Travis Davis
A15485288

ECE 111 Final Report

PART 1: SHA-256

For part 1 of this project, I developed a SHA-256 algorithm to be used on an FPGA. SHA-256 is a hashing algorithm that takes an input message and outputs a unique hash value that will always be the same for a particular input. Regardless of the size of the input, the hash output will always be eight 32 bit words. The algorithm is also irreversible and the output cannot be used to recover the original message.

I begin with initializing an array of 64 “k” constants. I also define a variety of other helper functions, constants, and logic arrays that will be used as part of the algorithm. I define six states {IDLE, READ, BLOCK1, COMPUTE1, BLOCK2, COMPUTE2, WRITE} to be used as part of the state machine in the body of the code.

In the IDLE state, we first check if start is equal to 1. If so, I set cur_we and thus mem_we to 0 so that the algorithm will read data. I also set the curr_addr and thus the mem_addr to the message_addr where the input message is stored. Additionally, I reset any indices, the done value, and the cur_write_data to zero.

Moving onto the READ state, I pass on the contents of mem_read_data to a 640 bit (20 32 bit indices) message array while incrementing the message address by 1 after every clock cycle.

For BLOCK1, I pass specific initial hash values to each output hash logic {ha[0], ha[1], ha[2], ha[3], ha[4], ha[5], ha[6], ha[7]} and some logic {a,b,c,d,e,f,g,h} that will eventually be added to the hash values respectively.

In COMPUTE1, we have 65 iterations. If the index i is less than 16, then it merely passes the output of sha256_op to {a, b, c, d, e, f, g, h} then iterating to the next cycle of the state. If i is greater or equal to 16, word expansion is performed on our w array. Since we are using blocking assignments, we must wait for the next clock cycle for w[i] to be updated. Therefore, if i is not equal to 16 in this conditional statement, we pass the output of the sha256_op to {a,b,c,d,e,f,g,h} with the arguments w[step] and step representing the values of the previous index. Once the 64 iterations of sha256_op have been completed, the sha_256 output is finally added to the hash values.

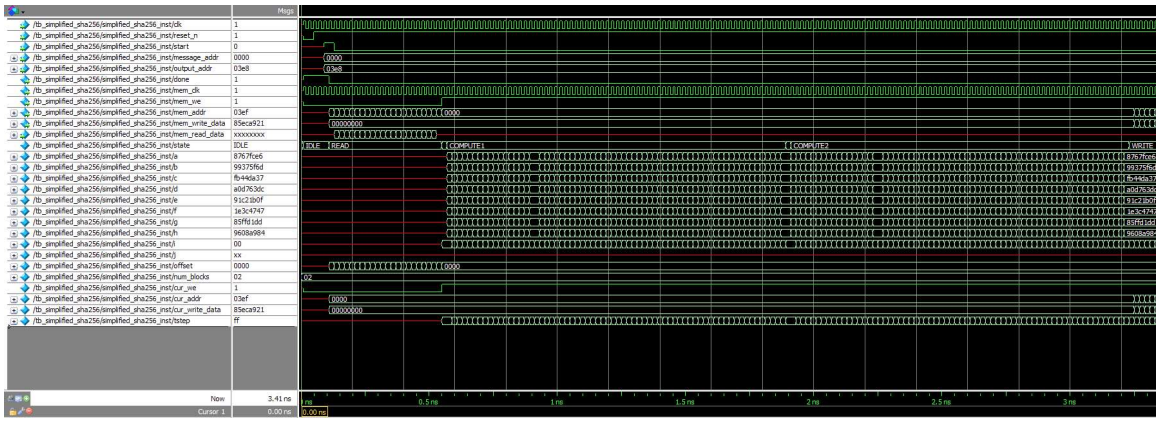
We then move to BLOCK2, where I pass the remaining 4 message indices to the

first four indices of the w array. The next index is assigned 32'h80000000. The remaining indices before the final index are assigned 0. The last index is assigned the size of the message in bits, which is 640 in this case. The {a,b,c,d,e,f,g,h} are then assigned the corresponding hash values derived in COMPUTE1.

COMPUTE2 functions exactly the same as COMPUTE1 only with the updated w and {a, b, c, d, e, f, g, h} values.

Finally, in the WRITE state, we have 8 iterations for the 8 hash values. We set the curr_addr to the output_addr plus the index i, curr_write_data to ha[i], and cur_we to 1 to tell the program to write before updating the index and moving to the next iteration of WRITE. Once the 8 iterations are finished, we pass the IDLE state to the state value and reset the index.

Simulation:



For this simulation of the SHA 256 algorithm, we see each variable in hexadecimal form and how each variable changes across the different states of my algorithm.

Transcript:

```
# Top level modules:
#       simplified_sha256
# End time: 20:36:38 on May 29,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
vlog -reportprogress 300 -work work C:/Users/t3davis/Downloads/Final_Project-20220529T202801Z-001/Final_Project/simplified_sha256/tb_simplified_sha256.
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 20:36:38 on May 29,2022
# vlog -reportprogress 300 -work work C:/Users/t3davis/Downloads/Final_Project-20220529T202801Z-001/Final_Project/simplified_sha256/tb_simplified_sha25
# -- Compiling module tb_simplified_sha256
#
# Top level modules:
#       tb_simplified_sha256
# End time: 20:36:38 on May 29,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
ModelSim> vsim work.tb_simplified_sha256
# vsim work.tb_simplified_sha256
# Start time: 20:37:02 on May 29,2022
# Loading sv_std.std
# Loading work.tb_simplified_sha256
# Loading work.simplified_sha256
add wave -position insertpoint sim:/tb_simplified_sha256/simplified_sha256_inst/*
VSIM 6> 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:         168
#
#
# ****************************
#
# ** Note: $stop    : C:/Users/t3davis/Downloads/Final_Project-20220529T202801Z-001/Final_Project/simplified_sha256/tb_simplified_sha256.sv(262)
#    Time: 3410 ps  Iteration: 2  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at C:/Users/t3davis/Downloads/Final_Project-20220529T202801Z-001/Final_Project/simplified_sha256/tb_simplified_sha256.sv li

VSIM 7>
```

PART 2: BITCOIN HASHING

In part 2, we build upon the concepts of SHA 256 detailed in part 1. Bitcoin hashing is a hashing algorithm that also takes an input of 20 words but returns an output

of 16 sets of eight 32-bit words. The bitcoin hashing algorithm essentially functions as a chained SHA 256 algorithm.

Similar to part 1, we initialize k constants, and define various variables and helper functions. Our states in this part are {IDLE, READ, BLOCK1, COMPUTE1, BLOCK2, COMPUTE2, WRITE}. The main difference revolves around how word expansion is performed. Since there is a finite amount of logic in the FPGA available to us, we must be a little more efficient with our resources. Therefore, our w array has 16 indices here instead of 64. The output for word expansion is merely one 32 bit value. During the word expansion portion of the “compute” states, each index of the 16 index w array until the 16th index is assigned the value of the next index. The 16th index is then assigned the output of the word expansion function.

The IDLE state functions similarly as in part 1; however, in this case, the pass value is assigned 1 and the j value is assigned 0. This accounts for the second round of hashing we must perform out of necessity of efficiently using the FPGA logic. The READ state functions exactly as in part 1.

The BLOCK1 functions as it does in part1. In this case, each initial hash is assigned to {fh1, fh2, fh3, fh4, fh5, fh6, fh7}. The COMPUTE1 state also function similarly as in part 1, with {fh1, fh2, fh3, fh4, fh5, fh6, fh7} replacing {ha[1], ha[2], ha[3], ha[4], ha[5], ha[6], ha[7]}. We also factor in our new aforementioned method of word expansion. Finally, the w[i] argument in part 1 is replaced with w[0].

In the BLOCK2 state, we have 8 iterations. The first 3 wn[n][] are assigned the remaining message values. Then, wn[n][3] is assigned the index value n+j representing the nonce. As part of the padding process, wn[n][4] is assigned 32’h80000000, the remaining wn[n][] indices before the last are assigned 0, and the final index is assigned the size of the message which is 640. The index is updated and moves onto the next iteration until the process is complete.

The COMPUTE2 state functions similarly to COMPUTE1; however the procedures in this state are also performed across 8 iterations. We also replace w with wn[n][], {a, b, c, d, e, f, g, h} with {an[n], bn[n], cn[n], dn[n], en[n], fn[n], gn[n], hn[n]}, and {fh1, fh2, fh3, fh4, fh5, fh6, fh7} with {ha[n][1], ha[n][2], ha[n][3], ha[n][4], ha[n][5], ha[n][6], ha[n][7]}. The index n is updated and moves onto the next iteration until the process is complete.

In BLOCK3, we again have 8 iterations. The first seven wn[n][] indices are assigned the corresponding ha[n][] . Then, wn[n][8] is assigned 32'h80000000, the remaining wn[n][] indices before the last are assigned 0, and the final index is assigned the size of the message which is 256 in this case. Additionally, {ha[n][1], ha[n][2], ha[n][3], ha[n][4], ha[n][5], ha[n][6], ha[n][7]} and {an[n], bn[n], cn[n], dn[n], en[n], fn[n], gn[n], hn[n]} are assigned the original corresponding hash values. The index is updated and moves onto the next iteration until the process is complete. We then move to COMPUTE3, which functions exactly as COMPUTE2.

Finally, we come to WRITE, where we set the curr_addr to the output_addr plus the index i and the value j, curr_write_data to the collection of hash outputs for the 0 nonce ha[0][i], and cur_we to 1. Once the 8 iterations are finished, we check to see if the pass value is equal to 1. If so, we assign our pass variable the original pass value minus 1. We also assign j the value 8 and reset all indices before moving back to the BLOCK2 state, where we run through each sequential process again only with updated pass and j values. Once it writes the ha[0][i] values again, the algorithm recognizes the pass variable is equal to 0, and we pass the IDLE state to the state value and reset the index.

Flow Summary:

Flow Summary

<<Filter>>

Flow Status	Successful - Sun May 29 21:11:47 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	41 %
Total registers	9941
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 %)

Resource Usage:

	Resource	Usage
1	Estimated ALUTs Used	5365
1	-- Combinational ALUTs	5365
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	9941
3		
4	Estimated ALUTs Unavailable	548
1	-- Due to unpartnered combinational logic	548
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	5365
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	548
2	-- 6 input functions	1945
3	-- 5 input functions	213
4	-- 4 input functions	802
5	-- <=3 input functions	1857
8		
9	Combinational ALUTs by mode	
1	-- normal mode	3755
2	-- extended LUT mode	548
3	-- arithmetic mode	822
4	-- shared arithmetic mode	240
10		
11	Estimated ALUT/register pairs used	13254
12		
13	Total registers	9941
1	-- Dedicated logic registers	9941
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	9942
22	Total fan-out	61654
23	Average fan-out	3.97

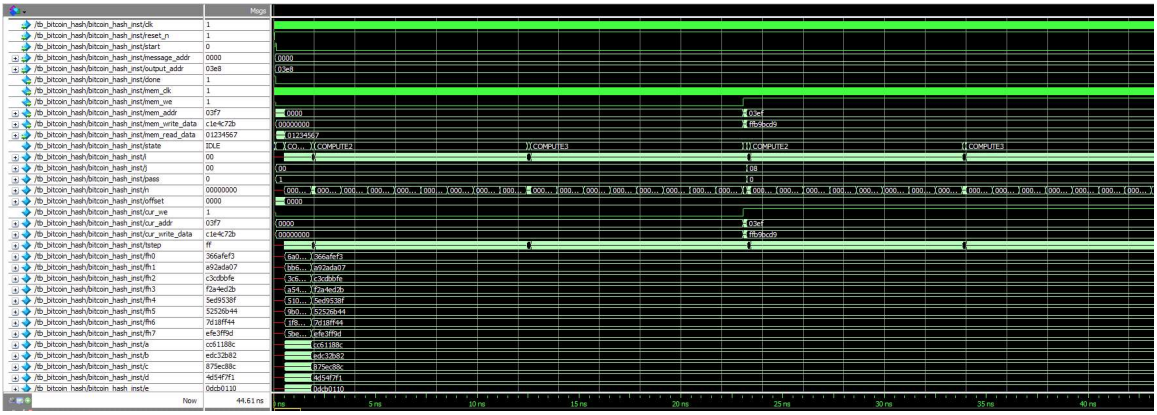
Timing Report:

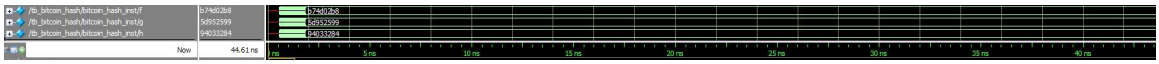
Slow 900mV -40C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	108.93 MHz	108.93 MHz	clk	

Simulation:





For this simulation of my bitcoin hashing algorithm, we see each variable in hexadecimal form and how each variable changes across the different states of my algorithm.

Transcript:

Transcript

```
# Top level modules:
#       bitcoin_hash
# End time: 21:18:55 on May 29,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
vlog -reportprogress 300 -work work C:/Users/t3davis/Downloads/Final_Project-20220529T202801Z-001/Final_Project/bitcoin_hash/tb_bitcoin_hash.sv
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 21:18:55 on May 29,2022
# vlog -reportprogress 300 -work work C:/Users/t3davis/Downloads/Final_Project-20220529T202801Z-001/Final_Project/bitcoin_hash/tb_bitcoin_hash.
# -- Compiling module tb_bitcoin_hash
#
# Top level modules:
#       tb_bitcoin_hash
# End time: 21:18:55 on May 29,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
ModelSim> vsim work.tb_bitcoin_hash
# vsim work.tb_bitcoin_hash
# Start time: 21:18:59 on May 29,2022
# Loading sv_std.std
# Loading work.tb_bitcoin_hash
# Loading work.bitcoin_hash
add wave -position insertpoint sim:/tb_bitcoin_hash/bitcoin_hash_inst/*
VSIM 6> 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:       2228
#
#
# ****************************
#
# ** Note: $stop    : C:/Users/t3davis/Downloads/Final_Project-20220529T202801Z-001/Final_Project/bitcoin_hash/tb_bitcoin_hash.sv(334)
#    Time: 44610 ps  Iteration: 2  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at C:/Users/t3davis/Downloads/Final_Project-20220529T202801Z-001/Final_Project/bitcoin_hash/tb_bitcoin_hash.sv lin

VSIM 7>
```