ECE 111: Advanced Digital Design Project
Prof. Yatish Turakhia

SHA-256 and Bitcoin Hashing Report

June 10, 2023

Conner Hsu (A16665092)
Haozhang Chu (A16484292)
Kirtan Shah (A16227067)

Contents

1 Simplified SHA-256

1.1 Introduction

Hash functions are powerful tools that play a vital role in various areas of computer science and information security. At their core, hash functions are mathematical algorithms that transform data of any size into a fixed-size string of characters, known as a hash value or hash code. This transformation is designed to have a few important properties:

- compression; the output hash value is fixed in size regardless of the size of the input.
- avalanche effect: a small change in the input results in a huge change in the output.
- determinism: the same input must always generate the same output.
- pre-image resistant: an inverse hash function should not exist and it should be very difficult to determine the input that generated a given output.
- collision resistance: the hash function should be nearly completely injective and thus should almost never have two inputs map to the same output.

For cryptographic applications, avalance effect, noninvertibility and injectiveness are very important properties. These properties all together make hash functions invaluable for tasks such as data retrieval, data integrity verification, password storage, and digital signatures. By producing unique and irreversible hash values, hash functions enable efficient data indexing, quick data comparison, and secure authentication. Whether it's ensuring data integrity, enhancing search performance, or providing robust security measures, hash functions are essential components in modern computing systems.

In this report, we explore the implementation of a particular hash function called Secure Hashing Algorithm - 256 or SHA-256 for short. This hash function takes in any length input and outputs a 256-bit digest/hash value. We will build this using System Verilog and use the Arria-II FPGA to realize the implementation on hardware.

1.2 Algorithm

Before running the SHA-256 algorithm, the input message must be broken up into blocks of 512 bits. The final block must contain at least 66 extra bits; a 1 followed by at least one 0, and 64 bits equal to the size of the original input message. The number of zeros padding between the first 1 and the message size bits depends on how many bits are needed to make the last block 512 bits long.

For example, suppose the message is 640 bits long. This means that the first block will contain the first 512 bits of the message, and the second block will contain the last 128 bits of the message. Next, a 1 will be appended to the last block making it 129 bits long. Then, 319 zeros will be appended to make the last block now 448 bits long. Last, 64 bits will be appended to store the message size and the last block will not be 512 bits long.

As another example, suppose that the input message divides evenly into 512-bit blocks meaning that the message size is some multiple of 512. In this case, the final block will have a 1 in the beginning, then 448 zeros, then end off 64 bits for the message size. The number of zeros is increased to 448 to ensure that this final block is 512 bits long.

Throughout the SHA-256 algorithm, a set of hash values $H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7$

are updated after each block in the input message is processed. These variables are each 32 bits long and thus when put together form the 256-bit output digest.

From here, the states of the SHA-256 are described as follows. The algorithm begins in the IDLE state. In this state, the algorithm waits for a start signal to begin computation. Once a start signal is recieved, it initializes iterative variables, the address to read in the input message, and the hash values, H_i for $0 \leq i \leq 7$. Once this is done, the algorithm will move into the READ state in the next clock cycle.

In the READ state, the SHA-256 algorithm will read data from the memory read in input of the module. Since the memory module being used can only read out 32 bits (or one word) at a time, it takes multiple clock cycles before the entire input message will be read in. In our implementation, we designed the SHA-256 to take a fixed input message length of 640 bits. This means that the READ state will use at least 20 clock cycles to read in $20 \cdot 32$ bits. Once the entire input message is read in, the algorithm moves to the BLOCK state in the next clock cycle.

In the BLOCK state, the algorithm will separate the input message into 512-bit blocks in the same process as described earlier. For the 640-bit input message, 512 of the bits will be placed in the first block, and the last 128 bits will be placed in the second block. Some logic is needed to fill the remaining 384 bits of the second block with the zeros padding and input message size. A for-loop is used to initialize the first 10 words with $100\ldots00$ and then the last two words are set to the input message size. Additionally, before moving into the next state, another set of variables, a, b, c, d, e, f, g, h are initialized to the hash values from earlier. Once this is completed, the algorithm moves to the COMPUTE state in the next clock cycle. When the algorithm is in the BLOCK state for the first time, it will send the first 512 bits of the input message to the COMPUTE state. When the algorithm is in the BLOCK state of the second time, it will send the last 128 bits of the input message plus the padding and message size to the COMPUTE state.

In the COMPUTE state, the algorithm processes the 512-bit block. The algorithm will break these 512 bits into 16 words and then use a process called word expansion to expand this into 64 words. Each of these blocks of 32 bits are processed sequentially over 64 clock cycles. A 32-bit block along with the a, b, c, d, e, f, g, h values are passed to a SHA-256 hash round function each clock cycle.

```
1  // SHA256 hash round
2  function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f,
       ↪ g, h, w,
3      input logic [7:0] t);
4    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
5    begin
6      S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
7      ch = (e & f) ^ ((~e) & g);
8      t1 = h + S1 + ch + k[t] + w;
9      S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
10     maj = (a & b) ^ (a & c) ^ (b & c);
11     t2 = S0+maj;
12     sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
13   end
14 endfunction
```

While it is difficult to explain everything that is going on in the function, some important properties should be noted.

- The AND operation is a noninvertible function if the function is not outputting 1. It is not possible to determine the inputs if the output of an AND operation is 0.
- The XOR operation is always noninvertible.
- While the function is complicated it can be realized that there is no nondeterministic processes in this function.

These properties help SHA-256 to realize properties such as determinism and pre-image resistance. After the SHA-256 hash round is complete, the output of the function will be assigned to a, b, c, d, e, f, g, h and will be used in the next clock cycle. The processs of updating the a, b, c, d, e, f, g, h values for each word and then using the same values for the next 32-bit block helps to realize the avalanche effect. A small change in the input message will affect the output of one word, which then affects the input/output of the next 32-bit block, and will then change the inputs/outputs for all of the 32-bit blocks down the line.

Once all 64 words blocks are processed in the COMPUTE state, the algorithm will move back to the BLOCK state. If there are more blocks of 512 bits to be processed, then it will take the next block and move back to the COMPUTE state. However, if there are no more blocks to be processed, then the algorithm will move to the WRITE state.

In the WRITE state, the algorithm will write the hash values, $H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7$ to memory. Since the memory module being used can only write 32 bits to memory per clock cycle, it will take at least 8 clock cycles to write the hash values to memory.

1.2.1 Finer details of reading/writing from memory

The SHA-256 uses an external memory module to read in the input message and write out the output hash. This memory module can

1.2.2 Optimizations

1.3 Simulation Results

```
# --------
# 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:         168
#
#
# ***************************
#
# ** Note: $stop    : E:/Desktop/ucsd stuff/ECE 111/ECE111_Collaboration/simplified_sha25
#    Time: 3410 ps  Iteration: 2  Instance: /tb_simplified_sha256
```

2 Bitcoin Hashing

2.1 Introduction

Bitcoin hashing is a critical aspect of the cryptocurrency's underlying technology, blockchain. In the context of Bitcoin, hashing refers to the process of converting transaction data into a fixed-length alphanumeric string, i.e. a hash value. In order for a block to be accepted as part of the block chain, its hash must be meet a specific criteria. Bitcoin miners must repeatedly hash different combinations of the original transaction data plus a random string of numbers called the "nonce". By changing the nonce, different hash values can be generated and Bitcoin miners will keep trying different nonces until the hash meets the specific criteria. The process of finding the nonce that meets the criteria requires a lot of computational power and since miners compete with each other to see who meets the criteria, speed is also very essential.

In this part of the report, we will investigate the implementation of a Bitcoin hashing algorithm that will attempt to generate multiple hash values at a time for different nonce values. We are not trying to meet any specific criteria for the final hash value, but instead are trying to design a module that will compute many hash values with different nonces very quickly.

2.2 Algorithm

The goal of this algorithm is to compute 16 hash values for 16 different nonce values. Since speed is highly important in the context of Bitcoin mining, we have designed our implementation to computes these hashes as fast as possible.

This algorithm will be designed to take in a 640-bit input message. The first 608 bits are reserved for the transaction data and the last 32 bits of the input message are reserved for the nonce. The hash function for the Bitcoin hashing will be realized by the SHA-256 hash function from earlier. The first block of 512 bits will contain the first 16 words of the transaction data. The second block will contain the last 3 words of the transaction data and the word containing the nonce. These two blocks will then be processed by the SHA-256 algorithm.

We have designed one module to process the first 512-bit block that only contains transaction data. This module then feeds into 16 more SHA-256 instances that processes the second 512-bit block. This is necessary because the second block is different for all 16 nonce values since the second block contains the nonce value. From here, all of the hash values for each nonce value can be computed in parallel.

2.3 Resource Usage

2.4 Simulation Results