Final Project

ECE 111 (Advanced Digital Design Project)

Aryan Devrani, Krish Mehta, Kumar Divij

June 8th 2024

1. Introduction

1a. SHA-256

Secure Hash Algorithm 256-bit (SHA-256) is a cryptographic hash function part of the SHA-2 family. SHA-256 generates a unique 256-bit (32-byte) signature for a text. It's a one-way function that turns any input into a fixed-size hash value, which is nearly impossible to reverse. The properties that make SHA-256 secure - ① Deterministic: The same message always results in the same hash. ② Quick computation: The hash value is quick to compute for any given input. ③ Irreversibility: It's infeasible to generate the original input by knowing the hash value ④ Small changes to input alter the hash: Even a tiny change in the input drastically changes the output hash. ⑤ Collision-resistant: Two different inputs will not produce the same output hash.

1b. Use of SHA-256 in Bitcoin Mining

Bitcoin uses SHA-256 to provide secure transactions. Bitcoin mining involves discovering a number (nonce) that when combined with the block data results in a hash that meets certain conditions (e.g., a specific number of leading zeros). This process secures the network and validates transactions. SHA-256 is crucial in maintaining the integrity of the Bitcoin blockchain. It is used to generate cryptographic hashes that chain the blocks together, ensuring data within a block hasn't been altered once added to the blockchain.

2. Optimizations

We have performed several optimizations on top of the basic functionality.

1. **Reduction of w[64] to w[16] with shift register approach:** The array of w values is cut down to 16 entries, with a new value being pushed into w[15] every cycle. This allows forthcoming w[16:64] values to be produced at location w[15] every cycle, thus eliminating the need to instantiate 64 registers for each w.
   Although the naive optimization suggests the use of w[15] for processing compression rounds 16 to 63, it required a lot more combinational logic compared to using a Queue-based approach. In our implementation, we constantly read out w[0] for every processing round starting from i=0 to i=63, as the left-shift operation ensures that the required w[i] is located at w[0] at every i^{th} iteration. This further reduced the combinational LUTs required, and boosted the Fmax for our design.

2. **Vectorization of Core SHA256_op computations to reduce Bitcoin Hashing latency:** Instantiating 16 or 8 SHA modules leads to a lot of redundant hardware overhead, which is very inefficient and not really necessary. Instead, we simply vectorize the core blocking and computation portion of the SHA operation. This allows us to fit 16X SHA operations in every cycle onto the Aria II GX FPGA, and achieve an extremely low latency of 237 cycles.
   This is achieved by initializing inputs w[nonce][0:15] and local accumulators a[nonce], b[nonce],.....h[nonce] in a vectorized 2-D array fashion for Phases 2 & 3 of Bitcoin hashing using SHA256. This allows us to parallelly compute the last 2 phases for each nonce simultaneously, using a single FSM logic. This not only leads to resource-efficiency, but also better timing and Fmax due to simplification of the FSM logic design.

3. **Eliminating the need for separate hash storage registers per phase :** Instead of declaring additional registers to store the output hashes for each phase, we designed the FSM logic to reuse the local registers a[], b[], c[],

….h[] to store the output hash of each stage, which then acts as the input hash to the next stage. This eliminated the need to store output hash values for any stage except the original hash contents.

4. **FSM State Reduction Experiment:** We experimented with reducing the Bitcoin Hashing FSM states such that Phase 2 and Phase 3 computations would use the same state. However, in our experiments, this was leading to a significant increase in the number of Logic Array Blocks (LABs) required during Fitting, so we abandoned this approach.

3. Algorithm

3a. Algorithm for SHA-256

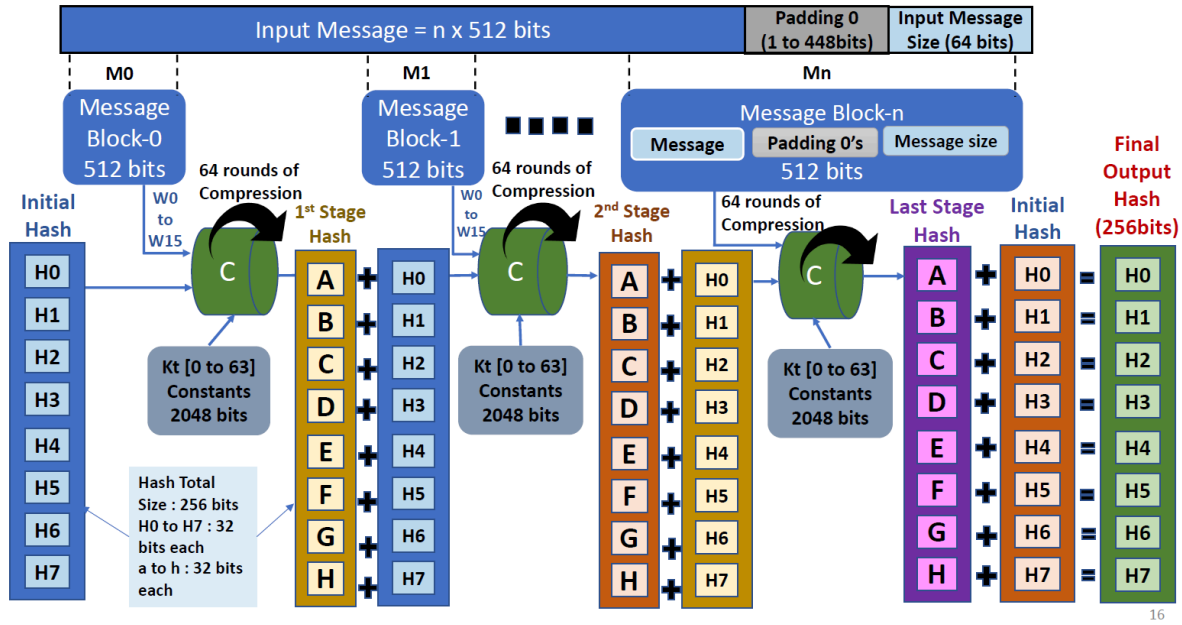

Figure 1 SHA256 Algorithm

1. **Input Message = n x 512 bits** – The entire input message is broken down into blocks, each consisting of 512 bits.
2. **Message Block-0 to Message Block-n** – Each block represents a segment of the total input message. The first block is labelled as M0 and subsequent blocks as M1, M2, ..., Mn.
3. **Initial Hash (H0 to H7)** – These are the initial hash values used at the start of the SHA-256 algorithm, derived from the square roots of the first eight primes. They serve as the initial inputs for the compression function.
4. **W0 to W15** – These are the first 16 words of the message schedule array created from the 512-bit message block. They are directly derived from the message block's data. These are expanded to fill up W16 – W63. (In actual implementation, we have optimized it to left-shift and use only W0-W15).
5. **Compression (C)** – This represents the compression function of SHA-256, which takes the initial hash values, the message schedule words, and constants to produce a new set of hash values. This function is applied 64 times per block.
6. **64 Rounds of Compression** – Each message block undergoes 64 rounds of compression. In each round, a different word from the message schedule (W0 to W63) and a corresponding constant (Kt) are used.
7. **Kt [0 to 63] Constants** – These are 64 constant values used in the compression function, derived from the cube roots of the first 64 primes.
8. **Output per phase** - Once the 64 rounds of compression are completed, the initial hash values are added to the accumulated values in {a,b,c,d,e,f,g,h} <= {a+h0, b+h1, c+h2, d+h3, e+h4, f+h5, g+h6, h+h7} to produce the output hash of one phase.

9. **1st Stage Hash to Last Stage Hash** – The output hash for one block becomes the input hash values for the next block. The process repeats until all blocks are processed.
10. **Padding 0s and Message Size** – The final block includes padding to make its size up to 512 bits. This padding consists of 1'b1, followed by zeros, followed by the original message length, ensuring the total bit length of the padded message is a multiple of 512 bits.
11. **Final Output Hash (256 bits)** – After all message blocks have been processed, the final values of H0 to H7 are concatenated to form the 256-bit output hash of the SHA-256 algorithm.

3b. Algorithm for Bitcoin hashing using SHA256

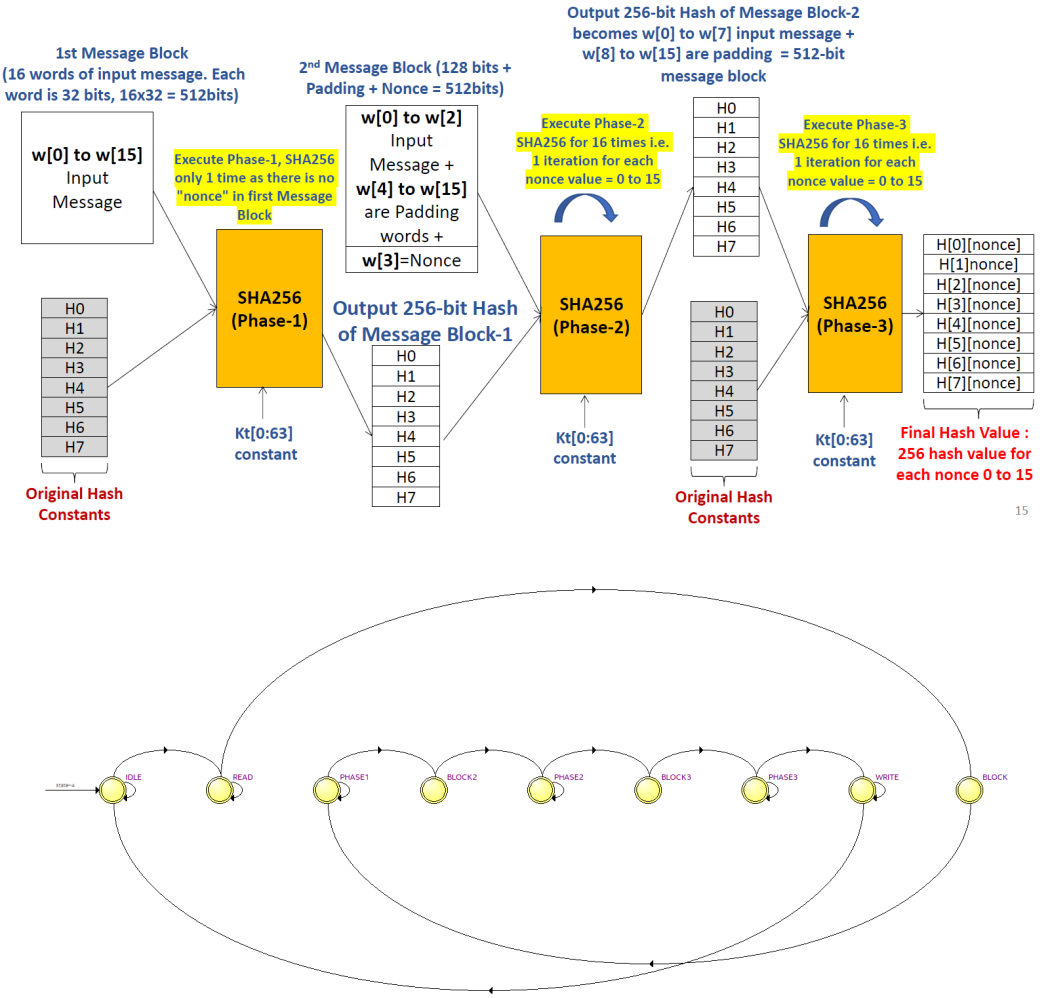

Figure 2 Bitcoin Hashing Algorithm

In our implementation, the Bitcoin hashing using SHA256 has been broken down into **3 key phases** as described in the above figure. In order to process Phases 2 and 3 for 16 nonce initializations, we have **vectorized** the input w[][] and the intermediate accumulators a[], b[], c[], ….h[] which also serve to store the output hash values.

Block : Initial Block Processing – This phase processes the first message block of the Bitcoin transaction data. The original hash constants are initialized, w[][0:15] are created using the 512 bits of message data.

Phase 1: The 512-bit message block (comprising the first 16 words) is hashed. This hashing integrates the original hash contents with the input data, and produces a 256-bit Hash to be used as input to next phase. While these 64 rounds of sha256_op are being processed, the shift-reg approach keeps generated the required w[16:63] which are available appropriately at w[][0] for every cycle.

Block 2 (Vectorized): The last 3 words of message data are read into w[nonce][0:2], w[nonce][3] is initialized in a vectorized form to the nonce value. The remaining bits w[][4] to w[][15] are padding bits, which include the 640-bit message size.

Phase 2 (Vectorized): The 512-bit message block (comprising the first 16 words) is hashed. This hashing integrates the hash contents from Phase-1 with the input data, and produces a 256-bit Hash, corresponding to each nonce value to be used as input to the next phase.

Block 3 (Vectorized): For the final phase, the input w[][0:7] are derived from the output hash of Phase-2. The remaining w[][8:15] are padding bits which also include the 256-bit message size.

Phase 3 (Vectorized): The prepared input from Block-3 is hashed using the original hash contents. This double hashing reinforces the security of the hash. Finally, it produces a 256-bit Hash corresponding to each nonce value, i.e, 256-bit hash values for Nonce = 0 to 15.

Write: In order to verify the correctness of the hashing process, we write the first hash value for each nonce, i.e, H0[0], H0[1], …H0[15] into the memory at the pointed write address, which is then verified by the testbench self-checking mechanism.

4. Simulation Waveform Snapshots

4a. SHA-256

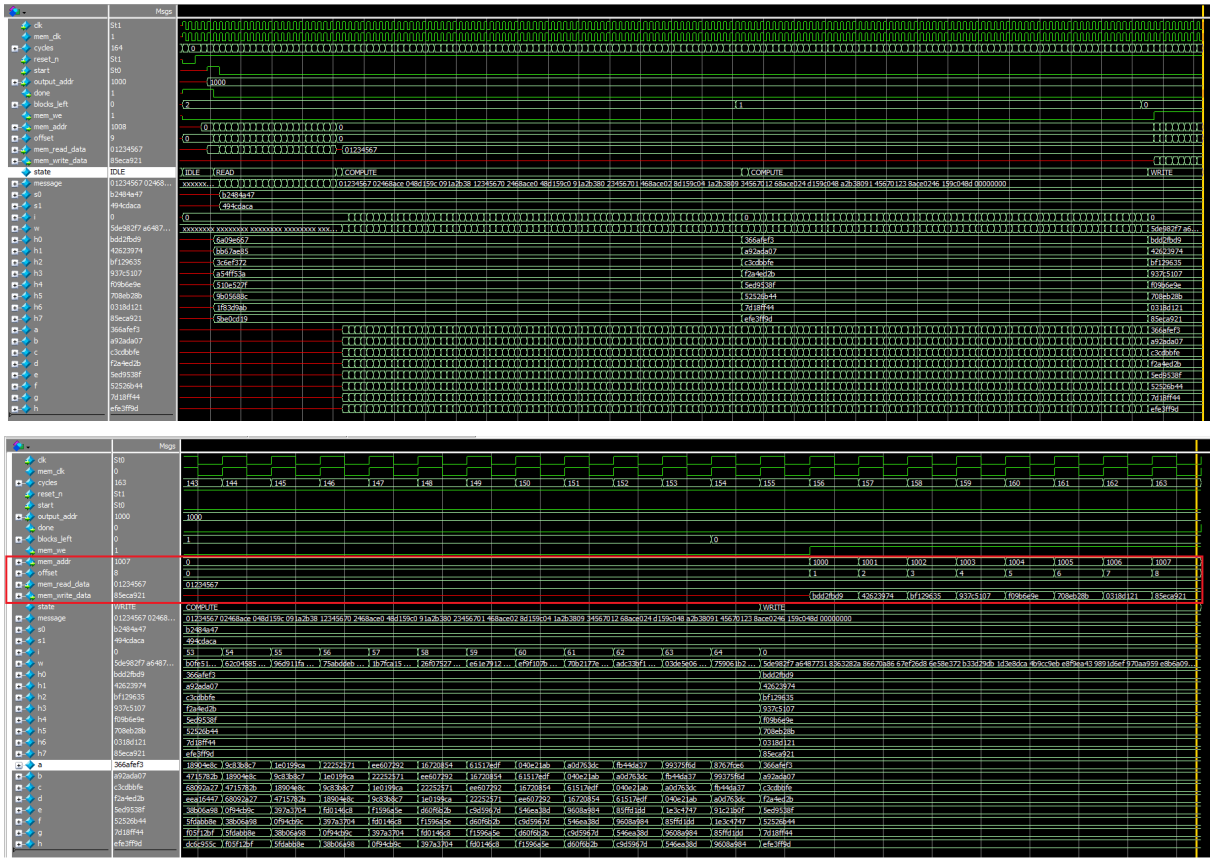

Figure 3(a) and 3(b) simplified_sha256 Module Simulation Waveforms

4b. Bitcoin hashing

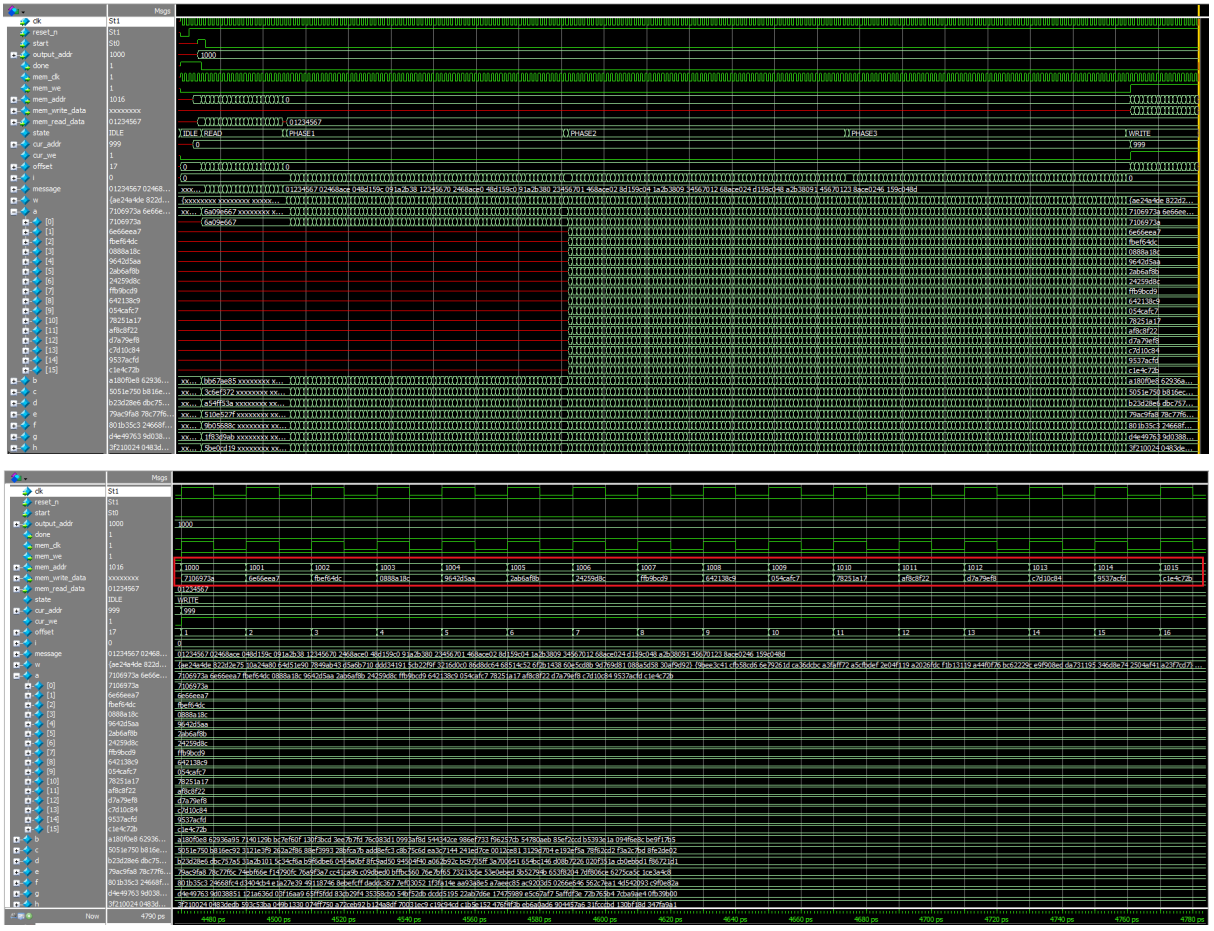

Figure 4(a) and 4(b) bitcoin_hash Module Simulation Waveforms

5. Simulation Transcript Output

Modelsim transcript window output indicating passing test results generated from self-checker in testbench:

5a. SHA-256

```
# vsim -c tb_simplified_sha256 -do "run -all; quit"
# Start time: 04:19:14 on Jun 08,2024
# Loading sv_std.std
# Loading work.tb_simplified_sha256
# Loading work.simplified_sha256
# 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:         164
#
#
# ****************************
#
# ** Note: $stop    : ./tb_simplified_sha256.sv(262)
#    Time: 3330 ps  Iteration: 4  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at ./tb_simplified_sha256.sv line 262
# Stopped at ./tb_simplified_sha256.sv line 262
#  quit
# End time: 04:19:14 on Jun 08,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
```

Figure 5 SHA256 Simulation Transcript

5b. Bitcoin hashing

```
# vsim -c tb_bitcoin_hash -do "run -all; quit"
# Start time: 04:21:39 on Jun 08,2024
# Loading sv_std.std
# Loading work.tb_bitcoin_hash
# Loading work.bitcoin_hash
# 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:         237
#
#
# ***************************
#
# ** Note: $stop    : ./tb_bitcoin_hash.sv(334)
#    Time: 4790 ps  Iteration: 4  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at ./tb_bitcoin_hash.sv line 334
# Stopped at ./tb_bitcoin_hash.sv line 334
#  quit
# End time: 04:21:40 on Jun 08,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
```

Figure 6 bitcoin_hash Simulation Transcript

6. Synthesis resource usage and timing report – bitcoin_hash

Compilation Setting (lowest Area*Delay) : Performance (High Effort)

Device used: Arria II GX EP2AGX45DF29I5 Timing Corner: Slow, 900mV, 100C

6a. ALUTs, Registers, Area snapshots

Compilation Report - bitcoin_hash

Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage
1	ALUTs Used	22,165 / 36,100 (61 %)
1	-- Combinational ALUTs	22,165 / 36,100 (61 %)
2	-- Memory ALUTs	0 / 18,050 (0 %)
3	-- LUT_REGs	0 / 36,100 (0 %)
2	Dedicated logic registers	13,225 / 36,100 (37 %)
3		
4	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	492
3	-- 5 input functions	298
4	-- 4 input functions	3447
5	-- <=3 input functions	17928
5		
6	Combinational ALUTs by mode	
1	-- normal mode	14943
2	-- extended LUT mode	0
3	-- arithmetic mode	5174
4	-- shared arithmetic mode	2048
7		
8	Logic utilization	24,235 / 36,100 (67 %)
1	-- Difficulty Clustering Design	Low
2	-- Combinational ALUT/register pairs used in final Placement	23178
1	-- Combinational with no register	9953
2	-- Register only	1013
3	-- Combinational with a register	12212
3	-- Estimated pairs recoverable by pairing ALUTs and registers as design grows	-129
4	-- Estimated Combinational ALUT/register pairs unavailable	1186
1	-- Unavailable due to Memory LAB use	0
2	-- Unavailable due to unpartnered 7 LUTs	0
3	-- Unavailable due to unpartnered 6 LUTs	411
4	-- Unavailable due to unpartnered 5 LUTs	2
5	-- Unavailable due to LAB-wide signal conflicts	482
6	-- Unavailable due to LAB input limits	291
9		
10	Total registers*	13225
1	-- Dedicated logic registers	13,225 / 36,100 (37 %)

Figure 7 Resource Usage Summary, Register Statistics and Fmax summary

6b. Fitter report snapshot

bitcoin_hash.fit.summary

```
Fitter Status : Successful - Sat Jun 08 14:50:06 2024
Quartus Prime Version : 20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name : bitcoin_hash
Top-level Entity Name : bitcoin_hash
Family : Arria II GX
Device : EP2AGX45DF29I5
Timing Models : Final
Logic utilization : 67 %
    Combinational ALUTs : 22,165 / 36,100 ( 61 % )
    Memory ALUTs : 0 / 18,050 ( 0 % )
    Dedicated logic registers : 13,225 / 36,100 ( 37 % )
Total registers : 13225
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 % )
```

Figure 8 Fitter Report

6c. Timing Fmax report snapshots

Compilation Report - bitcoin_hash

Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	137.34 MHz	137.34 MHz	clk	

Figure 9 Fmax Summary

7. References

[1] T. Hansen and D. E. Eastlake 3rd, “US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF),” *IETF*, May 01, 2011. https://www.rfc-editor.org/rfc/rfc6234#section-6 (accessed Jun. 07, 2024).

[2] Y. Turakhia, “ece111_Final_Project_SHA256.pdf,” University of California San Diego.

[3] Y. Turakhia, “ece111_Final_Project_Bitcoin.pdf,” University of California San Diego.

END