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: December 12, 2020

Name: Christian Kouris

**Introduction**

Bitcoin is a currency that is used by many people in order to get an untraceable way to spend money. Bitcoin uses a blockchain which is a giant chain of every single transaction on the Internet. Every new transaction is added to the blockchain making it change as a lot of the time. Blockchains are secure because it takes a lot of computational effort in order to find a valid hash for a new block. The hashing algorithm used for the bitcoin blockchain is SHA-256. In this project I will test the results of the bitcoin hasher by using different nonces.

**Description of the SHA-256 Algorithm**

The SHA-256 algorithm has 8 different sections to its hashing function and uses many bitwise operations in order to achieve its goal of being irreversible. Right rotates make the backbone of the sha 256 algorithm as they are used commonly. Here is some code for the sha-256 algorithm

**S1 = rrot(e, 6) ^ rrot(e, 11) ^ rrot(e, 25);**

**ch = (e & f) ^ ((~e) & g);**

**t1 = h + S1 + ch + sha256\_k[t] + w;**

**S0 = rrot(a, 2) ^ rrot(a, 13) ^ rrot(a, 22);**

**maj = (a & b) ^ (a & c) ^ (b & c);**

**t2 = S0 + maj;**

**return = {t1 + t2, a, b, c, d + t1, e, f, g};**

In this case rrot stands for a right rotate.

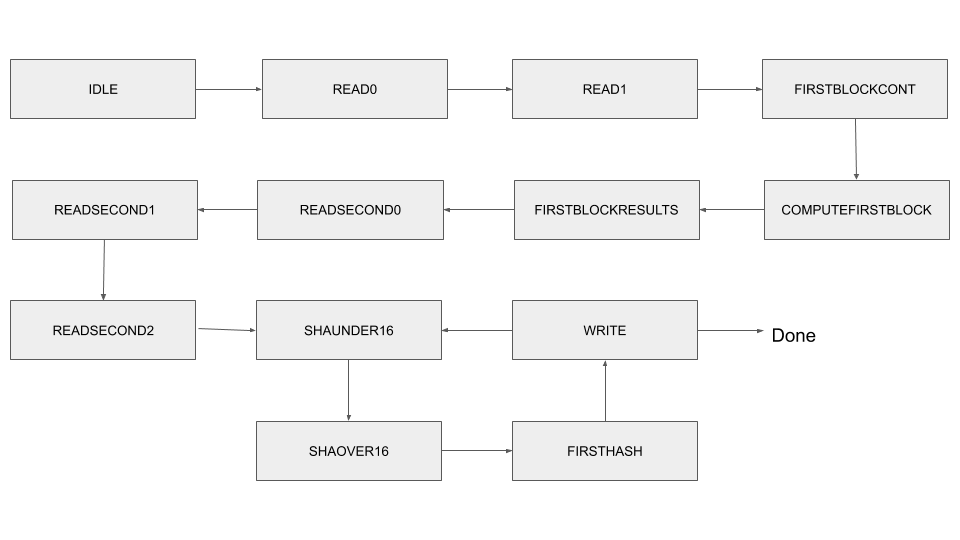
**Description of the Bitcoin-Hash Processor Final Project**

This final project takes in 19 “words” of message and affects one word of “nonce” in order to see what the resulting hash is. In this case a word is 4 bytes of data. The message is supposed to be metadata that we cannot change, such info as a timestamp and a recipient. The nonce is the only thing that we can change and as we see in the project has drastic effects on the final hash. The Project basically reads the first 16 words in a block and uses the SHA-256 hash function to hash those words. Then it takes the next 3 words, the nonce, and 12 words of padding in, hashes those, and adds them to the previous hash. With this new hash, the algorithm does another SHA-256 hash and comes to a final result. Essentially it double hashes the input message.

**Design Details**

The Strategy I took for this design is to be able to rely on a state machine instead of using many for loops. The design has many states which serve a different purpose; for example, some are used to call the hashing function while others are used to read and write from memory. One strategy used was to make the wt array used in the sha hasher hold only 16 values instead of 64. This makes the total area of the design smaller and makes the program more optimal.

Here is a state diagram of the design:



**Summary of Results**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **bitcoin\_hash1.sv** (MIN AREA\*DELAY DESIGN) | | | | | | | |
| Compiler Settings | #ALUTs | #Registers | Area | Fmax (MHz) | #Cycles | Delay (microsec) | Area\*Delay (millisec\*area) |
| balanced | 2088 | 1240 | 3328 | 108.47 | 2283 | 21.047 | 70.045 |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **bitcoin\_hash2.sv** (MIN DELAY DESIGN) | | | | | | | |
| Compiler Settings | #ALUTs | #Registers | Area | Fmax (MHz) | #Cycles | Delay (microsec) | Area\*Delay (millisec\*area) |
| balanced | 2088 | 1240 | 3328 | 108.47 | 2283 | 21.047 | 70.045 |
|  |  |  |  |  |  |  |  |