Part 1:
Simplified_sha256.sv

Purpose
The purpose of SHA256 is to convert input data of arbitrary size, called a message, into a fixed output called a hash value of 256 bits in order to ensure the security of the message.
Our input message needs to be a multiple of 512 bits, so to determine the number of bits we need, we round up to the nearest multiple of 512. Since ours is 20 words, we have a 640 bit message, and thus we need 2 blocks. To add the padding, we append a single one bit then add zeros until we are 64 bits away from 512 (reach 448 bits). In our case this padding becomes part of the second block and is represented by 448-128 = 320 bits.The last 64 bits are used to encode the original length of our message into its 64 bit binary representation. After we split our padded input into N * 512 bit blocks, we split each block into 16 32 bit words

Algorithm Explanation
First, we begin at the IDLE state where we initialize our 8 initial hash values h0-h7(each 32 bits) and assign a-h our initial 8 hash values, along with other variables. We then go to the buffer stage. From there we go to the READ stage. Here is where our message block reads a block from memory and stores the 20 word message into its first 20 blocks of words (each of size 32 bits of course). Then begins our padding, starting with a single one bit and adding zeros to the remaining bits up until we reach message[31], where we store the encoded length of the original message. Next is our BLOCK stage where we fetch our message in 512 bit blocks, each of which initiates a hash value computation. We prepare a message schedule of 64 w's, each w being 32 bits long. The first 16 w's are just the same as the first 16 values of message. The remaining w values (w[16]-w[63]) hold the result after several rotate and shift, addition and modulo operations have been performed on the specific bits of the previous w values. We then implemented the function wtnew that optimizes the numbers of registers we use. Because our next value of w (w[15]) only relies on the previous 15 values of w, we can therefore significantly decrease the number of registers we use to compute these remaining w values. This optimization is utilized in the following stages. When passing our parameters into sha_op, we see that 'w' and 'h' do not depend on any of the other parameters (a-h). Therefore, we can precompute the operation w+k in t1 one cycle ahead by computing w 2 cycles ahead. Additionally, our next value of 'h' is only dependent on the previous value of 'g', hence we can add a more aggressive pipeline that computes 'h' one cycle ahead. Precomp1 computes the intermediate hash value new_h by adding the current word (w[0]), a constant (k[0]), and the current hash value (h). The constant k[0] is a predefined value specific to the SHA-256 algorithm and is used in the computation. The addition operation is performed using the + operator in Verilog or a similar language. The result of the addition is stored in the temporary variable new_h. After computing new_h, the module calls the word_exp() task to perform word expansion on the message schedule array w. Word expansion is a crucial step in the SHA-256 algorithm, where the 16-word message schedule array w[0] to w[15] is expanded to generate the remaining 48 words (w[16] to w[63]) used in the computation. The word_exp() task takes the current state i (round index) and the w array as inputs. Inside the task, a for loop is used to

iterate from 16 to 63, filling the remaining words of w based on specific bit operations and logical functions applied to the previous words. This expansion ensures that each word in the message schedule array has a unique value based on the input message and the previous words. Similar to the PRECOMP1 state, in the PRECOMP2 state, the module computes the intermediate hash value new_h by adding the current word (w[i]), a constant (k[i]), and the current hash value (h). The constant k[i] used in this state is different from k[0] and is specific to the current round index i. The addition operation is performed using the + operator, and the result is stored in the temporary variable new_h.Precomp2 does the hash computations only for a-h using the new_h value previously computed and then updates the new_h with the incremented w, k, and h constants. It later updates the message schedule and transitions us into the COMPUTE state to perform the final hashing on the values. Then in the COMPUTE state, we perform the remaining rounds of sha256_op until we reach 64 rounds of sha_256 operations. We do this by passing our parameters into the sha op which performs several operations and returns new values of a-h.Then we update our 8 hash values by adding these new a-h values to our previous hash values We repeat this process over again until we have no blocks remaining. In our case, since we have 2 blocks, the second block takes in the last 128 bits of the message along with the additional padding and message size. After repeating this process, we recognize that we are finished and go to the TEMP state where our final 8 hash values are stored into a temporary variable and then written to memory.

Resource Usage

Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated ALUTs Used	1796
1	-- Combinational ALUTs	1796
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	2426
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	1796
7	Combinational ALUT usage by number of inputs	
8		
9	Combinational ALUTs by mode	
1	-- normal mode	1207
2	-- extended LUT mode	0
3	-- arithmetic mode	461
4	-- shared arithmetic mode	128
10		
11	Estimated ALUT/register pairs used	2730
12		
13	Total registers	2426
1	-- Dedicated logic registers	2426
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	2427
22	Total fan-out	15522
23	Average fan-out	3.48

Fitter Summary

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Sat Jun 10 12:06:10 2023
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	simplified_sha256
Top-level Entity Name	simplified_sha256
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	8 %
Total registers	2426
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 %)

FMax Report Snapshot

Slow 900mV 100C Model Fmax Summary

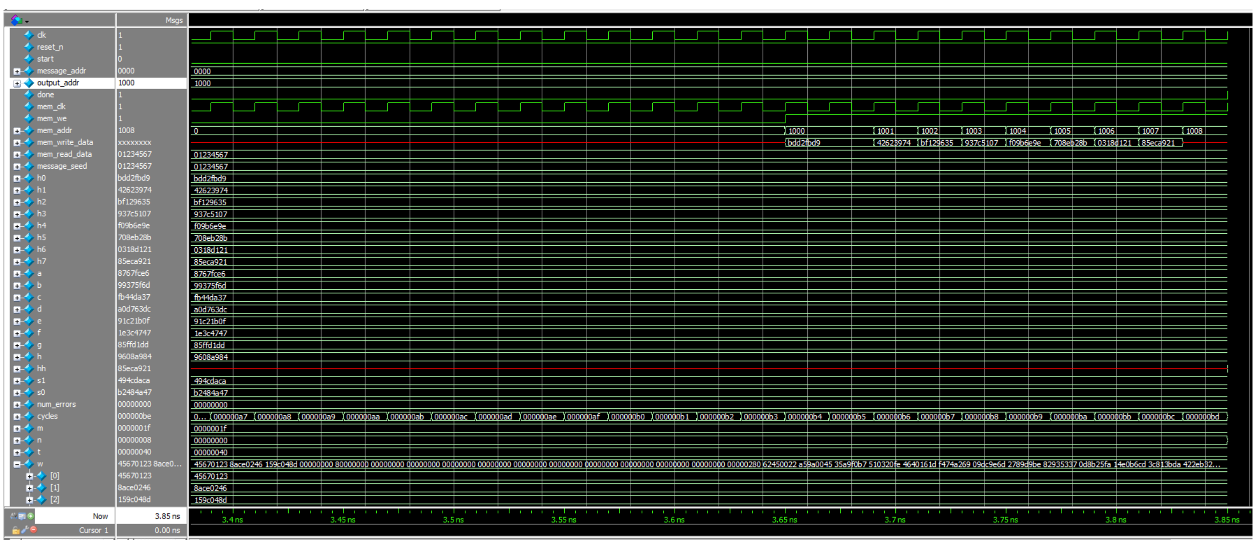
<<Filter>>

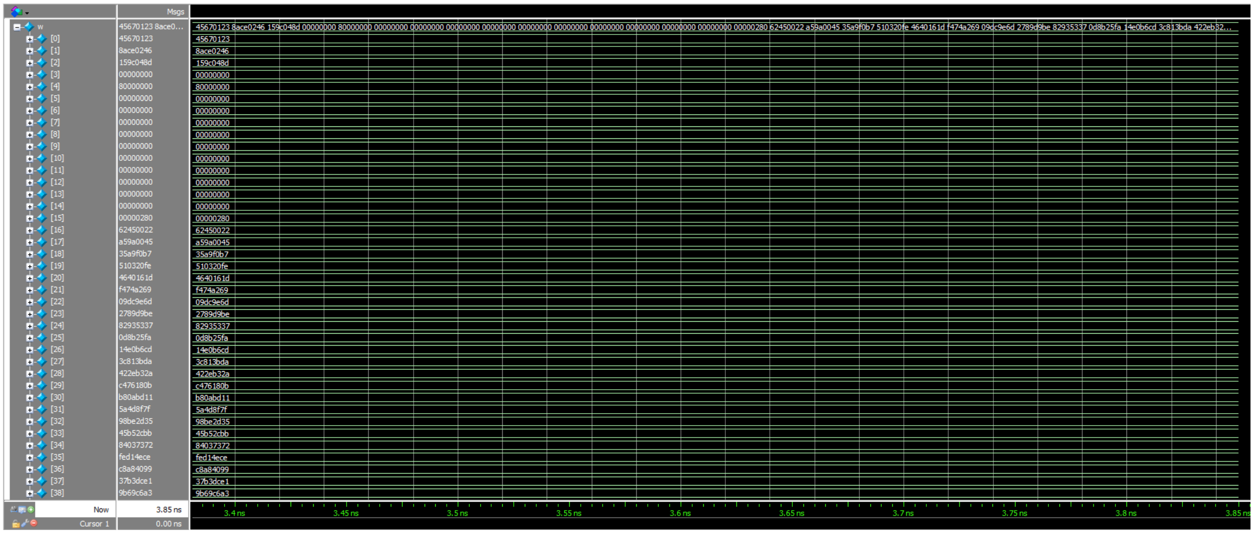
	Fmax	Restricted Fmax	Clock Name	Note
1	183.89 MHz	183.89 MHz	clk	

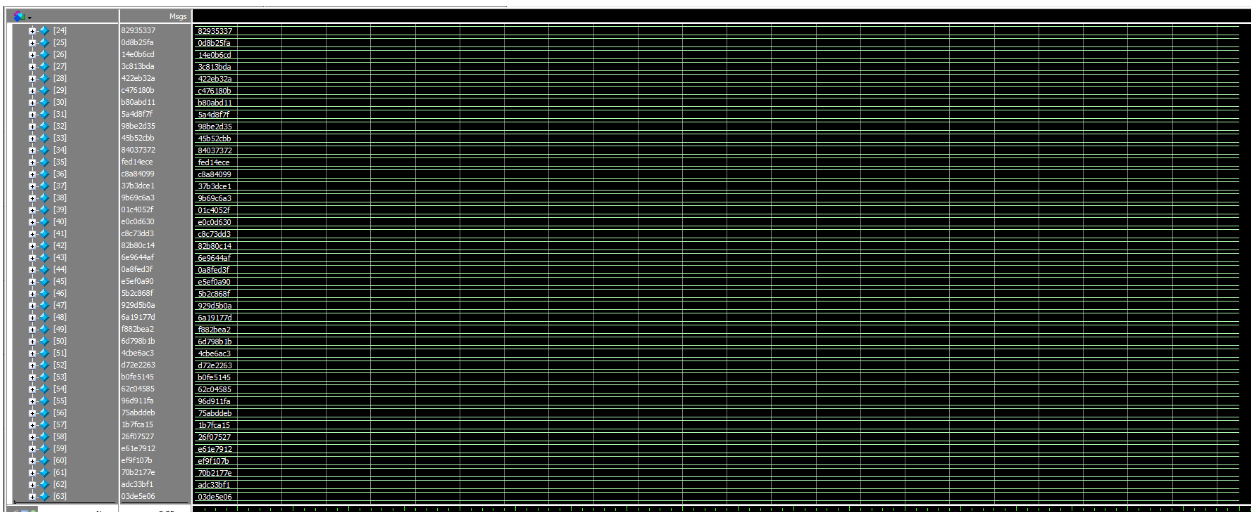
Transcript and Simulation

```
sim:/tb_simplified_sha256/w
VSIM 7> 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:         190
#
#
# ***************************
#
# ** Note: $stop    : C:/Users/Brian Mendez/Desktop/Verilog-ECE111/Project_Files/simplified_sha256/tb_simplified_sha256.sv(263)
#    Time: 3850 ps  Iteration: 2  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at C:/Users/Brian Mendez/Desktop/Verilog-ECE111/Project_Files/simplified_sha256/tb_simplified_sha256.sv line 263

VSIM 8>
```




Msgs
w 45670123 8ace0...
[0] 45670123
[1] 8ace0246
[2] 159c048d
[3] 00000000
[4] 80000000
[5] 00000000
[6] 00000000
[7] 00000000
[8] 00000000
[9] 00000000
[10] 00000000
[11] 00000000
[12] 00000000
[13] 00000000
[14] 00000000
[15] 00000280
[16] 62450022
[17] a59a0045
[18] 35a9f0b7
[19] 510320fe
[20] 4640161d
[21] f474a269
[22] 09dc9e6d
[23] 2789d9be
[24] 82935337
[25] 0d8b25fa
[26] 14e0b6cd
[27] 3c813bda
[28] 422eb32a
[29] c476180b
[30] b80abd11
[31] 5a4d8f7f
[32] 98be2d35
[33] 45b52cbb
[34] 84037372
[35] fed14ece
[36] c8a84099
[37] 37b3dce1
[38] 9b69c6a3
Now 3.85 ns
Cursor 1 0.00 ns
3.4 ns
3.45 ns
3.5 ns
3.55 ns
3.6 ns
3.65 ns
3.7 ns
3.75 ns
3.8 ns
3.85 ns


Msgs
[24] 82935337
[25] 0d8b25fa
[26] 14e0b6cd
[27] 3c813bda
[28] 422eb32a
[29] c476180b
[30] b80abd11
[31] 5a4d8f7f
[32] 98be2d35
[33] 45b52cbb
[34] 84037372
[35] fed14ece
[36] c8a84099
[37] 37b3dce1
[38] 9b69c6a3
[39] 01c4052f
[40] e0c0d630
[41] c8c73dd3
[42] 82b80c14
[43] 6e9644af
[44] 0a8fed3f
[45] e5ef0a90
[46] 5b2c868f
[47] 929d5b0a
[48] 6a19177d
[49] f882bea2
[50] 6d798b1b
[51] 4cbe6ac3
[52] d72e2263
[53] b0fe5145
[54] 62c04585
[55] 96d911fa
[56] 75abddeb
[57] 1b7fca15
[58] 26f07527
[59] e61e7912
[60] ef9f107b
[61] 70b2177e
[62] adc33bf1
[63] 03de5e06

Part 2
Bitcoin_Hash.sv

Purpose
The purpose of Bitcoin blockchain is to use SHA -256 cryptographic hashing algorithm to chain blocks. Each block stores digital information about financial transactions. The blocks are linked with a signature that allows users to determine if any data was altered by a malicious user. Once the blocks are chained together, its data is immutable - can never be changed again. The nonce value is a completely random string of numbers that is repeatedly changed/hashed in order to find a signature that meets the requirements for an eligible signature. The bitcoin hashing consists of three phases. Phase 1 processes the first block of the first SHA 256 hash function. Phase 2 processes the second block of the second SHA 256 hash function. And Phase 3 processes the second SHA 256 hash function. Phase-2 and 3 are performed 16 times and will produce 16 final hashes.

Algorithm Explanation
Our newly implemented sha256 is a variation of our implementation in part 1. Similarly to the algorithm, we pass the clk, reset, start, state, hash num, mem_read_data as our inputs instantiated in the bitcoin_hash module and output hashout to represent the hashes being thrown into the sha256 operator. The algorithm uses a state machine to organize the SHA-256 hashing process. The states include IDLE, READ1, READ2, PRECOMP, PHASE1, PHASE2, PHASE3, COMPUTE, and WRITE. The sha-256 module updates the variables depending on the state instantiated in bitcoin_hash. It operates similarly to our implementations in the first part.

In the bitcoin hash module, our always@ block begins at the IDLE stage where we initialize all our variables then move to the READ1 state. In this state, the machine sets an index i to 0, reads a byte from the message at the current memory address and increments the offset (to read the next byte in the next state). Our offset keeps track of the memory address from which data is being read and transitions to the READ2 state. This increments offset again and instantiates the sha module and updates the parameters to create another instance and moves to the next state. Similar to READ1, the READ2 state continues reading another byte from the message at the updated memory address, increments the offset, and transitions to the PRECOMP state which again updates a new instance for the sha256 module and begins to prepare the information for the first phase. Once we read more bytes in the PRECOMP state we move to PHASE1 where we continue reading the message as long as ‘i’ is less than 15. It increments i, and if i equals 64, it increments phase_cnt, which allows us to move to the next phase once everything has been read, and transitions to the COMPUTE state. The compute state creates the instance to compute hash values using the sha256 operator. Upon finishing computing all the values we transition back to the READ1 stage where the same process occurs. Precomp begins preparing the message for phase2 and similarly to phase1, begins reading the message only if ‘i’ is less than 2. If i equals 64, it increments phase_cnt and transitions to the COMPUTE state. It is worth noting that Phase 2 processes the 2nd block of the first in SHA 256 hash function. We repeat this process in Phase 3. When we reach the compute state, we recognize that we have completed all 3 phases and move to the write state where our final hash values for H0[0], H0[1] ...H0[15] in 16 words are written to memory starting at output_addr and incrementing write offset to update mem_addr accordingly.

Because our generate function is sensitive to the inputs of the sha256 function being called, everytime we transition from one state to the next, this triggers the generate_sha256_modules and creates a new instance of the sha with new inputs that are passed in, allowing sha and bitcoin modules to run in parallel. This sha function is similar to the one in part a of this project, but it performs different operations based on the state we are in as the state is passed in as an input to the function. Again, IDLE initializes the variables, READ1 sets i = 0, READ2 reads the input data and stores it in w[15]. PRECOMP calculates the new_h variable as the sum of the current message word (w[15]), a constant (k[i]), and the last value of the final hash (fh[7]). The current message word w[15] is updated with the value from mem_read_data. A loop is then performed to shift the values of the message words (w[n]) and update them with the next values in the sequence. In Phase1 the first message block, which consists of the first 16 words of our input message, along with our original hash values and Kt constants, get passed in to the sha operator. Our new_h is the computed one cycle ahead. In Phase 2, we process the second block of the message (which includes the padding and 1 word reserved for the nonce value), along with the output hash values generated from phase one and Kt constants. This time Sha256 is executed 16 times, one for each nonce value. Hence, we have 16 hashes, each of size 8 bits. In phase 3, since we have reached the end of our message, the final output of our 256-bit hash message from phase 2 becomes the first 8 words of our new input message and the remaining 8 are padding (equaling a 512 bit block with no nonce value passed in). Sha256 is again instantiated 16 times, once for each nonce value.

Resource Usage

Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated ALUTs Used	19646
1	-- Combinational ALUTs	19646
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	17514
3		
4	Estimated ALUTs Unavailable	5
1	-- Due to unpartnered combinational logic	5
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	19646
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	5
2	-- 6 input functions	3425
3	-- 5 input functions	1436
4	-- 4 input functions	148
5	-- <=3 input functions	14632
8		
9	Combinational ALUTs by mode	
10		
11	Estimated ALUT/register pairs used	25008
12		
13	Total registers	17514
1	-- Dedicated logic registers	17514
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	17515
22	Total fan-out	136943
23	Average fan-out	3.66

Fitter Usage

Fitter Summary

<<Filter>>

Fitter Status	Successful - Sat Jun 10 12:58:42 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	EP2AGX45CU17I3
Timing Models	Final
Logic utilization	83 %
Total registers	17514
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 %)

Fmax Report Summary

Slow 900mV 100C Model Fmax Summary

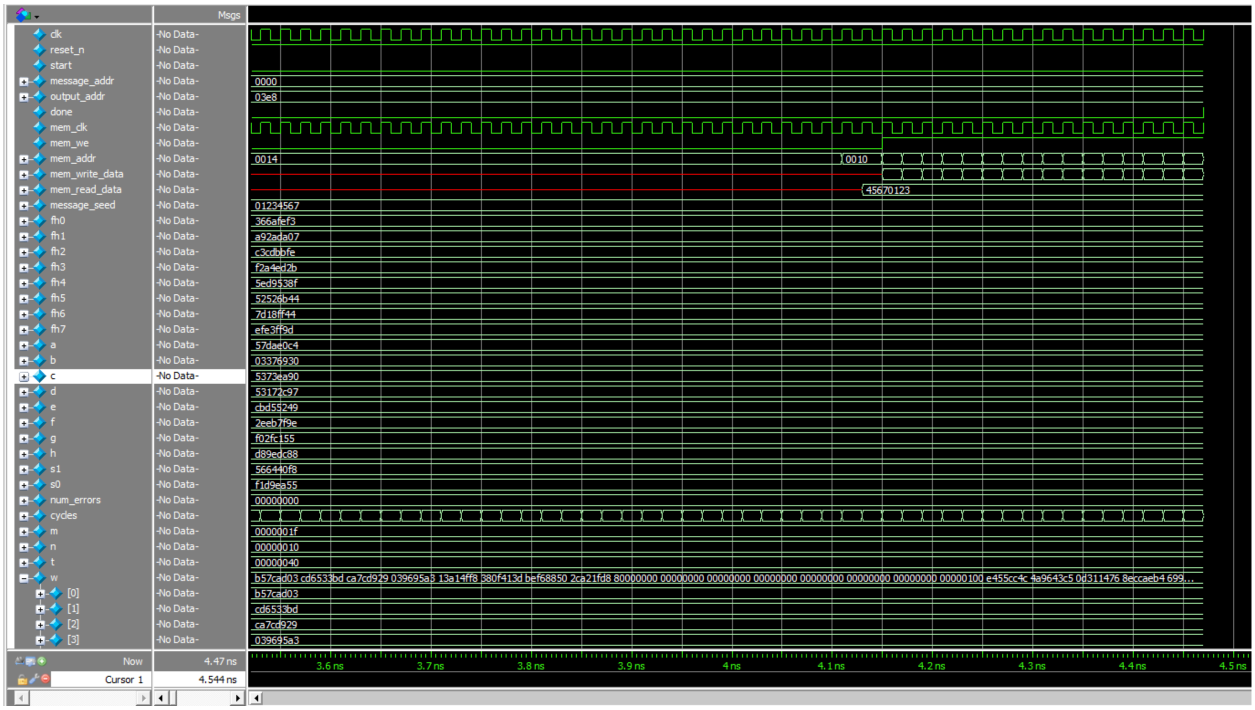
<<Filter>>

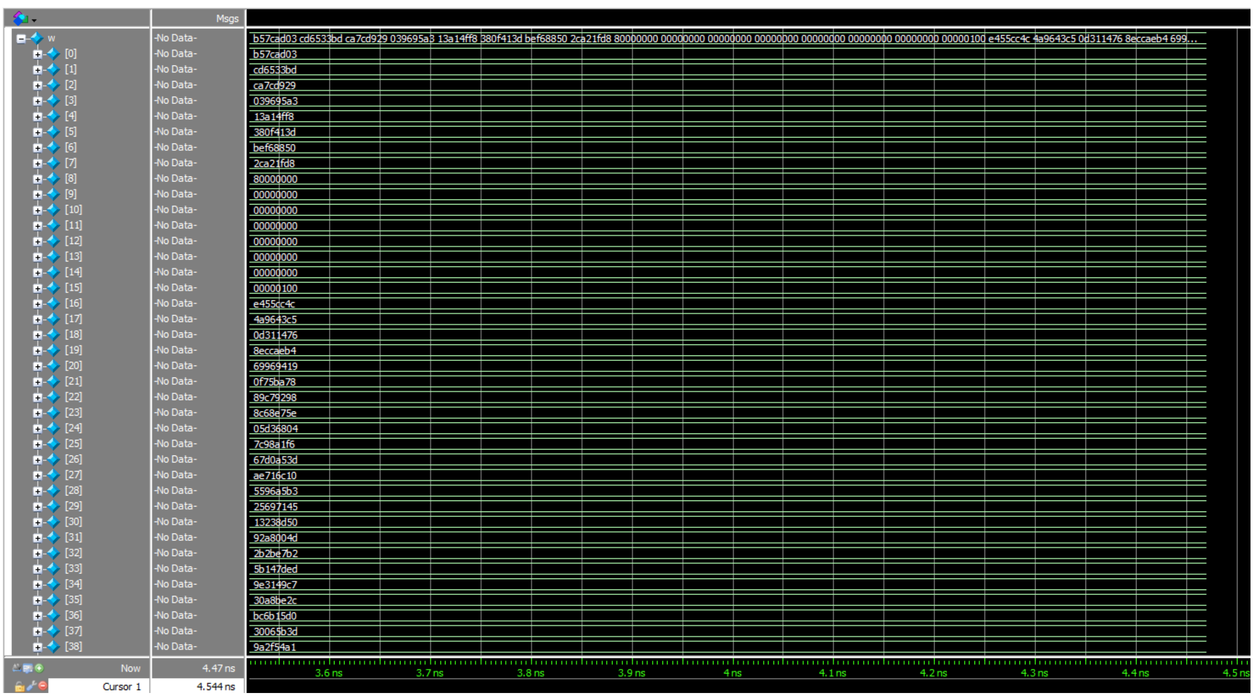
	Fmax	Restricted Fmax	Clock Name	Note
1	177.62 MHz	177.62 MHz	clk	

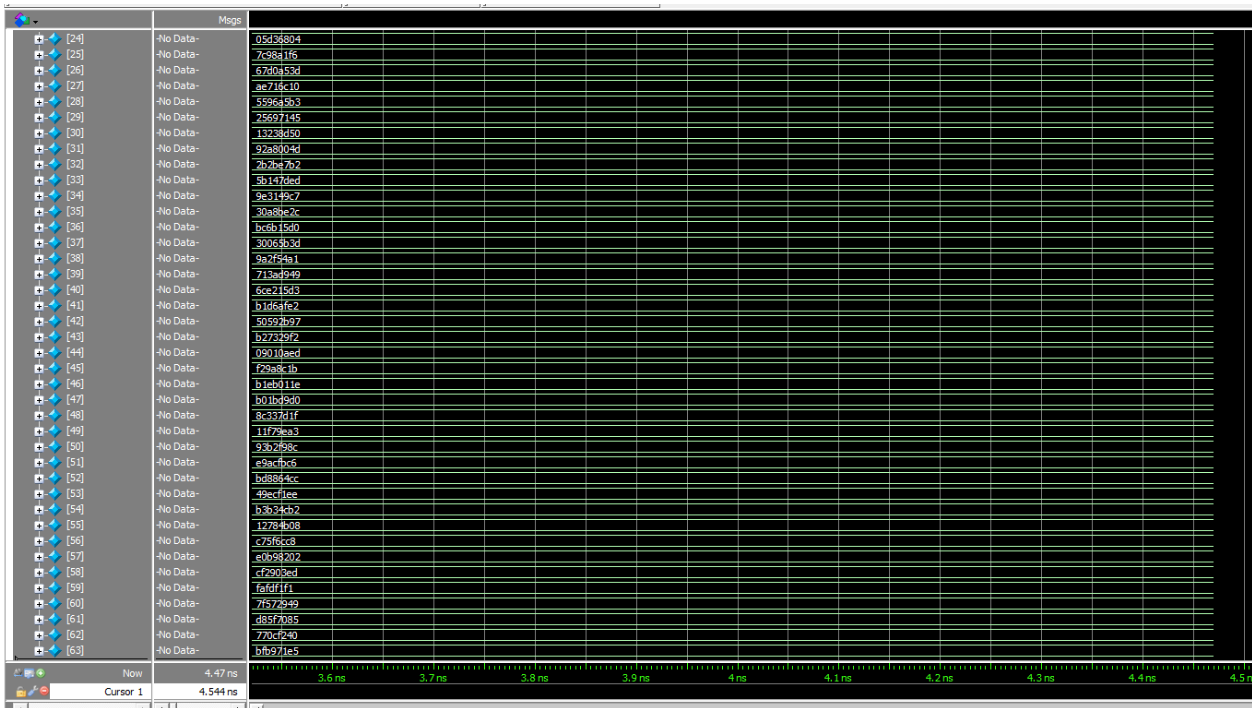
Transcript and Simulation

```
VSIM 8> 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
# ***************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:         221
#
#
# ***************************
#
# ** Note: $stop    : C:/Users/Brian Mendez/Desktop/Verilog-ECE111/Project_Files/bitcoin_hash/tb_bitcoin_hash.sv(334)
#    Time: 4470 ps  Iteration: 2  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at C:/Users/Brian Mendez/Desktop/Verilog-ECE111/Project_Files/bitcoin_hash/tb_bitcoin_hash.sv line 334

VSIM 9>
```


Msgs
clk -No Data-
reset_n -No Data-
start -No Data-
message_addr -No Data- 0000
output_addr -No Data- 03e8
done -No Data-
mem_clk -No Data-
mem_we -No Data-
mem_addr -No Data- 0014 0010
mem_write_data -No Data-
mem_read_data -No Data- 45670123
message_seed -No Data- 01234567
fh0 -No Data- 366afef3
fh1 -No Data- a92ada07
fh2 -No Data- c3cdbbfe
fh3 -No Data- f2a4ed2b
fh4 -No Data- 5ed9538f
fh5 -No Data- 52526b44
fh6 -No Data- 7d18ff44
fh7 -No Data- efe3ff9d
a -No Data- 57dae0c4
b -No Data- 03376930
c -No Data- 5373ea90
d -No Data- 53172c97
e -No Data- cbd55249
f -No Data- 2eeb7f9e
g -No Data- f02fc155
h -No Data- d89edc88
s1 -No Data- 566440f8
s0 -No Data- f1d9ea55
num_errors -No Data- 00000000
cycles -No Data-
m -No Data- 0000001f
n -No Data- 00000010
t -No Data- 00000040
w -No Data- b57cad03 cd6533bd ca7cd929 039695a3 13a14ff8 380f413d bef68850 2ca21fd8 80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000100 e455cc4c 4a9643c5 0d311476 8eccaeb4 699...
[0] -No Data- b57cad03
[1] -No Data- cd6533bd
[2] -No Data- ca7cd929
[3] -No Data- 039695a3
Now 4.47 ns
Cursor 1 4.544 ns
3.6 ns
3.7 ns
3.8 ns
3.9 ns
4 ns
4.1 ns
4.2 ns
4.3 ns
4.4 ns
4.5 ns


Msgs
w -No Data- b57cad03 cd6533bd ca7cd929 039695a3 13a14ff8 380f413d bef68850 2ca21fd8 80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000100 e455cc4c 4a9643c5 0d311476 8eccaeb4 699...
[0] -No Data- b57cad03
[1] -No Data- cd6533bd
[2] -No Data- ca7cd929
[3] -No Data- 039695a3
[4] -No Data- 13a14ff8
[5] -No Data- 380f413d
[6] -No Data- bef68850
[7] -No Data- 2ca21fd8
[8] -No Data- 80000000
[9] -No Data- 00000000
[10] -No Data- 00000000
[11] -No Data- 00000000
[12] -No Data- 00000000
[13] -No Data- 00000000
[14] -No Data- 00000000
[15] -No Data- 00000100
[16] -No Data- e455cc4c
[17] -No Data- 4a9643c5
[18] -No Data- 0d311476
[19] -No Data- 8eccaeb4
[20] -No Data- 69969419
[21] -No Data- 0f75ba78
[22] -No Data- 89c79298
[23] -No Data- 8c68e75e
[24] -No Data- 05d36804
[25] -No Data- 7c98a1f6
[26] -No Data- 67d0a53d
[27] -No Data- ae716c10
[28] -No Data- 5596a5b3
[29] -No Data- 25697145
[30] -No Data- 13238d50
[31] -No Data- 92a8004d
[32] -No Data- 2b2be7b2
[33] -No Data- 5b147ded
[34] -No Data- 9e3149c7
[35] -No Data- 30a8be2c
[36] -No Data- bc6b15d0
[37] -No Data- 30065b3d
[38] -No Data- 9a2f54a1
Now 4.47 ns
Cursor 1 4.544 ns
3.6 ns
3.7 ns
3.8 ns
3.9 ns
4 ns
4.1 ns
4.2 ns
4.3 ns
4.4 ns
4.5 ns


Msgs
[24] -No Data- 05d36804
[25] -No Data- 7c98a1f6
[26] -No Data- 67d0a53d
[27] -No Data- ae716c10
[28] -No Data- 5596a5b3
[29] -No Data- 25697145
[30] -No Data- 13238d50
[31] -No Data- 92a8004d
[32] -No Data- 2b2be7b2
[33] -No Data- 5b147ded
[34] -No Data- 9e3149c7
[35] -No Data- 30a8be2c
[36] -No Data- bc6b15d0
[37] -No Data- 30065b3d
[38] -No Data- 9a2f54a1
[39] -No Data- 713ad949
[40] -No Data- 6ce215d3
[41] -No Data- b1d6afe2
[42] -No Data- 50592b97
[43] -No Data- b27329f2
[44] -No Data- 09010aed
[45] -No Data- f29a8c1b
[46] -No Data- b1eb011e
[47] -No Data- b01bd9d0
[48] -No Data- 8c337d1f
[49] -No Data- 11f79ea3
[50] -No Data- 93b2f98c
[51] -No Data- e9acfbc6
[52] -No Data- bd8864cc
[53] -No Data- 49ecf1ee
[54] -No Data- b3b34cb2
[55] -No Data- 12784b08
[56] -No Data- c75f6cc8
[57] -No Data- e0b98202
[58] -No Data- cf2903ed
[59] -No Data- fafdf1f1
[60] -No Data- 7f572949
[61] -No Data- d85f7085
[62] -No Data- 770cf240
[63] -No Data- bfb971e5
Now 4.47 ns
Cursor 1 4.544 ns
3.6 ns
3.7 ns
3.8 ns
3.9 ns
4 ns
4.1 ns
4.2 ns
4.3 ns
4.4 ns