E4840 Project Report

Jules Comte[1], Mingyang Song[1], Tianyu Qin[1], Xueji Zhao[1], and Zhe Mo[1]

[1]Department of Electrical Engineering, Columbia University

May 2023

1 Project Overview

1.1 Introduction

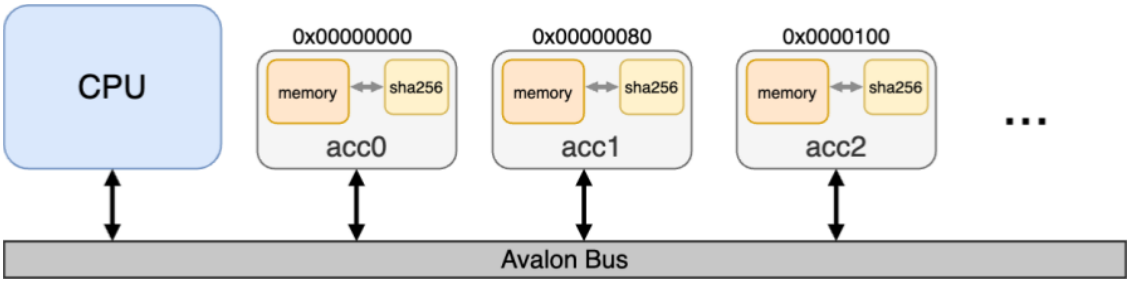
Bitcoin is the first and most widely used cryptocurrency. It is proposed by pseudonymous Satoshi Nakamoto in "Bitcoin: A Peer-to-Peet Electronic Cash System". Bitcoin is a completely digital currency built upon computer science, cryptography, and economics. Bitcoin has various properties, such as decentralized, pseudonymous, immutable, trustless, and transparent [1]. In this project, a Bitcoin miner is introduced to do the Bitcoin mining process through a connection with the mining pool.

1.2 Process of Bitcoin Mining

The overall project has been split into three parts. Firstly, based on the stratum V1 protocol, **Hard Processor System (HPS)** will be implemented to build up the communication between **Mining Pool** and **Bitcoin Miner Unit**. After the mining request is sent to the mining pool, a message block containing a variety of information fields will be pulled down to the memory block of the board. A message block called "**coinbase**" will be created

then based on some parts of the previous message block pulled down from the mining pool. After doing the **message padding** to expand the length of the message block to 1024-bit, the data will be sent to **accelerator** through Avalon Bus to accomplish **Double SHASH-256 Algorithm** and calculate the **Merkle Root**. To use the resource more efficiently, the method of **Pipelining** with three accelerators is introduced in the design. Each accelerator contains the memory block and the SHA-256 module. After the result is calculated from the accelerator, it will be sent back to the mining pool to check whether the calculated result is correct or not.

- Block Diagram of Bitcoin Mining Process



2 Detailed Project Design Documents

2.1 Detailed Description of The Project

This project aims to develop a Bitcoin miner based on the DE1-SoC board. The SHA256 hashing algorithm used in Bitcoin mining will be performed based on the FPGA. The ARM-based Hard Processor System (HPS) will be responsible for communicating between the Bitcoin miner and the Bitcoin mining pool by implementing and running the Stratum v1 protocol.

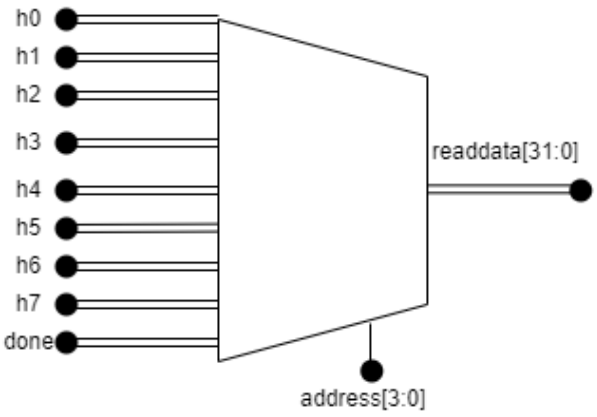
The Stratum protocol is a popular mining protocol used in the Bitcoin network, and it allows the miner to communicate with the mining pool, receive work units and submit solutions. The HPS will be responsible for handling the network communication, parsing the received data and sending it to the FPGA for processing.

The FPGA implementation of the SHA256 algorithm is highly efficient and can perform faster calculations than a traditional CPU. By using an FPGA, the hash rate and the possibility of successfully mining a block can be potentially improved.

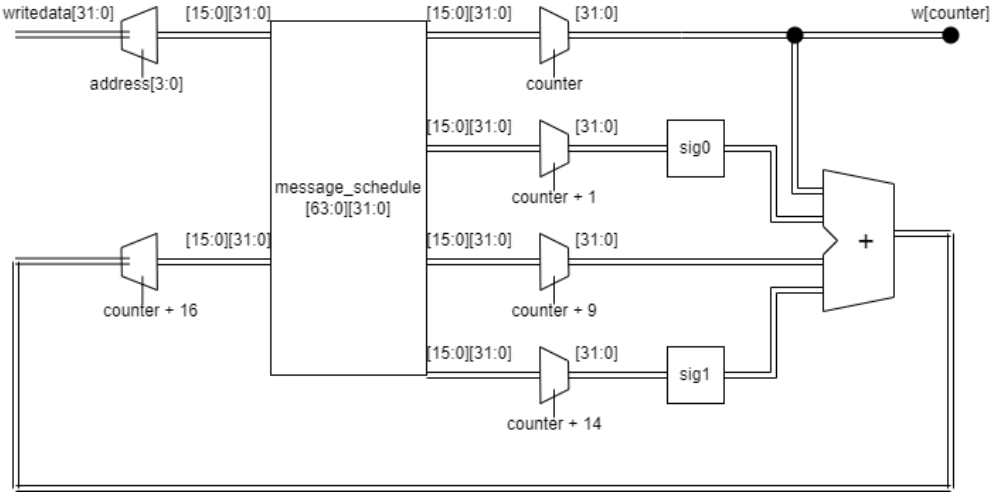
We have two hardware versions of the project. One is mainly created by Jules Comte, and another is mainly created by Mingyang Song. They both want to spend 1000% effort and enthusiasm on this project, so they create their own modules, called Multi-Devices and Multi-Hardware seperately.

2.2 Block Diagram

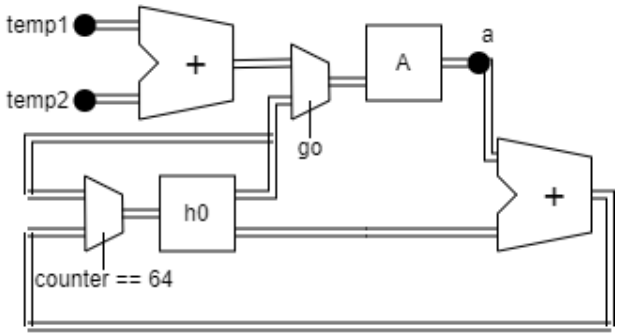
- Read the data from the Avalon bus to local register



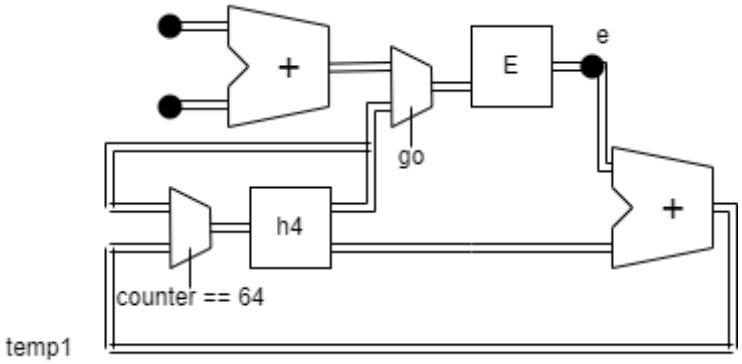
- Message Scheduler



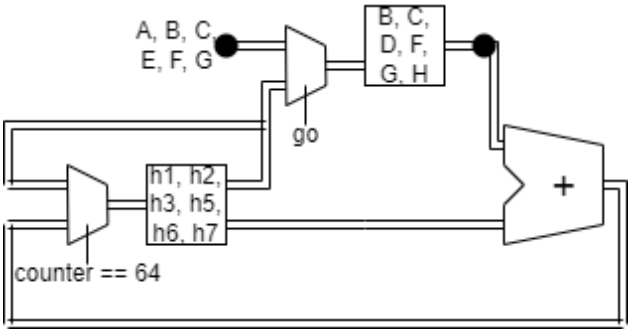
- Message Compression: update A and add H_0 when finish



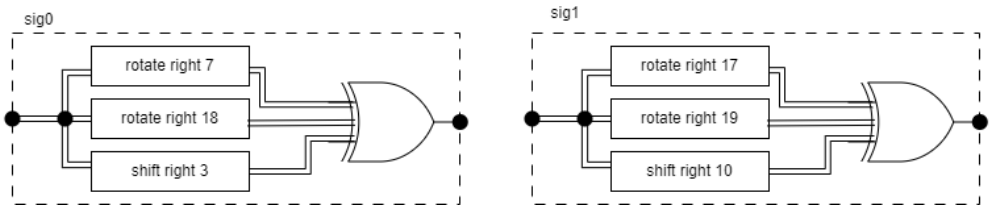
- Message Compression: update E and add H_4 when finish



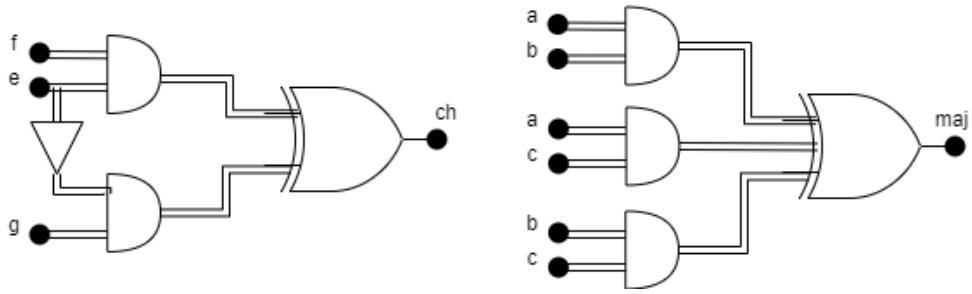
- Message Compression: update B, C, D, F, G, H and add initial Message Digest when finish



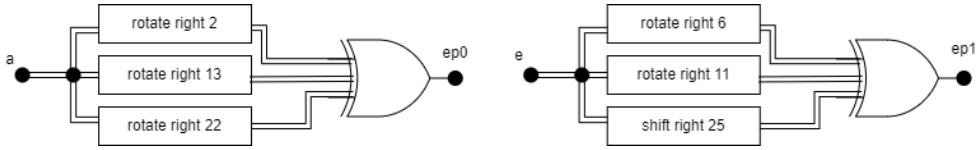
- σ_0 and σ_1 functions



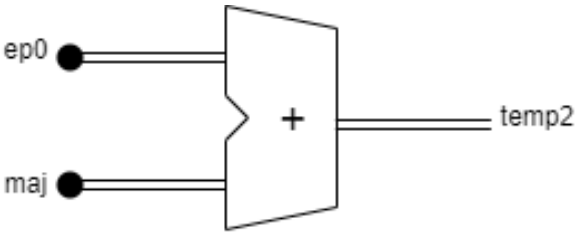
- Choice and Majority functions



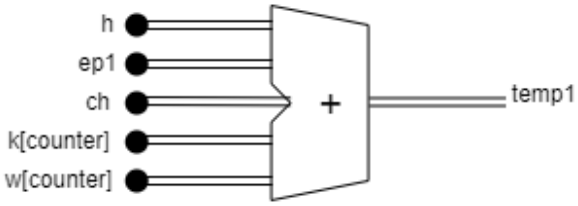
- Σ_0 and Σ_1 functions



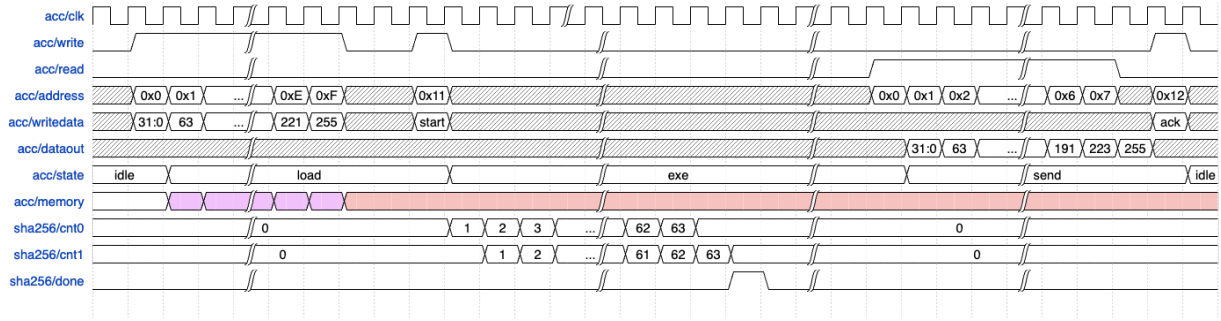
- Sum of Majority and Σ_0



- Sum of Choice, Σ_1, $K_{constant}$[counter], and Word[counter]



2.3 Hardware Timing Diagram



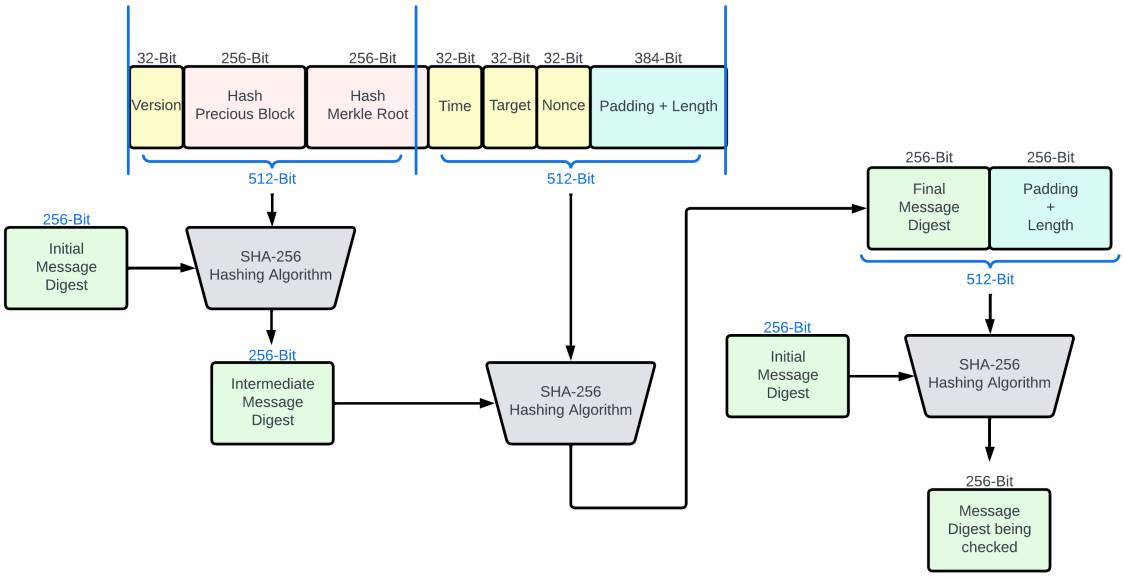
2.4 Algorithms

2.4.1 Overall Mining Process

SHA256 algorithm is the most fundamental algorithm used in Bitcoin Mining. Assuming we join a mining pool through Stratum v1 protocol, we will receive notifications, **'min-**

ing.notify', from the server. The notification is in JSON-RPC format and contains the necessary information. We first need to use the received information to calculate the **Merkle root** by applying **Double-SHA256**. Double-SHA256, or Double-Hash, implements the SHA256 algorithm twice on the 1024-bit input and gets a 256-bit output. Once we have the Merkle root, we can combine it with other binary numbers, namely 'Version', 'Previous Block Hash', 'Time Stamp', 'Target', 'Nonce', and 'Padding'. This 1024-bit combination is the Bitcoin Block Header which is fed into the miner to implement Double-SHA256 again. And finally, we will have a 256-bit hash output. We compare it with the 'Target'. If this value is larger than 'Target', we will increment 'Nonce' and subsequently change the Bitcoin Block Header, and do Double-SHA256 again; otherwise, we will send this hash value to the server to check its validity. If it is valid, we collect a reward; otherwise, we do the whole process again with a newly received notification.

- Block Diagram of SHA-256 Process



2.4.2 Basic SHA-256 Function:

SHA-256 algorithm implements 6 main functions. The first 4 functions are used in the Compression process and the next 2 functions are utilized in the Message Scheduler process.

For each function, it takes 32-bit words as inputs. And all these functions will generate 32-bit word output as well [2].

Choice(a, b, c) = $(a \wedge b) \oplus (\neg a \wedge c)$

Majority(a, b, c) = $(a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$

$\Sigma 0(a)$ = rotR(a, 2) $\oplus$ rotR(a, 13) $\oplus$ rotR(a, 22)

$\Sigma 1(a)$ = rotR(a, 6) $\oplus$ rotR(a, 11) $\oplus$ rotR(a, 25)

$\sigma 0(a)$ = rotR(a,7) $\oplus$ rotR(a, 18) $\oplus$ shiftR(x, 3)

$\sigma 1(x)$ = rotR(a,17) $\oplus$ rotR(a, 19) $\oplus$ shiftR(x, 10)

rotR and shiftR, are two pre-defined functions for SHA-256 algorithms. For rotR function, it realizes the right rotation for 32-bit word input. For shiftR function, it realizes the right shift of input and fills with ones.

2.4.3 Message Padding

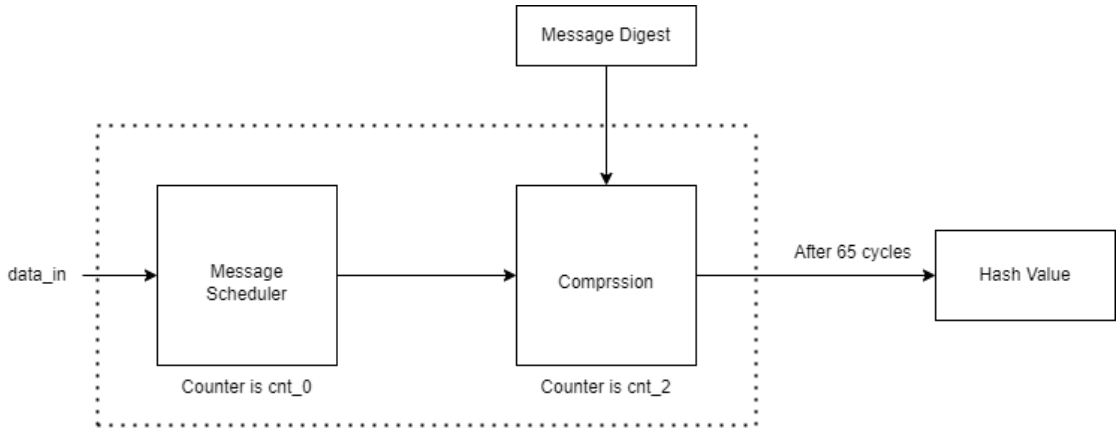
Message padding is used to expand the length of the information bit to multiples of 512 bits. The reason is that the SHA256 hashing algorithm required a 512-bit word as input. Firstly, a single "1" is appended to the message block. Then, the length of the original message will be appended at the end of the message block as a 64-bit big-endian integer. Finally, depending on the current length of the message block, several zeros will be added between the encoded message and the length integer that was just added in the previous step. By these three steps, a 512-bit message block will be successfully created [3]. The message padding will happen several during the SHA256 process.

2.4.4 SHA256 Algorithm Module With 512-bit Input

Considering that we have a **512-bit input**, this input is divided into sixteen 32-bit words, equivalent to sixteen 4-byte words. Basically, a single SHA256 Algorithm has two parts: **Message Scheduler** and **Compression**. In the Message Scheduler, we send these sixteen 4-byte words into W[0] to W[15], and use them to generate the rest 48 words, W[16] to W[63],

by doing a series of bitwise operations including XOR, right rotation, and right shifting. This process takes 64 clock cycles. In Compression, we initialize eight hash values A, B, C, D, E, F, G, and H with **input Message Digests**. The Compression consists of 64 iterations, applying a series of bitwise operations, logical functions, and constants to the hash values and Message Scheduler words (W[0] to W[63]). In each iteration, hash values are updated. And after 64 iterations, the hash values are added to the input Message Digests to produce the **output Message Digests**. The final 256-bit **hash value** is obtained by concatenating these eight output Message Digests.

Therefore, a naive way to implement the SHA256 Module needs 128 clock cycles, but we use two counters to "parallelize" these two parts. We use counter0 in the Message Scheduler and counter2 in the Compression. And counter2 is one clock cycle delayed than counter0. When in the first cycle of Message Scheduler, counter0 = 0, the Compression initializes the parameters, and in following cycles, updates them. Therefore, within one clock cycle, the Message Scheduler produces a new result, and the Compression takes the Message Scheduler result produced in the previous clock cycle. As a result, we only need 65 clock cycles to finish the SHA256 Algorithm. This saves half time.



2.4.5 Single SHA256 Algorithm

As introduced above, a SHA256 Module takes a **512-bit number** and **256-bit Message Digests** as inputs, and produces **256-bit Message Digest** as the output. The single

SHA256 Algorithm uses one or more SHA256 Modules to calculate the **256-bit Message Digest output**.

The input size must be an integral multiple of 512-bit which is achieved by Message Padding. Then, this input is broken down into 512-bit trunks and passed into SHA256 Modules. For example, 2048-bit input is broken down into 4 512-bit trunks and passed into 4 SHA256 Modules: the first trunk is passed into the first SHA256 Module; the second trunk is passed into the second module... The input Message Digests of the first SHA256 Module are the default Message Digests, and for other SHA256 Modules, the input Message Digests are the output Message Digests from the previous SHA256 Modules. Therefore, the last SHA256 Module has the **256-bit output Message Digests** as the final output of this process.

2.4.6 Double-SHA256 Algorithm

Double-SHA256 implements SHA256 Algorithm twice on an input. The input is first passed into the Single SHA256 Algorithm as described above. The output of the Single SHA256 Algorithm is the 256-bit Message Digests. Then, this **256-bit Message Digest** is padded to 512-bit and passed to a SHA256 Module with the **default Message Digests** together as inputs. The final output of the Double-SHA256 Algorithm is a **256-bit Message Digest**.

2.4.7 Merkle Root Calculation

We receive the message from the mining pool and the message contains the following field:

- **'id'**: This is always **'null'** for notification
- **'method'**: The name of the method being called, in this case, **'mining.notify'**
- **'params'**: An array containing the parameters for the method being called. The parameters for the mining.notify messages are as follows:
  - **'Job id'**: A unique identifier for the mining job

- **'Previous Block Hash'**: The hash of the previous block in the blockchain, in little-endian format
- **'Coinbase Transaction Part 1'**: The first part of the coinbase transaction, up to and including the Extra Nonce 1 field
- **'Coinbase Transaction Part 2'**: The second part of the coinbase transaction, starting after the Extra Nonce 2 field
- **'Merkle Branch'**: An array of hashes that form the Merkle branch used to build the Merkle root of the transaction tree
- **'Block Version'**: The version number of the block
- **'nBits'**: The compressed target difficulty for the block
- **'nTime'**: The current network time, in little-endian format
- **'Clean Jobs'**: A Boolean value indicating whether the client should discard previous jobs and start working on the new one

Besides, the mining pool sends unique information, **'Extranonce1'** and **'Extranonce2_size'**, to the client when the client subscribes current connection for receiving mining jobs. We then generate appropriate **Extranonce2** according to the given size. We concatenate **Coinbase Transaction Part 1**, **Extranonce1**, **Extranonce2**, and **Coinbase Transaction Part 2** to make the **Coinbase**. Its size is variate, Message Padding it and passing it to the Double-SHA256 Algorithm to produce a 256-bit output.

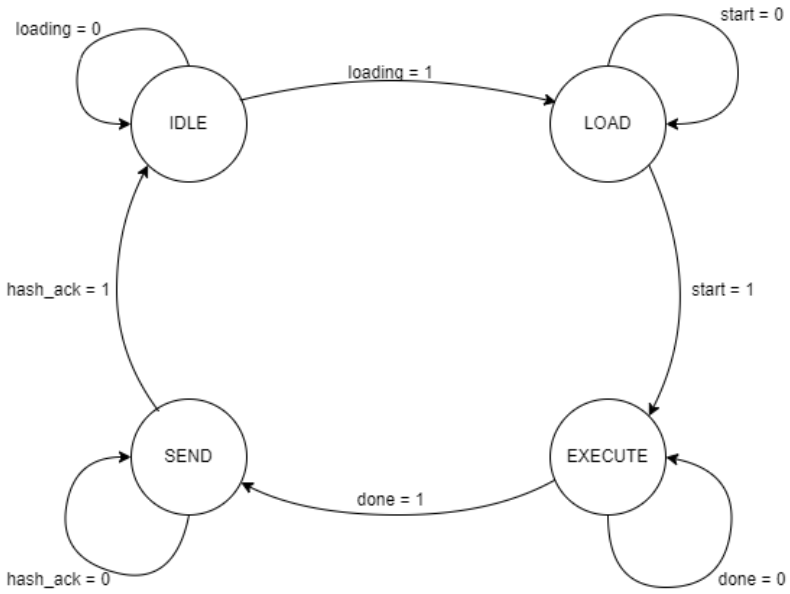
With this 256-bit Message Digest, we concatenate it with the first hash in **Merkle Branch**, Message Padding to 1024-bit, and pass it to the Double-SHA256 Algorithm. With the obtained 256-bit output, we concatenate it with the second hash in **Merkle Branch**, Message Padding to 1024-bit, and pass it to the Double-SHA256 Algorithm. Repeating this process with all hash values in **Merkle Branch** in sequence, we finally get a 256-bit Message Digest. Ultimately, we need to reverse its byte order, and the result is the **Merkle Root**.

2.4.8 Building Block Header

Every time we want to compute the Block Header hash, we need to generate a **Nonce**. At this point, we have got all the necessary information to build a block header. We just concatenate **Block Version**, **Previous Block Hash**, **Merkle Root**, **nTime**, **nBits**, **Nonce** to get the block header. This block header should be 640-bit which is exactly 80 bytes. With this block header, we do Message Padding and pass it to the Double-SHA256 Algorithm and compare the result with the target. If the result is not accepted, we need to increment **Nonce** and update the block header and repeat the Double-SHA256 process.

2.4.9 Top Module

With the above information, we know how to get the Merkle Root, the Block Header, and use them to calculate the 256-bit hash value. The execution of SHA256 Module is only a state in the overall process. In the top module, we have four states, namely, **IDLE**, **LOAD**, **EXECUTE**, and **SEND**. We send a control signal from the software to the hardware to drive the state from IDLE to LOAD. In state LOAD, we load the 16 32-bit inputs from the Avalon bus to the registers. When the loading is done, a "start" signal is set to 1, indicating the SHA256 Module starts to execute. Then, in the state EXECUTE, the SHA256 Module is executed for 65 clock cycles, as described in Section 2.4.4. The "done" signal from the SHA256 Module then drives the state to SEND, sending the output to the Avalon bus. Ultimately, the state go back to IDLE. By using different input data, we can achieve Double-SHA256 Algorithm.

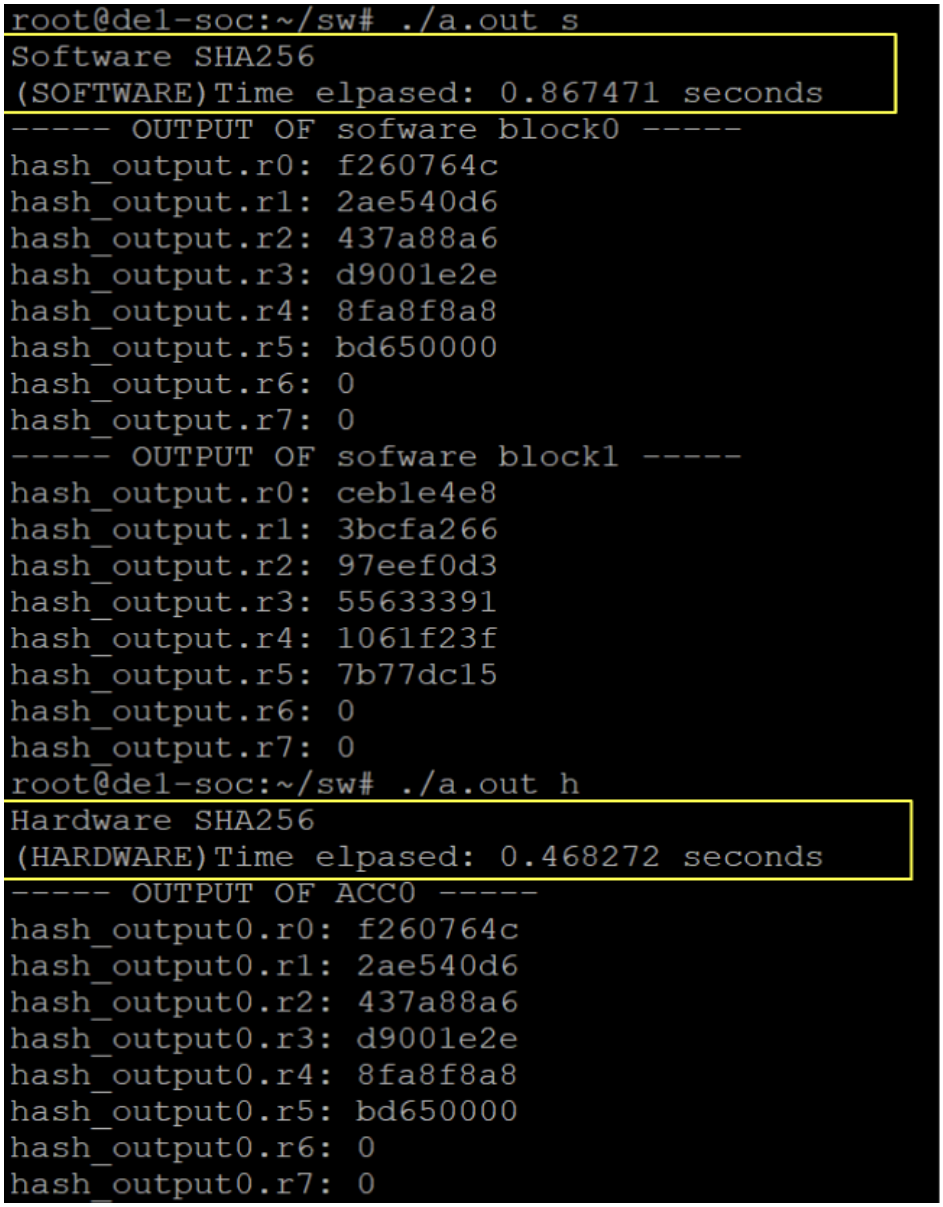


2.5 Miner-Mining Pool Connection

(This section is still undetermined. Currently, we cannot displ(This section is still undetermined. Currently, we cannot display our devices on the Mining Pool, but we indeed receive notifications from the Mining Pool. Hopefully, we can connect to the Mining Pool, otherwise, we can only hard-code some inputs from software to hardware to show its performance.)ay our devices on the Mining Pool, but we indeed receive notifications from the Mining Pool. Hopefully, we can connect to the Mining Pool, otherwise, we can only hard-code some inputs from software to hardware to show its performance.)

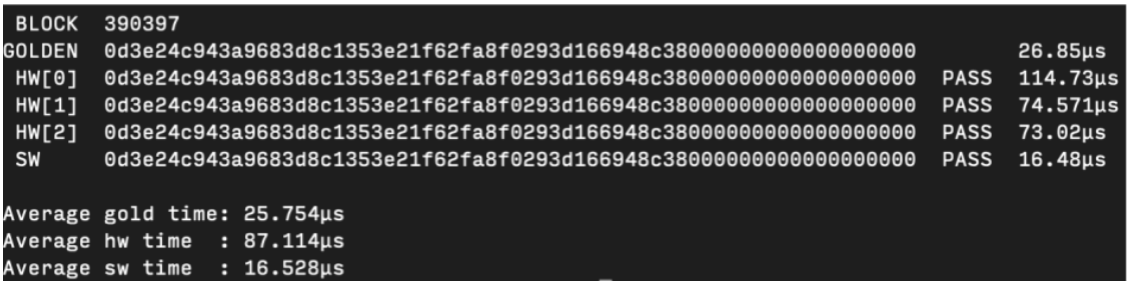
2.6 Result Comparasion

- Result Comparison of Multi-Hardware Version



In the Multi-Hardware version, to run 10000 cycles of SHA-256 algorithm, it spends 0.468272 seconds while the software takes 0.867471 seconds. Therefore, the speedup in the Multi-Hardware design is about 85 %.

- Result Comparison of Multi-Device Version



See above for the time various implementations take for one single hash of the block header. Golden refers to a well-known Rust Bitcoin software library used to check for correctness, HW[0,1,2] refer to each of the three devices in the multi-device design, and SW refers to our own software implementation for our own learning. As shown above, the HW implementation takes longer than both software implementations.

3 Effort, Learned Skills, Advice for Future Project

3.1 Effort

All contribute equally

- Jules Comte:

  Hardware architect and design of multi-device design

  Software(Rust) implementation of device driver and data fetching & process

- Mingyang Song:

  Hardware architect and design of multi-hardware design

  Software(C) kernel space implementation and mining function design

- Zhe Mo:

  Hardware SHA256 datapath design and design verification on RTL level

  Software data fetching, post-process and formal verification

- Tianyu Qin:

  Hardware RTL implementation of multi-hardware design and post synthesis verification

  Software user space function implementation of multi-hardware design

- Xueji Zhao

  Hardware implementation and testing

  Software golden model design and formal verification

3.2 Learned Skills

1. Realization of complex algorithms on hardware, from algorithm understanding, software model building, to hardware data path design and behavioral simulation using System Verilog on Quartus FPGA platform.

2. Hardware optimization, push the data path to achieve more aggressive performance through pipeline technique.

3. Software and hardware interface design. For the operating system, all peripherals are memory, and what we do is read and write operations on the memory, and care should be taken when manipulating memory especially using C.

4. Bug solving skills, behavioral simulation could be different from post synthesis simulation as well as the on board behavior. Timing requirement should meet the definition every time we synthesis the design and a good RTL coding style is a good start.

3.3 Advice for Future Project

Never do systematic work without a validation framework.

4 Reference

[1] Lecture Note 02, ELEN E6883: An Introduction to Blockchain Technology, Spring 2023;

[2] http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[3] https://sha256algorithm.com/

5 File List

In order to make every team member experience the full-stack process of Bitcoin mining as much as possible. We designed and implemented two different versions (multi-device & multi-hardware) and present them here:

5.1 Multi-device Source Codes

Directory Structure:

```
multi-device/
├── hello.c
├── sha256.c
├── sha256.h
├── soc_system_top.sv
├── vga_ball.sv
└── block-hash-testbench
    ├── Cargo.lock
    ├── Cargo.toml
    └── src
        ├── acc.rs
        ├── main.rs
        ├── sha256_hw.rs
        └── sha256_sw.rs
```

File name	Description
vga_ball.sv	Hardware implementation of sha256 module
soc_system_top.sv	Hardware specification of the top-level module
sha256.c	C implementation of the Linux driver
sha256.h	Header file for the Linux driver
block-hash-testbench	The userspace testbench for the hardware, written as a Rust application

5.2 Multi-hardware Source Codes

Directory Structure:

```
multi-hardware/
├── hw
│   ├── acc.sv
│   ├── sha256_incl.svh
│   └── sha256_module.sv
└── sw
    ├── acc.c
    ├── acc.h
    ├── miner.c
    ├── miner.h
    └── sha256.h
```

File name	Description
acc.sv	Hardware top module and finite state machine control unit
sha256_incl.svh	included SHA256 algorithm header file
sha256_module.sv	RTL for SHA256 data path and registers-based memory control unit
acc.c	C implementation of the Linux driver
acc.h	Header file for the Linux driver
miner.c	main function for mining control, speed test and verification
miner.h	hard coded block header for testing
sha256.h	SHA256 algorithm sofware implementation (golden model)