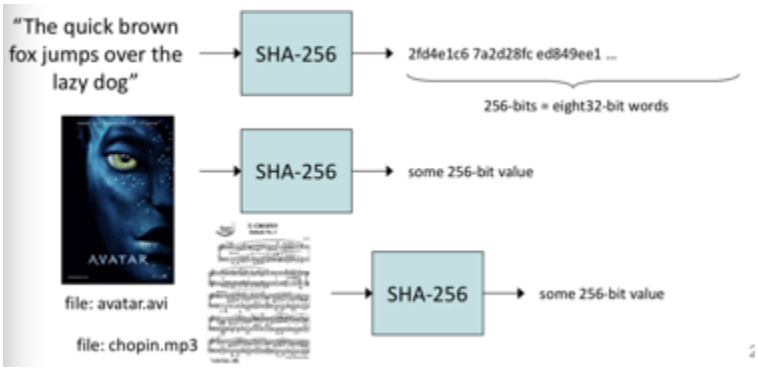
FPGA Design and Simulation of SHA-256 Bitcoin Hashing in SystemVerilog RTL

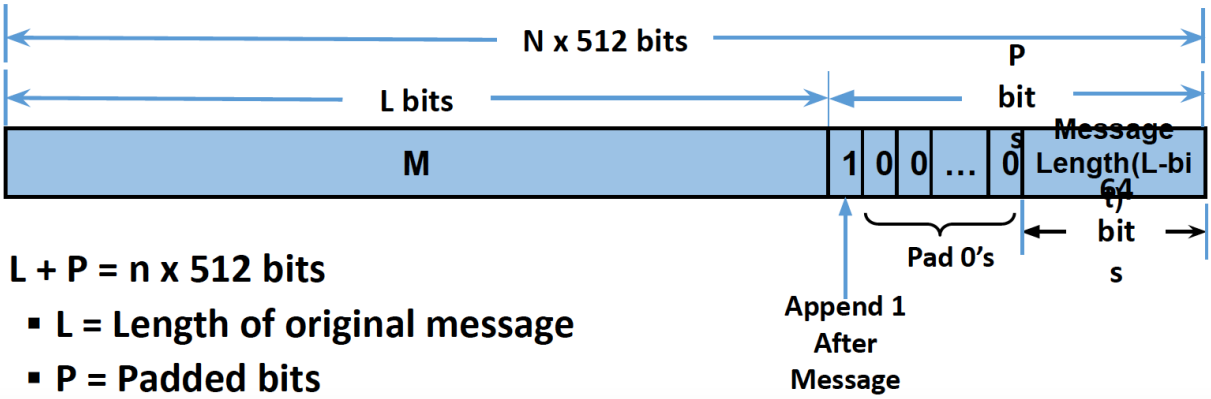
Jason Liang (A16652882)

Introduction

SHA-256 algorithm is a secure cryptographic algorithm for mapping a uniquely generated hash value for any input message, even if two messages appear very similar but only differ by one bit. The return value is a 256-bit hash value.



The general algorithm consists of five steps. First, a message *M* will be fed into the algorithm and a single '1' bit will be appended to the end of the message. The remainder of the message is padded with '0' bits until the message equals 448 mod 512. In step 2, the length of the message is assigned an unsigned value at the last 64 least significant bits. The division of bits can be shown in the following image.



In step 3, we initialize the message digest to eight 32-bit words. Then in step 4, the message M is processed by being divided into 512-bit blocks M_0 , M_1, …, M_j and each of these blocks is further processed. First, the 8 32-bit words need to be initialized as (A, B, C, D, E, F, G, H) = H_0 , H_1, …, H_7. Then, there is a 64-iteration processing of the 512-bit blocks computing a word expansion of a 32-bit block. This can be shown below.

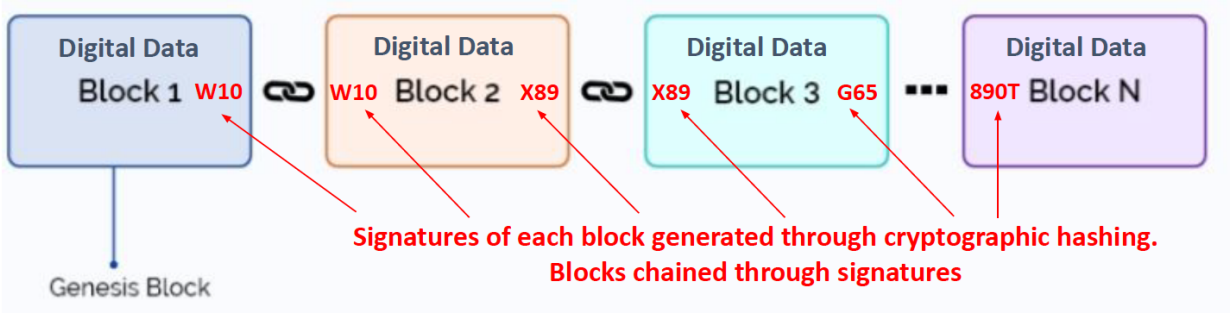
Each step t ($0 \le t \le 63$): Word expansion for W_t

- If $t < 16$
  - W_t = t^{th} 32-bit word of block M_j
- If $16 \le t \le 63$
  - $s_0 = (W_{t-15}$ **rightrotate** 7) **xor** (W_{t-15} **rightrotate** 18) **xor** (W_{t-15} **rightshift** 3)
  - $s_1 = (W_{t-2}$ **rightrotate** 17) **xor** (W_{t-2} **rightrotate** 19) **xor** (W_{t-2} **rightshift** 10)
  - $W_t = W_{t-16} + s_0 + W_{t-7} + s_1$

Further arithmetic operations continue in step 4 and the results are to be appended to the eight hash values we initialized earlier. The final 256-bit hash value of message M will be provided as the bitwise concatenation of H_0 , H_1, …, H_7.

Because of its success as a hashing algorithm, SHA-256 is commonly used in security applications, and in this paper, we use SHA-256 as the core algorithm behind encoding a message for bitcoin mining. Bitcoin is a digital type of cryptocurrency in which bitcoin units can be sent from person to person without the need for an administrator or intermediaries. This can be somewhat insecure, which is why hashing cryptography algorithms must be foolproof, reliant, and performance-efficient. In bitcoin lies the concept of blockchain, a distributed and

decentralized public ledger that records all bitcoin transactions. This consists of a chain of blocks, hence the name, where each block contains the hash of the previous block up to the genesis block.

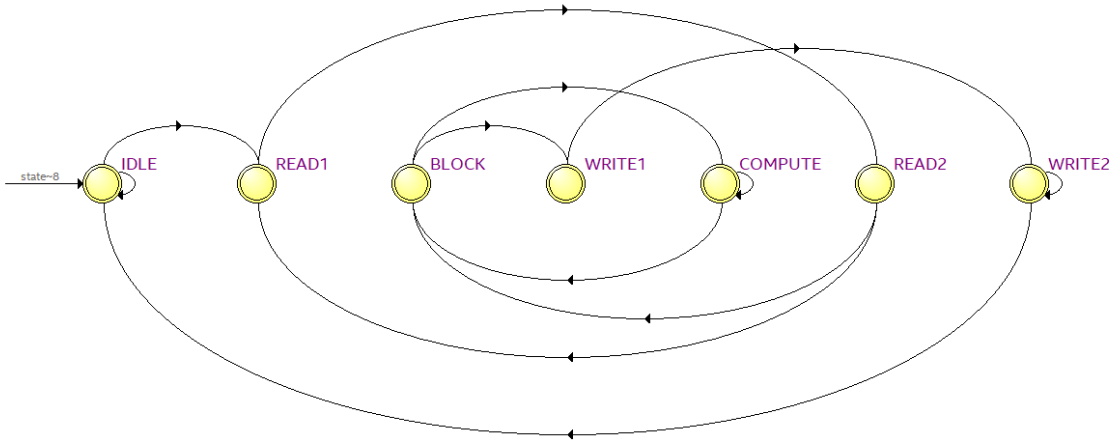


Blocks which are connected are immutable. One other good security feature of the blockchain is that each block contains a SHA-256 cryptographic message which is connected to previous blocks. The network is linked via a "proof-of-work", which is a piece of data which is costly and time consuming to produce but easy to verify. It requires many trials and errors on average before a valid "proof-of-work" is generated, hence yielding an unalterable set of blocks. "Proof-of-work" is generated by looking for nonces, which is also very difficult to generate.

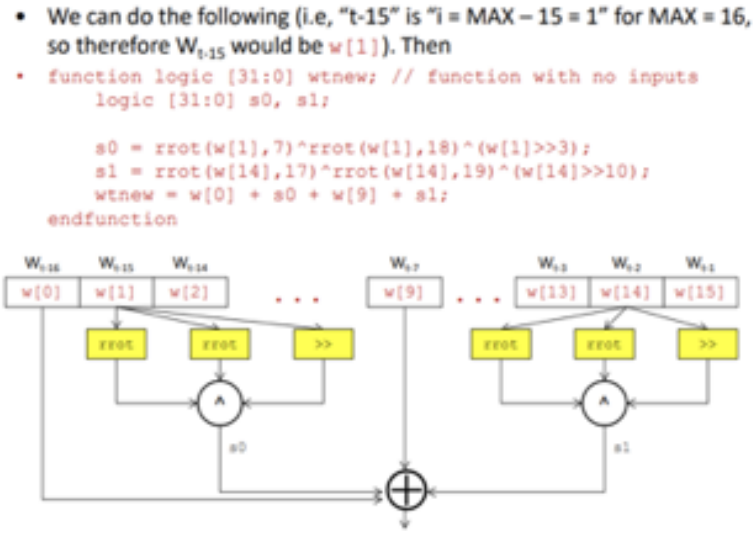
PART 1: SHA-256 Algorithm

Implementation and Optimizations

The approach can be described by the finite state machine shown below. We first begin with an IDLE stage which computes the initial hash values of a given message input.

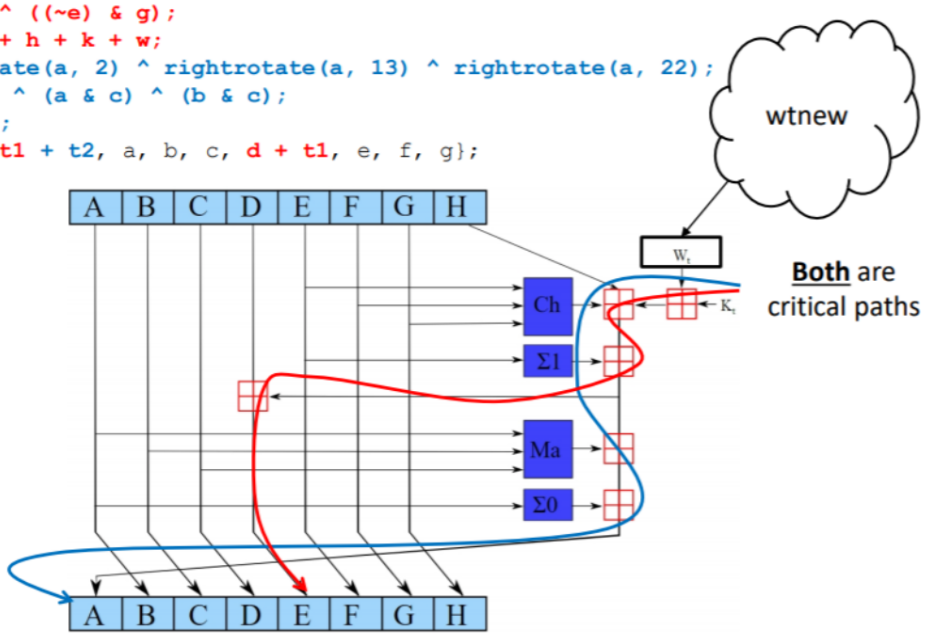


Extra read and write states ensure that data arrives at the correct cycle before assuming calculations in the next state. Non-blocking statements are also used to ensure that the clock frequency is high and to reduce resource utilization and area. Most importantly, however, are the optimization design choices. In particular, READ2 can read the first N-1 bits where N is the NUM_OF_WORDS parameter. We also realized that the newest ‘w’ computed can be placed in w[15] with all the previously stored values after an arithmetic shift left by 1. This method of storing the ‘w’ values is efficient since only the previous 16 of ‘w’ values are required each time the SHA-256 hash operation is computed. This reduces the number of muxes, and hence, the area of the design. The below image illustrates this logic.



The other significant optimization is shown in the image below when we write the function sha256_op for vectorization of initializing the hash computations. A critical path exists when computing t1 and t2; we remedied this by pre-computing h + k + w, which are the only values that were previously available for computation. Specifically, the values of ‘k’ and ‘w’ were already known. In addition, the value of ‘h’ could be precomputed one cycle earlier as it is just g from the previous cycle. This optimization preserved the delay efficiency of the design.

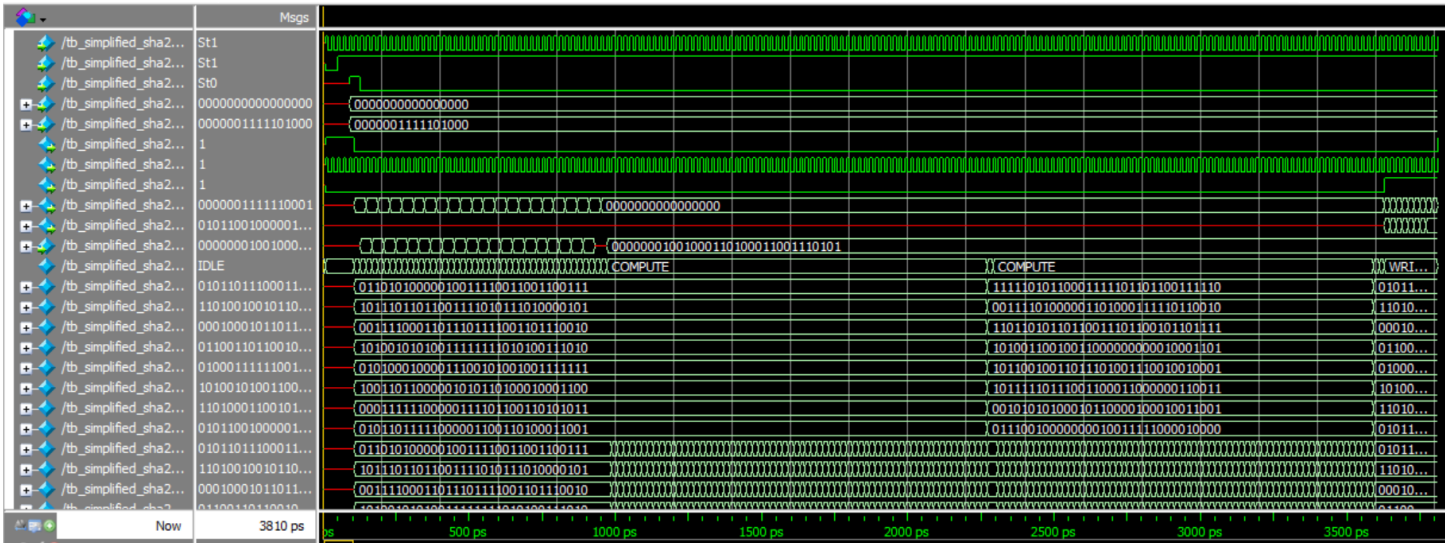
```
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w, k);
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
begin
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
    ch = (e & f) ^ ((~e) & g);
    t1 = ch + S1 + h + k + w;
    S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
    maj = (a & b) ^ (a & c) ^ (b & c);
    t2 = maj + S0;
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
end
endfunction
```



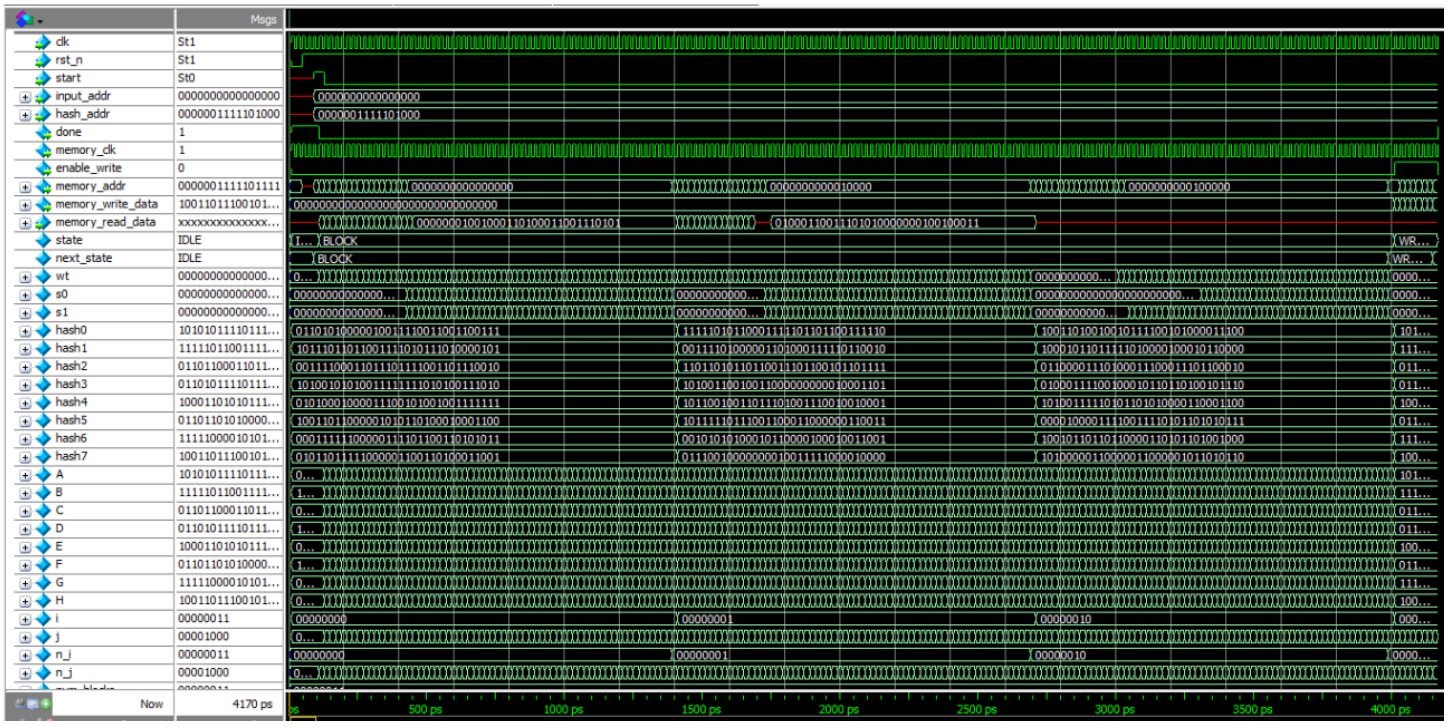
In our implementation, resource utilization remains the same regardless of the number of words we read because none of the RTL depends on this parameter; the testbench automatically changes the RTL to deal with such cases so the synthesis is the same.

WAVEFORMS

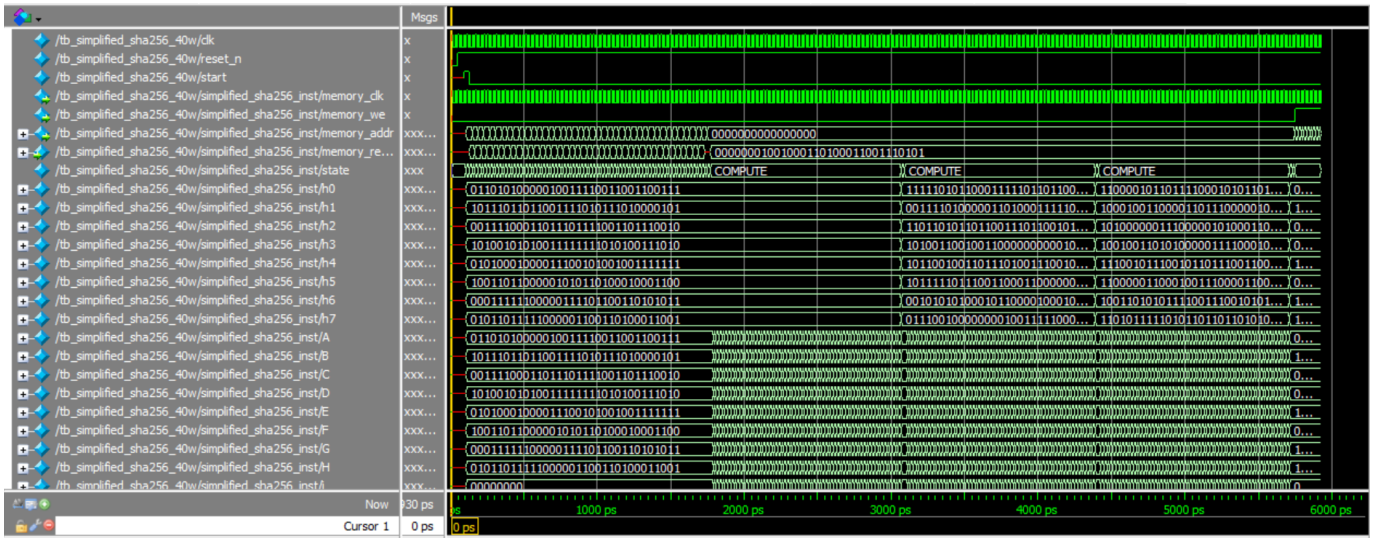
20 words



30 words



40 words



TRANSCRIPTS

20 words

```
# ---------------------
# COMPARE HASH RESULTS:
# ---------------------
# Correct H[0] = 5b8feb0a Your H[0] = 5b8feb0a
# Correct H[1] = d258a227 Your H[1] = d258a227
# Correct H[2] = 116df790 Your H[2] = 116df790
# Correct H[3] = 66c9d8cc Your H[3] = 66c9d8cc
# Correct H[4] = 47e75276 Your H[4] = 47e75276
# Correct H[5] = a5316e2f Your H[5] = a5316e2f
# Correct H[6] = d1965a81 Your H[6] = d1965a81
# Correct H[7] = 5904edff Your H[7] = 5904edff
# ***************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:         188
#
#
# ***************************
```

30 words

```
# ---------------------
# COMPARE HASH RESULTS:
# ---------------------
# Correct H[0] = abde57db Your H[0] = abde57db
# Correct H[1] = fb3f1bda Your H[1] = fb3f1bda
# Correct H[2] = 6c6f969c Your H[2] = 6c6f969c
# Correct H[3] = 6bdea244 Your H[3] = 6bdea244
# Correct H[4] = 8d5ff7cf Your H[4] = 8d5ff7cf
# Correct H[5] = 6d41389a Your H[5] = 6d41389a
# Correct H[6] = f85598ec Your H[6] = f85598ec
# Correct H[7] = 9b97cccl Your H[7] = 9b97cccl
# ***************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:         274
#
#
# ***************************
```

40 words

```
# ---------------------
# COMPARE HASH RESULTS:
# ---------------------
# Correct H[0] = 0244238c Your H[0] = 0244238c
# Correct H[1] = d2a3d7be Your H[1] = d2a3d7be
# Correct H[2] = 63ad61e7 Your H[2] = 63ad61e7
# Correct H[3] = 44f2fad8 Your H[3] = 44f2fad8
# Correct H[4] = f0da0c1b Your H[4] = f0da0c1b
# Correct H[5] = 10d16d52 Your H[5] = 10d16d52
# Correct H[6] = 86e5e36d Your H[6] = 86e5e36d
# Correct H[7] = a108b1c4 Your H[7] = a108b1c4
# ****************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:         294
#
#
# ****************************
```

RESOURCE UTILIZATION

20 words

	Resource	Usage
1	-- Combinational ALUTs	1707
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	2136
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	1707
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	140
3	-- 5 input functions	102
4	-- 4 input functions	520
5	-- <=3 input functions	945
8		
9	Combinational ALUTs by mode	
1	-- normal mode	1210
2	-- extended LUT mode	0
3	-- arithmetic mode	369
4	-- shared arithmetic mode	128
10		
11	Estimated ALUT/register pairs used	2627
12		
13	Total registers	2136
1	-- Dedicated logic registers	2136
2	-- I/O registers	0
3	-- LUT_REGs	0

30 words

	Resource	Usage
1	-- Combinational ALUTs	1707
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	2136
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	1707
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	140
3	-- 5 input functions	102
4	-- 4 input functions	520
5	-- <=3 input functions	945
8		
9	Combinational ALUTs by mode	
1	-- normal mode	1210
2	-- extended LUT mode	0
3	-- arithmetic mode	369
4	-- shared arithmetic mode	128
10		
11	Estimated ALUT/register pairs used	2627
12		
13	Total registers	2136
1	-- Dedicated logic registers	2136
2	-- I/O registers	0
3	-- LUT_REGs	0

40 words

	Resource	Usage
1	-- Combinational ALUTs	1707
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	2136
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	1707
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	140
3	-- 5 input functions	102
4	-- 4 input functions	520
5	-- <=3 input functions	945
8		
9	Combinational ALUTs by mode	
1	-- normal mode	1210
2	-- extended LUT mode	0
3	-- arithmetic mode	369
4	-- shared arithmetic mode	128
10		
11	Estimated ALUT/register pairs used	2627
12		
13	Total registers	2136
1	-- Dedicated logic registers	2136
2	-- I/O registers	0
3	-- LUT_REGs	0

PART 2: Bitcoin Hashing

Implementation and Optimizations

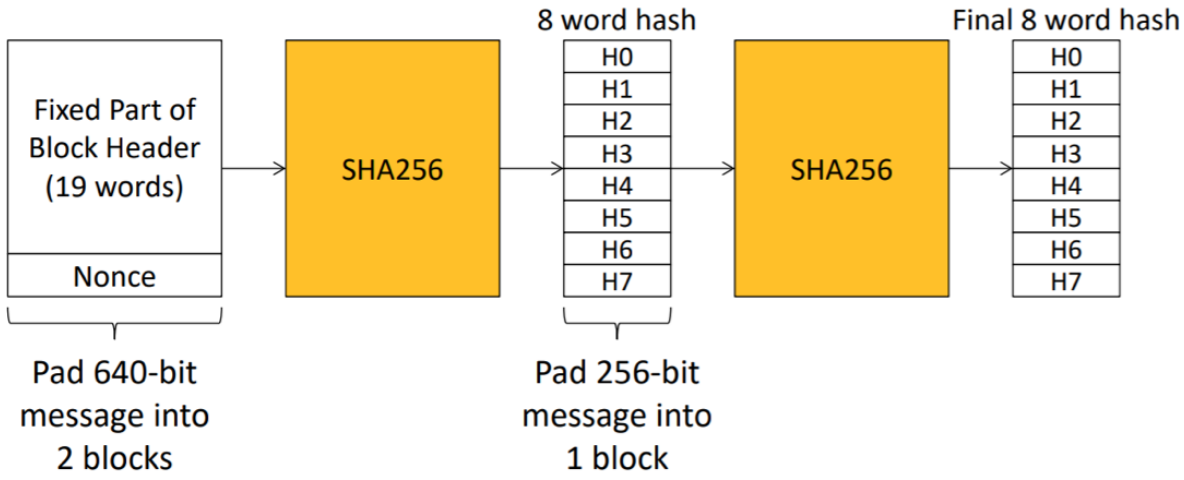
The bitcoin hashing algorithm involves using the SHA-256 algorithm to generate nonces. We are computing the hash values of 16 nonces without checking if any is less than target. This time, the finite state machine will be based on phases of computing the nonce as shown.

There are 3 phases:

- **Phase 1:** Processing 1st block of the 1st SHA 256 hash function
  - H0…H7 correspond to constants, 32'h6a09e667, etc.
  - Wt's correspond to first 16 words in memory
- **Phase 2:** Processing 2nd block of the 1st SHA 256 hash function
  - H0…H7 come from the Phase 1
  - Wt's correspond the last 3 words in memory, the nonce, 32'h80000000, ten 32'h00000000 padding, and 32'd640
- **Phase 3:** Processing the 2nd SHA 256 hash function
  - H0…H7 correspond to constants, 32'h6a09e667, etc.
  - Wt's correspond the H0…H7 from Phase 2, 32'h80000000, six 32'h00000000 padding, and 32'd256
- **Phase-2 and 3 are performed 16 times. This will produce 16 finals hashes.**

The first SHA-256 operation result is computed by processing 19 words of the block header in the READ state. The input message is then divided into 512-bit blocks and each block is run with the SHA-256 hash operation twice with the output of the first computation fed into the second computation. For design optimization choices, we greatly reduced the logic for the previous part of the project so that it would suit the needs of the part, including removing any non-blocking statements, if statements, and variables, which reduced the area. Another important realization is that phases 1, 2, and 3 are inherently serial; we cannot compute the next phase without information from the previous phase. However, we can perform phases 2 and 3 in parallel by initializing 16 instances of our simplified SHA-256 module. In addition, we used vectorization as

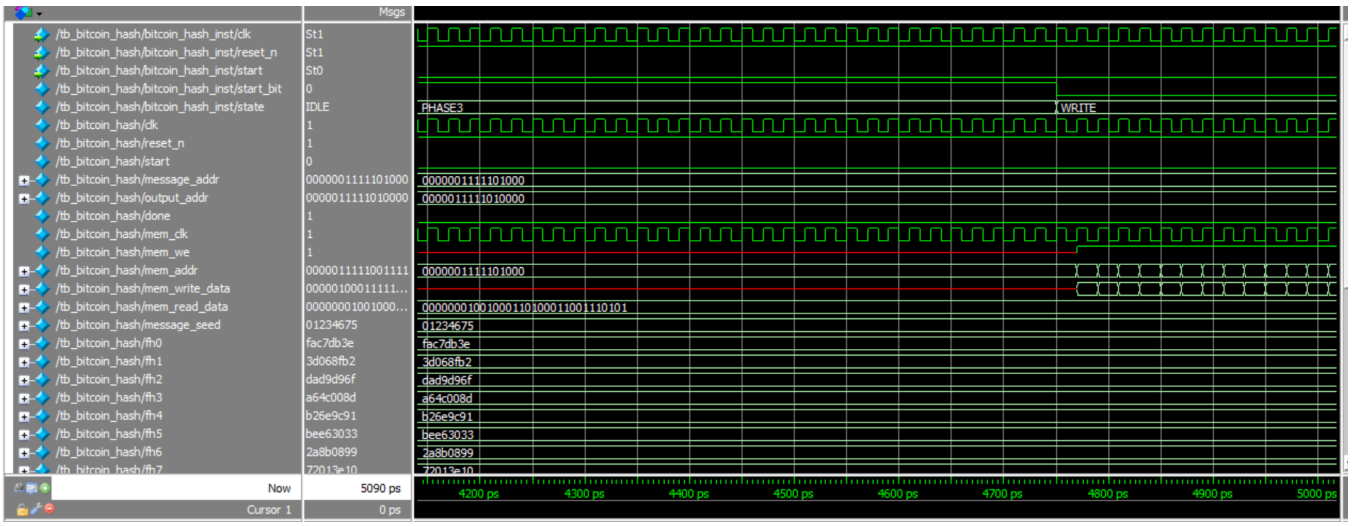
previously mentioned in Part 1 to initialize the 16 sets of the hash registers under the same state machine control to further reduce delays.



Final area and delay optimizations from the Quartus compilation is shown below.

bitcoin_hash.sv (MIN AREA*DELAY DESIGN)							
Compiler Settings	#ALUTs	#Registers	Area	Fmax (MHz)	#Cycles	Delay (microsec)	Area*Delay (millisec*area)
Balanced	19028	26879	45907	140.77	252	1.790	82.174

WAVEFORM



TRANSCRIPT

```
# ---------------------
# COMPARE HASH RESULTS:
# ---------------------
# Correct H0[ 0] = a0211662 Your H0[ 0] = a0211662
# Correct H0[ 1] = bfbb6ccd Your H0[ 1] = bfbb6ccd
# Correct H0[ 2] = da017047 Your H0[ 2] = da017047
# Correct H0[ 3] = 1c34e2aa Your H0[ 3] = 1c34e2aa
# Correct H0[ 4] = 58993aea Your H0[ 4] = 58993aea
# Correct H0[ 5] = b41b7a67 Your H0[ 5] = b41b7a67
# Correct H0[ 6] = 04cf2ceb Your H0[ 6] = 04cf2ceb
# Correct H0[ 7] = 85ab3945 Your H0[ 7] = 85ab3945
# Correct H0[ 8] = f4539616 Your H0[ 8] = f4539616
# Correct H0[ 9] = 0e4614d7 Your H0[ 9] = 0e4614d7
# Correct H0[10] = 6bec8208 Your H0[10] = 6bec8208
# Correct H0[11] = ce75ecf2 Your H0[11] = ce75ecf2
# Correct H0[12] = 672cb1a0 Your H0[12] = 672cb1a0
# Correct H0[13] = 4d48232a Your H0[13] = 4d48232a
# Correct H0[14] = cfe99db3 Your H0[14] = cfe99db3
# Correct H0[15] = 047d81b9 Your H0[15] = 047d81b9
# ***************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:         252
#
#
# ***************************
```

RESOURCE UTILIZATION

	Resource	Usage
1	Estimated ALUTs Used	19028
1	-- Combinational ALUTs	19028
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	26879
3		
4	Estimated ALUTs Unavailable	388
1	-- Due to unpartnered combinational logic	388
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	19028
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	3826
3	-- 5 input functions	3597
4	-- 4 input functions	41
5	-- <=3 input functions	11564
8		
9	Combinational ALUTs by mode	
1	-- normal mode	11779
2	-- extended LUT mode	0
3	-- arithmetic mode	5201
4	-- shared arithmetic mode	2048
10		
11	Estimated ALUT/register pairs used	34434
12		
13	Total registers	26879
1	-- Dedicated logic registers	26879
2	-- I/O registers	0

STATIC TIMING ANALYSIS REPORT

```
+-----------------------------------------------------+
; Slow 900mV 100C Model Fmax Summary                  ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 140.77 MHz ; 140.77 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
```

FITTER REPORT

Fitter Summary

<<Filter>>

Fitter Status	Successful - Fri Dec 09 23:25:17 2022
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	92 %
Total registers	26879
Total pins	118 / 26879 (%)
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 %)