ECE 111 SP24: SHA 256 and Bitcoin Hashing Model

1. SHA 256 and Bitcoin hashing

1) SHA 256

SHA-256, which stands for Secure Hash Algorithm 256-bit, is a cryptographic hash function that converts a variable length input into a fixed size 256-bit hash value. The maximum length of the input be 2^64 bits. It has 6 properties:

i. Compression: SHA 256 will produce a fixed length hash value regardless the length of the input.
ii. Avalanche Effect: Even a small change in the input will produce a drastic change in the output.
iii. Determinism: The same input will always produce the same hash output.
iv. One-Way function: It is computationally infeasible to reverse the process to obtain the original input from the hash output.
v. Collision Resistance: It is practically impossible to find two different inputs that produce the same output.
vi. Efficient: Creating the output hash should be a fast process that doesn't make heavy use of computing power.

2) Blockchain and Bitcoin Hashing

A blockchain is a distributed database or ledger shared among a network's nodes. It's widely used in cryptocurrency systems like Bitcoin, where it maintains a secure and decentralized record of transactions. The data is stored in blocks, and each block is cryptographically linked to the previous one, forming a chain. Chaining of blocks is done through cryptographic hashing algorithms, such as SHA-256. Each block contains the hash value, or signature of itself and its previous block. Since the signatures are generated by secure algorithm like SHA-256, it is impossible to modify the transaction data in one block because it would result in a mismatch between this block and next block's signature.
In order for the signature to be accepted in the block chain, a small piece of data called nonce need to be added to the block. Bitcoin miners repeatedly change this nonce until a specific nonce leads to a signature that is accepted by the block chain. The process of finding that specific nonce is called mining.

2. SHA-256 and Bitcoin hashing implemented in the project

The SHA 256 hash function is implemented by a FSM, which has 5 states:

i. IDLE: Initializing hash values and wait for the start signal. When the start signal is high, the FSM goes to READ state.

ii. READ: Read one word from the memory in one cycle. If all words are read, go to BLOCK state, otherwise go to REDA state.

iii. BLOCK: Register 16 words (512 bits) from the input message into a block, then go to COMPUTE state. If the remaining number of words is less than 16, perform padding. Suppose the remaining number of bits are L, an additional ‘1’ bit is appended, followed by K ‘0’ bits, where K is the minimum positive integer such that (L + 1 + K + 64) is a multiple of 512. Then a 64-bit big-endian binary number that equals the length of the input message is appended, which makes the last block of length 512 bits. These 512 bits are registered into a block, and the FSM goes to COMPUTE state. If all words are processed, go to WRITE state.

iv. COMPUTE: This stage takes a block as input, and generates 8 hash values as outputs. It can be divided into 3 substages:

a) Word expansion: First we create a 64-entry message schedule array w[0..63] of 32-bit words out of the block. The first 16 entries, w[0:15], are directly copied from the block. And for w[16:63], they are given by the following process:

```
Extend the first 16 words into the remaining 48 words w[16..63] of the message schedule array:
for i from 16 to 63
    s0  := (w[i-15] rightrotate  7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift  3)
    s1  := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightshift 10)
    w[i]  := w[i-16] + s0 + w[i-7] + s1
```

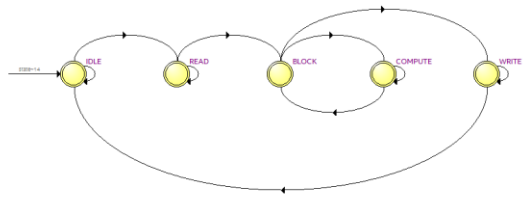
b) SHA 256 operation: This step takes the message schedule array as its input, and produces 8 words, {a,b,c,d,e,f,g,h} as outputs. The process is described as:

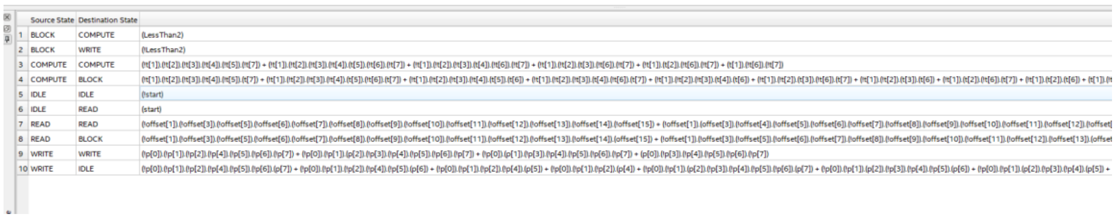
```
Compression function main loop:
for i from 0 to 63
    S1  := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
    ch  := (e and f) xor ((not e) and g)
    temp1  := h + S1 + ch + k[i] + w[i]
    S0  := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
    maj  := (a and b) xor (a and c) xor (b and c)
    temp2  := S0 + maj

    h  := g
    g  := f
    f  := e
    e  := d + temp1
    d  := c
    c  := b
    b  := a
    a  := temp1 + temp2
```

c) Add a to h to the hash value h0 to h7. Then go back to BLOCK stage.

v. WRITE: Write h0 to h7 sequentially to the memory. When all hash values are written into memory, go back to IDLE state and assert the done signal.

The state transition diagram is shown in figure 1.





	Source State	Destination State	
1	BLOCK	COMPUTE	(LessThan2)
2	BLOCK	WRITE	(!LessThan2)
3	COMPUTE	COMPUTE	(!t[1]).(!t[2]).(!t[3]).(!t[4]).(!t[5]).(!t[7]) + (!t[1]).(!t[2]).(!t[3]).(!t[4]).(t[5]).(!t[6]).(!t[7]) + (!t[1]).(!t[2]).(!t[3]).(t[4]).(!t[6]).(!t[7]) + (!t[1]).(!t[2]).(t[3]).(!t[6]).(!t[7]) + (!t[1]).(t[2]).(!t[6]).(!t[7]) + (t[1]).(!t[6]).(!t[7])
4	COMPUTE	BLOCK	(!t[1]).(!t[2]).(!t[3]).(!t[4]).(!t[5]).(t[7]) + (!t[1]).(!t[2]).(!t[3]).(!t[4]).(t[5]).(!t[6]).(t[7]) + (!t[1]).(!t[2]).(!t[3]).(!t[4]).(t[5]).(t[6]) + (!t[1]).(!t[2]).(!t[3]).(t[4]).(!t[6]).(t[7]) + (!t[1]).(!t[2]).(!t[3]).(t[4]).(t[6]) + (!t[1]).(!t[2]).(t[3]).(!t[6]).(t[7]) + (!t[1]).(!t[2]).(t[3]).(t[6]) + (!t[1]).(t[2]).(!t[6]).(t[7]) + (!t[1]).(t[2]).(t[6]) + (t[1]).(!t
5	IDLE	IDLE	(!start)
6	IDLE	READ	(start)
7	READ	READ	(!offset[1]).(!offset[3]).(!offset[5]).(!offset[6]).(!offset[7]).(!offset[8]).(!offset[9]).(!offset[10]).(!offset[11]).(!offset[12]).(!offset[13]).(!offset[14]).(!offset[15]) + (!offset[1]).(offset[3]).(!offset[4]).(!offset[5]).(!offset[6]).(!offset[7]).(!offset[8]).(!offset[9]).(!offset[10]).(!offset[11]).(!offset[12]).(!offset
8	READ	BLOCK	(!offset[1]).(!offset[3]).(!offset[5]).(!offset[6]).(!offset[7]).(!offset[8]).(!offset[9]).(!offset[10]).(!offset[11]).(!offset[12]).(!offset[13]).(!offset[14]).(offset[15]) + (!offset[1]).(!offset[3]).(!offset[5]).(!offset[6]).(!offset[7]).(!offset[8]).(!offset[9]).(!offset[10]).(!offset[11]).(!offset[12]).(!offset[13]).(offset
9	WRITE	WRITE	(!p[0]).(!p[1]).(!p[2]).(!p[4]).(!p[5]).(!p[6]).(!p[7]) + (!p[0]).(!p[1]).(p[2]).(!p[3]).(!p[4]).(!p[5]).(!p[6]).(!p[7]) + (!p[0]).(p[1]).(!p[3]).(!p[4]).(!p[5]).(!p[6]).(!p[7]) + (p[0]).(!p[3]).(!p[4]).(!p[5]).(!p[6]).(!p[7])
10	WRITE	IDLE	(!p[0]).(!p[1]).(!p[2]).(!p[4]).(!p[5]).(!p[6]).(p[7]) + (!p[0]).(!p[1]).(!p[2]).(!p[4]).(!p[5]).(p[6]) + (!p[0]).(!p[1]).(!p[2]).(!p[4]).(p[5]) + (!p[0]).(!p[1]).(!p[2]).(p[4]) + (!p[0]).(!p[1]).(p[2]).(!p[3]).(!p[4]).(!p[5]).(!p[6]).(p[7]) + (!p[0]).(!p[1]).(p[2]).(!p[3]).(!p[4]).(!p[5]).(p[6]) + (!p[0]).(!p[1]).(p[2]).(!p[3]).(!p[4]).(p[5]) +

Figure 1: state transition diagram of SHA 256 FSM

Optimization of SHA 256:

First notice that each round of SHA 256 operation only depends the latest message schedule, so the word expansion and SHA 256 operation can be fused into one loop. Every cycle we perform the SHA 256 operation, then compute the next message schedule. By doing so the cycle number of computing a block can be greatly reduced.

Also notice that for w[16] to w[63], they are only dependent on the most recent 16 w values. Therefore, it is redundant to store all 64 w values in an array. A better approach is only storing the most recent 16 w values in an array, and left shift the entire array whenever a new w comes to keep the last element in the array the latest w value. This way we can get rid of the large array and complex logic of choosing a w value.

Another optimization is pipelining the critical path. It can be easily shown that the most critical path in the design is the logic to get A_{t+1} in the SHA 256 operation, which can be expressed as:

$$A_{t+1} = \sum\nolimits_0 (A_t) + Maj(A_t, B_t, C_t) + \sum\nolimits_1 (E_t) + Ch(E_t, F_t, G_t) + K_t + W_t + H_t$$

Notice that K_t and W_t are independent of A to H, so $K_t + W_t$ can be precomputed in the previous clock cycle. Also notice that $H_t = G_{t-1}$, so we can compute $K_t + W_t + G_{t-1}$ in the previous clock cycle, and forward it to the SHA 256 operation in the next cycle, so that the critical path is shortened.

Bitcoin hashing implementation:

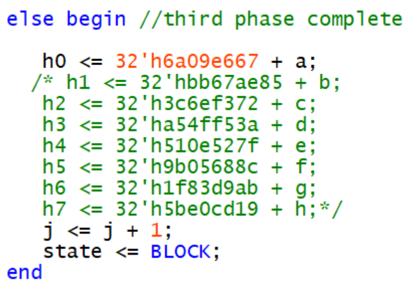
Input message to the bitcoin hashing model in this project contains 20 words. The first 16 words will go through a SHA 256 operation, and the 8 output hash values will be the initial hash value of the next phase. The input to the next phase are the last 4 words plus padding, where the last word is the nonce. The output hash values of this phase are the input to the next phase, and the we use the original hash constants as the initial hash values of the next phase. Finally we get the 8 hash values for this nonce, from which we only pick first one and store it in the memory. Phase 2 and phase 3 will be repeat 16 times for nonce 0 to 15, and there will be 16 hash values stored in the memory in the end.

Optimization of Bitcoin hashing:

The above process can be parallelized to get higher performance. We first perform phase 1, and then send the hash values to 16 instances of SHA 256 modules, each with a different nonce. Each instance will perform phase 2 and 3, and upon completion, it will present the hash value on the output and assert the valid signal. When all 16 instances are done, we sequentially write the 16 hash values to the memory. By doing so, we can greatly the cycle count and get a better performance than the serial implementation.

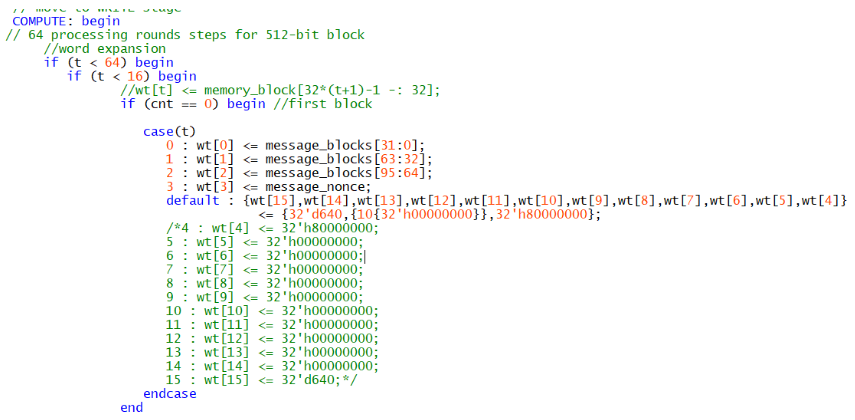
However, even with the above optimizations on SHA 256 modules, it is not possible to fit 16 instances of it into the designated FPGA device, so additional optimizations on area are required.

First we notice that we only write 16 h0 of each nonce to memory, so we don't need to compute h1 through h7 in the third phase. Reducing the computation logic can moderately reduce the ALUTs usage.



```
else begin //third phase complete

   h0 <= 32'h6a09e667 + a;
  /* h1 <= 32'hbb67ae85 + b;
   h2 <= 32'h3c6ef372 + c;
   h3 <= 32'ha54ff53a + d;
   h4 <= 32'h510e527f + e;
   h5 <= 32'h9b05688c + f;
   h6 <= 32'h1f83d9ab + g;
   h7 <= 32'h5be0cd19 + h;*/
   j <= j + 1;
   state <= BLOCK;
end
```

Another thing to notice is that the memory_block[511:0] variable in each SHA 256 instance will consume a lot of area because it will be synthesized into 512 registers. It would be very advantageous to area if we can get rid of it. Recall that in the previous SHA 256 logic, the memory_block variable is assigned in the BLOCK stage, and is copied to message schedules in the COMPUTE stage. Also recall that th re are only 4 input messages to each SHA 256 instance (because we are in phase 2), so it is possible to directly assign input messages to message schedules in the COMPUTE stage, and not use memory_block to register the input messages at all. One subtle thing is that we are copying one input message to one message schedule in a cycle, but this won't be a problem because only one message schedule is needed to compute the next {a,b,c,d,e,f,g,h} in each round.



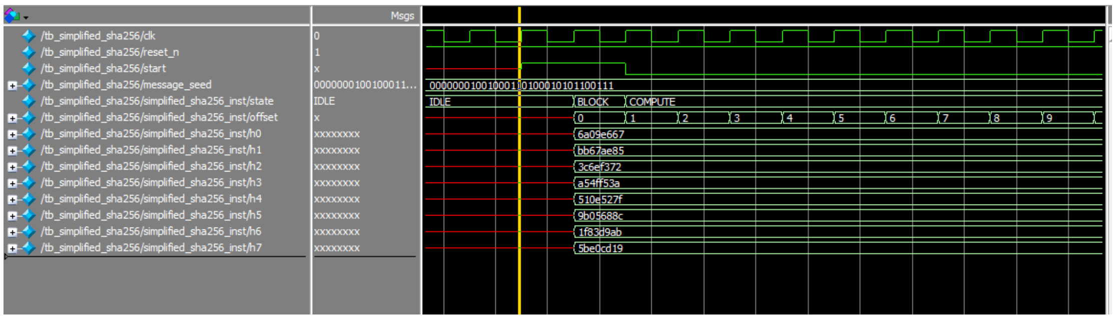
```
 COMPUTE: begin
// 64 processing rounds steps for 512-bit block
     //word expansion
     if (t < 64) begin
        if (t < 16) begin
              //wt[t] <= memory_block[32*(t+1)-1 -: 32];
              if (cnt == 0) begin //first block

                 case(t)
                    0 : wt[0] <= message_blocks[31:0];
                    1 : wt[1] <= message_blocks[63:32];
                    2 : wt[2] <= message_blocks[95:64];
                    3 : wt[3] <= message_nonce;
                    default : {wt[15],wt[14],wt[13],wt[12],wt[11],wt[10],wt[9],wt[8],wt[7],wt[6],wt[5],wt[4]}
                           <= {32'd640,{10{32'h00000000}},32'h80000000};
                    /*4 : wt[4] <= 32'h80000000;
                    5 : wt[5] <= 32'h00000000;
                    6 : wt[6] <= 32'h00000000;
                    7 : wt[7] <= 32'h00000000;
                    8 : wt[8] <= 32'h00000000;
                    9 : wt[9] <= 32'h00000000;
                    10 : wt[10] <= 32'h00000000;
                    11 : wt[11] <= 32'h00000000;
                    12 : wt[12] <= 32'h00000000;
                    13 : wt[13] <= 32'h00000000;
                    14 : wt[14] <= 32'h00000000;
                    15 : wt[15] <= 32'd640;*/
                 endcase
              end
```

As shown in the figure, we can directly copy the 4 input messages to message schedules in the COMPUTE stage and get rid of memory_block. Since most of bits in the block are padding, we don't need a large 16:1 mux, which is another benefit for area.

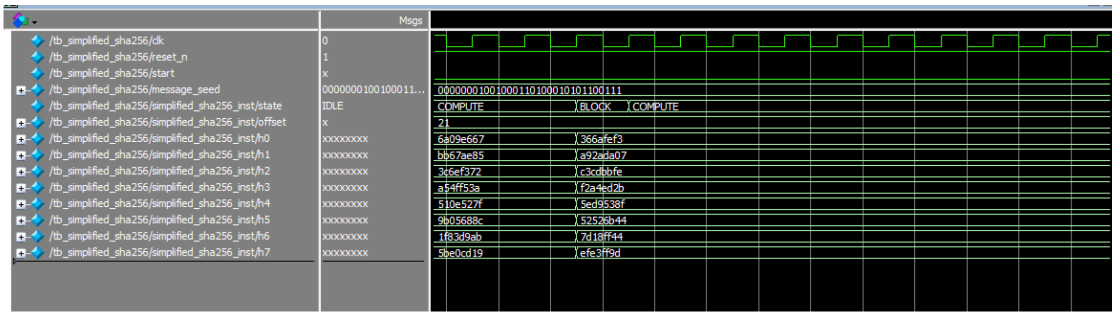
With these additional optimizations on area, we manage to fit 16 SHA 256 instances into the FPGA. And we can further reduce the cycle count by doing read on the fly. Notice that memory read can

be performed parallelly with later stages because we only need the lates input message in a cycle. By doing so we effectively remove the READ stage, and reduce the cycle count by 20.
The “read on the fly” optimization is propagated to the SHA 256 model to eliminate the READ state. Therefore, in the simulation waveform there is no READ state.

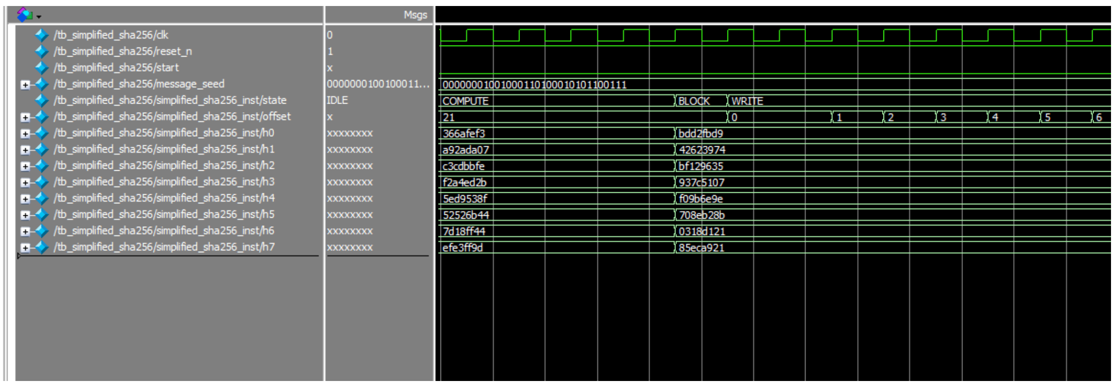
3. Simulation waveform for SHA-256 and Bitcoin hashing


Msgs
/tb_simplified_sha256/clk
/tb_simplified_sha256/reset_n
/tb_simplified_sha256/start
/tb_simplified_sha256/message_seed
/tb_simplified_sha256/simplified_sha256_inst/state
/tb_simplified_sha256/simplified_sha256_inst/offset
/tb_simplified_sha256/simplified_sha256_inst/h0
/tb_simplified_sha256/simplified_sha256_inst/h1
/tb_simplified_sha256/simplified_sha256_inst/h2
/tb_simplified_sha256/simplified_sha256_inst/h3
/tb_simplified_sha256/simplified_sha256_inst/h4
/tb_simplified_sha256/simplified_sha256_inst/h5
/tb_simplified_sha256/simplified_sha256_inst/h6
/tb_simplified_sha256/simplified_sha256_inst/h7
0
1
x
000000010010011...
IDLE
x
xxxxxxxx
00000001001000110100010101100111
IDLE
BLOCK
COMPUTE
0 1 2 3 4 5 6 7 8 9
6a09e667
bb67ae85
3c6ef372
a54ff53a
510e527f
9b05688c
1f83d9ab
5be0cd19

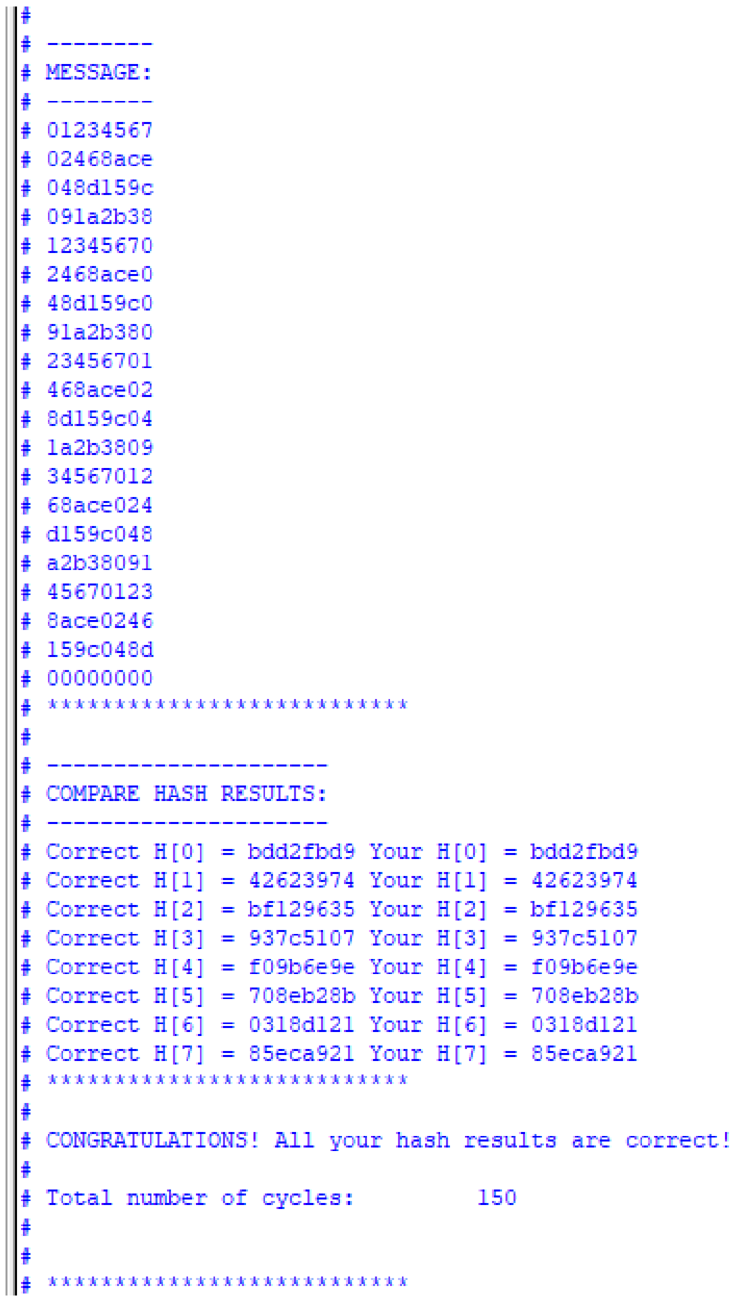
SHA 256 waveform 1


Msgs
/tb_simplified_sha256/clk
/tb_simplified_sha256/reset_n
/tb_simplified_sha256/start
/tb_simplified_sha256/message_seed
/tb_simplified_sha256/simplified_sha256_inst/state
/tb_simplified_sha256/simplified_sha256_inst/offset
/tb_simplified_sha256/simplified_sha256_inst/h0
/tb_simplified_sha256/simplified_sha256_inst/h1
/tb_simplified_sha256/simplified_sha256_inst/h2
/tb_simplified_sha256/simplified_sha256_inst/h3
/tb_simplified_sha256/simplified_sha256_inst/h4
/tb_simplified_sha256/simplified_sha256_inst/h5
/tb_simplified_sha256/simplified_sha256_inst/h6
/tb_simplified_sha256/simplified_sha256_inst/h7
0
1
x
000000010010011...
IDLE
x
xxxxxxxx
00000001001000110100010101100111
COMPUTE
BLOCK
COMPUTE
21
6a09e667 366afef3
bb67ae85 a92ada07
3c6ef372 c3cdbbfe
a54ff53a f2a4ed2b
510e527f 5ed9538f
9b05688c 52526b44
1f83d9ab 7d18ff44
5be0cd19 efe3ff9d

SHA 256 waveform 2

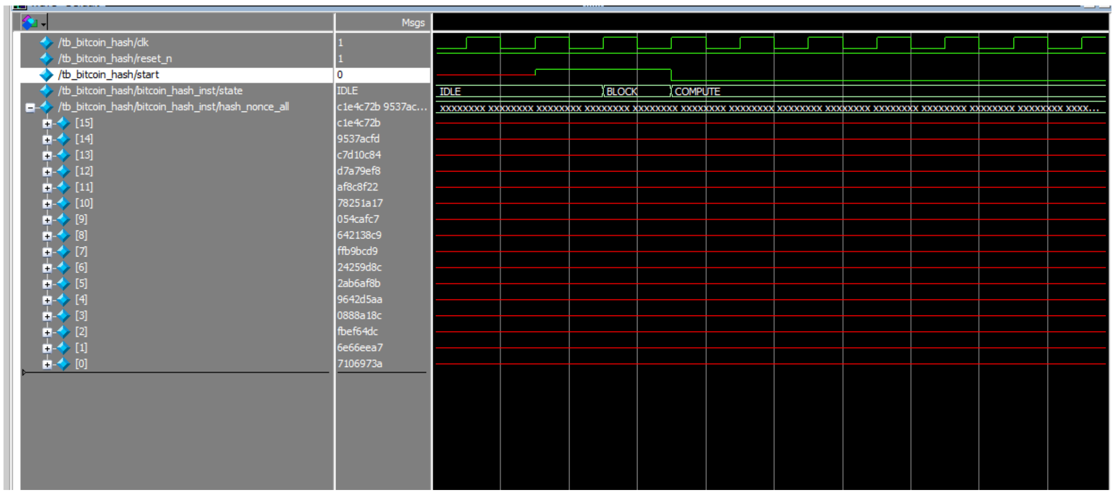

Msgs
/tb_simplified_sha256/clk
/tb_simplified_sha256/reset_n
/tb_simplified_sha256/start
/tb_simplified_sha256/message_seed
/tb_simplified_sha256/simplified_sha256_inst/state
/tb_simplified_sha256/simplified_sha256_inst/offset
/tb_simplified_sha256/simplified_sha256_inst/h0
/tb_simplified_sha256/simplified_sha256_inst/h1
/tb_simplified_sha256/simplified_sha256_inst/h2
/tb_simplified_sha256/simplified_sha256_inst/h3
/tb_simplified_sha256/simplified_sha256_inst/h4
/tb_simplified_sha256/simplified_sha256_inst/h5
/tb_simplified_sha256/simplified_sha256_inst/h6
/tb_simplified_sha256/simplified_sha256_inst/h7
0
1
x
000000010010011...
IDLE
x
xxxxxxxx
00000001001000110100010101100111
COMPUTE
BLOCK
WRITE
21 0 1 2 3 4 5 6
366afef3 bdd2fbd9
a92ada07 42623974
c3cdbbfe bf129635
f2a4ed2b 937c5107
5ed9538f f09b6e9e
52526b44 708eb28b
7d18ff44 0318d121
efe3ff9d 85eca921

SHA 256 waveform 3



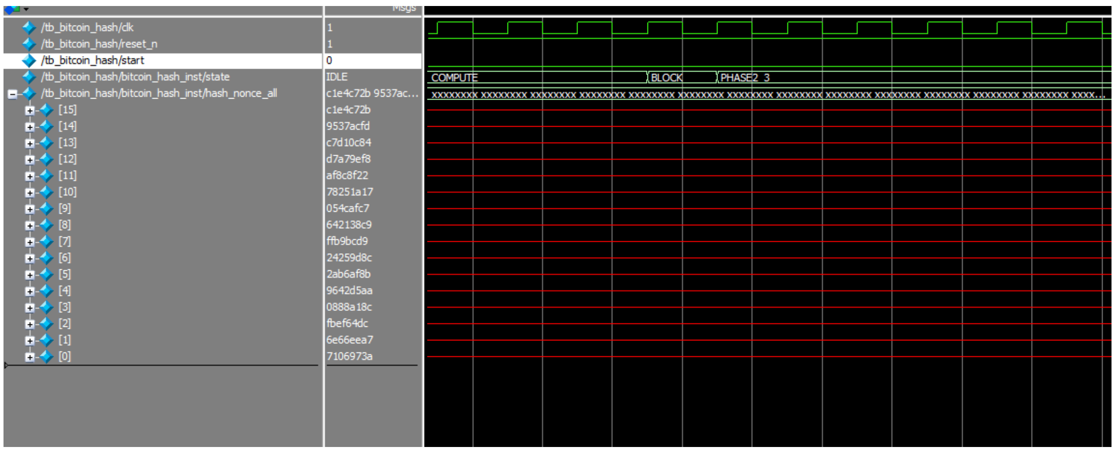
```
#
# --------
# MESSAGE:
# --------
# 01234567
# 02468ace
# 048d159c
# 091a2b38
# 12345670
# 2468ace0
# 48d159c0
# 91a2b380
# 23456701
# 468ace02
# 8d159c04
# 1a2b3809
# 34567012
# 68ace024
# d159c048
# a2b38091
# 45670123
# 8ace0246
# 159c048d
# 00000000
# ***************************
#
# ---------------------
# 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:         150
#
#
# ***************************
```

SHA 256 simulation transcript



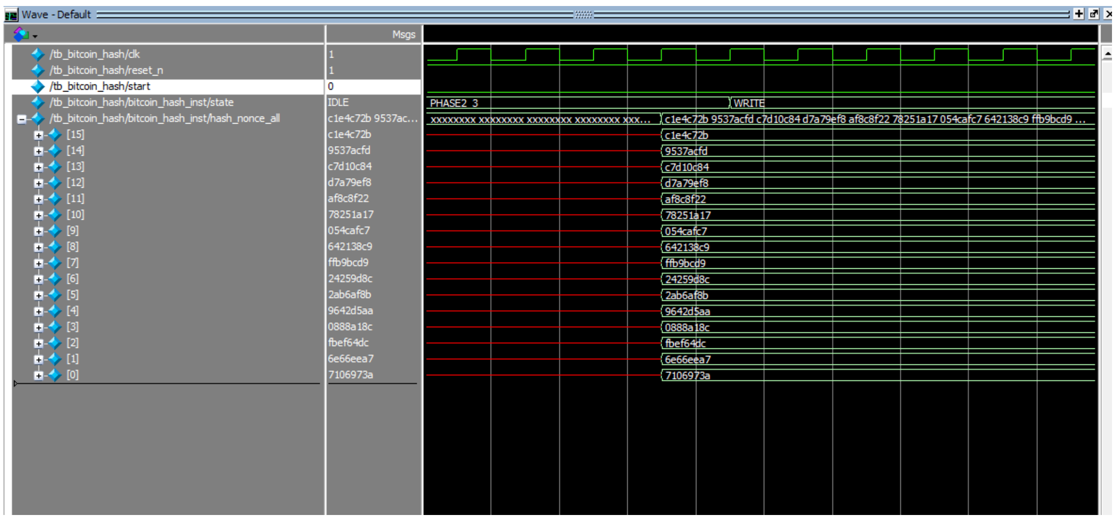
```
/tb_bitcoin_hash/clk                                1
/tb_bitcoin_hash/reset_n                            1
/tb_bitcoin_hash/start                              0
/tb_bitcoin_hash/bitcoin_hash_inst/state            IDLE
/tb_bitcoin_hash/bitcoin_hash_inst/hash_nonce_all   c1e4c72b 9537ac...
  [15]   c1e4c72b
  [14]   9537acfd
  [13]   c7d10c84
  [12]   d7a79ef8
  [11]   af8c8f22
  [10]   78251a17
  [9]    054cafc7
  [8]    642138c9
  [7]    ffb9bcd9
  [6]    24259d8c
  [5]    2ab6af8b
  [4]    9642d5aa
  [3]    0888a18c
  [2]    fbef64dc
  [1]    6e66eea7
  [0]    7106973a
```

Bitcoin hash waveform 1



```
/tb_bitcoin_hash/clk                                1
/tb_bitcoin_hash/reset_n                            1
/tb_bitcoin_hash/start                              0
/tb_bitcoin_hash/bitcoin_hash_inst/state            IDLE
/tb_bitcoin_hash/bitcoin_hash_inst/hash_nonce_all   c1e4c72b 9537ac...
  [15]   c1e4c72b
  [14]   9537acfd
  [13]   c7d10c84
  [12]   d7a79ef8
  [11]   af8c8f22
  [10]   78251a17
  [9]    054cafc7
  [8]    642138c9
  [7]    ffb9bcd9
  [6]    24259d8c
  [5]    2ab6af8b
  [4]    9642d5aa
  [3]    0888a18c
  [2]    fbef64dc
  [1]    6e66eea7
  [0]    7106973a
```

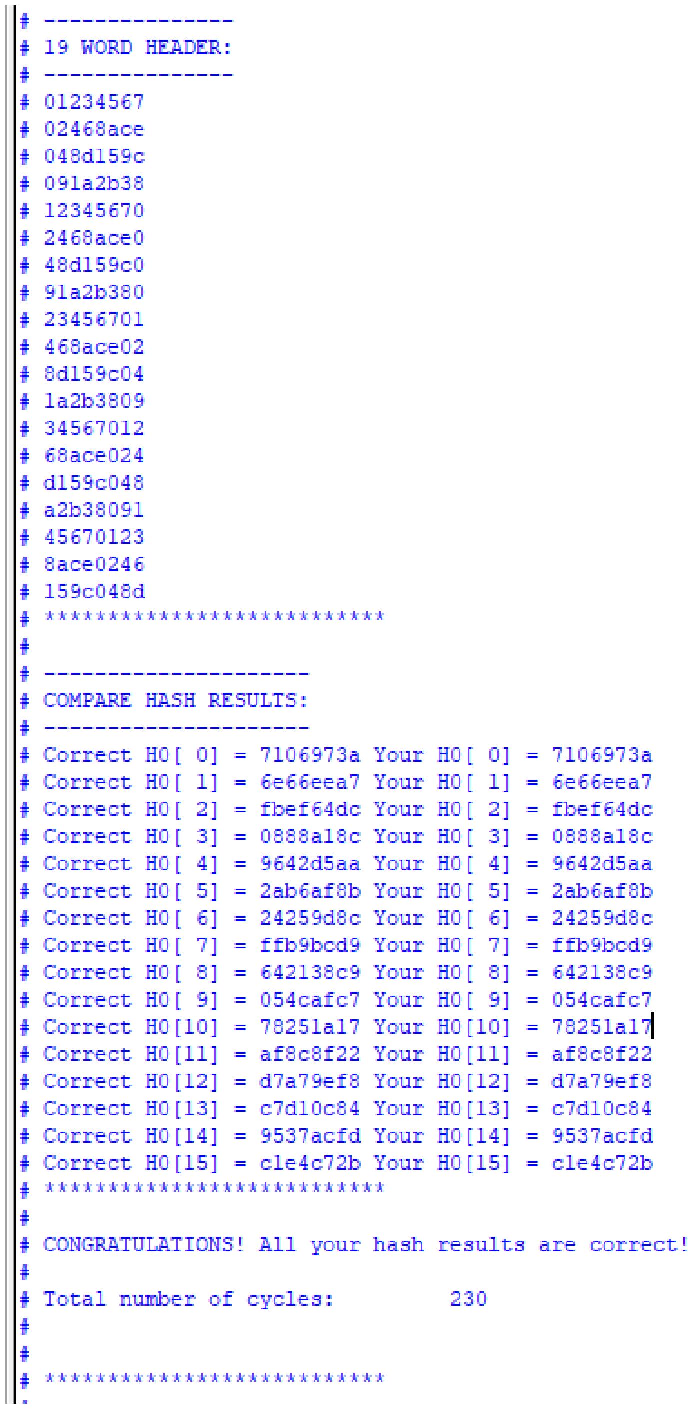
Bitcoin hash waveform 2



```
/tb_bitcoin_hash/clk                                1
/tb_bitcoin_hash/reset_n                            1
/tb_bitcoin_hash/start                              0
/tb_bitcoin_hash/bitcoin_hash_inst/state            IDLE
/tb_bitcoin_hash/bitcoin_hash_inst/hash_nonce_all   c1e4c72b 9537ac...
  [15]   c1e4c72b
  [14]   9537acfd
  [13]   c7d10c84
  [12]   d7a79ef8
  [11]   af8c8f22
  [10]   78251a17
  [9]    054cafc7
  [8]    642138c9
  [7]    ffb9bcd9
  [6]    24259d8c
  [5]    2ab6af8b
  [4]    9642d5aa
  [3]    0888a18c
  [2]    fbef64dc
  [1]    6e66eea7
  [0]    7106973a
```

```
c1e4c72b 9537acfd c7d10c84 d7a79ef8 af8c8f22 78251a17 054cafc7 642138c9 ffb9bcd9 ...
```

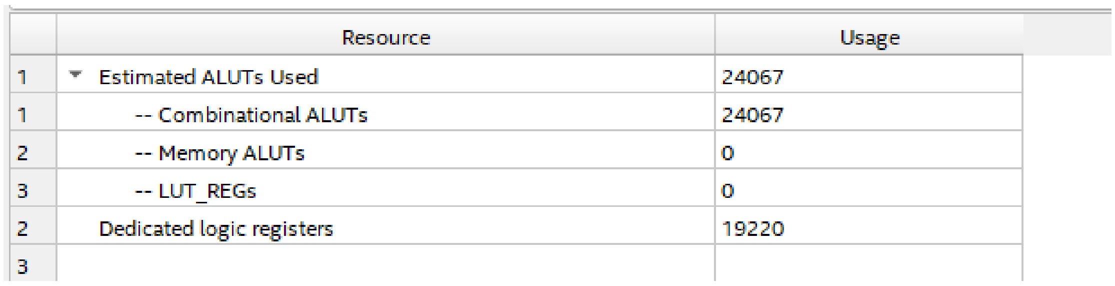
Bitcoin hash waveform 3



```
# ---------------
# 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:         230
#
#
# ***************************
```

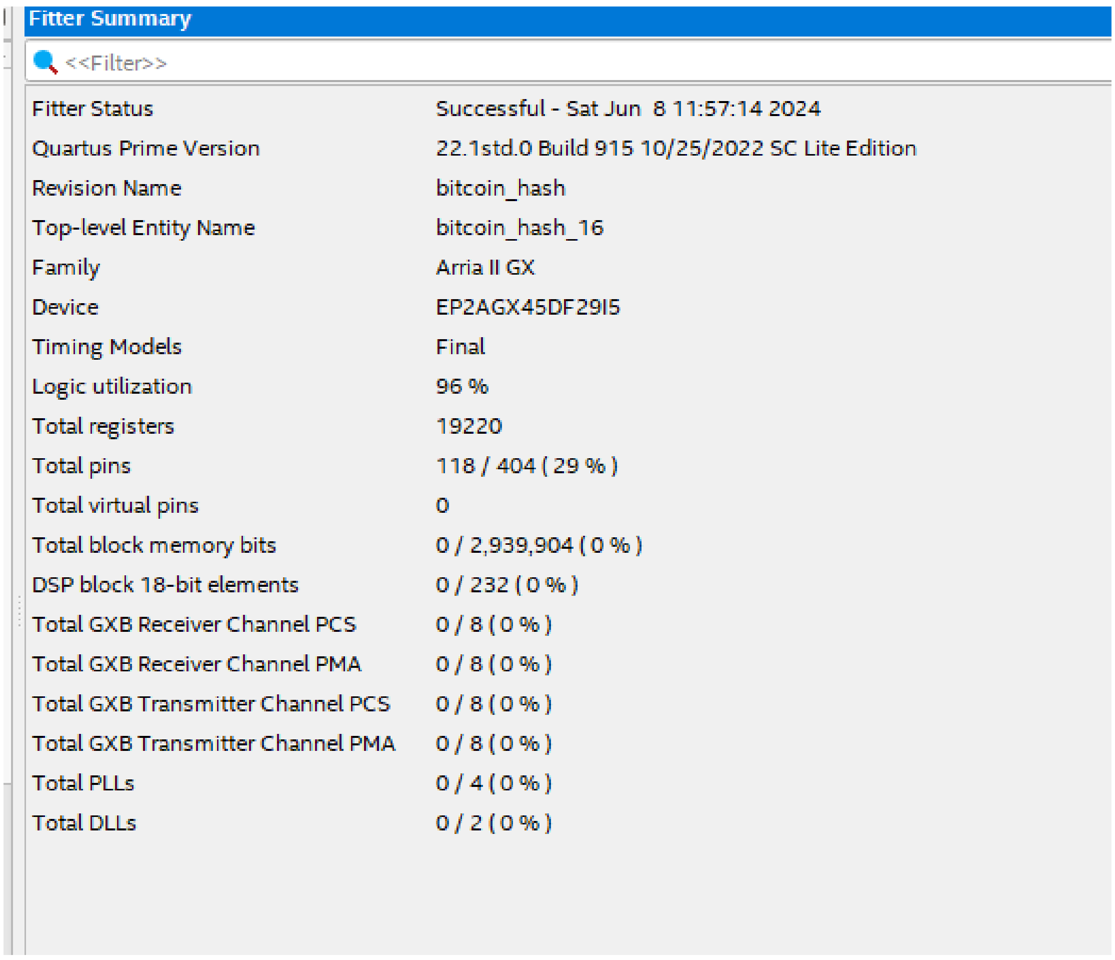
Figure 10: Bitcoin hash simulation transcript

4. Resource usage and timing report for bitcoin_hash



	Resource	Usage
1	Estimated ALUTs Used	24067
1	-- Combinational ALUTs	24067
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	19220
3		

Synthesis resource usage 1

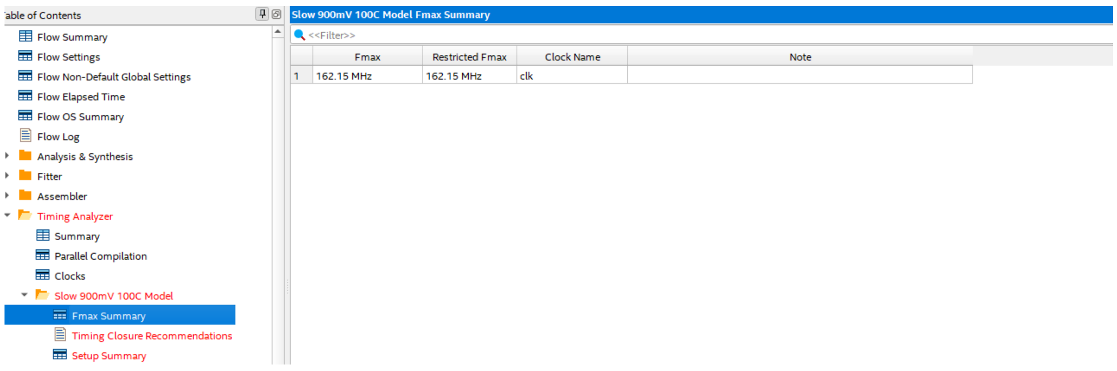


Fitter Summary

<<Filter>>

Fitter Status	Successful - Sat Jun 8 11:57:14 2024
Quartus Prime Version	22.1std.0 Build 915 10/25/2022 SC Lite Edition
Revision Name	bitcoin_hash
Top-level Entity Name	bitcoin_hash_16
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	96 %
Total registers	19220
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 %)

Fitter report



able of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- Timing Analyzer
  - Summary
  - Parallel Compilation
  - Clocks
  - Slow 900mV 100C Model
    - Fmax Summary
    - Timing Closure Recommendations
    - Setup Summary

Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	162.15 MHz	162.15 MHz	clk	

Fmax timing report