Bitcoin Hashing Report ECE-111

What is Bitcoin Hashing?

Bitcoin is based on a blockchain. A blockchain consists of several digital data blocks. The data that these blocks store can range across a variety of subjects, as they can be used in finance to keep track of transactions or in medicine to keep track of medical records. These blocks are chained together, and once blocks are chained together, they are immutable, meaning that the data within the blocks can not be changed. This chaining is done through cryptographic hashing algorithms. The bitcoin blockchain is the oldest blockchain, and the hashing done to chain the blocks together is what is known as bitcoin hashing.

Each block on the bitcoin blockchain uses the SHA-256 algorithm to generate its own signature, and this signature is what is used to chain it to the next block on the blockchain. However, not all signatures are valid signatures for the blockchain. Each signature must follow a certain pattern to be a valid bitcoin blockchain signature (i.e. it must start with ten zeros). Because of the use of a hashing algorithm, the only way to generate a valid signature is to try as many different inputs to the algorithm as possible. However, much of the data of the blockchain can not be mutated, as it is the transaction data and the metadata. Therefore, each block has a small amount of data added to it, and this data is called the 'nonce'. The nonce gets altered over and over again to try to find an input that would generate a valid signature. Finding the value of the nonce that generates a valid signature is what is known as bitcoin mining. There are always many bitcoin miners trying to generate valid nonces for new blocks, because the miners who generate valid nonces are given bitcoin as a reward.

Because the SHA-256 algorithm is a fairly intensive algorithm and because there are so many possible nonces that need to be tried, finding a valid nonce is quite difficult. This increased difficulty is what makes it nearly impossible to mutate blocks on the blockchain. If a malicious party mutates an existing block, they then need to generate new valid signatures for all of the blocks that follow. However, this malicious party is competing against every single miner that is trying to generate valid signatures for the blocks that follow. The malicious party can never win this competition, as all the other miners combined will clearly have more computing power than the single malicious miner. This shows the importance of using hashing to chain the blocks together. The use of hashing leads to the requirement of using heavy computational power to add new blocks to the blockchain, and this ensures that the blockchain is essentially immutable.

Reference: Blockchain, Bitcoin Hashing and Final Project Part-2 Slide Deck -Vishal Karna

Algorithm

We receive an input message of 608 bits that we pad with a 32 bit nonce along with 384 other padding bits. We then split this into two 512 bit message blocks. We feed the first block into a SHA-256 algorithm with a pre-generated 256-bit block of h values, and get a 256 bit output. We then generate 16 versions of the second block, with 16 different nonce values. For each version, we feed the block into a second SHA-256 algorithm, while using the output of the first SHA-256 algorithm as the h values, and get a 256 bit output. Finally, for each version of the 256 bit output, we pad the output so that it has 512 bits, then we feed the block into a final SHA-256 algorithm, while using the initial pre-generated 256-bit block of h values. The output of this final SHA-256 algorithm is the hash signature of the input function, and we have sixteen versions of this signature, with one signature corresponding to each nonce.

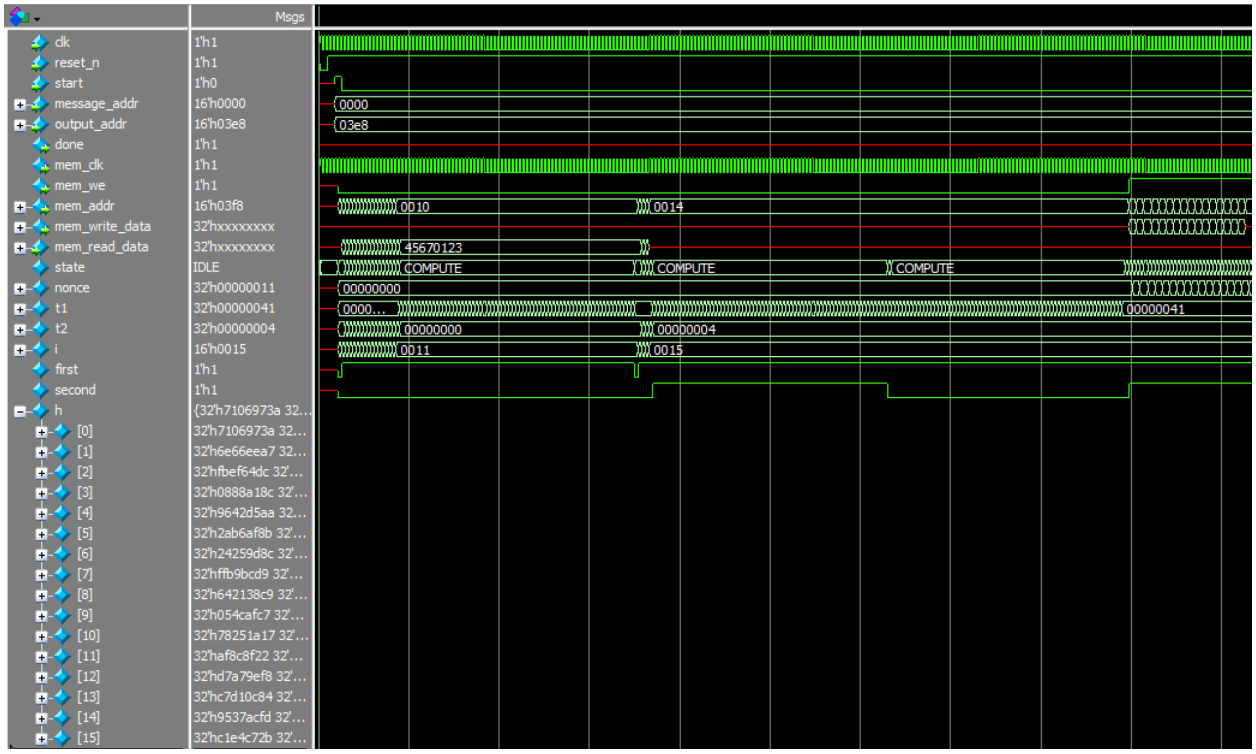
Serial: This implementation was done in SystemVerilog using states: IDLE, READ_BUFFER, READ, SETUP, SETUP_BUFFER, PHASE1, PHASE2, PHASE3, WRITE_SETUP, WRITE_BUFFER, WRITE. IDLE is the state we start in and it is the state that marks the hashing is complete. READ_BUFFER and READ states are used to ingest the input message. SETUP and SETUP_BUFFER are used to create the input messages for each SHA-256 algorithm. PHASE1 runs the first algorithm, PHASE2 runs the second algorithm, and PHASE3 runs the final algorithm. Finally, WRITE_SETUP, WRITE_BUFFER, and WRITE are used to write the output hash to the output port.

For the second and last SHA-256 algorithm, this implementation had each version run in order. This means that as soon as one nonce version is run, the next nonce version is run immediately after. Therefore, they are run in a serial manner.

Parallel: This implementation was done in SystemVerilog using states: IDLE, START_BUFFER, START, READ, FINISH, COMPUTE, WRITE_BUFFER, WRITE. IDLE is the state we start in and it is the state that marks the hashing is complete. START and START_BUFFER are the states that start the hash and start the read operation. READ is the state where we continue reading the message from memory. COMPUTE is the state where all three SHA-256 algorithms are run. FINISH is the state where the hash computation is completed. Finally, WRITE_BUFFER and WRITE are the states to write the output hash to the output port.

For the second and last SHA-256 algorithm, this implementation has each nonce version run in parallel. This means that all nonces run simultaneously, which makes the full bitcoin hashing run more efficiently. Therefore, they are run in a parallel manner.

Simulation


Msgs
clk 1'h1
reset_n 1'h1
start 1'h0
message_addr 16'h0000
output_addr 16'h03e8
done 1'h1
mem_clk 1'h1
mem_we 1'h1
mem_addr 16'h03f8
mem_write_data 32'hxxxxxxxx
mem_read_data 32'hxxxxxxxx
state IDLE
nonce 32'h00000011
t1 32'h00000041
t2 32'h00000004
i 16'h0015
first 1'h1
second 1'h1
h {32'h7106973a 32...
[0] 32'h7106973a 32...
[1] 32'h6e66eea7 32...
[2] 32'hfbef64dc 32'...
[3] 32'h0888a18c 32'...
[4] 32'h9642d5aa 32...
[5] 32'h2ab6af8b 32'...
[6] 32'h24259d8c 32'...
[7] 32'hffb9bcd9 32'...
[8] 32'h642138c9 32'...
[9] 32'h054cafc7 32'...
[10] 32'h78251a17 32'...
[11] 32'haf8c8f22 32'...
[12] 32'hd7a79ef8 32'...
[13] 32'hc7d10c84 32'...
[14] 32'h9537acfd 32'...
[15] 32'hc1e4c72b 32'...
0000
03e8
0010
0014
45670123
COMPUTE
COMPUTE
COMPUTE
00000000
0000...
00000041
00000000
00000004
0011
0015

Transcript

```
# ---------------
# 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:         256
#
#
# ****************************
```

Resource Usage

	Resource	Usage
1	Estimated ALUTs Used	18219
1	-- Combinational ALUTs	18219
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	17066
3		
4	Estimated ALUTs Unavailable	2801
1	-- Due to unpartnered combinational logic	2801
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	18219
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	3
2	-- 6 input functions	2798
3	-- 5 input functions	2579
4	-- 4 input functions	524
5	-- <=3 input functions	12315
8		
9	Combinational ALUTs by mode	
1	-- normal mode	10904
2	-- extended LUT mode	3
3	-- arithmetic mode	5264
4	-- shared arithmetic mode	2048
10		
11	Estimated ALUT/register pairs used	31259
12		
13	Total registers	17066
1	-- Dedicated logic registers	17066
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	17067
22	Total fan-out	152198
23	Average fan-out	4.28

1. **ALUTs:** 18219
2. **Registers:** 17066
3. **Area:** 35285 - Calculated in “bitcoin_hash/finalsummary.xlsx”
4. Fitter file located in “bitcoin_hash/bitcoin_hash.fit.rpt”

Fitter Report

Fitter Status	Successful - Fri Sep 13 19:49:13 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
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	90 %
Total registers	17066
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 Fmax Report

Revision Name	bitcoin_hash
Device Family	Arria II GX
Device Name	EP2AGX45DF29I5

	Fmax	Restricted Fmax	Clock Name	Note
1	140.08 MHz	140.08 MHz	clk	

1. **Fmax:** 140.08 MHz
2. Static timing analysis file located in “bitcoin_hash/bitcoin_hash.sta.rpt”