SHA-256 & Bitcoin Hashing

ECE 111

Uriel Salazar

Part 1: SHA-256

I: What is SHA-256?

SHA-256 stands for Secure Hash Algorithm. This is a simplified secure hash algorithm, which takes imputed data and converts this into 256 bits output hash. This unique property is what enables the ability to verify a file's integrity, as every input has its own unique hash output. Ultimately, this is a way for users to have their data protected, in the event a type of transaction is made or sensitive information is being shared and used.

II: SHA-256 SystemVerilog Algorithm

We begin by initializing our hash constants, `k[0:63]`. Following this, our function `determine_num_blocks` is called—here we determine the number of blocks required, with respect to our variable `NUM_OF_WORDS`.

The function `sha256_op` is designed to calculate the next `rightrotate` with use of `exp_word` compliments from step 4 under SHA-256 algorithm slides, during the `COMPUTE` state of the algorithm.

During the `IDLE` state, `h0-h07` and `a-h` are initialized to `0` when variable `start = 1`. Variables `cur_we`, `cur_write_data`, and `offset` are set to `0` also, as data from memory must first be read. `message_addr` is assigned to `cur_addr`, the address for reading data, prior to moving to `DELAY` state. However, if `start = 0`, the state remains in `IDLE`.

In `DELAY` state, `cur_we` is inspected to see if we are either writing (when `cur_we = 1`) or reading (when `cur_we = 0`). If `cur_we = 1`, we transit to `WRITE` state, in order to begin writing to memory. Otherwise, `cur_we = 0` and we commence to `READ` state.

In `READ` state, the data from each cycle found in `mem_read_data` is copied to `message[0:19]`. Afterwards, `NUM_OF_WORDS` is transitioned to `READ` as well, in order for the data to be read, then the algorithm moves onto `BLOCK` state.

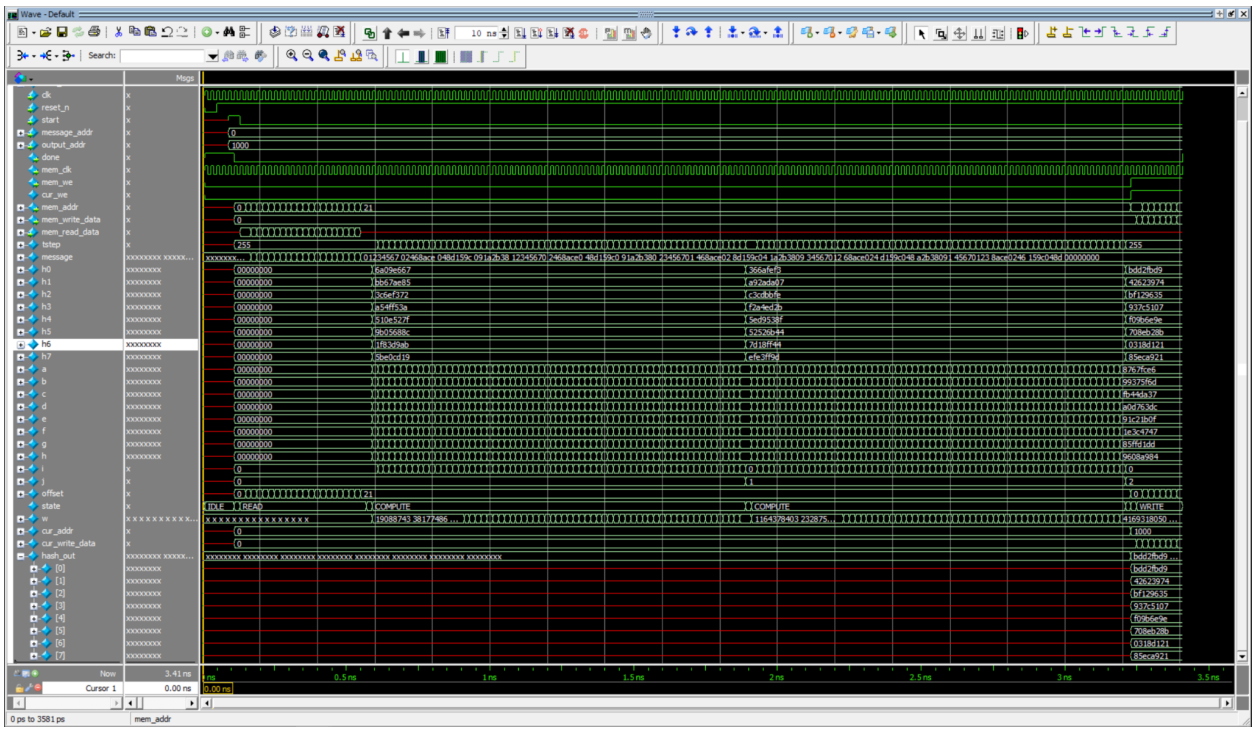
In `BLOCK` state, the message is fetched by initiating a hash value computation, in order to read the message block from memory. This computation entails the following:

- As `j < num_blocks`, we read through the 512-bit block. As soon as the number of blocks is reached, `j` is incremented each time it is passed to `COMPUTE` state.

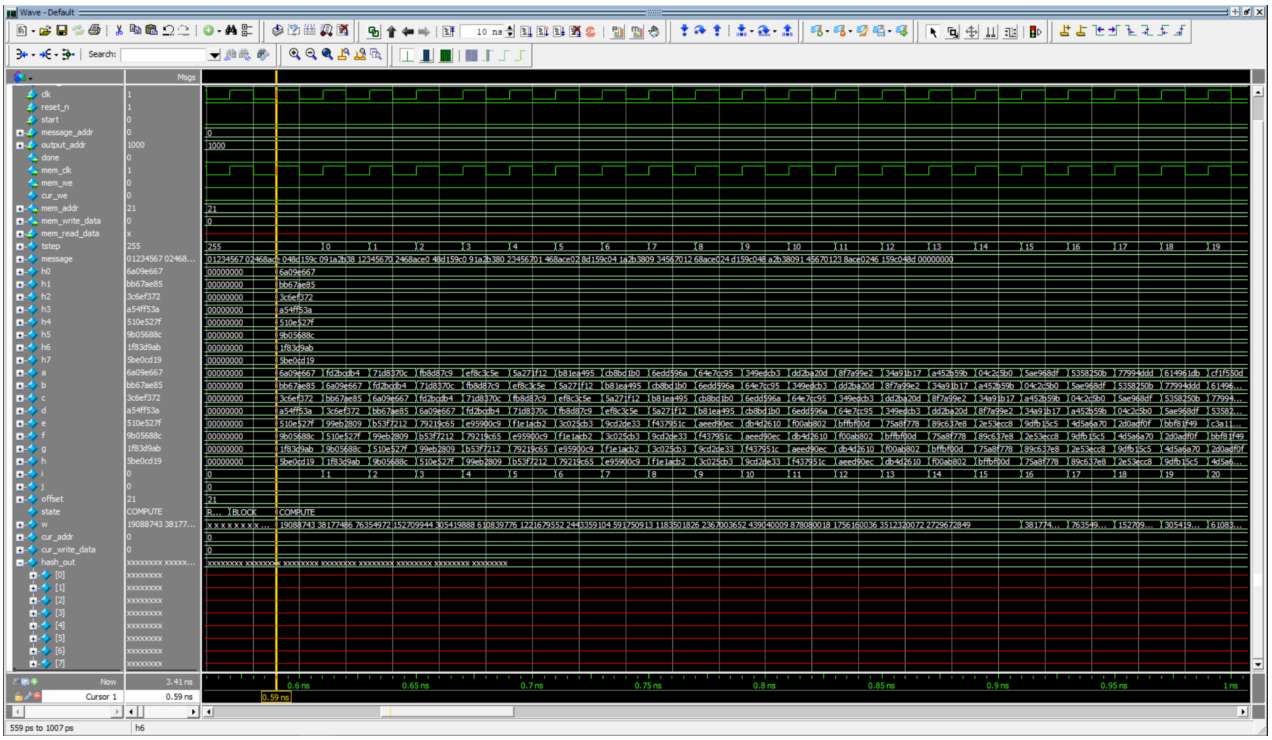
- As `j >= num_blocks`, `h0-h7` is assigned to `sha256[0:7]`, `cur_we` is assigned the value of `1`, thus resetting `offset` to `0`. `cur_addr` is assigned `output_addr`, and we transition to `DELAY` state.

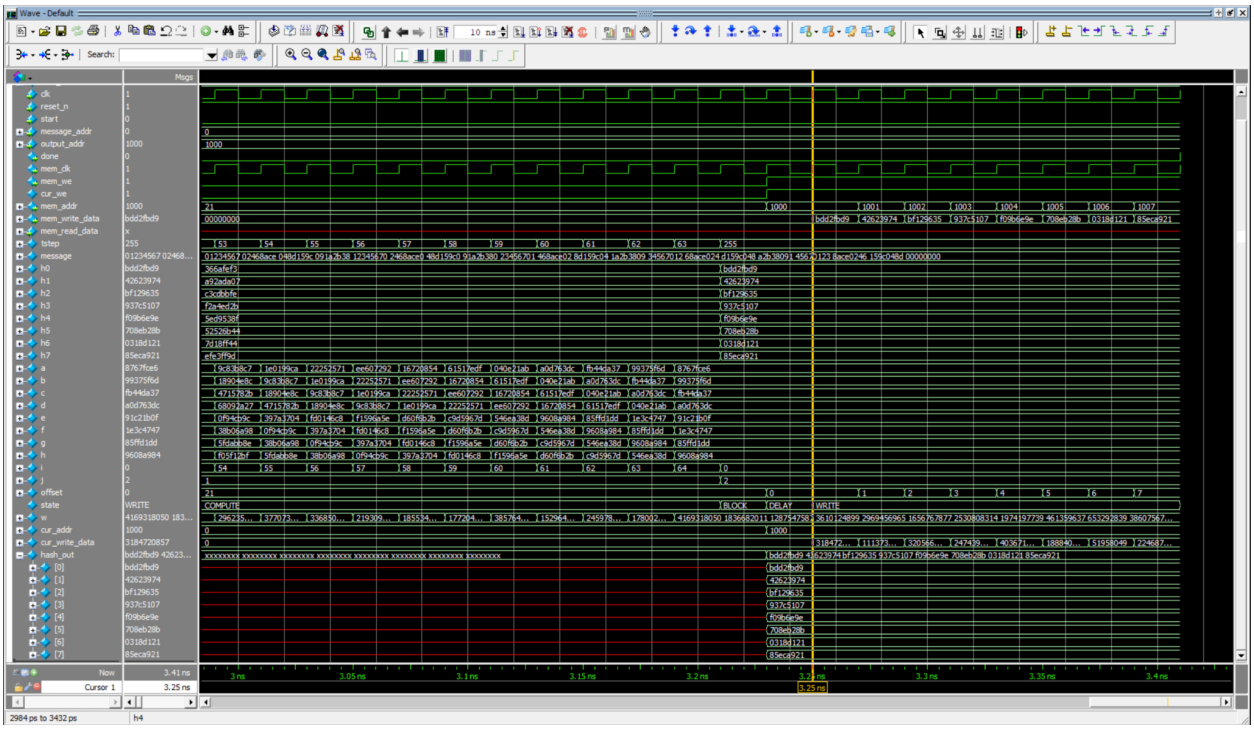
In `WRITE` state, a word is written one at a time into memory. `offset` is used to keep track of which memory address we are located at, hence the reason for incrementing it each time we move onto another address. In the end, once computing and writing is completed, we return to `IDLE` state.

III: ModelSim Simulation



Here we can see the behavior of the `simplified_sha256` module during the entire message-hashing process. The total time it takes to hash the message is approximately 3.25ns.





Here we view the SHA256 output array, `hash_out`, containing the correct hash results at the end of the simulation.

IV: ModelSim Transcript

Transcript

```
# Loading work.tb_simplified_sha256
# Loading work.simplified_sha256
add wave sim:/tb_simplified_sha256/*
VSIM 6> run
# Invalid state! SHA256 FSM remains in IDLE.
# --------
# 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    : C:/Users/lasiru/Desktop/simplified_sha256/tb_simplified_sha256.sv(262)
#    Time: 3410 ps  Iteration: 2  Instance: /tb_simplified_sha256
```

Now: 3,410 ps Delta: 2 | offset

Part 2: Bitcoin Hashing

I: What is Bitcoin Hashing?

A blockchain is a chain of digital blocks of data, which are immutable. Bitcoin is the oldest blockchain, consisting of data regarding Bitcoin transactions. The Bitcoin blockchain is dependent on SHA-256 hashing to chain the data blocks together and ensure that the transaction data is kept securely.

II: Bitcoin Hashing SystemVerilog Algorithm

The bitcoin hashing algorithm is based on the SHA-256 hashing algorithm. Our first functions `sha256_op`, `rightrotate`, and `exp_word` are all the same as in Part 1.

In `IDLE` state, similar to SHA-256, `h0-h9`, `a-h`, `i`, `j`, `cur_we`, `cur_write_data`, and `offset` are initialized to `0`, when `start = 1`. Also, final hash values `Hf0[m]-Hf7[m]`, `iter`, and placebo placeholder values `A[m]-H[m]` are initialized to `0` during the 16 loop repeats. The address for reading data from `message_addr` is given to `cur_addr`, and the `DELAY` state is begun. However, if `start = 0`, we stay in `IDLE` state.

In `WAIT` state, the process is the same as SHA-256, where an if-else statement, signaled by the value of `cur_we`, determines whether to start writing hash value data or begin reading data. If `cur_we = 1`, we move to `WRITE` state. Otherwise, we move onto `READ` state.

In `READ` state, the message is copied from the `mem_read_data` variable. Afterwards, we move onto `BLOCK` state.

In the first `BLOCK` state, the first and second message blocks are read. To do this, we fetch the message and initialize a hash value computation:

- As `j == 0`, this indicates phase 1, the first message block `w` is assigned the message and variables `h0-h7` and `a-h` are assigned their hash values. We then move onto `COMPUTE` state.

- As `j != 0`, we assign `hash2` variable to `0` and assign three more input words, in the form of `message[16]-message[18]` to the first three placeholder placebo variables `w[0][m]-w[2][m]`. A fourth placeholder placebo variable `w[3][m]` is assigned a nonce value and the rest are given passing values. Once `m` is incremented, it is passed onto `COMPUTE2` state each time.

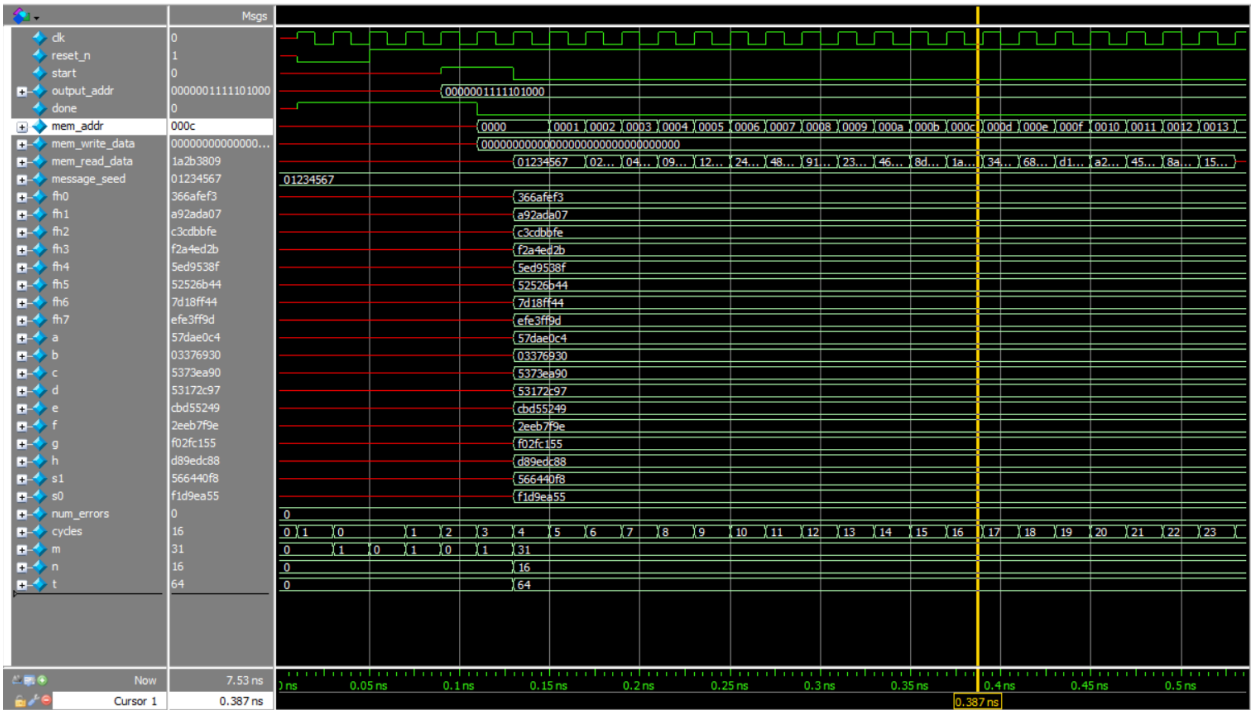
- As `j >= num_blocks`, we assign `h0-h7` to `Sha256[0:7]`, `cur_we` to `1`, `offset` is reset to `0`, `cur_addr` is assigned to `output_addr`, and we commence `WAIT` state.

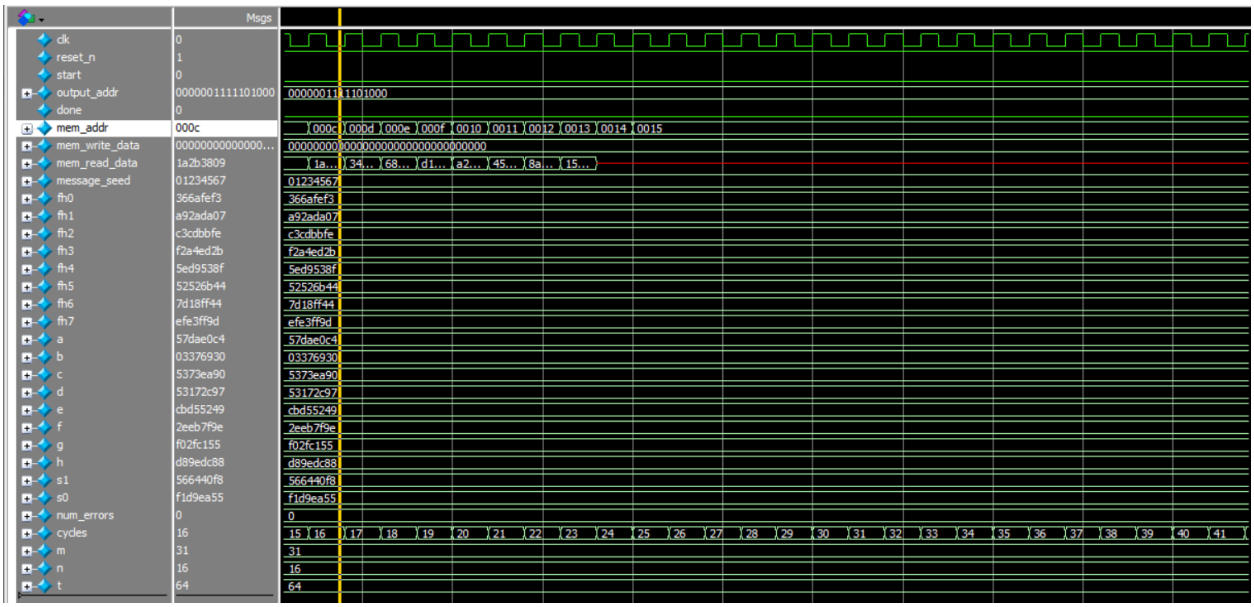
In `COMPUTE` state, the first phase of block hashing happens. The first 15 SHA256 and the remaining 49 SHA256 are hashed using `exp_word`. If `j <= 0`, then the first hashing has happened and will return to the `BLOCK` state. However, if `j > 0`, we transition to the `BLOCK2` state.

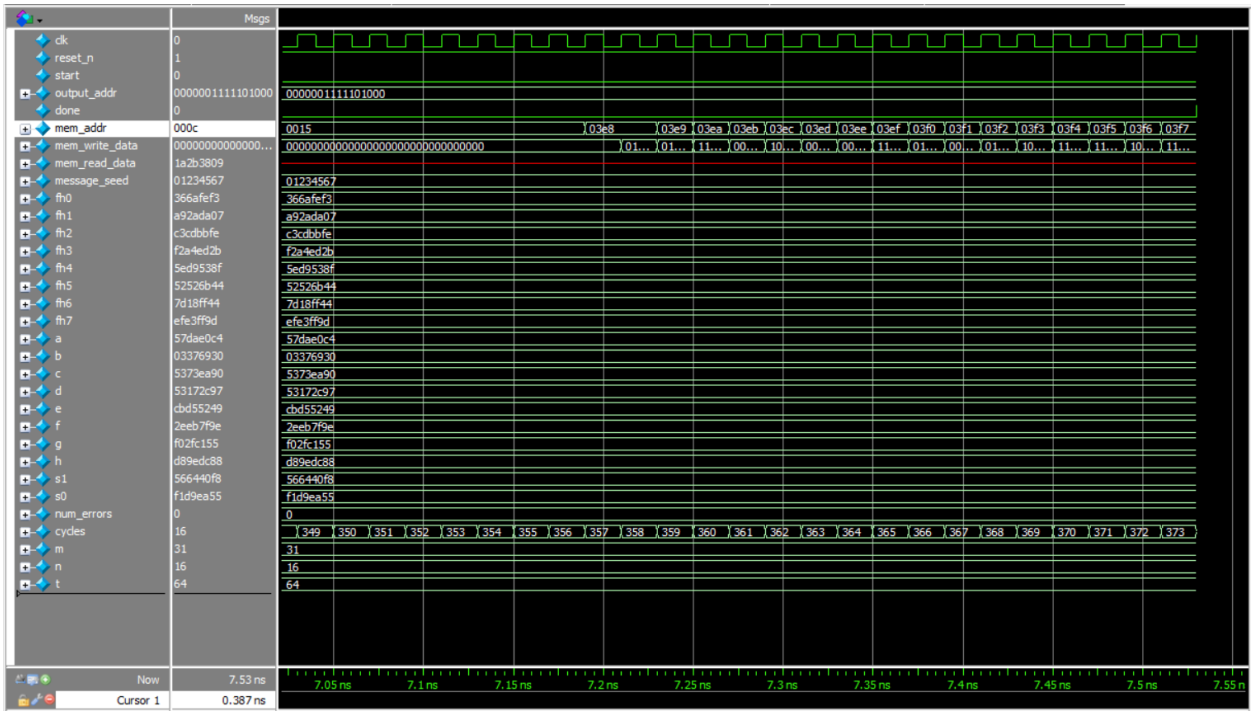
`COMPUTE2` state contains parallel hashing. First, the 15 SHA256 and remaining 49 SHA256 are hashed. The `Hash2` variable is used to determine what state we commence to next. `Hash2` is given the value of `0`, when the second hashing starts, to move to `BLOCK2` and commence reading. If `hash2 == 1`, `BLOCK2` is recognized as being passed and the third hashing will begin. This in turn assigns the final hash values to `Hf0[m]-Hf7[m]`, and we return to `BLOCK` state to repeat the process until `j < num_blocks`.

In `WRITE` state, we write one word at a time into memory. `Offset` is used to keep track of memory address location, hence why it is incremented with each iteration. Once we are done writing and computing, we return to the `IDLE` state.

III: ModelSim Simulation







`Hf0` array is not outputted in the modelsim waveform, but the modelsim transcript output is correct, which means the array is being assigned correctly and the output array is correct.

IV: ModelSim Transcript

```
VSIM 5> run
# ---------------
# 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:         374
#
#
# ******************************
```

V: Synthesis: Resource Usage & Timing Report

	Resource	Usage
1	Estimated ALUTs Used	12545
1	-- Combinational ALUTs	12545
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	8383
3		
4	Estimated ALUTs Unavailable	1609
1	-- Due to unpartnered combinational logic	1609
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	12545
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	2185
3	-- 5 input functions	14
4	-- 4 input functions	1609
5	-- <=3 input functions	8737
8		

9	⌄ Combinational ALUTs by mode	
1	-- normal mode	6752
2	-- extended LUT mode	0
3	-- arithmetic mode	5217
4	-- shared arithmetic mode	576
10		
11	Estimated ALUT/register pairs used	17922
12		
13	⌄ Total registers	8383
1	-- Dedicated logic registers	8383
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	118
17		
18	DSP block 18-bit elements	0
19		

20	Maximum fan-out node	clk~input
21	Maximum fan-out	8384
22	Total fan-out	84418
23	Average fan-out	3.99

VI: Fitter Report

Fitter Status	Successful - Sat Jun 04 21:11:01 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	bitcoin_hash
Top-level Entity Name	bitcoin_hash
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	53 %
Total registers	8383
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 %)

VII: Timing Fmax

Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	108.85 MHz	108.85 MHz	clk	