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

Herrick Fang and Teerapat (Mek) Jenrungrot
Email: {hfang,mjenrungrot}@hmc.edu

November 21, 2017

1 Introduction

For our final project, we proposed to create a simplified version of a cryptocurrency based on the blockchain technology in which we use an FPGA to handle proof-of-work. This project prototypes an extensible system that consists of a Raspberry Pi for handling the application interface (e.g. making transactions, operating on the blockchain, etc.) and an FPGA as a hardware accelerator for running costly, time-consuming algorithms to produce hashes that satisfies certain requirements of proof of work. In this report, Section 2 describes the overview design of our system; Section 3 discusses the technical difficulties we have encountered in the project; and Section 4 outlines the specification of our project in terms of what we have finished and planned to do next.

2 Overview Design

This section describes the overview of the design of our system. Figure 2 describes the overview connection with 2 participants and 3 participants respectively. Based on our design, we constrained the number of miners to 2 and the number of users in the network to be at least 2. For the purpose of simplicity of testing our prototype, we will demonstrate a system of 2 users with 2 miners for the demo. Note that a miner is a user, but a user is not necessarily a miner. The miner is represented by an FPGA, and a user is represented by a Raspberry Pi. The work handled by the Raspberry Pi and the FPGA is briefly summarized in the Table 1.

Our cryptocurrency protocol relies on the blockchain protocol. The actual implementation of the Bitcoin protocol on GitHub has approximately 500k lines of code[1]. Replicating the entire Bitcoin system as-is would not be feasible for the scope of our final project. Hence, we apply simplifications to the Bitcoin protocol so that it focuses mostly on digital transaction, especially proof-of-work on the blockchain protocol. The details of our simplified blockchain protocol and background for the technology are provided in Appendix B. Essentially, the major simplification is based on the usage of an HTTP protocol for making connections between any two users, requesting information from other users, and performing operations related to the blockchain. The detailed explanation of operations available over the HTTP server is described in Appendix D.

The main purpose of our hardware is to accelerate the process of proof-of-work, which essentially requires intense computations. To transfer data, we use the SPI protocol for communicating between the Raspberry Pi and the FPGA because of its high reliability and speed. In on our design, the FPGA takes in a block of a blockchain from the Raspberry Pi using the SPI interface. The FPGA then adds a number, called a nonce, to the block and computes the hash using the SHA-256 algorithm [3]. The hash can be used to determine whether a block can be added to the blockchain. If the block cannot yet be added, a new nonce is continuously generated to create a hash that satisfies certain constraint for being able to add to the blockchain.

[1] https://github.com/bitcoin/bitcoin

2.1 Block Diagram

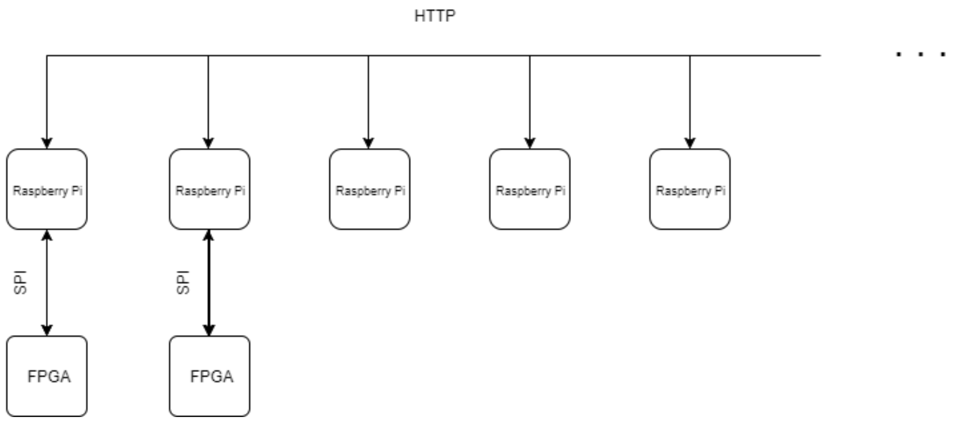

Figure 1: Illustration of blockchain network of Raspberry Pis and FPGAs

2.2 Implementation 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 1. All interfaces over the HTTP protocol are listed in the Appendix D.

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	Compute cryptographic hash for Proof-Of-Work

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

2.2.1 Raspberry Pi

The Raspberry Pi acts as the backbone of our project for handling all HTTP requests related to accessing/modifying the blockchain and making transactions. 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 D, 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 a HTTP GET request to an IP address `142.129.183.125` on port

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

`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`.

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. 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. The process of resolving different multiple blockchains is discussed in Appendix B.

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 blockchain protocol is described in Appendix B.

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. More details of signing and verifying are described in Appendix B.

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 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.

2.2.2 FPGA

The FPGA acts as the central piece of our cryptographic hash functions. In the real world, FPGAs and ASICS are used for Bitcoin mining because they are much faster through the use of parallelization to solve proof-of-work. Thus, our FPGA has an intimate relationship with the Miner process in the Raspberry Pi. The hash function is used for the generations of many addresses, IDs, and the like.

We have written code in SystemVerilog to apply the SHA-256 algorithm to messages based on FIPS 180-4 [3]. The SHA-256 algorithm is a one-way hash function to provide a condensed representation of a message. It is divided into pre-processing and the hash computation. In the pre-processing stage, we pad the message by adding bits to the length of the message such that it satisfies an equivalence relation, parse the message into message blocks to add more difficulty in larger messages, and set the initial hash value based on pre-defined values. In the hash computation stage, we prepare a message schedule from the message and several previous functions/constants. This is done for the N message blocks from the pre-processing stage. We apply several transformations on the data to produce the SHA-256 representation of the value. In addition, we communicate with the Raspberry Pi using the FPGA through the SPI protocol. Since we want to produce a difficult block, we allow the FPGA to take control of the SHA-256 algorithm until it finds a hash together with a nonce value that satisfies our difficulty equation shown in Appendix B.

3 Technical Difficulties

In our current implementation, the Raspberry Pi's Miner sends a block to the FPGA to get the hash value. Note that the block will have a nonce value used to determine if the block can be added to the blockchain. So, if the nonce has to be regenerated, our current implementation requires the Raspberry Pi to send a new block to the FPGA to get a new hash. In the new block sent to the FPGA, most information of that block will remain the same, such as transactions, block ID, etc. The only difference is the nonce value. To improve this implementation, we want to send only one message at the beginning to the FPGA. Then, the nonce will instead be generated inside the FPGA. Once the proof-of-work is done, the nonce together with the hash will be sent back to the Raspberry Pi. Our main technical difficulty is that we are not sure about the best way to create this implementation in our FSM as specified earlier.

4 Specification

Task	Progress
FPGA	
Implement SHA-256 algorithm according to the specification	Finished
Create a testbench with message of one block and Test on Modelsim	Finished
Create a testbench with message of more than one block and Test on Modelsim	In progress
Raspbery Pi	
Implement *HTTP Server* component and Test corresponding HTTP requests	Finished
Implement *Blockchain* component and Test corresponding HTTP requests	Finished
Implement *Miner* component and Test corresponding HTTP requests	Finished
Implement *Operator* component and Test corresponding HTTP requests	Finished
Implement *Node* component and Test corresponding HTTP requests	Finished
Test Peer-to-peer connection between two different Raspberry Pis using the created HTTP requests	Finished
FPGA + Raspbery Pi	
Test connections between the FPGA and Raspberry Pi	Finished
Redirect all the proof-of-work in *Miner* to the FPGA	In progress
Create a benchmark and test the full system	In Progress

Table 2: Summary of tasks categorized by components

Table 2 summarizes the tasks we need to do or tasks we have done to meet the specification. We have separated the tasks into three main categories. First, for the FPGA, we have implemented the SHA-256 algorithm according to the specification [3] correctly. Two testbench files, one with only one message block and another one with more than one message block, were created in order to test our implementation; we have tested our implementation with one message block. Second, for the Raspberry Pi, we have finished implementing 5 core components to the system *HTTP Server*, *Blockchain*, *Miner*, *Operator*, and *Node*. All HTTP requests as described in Appendix D are tested. As of the current system, two Raspberry Pis are able to connect, make transaction, perform the mining on the Pi, and request information from the other Raspberry Pi. Finally, for the FPGA and the Raspberry Pi, we haven't finished the tasks specified in the table yet.

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.

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
        $$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$
    2. Initialize the variables as follows:
        $a \leftarrow H_0^{(i-1)}$   $b \leftarrow H_1^{(i-1)}$   $c \leftarrow H_2^{(i-1)}$   $d \leftarrow H_3^{(i-1)}$
        $e \leftarrow H_4^{(i-1)}$   $f \leftarrow H_5^{(i-1)}$   $g \leftarrow H_6^{(i-1)}$   $h \leftarrow H_7^{(i-1)}$
    3. for $t = 1$ to $64$ do
            $T_1 \leftarrow h + \Sigma_1^{\{256\}}(e) + Ch(e, f, g) + K_t^{\{256\}} + W_t$
            $T_2 \leftarrow \Sigma_0^{\{256\}}(a) + Maj(a, b, c)$
            $h \leftarrow g$   $g \leftarrow f$   $f \leftarrow e$   $e \leftarrow d + T_1$
            $d \leftarrow c$   $c \leftarrow b$   $b \leftarrow a$   $a \leftarrow T_1 + T_2$
       end
    4. Compute the ith intermediate hash value $H^{(i)}$: Initialize the variables as follows:
        $H_0^{(i)} \leftarrow a + H_0^{(i-1)}$   $H_1^{(i)} \leftarrow b + H_1^{(i-1)}$   $H_2^{(i)} \leftarrow c + H_2^{(i-1)}$   $H_3^{(i)} \leftarrow d + H_3^{(i-1)}$
        $H_4^{(i)} \leftarrow e + H_4^{(i-1)}$   $H_5^{(i)} \leftarrow f + H_5^{(i-1)}$   $H_6^{(i)} \leftarrow g + H_6^{(i-1)}$   $H_7^{(i)} \leftarrow h + H_7^{(i-1)}$
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.

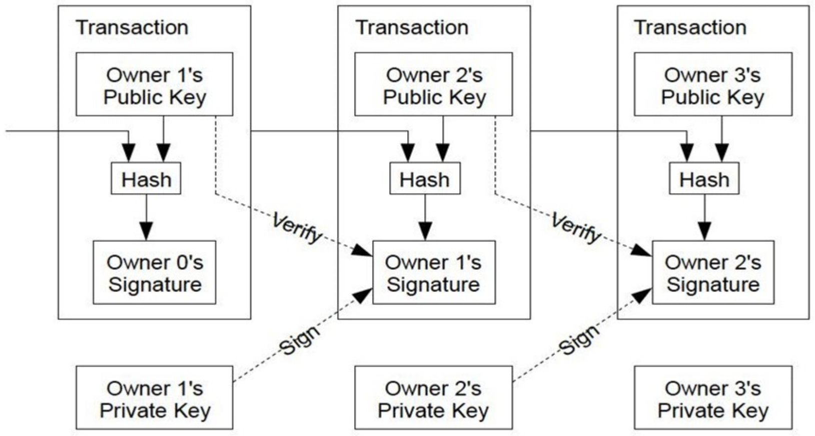

Figure 2: Illustration of blocks of a blockchain. Courtesy of `https://www.cloudtp.com/wp-content/uploads/2016/07/bitcoin-table.gif`

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 2. 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

$$\texttt{difficulty}(\texttt{block\_idx}) = \begin{cases} 1 & \text{if } \texttt{block\_idx} = 1 \text{ or it's a genesis block,} \\ \lfloor \frac{\texttt{block\_idx}}{2} \rfloor + 1. & \text{otherwise} \end{cases}$$

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:

$$\texttt{difficulty}(\texttt{block\_hash}) = \texttt{number of consecutive 0s since the beginning of the hashinbinary}.$$

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: Schematics

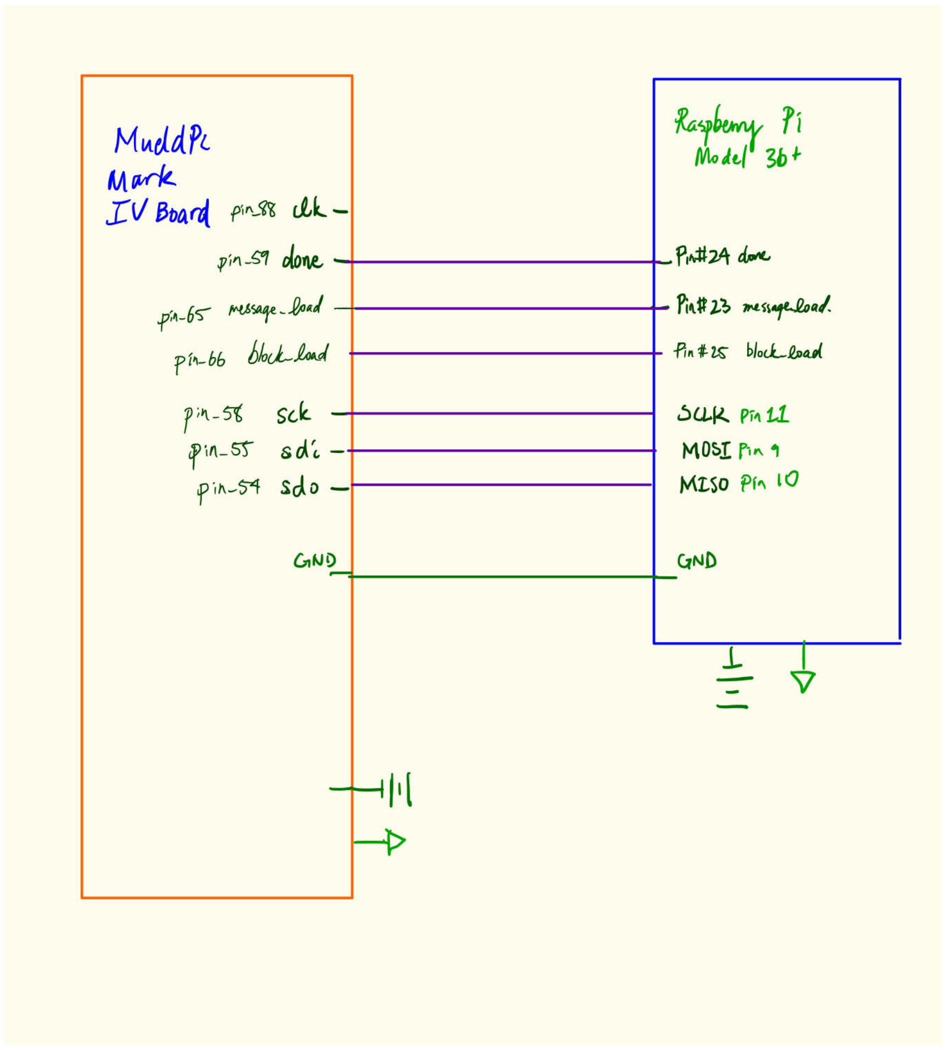

Figure 3: Schematic

Appendix D: 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 3: 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 4: 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 5: Node Requests

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

Table 6: Miner Requests

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

Appendix E: Finite State Machine, WaveForms

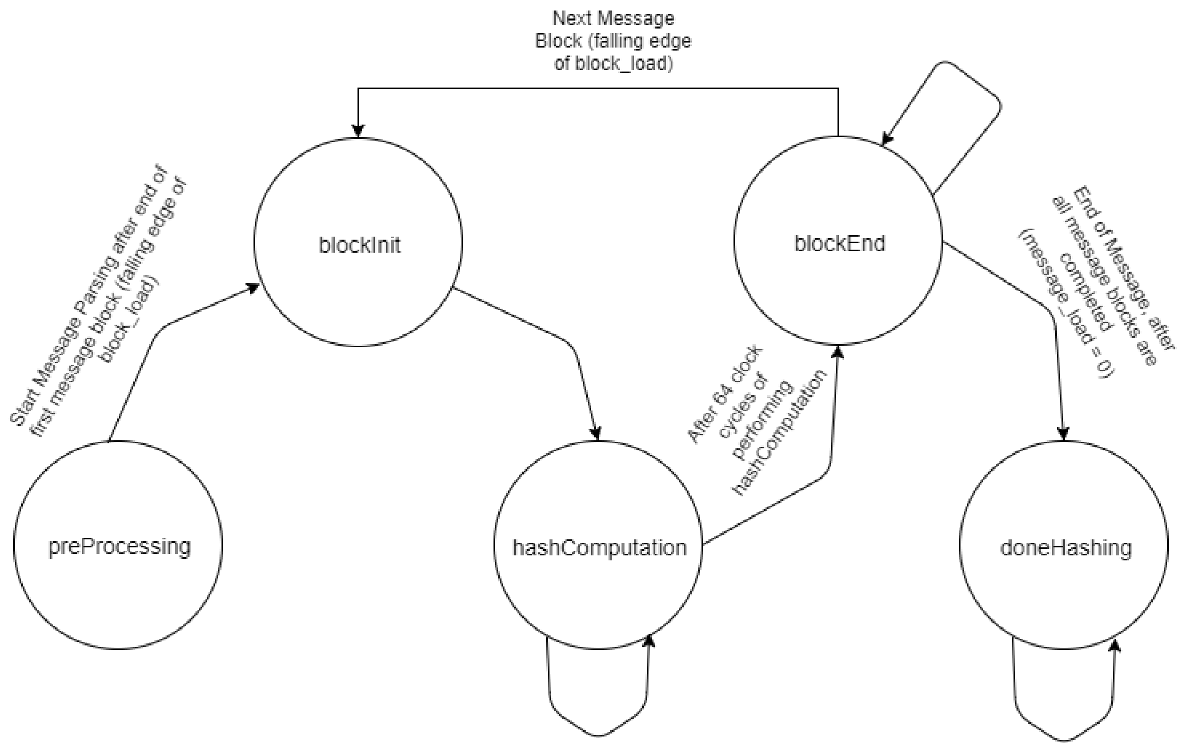
Next Message Block (falling edge of block_load)
Start Message Parsing after end of first message block (falling edge of block_load)
blockInit
blockEnd
End of Message, after all message blocks are completed (message_load = 0)
After 64 clock cycles of performing hashComputation
preProcessing
hashComputation
doneHashing

Figure 4: Finite State Machine

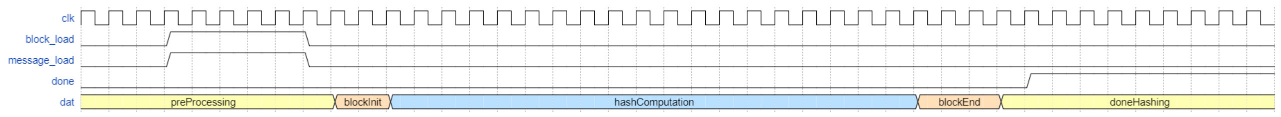

Figure 5: Wave Forms

In our FSM and waveforms, we enter into a preProcessing stage that waits for the first message block's falling edge. We create our modules such that there are message loads and block loads. Since a message is divided into message blocks of 512 bits each, we wait for the end of the first message block to start our initialization. In the example above, there is only one message block and thus the block load and message load end at the same time. When we notice a falling edge of block load, we start the block initialization and continue to hashComputation. Block Initializition sets up the required parameters before This cycles for 64 rounds to complete the SHA hash computation. Then, it moves to the blockEnd state, which determines if it is necessary to take in more message blocks or finish in a doneHashing stage. In addition, the doneHashing state causes the done signal to go to 1.

Appendix F: SystemVerilog Code

```
/*
 * uPcoin main module based on FIPS 180-4 for SHA 256
 * Herrick Fang and Teerapat (Mek) Jenrungrot
 * 11/17/2017
 *
 */
module uPcoin(input logic  clk,
              input logic  sck,
              input logic  sdi,
              output logic sdo,
              input logic  block_load,
              input logic  message_load,
              output logic done);

  logic [255:0] hash;
  logic [511:0] message;

  uPcoin_spi  spi(sck, sdi, sdo, done, message, hash);
  uPcoin_core core(clk, block_load, message_load, message, done, hash);

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



/*
 *  uPcoin_spi function
 *  Sets up the transfer rate for the SPI protocol
 *
 */
```

```
module uPcoin_spi(input  logic sck,
                  input  logic sdi,
                  output logic sdo,
                  input  logic done,
                  output logic [511:0] message,
                  input  logic [255:0] hash);

  logic         sdodelayed, wasdone;
  logic [255:0] hashcaptured;

  always_ff @(posedge sck)
    if (!wasdone)  {hashcaptured, message} = {hash, message[510:0], sdi};
  else           {hashcaptured, message} = {hashcaptured[254:0], message, sdi};

  // sdo should change on the negative edge of sck
  always_ff @(negedge sck) begin
    wasdone = done;
    sdodelayed = hashcaptured[254];
  end

  // when done is first asserted, shift out msb before clock edge
  assign sdo = (done & !wasdone) ? hash[255] : sdodelayed;
endmodule


/*
 *  uPcoin_core function
 *  Generates a hash from a set of 512 bit message blocks
 *
 */
module uPcoin_core(input logic clk,
                   input logic block_load,
                   input logic message_load,
                   input logic [511:0] block,
                   output logic done,
                   output logic [255:0] hash);

  // Set falling/rising block and message signals;
  logic falling_edge_block, rising_edge_block, falling_edge_message;
  // Set up the round numbers counter
  logic [5:0] roundNumber;
  // Set up counters for preparing the message schedule
  logic [3:0] counter3, counter2, next14, next9, 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, blockInit, blockEnd, hashComputation, 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, posedge message_load)
  if (message_load) state <= preProcessing;
else begin
  state <= nextstate;
  if(nextstate == blockInit) begin
    {a,b,c,d,e,f,g,h} <= 256'h6a09e667bb67ae853c6ef372a54ff53a510e527f9b05688c1f83d9ab5be0cd19;
    intermediate_hash <= 256'h6a09e667bb67ae853c6ef372a54ff53a510e527f9b05688c1f83d9ab5be0cd19;
  end else if(state == blockInit && nextstate == hashComputation) begin
    intermediate_hash <= intermediate_hash;
  end else if(state == blockEnd && nextstate == doneHashing) 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 if(state == blockEnd && nextstate == blockInit) 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 == blockInit)      roundNumber <= 0;
    else                                roundNumber <= roundNumber + 1;

    if(state == blockInit)       counter2 <= 0;
    else if(state == hashComputation)         counter2 <= counter2 + 1;
    else                                counter2 <= counter2;

    // Generate the W values in 6.2.2.1
    if(state == blockInit) 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 == hashComputation) begin
      if(roundNumber < 16) W <= W;
      else                W[counter3] <= newW;
    end

    // Update the variables in 6.2.2.4
    if(state == hashComputation) 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
  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 = blockInit;
    else                                   nextstate = preProcessing;
    blockInit:                      nextstate = hashComputation;
```

```
      hashComputation:
        if(roundNumber == 63)               nextstate = blockEnd;
      else                                  nextstate = hashComputation;
      blockEnd:
        if(message_load == 0)               nextstate = doneHashing;
      else if(falling_edge_block == 1)      nextstate = blockInit;
      else                                  nextstate = blockEnd;
      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[snext15], 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 done = (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
```

Appendix G: Blockchain Code

Please see `https://github.com/fangherk/MicroPCoin` since the code is somewhat lengthy.