Tashdeed Kader Faruk
Professor Vishal Karna
ECE 111
A14001958

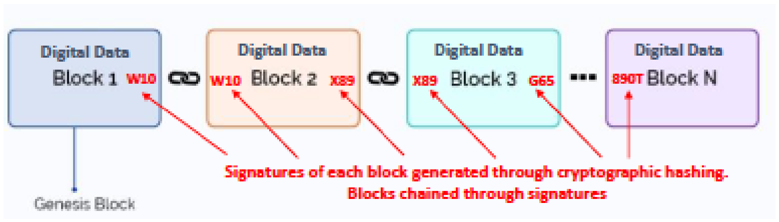
<u>ECE 111 – Advanced Digital Design Final Project:</u>
<u>SHA-256, Blockchain, and Bitcoin Hashing</u>

<u>Introduction:</u>

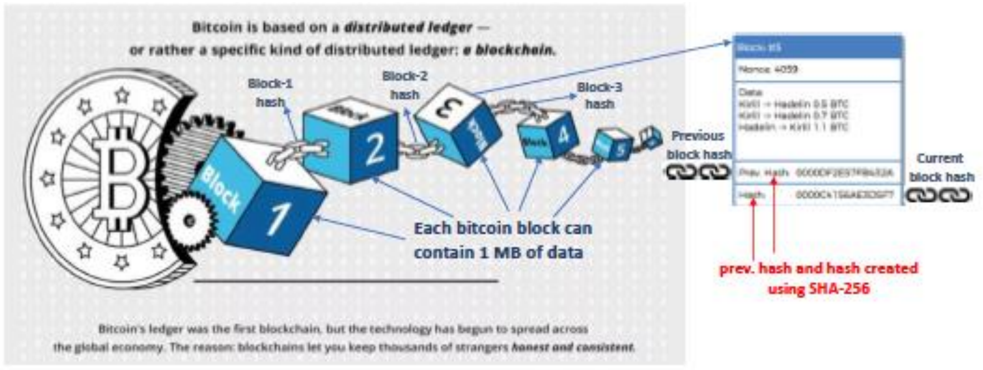
SHA-256 belongs to the SHA-2 (or Secure Hash Algorithm 2) set of cryptographic hash functions. A cryptographic hash function (CHF) is a mathematical algorithm that maps data of arbitrary size, commonly referred to as the "message", to a bit array of fixed size, commonly referred to as the "hash" or "hash value". Cryptographic hash functions are one-way functions, which means it is hypothetically infeasible to reverse the hashing process and obtain the original input message from the final hash. The previous property is often called "*Pre-Image Resistant*" and one of many main properties of ideal cryptographic hash functions. Others include:

1. *Compression*: The output hash should be a fixed number of characters, regardless of the input message size. For SHA-256, the input message can be up to 2^64 bits in size and the output hash are 256-bits.
2. *Avalanche Effect*: A minimal change in the input messages significantly changes the output hash. This helps prevent hackers from predicting the input message through trial and error of the output hash.
3. *Determinism*: The same input must always generate the same output by different computer systems. Therefore, any machine in the world which understands the specific secure hashing algorithm should produce the same output hash for the same input message.
4. *Efficiency*: Creating the output hash should be a fast process that doesn't require heavy use of computing power (i.e., supercomputers or high-end machines).
5. *Collision Resistance*: Should be <u>practically</u> impossible for two different input messages to produce the same output hash. Since the input message can be a large combination of values (2^64 bits) and the output is a much smaller fixed value (256-bits), it is mathematically possible to find two input messages with the same output hash.

Applications of SHA-256 include verifying file integrity and authentication, which involves the file provider sending the file receiver an output hash of the file for them to verify that they have received the correct file unharmed by hackers. Cryptographic hash functions are an important aspect of technological security as they are a basic tool of modern cryptography.

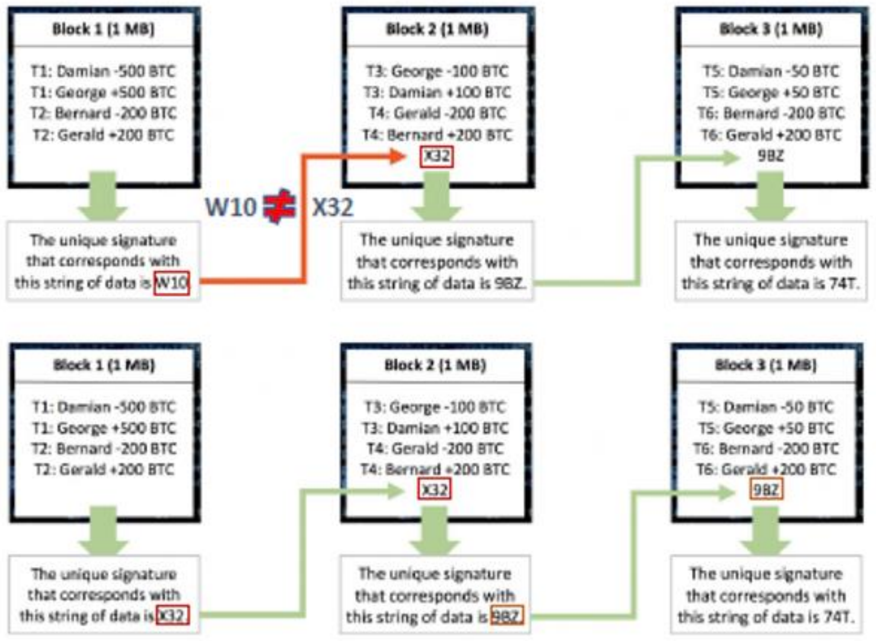


A Blockchain is a chain of digital data blocks. These digital data blocks can store information about financial transactions (such as date, time, dollar, sender, receiver), medical records, and property purchase deeds among many other things. Chaining of the blocks is achieved through cryptographic hashing algorithms like Scrypt and SHA-256. Once changed together, the data in blocks can never be changed again, which means the process is immutable. The entire blockchain is usually publicly available to anyone for viewing purposes and therefore, the blockchain essentially acts like a distributed and decentralized public ledger.


Bitcoin is based on a distributed ledger —
or rather a specific kind of distributed ledger: a blockchain.
Block-1 hash
Block-2 hash
Block-3 hash
Block
1
2
4
Each bitcoin block can contain 1 MB of data
Bitcoin's ledger was the first blockchain, but the technology has begun to spread across the global economy. The reason: blockchains let you keep thousands of strangers honest and consistent.
Nonce 4059
Data
Previous block hash
Current block hash
Prev. Hash
Hash
prev. hash and hash created using SHA-256

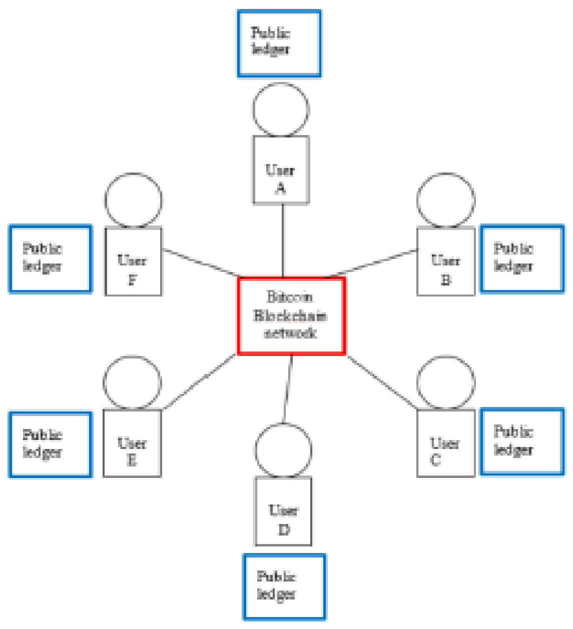
Blockchains are dependent on cryptographic hashing functions and a popular example of this is the Bitcoin Blockchain, which uses SHA-256 and is believed to be the oldest blockchain in existence. In the Bitcoin Blockchain, the digital data blocks consist of approximately 1 MB of transaction data. The Bitcoin Blockchain is a largescale track record of every Bitcoin transaction that has ever occurred, all the way back to the very first Bitcoin transaction.

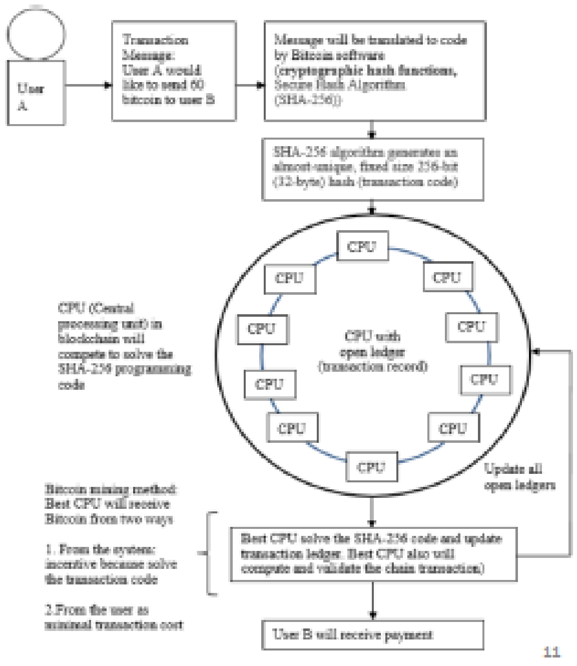
In the Bitcoin Blockchain, the signature of each blocks links the blocks to each other. Due to the Avalanche effect and Collision Resistance, any malicious attempt to alter a block in the blockchain will cause all users to reject this change by shifting back to their previous record of the blockchain where all the blocks are still chained together. This makes blockchains immutable.


Block 1 (1 MB)
T1: Damian -500 BTC
T1: George +500 BTC
T2: Bernard -200 BTC
T2: Gerald +200 BTC
Block 2 (1 MB)
T3: George -100 BTC
T3: Damian +100 BTC
T4: Gerald -200 BTC
T4: Bernard +200 BTC
X32
Block 3 (1 MB)
T5: Damian -50 BTC
T5: George +50 BTC
T6: Bernard -200 BTC
T6: Gerald +200 BTC
9BZ
W10 ≠ X32
The unique signature that corresponds with this string of data is W10
The unique signature that corresponds with this string of data is 9BZ.
The unique signature that corresponds with this string of data is 74T.
The unique signature that corresponds with this string of data is X32
The unique signature that corresponds with this string of data is 9BZ
The unique signature that corresponds with this string of data is 74T.

A signature does not always qualify for the block to be accepted into the block chain. Often a block will only be accepted onto the blockchain if its digital signature starts with – for example – 7 consecutive number of zeroes. In order to find a signature that meets the requirements needed for a block to be accepted, the data must be change repeatedly until a specific string is found that leads to the specific signature type – such as “starting with 10 zeroes” – that we are looking for. As the transaction data and metadata (i.e., block number, timestamp, etc.) must remain the same, a small specific piece of data is assigned to every block with the sole purpose of being change repeatedly until an eligible signature is found. This piece of data is called the *nonce* of a block. The nonce is a completely random string of numbers.

“Bitcoin Miners” has recently received a lot of attention because of the increasing popularity of Bitcoin and other cryptocurrencies in general. Essentially “bitcoin miners” provide services to customers looking to perform new bitcoin transactions. Bitcoin miners all have the same copy of the bitcoin blockchain on their local hardware, which is why the bitcoin blockchain is considered a de-centralized and distributed ledger, and they are each generating different hashes for a customer’s bitcoin transaction block until they meet the required acceptance criteria for a output hash value (i.e., the signature’s conditions). All bitcoin miners are competing with each other and they will keep using different nonce values until they are able to generate the hash value meeting a specific criterion. The first person to generate the correct output hash value will successfully add the customers transaction block to the bitcoin blockchain ledger and then collect either a service fee or reward in bitcoin. All the other bitcoin miners who did not win will then need to update their bitcoin blockchain ledger on their local hardware. All bitcoin miners usually invest in expensive local hardware consisting of several GPUs and CPUs and usually these "rigs"/setups consumer a lot of electricity and power, which is why bitcoin mining is often criticized as not being ecofriendly.




User A
Transaction Message: User A would like to send 60 bitcoin to user B
Message will be translated to code by Bitcoin software (cryptographic hash functions, Secure Hash Algorithm (SHA-256))
SHA-256 algorithm generates an almost-unique, fixed size 256-bit (32-byte) hash (transaction code)
CPU (Central processing unit) in blockchain will compete to solve the SHA-256 programming code
CPU
CPU with open ledger (transaction record)
Update all open ledgers
Bitcoin mining method: Best CPU will receive Bitcoin from two ways
1. From the system: incentive because solve the transaction code
2.From the user as minimal transaction cost
Best CPU solve the SHA-256 code and update transaction ledger. Best CPU also will compute and validate the chain transaction)
User B will receive payment
11

In our example below, we test 16 different nonces ranging from 0 to 15 and we store the first hash value h0 for each nonce. As opposed to finding a specific signature, we will just show that our algorithm can use the same data and metadata to produce different output hashes based on the changing nonce values.

SHA-256 Algorithm Explanation:

This implementation of SHA-256 involves a Finite State Machine (FSM) with five states: IDLE, READ, BLOCK, COMPUTE, and WRITE.

1. In the IDLE state we initialize hash values "h0"-to-"h7", the "block_num" (which is set to 0), "t" (which is set to 0), and "j" (which is set to 0). We set the current write enable variable "cur_we" to 0, as we will be reading in the next state, and the current address

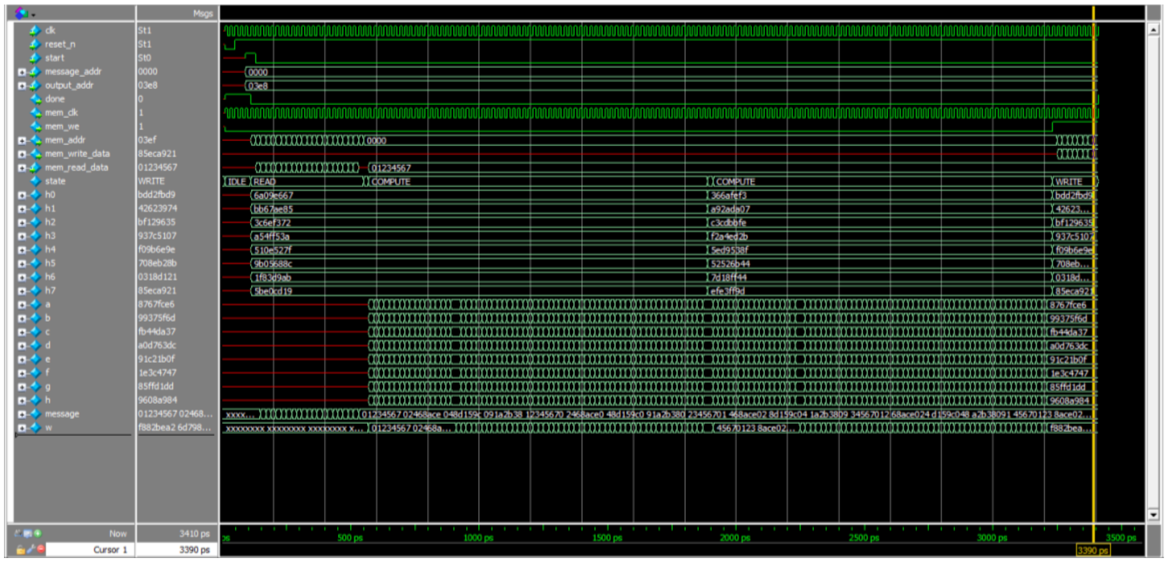
variable "cur_addr" is set to the start of the address variable given for the input message, "message_addr". Next the "state" variable, which controls the state of the FSM in the next positive edge of the clock cycle, is set to READ.

2. In the READ state we first skip a clock cycle before we begin reading in the input message words. The message input is assigned in the input variable "mem_read_data" where the word changes every clock cycle for 20 clock cycles, as there is 20 words in our message. The message is read into a local "message" variable the size of 20 32-bit words (20 arrays, each 32 bit in size) using an iterating "offset" variable. After the 20 words are copied into the local "message" variable, the "offset" variable is reset to 0, the "block_num" increments to 1, and the "state" variable is changed to BLOCK.
3. In the first encounter of the BLOCK state we begin by setting our "a"-to-"h" variables to our already set "h0"-to-"h7". Then depending on the value of the "block_num" variable, we execute different logic. In the first encounter of the BLOCK state, block_num = 1 and here we load the first 16 words from the local "message" variable into a local "w" variable. Then we move states to the COMPUTE state.
4. In the first encounter of the COMPUTE state, we perform 64 iterations of an optimized SHA-256 operation. For the first 16 iterations we perform SHA-256 operations with the input arguments of our function being: a, b, c, d, e, f, g, h, w[j], and k[j]. In the next iterations we shift use a for loop and a "wtnew" function to shift around and change the values inside the local "w" variable. After skipping the first iteration of this, we then perform SHA-256 operations with the input arguments of our function being: a, b, c, d, e, f, g, h, w[15], and k[j-1]. After this we take the sum of "a"-to-"h" and "h0"-to-"h7" and assign it back to "h0"-to-"h7". As this is our first iteration of the COMPUTE function, our "block_num" is set to 1. However we have 2 blocks as our input message is 20 words long and therefore, we end this state by incrementing "block_num" to 2 and going back to BLOCK stage.
5. In the second encounter of the BLOCK state we again begin by setting our "a"-to-"h" variables to our already set "h0"-to-"h7". Then depending on the value of the "block_num" variable, we execute different logic. In the second encounter of the BLOCK state, block_num = 2 and here we load the last 4 words from the local "message" variable into a local "w" variable, we set the next spot to "32'h80000000" to signify the padding

bits starting with a leading 1, then we add 10 spots of "32'h00000000" to act as padding bits, and the final spot of "w" is "32'd640" to signify the size of the message. Then we again move states to the COMPUTE state.

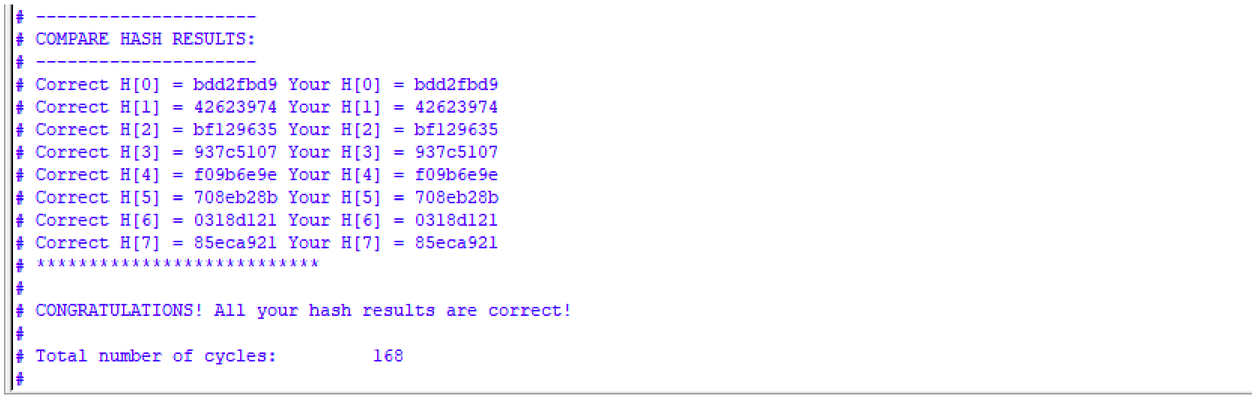
6. In the second encounter of the COMPUTE state, we again perform another 64 iterations of an optimized SHA-256 operation. For the first 16 iterations we perform SHA-256 operations with the input arguments of our function being: a, b, c, d, e, f, g, h, w[j], and k[j]. In the next iterations we shift use a for loop and a "wtnew" function to shift around and change the values inside the local "w" variable. After skipping the first iteration of this, we then perform SHA-256 operations with the input arguments of our function being: a, b, c, d, e, f, g, h, w[15], and k[j-1]. After this we take the sum of "a"-to-"h" and "h0"-to-"h7" and assign it back to "h0"-to-"h7". As this is our second iteration of the COMPUTE function, our "block_num" is set to 2. Now that we have already dealt with the SHA-256 calculations of our entire message, we set the current write enable to 1, as we will be writing to memory in the next state. We also set the "offset" variable to 0 and then move to the WRITE state.
7. In the WRITE state we write the "h0"-to"h7" variables to memory. We start by setting the current address variable to the given output address. Then at each clock cycle we write a different value to memory, starting from "h0" and iterating up to "h7". At the end we return to the IDLE state, which signifies to a separate assign statement that the "done" variable should be set to high.

SHA-256 Simulation Waveform:



	Msgs
clk	St1
reset_n	St1
start	St0
message_addr	0000
output_addr	03e8
done	0
mem_clk	1
mem_we	1
mem_addr	03ef
mem_write_data	85eca921
mem_read_data	01234567
state	WRITE
h0	bdd2fbd9
h1	42623974
h2	bf129635
h3	937c5107
h4	f09b6e9e
h5	708eb28b
h6	0318d121
h7	85eca921
a	8767fce6
b	99375f6d
c	fb44da37
d	a0d763dc
e	91c21b0f
f	1e3c4747
g	85ffd1dd
h	9608a984
message	01234567 02468...
w	f882bea2 6d798...
Now	3410 ps
Cursor 1	3390 ps

SHA-256 Simulation Transcript:



```
# ---------------------
# 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
#
```

Bitcoin Hashing Algorithm Explanation:

This implementation of Bitcoin Hashing involves a Finite State Machine (FSM) with nine states: IDLE, READ, PHASE1_BLOCK, PHASE1_COMPUTE, PHASE2_BLOCK, PHASE2_COMPUTE, PHASE3_BLOCK, PHASE3_COMPUTE and WRITE.

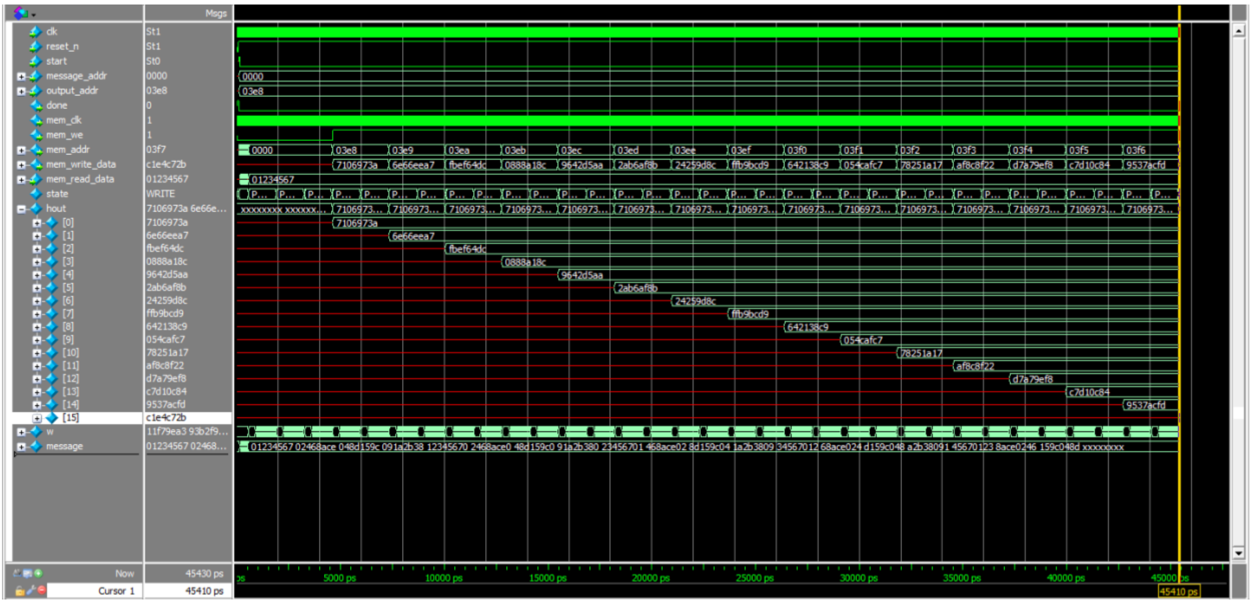
1. In the IDLE state we initialize hash values "h0"-to-"h7", the "block_num" (which is set to 0), "t" (which is set to 0), and "j" (which is set to 0). We set the current write enable variable "cur_we" to 0, as we will be reading in the next state, and the current address variable "cur_addr" is set to the start of the address variable given for the input message, "message_addr". The "nonce_count", which tracks the nonce that we are dealing with in late states, is initialized to 0. Next the "state" variable, which controls the state of the FSM in the next positive edge of the clock cycle, is set to READ.
2. In the READ state we first skip a clock cycle before we begin reading in the input message words. The message input is assigned in the input variable "mem_read_data" where the word changes every clock cycle for 20 clock cycles, as there is 20 words in our message. The message is read into a local "message" variable the size of 20 32-bit words (20 arrays, each 32 bit in size) using an iterating "offset" variable. After the 20 words are copied into the local "message" variable, the "offset" variable is reset to 0, the "block_num" increments to 1, and the "state" variable is changed to PHASE1_BLOCK.
3. In the PHASE1_BLOCK state we begin by setting our "a"-to-"h" variables to our already set "h0"-to-"h7". Then we load the first 16 words from the local "message" variable into a local "w" variable. Then we move states to the PHASE1_COMPUTE state.

4. In the PHASE1_COMPUTE state, we perform 64 iterations of an optimized SHA-256 operation. For the first 16 iterations we perform SHA-256 operations with the input arguments of our function being: a, b, c, d, e, f, g, h, w[j], and k[j]. In the next iterations we shift use a for loop and a "wtnew" function to shift around and change the values inside the local "w" variable. After skipping the first iteration of this, we then perform SHA-256 operations with the input arguments of our function being: a, b, c, d, e, f, g, h, w[15], and k[j-1]. After this we take the sum of "a"-to-"h" and "h0"-to-"h7" and assign it back to "h0"-to-"h7" and then end this state by moving to state PHASE2_BLOCK.
5. In the PHASE2_BLOCK state we begin by setting our "a"-to-"h" variables to our already set "h0"-to-"h7". Then we check if we are on the first nonce (i.e., nonce = 0) and if this condition is met we save the "h0"-to-"h7" values in local "h0_block1"-to-"h7_block1" variables for use later. Then we load the next 3 words from the local "message" variable into a local "w" variable. Next we set the fourth spot of the "w" variable to our nonce value, the fifth spot to "32'h80000000" to signify the padding bits starting with a leading 1, the next 10 spots are "32'h00000000" to act as padding bits, and the final spot of "w" is "32'd640" to signify the size of the message. Then we move to the PHASE2_COMPUTE state.
6. In the PHASE2_COMPUTE state, we perform 64 iterations of an optimized SHA-256 operation which is the same as those in PHASE1_COMPUTE. After this we skip one clock cycle before taking the sum of "a"-to-"h" and "h0"-to-"h7" and assign it back to "h0"-to-"h7". Then we save the "h0"-to-"h7" values in local "h0_block2"-to-"h7_block2" variables for use later and we change the "h0"-to-"h7" to the default values set in the IDLE state. Next, we move states to the PHASE3_BLOCK state.
7. In the PHASE3_BLOCK state we begin by setting our "a"-to-"h" variables to our already set "h0"-to-"h7". Next, we load the first 7 values of "w" with the previously set "h0_block2"-to-"h7_block2" variables. Then eighth spot is set to "32'h80000000" to signify the padding bits starting with a leading 1, the next 6 spots are "32'h00000000" to act as padding bits, and the final spot of "w" is "32'd256" to signify the size of the message. Then we move to the PHASE3_COMPUTE state.
8. In the PHASE3_COMPUTE state, we perform 64 iterations of an optimized SHA-256 operation which is same as those in PHASE1_COMPUTE. After this we skip one clock

cycle before taking the sum of "a"-to-"h" and "h0"-to-"h7" and assign it back to "h0"-to-"h7". Next, we move states to the write state.

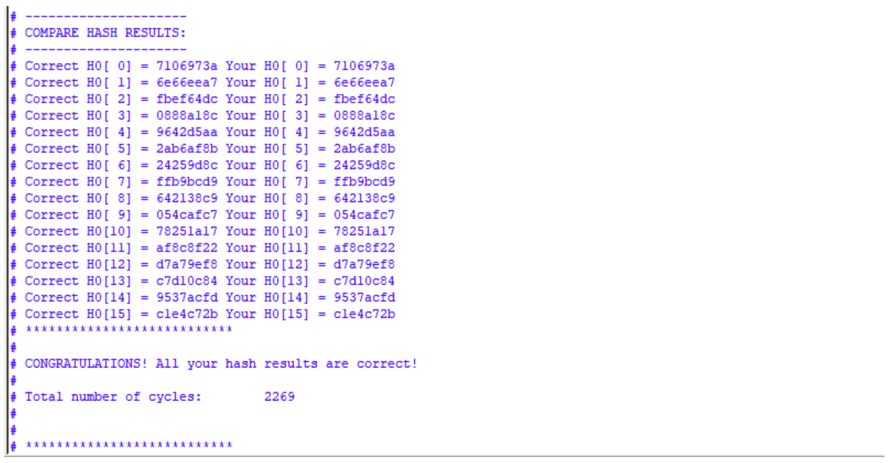
8. In the WRITE state we begin by changing the write enable to 1, which means we are writing to memory, and then we write the current value of "h0" to memory. Then we check if the nonce_count is less than 15 and perform different logic based on this conditional statement. If the nonce_count is less than 15 we increment the nonce_count, load the "h0_block1"-to-"h7_block1" values into "h0"-to-"h7" and then return to PHASE2_BLOCK. If the nonce_count is equal to 15 then we are on our last nonce value and therefore we reset some variables and return to the IDLE state, which signifies to a separate assign statement that the "done" variable should be set to high.

SHA-256 Simulation Waveform:



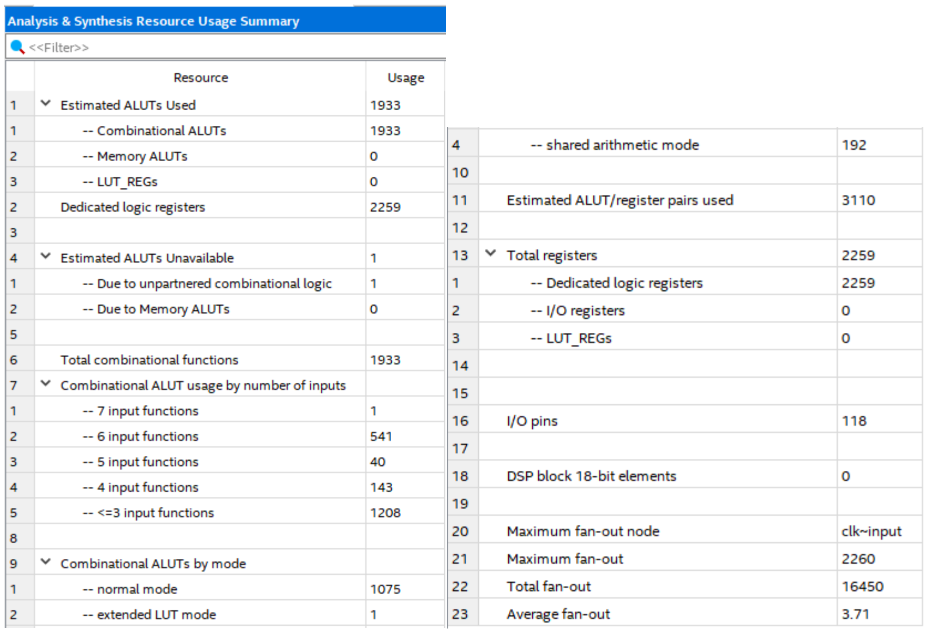
```
                   Msgs
clk                St1
reset_n            St1
start              St0
message_addr       0000
output_addr        03e8
done               0
mem_clk            1
mem_we             1
mem_addr           03f7
mem_write_data     c1e4c72b
mem_read_data      01234567
state              WRITE
hout               7106973a 6e66e...
  [0]              7106973a
  [1]              6e66eea7
  [2]              fbef64dc
  [3]              0888a18c
  [4]              9642d5aa
  [5]              2ab6af8b
  [6]              24259d8c
  [7]              ffb9bcd9
  [8]              642138c9
  [9]              054cafc7
  [10]             78251a17
  [11]             af8c8f22
  [12]             d7a79ef8
  [13]             c7d10c84
  [14]             9537acfd
  [15]             c1e4c72b
w                  11f79ea3 93b2f9...
message            01234567 02468...

Now               45430 ps
Cursor 1          45410 ps
```

SHA-256 Simulation Transcript:



```
# ---------------------
# 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:        2269
#
#
# ***************************
```

Resource Usage Summary:

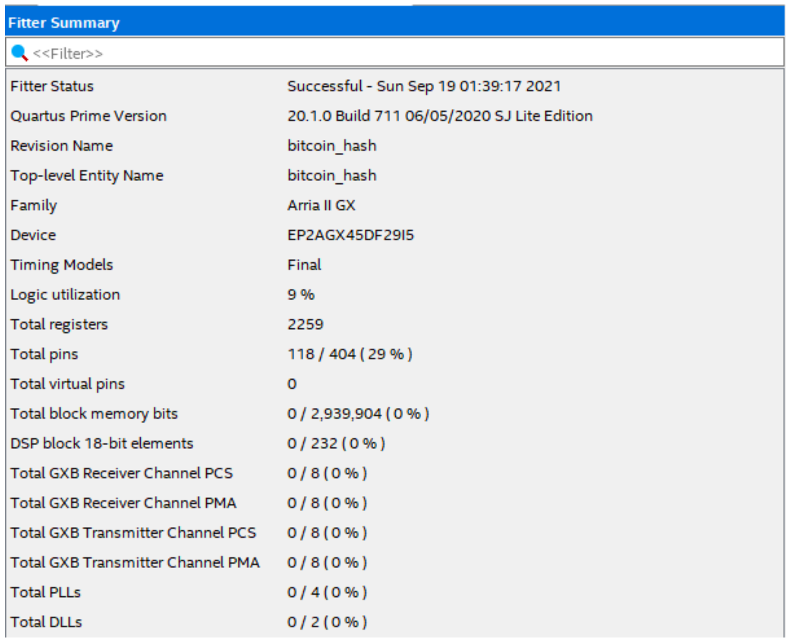


Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated ALUTs Used	1933
1	-- Combinational ALUTs	1933
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	2259
3		
4	Estimated ALUTs Unavailable	1
1	-- Due to unpartnered combinational logic	1
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	1933
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	1
2	-- 6 input functions	541
3	-- 5 input functions	40
4	-- 4 input functions	143
5	-- <=3 input functions	1208
8		
9	Combinational ALUTs by mode	
1	-- normal mode	1075
2	-- extended LUT mode	1
4	-- shared arithmetic mode	192
10		
11	Estimated ALUT/register pairs used	3110
12		
13	Total registers	2259
1	-- Dedicated logic registers	2259
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	2260
22	Total fan-out	16450
23	Average fan-out	3.71

Fitter Report Summary:

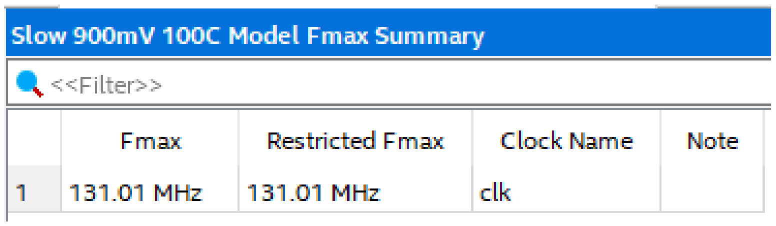


Fitter Summary

<<Filter>>

Fitter Status	Successful - Sun Sep 19 01:39:17 2021
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	bitcoin_hash
Top-level Entity Name	bitcoin_hash
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	9 %
Total registers	2259
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 Fmax (Slow 900 mV 100C Model) Report Summary:



Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	131.01 MHz	131.01 MHz	clk	