Yingjian Pei
A13595990
Alexander Zhu
A13578906

Final report

SHA-256 is and bitcoin hashing explanation

In this final project, SHA-256 is a type of simplified secure hash algorithm. The goal is to compute a unique hash value for any input “message”, where a “message” can be any 20 words. Then it will output eight 32-bits words as hashed output.

In part II of our final project, bitcoin hashing is a type of hashing algorithm that takes an input of 20 words (19 words and 1 nonce value) and return an output of eight 32-bits words using chained SHA-256 algorithm.

SHA-256 and Bitcoin hashing algorithm description

SHA-256 algorithm description:

First we initialize the hash constants int k[0..63] and write helper functions *wtnew* to calculate the next wt and *rightrotates* to rotate right the input values.

In the IDLE state, we initialize eight 32-bit hash values to some constants as well as setting the done and mem_we signals to 0. In addition, we set the mem_addr signal to the message_addr signal and reset the index. If start == 1, we transit to the READ_FIRST_BLOCK state, otherwise we stay at the idle state.

In the READ_FIRST_BLOCK state, we set the w[index] to the mem_read_read and increment the index and mem_addr by 1. If index == 16, we transit to the HASH_FIRST_BLOCK state and set the a...h constants to some hash constants. We then offset the mem_addr signal by -1. Else, we stay at the READ_FIRST_BLOCK state.

In the HASH_FIRST_BLOCK state, we use the *rightrotate* function and a...h to calculate the hash[0..7]. If index == 64, we update the hash[0...7] to its value plus the a...h respectively and transit to the READ_SECOND_BLOCK, as well as resetting the index. Else, we stay at the the HASH_FIRST_BLOCK state.

In the READ_SECOND_BLOCK state, we set the w[index] to the mem_read_data and w[4] to 32’h8000000. In addition, we set the w[5..14] to 0 and w[15] to 32’d640. Then, we increment the index and mem_addr by 1. If index == 4, we store a...h to be the hash[0...7] and transit to the HASH_SECOND_BLOCK state. Else, we stay at the READ_SECOND_BLOCK state.

In the HASH_SECOND_BLOCK state, we use the *rightrotate* function and a...h to calculate the hash[0..7]. If index == 64, we update the hash[0...7] to its value plus a...h respectively and transit to the WRITE_HASH state, as well as resetting the index and set write_we == 1. Else, we stay at the HASH_SECOND_BLOCK state.

In the WRITE_HASH state, we increment the mem_addr and index by 1 and start dumping the hash[index] into mem_write_data one by one. If index == 8, we set done == 1 and transit to the IDLE state. Else, we stay the WRITE_HASH state.

Bitcoin hash algorithm description:

For the bitcoin hash, we created a helper module to hash one block at a time using SHA-256. We use one module to hash the first 16 words, and then we created 4 modules in parallel to compute the remaining hashes 4 at a time. Since there are 16 nonces we want to create hashes for, and we have 4 modules working in parallel, we repeat this hashing 4 times.

In the IDLE state, we initialize the values we need, like the a-h values for the SHA-256 hash, and the start control signals for the hashing modules. We stay in this state until start is asserted, where we transition to FIRST_BLOCK.

In the FIRST_BLOCK state, we read the first 16 words of the input message and use them as inputs for our first hash. Once 16 words are read, we assert the hash_start control signal to begin hashing the first block, and move on to the WAIT_BLOCK state, where we wait for the hashing to finish.

In the WAIT_BLOCK state, we essentially wait for the first block to be hashed. We then store the outputs in some temporary variables because we'll need to use them when hashing the remaining 3 words + nonce. Once the done signal from the hashing is asserted, we transition to the SECOND_BLOCK state.

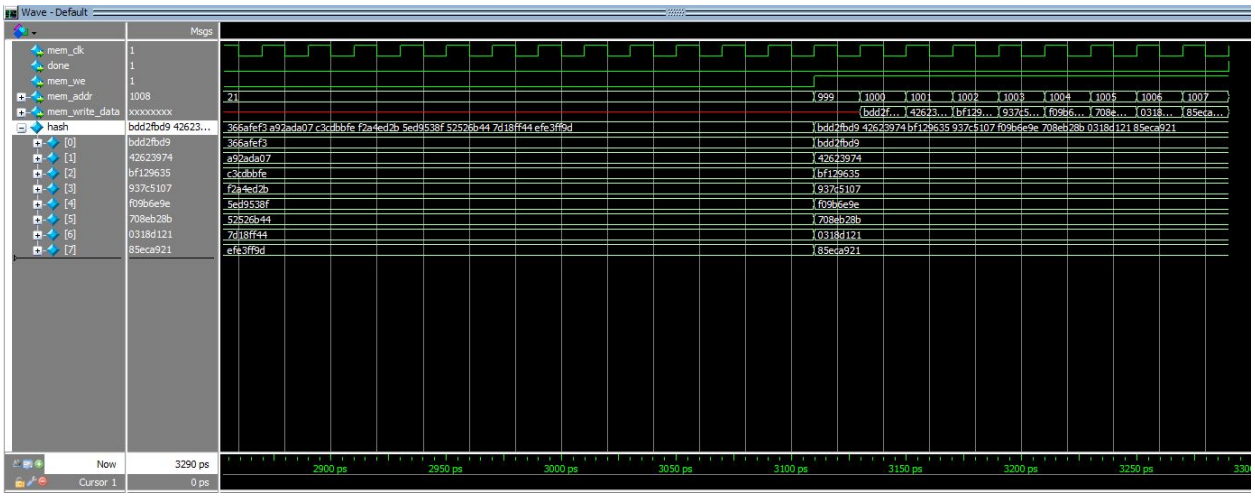
In the SECOND_BLOCK state, we read the remaining 3 words of the message and use them as the first 3 input words of our 4 parallelized hashing modules. After reading these 3 words, we update the remaining 13 input words with our nonce, padding, and length of the message. This is done for each module running in parallel. We then assert the hash_start_all signal in order to begin hashing 4 messages at once, and transition to WAIT_SECOND_BLOCK, where we just wait for the hashing to complete before moving on to SECOND_HASH.

In the SECOND_HASH state, we update the inputs of our parallelized modules with the output hash taken from the previous state, along with some padding. The purpose of this is to take the 8-word hash from the previous state, and hash it again. This is why the a-h values are reset to the default values, and why the length of this message is 256 bits. After doing this assignment, we assert hash_start_all again to begin the second hash, and wait for it to finish in WAIT_SECOND_HASH. Once that is complete, we transition to WRITE_HASH.

In the WRITE_HASH state, we take the H0 output from each of the 4 parallelized hashing modules, and store them in a final_hash array. This array is offset by the nonce value (e.g. if

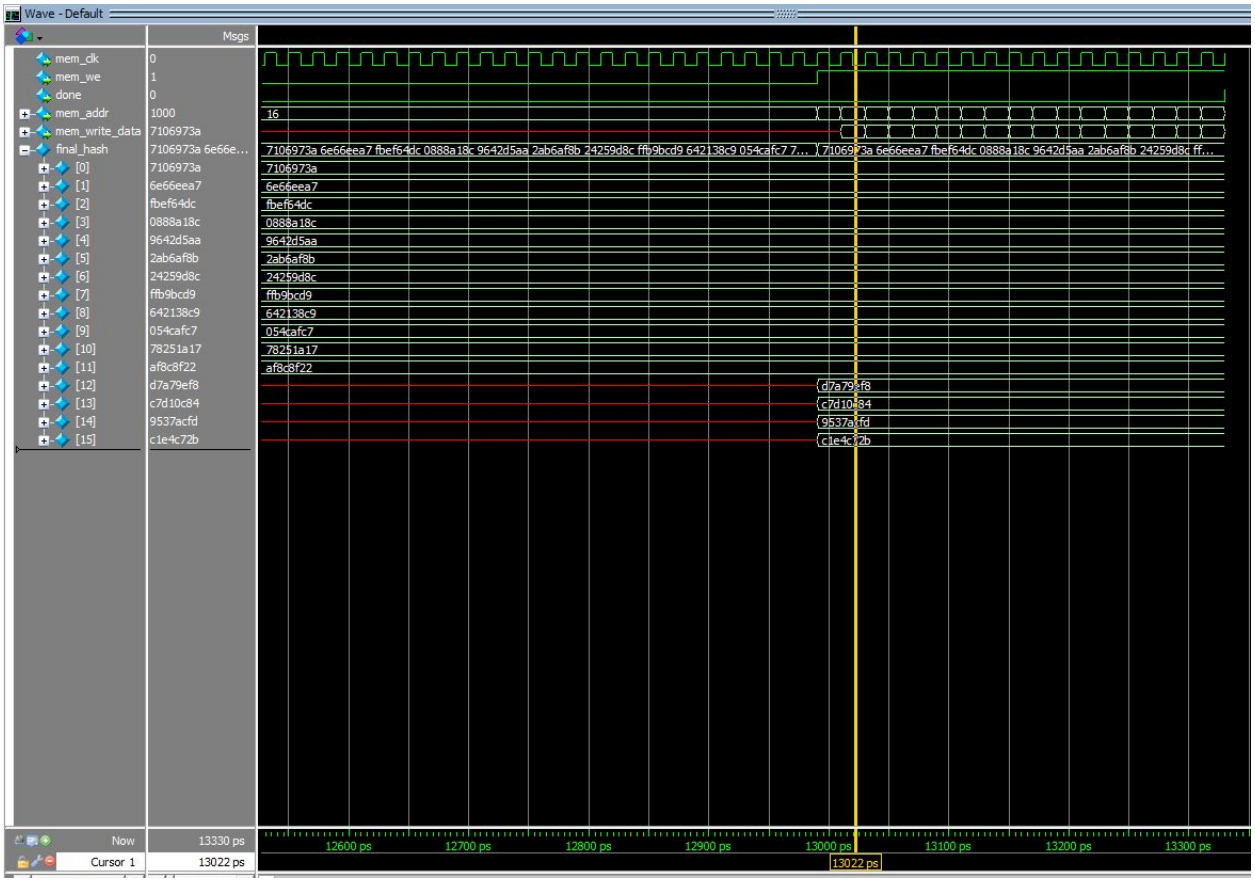
nonce = 4, the first hash module writes back to final_hash[0+4]), so that we are writing the hashes into the correct place. Final_hash[index] should correspond to the H0 from hashing the message with nonce = index. If we haven't written all 16 hashes to final_hash yet, we go back to the SECOND_HASH state to start hashing again. Otherwise, we can transition to DUMP_HASH amd write the values back to memory.
In the DUMP_HASH state, we write back the 16 H0's we stored in final_hash back to memory over 16 cycles. After writing these values back to memory, we revert back to the IDLE state.

SHA-256 and Bitcoin hashing simulation waveform snapshot
SHA-256 waveform snapshot



This is the waveform snapshot of final done signals and showing final output hash hexadecimal values for the SHA-256 project. We see that clearly correct hash values are written into the memory when the done signal is set to 1 at around 3125ps.

Bitcoin hashing waveform snapshot



This is the waveform snapshot of final done signals and showing final output hash hexadecimal values for the Bitcoin project. We see that clearly correct hash values are written into the memory when the done signal is set to 1 at around 13000ps.

Modelsim transcript window output for SHA-256 and Bitcoin hashing

Simplified SHA

```
VSIM 15> 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:         162
#
#
# ****************************
#
# ** Note: $stop    : H:/ecelll_final/tb_simplified_sha256.sv(260)
#    Time: 3290 ps  Iteration: 2  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at H:/ecelll_final/tb_simplified_sha256.sv line 260
```

Bitcoin

```
VSIM 4> 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:         664
#
#
# ****************************
#
# ** Note: $stop    : H:/ecelll_final/bitcoin_hash/tb_bitcoin_hash.sv(334)
#    Time: 13330 ps  Iteration: 2  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at H:/ecelll_final/bitcoin_hash/tb_bitcoin_hash.sv line 334
```

Synthesis resource usage and timing report for bitcoin_hash only.

Synthesis resource usage

	Resource	Usage
1	Estimated ALUTs Used	4859
1	-- Combinational ALUTs	4859
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	7877
3		
4	Estimated ALUTs Unavailable	98
1	-- Due to unpartnered combinational logic	98
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	4859
7	Combinational ALUT ...by number of inputs	
1	-- 7 input functions	6
2	-- 6 input functions	948
3	-- 5 input functions	19
4	-- 4 input functions	193
5	-- <=3 input functions	3693
8		
9	Combinational ALUTs by mode	
1	-- normal mode	2447
2	-- extended LUT mode	6
3	-- arithmetic mode	1766
4	-- shared arithmetic mode	640
10		
11	Estimated ALUT/register pairs used	9660
12		
13	Total registers	7877
1	-- Dedicated logic registers	7877
2	-- I/O registers	0
3	-- LUT_REGs	0

	Resource	Usage
7	Combinational ALUT ...by number of inputs	
1	-- 7 input functions	6
2	-- 6 input functions	948
3	-- 5 input functions	19
4	-- 4 input functions	193
5	-- <=3 input functions	3693
8		
9	Combinational ALUTs by mode	
1	-- normal mode	2447
2	-- extended LUT mode	6
3	-- arithmetic mode	1766
4	-- shared arithmetic mode	640
10		
11	Estimated ALUT/register pairs used	9660
12		
13	Total registers	7877
1	-- Dedicated logic registers	7877
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	7878
22	Total fan-out	48838
23	Average fan-out	3.76

Fitter report

Fitter Summary

<<Filter>>

Fitter Status	Successful - Mon Dec 09 12:12:34 2019
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	bitcoin_hash
Top-level Entity Name	bitcoin_hash
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	28 %
Total registers	7877
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	155.06 MHz	155.06 MHz	clk	