

## Final Report

### SHA-256 and Bitcoin Hashing Summary Explanation

SHA-256 is a simplified hash algorithm that takes in any sized input “message” and outputs some encrypted hash value. In this case, the program is further simplified as we are hard-coded with a message word length of 20, so anything more or less would not work for the algorithm below. The program outputs an eight 32-bit word hashed value.

The Bitcoin Hashing program is another type of hashing algorithm that takes in 20 words (19 words and 1 nonce value) and outputs an eight 32-bit word hashed value using the sha-256 algorithm multiple times.

### SHA-256 and Bitcoin Hashing Algorithm Explanation

#### SHA-256 Algorithm Explanation

First we initialize the hash constants  $k$  which is an array of 32 bits values with a size of 64 ([0:63]). We also write helper functions `wtnew`, `rightrotate`, and `sha256_op` which will be essential for our program. The `wtnew` function uses `rightrotate` to get a new word and the `sha256_op` function uses `rightrotate` to get new hash values. After creating and filling in some local variables that we use in the program, we can proceed to the case statements which our program will be able to use to read our inputs, calculate our hash values from the inputs, and write them to memory.

We start off in our initial state IDLE. Here we initialize eight 32 bit hash values with some constants and set the done and write enabler flag to 0. We also set the current address to equal the message address given and reset all of our index values. We do not leave this state until the start variable given triggers. When it is triggered, we move to the READ state.

In the READ state, we begin by reading in the message given one word at a time into our message array. We cycle through this state using offset and until we have read the entire message, we do not move to the BLOCK state.

In the BLOCK state, if we have yet to assign the words in message to  $w$ , we assign the first 16 words into  $w[0:15]$ . If words from message have already been assigned to  $w$  before, we set the last 4 messages to  $w$  and fill the rest of the spaces with a constant followed by zeros for padding. In both cases, we set the  $a-h$  variables to the  $h1-h7$  has values respectively and move to the COMPUTE state after setting  $i \leq 0$  for iteration in the COMPUTE state. If we have already gone through the BLOCK state twice, then instead of assigning values to  $w$  and fetching words, we immediately go to the WRITE state.

In the COMPUTE state, we use the `sha256_op` function to give our given  $a-h$  values from the BLOCK state different hash values 64 times using  $i$  for every value within  $w$ . However since  $w$  is

an array of only size 16, we have to perform a word expansion using the wtnew function and use it to constantly change the w[15] value. This means that we change the a-h values 64 times even though w is only size 16. After running through the sha256\_op function 64 times, we add the completely altered a-h values to the h1-h7 values changing them and then move back to the BLOCK state.

The WRITE state should only be arrived to after the BLOCK state has ran twice so that all of the words in the message have gone through the sha256\_op function. Here, we set the write enabler to 1 and the current address to the output address given and write each hash value we have (h1-h7). This is done using the offset variable to update the address which was reset before arriving to this state and using mem\_write\_data to write each value one-by-one. Once the eight value is written, then the next state would be the IDLE state, which also triggers the done flag.

### Bitcoin Hash Algorithm

Much like the sha256 file, we initialize k and write the same helper functions, along with the same variables. However, this time there are more state cases, the hash values h0-h7 are now arrays of size NUM\_NONCES and there is another set of hash values fh1-fh7. The helper functions wtnew, rightrotate, and sha256\_op, all serve the same purpose and have not changed from what they were in sha256.

The program starts off in the IDLE state which is very similar to how it was in the sha256 program, but this time, fh0-fh7 are initialized instead of h1-h7 and there is another iteration variable. The offset is reset, the write enabler is set to 0, the current address equals the message address given, and we wait until start equals 1 before heading to the next state, READ.

The READ state is exactly like how it was in the sha256 program, but instead we are moving to the BLOCK\_1 state. We begin by reading in the message given one word at a time into our message array. We cycle through this state using offset and until we have read the entire message, we do not move to the BLOCK\_1 state.

In the BLOCK\_1 state we fill in a-h with fh0-fh7 and fill in the first 16 words of message into w. We assign j, which we use as an indicator to know where we are in the program, to equal 0 and move to the COMPUTE state after setting i to equal 0 for iteration.

In the BLOCK\_2 state we fill in h0[m]-h7[m] and a-h with fh0-fh7 and assign the last 4 words in message to w along with the current value of m in w[3], followed by a constant and zeros after to fill in the rest of the values within w[0:15]. We assign j to equal 1, i to equal 0, and set the next state to be the COMPUTE state. This block should be called 16 times with m being the indicator for how many times this state has been called.

In the BLOCK\_3 state, w[0:7] are filled with the values in h0[m]-h7[m]. w[8] equals a constant followed by zeros for padding. The last value in w, w[15] equals 256. After which, h0[m]-h7[m] and a-h are assigned the same constant hash values of what fh0-fh7 was in the IDLE state, j now

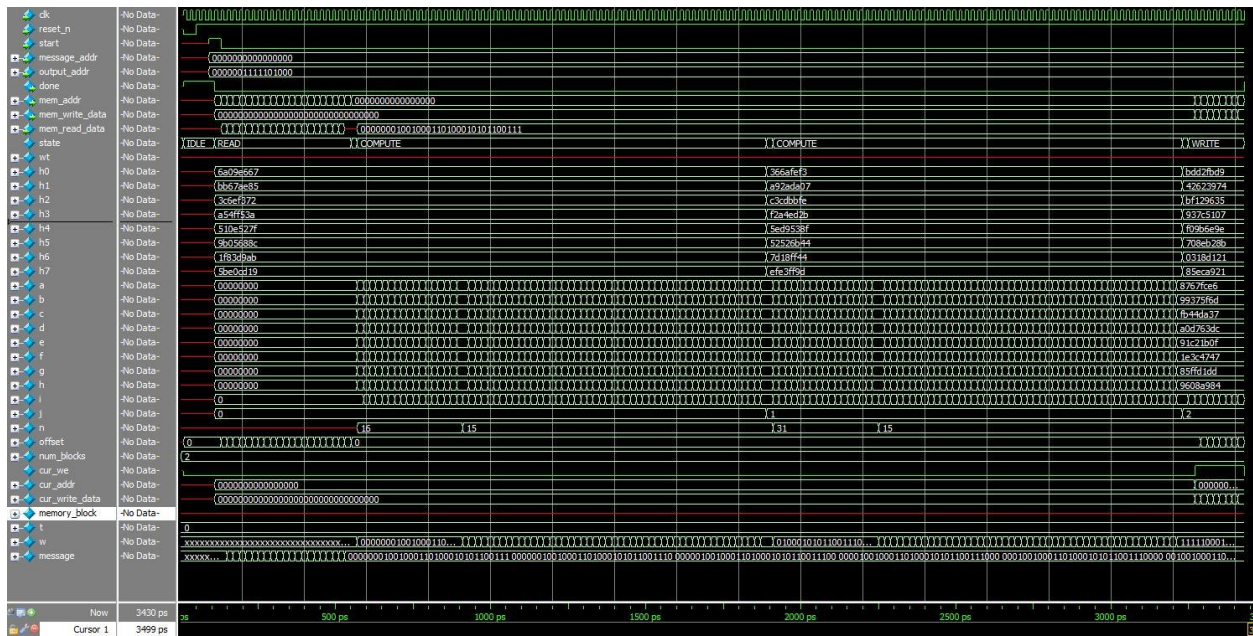
equals 2, i equals 0 for iteration, and then the next state equals COMPUTE. This block should be called 16 times with m being the indicator for how many times this state has been called.

In the COMPUTE state, no matter what j equals or which state came before the current COMPUTE state, they all go through the sha256\_op function 64 times using i for iteration and like how it was in the sha256 program, after performing the sha256\_op function 16 times, a word expansion (wtnew) is done on the last word in w (w[15]) and then the function used for each time the w[15] is changed until the sha256\_op is performed 64 times. After that, it depends on what j equals. As said before, j is used as an indicator on which BLOCK state was used to go to the compute state. After running the sha256\_op 64 times, if j equals 0, fh0-fh7 are incremented by whatever values a-h currently are, the next state equals BLOCK\_2, and m is assigned 0 for the NONCES loop. If j does not equal 0 and if BLOCK\_2 or BLOCK\_3 have finished 16 loop indicated when m is greater than or equal to NONCES-1, that means that the next state equals BLOCK\_3 and m is reset to 0 if j equals 1 or the next state equals the WRITE state and the variable t is assigned to zero if j equals 2. If m has not reached 15 (0-15 equals 16 cycles), that means the next state equals BLOCK\_2 if j equals 1 or the next state equals BLOCK\_3 if j does not equal 1. In both cases, m is incremented.

In the WRITE state, we are going through this state NONCES amount of time (16 times) using t as an indicator for how many times we have gone through the WRITE state. The write enabler is flagged, the offset equals t, and the current address equals the output address for every iteration. Every iteration we are using cur\_write\_data to output h0[t] to memory and incrementing t. If t is less than 16, then the next state is still the WRITE state, else the next state equals the IDLE state. Once it reaches the IDLE state, the done flag is triggered and the program ends.

## **SHA-256 and Bitcoin Hashing Waveform and Transcript**

### SHA-256 Wave and Transcript



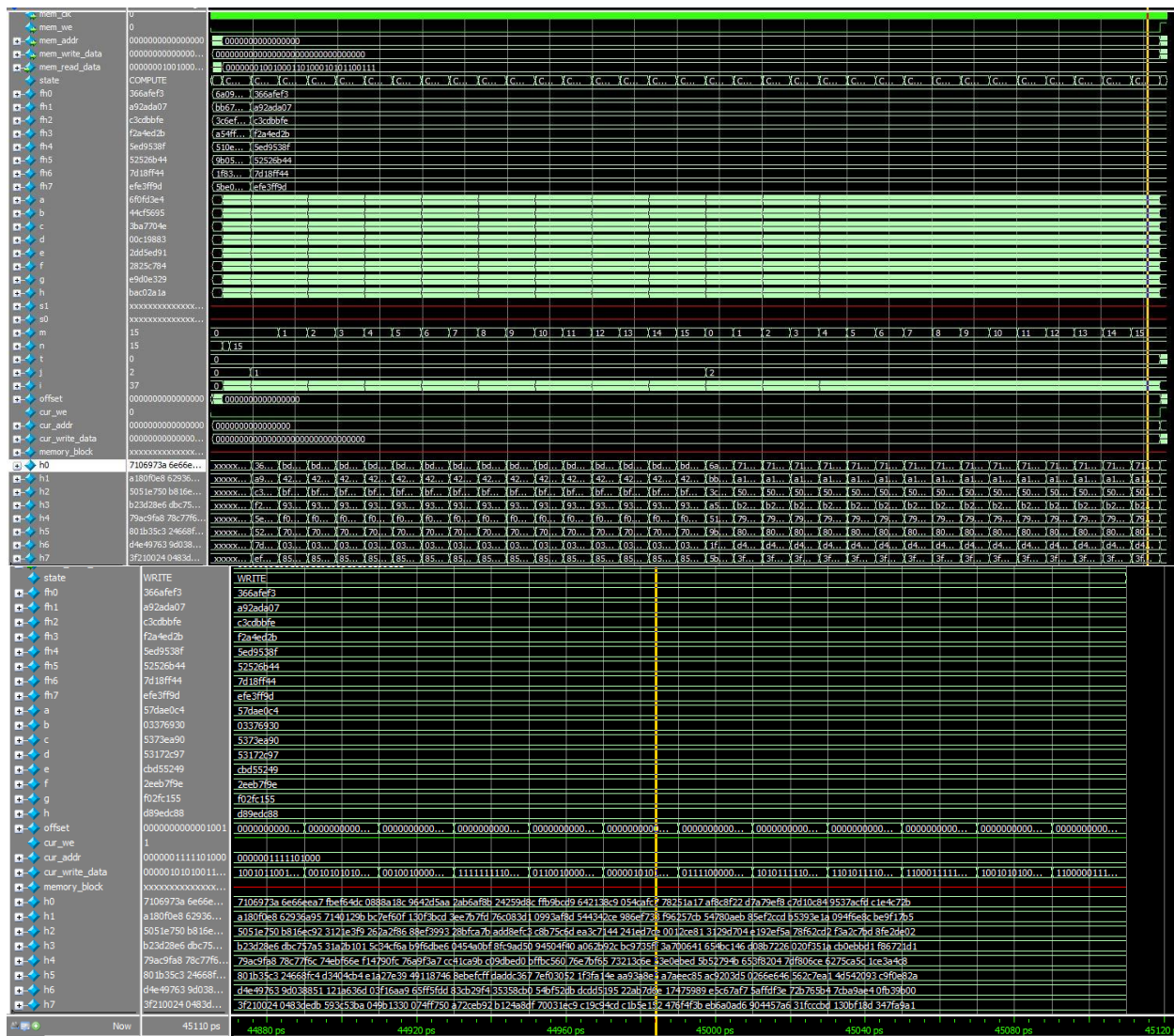
This is the waveform for the simplified\_sha256.v file which shows the final output hash values at the very right of the h0-h7 lines. We can see that after the values are calculated, the write enabler turns on and we right to the output address. After we are done writing, the done flag is triggered right before 3500 ps before the program ends.

```

#       simplified_sha256
# End time: 16:39:34 on Sep 12,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
VSIM 19> run -all
VSIM 20> restart
# Loading work.simplified_sha256
VSIM 21> 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!
#
# Total number of cycles:      169
#
# *****
#

```

## Bitcoin-hash Wave and Transcript



These are the waveforms for the bitcoin\_hash.sv file which shows the final output hash values of h0. The first wave is a picture of the values of the entire program from start to finish and the second wave is a picture of the very last moments of the program. We can see that after the values are calculated, the write enabler turns on and we right to the output address. After we are done writing, the done flag is triggered right around 450110 ps before the program ends.



```

#
# Top level modules:
#   bitcoin_hash
# End time: 16:17:07 on Sep 13,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
VSIM 9> restart
# Loading work.bitcoin_hash
VSIM 10> run -all
# -----
# 19 WORD HEADER:
# -----
# 01234567
# 02468ace
# 048dl59c
# 091a2b38
# 12345670
# 2468ace0
# 48dl59c0
# 91a2b380
# 23456701
# 468ace02
# 8dl59c04
# 1a2b3809
# 34567012
# 68ace024
# dl59c048
# 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] = c7dl0c84 Your H0[13] = c7dl0c84
# Correct H0[14] = 9537acfd Your H0[14] = 9537acfd
# Correct H0[15] = cle4c72b Your H0[15] = cle4c72b
# *****
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:      2253
#
#


```

## Bitcoin Hashing Synthesis Resource Usage and Timing Report

### Synthesis Resource Usage


	Resource	Usage			
1	▼ Estimated ALUTs Used	3598	13	▼ Total registers	5928
1	-- Combinational ALUTs	3598	1	-- Dedicated logic registers	5928
2	-- Memory ALUTs	0	2	-- I/O registers	0
3	-- LUT_REGS	0	3	-- LUT_REGS	0
2	Dedicated logic registers	5928	14		
3			15		
4	▼ Estimated ALUTs Unavailable	32	16	I/O pins	118
1	-- Due to unpartnered combinational logic	32	17		
2	-- Due to Memory ALUTs	0	18	DSP block 18-bit elements	0
5			19		
6	Total combinational functions	3598	20	Maximum fan-out node	clk~input
7	▼ Combinational ALUT usage by number of inputs		21	Maximum fan-out	5929
1	-- 7 input functions	32	22	Total fan-out	36462
2	-- 6 input functions	1796	23	Average fan-out	3.74
3	-- 5 input functions	83			
4	-- 4 input functions	459			
5	-- <=3 input functions	1228			
8					
9	▼ Combinational ALUTs by mode				
1	-- normal mode	2557			
2	-- extended LUT mode	32			
3	-- arithmetic mode	881			
4	-- shared arithmetic mode	128			
10					
11	Estimated ALUT/register pairs used	7781			
12					

## Fitter Report

Flow Summary	
 <<Filter>>	
Flow Status	Successful - Thu Sep 16 17:16:24 2021
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	21 %
Total registers	5928
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 % )



## Timing Report

Slow 900mV 100C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	129.33 MHz	129.33 MHz	clk	