ECE 111 Final Project Report
Bitcoin Hasher

Hamad Alajeel, Mack Markham, & Isabel Senske
A16369878, A16391630, & A15876840

➢ Explanation of What Bitcoin Hashing Is:

The Bitcoin blockchain is the secure system Bitcoin uses to keep track of all transactional data that has occurred since the start of Bitcoin. This blockchain uses the SHA-256 hash to chain together the blocks, which are immutable once they have been chained. This works because the hash of the second block depends on the first block, which is what makes the blockchain immutable – if data were to be altered in the first block, it would change the hash of the second block. This would unchain the two blocks, alerting Bitcoin users that a change had been made, and they would then revert to their original version of the blockchain before the malicious user could alter the hashes of the subsequent blocks. Users hire Bitcoin miners to correctly add their transactional data to the blockchain. However, to add a block to the blockchain, it needs a specific signature to be accepted – this is where the hashing comes in. To match the correct starting signature, a completely random string of numbers called a nonce is added to the end of the transactional data in order to change the output hash. Bitcoin miners run the algorithm for multiple different nonce values until they generate an output hash with the desired signature, then receive payment for adding the transactional data to the blockchain.

➢ Explanation of Code Implementation, and Algorithm:

We will begin by explaining fundamental things about our design that are important to know to understand our implementation. Firstly, this design has two modes, mode 0 and mode 1. By default, we are using mode 0. These modes are chosen by changing a parameter that has been added to our design. This is the "mode" parameter. This mode parameter changes a couple of things regarding the design and the implementation. At mode 0, we instantiate one sha256 for stage 1 and then another amount of instantiations equaling the number of nonces for stages 2 and 3. So, at mode 0, at stages 2 and 3, the hashes for all the nonces are completed all at once and are ready to be used. On the other hand, for mode 1 we instantiate one sha256 for stage 1 and then another amount of instantiations equaling the number of nonces divided by 2. So, for stages 2 and 3 there are two phases each. One phase to process the hashes for nonces 0 - 7 and then another phase to process the nonces 8 - 15. This way we decrease the amount of area used on the FPGA, but in return we increase the amount of cycles. In our design the default is mode 0 because only when the amount of sha256 instantiations is equal to the amount of nonces and all hashes are computed at once at each stage will we be able to complete this bitcoin hashing operation in under 250 cycles which we did achieve. This is in return for more area needed because there are more instantiations. Therefore, this parameter, “mode”, changes this aspect of the design. It also determines the values of two other parameters, para1 and hash_complete_para, which are used as constants and indexes that determine the sizes of arrays that are inputs and outputs to the instantiations, as well as how many times for loops in different states of the design run for. Moreover, the mode choice affects which states in our FSM are used and which aren’t used. So, I will now begin to explain the states we have chosen for our design.

There are 8 states in our FSM: IDLE, READ, STAGE11, STAGE21, STAGE22, STAGE31, STAGE32, and WRITE. Of these 8 states 2 may not be used depending on the choice of “mode”. If you have chosen mode 1, then we need two phases for stages 2 and 3, corresponding to processing hashes for the first 8 nonces and then for the last 8 nonces. STAGE21 and STAGE31 process the first 8 nonces, and STAGE 22 and STAGE 32 process the last 8 nonces. Our FSM is implemented using an always_comb block that determines the next_state of the process using the signals from the instantiations that signal that the sha256’s have finished processing new hashes and we are ready to transition.

At IDLE, we wait for start, then if start is 1, we initialize our initial hashes and transition to READ where we begin reading from memory using the same pipelining method we used in part 1 of this 2-part project. This pipelining method takes into account the fact that when we send a command to the memory to obtain data from some memory address, it takes one extra clock cycle for the data to be available to mem_write_data. We then input that data to the message array at an index of offset - 1. When all the data has been read, we prepare the 16 word array to be used in processing hashes for the first stage of the bitcoin hashing operation. These 16 words correspond to the first 16 words we have read from memory. We complete this and transition to STAGE11 at the same time where we begin processing the hashes that will be used in stage 2.

At STAGE11 we wait for sha256 inst1 to signal to us that it has finished producing hashes, then we read them to the array hash_in the input to the sha256 instantiations for stage 2 which will be the initial hash values they use. Depending on if we have chosen mode 0 or mode 1, things may differ. If we have chosen mode 0, then we transition to STAGE32 not STAGE31. Moreover, STAGE11 prepares the words for 16 nonces, not the first 8, and inputs that to the array w_block which holds the words for each of the 16 or 8 instantiations depending on the mode. We switch begin_stage1 to 0 because we don't need this instantiation to do anything anymore. The words are prepared in this manner: first 3 words correspond to the last words we have read from memory, then the 4th word is the nonce for the corresponding sha256 inst, and then the usual padding that we do (1, ..000.., message size).

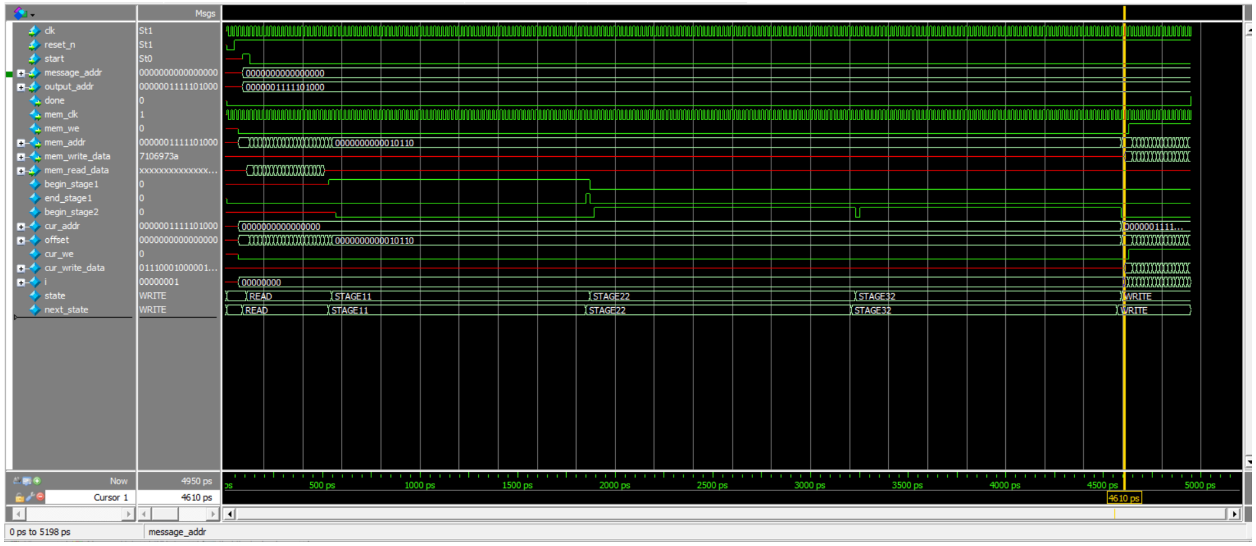
If we have transitioned to STAGE21, we process the hashes of the second stage for the first 8 nonces then we prepare the words for the next 8 nonces that will be computed at the state STAGE22. We input the completed hashes into the array hash_complete which will now hold the hashes of the first 8 nonces for stage 2. Note we initialize the begin_stage2 signal to 1 because if we don't the sha256 instances will begin processing hashes with the words and hashes that were used in the computation for stage 21. This is also done in stages 22 and 31 because, again, if we don't, then the sha256 instances will begin processing words and hashes used for the stage it has currently completed, instead of the words and hashes for the next stage. This is also why our modified version of the sha256 module contains a state, FINISH, which acts as a buffer to wait for the begin_stage signal to transition to 0.

At stage 22, two things could happen, again, depending on the mode. If we have chosen mode 0, then we compute the hashes of all 16 nonces for stage 2 here and we prepare the words and hashes for all 16 nonces in stage 3. Otherwise, for mode 1, we prepare the words and hashes for only the first 8 nonces for stage 3. These differences are implemented using the parameters we have talked about previously: para1 and hash_complete_para. These are assigned values based on a ternary operation that uses the mode value, 0 or 1, as its condition. So, we transition to STAGE32 if we have chosen mode 0 and STAGE31 if we have chosen mode 1. For stage 3, we use the original hashes we used for stage 1 of the bitcoin hashing operation, and for the words that will be used, we use the 8 hashes of each nonce as the first 8 words, then we pad as we usually do (1, ...000…, message size).

STAGE31 operates similarly to STAGE21, as it prepares for the second phase of stage 3 when it has completed processing hashes for the first 8 nonces and then inputs them to the hash_complete array. STAGE 32 either completes all the hashes for all 16 nonces if we use mode 0, otherwise it will complete the last 8. At the end of STAGE32, we will have calculated the final hashes for all the nonces and inputted them into the array hash_complete. We now transition to the WRITE state to write the h0 of all the nonces to memory. This state is implemented similarly to part 1 of this two part project. We have to wait one clock cycle to be able to write data to memory. We increase the offset by 1 each iteration to write all the hashes to memory. After we complete this process, and our iteration variable, i, is greater than the number of nonces + 1, we transition back to the IDLE state and we switch the "done" output signal to high, signifying that we have completed the bitcoin hashing operation (this is a mealy model).

This is the description of how the top module works. We will end our explanation by highlighting the differences and similarities of our modified sha256 module from the first part of the project. Firstly, there are no READ, WRITE and BLOCK states because they are superfluous. We don't need the BLOCK state because we are processing one block at each stage. Moreover, our hashes and words are Immediately available from the top module and we can immediately output them from the sha256 module, so we don't need the READ and WRITE states. Moreover, as we have mentioned previously, the FINISH state acts as a buffer to wait for the start signal to transition to 0, so that the sha256 instances don't begin processing words and hashes used for the stage it has currently completed. Instead, it waits for the new words and hashes computed for the next stage, and then the start signal becomes high again and it begins processing hashes for the new stage it is at. The sha256 module is similar to part 1 of the project in the way it pipelines the k + h term of t1 and how it does the word expansion and sha256 operation at the same time using a function that generates new words for the word expansion and a shift register that shifts the words in the w array. Thus, these optimizations are why we were able to implement an efficient bitcoin hasher with a minimal amount of space we were able to use and the best performance we were able to obtain.

➢ Simulation Waveform Snapshot:

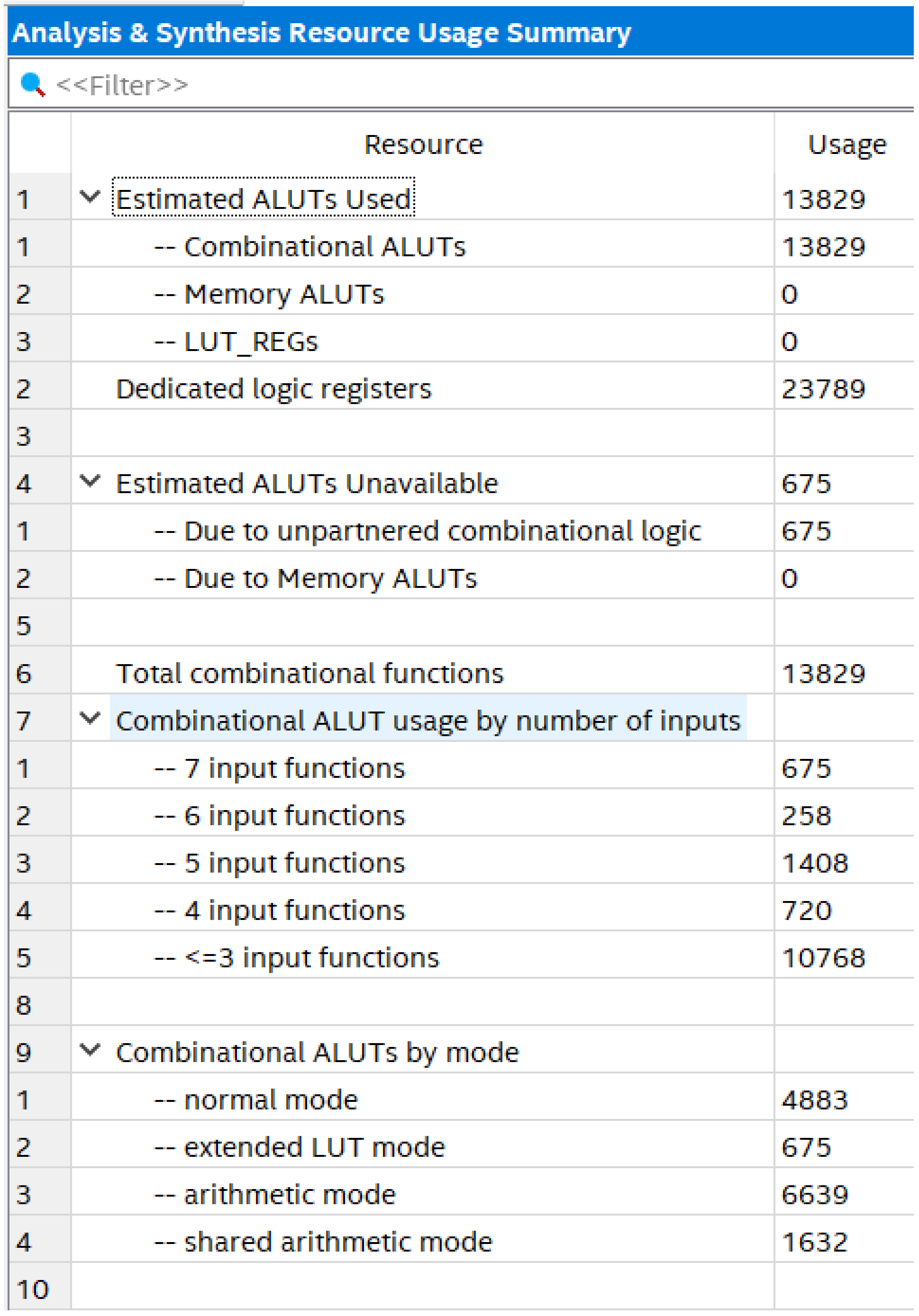


```
                    Msgs
clk                 St1
reset_n             St1
start               St0
message_addr        0000000000000000
output_addr         0000001111101000
done                0
mem_clk             1
mem_we              0
mem_addr            0000001111101000
mem_write_data      7106973a
mem_read_data       xxxxxxxxxxxxxx...
begin_stage1        0
end_stage1          0
begin_stage2        0
cur_addr            0000001111101000
offset              0000000000000000
cur_we              0
cur_write_data      01110001000001...
i                   00000001
state               WRITE
next_state          WRITE

Now                 4950 ps
Cursor 1            4610 ps
0 ps to 5198 ps     message_addr
```

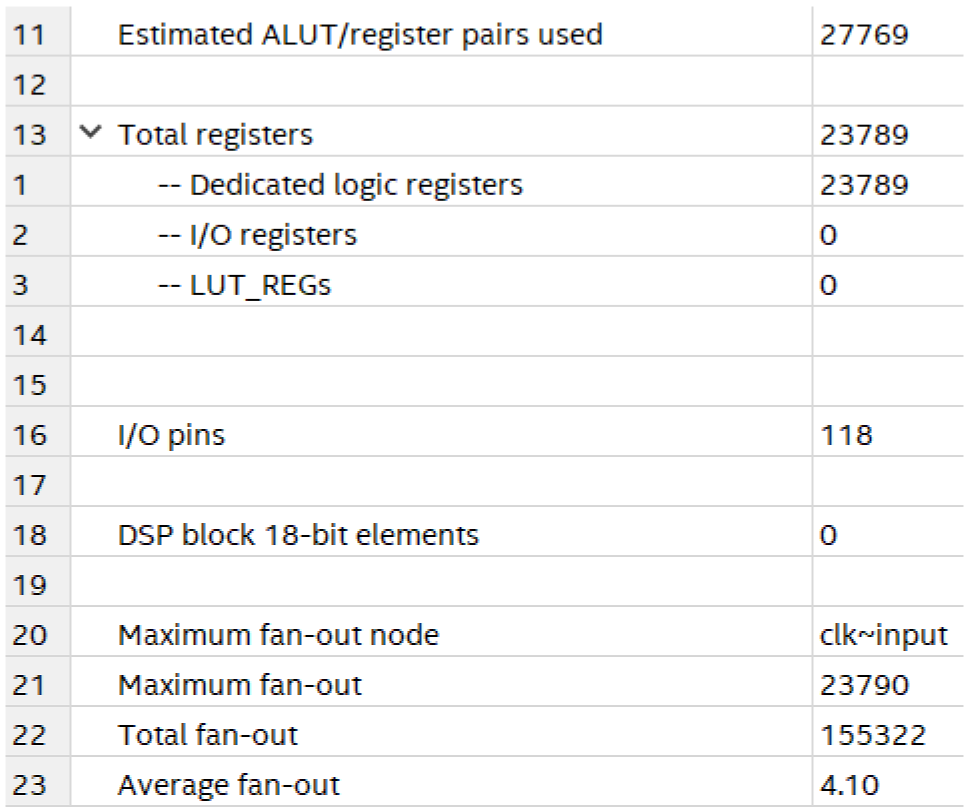
- Note: The waveform you are seeing here are all the items in the bitcoin_hash_inst (The instantiation of our design module). These aren't the items from the testbench.

➢ Resource Usage Snapshot


Analysis & Synthesis Resource Usage Summary

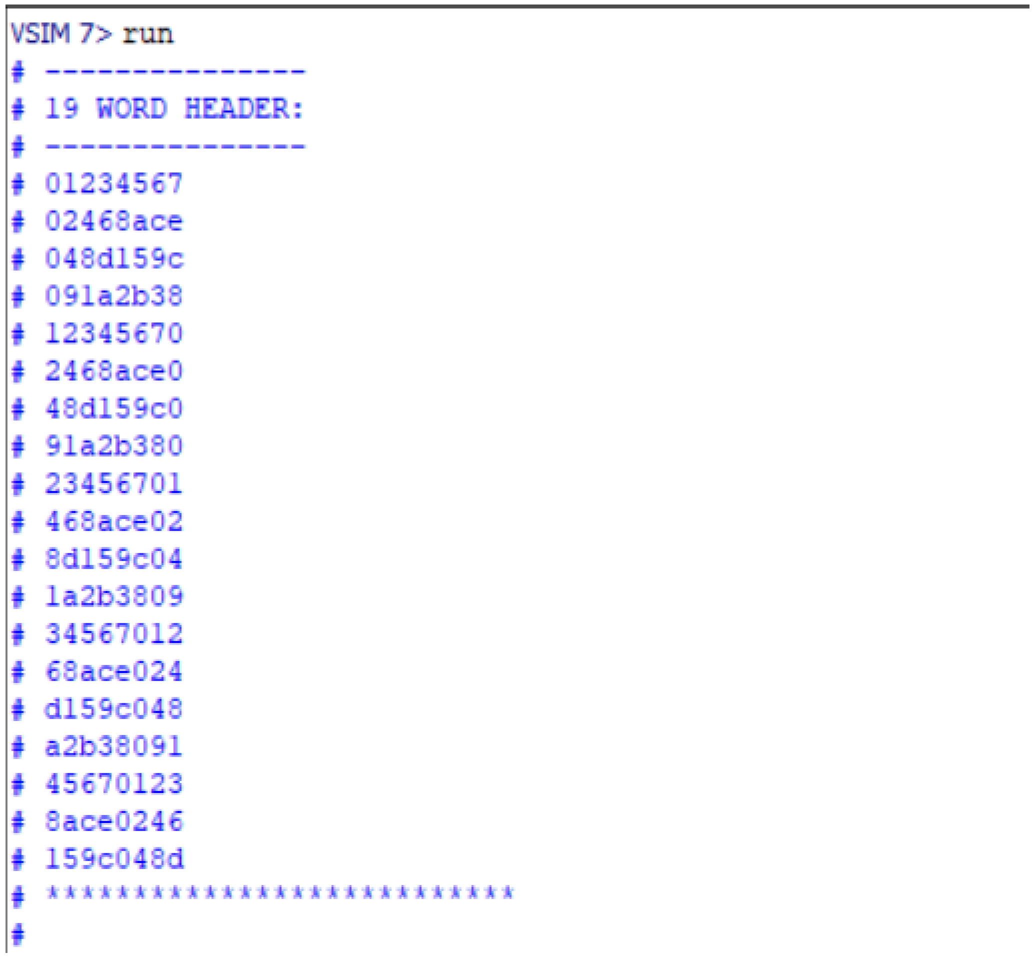
<<Filter>>

	Resource	Usage
1	Estimated ALUTs Used	13829
1	-- Combinational ALUTs	13829
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	23789
3		
4	Estimated ALUTs Unavailable	675
1	-- Due to unpartnered combinational logic	675
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	13829
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	675
2	-- 6 input functions	258
3	-- 5 input functions	1408
4	-- 4 input functions	720
5	-- <=3 input functions	10768
8		
9	Combinational ALUTs by mode	
1	-- normal mode	4883
2	-- extended LUT mode	675
3	-- arithmetic mode	6639
4	-- shared arithmetic mode	1632
10		

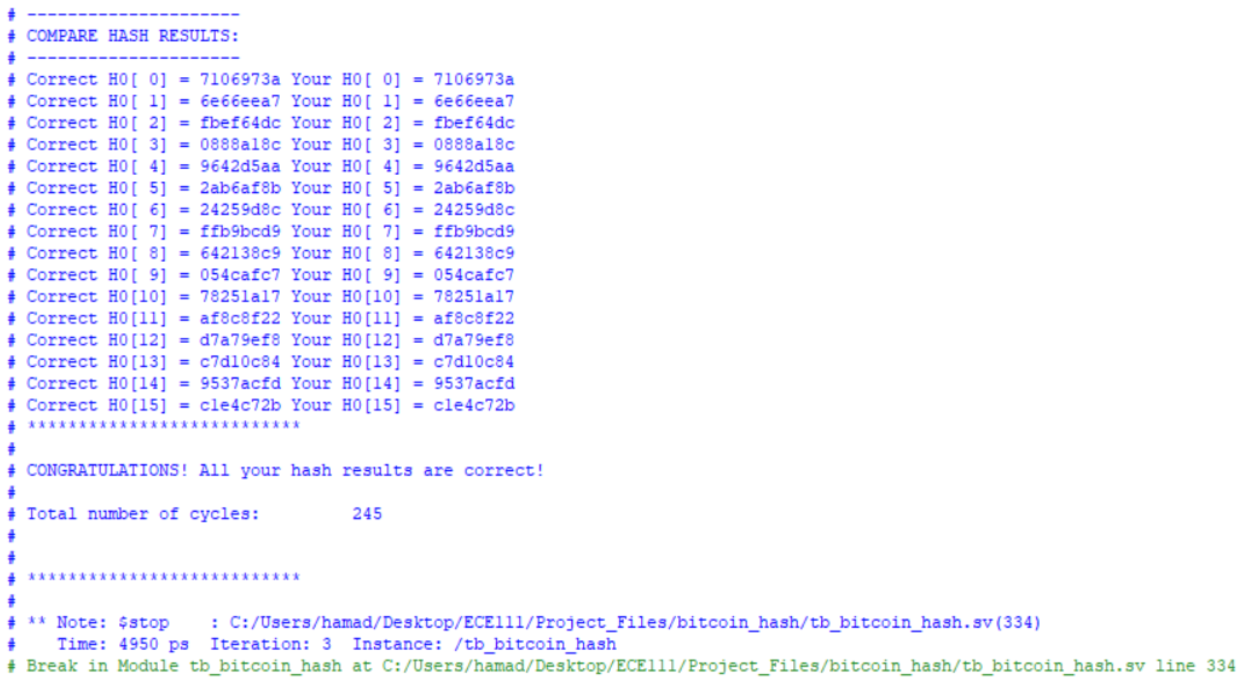


11	Estimated ALUT/register pairs used	27769
12		
13	Total registers	23789
1	-- Dedicated logic registers	23789
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	118
17		
18	DSP block 18-bit elements	0
19		
20	Maximum fan-out node	clk~input
21	Maximum fan-out	23790
22	Total fan-out	155322
23	Average fan-out	4.10

➢ Simulation Transcript Window Output:

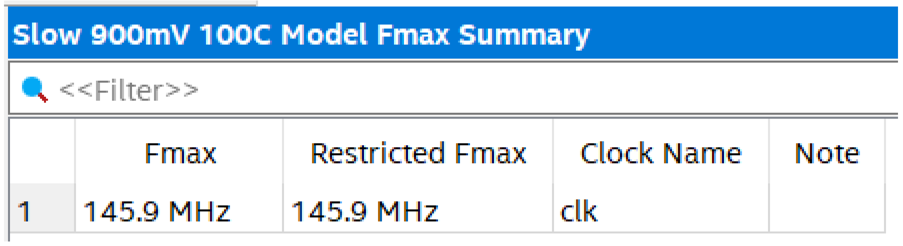


```
VSIM 7> run
# ---------------
# 19 WORD HEADER:
# ---------------
# 01234567
# 02468ace
# 048d159c
# 091a2b38
# 12345670
# 2468ace0
# 48d159c0
# 91a2b380
# 23456701
# 468ace02
# 8d159c04
# 1a2b3809
# 34567012
# 68ace024
# d159c048
# a2b38091
# 45670123
# 8ace0246
# 159c048d
# ***************************
#
```



```
# ---------------------
# 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:         245
#
#
# ***************************
#
# ** Note: $stop    : C:/Users/hamad/Desktop/ECE111/Project_Files/bitcoin_hash/tb_bitcoin_hash.sv(334)
#    Time: 4950 ps  Iteration: 3  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at C:/Users/hamad/Desktop/ECE111/Project_Files/bitcoin_hash/tb_bitcoin_hash.sv line 334
```

➢ Timing Analysis Snapshot:

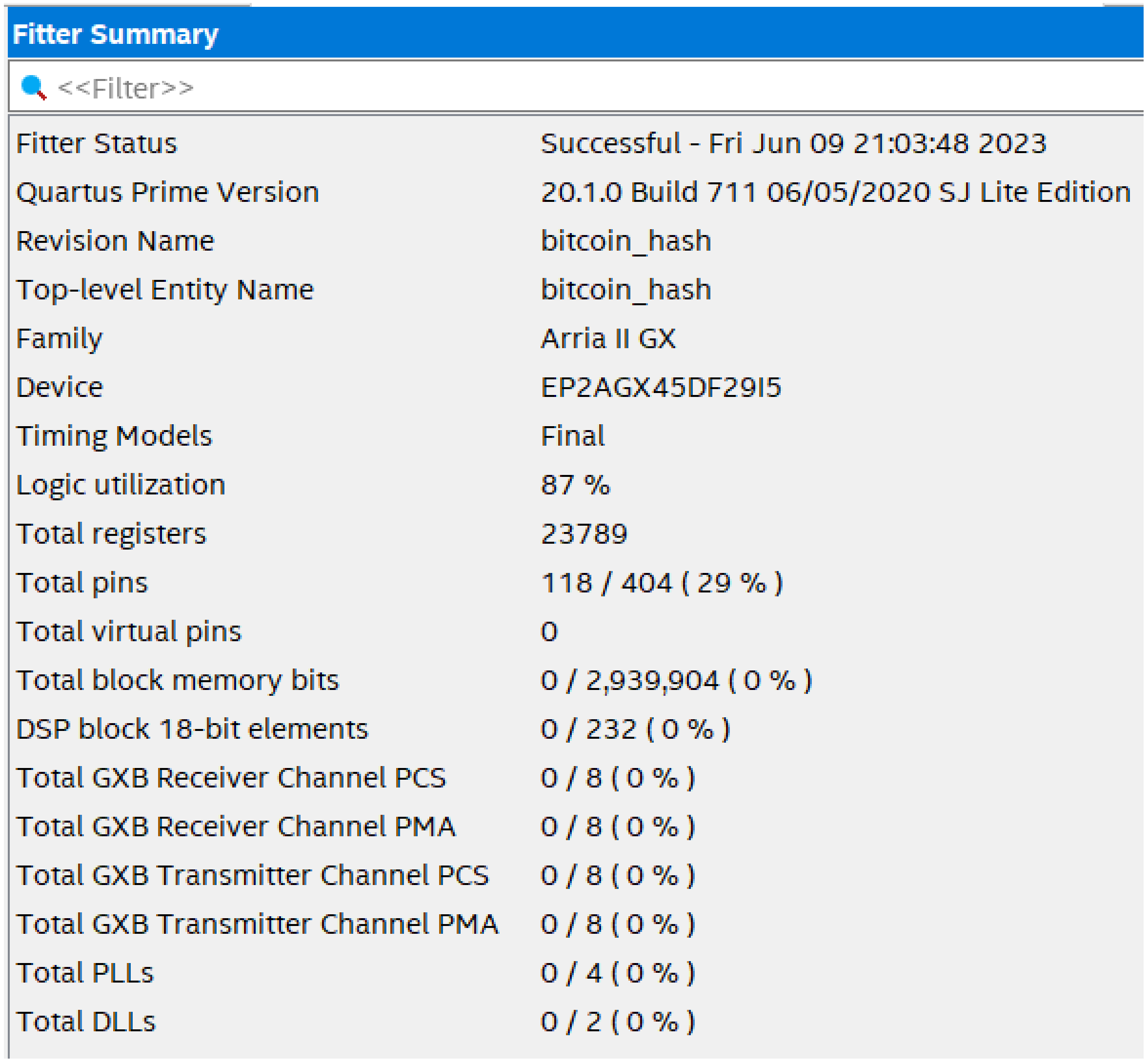


Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	145.9 MHz	145.9 MHz	clk	

➢ Fitter Report Snapshot:



Fitter Summary

<<Filter>>

Fitter Status	Successful - Fri Jun 09 21:03:48 2023
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	87 %
Total registers	23789
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 %)