The Design of a Bitcoin Hash Processor

ECE 111 Final Project

Fall Quarter 2020

(If you are in a team of two, only one report needs to be submitted)

Date: 12/18/2020

Team Members: Alex Wang(A15234161), Euicheon Hwang (A14984264)

**Introduction**

There are many cryptocurrencies that are being used currently throughout the world, like Dogecoin, Litecoin, Ethereum, and the most famous one, Bitcoin. Cryptocurrency allows users to directly exchange the “electronic cash” with other users without having to interact with a central bank or the administrator of the cryptocurrency. But how does Bitcoin work as a currency?

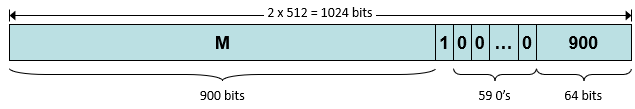
The first important feature of Bitcoin to mention is its security as a currency. How can each transaction be trustworthy, when there is no central system that keeps track of the transactions to make sure that they went through? To deal with this issue, Bitcoin uses blockchain technology to keep track of every single transaction. A blockchain is digital blocks of data chained together. Starting from the data block of the first ever Bitcoin transaction, blockchain continuously chains newer transaction data onto existing ones, essentially creating “a chain of blocks” as the name suggests. This blockchain is kept forever, and thus it is able to act as a public and global ledger for existing and future transactions.

Some may wonder how the blockchain technology can be secure when it's publicly available to everyone; wouldn’t a malicious miner be able to alter the data to get free money? However, this idea would realistically never happen. A new block of data can only be added when he or she finds the according signature to be linked between the blocks, including the newer ones that are constantly being chained. The signatures can only be calculated by finding a cryptographic hash for computationally difficult problems, which get progressively harder for newer blocks. Because the rest of the miners are also calculating this new, correct version of the signature, the malicious miner will never be able to push his new signatures without having significantly more computational power than the rest of the network.

Each block contains a cryptographic hash calculation of the previous block through a SHA-256 algorithm (signature). However, a signature does not always qualify to be chained; there exists a certain “rule” that accepts only certain digital signatures created from the hashing. Therefore, each block has a piece of data called “nonce”, which changes repeatedly until the block calculates an acceptable signature. In our project, we will be replicating the process of computing different signatures based on different values of the nonces.

**Description of the SHA-256 Algorithm**

SHA-256 algorithm creates a 256 bit encrypted message from a given input. The input is padded with a single 1 and multiple 0’s until the total message (with the padding) is 64 bits less than some multiple of 512. Within the 64 bit space between the padded message and the end, the length of the padded message is written down. As an example below:



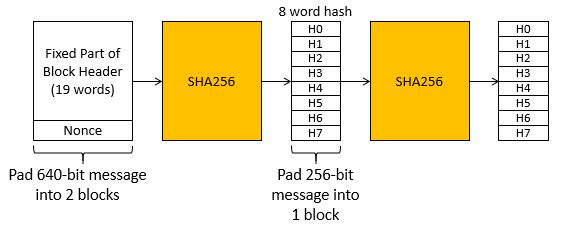
A message M that is 900 bits long will be padded until 64 bit less than the closest multiple of 512, which would be 1024 - 64 = 960. Therefore, there would be a single 1 followed by 59 0’s, and then the length of M would be represented by the 64 bit message at the end. Each of the 512 bits are represented as a block.

Then, for every block, the message digest (eight 32-bit words) changes. For each of the blocks, the message digest initially changes through a word expansion, and then processed with rotation, shifts, and logic gates like AND and XOR gate. Then the message digest is updated by adding the processed version with the non-processed version.

After all blocks have been processed, the final version of the message digest is the output (hash value) of our SHA-256 algorithm encryption.

**Description of the Bitcoin-Hash Processor Final Project**

Mentioned in the overview, Bitcoin-Hash processor calculates the proper signature for the blocks to be chained by running the SHA-256 algorithm on the data block, which contains a “changing” nonce value. Given an input message of 20 words where each word is 32 bit, we cannot fit the message (640 bits) into a single data block in a bitcoin block chain, which has a size of 512 bits. Therefore, the input data is split into two blocks. The last word of the input message is the nonce value. The second block would be padded to fit the block through the same method that is used for SHA-256, where the message is padded with a single 1, bunch of 0’s, and then the 64 bit representation of the message length. Then the message is input through a SHA-256 algorithm, which outputs a 8 word hash value. This 256-bit message is padded again like before to be run through another SHA-256 algorithm, which returns the final hash values for our processor. The second block and the hash function of the second block is processed 16 times, and each of the final results (the H0-H7 on the most right) are stored per iteration. For our project, we are only storing H0 for each of the iterations, and returning the 16 different H0’s from the iterations as our final result.



For our project, the nonce value only goes up to 15 for convenience. Another altercation made for this project was that there is no target signature that the hash function is aiming for. The hash function would run a fixed 16 iterations, from 0 to 15, and the final iteration would be the signature that is used for the next block.

**Design Details**

We had two designs: a sequential design and a parallel design. The goal of the sequential design was to minimize the Area\*Delay while the goal of the parallel design was to just minimize the Delay.

When creating the sequential design, our first and foremost concern was to reduce the area as much as possible. We found that using a temp array to hold our H values, then transferring the calculated values to our h0, h1,... h7 arrays after the calculations improved our area usage. We think this is because during the calculations, instead of navigating through 7 arrays, it would only need to navigate through 1 array. One of the most important optimizations we implemented was only using 16 w arrays. To do this, we had to do our expansion using wtnew in the same state as when we did our calculations using sha256\_op. Not only does this greatly reduce the area because our w array was much smaller than if we were using 64 of them, it also reduces area and number of cycles by merging the expansion state and the calculation state into a single state. One of the important optimizations we tried to implement was pipelining. The idea of pipelining is to calculate the inputs for sha256\_op one state beforehand. We successfully implemented this but found that it actually decreased our Area\*Delay. While it did increase our fmax by a considerable amount, it also increased our Area, which in total decreased our total efficiency.

Our Parallel design was built upon our sequential design. We decided to go the route of vectorization instead of module instantiation. To do this, we changed our w array and H array into a 2d array: w[NUM\_NONCES][16], H[NUM\_NONCES][8]. We also changed our a, b, c, d, e, f, g, and h variables into 1d arrays (a[NUM\_NONCES], b[NUM\_NONCES], etc). Everytime we would initialize w, H, w, or a-h, we would run a loop from i=0 to i<=NUM\_NONCES to initialize every index in the array. In addition, when we run the sha\_256 function hash function, we would use a loop to run it on all 16 nonces at once. This allowed us to run our program with the same number of cycles as if we ran our sequential design with only 1 nonce. With this design, our area increased by a large magnitude (because of all the arrays) as compared to the sequential design, but the number of cycles also decreased by a factor of 10. As stated in the slides, even though area is not a concern for calculating the delay, optimizing it is still important because Quartus will fail the fitter if the design is too large. A lot of our time was spent trying to decrease our number of ALUTs in order for the fitter to be able to accept our design. We had to pipeline our inputs for sha256\_op in our parallel design in addition to all the optimizations from our sequential design. With these changes, we were barely able to fit our design into the fitter.

**Summary of Results**

Our sequential design used 2103 ALUTs, 1448 Registers, had an Fmax of 119.02MHz, and 2235 cycles. This resulted in an area of 3551, a delay of 18.778 microsec, and finally an Area\*Delay of 66.682 millisec\*area. Although our area and number of cycles were slightly better than the median, our Fmax was significantly worse, which resulted in an Area\*Delay that is worse than the median. We believe that the reason this is happening is because although the number of cycles we managed to get was decent, the amount of work we are doing per cycle is quite a lot. I think if we had more time we could have tried to optimize our calculations so the frequency per cycle would be higher.

Our Parallel design had the same problem as our sequential design, where our area and number of cycles was slightly better than the median but our Fmax was significantly worse. This is because we reused and built upon our sequential design, so the inefficiency in the calculations was transferred over to this new design. Our numer of ALUTs and Registers were 26176 and 17377 respectively. Our number of cycles was 236 and our Fmax was 97.47. This resulted in a delay of 2.422 microsec which is significantly better than our sequential design, but still slower than the median due to our low fmax. Unfortunately, a lot of our time was spent trying to decrease the area in order for the Quartus fitter to run successfully, so we did not have any time to optimize our Fmax.