Design for SHA256 and Bitcoin Hashing

ECE 111 Final Project

Spring Quarter 2020

Team Members:

Girish Gowtham Aravindan (A53310752)
Richa Pallavi (A53302809)
Mahima Rathore (A53297810)

Introduction

Bitcoin is a cryptocurrency that enables users to exchange this form of electronic cash with each other without the need for intermediaries. It is a decentralized digital currency without a central bank. To keep track of all transactions, a "blockchain" is used to maintain a growing list of records. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. This "blockchain" acts as a global ledger for all transactions, where new transactions are added as new blocks to the blockchain. By design, a blockchain is resistant to modification of the data.

A blockchain is secure because new blocks can only be added to the blockchain by someone (a bitcoin miner) finding a cryptographic hash for the new block that satisfies some computationally difficult condition. Bitcoin is backed by millions of computers across the world called "miners.". When bitcoin miners add a new block of transactions to the blockchain, part of their job is to make sure that those transactions are accurate. Bitcoin miners also make sure that bitcoin is not being duplicated, a unique quirk of digital currencies called "double-spending". This "proof-of-work" (which is referred to as "Bitcoin mining") is what makes blockchains secure and unalterable.

Description of the SHA-256 Algorithm

SHA-256 algorithm is a core of Bitcoin mining. SHA-256 algorithm is performed in 5 steps. The input message should be less than 2^{64}bits. Message is processed in 512-bit block size sequentially.

Step 1: Padding the message

- Single 1 bit is added to end of the message
- Rest of the message is padded with zeros until the message length is 64-bit less than some multiple of 512.

Step 2: Appending length as 64 bits

- Last 64 bits represent the message size

Step 3: Buffer initiation

- Message digest is initialized with the following eight 32-bit words
  H0 = 32'h6a09e667
  H1 = 32'hbb67ae85
  H2 = 32'h3c6ef372
  H3 = 32'ha54ff53a
  H4 = 32'h510e527f
  H5 = 32'h9b05688c
  H6 = 32'h1f83d9ab
  H7 = 32'h5be0cd19

Step 4: Processing of the message

- Message is divided into the blocks of 512 bits which are processed sequentially for 64 times.
- Input to this are Wt (32-bit word from message), Kt (a constant array), current message digest. Output is the new message digest. At the beginning of the processing, initialize
  (A, B, C, D, E, F, G, H) = (H0, H1, H2, H3, H4, H5, H6, H7)
- For each 64 round of processing Wt is determined as following:
  If $t < 16$: $W_t = t^{th}$ 32-bit word of block Mj
  If $16 \le t \le 63$:
  – s0 = (W_{t-15} **rightrotate** 7) **xor** (W_{t-15} **rightrotate** 18) **xor** (W_{t-15} **rightshift** 3)
  – s1 = (W_{t-2} **rightrotate** 17) **xor** (W_{t-2} **rightrotate** 19) **xor** (W_{t-2} **rightshift** 10)
  – $W_t = W_{t-16} + s0 + W_{t-7} + s1$

At each step t ($0 \le t \le 63$) processing is done in following manner:

S0 = (A **rightrotate** 2) **xor** (A **rightrotate** 13) **xor** (A **rightrotate** 22)
maj = (A **and** B) **xor** (A **and** C) **xor** (B and C)
t2 = S0 + maj
S1 = (E **rightrotate** 6) **xor** (E **rightrotate** 11) **xor** (E **rightrotate** 25)
ch = (E **and** F) **xor** ((**not** E) **and** G)
t1 = H + S1 + ch + Kt + Wt
(A, B, C, D, E, F, G, H) = (t1 + t2, A, B, C, D + t1, E, F, G)

After processing it for 64 rounds, final A, B, C…. are added back to the initial message digest H0, H1, H2….

- Step 5: Final output hash values are generated in H0, H1, H2, H3, H4, H5, H6, H7, H8. The generated hash is unique as the message went through a lot of shuffling and processing.

Transcript:

```
VSIM 5> restart
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:         189
#
```

Resource usage summary:

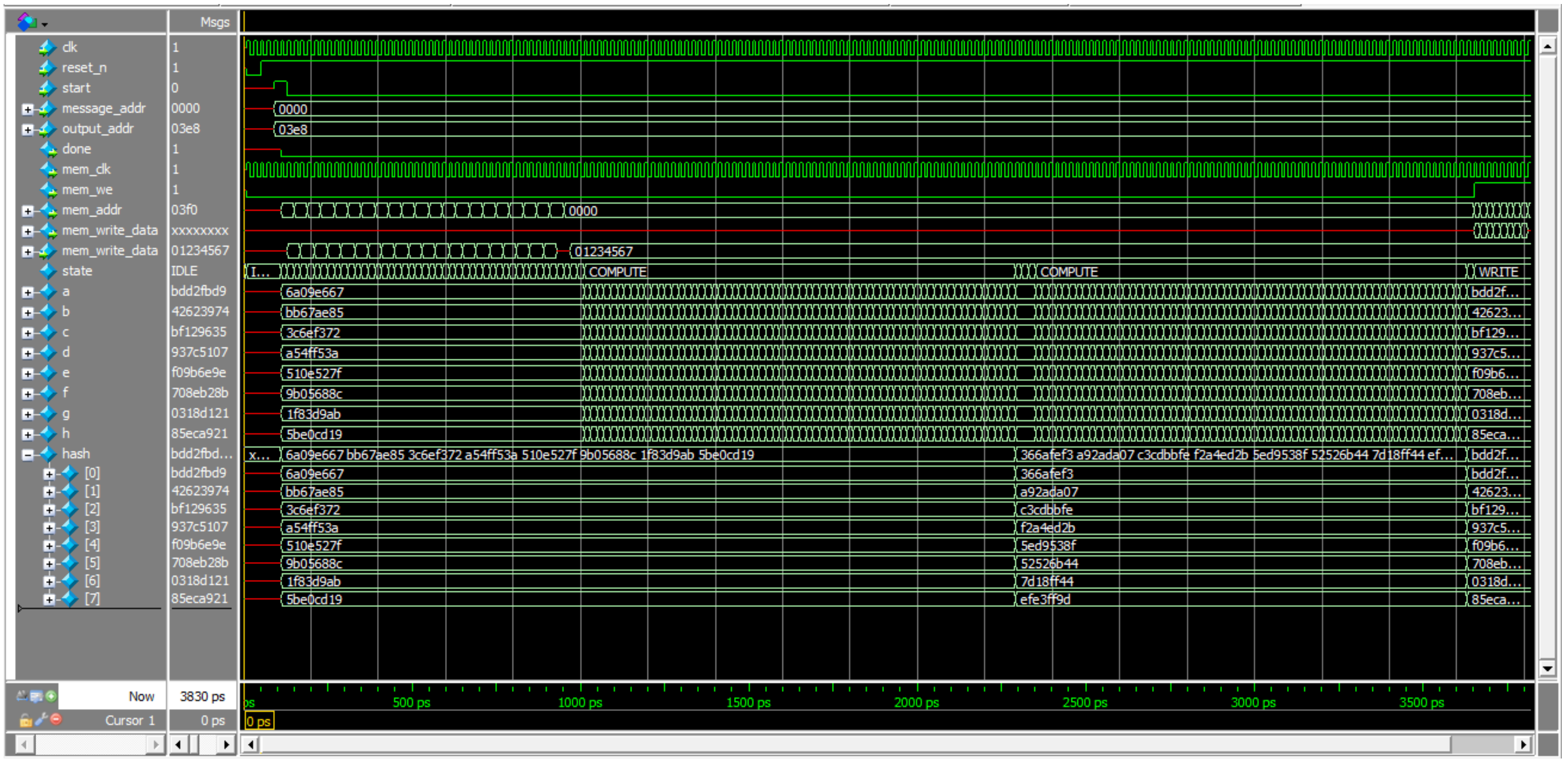
Analysis & Synthesis Resource Usage Summary	
Resource	**Usage**
Estimated ALUTs Used	1752
-- Combinational ALUTs	1752
-- Memory ALUTs	0
-- LUT_REGs	0
Dedicated logic registers	1672
Estimated ALUTs Unavailable	0
-- Due to unpartnered combinational logic	0
-- Due to Memory ALUTs	0
Total combinational functions	1752
Combinational ALUT usage by number of inputs	
-- 7 input functions	0
-- 6 input functions	193
-- 5 input functions	11
-- 4 input functions	56
-- <=3 input functions	1492
Combinational ALUTs by mode	
-- normal mode	1149
-- extended LUT mode	0
-- arithmetic mode	475
-- shared arithmetic mode	128
Estimated ALUT/register pairs used	2057
Total registers	1672
-- Dedicated logic registers	1672
-- I/O registers	0
-- LUT_REGs	0
I/O pins	118
DSP block 18-bit elements	0
Maximum fan-out node	clk~input
Maximum fan-out	1673
Total fan-out	13288
Average fan-out	3.63

Fmax:

Slow 900mV 100C Model Fmax Summary

Fmax	Restricted Fmax	Clock Name	Note
185.63 MHz	185.63 MHz	clk	

Modelsim Waveform:



Description of the Bitcoin-Hash Processor Final Project

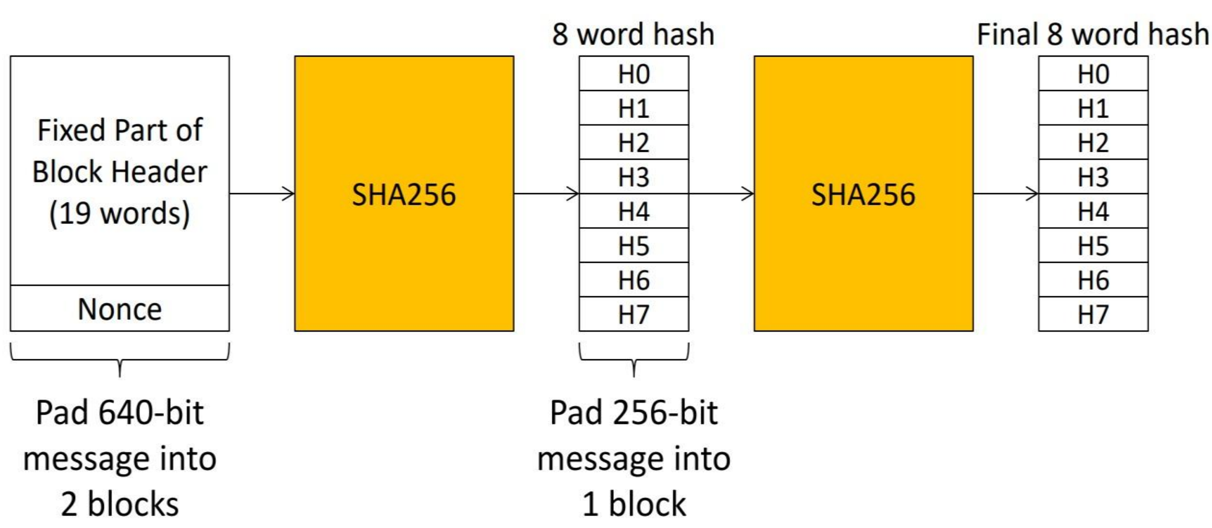
The final hash is calculated for 16 nonces by changing the input message for each nonces. Bitcoin hashing is performed in 3 phases:

1. Processing the 1st block of the 1st SHA 256 hash function in which W_{ts} are first 16 words in the memory and H0, H1, H2… are constants.

2. Processing the 2nd block of the 1st SHA 256 hash function where H0…H7 comes from the 1st block and W_{ts} corresponds to the last 3 words in the memory, the nonce, padding.

3. Processing the 2nd SHA 256 hash function where H0…H7 corresponds to the constants and W_{ts} are H0…H7 from the 2nd phase with the padding.

The above three steps are performed 16 times. This will produce 16 finals hashes. Final H0[0], H1[0], …H7[0] are stored back to the memory. For 16 nonces the above processing can be performed sequentially or in parallel manner. The following figure depicts the 3 phases of Bitcoin-Hash Processor.



Transcript:

```
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:         221
#
#
# ***************************
#
```

Fmax:

Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	138.99 MHz	138.99 MHz	clk	

Resource Usage Summary:

Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated ALUTs Used	19207
1	-- Combinational ALUTs	19207
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	17512
3		
4	Estimated ALUTs Unavailable	1
1	-- Due to unpartnered combinational logic	1
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	19207
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	1
2	-- 6 input functions	1822
3	-- 5 input functions	1707
4	-- 4 input functions	80
5	-- <=3 input functions	15597
8		
9	Combinational ALUTs by mode	
1	-- normal mode	10925
2	-- extended LUT mode	1
3	-- arithmetic mode	6745
4	-- shared arithmetic mode	1536
10		
11	Estimated ALUT/register pairs used	23496
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	131985
23	Average fan-out	3.57

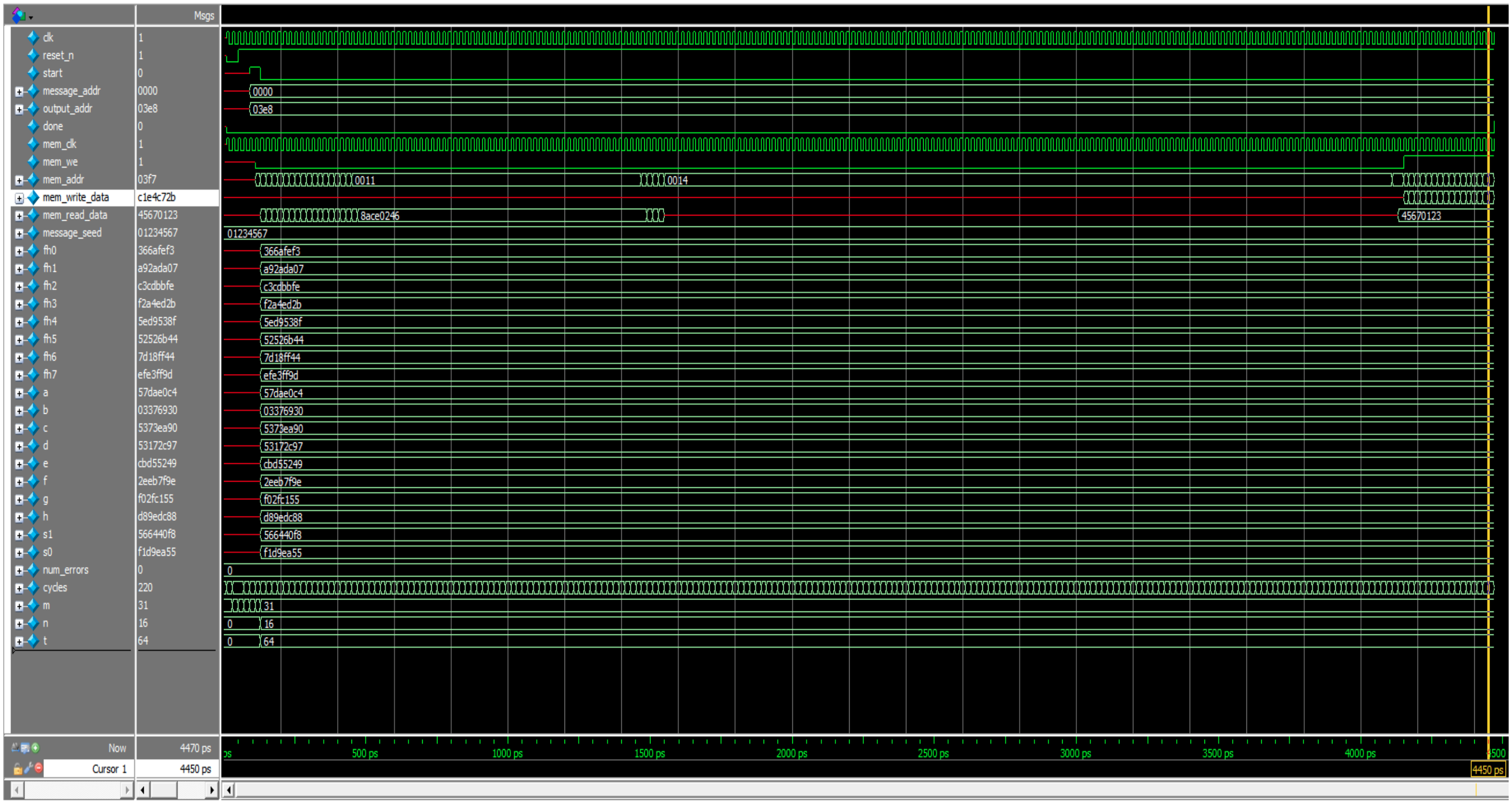
Fitter Report:

Fitter Summary

<<Filter>>

Fitter Status	Successful - Thu Jun 04 06:41:36 2020
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	75 %
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 %)

Modelsim Waveform:



Design Details

The following approaches are used to optimize the Fmax of the design:

1. Precomputed 'h+k+w' part in the sha256_op () function a cycle before every time the sha256_op function is executed. This improves the delay in critical paths.

```
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, p);
    logic [31:0] A1, A0, ch, maj, t1, t2;
    begin
        A1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
        ch = (e & f) ^ ((~e) & g);
        t1 = A1 + ch + p;
        A0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
        maj = (a & b) ^ (a & c) ^ (b & c);
        t2 = A0 + maj;
        sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
    end
endfunction
```

Used as:

```
{a, b, c, d, e, f, g, h} <= sha256_op(a, b, c, d, e, f, g, p);
```

2. wtnew () function which uses W[0], W[1], W[9] and W[14] to reduce the memory requirement from W[64] to W[16]. 32 bit words W[0].......W[15] are shifted to the left so that corresponding W[0], W[1], W[9] and W[14] are used every time to find new W[15]. This avoids the usage of a 64:1 multiplexor and helps to eliminate w-expansion logic from the critical path.

```
function logic [31:0] wtnew(); // function with no inputs
    logic [31:0] s0, s1;
    s0 = rightrotate(w1[1], 7) ^ rightrotate(w1[1], 18) ^ (w1[1] >> 3);
    s1 = rightrotate(w1[14], 17) ^ rightrotate(w1[14], 19) ^ (w1[14] >> 10);
    wtnew = w1[0] + s0 + w1[9] + s1;
endfunction
```

```
w1[15] <= wtnew();
for (t=0; t<15; t++) w1[t] <= w1[t+1];
```

3. The result of wtnew () function is directly stored in w [15] and this value is shifted left in each cycle.
4. Aggressive pipeline technique is implemented so that no state is the state machine has a serial execution in it. It ensures that all the executions in each state are parallel.
5. Used module instantiation to create multiple instances of SHA256 unit.

```
genvar q;
    generate
        for (q=0; q<NUM_NONCES; q++) begin : generate_sha256_modules
            sha256 sha256_inst (
                .clk(clk),
                .reset_n(reset_n),
                .state(state),
                .start(start),
                ._nonce(nonce[q]),
                .mem_read_data(mem_read_data),
                .hh(hash[q]));
        end
endgenerate
```

Summary of Results

1. Design for best delay - for SHA Design

simplified_sha256.sv (MIN DELAY DESIGN)							
Compiler Settings	#ALUTs	#Registers	Area	Fmax (MHz)	#Cycles	Delay (microsec)	Area*Delay (millisec*area)
Performance (Balanced Effort)	1752	1672	3424	185.63	189	1.018	3.486
Performance (Balanced Effort)	1752	1672	3424	185.63	189	1.018	3.486
Performance (Balanced Effort)	1752	1672	3424	185.63	189	1.018	3.486

2. Design for best delay - for BitCoin Design

bitcoin_hash.sv (MIN DELAY DESIGN)							
Compiler Settings	#ALUTs	#Registers	Area	Fmax (MHz)	#Cycles	Delay (microsec)	Area*Delay (millisec*area)
Performance (Balanced Effort)	19207	17512	36719	138.99	221	1.590	58.385
Performance (Balanced Effort)	19207	17512	36719	138.99	221	1.590	58.385
Performance (Balanced Effort)	19207	17512	36719	138.99	221	1.590	58.385

Github repo:

https://github.com/richapallavi0627/ECE111_BitcoinHash

References

- https://en.wikipedia.org/wiki/SHA-2
- Lectures from ECE 111 Class (piazza & canvas)
- https://www.bitcoinmarketjournal.com/blockchain-technology/
- https://www.nasdaq.com/article/10-ways-cryptocurrency-will-make-the-world-a-betterplace-cm905663