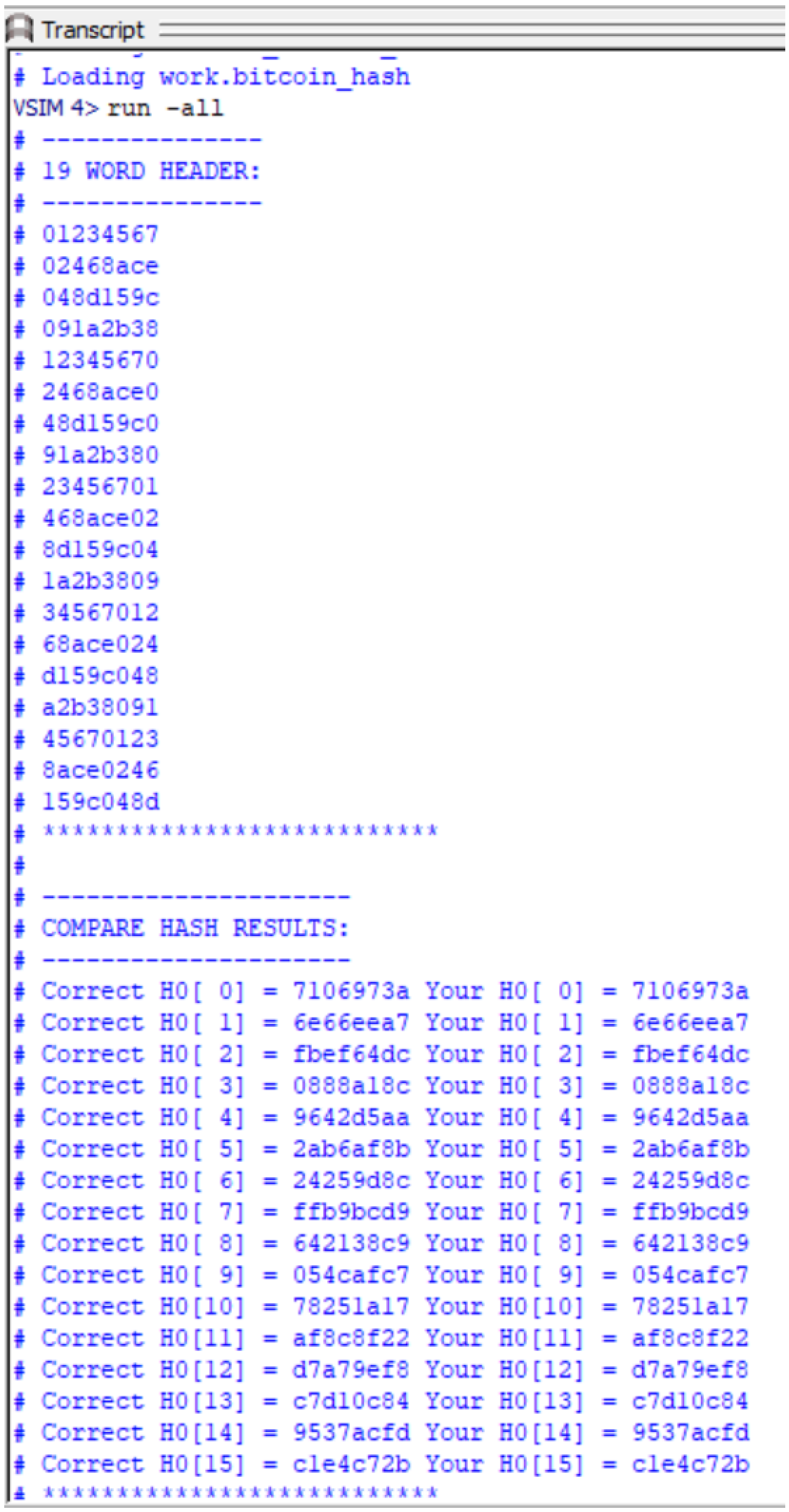


this is a zoomed out view of the hashing towards the end, before the testbench completes. we can see that we are computing the final hash, using sha(sha(message) for 16 nonces.

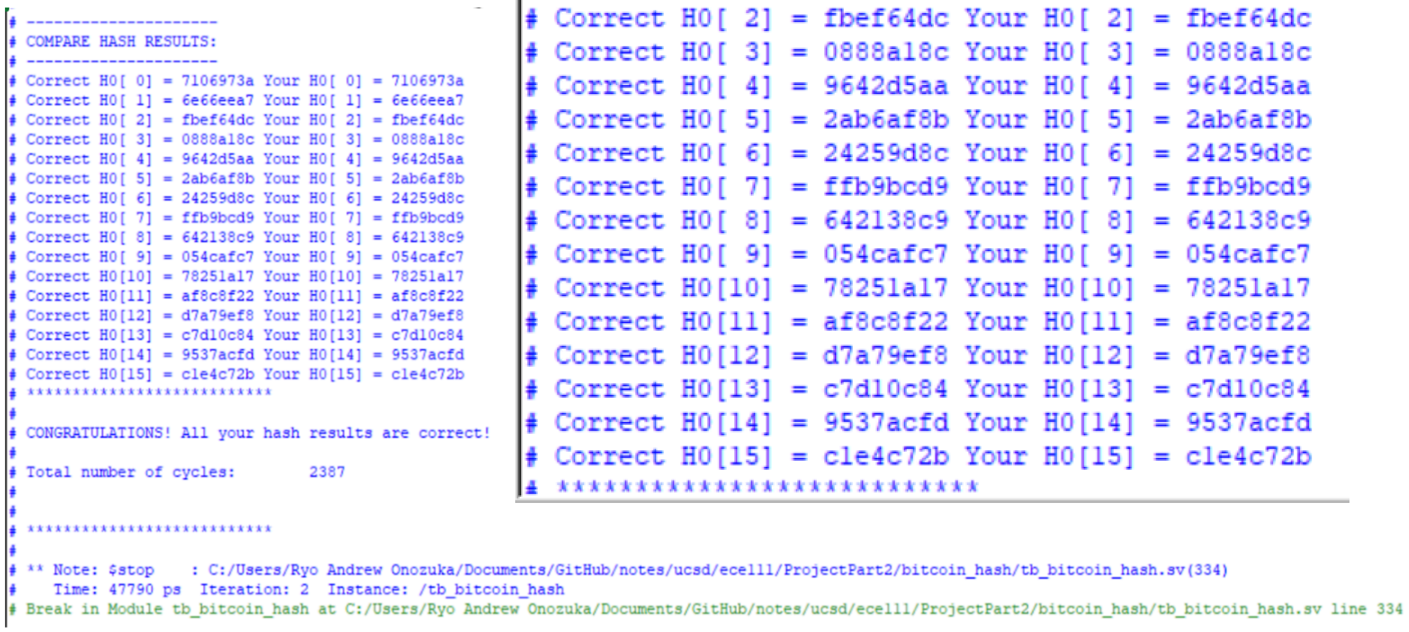


below is a zoomed in photo of the end, where we can see the 16 outputs H0[0], H0[1] …, H0[15] all being written.

8. ModelSim Transcript Window Bitcoin Hash

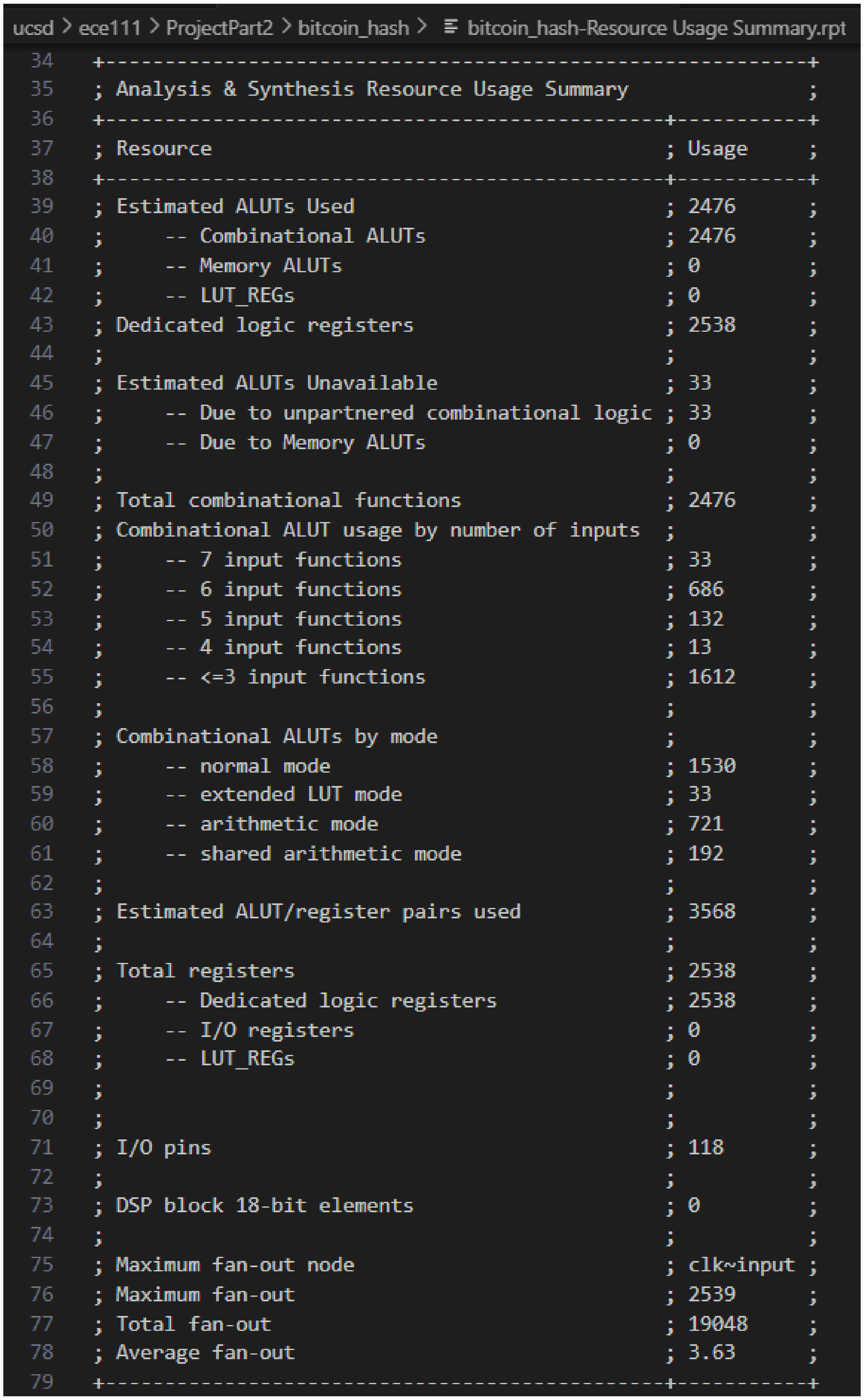


```
# Loading work.bitcoin_hash
VSIM 4> run -all
# ---------------
# 19 WORD HEADER:
# ---------------
# 01234567
# 02468ace
# 048d159c
# 091a2b38
# 12345670
# 2468ace0
# 48d159c0
# 91a2b380
# 23456701
# 468ace02
# 8d159c04
# 1a2b3809
# 34567012
# 68ace024
# d159c048
# a2b38091
# 45670123
# 8ace0246
# 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
# ***************************
```



```
# ---------------------
# 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:        2387
#
#
# ***************************
#
# ** Note: $stop    : C:/Users/Ryo Andrew Onozuka/Documents/GitHub/notes/ucsd/ece111/ProjectPart2/bitcoin_hash/tb_bitcoin_hash.sv(334)
#    Time: 47790 ps  Iteration: 2  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at C:/Users/Ryo Andrew Onozuka/Documents/GitHub/notes/ucsd/ece111/ProjectPart2/bitcoin_hash/tb_bitcoin_hash.sv line 334
```

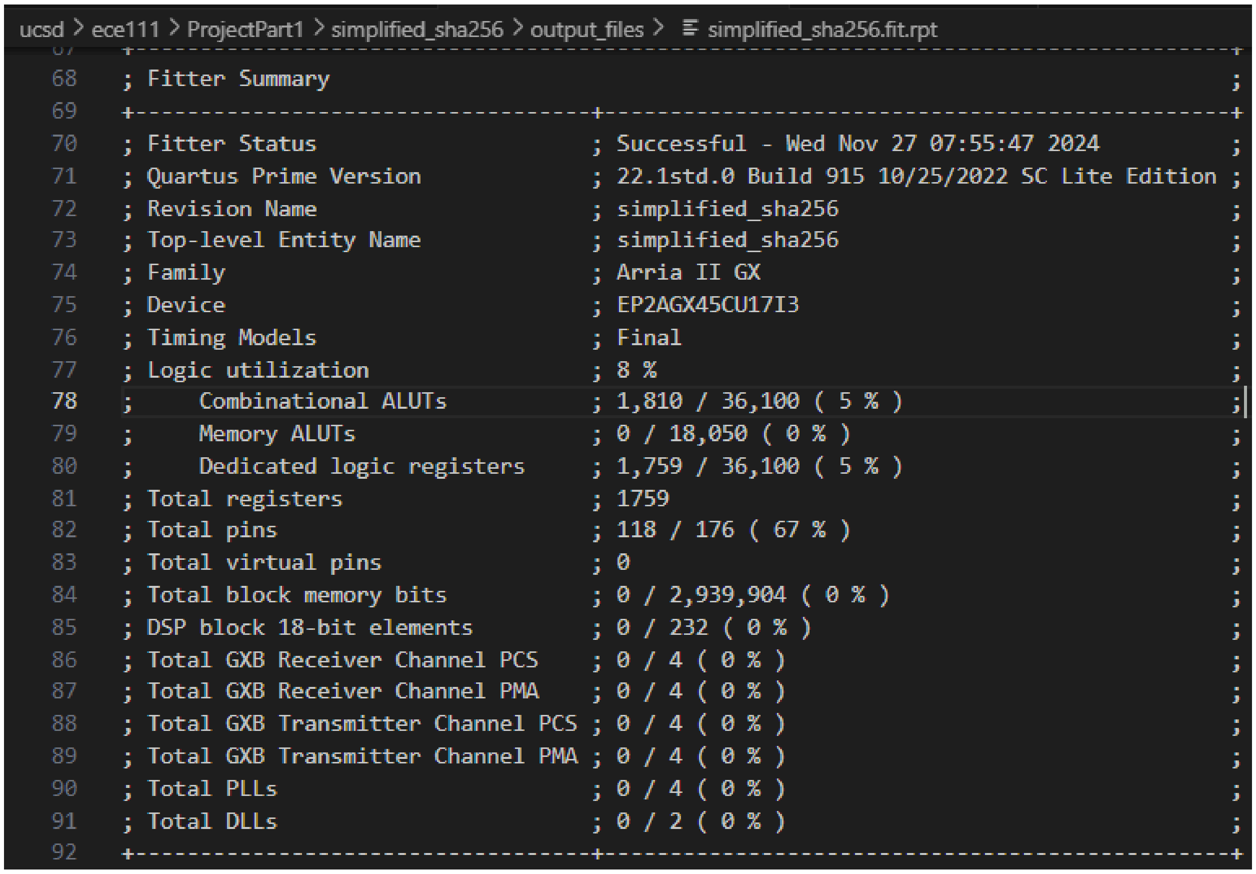
9. **Synthesis Resource Usage Snapshot Bitcoin Hash only**



ucsd > ece111 > ProjectPart2 > bitcoin_hash > bitcoin_hash-Resource Usage Summary.rpt

```
+---------------------------------------------------------+
; Analysis & Synthesis Resource Usage Summary             ;
+---------------------------------------------+-----------+
; Resource                                    ; Usage     ;
+---------------------------------------------+-----------+
; Estimated ALUTs Used                        ; 2476      ;
;     -- Combinational ALUTs                  ; 2476      ;
;     -- Memory ALUTs                         ; 0         ;
;     -- LUT_REGs                             ; 0         ;
; Dedicated logic registers                   ; 2538      ;
;                                             ;           ;
; Estimated ALUTs Unavailable                 ; 33        ;
;     -- Due to unpartnered combinational logic ; 33      ;
;     -- Due to Memory ALUTs                  ; 0         ;
;                                             ;           ;
; Total combinational functions               ; 2476      ;
; Combinational ALUT usage by number of inputs ;          ;
;     -- 7 input functions                    ; 33        ;
;     -- 6 input functions                    ; 686       ;
;     -- 5 input functions                    ; 132       ;
;     -- 4 input functions                    ; 13        ;
;     -- <=3 input functions                  ; 1612      ;
;                                             ;           ;
; Combinational ALUTs by mode                 ;           ;
;     -- normal mode                          ; 1530      ;
;     -- extended LUT mode                    ; 33        ;
;     -- arithmetic mode                      ; 721       ;
;     -- shared arithmetic mode               ; 192       ;
;                                             ;           ;
; Estimated ALUT/register pairs used          ; 3568      ;
;                                             ;           ;
; Total registers                             ; 2538      ;
;     -- Dedicated logic registers            ; 2538      ;
;     -- 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                             ; 2539      ;
; Total fan-out                               ; 19048     ;
; Average fan-out                             ; 3.63      ;
+---------------------------------------------+-----------+
```

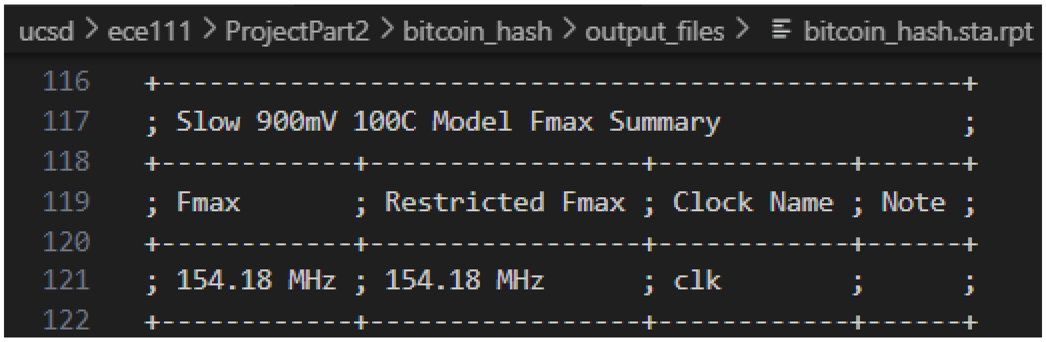
10. fitter report snapshot - bitcoin_hash.fit.rpt



ucsd > ece111 > ProjectPart1 > simplified_sha256 > output_files > simplified_sha256.fit.rpt

```
; Fitter Summary                                                                   ;
+----------------------------------+----------------------------------------------+
; Fitter Status                    ; Successful - Wed Nov 27 07:55:47 2024        ;
; Quartus Prime Version            ; 22.1std.0 Build 915 10/25/2022 SC Lite Edition ;
; Revision Name                    ; simplified_sha256                            ;
; Top-level Entity Name            ; simplified_sha256                            ;
; Family                           ; Arria II GX                                  ;
; Device                           ; EP2AGX45CU17I3                               ;
; Timing Models                    ; Final                                        ;
; Logic utilization                ; 8 %                                          ;
;     Combinational ALUTs          ; 1,810 / 36,100 ( 5 % )                       ;
;     Memory ALUTs                 ; 0 / 18,050 ( 0 % )                           ;
;     Dedicated logic registers    ; 1,759 / 36,100 ( 5 % )                       ;
; Total registers                  ; 1759                                         ;
; Total pins                       ; 118 / 176 ( 67 % )                           ;
; 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 / 4 ( 0 % )                                ;
; Total GXB Receiver Channel PMA   ; 0 / 4 ( 0 % )                                ;
; Total GXB Transmitter Channel PCS ; 0 / 4 ( 0 % )                               ;
; Total GXB Transmitter Channel PMA ; 0 / 4 ( 0 % )                               ;
; Total PLLs                       ; 0 / 4 ( 0 % )                                ;
; Total DLLs                       ; 0 / 2 ( 0 % )                                ;
+----------------------------------+----------------------------------------------+
```

11. timing Fmax report snapshots - bitcoin_hash.sta.rpt



ucsd > ece111 > ProjectPart2 > bitcoin_hash > output_files > bitcoin_hash.sta.rpt

```
+-------------------------------------------------+
; Slow 900mV 100C Model Fmax Summary              ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 154.18 MHz ; 154.18 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
```