$\mu\mathcal{P}$*coin*: A Simplified Cryptocurrency Protocol*

Herrick Fang and Teerapat (Mek) Jenrungrot
Email: {hfang,mjenrungrot}@hmc.edu
E155: Microprocessor-based Systems Design and Applications

December 8, 2017

Abstract

Cryptocurrency has been one of the main public interests since Bitcoin has risen to popularity in this past year. Our team has chosen to design a simplified version of the cryptocurrency protocol based on blockchain technology. This project prototypes a system consisting of two sets of a Raspberry Pi and an FPGA, where each set replicates one user in the network; the prototype demonstrates the core functionality of cryptocurrency: making transactions, storing transactions on the blockchain, and performing the proof-of-work algorithm. The Raspberry Pi handles the user interface, which includes the webserver and HTTP connections between different users. The FPGA, which connects to the Raspberry Pi, acts like a hardware accelerator to perform a proof-of-work algorithm based on the SHA-256 hashing algorithm. Once the proof-of-work is finished, the result is sent back to the Raspberry Pi and outputted to the webserver.

1 Introduction

Cryptocurrency, especially Bitcoin, is an emergent phenomenon that has been generating a lot of attention in 2017, especially since the price of Bitcoin has exceeded $10,000 USD[1]. Cryptocurrency is an electronic cash system that utilizes a decentralized peer-to-peer network such that all information about transactions is stored electronically online in a data structure called the blockchain. The success of many cryptocurrencies is based on the underlying design principle of the blockchain, which is robust for storing information distributively and makes the concept of decentralization possible.

A blockchain is a data structure consisting of a linked list containing blocks, where each block has a pointer to its predecessor through a reference to the predecessor's hash [5]. This specification makes using a blockchain possible for correctly representing and persistently storing information in the form of a chain of information starting with an original, genesis block. This is possible through the concept of hashing in which some arbitrarily large input is converted to some fixed-length hash. Note that the same input information will always generate the same output information. In our case, we use SHA-256, which is used in many modern cryptocurrency protocols to generate a 256-bit hash from some given data of a block. To add to the blockchain, a block must meet certain specifications. It must have the correct previous hash and generate a hash that is difficult enough to add to the chain. By difficult, we mean that a block must contain enough zeros that satisfy some metric that is created by the designers of the given technology.

Based on the application of blockchain technology on cryptocurrency, we propose a digital system that can be used as a prototype for accelerating computations of the proof-of-work algorithm using hardware. Then, we demonstrate our work by creating a simplified version of cryptocurrency. The combination of these two aspects define our projcet. The primary goals of this project is to extend the scope of proof-of-work algorithm in cryptocurrency to a hardware accelerator using an FPGA.

To make this project feasible within the time-span for the final project, we simplify the cryptocurrency protocol by designing a system consisting of two users, or nodes, connected to each other via the HTTP protocol using HTTP requests like GET, POST, or PUT. Individually, each user has an access to a user interface where one can create a transaction, add a block to the blockchain (commonly known as mining), and access its own copy of the blockchain or the list of pending transactions waiting to be added to the

*Implementation of the project is documented and available on https://github.com/fangherk/MicroPCoin
[1] https://coinmarketcap.com/currencies/bitcoin/

blockchain. Each FPGA is connected to the Raspberry Pi for performing the proof-of-work algorithm that repeatedly computes the cryptographic hash function SHA-256 until the block meets a certain criteria so that a block can be added to the blockchain as discussed in the difficulty section above and in Appendix B. Figure 1 illustrates the overview design of the system composed of two sets of Raspberry Pi and FPGA, and Figure 3 demonstrates the block diagram of our design for one user, one set of Raspberry Pi and FPGA.

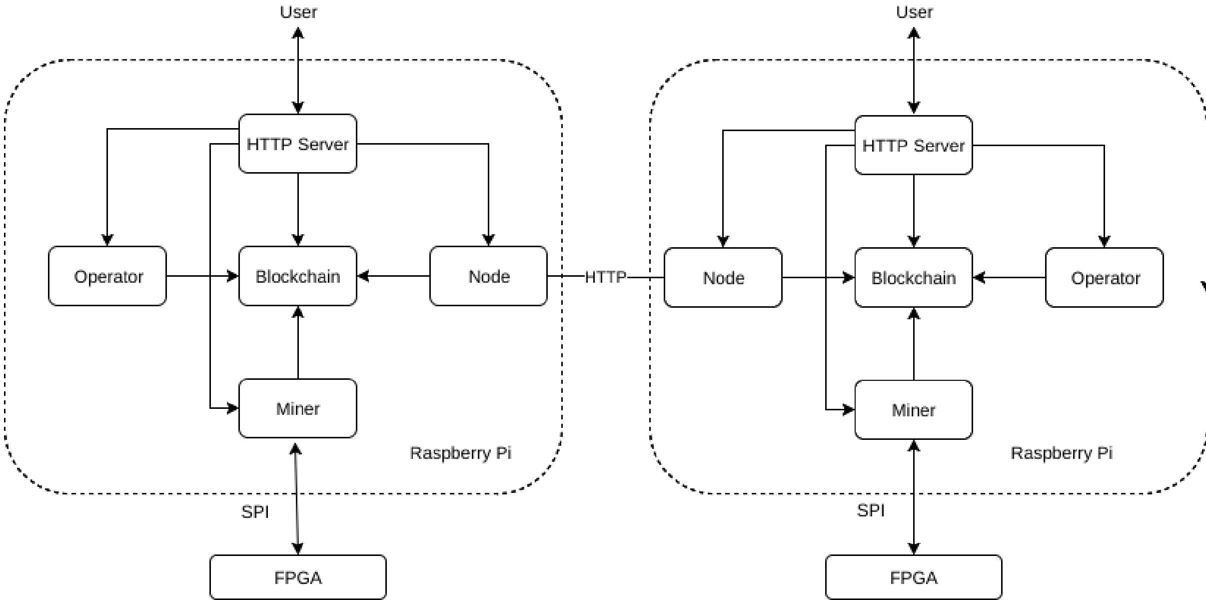


Figure 1: Pi + FPGA system user-blockchain model

2 Schematics

Figure 2 illustrates the connection between one set of a Raspberry Pi and an FPGA. As shown by Figure 1, two Raspberry Pis are connected to school network that allows them to communicate to each other wirelessly. Raspberry Pi communicates with the FPGA using the SPI interface using the SPI frequency of 150 kHz.

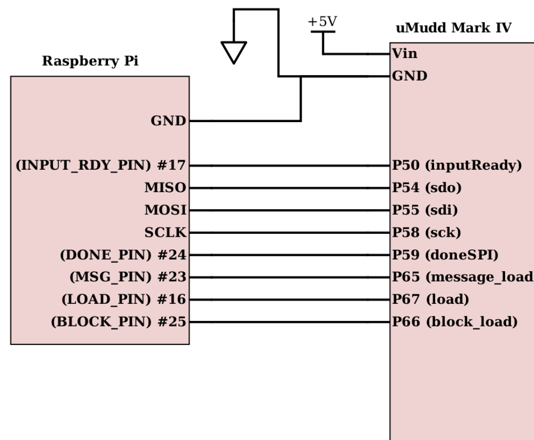


Figure 2: Board Schematic

3 Raspberry Pi Design

For our implementation design, the Raspberry Pi manages all the connections between users and all operations surrounding the typical cryptocurrency protocol, including creating a wallet, making a transaction, performing the proof-of-work, etc. Note that the proof-of-work process will be redirected to the FPGA, and the Raspberry Pi will only handle the data flow between components in our designed system. Even though our cryptocurrency protocol operates using the HTTP protocol, we separate our program into 5 components as described by Table 3. All interfaces over the HTTP protocol are listed in the Appendix C, which is self-documented. To understand Appendix C and parse the large amount of code, we recommend looking at Appendix C and go to the corresponding python file in Appendix G reading the descriptions of the functions to see the inner workings of MicroPcoin.

Task	Purpose
Raspberry Pi	
HTTP Server	Handles blockchain, wallets, transactions, mining and connecting to peers
Blockchain	Handles block and transaction arrival + synchronization of transactions and blocks
Operator	Handles wallets and addresses
Miner	Handles pending transactions and block creation process
Node	Handles receiving new blocks, peers, and transactions
FPGA	
Miner	Computes the cryptographic hash function for proof-of-work algorithm

Table 1: Work handled by the Raspberry Pi and the FPGA

The Raspberry Pi acts as the backbone of our project for handling all HTTP requests related to accessing/modifying the blockchain and making transactions. To handle the HTTP requests, we use the Python Flask library as a bridge to simplify sending data from one endpoint to another. We base our project on the given implementation [2], which we transfer and modify to a python-based solution of the cryptocurrency protocol. The Flask library allows us to specify endpoints, which are URLs, that nodes can send HTTP requests to, which are given in Appendix C. When these endpoints receive a specified request, they act accordingly to transfer the correct data to and from other Raspberry Pis on the distributed network.

Recall that we divide the main components of our proposed cryptocurrency into 5 components: HTTP Server, Node, Blockchain, Operator, and Miner. The specific aspects of each component are described below.

HTTP Server To ease the implementation of HTTP requests on different endpoints as described in Appendix C, we use the Python and Flask library. The main usage of Flask[2] is primarily for interfacing with a user. For instance, a user can make an HTTP GET request to an IP address `142.129.183.125` on port `5000`, which is the port our program operates at to get all blocks inside the blockchain kept in the database in JSON format of a machine with IP address of `142.129.183.125`. With this base URL, the node has full access to other nodes and thus can share data based on the existing endpoints.

Node The *Node* component handles connections to other users and the exchange of data with other Raspberry Pis. Recall that each Raspberry Pi has its own version of blockchain. So the work of synchronization of blockchain is handled by this component by applying HTTP requests to connect and transmit information. More can be found in Appendix G for the node functions. The main functionality of this component is making connections with a new peer, broadcasting its own version of blockchain, and handling received information regarding other peers' blockchain to update one's own blockchain. The process of resolving different multiple blockchains is discussed in Appendix B.

[2] `http://flask.pocoo.org/`

Blockchain The *Blockchain* component is the core of our cryptocurrency protocol. *Blockchain* handles two main things: a database that contains the blockchain and a database of pending transactions (e.g. transactions that are not yet undergone proof-of-work process). Given a HTTP request from the *HTTP Server*, *Blockchain* can, for example, get the latest block of the blockchain, add a block to the blockchain, or add a new transaction to the list of pending transactions. *Blockchain* handles everything related to reading from and writing to the blockchain database and the pending transaction database. More details of the blockchain protocol is described in Appendix B and the reference code can be seen in Appendix G.

Operator The *Operator* component handles operations regarding wallets, addresses, and transactions related to those addresses. For example, a user can make a transaction by making a HTTP request, containing the target address and amount of coins you want to send to the *HTTP Server* which then calls *Operator* component; the transaction is then signed using the private key corresponding with the address; *Operator* will then next invoke *Blockchain* for adding a signed transaction to the list of pending transactions. The main intent, however, is that operator handles what to do with wallets. More details of signing and verifying are described in Appendix B and Appendix G.

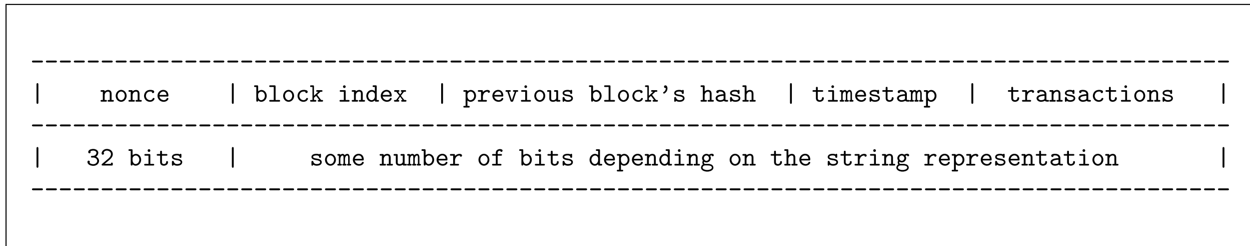
Miner The *Miner* applies the proof-of-work concept. Essentially, it obtains a list of pending transactions from *Blockchain* and attempts to create a new block from the transactions. The proof-of-work process is described in Appendix B. In our implementation, the *Miner* gets the earliest two transactions into the block, together with the fee transaction for those two transactions and the reward transaction. The fee transaction and reward transaction are used to pay the miner for its proof-of-work process. Note that the fee transaction and reward transaction has a slightly different concept. In the Bitcoin system, Bitcoin has a hard limit of the number of coins the entire network can produce. Miners, after hitting the hard cap, will receive a reward in the form of fee transactions. Hence, to replicate the Bitcoin system, we have two types of reward transactions: reward and fee. In our implementation, the *Miner* can also make an SPI connection to the FPGA to send a block to perform proof-of-work. Once the proof-of-work process is done, the Miner will receive the hash together with the discovered nonce value.

4 FPGA Design

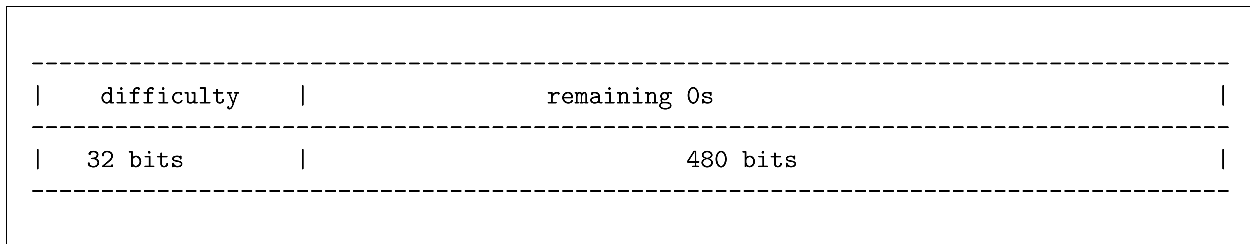
In our design, the FPGA is a central piece for computing the cryptographic hash function according to the specification FIPS 180-4 [3]. In the real world, FPGAs and ASICs are commonly used for cryptocurrency mining because they can be configured to perform the task at a better speed compared to the task done by typical CPU. Thus, our FPGA has an intimate relationship with the Miner process in the Raspberry Pi to process the proof-of-work algorithm, and, in this report, the term Miner will be used interchangeably with the term FPGA unless specified otherwise. Note that in our design, the FPGA is used to compute the hash for the proof-of-work process only; other processes such as recalculating the hash for verifying the entire blockchain are done on the Raspberry Pi.

In the blockchain protocol, we recall that the goal of a miner is to add a block to the blockchain and broadcast the just-mined block to the entire network, so everyone acknowledges that the miner has received the reward. The constraint on when a block can be added to the blockchain is described thoroughly in Appendix B. Essentially, a block can be added to the blockchain if the block is as difficult as the blockchain requires. The difficulty can be determined from the hashing algorithm by calculating the number of leading zeros in the bit representation of the 256-bit hash of the SHA-256 algorithm. The amount of difficulty required by the blockchain is calculated deterministically based on the block index, e.g. the number of blocks since the genesis block, and this number increases as the blockchain gets longer. The formula for calculating the required difficulty given the block index is provided in the Appendix B. Hence, the work of a miner is essentially to repeatedly change the nonce value inside the block and then compute the hash to see if that block has enough difficulty to be added to the blockchain, and the work is done when the difficulty criteria is met.

In this design, the Raspberry Pi sends a block–encapsulating the information like block index, nonce, timestamp, previous block's hash, and transactions–whose representation contains a 32-bit nonce value at the beginning of the representation and the sequence of ASCII characters corresponding to the string representation of block index, previous block's hash, timestamp, and transactions in order, as follows:



Given the representation of a block in binary as described earlier, a block, regardless of the number of bits in the representation, is separated, padded, and sent over to the FPGA in chunks, with each chunk containing 512 bits according to the SHA-256 specification. After sending the block's encoding to the FPGA, the Raspberry Pi sends a final 512-bit message to the FPGA containing the required difficulty for the block to be added. The format of the final 512-bit message is such that the beginning 32 bits represents the difficulty value and the remaining 480 bits are 0s, which indicates that the block will be the last chunk of the message sent from the Raspberry Pi to the FPGA:



The C program for padding the message and creating the communication between FPGA and Raspberry Pi is given in the Appendix section under the name `call_spi`.

Given the block's encoding, the goal of the FPGA is to find a nonce value that gives the hash value that meets the difficulty criteria. In our design, we start the nonce value with 0 and compute the hash using the SHA-256 algorithm. The hash is used to calculate the difficulty and compared with the difficulty criterion inside the FPGA. If the difficulty criteria is not met, the nonce is incremented by one, and the entire process from computing the hash to comparing the difficulty with the criteria is performed again. Once the difficulty criteria is met, the FPGA then sends the hash value and the corresponding nonce value back to the Raspberry Pi. This concludes the process of mining a block to add to the blockchain. Figure 3 illustrates the block diagram for the entire system of finding the hash and its corresponding nonce described earlier. Figure 4 shows the block diagram for computing the hash value according to the SHA-256 specification. Table 2 summarizes the functionality of each block in the block diagram. Note that the algorithm for computing the SHA-256 hash is provided in the Appendix A.

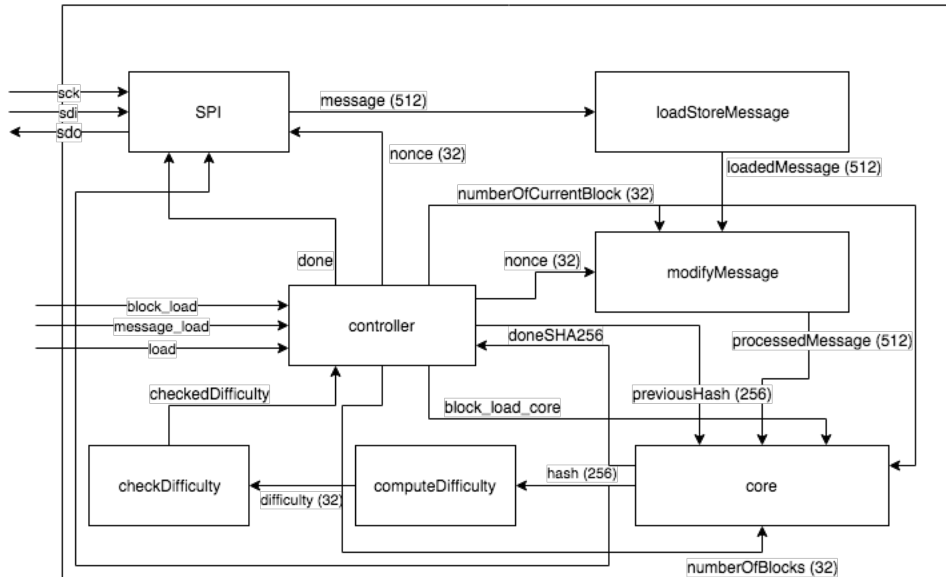


Figure 3: Top Module Block Diagram of System

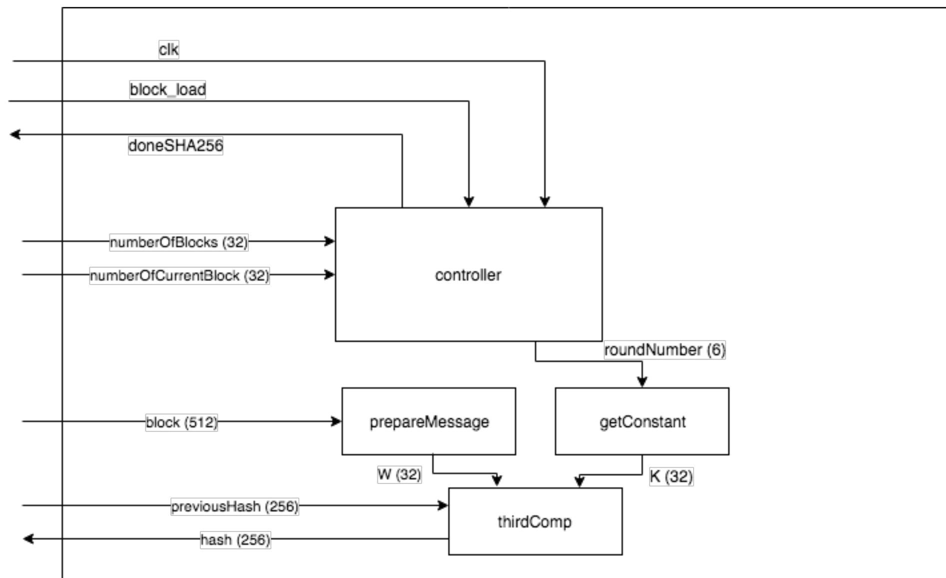


Figure 4: SHA 256 Block Diagram based FIPS180-4

Based on Figure 3, the FPGA operation is controlled by the controller block which receives the following control signals: `message_load`, `block_load`, and `load` which are used to determine when we finish sending the entire message from the Raspberry Pi to the FPGA, when we finish sending a block of 512 bits, and when we finish sending the first block respectively. Based on the control signals, the FPGA sends/receives data using the *SPI* block which is based on the SPI protocol.

When receiving a message block of 512 bits from the Raspberry Pi, the FPGA sends this message to the *loadStoreMessage* block in order to store the message in the RAM. Note that this message will be used later for computing the hash by *core* block after we have finished loading the entire message to the FPGA.

In our design, the last block of message contains 32 bits of the required amount of difficulty and 480 bits of 0s. Note that according to the specification and our encoding, only the last message block can contain this number of 0s.

After having finished receiving the entire message block from the Raspberry Pi, the FPGA starts setting the nonce with 0 and loading the block from the RAM in *loadStoreMessage*. Recall that the representation of a block of the blockchain has a nonce value in the first 32 bits of the first message block. We modify the nonce accordingly in the *modifyMessage* block. After modifying the message, we then compute the hash of the entire message according to the specification [3] in the *core* block, and the algorithm for calculating the hash is described in details in Appendix A.

When the hash calculation is finished, the difficulty of the hash is computed and checked with the required amount of difficulty. The control signal *doneSHA256* is also sent back to the controller. The process of loading the message block, modifying the message block, computing the hash, and computing the difficulty is repeated until the hash meets the difficulty criteria. Once the valid nonce value is found, the controller emits a signal to SPI to initiate some communication with the Raspberry Pi for sending back the hash together with the corresponding nonce.

Module	Purpose
computeDifficulty	Computes the difficulty given the difficulty of a given hash using helper functions
checkDifficulty	Compares the difficulty between the given hash and the given difficulty
roundController	Handles the large state machine for what the other modules do at a time
loadStoreMessage	Stores the RAM containing the 512-bit message blocks from a given message
modifyMessage	Changes the nonce a little bit to increase the message
core	Computes the actual 256-bit hash through N rounds from the specification
spi	Handles SPI communication between Raspberry Pi and FPGA

Table 2: Main Modules Summarized

5 Results and Findings

From our specifications, we have suceeded! We produced a distributed system that can take at least two Raspberry Pis and two FPGAs. In each operation from one Raspberry Pi to another, we are able to make a transaction and verify the hash correctly. Each Raspberry Pi and FPGA looks like Appendix D to allow an arbitrary user to create wallets and addresses to send MicroPCoins from one person to another. At its current state, anyone can personally mine their coins and there is an infinite supply. We may have to stop this in the future.

To turn on our system, we simply need to invoke "`sudo -E python3.6 app.py`", which allows us to use environment variables from non-root users for variables (we stored the IP address and the port in the `.bashrc` file). Then, we have a beautiful interface that pops up and is ready to roll. In addition, since we are using HTTP requests, the Raspberry Pis must be on the same network. At Harvey Mudd, the Wi-Fi and the Ethernet ports are not connected well, so we connect Ethernet cables to our Pi, and we can `ssh` into them by using a VPN from `vpn.claremont.edu` with our student credentials to obtain the same network.

During the process we faced several major challenges particularly in SPI timing, a protocol for flushing our system, and some RAM limitations. In SPI timing, we found that that we violated several timing constraints for our block logic because we were constrained by the limits of the FPGA. Therefore, we had to slow our clock from 40 MHz to 1.25 MHz using a clock divider to resolve our issues changing the accuracy rate of computing the hash and verifying with the Raspberry Pi from 50% to roughly 99%. Fixing timing constraints will be very helpful for any future people who are interested in systems with lots of logic.

Our system protocol was as follows for debugging SPI issues on long hanging:

a. Re-program the board.

b. Recompile the C program.

c. Reset the soft link, which is not permanent on a PI.

d. Remove any current databases

e. Reset the SPI config on the Raspberry Pi.

f. Reboot the system

We also found a RAM limitation in our FPGA logic design. Because we are limited to the Pi and FPGA's logic blocks, the size of our RAM limits our current design of how large an input string can be. Thus, as our blockchain grows from more and more transactions, we will see that our hashing may start to break due to memory constraints.

6 Parts List

Parts	Quantity	Price
Raspberry Pi	2	$70.00
MuddPi Mark IV Board	2	$133.46

Table 3: Parts List. One user needs one Pi and one MuddPi Mark IV board to be fully functional.

References

[1] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. O'Reilly Media, Inc., 2017.

[2] conradoqg. naivecoin. `https://github.com/conradoqg/naivecoin`, 2017.

[3] PUB FIPS. 180-4. *Secure hash standard (SHS)*, 2015.

[4] Internet Research Task Force (IRTF). Edwards-curve digital signature algorithm (eddsa). 2017.

[5] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

Appendices: Summary

Appendix A: SHA-256 Algorithm

Appendix B: Blockchain Protocol

Appendix C: WebServer

Appendix D: User Interface

Appendix E: SystemVerilog Code

Appendix F: SPI code on the PI

Appendix G: Blockchain Code

Appendix H: JSON Blockchain/Transactions

Appendix A: SHA-256 Algorithm

This section summarizes the SHA-256 algorithm as provided by the specification [3]. We define following functions operating on 64-bit values:

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \tag{1}$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{2}$$

$$\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \tag{3}$$

$$\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \tag{4}$$

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \tag{5}$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \tag{6}$$

given that $ROTR^n$ is the circular right shift performed n times and SHR^n is the right shift operation performed n times. Note that any arithmetic operation is performed modulo 2^{32}.

The sequence of sixty-four constant 32-bit words $K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}$ and the initial hash value $H^{(0)}$ are defined in the specification [3]. The algorithm starts by padding a message into 512-bit messages using the instruction provided in the specification. Note that one 512-bit message is treated as one block of the entire message, with 56 chars per message block at max.

Suppose you have N message blocks given by $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. Note that the subscript of message block or hash specifies the 32-bit segment of the message block. For instance, $M_0^{(1)}$ corresponds to the first 32-bits of the message block, and $M_{15}^{(1)}$ corresponds to the last 32-bits of the message block. Algorithm 1 is used to compute the hash of the N message blocks. The hash is then given by $H^{(N)}$.

for $i = 1$ **to** N **do**

1. Set W_t to

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63 \end{cases}$$

2. Initialize the variables as follows:

$$a \leftarrow H_0^{(i-1)} \quad b \leftarrow H_1^{(i-1)} \quad c \leftarrow H_2^{(i-1)} \quad d \leftarrow H_3^{(i-1)}$$
$$e \leftarrow H_4^{(i-1)} \quad f \leftarrow H_5^{(i-1)} \quad g \leftarrow H_6^{(i-1)} \quad h \leftarrow H_7^{(i-1)}$$

3. **for** $t = 1$ **to** *64* **do**

$$T_1 \leftarrow h + \Sigma_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$$
$$T_2 \leftarrow \Sigma_0^{\{256\}}(a) + Maj(a,b,c)$$
$$h \leftarrow g \quad g \leftarrow f \quad f \leftarrow e \quad e \leftarrow d + T_1$$
$$d \leftarrow c \quad c \leftarrow b \quad b \leftarrow a \quad a \leftarrow T_1 + T_2$$

end

4. Compute the ith intermediate hash value $H^{(i)}$: Initialize the variables as follows:

$$H_0^{(i)} \leftarrow a + H_0^{(i-1)} \quad H_1^{(i)} \leftarrow b + H_1^{(i-1)} \quad H_2^{(i)} \leftarrow c + H_2^{(i-1)} \quad H_3^{(i)} \leftarrow d + H_3^{(i-1)}$$
$$H_4^{(i)} \leftarrow e + H_4^{(i-1)} \quad H_5^{(i)} \leftarrow f + H_5^{(i-1)} \quad H_6^{(i)} \leftarrow g + H_6^{(i-1)} \quad H_7^{(i)} \leftarrow h + H_7^{(i-1)}$$

end

Algorithm 1: SHA256 Algorithm

Appendix B: Blockchain Protocol

The underlying data structure and technology of our cryptocurrency is based on the idea of a blockchain that is essentially a continuous growing list of records, called blocks. Blocks are linked together consecutively and secured using cryptography; each block also has a value of hash of its previous block. By principle design, the blockchain is resistant to modification of data because modifying one block requires modification of later blocks up to the latest block to contain correct previous hash values. This section will further explain the blockchain protocol.

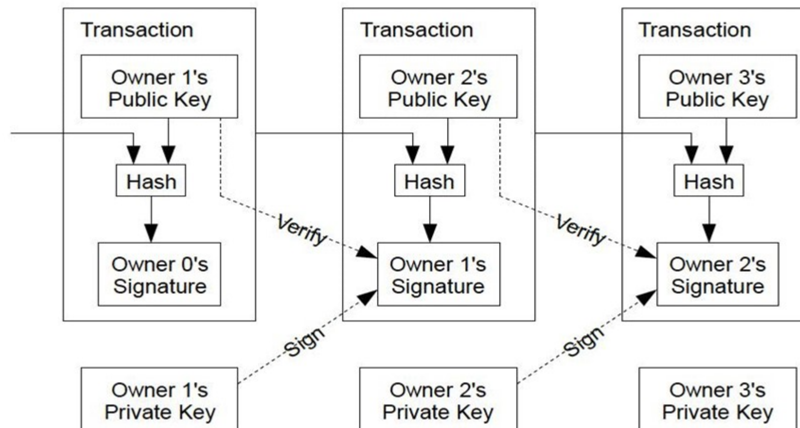


Figure 5: Illustration of blocks of a blockchain. Courtesy of `https://www.cloudtp.com/wp-content/uploads/2016/07/bitcoin-table.gif`

Glossary

- Block - a block contains the index of block, a hash value of the block, the previous hash value of the previous block, and transactions
- Blockchain - a chain of blocks
- Difficulty - a value that determines if it is possible to add a new block to the blockchain
- Hash - a value returned by the hash function like SHA-256
- Nonce - an arbitrary number that is used to compute hash
- Public key - a key shared with everyone; a public key is equal to the id of the address of a wallet; a public key is paired with a private key
- Private key - a key kept secret; a private key is identified by a password; a private key is paired with a public key
- Sign - a process of using the private key to create a signature for a transaction as a proof of the transaction owner's identity
- Signature - a 256-bit text generated by key signing
- SHA-256 - a cryptographic hash algorithm that converts a message into a 256-bit signature for a text based on the specification FIPS 180-4 [3]

- Verify - a process of using a public key and a message (e.g. signature) to verify if the message was created by the intended public key
- Wallet - a wallet contains the set of key pairs of public key and private key, an address, and a amount of money corresponded with the address

A blockchain is a data structure that contains blocks in a linked list order as shown by example in Figure 5. Each block contains information about transactions and its hash value of the previous block. To make a transaction valid for adding to a block, an individual who creates a transaction must also include a signature generated by using the individual's private key and the hash of the transaction. To add a transaction to a block, a miner (who is performing a proof-of-work that will be explained later) can verify that a signature is actually created correctly by the individual who signed the transaction using the signature and the public key. The process of signing and verifying is done using the Edwards-curve Digital Signature Algorithm (EdDSA25519) [4].

In the entire network, each individual has an entire blockchain. By design, it is possible that multiple blockchains are not necessarily the same because everyone has his/her own blockchain. So, we regard the longest blockchain as the blockchain that contains the most correct information. Note that it is possible for an attacker to forge a block and add it to the blockchain in his/her own interest. This attack is commonly known as 51% attack and can be feasible if the attacker has the majority of computation power in the network used to perform the proof-of-work. In our simplified blockchain protocol, we ignore this potential problem and resolve blockchain consensus by simply picking the longest blockchain.

To add a block to the blockchain, we introduce a concept of difficulty, and a block can be added to the blockchain if the block meets the minimum criteria of difficulty. Specifically, the blockchain can deterministically compute the difficulty of the minimum difficulty needed given a block index, e.g. a number of blocks since the genesis block or the first block. The formula for computing difficulty can be defined arbitrarily, but only one constraint is that the difficulty of later blocks must not be smaller than that of earlier blocks. The current formula for computing the minimum difficulty required is

$$\texttt{difficulty}(\texttt{block_idx}) = \begin{cases} 1 & \text{if } \texttt{block_idx} = 1 \text{ or it's a genesis block,} \\ \lfloor \frac{block_idx}{5} \rfloor & \text{otherwise} \end{cases}$$

Next, we need to calculate the difficulty of a block. The difficulty of a block is identified by a the hash value of that block as follows:

$$\texttt{difficulty}(\texttt{block_hash}) = \texttt{number of leading 0s of block_hash in binary representation}.$$

For instance, a block hash of `0x01AA...123` has a difficulty of 7, and a block hash of `0x0FF...000` has a difficulty of 4. Note that the length of block has is always 256-bit, and the probability of getting a block hash at difficulty d is given by 2^{-d}.

To perform the proof-of-work of a block, our goal is to find a block with enough of difficulty to add the blockchain. One field of the block is nonce which is an arbitrary number. According to the design of cryptographic hashing function, changing only one character of the message to be hashed will change the entire hash value to be totally different. Using this advantage, the proof-of-work is performed by trying different nonce values and computing the hash of the block until the difficulty of the block is at least the required difficulty to add to the blockchain.

Appendix C: WebServer

Libraries

1. flask - Python MicroFramework for serving HTTP Requests and endpoints
2. json - Standard library for working with json format
3. hashlib - Standard library for creating dummy SHA-256 hashes
4. pickle - Standard library for serializing objects for speed to save to a file
5. secrets - Random generation of secure numbers
6. requests - Simple wrapper for sending GET and POST requests for each node
7. ed25519[3] - Creates a signing key and verify key for wallet signing

Blockchain Requests

Method	Endpoint	Description
GET	/blockchain/blocks	Get all blocks
GET	/blockchain/blocks/latest	Get the last block
PUT	/blockchain/blocks/latest	Add a new block
GET	/blockchain/blocks/hash/_hash_val_	Get the block by its hash value
GET	/blockchain/blocks/index/_index_val_	Get the block by the index
GET	/blockchain/blocks/transactions/_transactionId_val_	Get the block by the transaction id
GET	/blockchain/transactions	Get all transactions
POST	/blockchain/transactions	Add a transaction
GET	/blockchain/transactions/unspent/_address_	Get all the unspent transactions

Table 4: Blockchain Requests

Method	Endpoint	Description
GET	/operator/wallets	Get all wallets
GET	/operator/wallets/_walletId_	Get the wallet by its id
POST	/operator/wallets/_walletId_/transactions	Get the wallet's transactions
GET	/operator/wallets/_walletId_/addresses	Get the wallet's addresses
POST	/operator/wallets/_walletId_/addresses	Create a new address for the wallet
GET	/operator/wallets/_walletId_/addresses/_addressId_/balance	Get the balance of the addresses

Table 5: Operator Requests

Method	Endpoint	Description
GET	/node/peers	Get all peers
POST	/node/peers	Add a peer
GET	/node/transactions/_transactionId_/confirmations	Get all confirmations

Table 6: Node Requests

Method	Endpoint	Description
POST	/miner/mine	Post relevant address to mine block

Table 7: Miner Requests

[3] https://github.com/warner/python-ed25519

Appendix D: User Interface

Welcome to uPCoin v1.0!

Instructions

Blockchain allows you to look at the data inside the blockchain.

Wallet allows you to operate your wallet, including making transactions.

Node allows you to connect with other *Node*.

Miner allows you to mine a block in order to put to the blockchain.

Functionality

Blockchain

Get Blockchain | Get Last Block | Get Transactions

hash_val | Get Block by Hash Value

index_val | Get Block by Index

transactionId_val | Get Transaction By Id

address | Get Unspent Transactions

Node

Get Peers

peer | Connect to Peers

transactionId | Confirmations of a Transaction Id

Miner

rewardAddress | Crypto Mining

Wallet

Get All Wallets

password | Make a Wallet

walletId | Get Wallet by Id

walletId | password

from | to

amount | changeAddress

Make a Transaction

walletId | password

Generate Address From Wallet

walletId | addressId

Balance of an address

Figure 6: User Interface

Appendix E: SystemVerilog Code

SystemVerilog Main Modules

```
/////////////////////////////////////////////
// uPcoin.sv
// HMC E155 8 December 2017
// hfang@hmc.edu, mjenrungrot@hmc.edu
/////////////////////////////////////////////


/////////////////////////////////////////////
// uPcoin top module
// Connects all the other modules together
// Also stores blockchain difficulty + previous hash
// and new counter
/////////////////////////////////////////////
module uPcoin(input logic  clk,
              input logic  sck,
              input logic  sdi,
              output logic sdo,
              input logic block_load,
              input logic message_load,
              input logic load,
              output logic inputReady,
              output logic doneSPI);

    logic [31:0] currentNonce;
    logic [31:0] blockchainDifficulty, blockDifficulty;
    logic [31:0] numberOfBlocks, numberOfCurrentBlock;
    logic        checkedDifficulty;
    logic        writeEnabled;
    logic        loadDifficulty;
    logic [255:0] previousHash, hash;
    logic [511:0] message, loadedMessage, processedMessage;
    logic         message_start, doneSHA256;
    logic         block_load_core, finalDone;
    logic [31:0] counter;
    logic        counterClk;
    assign counterClk = counter[4];

    // Create a counter to divide the clock speed
    always_ff @(posedge clk)
      counter <= counter + 1;

    // Check if we starting the message forom the load
    always_ff @(posedge counterClk)
      if(load)                      message_start <= 0;
      else                          message_start <= 1;

   // Save the blockchain Difficulty
    always_ff @(posedge counterClk)
      if(inputReady)                blockchainDifficulty <= message[511:480];
      else                          blockchainDifficulty <= blockchainDifficulty;


    // logic for previous hash
    always_ff @(posedge counterClk)
      if(numberOfCurrentBlock == 0) previousHash <= 256'h6a09e667bb67ae853c6ef372a54ff53a510e527f9b05688c1f83d9ab5be0cd19;
      else if(doneSHA256)           previousHash <= hash;
      else                          previousHash <= previousHash;



   // Main Modules
    uPcoin_spi spi(sck, sdi, sdo, doneSPI, message, {hash, currentNonce});

    uPcoin_loadStoreMessage loadstore(counterClk, message, writeEnabled, numberOfCurrentBlock, loadedMessage);
    uPcoin_modifyMessage modify(counterClk, currentNonce, loadedMessage, numberOfCurrentBlock, processedMessage);
    uPcoin_roundController controller(counterClk, message, message_start, block_load, message_load, writeEnabled,
                                      doneSHA256, checkedDifficulty, currentNonce, numberOfBlocks, numberOfCurrentBlock,
                                      loadDifficulty, block_load_core, inputReady, doneSPI);

    uPcoin_core core(counterClk, block_load_core, numberOfBlocks, numberOfCurrentBlock, processedMessage, previousHash, doneSHA256, hash);

    uPcoin_computeDifficulty compute(hash, doneSHA256, blockDifficulty);
```

```
    uPcoin_checkDifficulty check(blockDifficulty, blockchainDifficulty, checkedDifficulty);
endmodule


/////////////////////////////////////////////
// uPcoin roundController
// Handles the FSM for moving between states
// segmented to message handling, SHA256
// generation, and difficulty checking
/////////////////////////////////////////////
module uPcoin_roundController(input logic clk,
                    input logic [511:0] message,
                    input logic message_start,
                    input logic block_load,
                    input logic message_load,
                    output logic writeEnabled,
                    input logic doneSHA256,
                    input logic checkedDifficulty,
                    output logic [31:0] nonce,
                    output logic [31:0] numberOfBlocks,
                    output logic [31:0] numberOfCurrentBlock,
                    output logic loadDifficulty,
                    output logic block_load_core,
                    output logic inputReady,
                    output logic doneSPI);
    typedef enum logic [5:0] {initialization, getMessageSPI, storeMessage, doneStoreMessage, tryNonce,
                    loadMessage, doneLoadMessage, checkDifficulty,
                    runHash, doneHash, foundNonce} statetype;
    statetype state, nextstate;


    always_ff @(posedge clk)
      if(block_load && message_load && ~message_start) state <= initialization;
        else state <= nextstate;

    // inputReady logic
    assign inputReady = (state == doneStoreMessage);

    // block_load_core logic
    assign block_load_core = (state == doneLoadMessage);

    // number of current block logic
    always_ff @(posedge clk)
        if(state == initialization)        numberOfCurrentBlock <= 0;
        else if(state == getMessageSPI)    numberOfCurrentBlock <= numberOfCurrentBlock;
        else if(state == storeMessage)     numberOfCurrentBlock <= numberOfCurrentBlock;
        else if(state == doneStoreMessage) numberOfCurrentBlock <= numberOfCurrentBlock + 1;
        else if(state == tryNonce)         numberOfCurrentBlock <= 0;
        else if(state == loadMessage)      numberOfCurrentBlock <= numberOfCurrentBlock;
        else if(state == doneLoadMessage)  numberOfCurrentBlock <= numberOfCurrentBlock;
        else if(state == runHash)          numberOfCurrentBlock <= numberOfCurrentBlock;
        else if(state == doneHash)         numberOfCurrentBlock <= numberOfCurrentBlock + 1;
        else if(state == foundNonce)       numberOfCurrentBlock <= 0;

    // number of blocks logic
    always_ff @(posedge clk)
      if(state == initialization)                                        numberOfBlocks <= 0;
      else if(state == getMessageSPI)                                    numberOfBlocks <= numberOfBlocks;
      else if(state == storeMessage)                                     numberOfBlocks <= numberOfBlocks + 1;
      else if(state == doneStoreMessage && nextstate == getMessageSPI)  numberOfBlocks <= numberOfBlocks;
      else if(state == doneStoreMessage && nextstate == tryNonce)       numberOfBlocks <= numberOfBlocks - 2;
      else                                                               numberOfBlocks <= numberOfBlocks;

    // nonce logic
    always_ff @(posedge clk)
        if(state == getMessageSPI)                                         nonce <= 0;
        else if(state == storeMessage | state == doneStoreMessage | state == tryNonce |
                state == loadMessage | state == doneLoadMessage | state == runHash| state == doneHash)     nonce <= nonce;
        else if(state == checkDifficulty & nextstate == tryNonce)         nonce <= nonce + 1;
        else if(state == checkDifficulty & nextstate == foundNonce)       nonce <= nonce;
        else if(state == foundNonce)                                       nonce <= nonce;
        else                                                               nonce <= nonce;

    // nextstate logic
    always_comb
        case(state)
            initialization:                                nextstate = getMessageSPI;
```

```
            getMessageSPI:
                if(block_load == 0)                         nextstate = storeMessage;
                else                                        nextstate = getMessageSPI;
            storeMessage:                                   nextstate = doneStoreMessage;
            doneStoreMessage:
                if(message[200:0] == 0)                     nextstate = tryNonce;
                else                                        nextstate = getMessageSPI;
            tryNonce:                                       nextstate = loadMessage;
            loadMessage:                                    nextstate = doneLoadMessage;
            doneLoadMessage:                                nextstate = runHash;
            runHash:
                if(doneSHA256)                              nextstate = doneHash;
                else                                        nextstate = runHash;
            doneHash:
                if(numberOfCurrentBlock < numberOfBlocks)  nextstate = loadMessage;
                else                                        nextstate = checkDifficulty;
            checkDifficulty:
                if(checkedDifficulty)                       nextstate = foundNonce;
                else                                        nextstate = tryNonce;
            foundNonce:                                     nextstate = foundNonce;
        endcase

    assign writeEnabled = (state == storeMessage);
     assign doneSPI = (state == foundNonce);
endmodule



/////////////////////////////////////////////
// uPcoin spi controller
// Handles the messaging the hashAndNonce
// to receive input message blocks and output
// the final hash and the nonce
/////////////////////////////////////////////
module uPcoin_spi(input  logic sck,
                  input  logic sdi,
                  output logic sdo,
                  input  logic done,
                  output logic [511:0] message,
                  input  logic [287:0] hashAndNonce);

  logic         sdodelayed, wasdone;
  logic [287:0] hashcaptured;

  always_ff @(posedge sck)
    if (!wasdone)  {hashcaptured, message} = {hashAndNonce, message[510:0], sdi};
    else           {hashcaptured, message} = {hashcaptured[286:0], message, sdi};

  // sdo should change on the negative edge of sck
  always_ff @(negedge sck) begin
    wasdone = done;
    sdodelayed = hashcaptured[286];
  end

  // when done is first asserted, shift out msb before clock edge
  assign sdo = (done & !wasdone) ? hashAndNonce[287] : sdodelayed;
endmodule



/////////////////////////////////////////////
// uPcoin_loadStoreMessage
// Loads the given message into RAM if we
// have finished reading the message block
/////////////////////////////////////////////
module uPcoin_loadStoreMessage(input logic          clk,
                               input logic [511:0]  message,
                               input logic          writeEnabled,
                               input logic [31:0]   numberOfCurrentBlock,
                               output logic [511:0] loadedMessage);

    logic [511:0] RAM [127:0];
    always_ff @(posedge clk)
    begin
        if(writeEnabled) RAM[numberOfCurrentBlock] <= message;
        loadedMessage <= RAM[numberOfCurrentBlock];
    end
```

```
endmodule


////////////////////////////////////////////
// uPcoin_modifyMessage
// Changes the processedmessage to receive the
// only the first 32 bits corresponding to the nonce
// in the first block.
////////////////////////////////////////////
module uPcoin_modifyMessage(input logic            clk,
                            input logic  [31:0]    currentNonce,
                            input logic  [511:0]   loadedMessage,
                            input logic  [31:0]    numberOfCurrentBlock,
                            output logic [511:0]   processedMessage);

    always_comb
        if(numberOfCurrentBlock == 0) begin
            processedMessage[511:480] = currentNonce;
            processedMessage[479:0] = loadedMessage[479:0];
        end else begin
            processedMessage = loadedMessage;
        end
endmodule


////////////////////////////////////////////
// helperCount0_16
// Counts the number of zeros in some 16 bit number
////////////////////////////////////////////
module helperCount0_16(input logic [15:0] text,
                       output logic [31:0] numZeros);
    always_comb
        if(text[15])       numZeros = 0;
        else if(text[14])  numZeros = 1;
        else if(text[13])  numZeros = 2;
        else if(text[12])  numZeros = 3;
        else if(text[11])  numZeros = 4;
        else if(text[10])  numZeros = 5;
        else if(text[9])   numZeros = 6;
        else if(text[8])   numZeros = 7;
        else if(text[7])   numZeros = 8;
        else if(text[6])   numZeros = 9;
        else if(text[5])   numZeros = 10;
        else if(text[4])   numZeros = 11;
        else if(text[3])   numZeros = 12;
        else if(text[2])   numZeros = 13;
        else if(text[1])   numZeros = 14;
        else if(text[0])   numZeros = 15;
        else               numZeros = 16;
endmodule


////////////////////////////////////////////
// helperCount0_255
// Counts the number of zeros in some 255 bit number
////////////////////////////////////////////
module helperCount0_255(input logic [255:0] text,
                        output logic [31:0] numZeros);
    logic [31:0] tmpOutput [15:0];

   // Create the amount of zeros in each set of 16 bits
    helperCount0_16 a(text[255:240], tmpOutput[15]);
    helperCount0_16 b(text[239:224], tmpOutput[14]);
    helperCount0_16 c(text[223:208], tmpOutput[13]);
    helperCount0_16 d(text[207:192], tmpOutput[12]);
    helperCount0_16 e(text[191:176], tmpOutput[11]);
    helperCount0_16 f(text[175:160], tmpOutput[10]);
    helperCount0_16 g(text[159:144], tmpOutput[ 9]);
    helperCount0_16 h(text[143:128], tmpOutput[ 8]);
    helperCount0_16 i(text[127:112], tmpOutput[ 7]);
    helperCount0_16 j(text[111: 96], tmpOutput[ 6]);
    helperCount0_16 k(text[ 95: 80], tmpOutput[ 5]);
    helperCount0_16 l(text[ 79: 64], tmpOutput[ 4]);
    helperCount0_16 m(text[ 63: 48], tmpOutput[ 3]);
    helperCount0_16 n(text[ 47: 32], tmpOutput[ 2]);
```

```
    helperCount0_16 o(text[ 31: 16], tmpOutput[ 1]);
    helperCount0_16 p(text[ 15:  0], tmpOutput[ 0]);

   // Apply counting of the number of zeros in a 255 bit number
    always_comb
        if(tmpOutput[15] < 16)      numZeros = tmpOutput[15];
        else if(tmpOutput[14] < 16) numZeros = 16  + tmpOutput[14];
        else if(tmpOutput[13] < 16) numZeros = 32  + tmpOutput[13];
        else if(tmpOutput[12] < 16) numZeros = 48  + tmpOutput[12];
        else if(tmpOutput[11] < 16) numZeros = 64  + tmpOutput[11];
        else if(tmpOutput[10] < 16) numZeros = 80  + tmpOutput[10];
        else if(tmpOutput[ 9] < 16) numZeros = 96  + tmpOutput[ 9];
        else if(tmpOutput[ 8] < 16) numZeros = 112 + tmpOutput[ 8];
        else if(tmpOutput[ 7] < 16) numZeros = 128 + tmpOutput[ 7];
        else if(tmpOutput[ 6] < 16) numZeros = 144 + tmpOutput[ 6];
        else if(tmpOutput[ 5] < 16) numZeros = 160 + tmpOutput[ 5];
        else if(tmpOutput[ 4] < 16) numZeros = 176 + tmpOutput[ 4];
        else if(tmpOutput[ 3] < 16) numZeros = 192 + tmpOutput[ 3];
        else if(tmpOutput[ 2] < 16) numZeros = 208 + tmpOutput[ 2];
        else if(tmpOutput[ 1] < 16) numZeros = 224 + tmpOutput[ 1];
        else if(tmpOutput[ 0] < 16) numZeros = 240 + tmpOutput[ 0];
        else                        numZeros = 256;
endmodule


/////////////////////////////////////////////
// uPcoin_computeDifficulty
// Computes the difficulty of a hash by the
// first number of consecutive 0s
/////////////////////////////////////////////
module uPcoin_computeDifficulty(input logic [255:0] hash,
                                input logic         doneSHA256,
                                output logic [31:0] blockDifficulty);
    logic [31:0] tmpDifficulty;
    helperCount0_255 counter(hash, tmpDifficulty);
    always_comb
        if(doneSHA256) blockDifficulty = tmpDifficulty;
        else           blockDifficulty = 32'hffffffff;
endmodule



/////////////////////////////////////////////
// uPcoin_checkDifficulty
// Checks if the difficulty is larger from
// the block or if the blockchain is more
// difficult
/////////////////////////////////////////////
module uPcoin_checkDifficulty(input logic [31:0] blockDifficulty,
                              input logic [31:0] blockchainDifficulty,
                              output logic checkedDifficulty);
    always_comb
        if(blockDifficulty > blockchainDifficulty) checkedDifficulty = 1;
        else checkedDifficulty = 0;
endmodule



/////////////////////////////////////////////
// uPcoin_core
// Generates a hash from a set of 512 bit message blocks
/////////////////////////////////////////////
module uPcoin_core(input logic clk,
                   input logic block_load,
                   input logic [31:0] numberOfBlocks,
                   input logic [31:0] numberOfCurrentBlock,
                   input logic [511:0] block,
                   input logic [255:0] previousHash,
                   output logic doneSHA256,
                   output logic [255:0] hash);


  // Set falling/rising block and message signals;
  logic falling_edge_block;
  // Set up the round numbers and counter
  logic [5:0] roundNumber, messageScheduleCounter;
  // Set up counters for preparing the message schedule
```

```
logic [3:0] counter3, counter2, next14, next6, next1, next15;
// Store the intermediate hash value for future updating
logic [255:0] intermediate_hash;
// 6.2.2 variables and temp variables
logic [31:0] a, b, c, d, e, f, g, h;
logic [31:0] new_a, new_b, new_c, new_d, new_e, new_f, new_g, new_h;
logic [31:0] W[0:15], newW;
logic [31:0] K;

// Set up State transition diagram
typedef enum logic [5:0]{preProcessing, intermediateStep, waiting, thirdStep, doneHashing} statetype;
statetype state, nextstate;

//  Set up the intermediate hash values to change only when returning to the doneHashing or intermediateState steps.
//  Set up initial hashes.
always_ff @(posedge clk)
  if (block_load) begin
  state <= preProcessing;
end
else begin
  state <= nextstate;
  if(state == preProcessing || nextstate == intermediateStep) begin
  intermediate_hash <= previousHash;
  end else if(state == intermediateStep && nextstate == thirdStep) begin
    intermediate_hash <= intermediate_hash;
  end else if(state == waiting && (nextstate == doneHashing || nextstate == intermediateStep)) begin
    intermediate_hash[255:224]  <= a + intermediate_hash[255:224];
    intermediate_hash[223:192]  <= b + intermediate_hash[223:192];
    intermediate_hash[191:160]  <= c + intermediate_hash[191:160];
    intermediate_hash[159:128]  <= d + intermediate_hash[159:128];
    intermediate_hash[127:96]   <= e + intermediate_hash[127:96];
    intermediate_hash[95:64]    <= f + intermediate_hash[95:64];
    intermediate_hash[63:32]    <= g + intermediate_hash[63:32];
    intermediate_hash[31:0]     <= h + intermediate_hash[31:0];
  end else begin
    intermediate_hash <= intermediate_hash;
  end
end

// Set up falling edge block
always_ff @(posedge clk, negedge block_load)
  if(~block_load) falling_edge_block <= 1;
else              falling_edge_block <= 0;

// Increase counters for the number of rounds in 6.2.2
// Set up the intermediate values for the intermediate steps
always_ff @(posedge clk)
  begin
    if (state == intermediateStep)      roundNumber <=0;
    else                                roundNumber <= roundNumber + 1;

    if(state == preProcessing)          messageScheduleCounter <= 0;
    else if(state == intermediateStep)  messageScheduleCounter <= messageScheduleCounter + 1;
    else                                messageScheduleCounter <= messageScheduleCounter;

    if(state == intermediateStep)       counter2 <= 0;
    else if(state == thirdStep)         counter2 <= counter2 + 1;
    else                                counter2 <= counter2;

    // Generate the W values in 6.2.2.1
    if(state == intermediateStep) begin
      W[15] <= block[31:0];
      W[14] <= block[63:32];
      W[13] <= block[95:64];
      W[12] <= block[127:96];
      W[11] <= block[159:128];
      W[10] <= block[191:160];
      W[9] <= block[223:192];
      W[8] <= block[255:224];
      W[7] <= block[287:256];
      W[6] <= block[319:288];
      W[5] <= block[351:320];
      W[4] <= block[383:352];
      W[3] <= block[415:384];
      W[2] <= block[447:416];
      W[1] <= block[479:448];
      W[0] <= block[511:480];
```

```
      end
      else if(state == thirdStep) begin
        if(roundNumber < 15) W <= W;
        else                 W[counter3] <= newW;
      end
    else W <= W;

      // Update the variables in 6.2.2.4
      if(state == thirdStep) begin
        a <= new_a;
        b <= new_b;
        c <= new_c;
        d <= new_d;
        e <= new_e;
        f <= new_f;
        g <= new_g;
        h <= new_h;
      end
    // Fix this, only set the initial hash value on the first block. Another Counter?
    else if(nextstate == intermediateStep) begin
      {a,b,c,d,e,f,g,h} <= previousHash;
    end
    end

  // Set up the next state logic, which depends on the roundNumber and the
  // falling edges of the blocks and full message
  always_comb
    case(state)
      preProcessing:
        if(falling_edge_block)                    nextstate = intermediateStep;
        else                                      nextstate = preProcessing;
      intermediateStep:                           nextstate = thirdStep;
      thirdStep:
        if(roundNumber == 63)                     nextstate = waiting;
        else                                      nextstate = thirdStep;
      waiting:                                    nextstate = doneHashing;
      doneHashing:                                nextstate = doneHashing;
    endcase


  // Prepare the message using newW in 6.2.2.1
  // Generate the K value for each round in 6.2.2.3
  // Apply the transformations in 6.2.2.3
  prepareMessage ppM(W[next1], W[next6], W[next14], W[next15], newW);
  getConstant kHelper(roundNumber, K);
  thirdComp  thirdComputation(a,b,c,d,e,f,g,h,W[counter2],K, new_a, new_b, new_c, new_d, new_e, new_f, new_g, new_h);


  // Increase the counters to match up with 6.2.2.3
  assign counter3 = counter2+1;
  assign next1 = counter2 - 1;
  assign next6 = counter2 - 6;
  assign next14 = counter2 - 14;
  assign next15 = counter2 - 15;

  // Assign final values for completion
  assign doneSHA256 = (state==doneHashing);
  assign hash = intermediate_hash;

endmodule


/////////////////////////////////////////////
// getConstant() function
//   Get the corresponding K value from the sha256constants.txt file
/////////////////////////////////////////////
module getConstant(input logic [5:0] roundNumber,
                   output logic [31:0] K);

  logic [31:0] constant[0:63];

  initial   $readmemh("sha256constants.txt", constant);
  assign K = constant[roundNumber];
endmodule
```

```
////////////////////////////////////////////
// thirdComp() function
//  Apply the thirdComp function given by 6.2.2.3
////////////////////////////////////////////
module thirdComp(input logic  [31:0] a,b,c,d,e,f,g,h,
                 input logic  [31:0] W, K,
                 output logic [31:0] new_a, new_b, new_c, new_d, new_e, new_f, new_g, new_h);
  logic [31:0] T1, T2;
  logic [31:0] tempSigma1, tempSigma0, tempCh, tempMaj;

  SIGMA0 sigma0_temp(a, tempSigma0);
  SIGMA1 sigma1_temp(e, tempSigma1);
  Maj    maj_temp(a,b,c, tempMaj);
  Ch     ch_temp(e,f,g, tempCh);

  assign T1 = h + tempSigma1 + tempCh + K + W;
  assign T2 = tempSigma0 + tempMaj;
  assign new_h = g;
  assign new_g = f;
  assign new_f = e;
  assign new_e = d + T1;
  assign new_d = c;
  assign new_c = b;
  assign new_b = a;
  assign new_a = T1 + T2;

endmodule


////////////////////////////////////////////
// prepareMessage() function
//  generate the message schedule based on 6.2.2.1
////////////////////////////////////////////
module prepareMessage(input logic [31:0] Wprev2, Wprev7, Wprev15, Wprev16,
                      output logic [31:0] newW);
  logic [31:0] output_sigma0;
  logic [31:0] output_sigma1;
  sigma0 s0(Wprev15 , output_sigma0);
  sigma1 s1(Wprev2, output_sigma1);
  assign newW = output_sigma1 + Wprev7 + output_sigma0 + Wprev16;
endmodule


////////////////////////////////////////////
// Ch(x,y,z) function
//   Defined by FIPS 180-4 on Page 10 Section 4.1.2
////////////////////////////////////////////
module Ch(input logic  [31:0] x, y, z,
          output logic [31:0] out);
  assign out = (x & y) ^ (~x & z);
endmodule


////////////////////////////////////////////
// Maj(x,y,z) function
//   Defined by FIPS 180-4 on Page 10 Section 4.1.2
////////////////////////////////////////////
module Maj(input logic  [31:0] x, y, z,
           output logic [31:0] out);
  assign out = (x & y) ^ (x & z) ^ (y & z);
endmodule


////////////////////////////////////////////
// SIGMA_0^{(256)}(x,y,z) function
//   Defined by FIPS 180-4 on Page 10 Section 4.1.2
////////////////////////////////////////////
module SIGMA0(input logic  [31:0] x,
              output logic [31:0] out);
  logic [31:0] ROTR2, ROTR13, ROTR22;
  assign ROTR2  = (x >>  2) | (x << 30);
  assign ROTR13 = (x >> 13) | (x << 19);
  assign ROTR22 = (x >> 22) | (x << 10);
  assign out = ROTR2 ^ ROTR13 ^ ROTR22;
endmodule
```

```
/////////////////////////////////////////////
//  SIGMA_1^{(256)}(x,y,z) function
//    Defined by FIPS 180-4 on Page 10 Section 4.1.2
/////////////////////////////////////////////
module SIGMA1(input logic  [31:0] x,
              output logic [31:0] out);
  logic [31:0] ROTR6, ROTR11, ROTR25;
  assign ROTR6  = (x >>  6) | (x << 26);
  assign ROTR11 = (x >> 11) | (x << 21);
  assign ROTR25 = (x >> 25) | (x << 7);
  assign out = ROTR6 ^ ROTR11 ^ ROTR25;
endmodule


/////////////////////////////////////////////
//  sigma_0^{(256)}(x,y,z) function
//    Defined by FIPS 180-4 on Page 10 Section 4.1.2
/////////////////////////////////////////////
module sigma0(input logic  [31:0] x,
              output logic [31:0] out);
  logic [31:0] ROTR7, ROTR18, SHR3;
  assign ROTR7  = (x >>  7) | (x << 25);
  assign ROTR18 = (x >> 18) | (x << 14);
  assign SHR3   = (x >>  3);
  assign out = ROTR7 ^ ROTR18 ^ SHR3;
endmodule


/////////////////////////////////////////////
//  sigma_1^{(256)}(x,y,z) function
//    Defined by FIPS 180-4 on Page 10 Section 4.1.2
/////////////////////////////////////////////
module sigma1(input logic  [31:0] x,
              output logic [31:0] out);
  logic [31:0] ROTR17, ROTR19, SHR10;
  assign ROTR17 = (x >> 17) | (x << 15);
  assign ROTR19 = (x >> 19) | (x << 13);
  assign SHR10  = (x >> 10);
  assign out = ROTR17 ^ ROTR19 ^ SHR10;
endmodule
```

SystemVerilog Testbench

```
/////////////////////////////////////////////
//  testbench.sv
//    Tests multiple blocks manually generating
//    the block and message_load signal
/////////////////////////////////////////////
module testbench();

// Set the intial logic elements
logic clk, block_load, message_load, done, sck, sdi, sdo, load;
logic [255:0] hash, expected;
logic [511:0] init_message, init_message_2, init_message_3, comb, comb_2, comb_3, comb_4, comb_5,
logic [511:0] init_message_4, init_message_5, init_message_6, init_message_7, comb_6, comb_7, difficulty, diff;
logic [31:0] i, nonce;
logic [287:0] hashNonce;
logic inputReady;
// Create the test device
uPcoin dut(clk, sck, sdi, sdo, block_load, message_load, load, inputReady, done);

// Get the test cases
initial begin
init_message  <=  512'h0000000031393564396164643533336230366162623239643663623963623164653332636537653566616
                        4333034396563323930383639363832646534653538;
init_message_2<=  512'h6438313132313531323233343231365b7b276964273a202761646537643735616131353962373935623866
                        6303537393135623439323039633133343564663130;
init_message_3<=  512'h6539656461653838633863656233636431373062303837662722c202768617368273a20273539363534633
                        7666138356438303336363232353962356565363236;
init_message_4<=  512'h6538393334613932303933303337336632386635623562386236306162646464323434838272c202774797
                        065273a2027726577617264272c202764617461273a;
init_message_5<=  512'h207b27696e70757473273a205b5d2c20276f757470757473273a205b7b27616d6f756e74273a203530303
                        03030303030302c2027616464726573273a202738;
init_message_6<=  512'h3466656563643039336336396265336433343264383937396163633963373030613233356335656164346
                        1323831653166643262353861333438653436313427;
init_message_7<=  512'h7d5d7d7d5d800000000000000000000000000000000000000000000000000000000000000000000000000
                        0000000000000000000000000000000000000000c28;
difficulty    <=  512'h0000000100000000000000000000000000000000000000000000000000000000000000000000000000000
                        0000000000000000000000000000000000000000000;
expected      <=  256'h3be646245fc945547d2f04462f3493e5bee47f40aa6702a7b038661fffd6f051;
end

// generate clock and load signals
initial
forever begin
clk = 1'b0; #5;
clk = 1'b1; #5;
end
initial begin
i = 0;
load = 1'b1;
block_load   = 1'b1;
message_load = 1'b1;
end

// Put the initial message in comb
assign comb = init_message;
assign comb_2 = init_message_2;
assign comb_3 = init_message_3;
assign comb_4 = init_message_4;
assign comb_5 = init_message_5;
assign comb_6 = init_message_6;
assign comb_7 = init_message_7;
assign diff = difficulty;

// Test for the positive edge clock cycles
always @(posedge clk) begin
// Read in values and stop loading after 512 bits
if (i == 512 && (~inputReady))begin
block_load = 1'b0;
load = 1'b0;
end else if (i == 512 && inputReady)begin
block_load = 1'b1;
#1; sdi = comb_2[1023-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (i == 1024 && (~inputReady ))begin
block_load = 1'b0;
```

```
end else if (i == 1024 && inputReady)begin
block_load = 1'b1;
#1; sdi = comb_3[1535-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (i == 1536 && (~inputReady)) begin
block_load = 1'b0;
end else if (i == 1536 && inputReady)begin
block_load = 1'b1;
#1; sdi = comb_4[2047-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if(i == 2048 && ~(inputReady)) begin
block_load = 1'b0;
end else if (i == 2048 && inputReady) begin
block_load = 1'b1;
#1; sdi = comb_5[2559-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if(i == 2560 && ~(inputReady)) begin
block_load = 1'b0;
end else if (i == 2560 && inputReady) begin
block_load = 1'b1;
#1; sdi = comb_6[3071-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if(i == 3072 && ~(inputReady)) begin
block_load = 1'b0;
end else if (i == 3072 && inputReady) begin
block_load = 1'b1;
#1; sdi = comb_7[3583-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if(i == 3584 && ~(inputReady)) begin
block_load = 1'b0;
end else if (i == 3584 && inputReady) begin
block_load = 1'b1;
#1; sdi = diff[4095-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if(i == 4096) begin
block_load = 1'b0;
message_load = 1'b0;
end



if (i < 512) begin
#1; sdi = comb[511-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (512 < i && i < 1024) begin
#1; sdi = comb_2[1023-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (1024 < i && i < 1536) begin
#1; sdi = comb_3[1535-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (1536 < i && i < 2048) begin
#1; sdi = comb_4[2047-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (2048 < i && i < 2560) begin
#1; sdi = comb_5[2559-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (2560 < i && i < 3072) begin
#1; sdi = comb_6[3071-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (3072 < i && i < 3584) begin
#1; sdi = comb_7[3583-i];
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (3584 < i && i < 4096) begin
#1; sdi = diff[4095-i];
```

```
#1; sck = 1; #5; sck = 0;
i = i + 1;
end else if (done && i < 4384) begin
#1; sck = 1;
#1; hashNonce[4383-i] = sdo;
#4; sck = 0;
i = i + 1;

end else if (i == 4384) begin
$display("Stuff %h", hashNonce);
{hash, nonce} = hashNonce;
if (hash == expected)
$display("Testbench ran successfully");
else $display("Error: hash = %h, expected %h", hash, expected);
$stop();
end
$display("i = ", i);
end

endmodule
```

Appendix F: SPI code on the PI

spi.c

```
// spi.c
// hfang@hmc.edu, mjenrungrot@hmc.edu 8 December 2017
// Send data to hardware accelator using SPI

// Libraries
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "EasyPIO.h"

// Find k
unsigned int findK(unsigned int l){
    return (((448 - l -1) % 512) + 512) % 512;
}

// Swap Big + Little Endian
unsigned long long swapBytesOrder(unsigned long long val){
    unsigned long long answer = 0;
    unsigned char buffer[20];
    int i,j;
    for(i=0;i<8;i++){
        buffer[i] = (val & (0xFFULL << (i << 3))) >> (i << 3);
    }
    for(i=7,j=0;i>=0;i--,j++){
        answer |= ((unsigned long long)buffer[i] << (j << 3));
    }
    return answer;
}

// Pad a given input to create input_message
void padding(unsigned char *input, int nBytes, unsigned char* output, int *len){
    unsigned int inputLength = nBytes * 8;
    unsigned int k = findK(inputLength);
    unsigned int outputLength = inputLength  + 1+ k + 64;

    outputLength /= 8;
    memset(output, 0, sizeof(output));

    unsigned long long l = 0;
    int i = 0;
    while(i < nBytes){
        output[i] = input[i];
        output[i+1] = 0x80;
        l += 8LL;
        i++;
    }

    unsigned long long* lengthPosition = (unsigned long long*)(output + ((l + 1 + k) >> 3));
    *lengthPosition = swapBytesOrder(l);
    *len = outputLength;

}

unsigned char difficultyBlock[64];

// Generate the difficulty from the terminal input
void generateDifficulty(unsigned int arg1){
    // convert bytes order
    unsigned short byte1 = (arg1 & 0xff000000);
    unsigned short byte2 = (arg1 & 0x00ff0000);
    unsigned short byte3 = (arg1 & 0x0000ff00);
    unsigned short byte4 = (arg1 & 0x000000ff);

    unsigned int* diffPosition = (unsigned int*)difficultyBlock;
    *diffPosition = (byte1 >> 24) | (byte2 >> 8) | (byte3 << 8) | (byte4 << 24);

}

// Constants
#define MSG_PIN 23
#define BLOCK_PIN 25
#define DONE_PIN 24
#define LOAD_PIN 16
```

```
#define INPUT_RDY_PIN 17

// Function prototypes
unsigned char msg[2000000];
unsigned char output[2000000];
unsigned int arg1;


// Main
int main(int argc, char *argv[]){

    // Create a place to store each block
    unsigned char key[64];

    // Prepare the message by setting it to 0s
    memset(msg, 0, sizeof(msg));

    // Read the input message through a text file
    FILE *fread = fopen("input_message.txt", "rb");
    fseek(fread, 0, SEEK_END);


    // Start the counter and sizes
    int lsize, idxCounter = 0;
    lsize = ftell(fread);
    rewind(fread);


    // Wait for the counter to be less than the size
    while(idxCounter < lsize){
        int first = fgetc(fread);
        msg[idxCounter] = (unsigned)first;
        idxCounter++;
    }


    // Generate a difficulty block to send to the FPGA
    memset(difficultyBlock, 0, sizeof(difficultyBlock));
    arg1 = atoi(argv[1]);
    generateDifficulty(arg1);

    fclose(fread);

    // Pad the message according to FIPS
    int paddingLength, i, nblock=0;
    padding((unsigned char *)msg, lsize, output, &paddingLength);
    int nBlocks = paddingLength / 64;
    int block;

    // Initialize pins for blocks
    pioInit();
    spiInit(150000, 0);
    pinMode(MSG_PIN, OUTPUT);
    pinMode(BLOCK_PIN, OUTPUT);
    pinMode(LOAD_PIN, OUTPUT);
    pinMode(DONE_PIN, INPUT);
    pinMode(INPUT_RDY_PIN, INPUT);

    unsigned char sha256[32];
    unsigned char nonce[4];
    memset(sha256, 0, sizeof(sha256));


    // Start the block sending process.
    // Start with high message and load pins.
    // Turn off the load pin and continue after the
    // first couple blocks. Continue sending blocks
    // and sending on/off signals accordingly.

    for(block=0; block<nBlocks;block++){
        for(i=0;i<64;i++){
            key[i] = output[64*block + i];
        }
        if(block == 0){
            digitalWrite(MSG_PIN, 1);
            digitalWrite(LOAD_PIN, 1);
        }
```

```
        digitalWrite(BLOCK_PIN, 1);

        for(i=0; i< 64; i++)  spiSendReceive(key[i]);

        if(block == 0) digitalWrite(LOAD_PIN, 0);
        digitalWrite(BLOCK_PIN, 0);

        if(block < nBlocks){
            while(!digitalRead(INPUT_RDY_PIN));
        }
    }

    // Send the last difficulty block
    for(i=0;i<64;i++){
        key[i] = difficultyBlock[i];
    }
    digitalWrite(BLOCK_PIN, 1);
    for(i=0; i< 64; i++)  spiSendReceive(key[i]);
    digitalWrite(BLOCK_PIN, 0);
    digitalWrite(MSG_PIN, 0);

    // End the output and wait for the response from the FPGA
    while (!digitalRead(DONE_PIN));

    // Get the Sha256 output hash
    for(i=0; i< 32; i++){
        sha256[i] = spiSendReceive(0);
    }

    // Get the nonce value
    for(i=0; i<4;i++){
        nonce[i] = spiSendReceive(0);
    }

    // Parse the nonce vlaue and output to a file, then output the sha256 value
    int nonceValue = (nonce[0] << 24) | (nonce[1] << 16) | (nonce[2] << 8) | nonce[3];

    FILE *fp = fopen("output.txt", "w");
    fprintf(fp, "%d\n", nonceValue);
    for(i=0; i< 32; i++){
        fprintf(fp, "%02x", sha256[i]);
    }
    fclose(fp);
    return 0;
}
```

hashing.py

```
# hashing.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu
# Helper function used to send written data files to the python blockchain

import subprocess
import time
ITER = 15



def get_spi(input_msg=None, difficulty=0):
    """Get the SPI output by running the C algorithm """
    # Set the input message
    if input_msg is not None:
        with open("input_message.txt", "wb") as f:
            f.write(input_msg)

    hashes = []
    previousHash = None
    i = 0
    counter = 0

    # Run the hashing process several times to gather the correct hash output from the C program
    while i  < ITER:
        # Run the C modules
        try:
            subprocess.run(["sudo", "./call_spi", str(difficulty)], timeout=2)
```

```
    except subprocess.TimeoutExpired:
        pass

    time.sleep(0.2)

    # Read the output from the C program
    with open("output.txt", "r") as f2:
        outputNonce = int(f2.readline())
        outputHash = f2.readline()

    # Wait until we have three of the same values to break
    if outputHash == previousHash:
        counter += 1
    else:
        counter = 0

    if counter == 3:
        print("Final Hash: {:}    nonce = {:}".format(outputHash, outputNonce))
        return outputHash, outputNonce

    hashes.append((outputHash, outputNonce))
    previousHash = outputHash

    f2.close()
    i += 1

# Return the actual hash and nonce from what the most common ones before
(actual_hash, actual_nonce) = max(set(hashes), key =hashes.count)

return  (actual_hash, actual_nonce)
```

Appendix G: Blockchain Code

app.py

```
# app.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

""" Library Imports """
from flask import Flask, jsonify, request, render_template
from flask.json import loads, JSONEncoder

import os
import hashlib
import json
import Block
import Blockchain
import Operator
import Transaction
import Miner
import Node


import ed25519

""" --------------- """
""" --------------- """

uPCoin = Flask(__name__)
blockchain = Blockchain.Blockchain("blockchainDb", "transactionDb")
operator = Operator.Operator('walletDb', blockchain)
miner = Miner.Miner(blockchain, None)
node = Node.Node(os.environ["ip"], os.environ["port"], [], blockchain)

""" Main Page """
@uPCoin.route('/')
def index():
    # TODO: Prettify This Pag
    return render_template("upCoin.html")


""" Blockchain GET/POST requests """

@uPCoin.route('/blockchain/blocks', methods=['GET'])
def get_blocks():
    """ Return all blocks in JSON format """
    response= json.dumps(blockchain.getAllBlocks(), default = lambda o:o.__dict__)
    return response
    return render_template("response.html", response=response)

@uPCoin.route('/blockchain/blocks/latest', methods=['GET', 'PUT'])
def latest_blocks():
    """ GET: Return the latest block in JSON format
        PUT: Adda block to the blockchain """
    if request.method == 'GET':
        response = blockchain.getLastBlock()
        response = json.dumps(response, default = lambda o:o.__dict__)
        return response
        return render_template("response.html", response=response)
    elif request.method == 'PUT':
        # Take in the request
        inputJSON = request.json

        # Create a block for the request
        blockToAdd = Block.Block()
        blockToAdd.index = inputJSON["index"]
        blockToAdd.previousHash = inputJSON["previousHash"]
        blockToAdd.timestamp = inputJSON["timestamp"]
        blockToAdd.nonce = inputJSON["nonce"]
        blockToAdd.transactions = [Transaction.createTransaction(transaction) for transaction in inputJSON["transactions"]]
        blockToAdd.hash = blockToAdd.toHash()

        # Add block
        response = blockchain.addBlock(blockToAdd)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)
```

```
@uPCoin.route('/blockchain/blocks/hash/', defaults={'hash_val':None}, methods=['GET', 'POST'])
@uPCoin.route('/blockchain/blocks/hash/<hash_val>', methods=['GET'])
def get_block_by_hash(hash_val=None):
    """ Return a block by its specified hash """
    if request.method == 'GET':
        response = blockchain.getBlockByHash(hash_val)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)
    elif request.method == 'POST':
        hash_val = request.form["hash_val"]
        response = blockchain.getBlockByHash(hash_val)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)


@uPCoin.route('/blockchain/blocks/index/', defaults={'index_val':None}, methods=['GET', 'POST'])
@uPCoin.route('/blockchain/blocks/index/<index_val>', methods=['GET'])
def get_block_by_index(index_val):
    """ Return a block by its specified index """
    if request.method == 'GET':
        index_val = int(index_val)
        response = blockchain.getBlockByIndex(index_val)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)
    elif request.method == 'POST':
        index_val = int(request.form["index_val"])
        response = blockchain.getBlockByIndex(index_val)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)



@uPCoin.route('/blockchain/blocks/transactions/', defaults={'transactionId_val':None}, methods=['GET', 'POST'])
@uPCoin.route('/blockchain/blocks/transactions/<transactionId_val>', methods=['GET', 'POST'])
def get_transaction(transactionId_val):
    """ Return the latest transaction by its id """
    if request.method == 'GET':
        response = blockchain.getTransactionFromBlocks(transactionId_val)
        response = json.dumps(response, default = lambda o:o.__dict__)
        return response
    elif request.method == 'POST':
        transactionId_val = request.form["transactionId_val"]
        response = blockchain.getTransactionFromBlocks(transactionId_val)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)



@uPCoin.route('/blockchain/transactions', methods=['GET', 'POST'])
def all_transactions(transactionId_val=None):
    """ GET: Return the latest transactions
        POST: Add a transaction """
    if request.method == 'GET':
        response = blockchain.getAllTransactions()
        response = json.dumps(response, default = lambda o:o.__dict__)
        return response

    elif request.method == 'POST':
        transaction = Transaction.createTransaction(request.json)
        try:
            blockchain.getTransactionById(transaction.id)
            return str("already exists")
        except:
            response = blockchain.addTransaction(transaction)
            response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
            return render_template("response.html", response=response)

@uPCoin.route('/blockchain/transactions/unspent/', defaults={'address':None}, methods=['GET', 'POST'])
@uPCoin.route('/blockchain/transactions/unspent/<address>', methods=['GET'])
def get_unspent_transactions(address):
    """ Get the unspent transactions for the address. """
    if request.method == 'GET':
        response = blockchain.getUnspentTransactionsForAddress(address)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)
    elif request.method == 'POST':
        address = request.form["address"]
```

```
        unspentTransaction = blockchain.getUnspentTransactionsForAddress(address)
        response = unspentTransaction
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)


"""
Operator
"""

@uPCoin.route('/operator/wallets', methods=['GET', 'POST'])
def wallets():
    """ GET: Get all wallets
        POST: Create a new Wallet by posting a password """
    if request.method == 'GET':
        response = operator.getWallets()
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)
    elif request.method == "POST":
        if request.data == b'':
            password = request.form["password"]
        else:
            jsonData = json.loads(request.data)
            password = jsonData["password"]
        createdWallet = operator.createWalletFromPassword(password)

        # Create a wallet representation that hides the secret and passwordHash
        walletRepresentation = {}
        walletRepresentation["id"] = createdWallet.id
        walletRepresentation["keypairs"] = createdWallet.keypairs
        response = walletRepresentation
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)


@uPCoin.route('/operator/wallets/', defaults={'walletId':None}, methods=['GET', 'POST'])
@uPCoin.route('/operator/wallets/<walletId>', methods=['GET'])
def getWalletById(walletId):
    """ Get a wallet by the specified ID """
    if request.method == "GET":
        response = operator.getWalletById(walletId)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)
    elif request.method == "POST":
        walletId = request.form["walletId"]
        response = operator.getWalletById(walletId)
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)

@uPCoin.route('/operator/wallets/<walletId>/transactions', methods=['POST'])
def createTransaction(walletId):
    """ Create a Transaction """

    if request.method == "POST":
        # Obtain relevant data:
        #   password, fromAddress, toAddress, amount, changeAddress
        if walletId == "Form":
            walletId = request.form["walletId"]
            password = request.form["password"]
            fromAddress = request.form["from"]
            toAddress = request.form["to"]
            amount = request.form["amount"]
            changeAddress = request.form["changeAddress"]
            if not walletId or not password or not fromAddress or not toAddress or not amount or not changeAddress:
                return "Incorrect Input"
        else:
            jsonData = json.loads(request.data)
            password = jsonData["password"]
            fromAddress = jsonData["from"]
            toAddress = jsonData["to"]
            amount = jsonData["amount"]
            changeAddress = jsonData["changeAddress"]
        # Compute the hash of the provided password
        passwordHash = hashlib.sha256(password.encode('utf-8')).hexdigest()

        # Check if the password hash is the same with the stored password hash
        if not operator.checkWalletPassword(walletId, passwordHash):
```

```
            # TODO: Change to 403 error
            return "Invalid Wallet Password. Try Again."

        # Create a transaction
        newTransaction = operator.createTransaction(walletId, fromAddress, toAddress, amount, changeAddress)

        # Check if the transaction is signed correctly
        newTransaction.check()

        # Add thte transaction to the list of pending transaction
        transactionCreated = blockchain.addTransaction(Transaction.createTransactionObject(newTransaction))

        response = transactionCreated
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)


@uPCoin.route('/operator/wallets/<walletId>/addresses', methods=['GET', 'POST'])
def addressesWallet(walletId):
    """ GET: Get all address of a wallet
        POST: Create a new address for a wallet """
    if request.method == "GET":
        # Get all addresses of a wallet
       response = operator.getAddressesForWallet(walletId)
       response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
       return render_template("response.html", response=response)
    elif request.method == "POST":
        # Create a new address

        # Obtain relevant data:
        #   password
        if walletId == "Form":
            walletId = request.form["walletId"]
            password = request.form["password"]
            print(password)
        else:
            jsonData = json.loads(request.data)
            password = jsonData["password"]

        # Compute the hash of the provided password
        passwordHash = hashlib.sha256(password.encode('utf-8')).hexdigest()

         # Check if the password hash is the same with the stored password hash
        if not operator.checkWalletPassword(walletId, passwordHash):
            # TODO: Change to 403 Error
            return "Wrong Wallet Password"

        newAddress = operator.generateAddressForWallet(walletId)
        response = json.dumps({"address": newAddress})
        return render_template("response.html", response=response)


@uPCoin.route('/operator/wallets/<walletId>/addresses/<addressId>/balance', methods=['GET', 'POST'])
def getBalance(walletId, addressId):
    """ Get the balance of a wallet """
    if request.method == "GET":
        # Get a balance for the specified addressId and walletId
        balance = operator.getBalanceForAddress(addressId)
        response = json.dumps({"balance": balance})
        return render_template("response.html", response=response)
    elif request.method == "POST":
        if walletId == "Form":
            walletId = request.form["walletId"]
            addressId = request.form["addressId"]
        balance = operator.getBalanceForAddress(addressId)
        response = json.dumps({"balance": balance})
        return render_template("response.html", response=response)



"""
Node
"""
@uPCoin.route('/node/peers', methods=['GET', 'POST'])
def peers():
    """ Find the nodes of a peer """
```

```
    if request.method == 'GET':
        response = node.peers
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)
    elif request.method == "POST":
        if request.data == b'':
            print(request.form["peer"])
            newPeer = node.connectWithPeer(request.form["peer"])
        else:
            jsonData = request.json
            newPeer = node.connectWithPeer(jsonData["peer"])
        return str(newPeer)


@uPCoin.route('/node/transactions/<transactionId>/confirmations', methods=['GET', "POST"])
def getConfirmations(transactionId):
    if request.method == 'GET':
        numConfirmations = node.getConfirmations(transactionId)
    elif request.method == 'POST':
        transactionId = request.form["transactionId"]
        numConfirmations = node.getConfirmations(transactionId)
    response = json.dumps({"confirmations" : numConfirmations})
    return render_template("response.html", response=response)

"""
Miner
"""
@uPCoin.route('/miner/mine', methods=['POST'])
def mine():
    """ Mine a new block """
    if request.method == 'POST':
        # Obtain relevant data:
        #   rewardAddress

        if request.data == b'':
            rewardAddress = request.form["rewardAddress"]
        else:
            jsonData = json.loads(request.data)
            rewardAddress = jsonData["rewardAddress"]

        # Mine with the reward address
        newBlock = miner.mine(rewardAddress)

        # When this function call succeeds, it adds the block to the blockchain
        blockchain.addBlock(newBlock);

        # Output the just created block
        response = newBlock
        response = json.dumps(response, default = lambda o:o.__dict__,indent = 4, separators = (',', ': ') )
        return render_template("response.html", response=response)


if __name__=='__main__':
    uPCoin.run(threaded=True, debug=True, host=os.environ["ip"])
```

Blockchain.py

```
# Blockchain.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

import Block
import Transaction
import pickle
import json
import sys
from pyee import EventEmitter

POW_CURVE = 5
EVERY_X_BLOCKS = 5
BASE_DIFFICULTY = 9007199254740991  # Maximum Safe Integer for Javascript
MINING_REWARD = 5000000000

class Blockchain:
    def __init__(self, dbName, transactionsDbName, init=True):
        """
```

```
        Constructor for a blockchain
        """
        if init:
            # Create genesis block
            try:
                temp_block = open(dbName, "rb")
                temp_block = pickle.load(temp_block)
                self.blocks = temp_block
            except OSError as e:
                self.blocks = []
                self.blocks.append(Block.getGenesis())
                pickle.dump(self.blocks, open(dbName, "wb"))
            try:
                temp_transactions = pickle.load(open(transactionsDbName, "rb"))
                self.transactions = temp_transactions
            except OSError as e:
                self.transactions = []
                pickle.dump(self.transactions, open(transactionsDbName, "wb"))
            self.dbName = dbName
            self.transactionsDbName = transactionsDbName
            self.ee = EventEmitter()
        else:
            # Create a blockchain from given files
            blockDb = pickle.load(open(dbName, "rb"))
            transactionsDb = pickle.load(open(transactionsDbName, "rb"))
            self.dbName = dbName
            self.transactionsDbName = transactionsDbName
            self.blocks = blockDb
            self.transactions = transactionsDb

    def __repr__(self):
        """
        String representation of blockchain in JSON format
        """
        # jsonDumper is used for recursively converting an object to correct JSON output format
        def jsonDumper(obj):
            return obj.__dict__
        return json.dumps(self, default=jsonDumper)

    def getAllBlocks(self):
        """
        Return all blocks inside the blockchain
        """
        return self.blocks

    def getBlockByIndex(self, index):
        """
        Return a block specified by the index (integer)

        If the block is not found, ValueError() is raised.
        """
        for block in self.blocks:
            if block.index == index:
                return block
        raise ValueError("Block with index={:} not found".format(index))

    def getBlockByHash(self, hash):
        """
        Return a block specified by the hash (string)

        If the block is not found, ValueError() is raised.
        """
        for block in self.blocks:
            if block.hash == hash:
                return block
        raise ValueError("Block with hash={:} not found".format(hash))

    def getLastBlock(self):
        """
        Return the latest block
        """
        return self.blocks[-1]

    def getDifficulty(self, index):
        """
        Calculate the difficulty based on the index. The difficulty should increase as the number
        of index increases.
```

```
        Note: This formula is taken from
        https://github.com/conradoqg/naivecoin/blob/master/lib/blockchain/index.js

        Note2: Readjust the original formula so that the difficulty is increasing instead of decreasing.
        """
        return index // EVERY_X_BLOCKS

    def getAllTransactions(self):
        """
        Return list of all pending transactions
        """
        return self.transactions

    def getTransactionById(self, id):
        """
        Return a pending transaction by id

        If the transaction is not found, ValueError() is raised.
        """
        for transaction in self.transactions:
            if transaction.id == id:
                return transaction
        raise ValueError("Transaction with id={:} not found".format(id))

    def getTransactionFromBlocks(self, transactionId):
        """
        Return a transaction from all blocks

        If the transaction is not found, ValueError() is raised.
        """
        for block in self.blocks:
            for transaction in block.transactions:
                try:
                    transaction = Transaction.createTransaction(transaction)
                except:
                    pass
                if transaction.id == transactionId:
                    return transaction
        raise ValueError("Transaction with id={:} not found".format(transactionId))

    def replaceChain(self, newChain):
        """
        Replace a current blockchain with the new blockchain. This is done by
        finding the block that starts to diverge. Then, replace all blocks after diversion.

        If the new blockchain is shorter than the current one, ValueError() is raised.
        """

        # Check the length of new blockchain whether it's valid.
        if(len(newChain.blocks) <= len(self.blocks)):
            raise ValueError("New blockchain is shorter than the current blockchain")

        # Check that blocks inside the new blockchain are valid.
        self.checkChain(newChain)

        # Replace blocks after diversion.
        self.blocks = newChain.blocks
        # Emit a blockchain replacement
        self.ee.emit("replacedBlockchain", newChain.blocks)

    def checkChain(self, chain):
        """
        Check if the input blockchain is valid.
        """
        # Check if the genesis block is the same
        if(self.blocks[0].hash != chain.blocks[0].hash):
            raise ValueError("Genesis blocks aren't the same")

        # Check that every two consecutive blocks' hash and previousHash are valid
        for idx in range(len(chain.blocks) - 1):
            self.checkBlock(chain.blocks[idx+1], chain.blocks[idx])

        return True

    def addBlock(self, block):
        """
```

```
    Add a block to the blockchain. Then, save to the blockchain database.

    If the new block is not valid, ValueError() is raised.
    """
    if self.checkBlock(block, self.getLastBlock()):
        self.blocks.append(block)
        pickle.dump(self.blocks, open(self.dbName, "wb"))
        self.removeBlockTransactionsFromTransactions(block)
         # Emit a blockchain replacement
        self.ee.emit("addedBlock", block)
        return block
    else:
        raise ValueError("can't add new block")

def addTransaction(self, transaction):
    """
    Add a transaction to the list of pending transactions. Then, svae to the
    pending transaction database.

    If the new transaction is not valid, ValueError() is raised.
    """
    if self.checkTransaction(transaction):
        self.transactions.append(transaction)
        pickle.dump(self.transactions, open(self.transactionsDbName, "wb"))
        self.ee.emit("addedTransaction", transaction)
        return transaction
    else:
        raise ValueError("can't add new transaction")

def removeBlockTransactionsFromTransactions(self, newBlock):
    """
    Remove all transactions in the new block from the list of pending transactions.
    """


    newtransactions = []
    # Remove any transaction in the pending transaction list that is in the new block.
    for transaction in self.transactions:
        found = False
        for transactionBlock in newBlock.transactions:
            if transaction.id == transactionBlock.id:
                found = True
                continue
        if not found:
            newtransactions.append(transaction)

    # Update transactions object and write to the database
    self.transactions = newtransactions
    pickle.dump(self.transactions, open(self.transactionsDbName, "wb"))


def checkBlock(self, newBlock, previousBlock):
    """
    Check that the new block is valid based on its previous block.
    """

    # Re-calculate the hash of the new block
    newBlockHash = newBlock.toHash()

    if(previousBlock.index + 1 != newBlock.index):
        raise ValueError("Expect new block of id = previous id + 1")
    if(previousBlock.hash != newBlock.previousHash):
        raise ValueError("Expect new block's previous hash to match newBlock.previousHash={:},
        previousBlock.hash={:}".format(newBlock.previousHash, previousBlock.hash))
    if(newBlock.hash != newBlockHash):
        raise ValueError("Expect new block's hash to match the calculation")
    if(newBlock.getDifficulty() <= self.getDifficulty(newBlock.index)):
        raise ValueError("Expect new block's difficulty to be larger \
                         [newBlock.diif = {:}] [{:}]".format(newBlock.getDifficulty(), self.getDifficulty(newBlock.index)))

    # Check that all transacations are valid
    for transaction in newBlock.transactions:
        try:
            self.checkTransaction(Transaction.createTransactionObject(transaction))
        except:
            pass
        try:
```

```
            self.checkTransaction(Transaction.createTransaction(transaction))
        except:
            pass
    # Check the sum of input transactions and output transactions to/from block.
    # The sum of input transactions must be greter than or equal to the sum of
    # output transactions.
    sumOfInputsAmount = 0
    sumOfOutputsAmount = 0
    nfeeTransactions = 0
    nrewardTransactions = 0
    for transaction in newBlock.transactions:
        try:
            transaction = Transaction.createTransaction(transaction)
        except:
            pass
        nfeeTransactions += (transaction.type == "fee")
        nrewardTransactions += (transaction.type == "reward")
        for inputTransaction in transaction.data["inputs"]:
            sumOfInputsAmount += int(inputTransaction["amount"])
        for outputTransaction in transaction.data["outputs"]:
            sumOfOutputsAmount += int(outputTransaction["amount"])

    sumOfInputsAmount += MINING_REWARD
    if(sumOfInputsAmount < sumOfOutputsAmount):
        raise ValueError("Expect sum of input transactions to be greater than the sum of output transactions,
        inputSum={:}, outputSum={:}".format(sumOfInputsAmount, sumOfOutputsAmount))

    if(nfeeTransactions > 1):
        raise ValueError("Expect to have only 1 fee transaction")

    if(nrewardTransactions > 1):
        raise ValueError("Expect to have only 1 reward transaction")

    return True

def checkTransaction(self, transaction):
    """
    Check that the new transaction is valid based on the blockchain, e.g. not already in the blockchain.
    """
    # Check that the transaction, in terms of signature, etc.
    transaction.check()

    # Check if the transaction is already in the blockchain.
    for block in self.blocks:
        for each in block.transactions:
            try:
                each = Transaction.createTransaction(each)
            except:
                pass
            if(each.id == transaction.id):
                raise ValueError("New transaction already exists in the blockchain")

    # Check if the transaction is already spent
    for inputTransaction in transaction.data["inputs"]:
        for block in self.blocks:
            for transaction in block.transactions:
                try:
                    transaction = Transaction.createTransaction(transaction)
                except:
                    pass
                for previousTransaction in transaction.data["inputs"]:
                    if(inputTransaction["index"] == previousTransaction["index"] and\
                       inputTransaction["transaction"] == previousTransaction["transaction"]):
                       raise ValueError("transaction is already spent")

    for pending in self.transactions:
        try:
            pending = Transaction.createTransaction(pending)
        except:
            pass
        if transaction.id == pending.id or transaction == pending:
            raise ValueError("pending transaction exists")
    return True

def getUnspentTransactionsForAddress(self, address):
    """
    Return a list of all unspent transaction identified by the given address.
```

```
        """
        inputs = []
        outputs = []
        # Obtain a list of input/output transactions for the given address
        for block in self.blocks:
            for transaction in block.transactions:
                try:
                    transaction = Transaction.createTransaction(transaction)
                except:
                    pass
                idx = 0
                for transactionOutput in transaction.data["outputs"]:
                    if transactionOutput["address"] == address:
                        transactionOutput["transaction"] = transaction.id;
                        transactionOutput["index"] = idx
                        outputs.append(transactionOutput)
                        idx += 1
                for transactionInput in transaction.data["inputs"]:
                    if transactionInput["address"] == address:
                        inputs.append(transactionInput)

        # Remove any output transaction that is also in the input transactions' list.
        unspentTransactionOutput = []
        for outputTransaction in outputs:
            found = False
            for inputTransaction in inputs:
                if(inputTransaction["transaction"] == outputTransaction["transaction"] and \
                   inputTransaction["index"] == outputTransaction["index"]):
                   found = True
                   break
            if(not found):
                unspentTransactionOutput.append(outputTransaction)
        return unspentTransactionOutput

def createBlockchain(blockchain, blocks):
    newBlockchain = Blockchain(blockchain.dbName, blockchain.transactionsDbName, init=False)
    newBlockchain.blocks = blocks
    newBlockchain.transactions = []

    return newBlockchain
```

Block.py

```
# Block.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

import hashlib
import hashing
import json

class Block:
    def __init__(self):
        """
        Constructor for block
        """
        self.index = 0                  # block index (first block = 0)
        self.previousHash = "0"         # hash of previous block (first block = 0) (512bits)
        self.timestamp = 0              # POSIX time
        self.nonce = 0                  # nonce used to identify the proof-of-work step
        self.transactions = []          # list of transactions inside the block
        self.hash = ""                  # hash taken from the contents of the block:
                                        # sha256(index + previousHash + timestamp + nonce + transactions)

    def __repr__(self):
        """
        String representation of block
        """
        # jsonDumper is used for recursively converting an object to correct JSON output format
        def jsonDumper(obj):
            return obj.__dict__
        return json.dumps(self, default=jsonDumper)

    def getDifficulty(self):
        """
        Return difficulty of the block by counting the number of leading zeros.
        """
```

```
        return 256 - len('{0:b}'.format(int(self.hash, 16)))

    def toHash(self):
        """
        Compute hash of the block
        """
        nonceBytesString = "{0:08X}".format(self.nonce)
        nonceBytes = bytes([int(nonceBytesString[0:2], 16), int(nonceBytesString[2:4], 16),
                            int(nonceBytesString[4:6], 16),  int(nonceBytesString[6:8], 16)])
        strInput = str(self.index) + str(self.previousHash) + str(self.timestamp) + str(self.transactions)
        strInput = strInput.replace("\"","\'")

        print("toHash() BytesString: {:}".format(nonceBytes + strInput.encode('utf-8')))
        return hashlib.sha256(nonceBytes + strInput.encode('utf-8')).hexdigest()

def getGenesis():
    """
    Create a genesis block (the first block).
    """
    block = Block()
    block.index = 0
    block.previousHash = "0"
    block.timestamp = 146515470
    block.nonce = 0
    block.transactions = []
    block.hash = block.toHash()
    return block

def createBlock(data):
    """
    Create a block from JSON object
    """
    block = Block()
    block.index = data["index"]
    block.previousHash = data["previousHash"]
    block.timestamp = data["timestamp"]
    block.nonce = data["nonce"]
    block.transactions = data["transactions"]
    block.hash = block.toHash()
    return block
```

Miner.py

```
# Miner.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

import Block
import Transaction
import hashing
import time
import secrets
import threading
import queue
import sys

FEE_PER_TRANSACTION = 1
MINING_REWARD = 5000000000


class Miner:
    def __init__(self, blockchain, logLevel):
        """
        Constructor for a miner
        """
        self.blockchain = blockchain
        self.logLevel = logLevel
        self.threadQueue = queue.Queue()

    def mine(self, address):
        """
        Wrapper function mining. The reward will be added to the specified when mining a block is done.
        """
        # Create a base block with important information based on the previous block (e.g. hash, etc.)
        baseBlock = self.generateNextBlock(address, self.blockchain.getLastBlock(), self.blockchain.transactions)

        print("base block:\n", baseBlock)
```

```
    print("base block difficulty:\n", baseBlock.getDifficulty())
    # Spawn a thread to perform proof-of-work and wait for output.
    # Wait for the thread to finish before moving on.
    thr = threading.Thread(target=self.proveWorkFor, args=(baseBlock, self.blockchain.getDifficulty(baseBlock.index),))
    thr.start()
    thr.join()

    # Collect the output from the created thread
    output = self.threadQueue.get()
    return output


def generateNextBlock(self, address, previousBlock, blockchainTransactions):
    """
    Create a next block based on its previous block.

    address - The address for the reward of mining
    previousBlock - Block.Block() object of the previous block
    blockchainTransactions - list of all pending transactions
    """

    # The index of the new block must be one after its previous block
    index = previousBlock.index + 1

    # Get the hash of its previous block
    previousHash = previousBlock.hash

    # Get the current timestamp
    timestamp = int(time.time())

    # Get the first 2 pending transactions
    transactions = []
    if len(blockchainTransactions) >= 1:
        transactions.append(blockchainTransactions[0])

    if len(blockchainTransactions) >= 2:
        transactions.append(blockchainTransactions[1])

    # Add fee transaction based on the number of transactions processed
    if len(transactions) > 0:
        data = {}
        data["id"] = secrets.token_hex(32) # 64-characters
        data["hash"] = None
        data["type"] = "fee"
        data["data"] = {
            "inputs": [],
            "outputs": [{
                "amount": FEE_PER_TRANSACTION * len(transactions),
                "address": address
            }]
        }
        feeTransaction = Transaction.createTransaction(data)
        transactions.append(feeTransaction)

    # Add reward transaction
    if address != None:
        data = {}
        data["id"] = secrets.token_hex(32) # 64-characters
        data["hash"] = None
        data["type"] = "reward"
        data["data"] = {
            "inputs": [],
            "outputs": [{
                "amount": MINING_REWARD,
                "address": address
            }]
        }
        rewardTransaction = Transaction.createTransaction(data)
        transactions.append(rewardTransaction)

    # Create a block intended to put to the blockchain
    # Note that nonce field will be dealt later in the proof-of-work process.
    return Block.createBlock({"index": index,
                              "nonce": 0,
                              "previousHash": previousHash,
                              "timestamp": timestamp,
                              "transactions": transactions
```

```
                            })

    def proveWorkFor(self, jsonBlock, difficulty):
        """
        Perform the proof-of-work by changing nonce field.
        """
        blockDifficulty = None
        block = Block.createBlock(jsonBlock.__dict__)

        while True:
            # Get the timestamp
            block.timestamp = int(time.time())

            # Recalculate the hash
            nonceBytesString = "{0:08X}".format(block.nonce)
            nonceBytes = bytes([int(nonceBytesString[0:2], 16), int(nonceBytesString[2:4], 16),
            int(nonceBytesString[4:6], 16),  int(nonceBytesString[6:8], 16)])
            strInput = str(block.index) + str(block.previousHash) + str(block.timestamp) + str(block.transactions)
            strInput = strInput.replace("\"", "\'")
            bytesString = nonceBytes + strInput.encode('utf-8')

            print("BytesString: {:}".format(bytesString))
            (block.hash, block.nonce) = hashing.get_spi(bytesString, difficulty)
            print("block.nonce = {:}".format(block.nonce))

            # Recompute the difficulty of the block
            blockDifficulty = block.getDifficulty()
            print("INFO: blockDifficulty={:} chainDifficulty={:}".format(blockDifficulty, difficulty), file=sys.stderr)

            # Once the block difficulty exceeds the required difficulty, we have proved the block.
            if blockDifficulty > difficulty: break

        # Output back to the wrapper function call
        print("Mined block = {:}".format(str(block)), file=sys.stderr)
        self.threadQueue.put(block)
        return block
```

Node.py

```
# Node.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

import requests
import Blockchain
import Transaction
import Block
import json
import os

class Node:
    def __init__(self, host, port, peers, blockchain):
        self.host = host
        self.port = port
        self.peers = peers
        self.blockchain = blockchain
        self.transmitChain()
        self.connectWithPeers(peers)


    def __repr__(self):
        """
        String representation of blockchain in JSON format
        """
        # jsonDumper is used for recursively converting an object to correct JSON output format
        def jsonDumper(obj):
            return obj.__dict__
        return json.dumps(self, default=jsonDumper)

    def transmitChain(self):
        """
        On event signals, transmit data to other peers
        """
        @self.blockchain.ee.on("replacedBlockchain")
        def data_handler(data):
            if self.peers:
                for peer in self.peers:
```

```
                self.sendLatestBlock(peer, data[-1])
        else:
            print("No peers to send to")
    @self.blockchain.ee.on("addedBlock")
    def data_handler(data):
        if self.peers:
            for peer in self.peers:
                self.sendLatestBlock(peer, data)
        else:
            print("No peers to send to")
    @self.blockchain.ee.on("addedTransaction")
    def data_handler(data):
        if self.peers:
            for peer in self.peers:
                self.sendTransaction(peer, data)
        else:
            print("No peers to send to")
    return

def connectWithPeer(self, peer):
    self.connectWithPeers([peer])
    return peer

def connectWithPeers(self, newPeers):
    """ Connect with the other Raspberry Pi Nodes """
    my_url = "http://{}:{}".format(self.host, self.port)
    for peer in newPeers:
        if (peer not in self.peers) and (peer != os.environ["ip"]):
            self.peers.append(peer)         # add the url to the list of peers
            self.sendPeer(peer, self.host)  # send your own URL
            self.initConnection(peer)       # create a connection with the peer
            # self.broadcast(self.sendPeer, peer)
        else:
            print("Peer already added. No more work needed.")


def initConnection(self, peer):
    """ Create a connection with the latest peer """
    self.getLatestBlock(peer)
    self.getTransactions(peer)

def sendPeer(self, peer, peerToSend):
    """ Tell the other peer that you exist """
    base_url = "http://{:}:{:}/node/peers".format(peer, 5000)
    headers = {'Content-Type' : 'application/json'}
    peerDict = {"peer" : peerToSend}
    r = requests.post(base_url, data = json.dumps(peerDict), headers =headers)
    return r.status_code

def getLatestBlock(self, peer):
    """ Get the latest block from your peer """
    base_url = "http://{}:{}/blockchain/blocks/latest".format(peer, 5000)
    r = requests.get(base_url)
    json_data = r.json()
    received_block = Block.createBlock(json_data)
    self.checkReceivedBlock(received_block)
    return json_data

def sendLatestBlock(self, peer, block):
    """ Send a block to your peer """
    base_url = "http://{}:{}/blockchain/blocks/latest".format(peer, 5000)

    json_output = {}
    json_output["index"] = block.index
    json_output["previousHash"] = block.previousHash
    json_output["timestamp"] = block.timestamp
    json_output["nonce"] = block.nonce


    temp_transactions = []
    for transaction in block.transactions:
        try:
            temp_transactions.append(transaction.__dict__)
        except:
            temp_transactions.append(transaction)
    json_output["transactions"] = temp_transactions
```

```
        headers = {'Content-Type' : 'application/json'}

        r = requests.put(base_url, data = json.dumps(json_output), headers = headers)
        print("Sent Latest Block with error message {}".format(r.status_code))
        return r.status_code

    def getBlocks(self, peer):
        """ Get all the blocks from your peer """
        base_url = "http://{}:{}/blockchain/blocks".format(peer, 5000)
        r = requests.get(base_url)
        json_data = r.json()

        blocks = []
        for block in json_data:
            block = Block.createBlock(block)
            blocks.append(block)

        self.checkReceivedBlocks(blocks)

    def sendTransaction(self, peer, transaction):
        """ Send a transaction from peer to peer using wallet implementation """
        base_url = "http://{}:{}/blockchain/transactions".format(peer, 5000)
        headers = {'Content-Type' : 'application/json'}
        r = requests.post(base_url, data = json.dumps(transaction.__dict__), headers=headers)
        return r.status_code

    def getTransactions(self, peer):
        """ Get transactions from your peers """
        base_url = "http://{}:{}/blockchain/transactions".format(peer, 5000)
        r = requests.get(base_url)
        json_data = r.json()
        transactions = []
        for transaction in json_data:
            transaction = Transaction.createTransaction(transaction)
            transactions.append(transaction)
        self.syncTransactions(transactions)
        print("Done Syncing")

    def getConfirmation(self, peer, transactionID):
        """ Get the confirmation on the transaction ID """
        base_url = "http://{}:{}/blockchain/blocks/transactions/{}".format(peer, 5000, transactionID)
        try:
            r = requests.get(base_url)
            return r.json()
        except:
            # maybe this could be None
            return "Error"


    def getConfirmations(self, transactionID):
        """ Get the confrimation from all of the transactions """
        transactions = self.blockchain.getTransactionFromBlocks(transactionID)
        numConfirmations = 1
        for peer in self.peers:
            if self.getConfirmation(peer, transactionID):
                numConfirmations += 1

        return numConfirmations

    def syncTransactions(self, transactions):
        """ Add missing transactions """
        for transaction in transactions:
            existent = None
            try:
                existent = self.blockchain.getTransactionById(transaction.id)
            except ValueError:
                print("Syncing transaction {}", transaction.id)
                self.blockchain.addTransaction(transaction)


    def checkReceivedBlock(self, block):
        """ Check the received block """
        return self.checkReceivedBlocks([block])

    def checkReceivedBlocks(self, blocks):
        """ Logic for appending and removing incoming blocks """
```

```
        currentBlocks = sorted(blocks, key=lambda x: x.index)
        latestBlockReceived = currentBlocks[len(currentBlocks) - 1]
        latestBlockHeld = self.blockchain.getLastBlock()

        # Don't do anything if the received blockchain is not longer than the actual blockchain
        if latestBlockReceived.index <= latestBlockHeld.index:
            print("---------------------------------")
            print("Received Block is not long enough!")
            print("---------------------------------")
            return False

        if latestBlockHeld.hash == latestBlockReceived.previousHash:
            print("Adding the received block to our chain")
            self.blockchain.addBlock(latestBlockReceived)
        elif len(currentBlocks) == 1:
            print("Query chain from peers")
            for peer in self.peers:
                self.getBlocks(peer)
        else:
            # Received block is longer than current chain, so replace it
            newBlockchain = Blockchain.createBlockchain(self.blockchain, currentBlocks)
            self.blockchain.replaceChain(newBlockchain)
```

Operator.py

```
# Operator.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

import Wallet
import TransactionBuilder
import secrets
import pickle
import hashlib
import json
import binascii

class Operator:
    def __init__(self, dbName, blockChain, init=True):
        self.dbName = dbName
        try:
            temp_wallets = pickle.load(open(dbName, "rb"))
            self.wallets = temp_wallets
        except OSError as e:
            self.wallets = []
            pickle.dump(self.wallets, open(dbName, "wb"))
        self.blockchain = blockChain

    def __repr__(self):
        return json.dumps(self.__dict__)

    def addWallet(self, wallet):
        """ Add a wallet to all the wallets of the operator """
        self.wallets.append(wallet)
        pickle.dump(self.wallets, open(self.dbName, "wb"))
        return wallet

    def createWalletFromPassword(self, password):
        """ Create a wallet from a password """
        hexed = hashlib.sha256(password.encode('utf-8')).hexdigest()
        wallet = Wallet.Wallet(secrets.token_hex(32), hexed)
        return self.addWallet(wallet)

    def createWalletFromHash(self, hashH):
        """ Create a Wallet from the hash """
        wallet = Wallet.Wallet(secrets.token_hex(32), hashH)
        return self.addWallet(wallet)

    def checkWalletPassword(self, walletId, passwordHash):
        """ Check if a wallet exists by its password """
        wallet = self.getWalletById(walletId)
        if wallet == None:
            raise ValueError("Wallet Not Found")

        return wallet.passwordHash == passwordHash
```

```
    def getWallets(self):
        """ Return all the wallets """
        return self.wallets

    def getWalletById(self, walletId):
        """ Get the wallet by its id """
        for wallet in self.wallets:
            if wallet.id == walletId:
                return wallet

        raise ValueError("Wrong wallet id")

    def generateAddressForWallet(self, walletId):
        """ Create address for a wallet """
        # Finde the wallet ID in the data structure
        targetIdx = None
        for idx in range(len(self.wallets)):
            if self.wallets[idx].id == walletId:
                targetIdx = idx

        # Raise an error if not found
        if targetIdx is None:
            raise ValueError("Cannot find address")

        # Generate a new address
        address = self.wallets[targetIdx].generateAddress(walletId)

        # Write to the database
        pickle.dump(self.wallets, open(self.dbName, "wb"))

        # Return a generated address
        return address


    def getAddressesForWallet(self, walletId):
        """ Get addresses contained in a wallet """
        wallet = self.getWalletById(walletId)
        addresses = wallet.getAddresses()
        return addresses

    def getAddressForWallet(self, walletId, addressId):
        """ Get the address for teh wallet by the public key """
        wallet = self.getWalletById(walletId)
        address = wallet.getAddressByPublicKey(addressId)
        return address

    def getBalanceForAddress(self, addressId):
        """ Get the balance of the address """
        utxo = self.blockchain.getUnspentTransactionsForAddress(addressId)
        summed = 0
        for outputTransaction in utxo:
            summed += int(outputTransaction["amount"])

        return summed

    def createTransaction(self, walletId, fromAddressId, toAddressId, amount, changeAddressId):
        """ Create a transaction for a wallet"""

        utxo = self.blockchain.getUnspentTransactionsForAddress(fromAddressId)
        wallet = self.getWalletById(walletId)
        secretKey = wallet.getSecretKeyByAddress(fromAddressId)

        transaction = TransactionBuilder.TransactionBuilder()
        transaction.fromAddress(utxo)
        transaction.to(toAddressId, amount)
        transaction.change(changeAddressId)
        transaction.fee(1)
        transaction.sign(secretKey)
        return transaction.build()
```

Transaction.py

```
# Transaction.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

import hashlib
```

```
import json
import ed25519
import binascii

FEE_PER_TRANSACTION = 1

class Transaction:
    def __init__(self):
        """
        Constructor for a transaction.
        """
        self.id = None
        self.hash = None
        self.type = None
        self.data = {
            "inputs": [],
            "outputs": []
        }

    def __repr__(self):
        """
        String representation for a transaction
        """
        # jsonDumper is used for recursively converting an object to correct JSON output format
        def jsonDumper(obj):
            return obj.__dict__
        return str(json.dumps(self, default=jsonDumper))

    def toHash(self):
        """
        Compute hash for the transaction
        """
        strInput = str(self.id) + str(self.type) + str(self.data)
        return hashlib.sha256(strInput.encode('utf-8')).hexdigest()

    def check(self):
        """
        Check that the transaction is valid, e.g. signed by the correct
        individual.
        """

        # Check if the computed hash matches the transaction's hash.
        transactionHash = self.toHash()
        if(transactionHash != self.hash):
            raise ValueError("hash value of transaction error")

        # Check if the signature of all input transactions are correct.
        for inputTransaction in self.data["inputs"]:
            transactionHash = inputTransaction["transaction"]
            index = inputTransaction["index"]
            amount = inputTransaction["amount"]
            address = inputTransaction["address"]
            signature = inputTransaction["signature"]
            publicKey = address


            # Generate the Message Hash from the transaction hash, index, and address
            messageHash = str(transactionHash) + str(index) + str(address)
            messageHashed = hashlib.sha256(messageHash.encode('utf-8')).digest()

            # Generate the verifying key from the public key
            verifying_key = ed25519.VerifyingKey(binascii.a2b_hex(publicKey))

            # Check if the the signature of the transaction is valid by checking if we can verify the messageHash
            # by the signature.
            verification = None
            try:
                verifying_key.verify(binascii.a2b_hex(signature), messageHashed)
                verification = True
                print("signature is good!")
            except ed25519.BadSignatureError:
                verification = False
                print("signature is bad!")

            # Verify the signed transaction
            if not verification:
                raise ValueError("Signed transaction is invalid by verification process")
```

```
        # Check if the sum of input transactions are enough for the sum of output transactions
        # plus fee.
        if self.type == "regular":
            sumOfInputsAmount = 0
            sumOfOutputsAmount = 0

            for inputTransaction in self.data["inputs"]:
                sumOfInputsAmount += int(inputTransaction["amount"])
            for outputTransaction in self.data["outputs"]:
                sumOfOutputsAmount += int(outputTransaction["amount"])

            if(sumOfInputsAmount < sumOfOutputsAmount):
                raise ValueError("Input amount is less than output amount")

            if((sumOfInputsAmount - sumOfOutputsAmount) < FEE_PER_TRANSACTION):
                raise ValueError("Not enough fee")

        return True

def createTransaction(data):
    """
    Create a transaction from dictionary.
    """
    transaction = Transaction()
    transaction.id = data["id"]
    transaction.type = data["type"]
    transaction.data = data["data"]
    transaction.hash = transaction.toHash()
    return transaction


def createTransactionObject(data):
    """
    Create a transaction from transaction.
    """
    transaction = Transaction()
    transaction.id = data.id
    transaction.type = data.type
    transaction.data = data.data
    transaction.hash = transaction.toHash()
    return transaction
```

TransactionBuilder.py

```
# TransactionBuilder.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

import secrets
import Transaction
import json
import hashlib
import binascii
import ed25519 # API Documentation is available at: https://github.com/warner/python-ed25519

class TransactionBuilder:
    def __init__(self):
        self.listOfUTXO = None
        self.outputAddresses = None
        self.totalAmount = None
        self.changeAddress = None
        self.feeAmount = 0
        self.secretKey = None
        self.typeW = 'regular'

    def __repr__(self):
        """
        String representation for a transaction
        """
        # jsonDumper is used for recursively converting an object to correct JSON output format
        def jsonDumper(obj):
            return obj.__dict__
        return json.dumps(self, default=jsonDumper)

    def fromAddress(self, listOfUTXO):
```

```
        """ Change the from address of the transaction"""
        self.listOfUTXO = listOfUTXO
        return self

    def to(self, address, amount):
        """ Change the outputAddress and total amount of the transaction"""
        self.outputAddress = address
        self.totalAmount = amount
        return self

    def change(self, changeAddress):
        """ Change the changeAddress """
        self.changeAddress = changeAddress
        return self

    def fee(self, amount):
        """ Change the fee of the transaction"""
        self.feeAmount = amount
        return self

    def sign(self, secretKey):
        """ Change secret key of the transaction"""
        self.secretKey = secretKey
        return self

    def type(self, typeT):
        """ Change the type of the transaction """
        self.typeW = typeT

    def build(self):
        """ Build a transaction from the internal parameters"""

        # check for valid transactions by checking
        if self.listOfUTXO == None:
            raise ValueError(" Unspent Output Transactions ")
        elif self.outputAddress == None:
            raise ValueError(" No Output Address ")
        elif self.totalAmount == None:
            raise ValueError(" No Total Amount ")

        # Add up all the amounts from the utxo
        totalAmountOfUTXO = 0
        for output in self.listOfUTXO:
            totalAmountOfUTXO += output["amount"]

        # Remove the transaction amount from the total utxo amounts
        changeAmount =  int(totalAmountOfUTXO) - int(self.totalAmount) - int(self.feeAmount)

        # Generate the inputs
        inputs = []
        for utxo in self.listOfUTXO:
            # Generate the transaction input and calculate the hash/sign the data
            inputStr  = str(utxo["transaction"]) + str(utxo["index"]) + str(utxo["address"])
            txiHash = hashlib.sha256(inputStr.encode('utf-8')).digest()

            # Generate the signing key to sign the transaction
            signing_key, verify_key = self.generateKeyPair(self.secretKey)
            utxo["signature"] = binascii.hexlify(signing_key.sign(txiHash)).decode("utf-8")

            inputs.append(utxo)

        # Generate the outputs
        outputs = []
        outputs.append({"amount":   int(self.totalAmount),
                        "address":  self.outputAddress})

        if changeAmount > 0:
            outputs.append({"amount":   changeAmount,
                            "address":  self.changeAddress})
        else:
            raise ValueError("Sender does not have enough money to  send transaction")

        # return a dictionary of all the values to the create transaction function in Transaction
        buildData = {}
        buildData["id"] = secrets.token_hex(32)
        buildData["hash"] = None
        buildData["type"] = self.typeW
```

```
        inOut = {}
        inOut["inputs"] = inputs
        inOut["outputs"] = outputs
        buildData["data"] = inOut

        return Transaction.createTransaction(buildData)



    def generateKeyPair(self, seed):
            """
            Generate Key Public and Private Key Pairs using the EdDSA algorithm library
            """
            keys = {}

            # Note that seed is expected to be 64 hexadecimal characters
            # Convert 64 hexadecimal characters to 32 bytes
            seed_bytes = binascii.a2b_hex(seed)

            # Create the signing key from 32 bytes
            signing_key = ed25519.SigningKey(seed_bytes)

            # Obtain the verify key for a given signing key
            verify_key = signing_key.get_verifying_key()

            return signing_key, verify_key
```

Wallet.py

```
# Wallet.py
# HMC E85 8 December 2017
# hfang@hmc.edu, mjenrungrot@hmc.edu

import hashlib, binascii
import ed25519 # API Documentation is available at: https://github.com/warner/python-ed25519
import json

class Wallet:
    def __init__(self, wallet_id, passwordHash, secret=None, keypairs=None):
        """
        Wallet Constructor
        - Initialization of Basic Parameters
        """
        self.id = wallet_id
        self.passwordHash = passwordHash
        self.secret = secret
        self.keypairs = []

    def __repr__(self):
        # jsonDumper is used for recursively converting an object to correct JSON output format
        def jsonDumper(obj):
            return obj.__dict__
        return json.dumps(self, default=jsonDumper)

    def generateAddress(self, walletId):
        """
        Generate an Address based on the secret
        """
        if self.secret == None:
            self.generateSecret(self.passwordHash + walletId, wallet_pass=True)

        # Last set of keypairs
        last_key_pair = None
        if not self.keypairs:
            last_key_pair = None
        else:
            last_key_pair = self.keypairs[-1]

        # Set the next seed based on the 1st seed or the last key pair's public key
        if last_key_pair is None:
            seed = self.secret
        else:
            temp_secret = last_key_pair["public_key"]
            seed = self.generateSecret(temp_secret)

        key_pair = self.generateKeyPair(seed)
```

```
    new_key_pair = {"index":        len(self.keypairs) +  1,
                    "secret_key":   key_pair["secret_key"],
                    "public_key":   key_pair["public_key"]}

    self.keypairs.append(new_key_pair)
    return new_key_pair["public_key"]


def generateKeyPair(self, seed):
    """
    Generate Key Public and Private Key Pairs using the EdDSA algorithm library
    """
    keys = {}

    # Note that seed is expected to be 64 hexadecimal characters
    # Convert 64 hexadecimal characters to 32 bytes
    seed_bytes = binascii.a2b_hex(seed)

    # Create a signing key from 32 bytes
    signing_key = ed25519.SigningKey(seed_bytes)

    # Obtain the verify key for a given signing key
    verify_key = signing_key.get_verifying_key()

    # Store secret key of length 128 hexadecimal characters (64 bytes)
    keys["secret_key"] = binascii.hexlify(signing_key.to_bytes()).decode('utf-8')

    # Store public key of length 64 hexadecimal characters (32 bytes)
    keys["public_key"] = binascii.hexlify(verify_key.to_bytes()).decode('utf-8')

    return keys

def generateSecret(self, secret, wallet_pass=False):
    """
    Create a secret using a password hash based on PBKDF2.
    """
    # secretX = hashlib.pbkdf2_hmac('sha256', secret.encode('utf-8'), b'salt', 100000)
    # hexed  = binascii.hexlify(secretX)
    # Hash the secret
    hexed = hashlib.sha256(secret.encode('utf-8')).hexdigest()

    if wallet_pass:
        self.secret = hexed
        return
    else:
        return hexed


def getPublicKeyByAddress(self, index):
    """
    Gather the address by its index
    """
    for wallet in self.keypairs:
        if wallet["index"] == index:
            return wallet["public_key"]

def getAddressByPublicKey(self, public_key):
    #TODO: Throw error
    for wallet in self.keypairs:
        if wallet["public_key"] == public_key:
            return wallet["public_key"]

def getSecretKeyByAddress(self, address):
    """
    Gather the secret key from the address
    """
    for wallet in self.keypairs:
        if wallet["public_key"] == address:
            return wallet["secret_key"]

def getAddresses(self):
    """
    Get all of the addresses
    """
    return [wallet["public_key"] for wallet in self.keypairs]
```

Appendix H: JSON Blockchain/Transactions


```
- {
      index: 1,
      previousHash: "95d9add533b06abb29d6cb9cb1de32ce7e5fad3049ec290869682de4e58d8112",
      timestamp: 1512593105,
      nonce: 3,
    - transactions: [
        - {
              id: "71cb905337030fa8733e2f84786fef0fcbde23aa21451b18e79bc43c1572b9a6",
              hash: "50f1050f994cf94b3e3955e29128c28f3ddac06525f1668340ed4f5514955f05",
              type: "reward",
            - data: {
                  inputs: [ ],
                - outputs: [
                    - {
                          amount: 5000000000,
                          address: "0ec342c73186bfa0888a9678b16964e5d45784298531b23ac04a5bf4c48c8300",
                          transaction: "71cb905337030fa8733e2f84786fef0fcbde23aa21451b18e79bc43c1572b9a6",
                          index: 0,
                          signature: "91e43fa231db2b7cf05b7354ed9796e303fa3100c5274c546076f7ff92d6486607b5e0c78060038f5efbfd1c71a92cbc62c56555f96b327a9ecb9710b49f4d0f"
                      }
                  ]
              }
          }
      ],
      hash: "520541ebf868dd5a85a7d01c18086d5b7623ab2bb75e516422c49f771d3596c9"
  },
```


Figure 7: Block JSON output


```
[
  - {
        id: "d084e05af5c7dd5b26049356b1f0109c103d86492bc6641ccf76437992b0c2a3",
        hash: "bd0e6df6f9660c8263513de218701d612481eb80462a6fd80676b7ed012b67c4",
        type: "regular",
      - data: {
          - inputs: [
              - {
                    amount: 2,
                    address: "a150b8e636f93c104542f41d9e20c27f7b17e21db44088ddc050a99ae5fc203e",
                    transaction: "05c4e4cbe8d7dd34043516da6fe1a940932d125cd2e8107fd31dff8a5192dcba",
                    index: 0,
                    signature: "156eeb06cb367d196008c3b8fabed3c5d5f9c32e36490df44f7d8915bf3e0feb1f878b1fad97f45a36aca4f0ba13e7ef0392d540babc2bd9b7d57a07fec6bc01"
                },
              - {
                    amount: 5000000000,
                    address: "a150b8e636f93c104542f41d9e20c27f7b17e21db44088ddc050a99ae5fc203e",
                    transaction: "b7e4934c30c0dc9e495a3242e3629108a989b053d02d6888bd3f717172ecbdf2",
                    index: 0,
                    signature: "1aa80f2c076f243c2fc581158f69954fc5004566886d278e0098f1f7c177a012914993703321138766c20b3fafdd694179a0db095aa01ec128b065aec5ec4e05"
                },
              - {
                    amount: 5000000000,
                    address: "a150b8e636f93c104542f41d9e20c27f7b17e21db44088ddc050a99ae5fc203e",
                    transaction: "dd0bf03d628c9aae1d7a83dfda5bffe680af865046369c231789f78d3d5e7ed3",
                    index: 0,
                    signature: "f1e14e2c88e538b853c57e90735cc25b0909909d4ec9bd6e31dce95e4dd6072b996f78fea32e22a293cbb7bea41c124a04ee1035926ee66120de03b40f7e0e02"
                }
            ],
          - outputs: [
              - {
                    amount: 100,
                    address: "c20cb9859bc53b2cdbc21c831b329d4486234f4ee1a5941801f1b910dea3a019"
                },
              - {
                    amount: 9999999901,
                    address: "a150b8e636f93c104542f41d9e20c27f7b17e21db44088ddc050a99ae5fc203e"
                }
            ]
        }
    }
]
```


Figure 8: Transaction output