**Final Report**

**SHA-256 and Bitcoin hashing explanation**

For this project, "SHA-256" stands for "Secure Hash Algorithm", it is a cryptographic method of converting input data of any kind and size, into a string of fixed number of characters. Our goal is to compute a unique hash value for any input data or message and the input message must be less than or equal to 2^64 bits. In our case, our input "message" size is "hardcoded" to 20 words (640 bits), then we will have our final hashed output as eight 32 bits words.

In part 2 of this project, hashing is a cryptographic method of converting input data of any kind and size, into a string of fixed number of characters. In our case, our input message (Wt) is of size 20 words (19 words and 1 nonce value), hence the total output message size is 20 x 32 = 640 bits. Also, the size of each data block in bitcoin block chain is 512 bits. Our code should return an output as eight 32-bits words using chained SHA-256.

**SHA-256 and Bitcoin hashing algorithm description**

**SHA-256 algorithm description:**

We first assign the hash constants int k[0:63] and wrote helper functions for determine the number of blocks (to determine number of blocks in memory to fetch), sha256_op (sha 256 hash round), right rotate and get_w15 (to calculate w[15] to be used in the stage COMPUTE).

If reset_n == 0, we set cur_we to 0, and set state to IDLE.

If reset_n == 1, and state == start, we initialize the message digest (MD) buffers / output hash h0 to h7 with indicated values corresponds to each of them in the simplified_sha256 slides, and also initialize a to h. Then we set cur_addr to message_addr, temp_addr to output_addr, cur_we to 0, offset to 0, i to 0, j to 0, and state to INTER.

If reset_n == 1, and state == INTER, we set state to READ.

If reset_n == 1, and state == READ. For offset less than the number of words, we set message[offset] to mem_read_data, offset to offset + 1, and set state to INTER. Otherwise (when offset > number of words), we set message[20] to 32'h8000000 for append 1 after the message, and message[31] to 32'd640 to indicate our message length. Then for message[21] to message[30], we set them to 0 for padding, offset to 0, and set state to BLOCK.

If reset_n == 1, and state == BLOCK. We get a block from the memory, COMPUTE hash output using SHA256 function and write back hash value back to the memory. We first fetch message in 512-bit block size, for each of the 512-bit block we initiate hash value computation. If i is less

than num_blocks, set w[n] to message[n + (i*16)] for n = 0 to n = 15, then set state to COMPUTE, else we set state to PIPE.

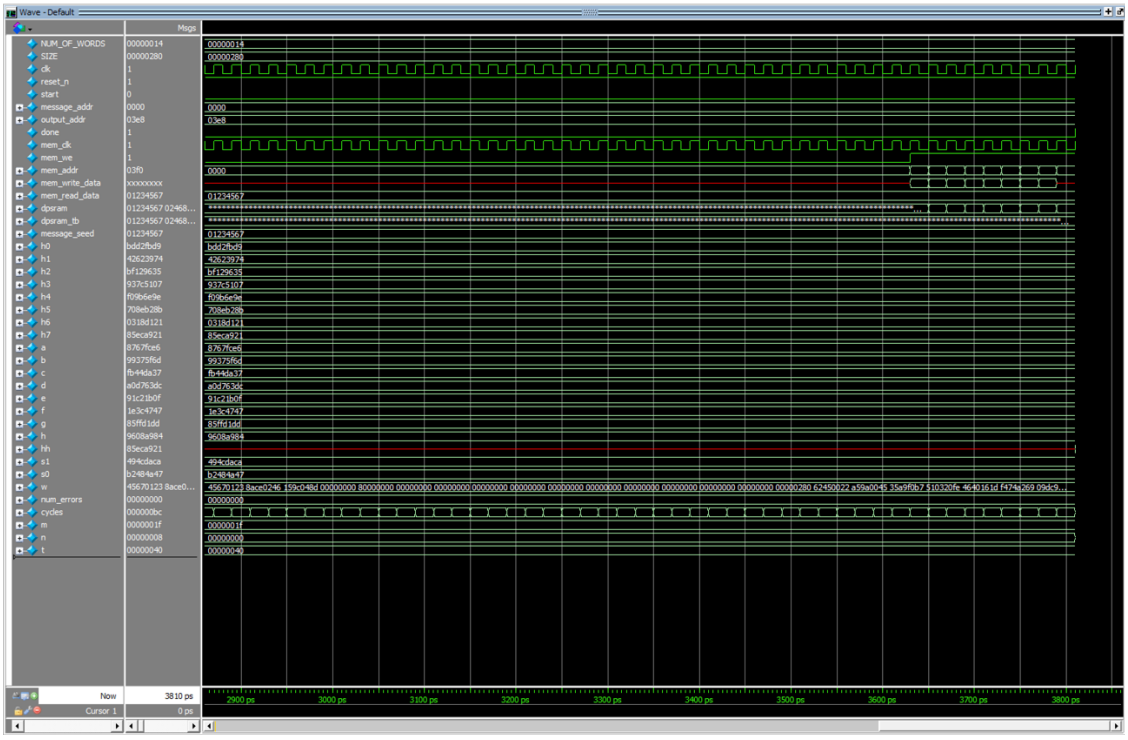
If reset_n == 1, and state == COMPUTE. For j less than 64, j is for counting the compression round, our hash values a to h are computed by our sha_op256 helper function. For w[0] to w[14], we shift every bit to the right by one, and use our get_w15 helper function to get our value for w[15]. J is set to j + 1, then state to COMPUTE. If j is greater than 64, we set h0 to a + h0, h1 to b + h1, h2 to c + h2, all the way to set h7 to h + h7. Also, we set a to a + h0, b to b + h1, all the way to h to h + h7. Then we reset j to 0 and move to the BLOCK state after each block hash computation is completed. At the end of COMPUTE stage, we have our final computed hash value store in h0 to h7.

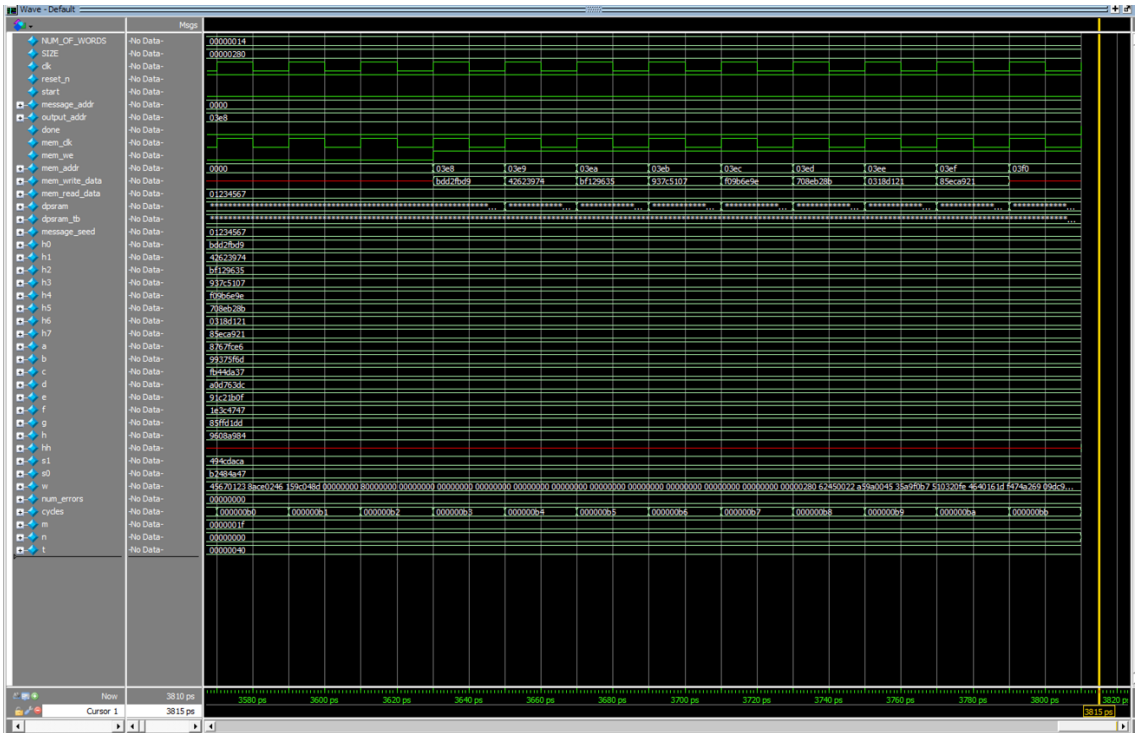
If reset_n == 1 and state == PIPE, we set temp_hash[0] to h0, all the way to temp_hash[7] to h7. Cur_we to 1, cur_addr to temp_addr cur_write_data to h0, then move to the WRITE state.

If reset_n == 1 and state == WRITE, if our offset value is less than 8, which means we did not finish writing our hash value for all eight blocks of output, then we need to set cur_write_data to temp_hash[offset + 1] indicating the index that we are writing on, offset to offset + 1 and move to state WRITE again. If offset = 8, we move to the IDLE state. Done is generated when SHA256 hash computation has finished and moved to IDLE state.

**Bitcoin hashing algorithm description:**

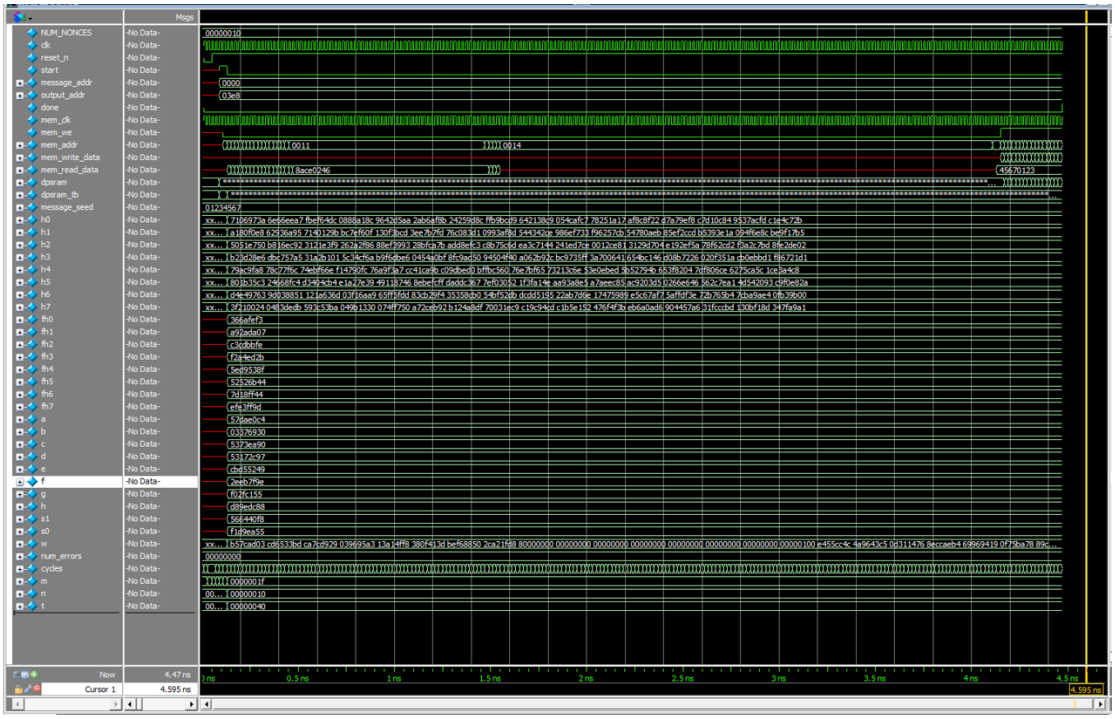
**SHA-256 and Bitcoin hashing simulation waveform snapshot**
**SHA-256 waveform snapshot**

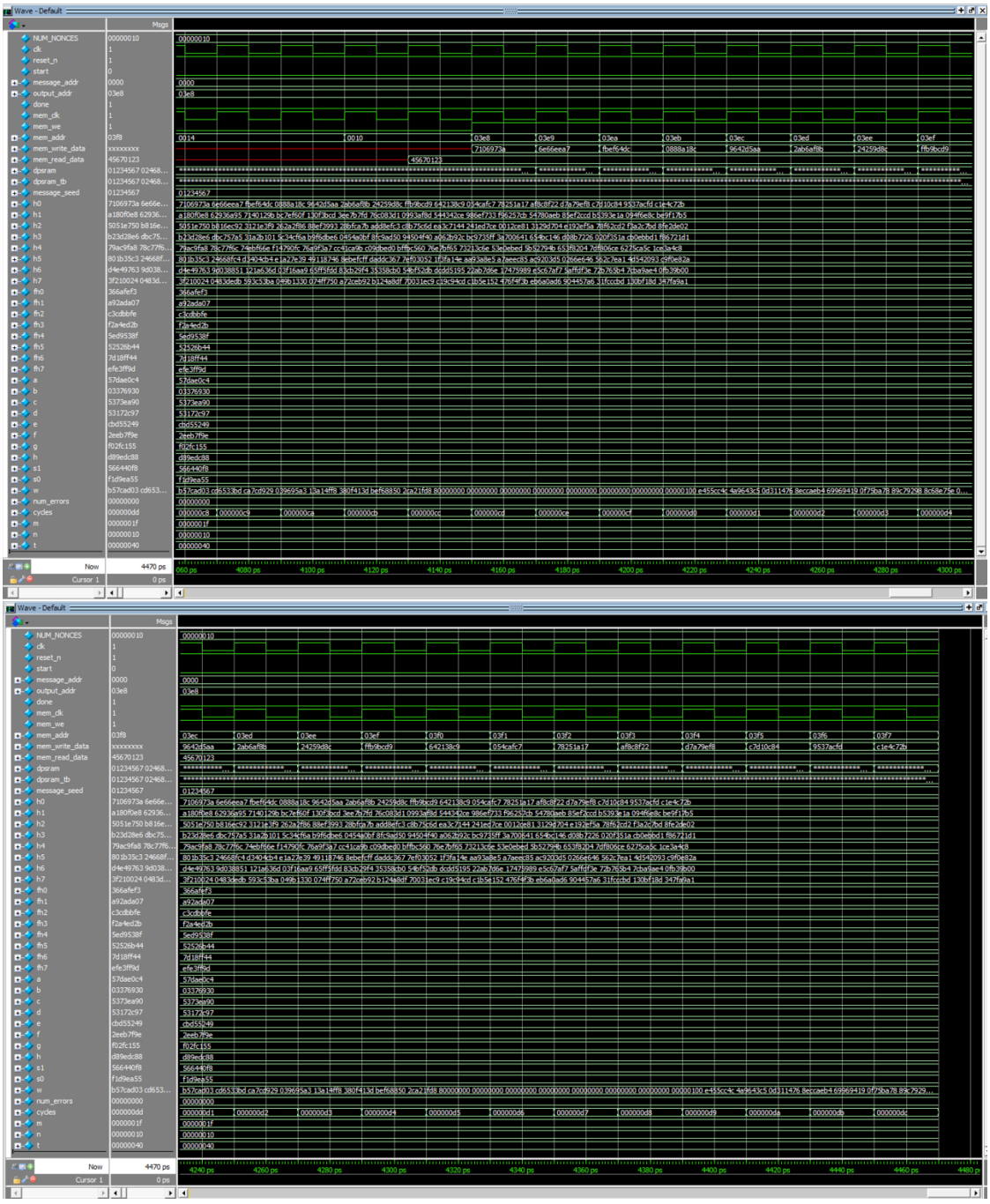




This is the waveform snapshot of final done signals and showing final output hash hexadecimal values for the SHA-256 project. We see that clearly correct hash values are written into the memory when the mem_we signal is set to 1 at around 3700ps.

## 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 mem_we signal is set to 1 at around 4.15ns.

**Modelsim transcript window output for SHA-256 and Bitcoin hashing**

Simplified SHA

```
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:         188
#
#
# ****************************
#
# ** Note: $stop    : L:/Final Project/tb_simplified_sha256.sv(262)
#    Time: 3810 ps  Iteration: 2  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at L:/Final Project/tb_simplified_sha256.sv line 262

VSIM 7>
```

# Bitcoin

Transcript

```
# 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    : L:/Final Project/tb_bitcoin_hash.sv(334)
#    Time: 4470 ps  Iteration: 2  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at L:/Final Project/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 | 18340 |
| 1 | -- Combinational ALUTs | 18340 |
| 2 | -- Memory ALUTs | 0 |
| 3 | -- LUT_REGs | 0 |
| 2 | Dedicated logic registers | 17512 |
| 3 | | |
| 4 | Estimated ALUTs Unavailable | 317 |
| 1 | -- Due to unpartnered combinational logic | 317 |
| 2 | -- Due to Memory ALUTs | 0 |
| 5 | | |
| 6 | Total combinational functions | 18340 |
| 7 | Combinational ALUT usage by number of inputs | |
| 1 | -- 7 input functions | 317 |
| 2 | -- 6 input functions | 2476 |
| 3 | -- 5 input functions | 1113 |
| 4 | -- 4 input functions | 16 |
| 5 | -- <=3 input functions | 14418 |
| 8 | | |
| 9 | Combinational ALUTs by mode | |
| 1 | -- normal mode | 9742 |
| 2 | -- extended LUT mode | 317 |
| 3 | -- arithmetic mode | 6745 |
| 4 | -- shared arithmetic mode | 1536 |
| 10 | | |
| 11 | Estimated ALUT/register pairs used | 23381 |
| 12 | | |
| 13 | Total registers | 17512 |
| 1 | -- Dedicated logic registers | 17512 |
| 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 | 17513 |
| 22 | Total fan-out | 131358 |

| | Resource | Usage |
|---|---|---|
| 23 | Average fan-out | 3.64 |

Fitter report

**Fitter Summary**

<<Filter>>

| | |
|---|---|
| Fitter Status | Successful - Sat Mar 20 07:35:56 2021 |
| Quartus Prime Version | 20.1.0 Build 711 06/05/2020 SJ Lite Edition |
| Revision Name | Final_Project |
| Top-level Entity Name | bitcoin_hash |
| Family | Arria II GX |
| Device | EP2AGX45DF29I5 |
| Timing Models | Final |
| Logic utilization | 77 % |
| Total registers | 17512 |
| 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 | 141.78 MHz | 141.78 MHz | clk | |