Development and Evaluation of an FPGA-based Proof of Work Miner for Power Efficiency, Performance, and Cost

Alex Ryan Spaulding
Frederich Lawrence Stine

Follow this and additional works at: https://louis.uah.edu/honors-capstones

Recommended Citation
https://louis.uah.edu/honors-capstones/597

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.
Development and Evaluation of an FPGA-based Proof of Work Miner for Power Efficiency, Performance, and Cost

by

Frederich Lawrence Stine and Alex Ryan Spaulding

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

4/22/2021

Honors Capstone Director: Dr. B. Earl Wells - Professor of Electrical and Computer Engineering

Student

B. Earl Wells

Date

4/23/2021

Director

Dr. Ravi Gorur

Date

Digitally signed by Dr. Ravi Gorur

Date: 2021.04.25 15:19:00 -05'00'

Department Chair

William Wilkerson

Date

Digitally signed by William Wilkerson

Date: 2021.04.26 09:03:15 -05'00'

Honors College Dean

Date
Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Alex Ryan Spaulding

Student Name (printed)

Alex Spaulding

Student Signature

04-25-2021

Date
Table of Contents:

Abstract: 4
Background: 4

Hashcash: 5
Development: 6

Modular Breakdown: 6
Communication Module: 8
Block Parser Module: 10
Clock Module: 11
Result Check Module: 13
Hash Module: 15
Final Design: 20

Evaluation: 25

Sources: 28

Appendix: 30

Communication Module: 30
Difficulty Calculator: 30
Block Parser: 31
Miner Top Level: 32
Miner Top Level Integration Test 1: 38
Miner Top Level Integration Test 2: 43
Abstract:

Proof of Work-based consensus algorithms are the backbone of modern cryptocurrency systems, allowing for consensus to be achieved across a decentralized network. With the fast rise in popularity of cryptocurrencies, the power consumption of running proof of work-based consensus algorithms has had a major impact on the world's power usage. As the popularity of cryptocurrencies increase it becomes increasingly important to optimize the power consumption of the mining process. The most efficient approach to Proof of Work-based mining is the use of application-specific integrated circuits (ASIC). Field Programmable Gate Arrays (FPGA) however offer a flexible hardware based alternative that is capable of similar efficiencies while also allowing for miners to experiment with hardware optimizations. With this research the team aims to determine the use case for FPGAs within the modern Proof of Work-based cryptocurrency mining systems. The team determined that FPGAs are a valuable asset in modern cryptocurrency systems as an intermediate step before the development of widely available ASICs. In this period, FPGAs allow for the best power consumption, performance, and flexibility. After ASICs become readily available, FPGAs can still allow for moderate improvement as users can experiment to improve the efficiency and profitability of the cryptocurrency mining process.

Background:

Most modern cryptocurrency networks operate off of a network of trust. This network of trust is established by requiring that every user who wishes to participate in the network must establish a proof of work in order to process transactions on the network. Proof of work is a cryptographic zero knowledge proof [12] that shows a party has expended computational effort. In order to process transactions miners must conduct a proof of work on every single mining attempt. Processing a proof of work computation expends power proportional to the complexity of the computation. With the fast rise in popularity of cryptocurrencies, the power consumption of running proof of work-based consensus algorithms has had a major impact on the world's power usage. As of April 2nd, 2021, the estimated power consumption of the bitcoin network is 92.747 TWh annually, which is comparable to the annual power consumption of the Philippines [1]. FPGAs offer miners the ability to experiment with power optimizations.

Field Programmable Gate Arrays are devices that provide a matrix of configurable logic blocks to allow custom hardware to be designed and implemented without going through the full ASIC design cycle. FPGAs act as a pedian in the market between general-purpose computing components and ASICs[2]. FPGAs allow for the design to be focused directly on parallelization and removes overhead associated with general-purpose computing while still allowing for increased design flexibility. This creates an increase in speed and decrease in power consumption when compared to traditional computing components, while still allowing for improved flexibility during the design of ASIC implementations.
For this project the team sought to implement the Hashcash proof of work-based consensus algorithm used by Bitcoin. The design was implemented on an Altera DE2-115 FPGA board to evaluate the design with regards to power consumption, performance, and cost. With bitcoin being the first widespread cryptocurrency system, ASIC chipsets are widely available and used, so this project takes a theoretical approach to efficiently analyze the current state of FPGAs in cryptocurrency systems.

**Bitcoin header:**

Each Bitcoin block contains a set of data that represents all of the transactions contained within the block [6]. The header follows a standard format as defined below:

- Version (4 Bytes): Version of the block being processed
- Previous Block (32 Bytes): The hash of the previous block processed in the chain
- Merkle Root (32 Bytes): The hashed value of the Merkle Tree, a data structure representing all of the transactions contained within the block
- Time (4 Bytes): Time since Unix Epoch (January 1, 1970)
- Bits(4 Bytes): Difficulty target for the current block
- Nonce (4 Bytes): Random value associated with each hash attempt - every nonce represents a single hash attempt

**Hashcash:**

The Bitcoin mining network utilizes a modified version of the Hashcash proof of work algorithm. Hashcash is a proof of work algorithm created by Adam Back in 1997 as a method to prevent spam in emails. The idea behind this is that anyone who wishes to send an email must expend computational effort to sign the email. Any email that did not contain a proof of work header could be easily identified as spam. Bitcoin uses a modified version of the Hashcash proof of work algorithm in order to allow for improved cryptographic security. Specifically the bitcoin implementation utilizes SHA2, double hashing, and a fine grained difficulty system.
SHA2 acts as a cryptographically secure hashing algorithm that can operate as the basis for the proof of work algorithm. The Bitcoin implementation will run the previously described bitcoin header through the SHA2 hashing algorithm twice before checking to see that the output of the hash function is less than that of the defined difficulty.

**Development:**

**Modular Breakdown:**

The team began development by creating a modular breakdown of the different components of the system that were required to have the functionality needed. This modular breakdown served as a way to distribute the work across the team, while taking an object-oriented approach to the design of the hardware.
The modular breakdown highlights six different modules that were determined to be required for correct operation of the FPGA based proof of work miner. The first of these modules is the communication module, which handles communication between the host computer and the FPGA. This module is used for sending new block header information and receiving valid hashes and the block header data that resulted in them. The second module is the block parser module, and this module takes the raw data that is received from the serial module and parses it into different sections of the block header so that it can be utilized by other modules within the system. The third module that was determined to be necessary was the clock module. The clock module was chosen as a system to increment time values on the hardware so that additional communication between the host and miner were not needed to ensure that the miner does not deplete the full nonce range while mining. The fourth module that was determined to be necessary was the dispatcher module. This module dealt with the breakdown of work between
multiple instantiations of a hash module, such that multiple hash modules could run in parallel with each other. This module would do this by sending specific nonce values to the hash modules to ensure that there is no overlap between the parallel hashing cores. The hash module is the module that creates the main functionality of the FPGA miner card. These hash modules will feature the ability to successfully hash one full block two times, such that the output aligns with the standards of the Hashcash algorithm utilized by Bitcoin. The last module is the result check module, and this module deals with the expansion and comparison of the custom floating point difficulty target, known as 'bits', in the Bitcoin header to a 256-bit long target value that can be compared to the output of the SHA-256 based hashing cores. This module would also need to be able to trigger and send signals to the communication module when a correct solution to the Proof of Work algorithm is determined.

With these modules the team had outlined a fully functional Bitcoin specific Hashcash based miner card. After this point the team was able to begin development of the first module, to build towards a fully functional system.

Communication Module:

The communication module was the first module that was created by the team. This module laid the groundwork for sending information between the host computer and the FPGA, such that other modules that were created could be tested on the FPGA in addition to within a ModelSim simulation testbench.

The team received VHDL files outlining an asynchronous serial communication framework from Dr. B Earl Wells, the project director. These files were included in addition to the remote access to a DE2-115 board through the electrical and computer engineering college’s Blackhawk server. Using the VHDL files the team was able to test the functionality of the serial communication framework provided. The framework provided could be used to send a maximum of 8 bytes at a time, and a minimum one 1 byte. The team needed to be able to send 80 byte Bitcoin block headers to the board, and receive both 80 byte headers and 32 byte hashes from the board. This required significant modifications to the serial communication framework that was provided.

With the modified data sizes for the communication blocks, the team was able to outline and create methods for both receiving and sending serial data. These methods were created using block design files (.bdf) within Quartus. This provides a visual representation of the instantiations of different modules that are utilized. The team first created the methods for receiving serial data.
Fig 3: Serial receive block design file

This block design file takes two input signals, a 50Mhz clock, and the serial communication receive line. These signals tie into the instantiation of the source_header block, which waits for a specific identifier byte to be received, and then receives an additional 80 bytes, which are stored in an internal register. This block also raises the data_valid signal high for one clock cycle whenever new data is received. The team chose to use a baud rate of 115200 for the serial communication due to the lower latency between sending and receiving between the host computer and the FPGA to minimize the communication time required for the system.

The team then created the methods for sending data from the FPGA. This was also done using block design files within Quartus Prime.
The serial transmit file works similarly to the serial receive method. The blocks that are instantiated wait for a specific identifier byte to be received and then determine which module needs to send data. This creates a master-slave communication system, where communication is controlled by the host computer and requests are returned by the FPGA. This allows the team to create methods in software to ensure that data transfers occur at the correct times in the software and that two transfers do not overlap. The FPGA uses a custom byte wide set-reset (SR) flip flop to signal to the host computer that it is ready to transmit information. This is tied to a ready-to-send signal that will be controlled by the result check module within the final design. This byte is able to be probed through the use of the probe_byte block. The software can read this byte and determine whether the FPGA is ready to return a result to the host computer. There are two main probe modules that are used for sending the data back to the board, these are the probe_hash and probe_header modules. The probe_hash module sends 32 bytes of data to the board, and the probe_header module sends 80 bytes of data to the board. The strobe_bit module works differently from all of the other modules within this file, and when it receives the correct identifying byte, it raises the strobe signal high for one clock cycle. This is used to reset the byte using polling after the software has read the information.

With the main functionality for the communication module finished, the team created the final top level module for communication (Communication Module). This module instantiates both the serial receive module and the serial transmit module to create a clean interface for addressing communication from the board. With this created the team moved on to creating the block parser module to allow for efficient creation of a unit test for the communication module.

**Block Parser Module:**

The team created the block parser module, which takes in raw data from the serial module and clock module, and outputs a standard breakdown of the sections of the Bitcoin header. This module updates a data_received signal whenever there is an update from the communication module, or an update from the clock module.

With this module created, the team was able to create a unit test for both the block parser and the communication module to fully encompass the functionality of the communication.
between the host computer and the FPGA. This driver receives and parses data, and when the data has been parsed it is loaded into the hash and header transmit buffers on the communication module. Once this is loaded the ready to send signal is held high for one clock cycle to set the polling byte to one so that the host computer knows that data is ready to be read.

The team also created software for the unit test, where data from 0-79 was initialized into an array to be sent to the FPGA. After the data was sent the software waited on the polling byte to go high before receiving the parsed data back from the FPGA.

![Fig 5: Communication module unit test](image)

Here the data is sent from the host computer to the FPGA, and is received back successfully. The ready signal is printed out, and the data was ready to be received after the first polling cycle. The team also was able to note that the data was returned from the board in the same ordering that it was sent to the board. With this functionality built and tested, the team was able to focus on implementing the Bitcoin specific Hashcash algorithm on the FPGA.

**Clock Module:**

The Hashcash proof of work algorithm utilizes the current time since Unix Epoch as one of the components for computing each hash. When a new block is received by the miner the block header contains the initial value for the time since epoch that should be used while mining the block. Typically though a new header will not be dispensed until a new block is generated, so it is important for miners to track the time during the mining process. The clock module will load in the time provided by the block header and increment the current time provided to the hash module every second. The goal of this module is to act as a time keeping device during the mining process so that the current time is never lost while seeking a correct solution to the current block.

This module takes the system clock, an initial time, and a reset signal as inputs, while providing the current time and an incremented time signal as outputs. On reset the module will set the current time equal to the value provided on the initial time bus. From there the module will begin incrementing a binary counter on every clock cycle. The counter will continue to increment until the value is equal to the clock frequency of the system, indicating that one second has passed. At this point the module will increment the output time and signal that a
clock update has occurred, essentially forcing the hash module to load a new block containing the valid system time.

The team also wrote a software unit test to ensure correct functionality of the clock module. The test would initialize serial communications with the FPGA. Upon initialization an initial time was sent to the FPGA miner in order to begin the clock counting process.

```
bash-4.2$ ./fpga_time_module_test
After init_serial_port
Timing module test:
Stage 1: Starting counting from 0
Sending header: done
Receiving time:
8 seconds since send
1 seconds since send
7 seconds since send
6 seconds since send
5 seconds since send
4 seconds since send
3 seconds since send
2 seconds since send
1 seconds since send
Stage 2: Starting counting from 10000
Sending header: done
Receiving time:
10000 seconds since send
10001 seconds since send
10002 seconds since send
10003 seconds since send
10004 seconds since send
10005 seconds since send
10006 seconds since send
10007 seconds since send
10008 seconds since send
Stage 3: Starting counting from current epoch time
Sending header: done
Receiving time:
1618808234 seconds since send
1618808235 seconds since send
1618808236 seconds since send
1618808237 seconds since send
1618808238 seconds since send
1618808239 seconds since send
1618808240 seconds since send
1618808241 seconds since send
1618808242 seconds since send
1618808243 seconds since send
```

Figure 6: Clock module unit test

Once the initial time was set in the clock module the software would begin to poll the data ready bit of the FPGA to determine when the clock module had completed a full iteration. Once the data ready bit was fired the current time output of the clock module was read from the FPGA and output in software. Using this functionality the team was able to determine that the clock module could take any arbitrary 32 bit input time and continuously increment the output.
time every second. This was tested with an initial value of 0 seconds, 10000 seconds, and time since Unix Epoch. The last test most reflects the behavior of the clock module as it utilizes a recent valid Unix Epoch time and still demonstrates proper operation of the clock module.

Though the current time since epoch needs to be updated during the mining process whether or not a module is specifically included for this purpose can be left up to the user. An alternative to including a hardware clock module would be to have the driving program push a new block to the FPGA every second in order to ensure that a valid time is held at all times. This however requires increased communication between the FPGA and the host computer, and care should be taken to not saturate the communication interface in highly parallel designs.

Result Check Module:

The result check module was designed next by the team. Bitcoin implements a custom difficulty system that is different from the initial difficulty system designed for Hashcash. Hashcash can change between difficulties that are a power of two by requiring an additional bit of the resulting hash to be zero. The time to mine a block in Bitcoin is very important, because it ensures that transactions are posted to the blockchain ledger in predictable times. The faster a block is mined, the sooner transactions are fully confirmed onto the blockchain. This time cannot be too fast because it takes miners time to collect new transactions to include in a block after a new block is mined, and it cannot be too slow as transactions would have a high processing time [3]. To ensure a predictable mining time for a new block, Bitcoin implements a custom difficulty system that allows the network to adjust difficulty such that a new block is mined around every 10 minutes regardless of the current mining hash rate of the network. To do this a custom difficulty is encoded into a four byte field using a custom floating point value. This value can be expanded to a 32 byte target that can be used when comparing the results of the hashes calculated [4].

With the information on how to convert between the packed value and the full target, the team was able to implement a system that expanded the result target and then compared it to another 32 byte value to determine whether a correct solution had been found. This module takes in the difficulty and a signal on when the value has been updated and outputs the full width target (link to code). The team also created a module that can determine whether a hash is correct by taking in the target and a hash output and doing a comparison on the two. This second module was used for creating a unit test for the full functionality of the result check module. The unit test was split into two separate portions. One portion checked that the difficulty target was being calculated properly by the FPGA, and the second portion checked that the result was being properly evaluated by the FPGA.

The first driver receives data, and parses the data. Once the data is parsed one clock cycle is given for the difficulty to be calculated by the module before being sent back to the computer in the hash portion of the communication module’s transmit function (link to hw driver). Software drivers were also written for this portion that encode a specific pre-known difficulty to
check into the block header in the correct location, and then send and receive the data from the board (link to sw driver).

Fig 7: Result module unit test part 1

Here the value that was sent to the board for difficulty was 0x1b0404cb, which expanded correctly to the target value shown as the output from this unit test. With this information the team knew that the difficulty was being properly calculated on the FPGA.

The second unit test was designed to test whether the difficulty target that was calculated was able to be compared to hash results on the board successfully. To do this the driver receives two headers, the difficulty is calculated from the first header, and the second header contains the data that is used to compare with the calculated difficulty target. If the hash that is sent is less than the target difficulty the polling byte becomes high and the software can confirm that the value is correct. If the polling byte stays low, the software knows that the value is incorrect. This was controlled via unit test software that sent the correct values and then returned different information based on the value of the polling byte.

Fig 8: Result module unit test part 2

In this unit test we can see that the hash that was sent was verified by the board as less than the target. This was the correct outcome in this run of the test, and allowed the team to know that the result target calculated was able to be compared with hash values successfully.
Hash Module:

With the foundation of the Bitcoin specific Hashcash algorithm laid out, the team was able to focus on the main hashing functionality of the board. The Hashcash algorithm used by Bitcoin uses two iterations of SHA-256 to calculate the final result that is compared with the difficulty target. The team found an open-source implementation of the SHA-256 hash algorithm in Verilog that was utilized for creating the functionality needed for the hash module [14]. The first thing that the team did was to learn the specific signals that the hash module needed to operate properly, and what the output characteristics of the module were. To do this the team used Modelsim, a hardware simulation tool, to quickly alter the design. The use of Modelsim allowed the team to simulate a design with very small compilation times, compared to compilation times of around 15 minutes for the physical board.

The first test driver that was created was a Modelsim testbench that tested the output of a hash that was less than 56 bytes. The size of the input to the SHA-256 algorithm was of importance due to the 64 byte input block size that is used by SHA-256. All blocks that are hashed by SHA-256 must feature some amount of padding that displays the amount of bits in the data as well as ensuring the size of the block is a multiple of 64 bytes. If the input is less than 56 bytes the data is capable of being calculated within one full iteration of the SHA-256 algorithm. The team needed a way of padding data such that it would be able to be hashed correctly by the SHA-256 algorithm. To do this the team found pseudocode for the SHA-256 algorithm and used it to create a c program that calculated the padding for an arbitrary input [5]. Using this program the team was able to create padding to be used with the simulation of the SHA-256 hash instantiation. The team still had to reverse engineer the byte ordering that was being sent to the board. On analysis of the driving logic for the board provided by the open-source project, the ordering of bytes was not the same as the padding calculated by the team. After some analysis it was determined that the ordering of bytes was being flipped in 4 byte segments. With this ordering determined, the team modified the c program to display the data in the exact ordering that was expected from the open-source SHA-256 module.

Using the padding information and the SHA-256 module, driving logic was created for the module in a testbench. The SHA-256 module chosen also has a variable size parameter, and the team chose the most unrolled option to allow for the highest hash throughput. This resulted in a pipelined execution with 64 iterations, where one hash result could be calculated every clock cycle once the pipeline was full. The team tested the testbench in Modelsim with multiple data values and compared the values to an online SHA-256 hash calculator. From the results the team could see that the correct byte ordering and driving logic of the hash module had been calculated properly.
At first glance the two values here do not match, but on further inspection, the output of the hash module in simulation is altered in the same manner that the input data was. Here we can see that the last 4 bytes are 0xad95131b, and that the first 4 bytes of the online hash calculator's result were 0xab97131b. The team knew that this needed to be taken into account when comparing the final result with the target difficulty.

After proving that a singular hash with an output less than 56 bytes could be calculated properly, the team decided to hash a value that was the full length of the Bitcoin header. The Bitcoin header is 80 bytes long, which means that the data will not fit in one full iteration of the SHA-256 algorithm. This means that the data needs to go through the hash algorithm twice to determine the result of the first hash. The team determined that the first 64 bytes of data never change in the Bitcoin block header. This means that the first iteration of the SHA-256 algorithm
for an 80 byte long block only has to be done one time. This calculation is known as the midstate of the hash, and the team uses this to greatly increase performance of the hashers. This state is used as the initialization vector for the next iteration of the SHA-256 algorithm to chain together the data.

In this Modelsim driver the team aimed to successfully calculate the hash of a full Bitcoin block header. To do this the team needed to do three iterations of the SHA-256 algorithm. The first iteration calculated the midstate, the second found the full result of the first hash, and the third found the full result of the double hash. The byte ordering was also important for this portion of the design, and the team determined which data had to be sent into the board first, and ensured that it aligned with the hash module's data ordering. This test was then simulated in Modelsim to ensure that the design was working properly.

Fig 11: Modelsim result of double hashing a full block

---

Binary hash

Hex bytes: c56705f8a5b110b8dc63688533ced21167e628017:

Hash

SHA-256 852c98044fb00507122ff63bda7b529566348fc204f72b00dff1af7b40501e4
In the results here the team can see the output of both the Modelsim’s final hash, and the final hash as calculated using an online hash calculator. As with the previous test the team needed to extract the valid hash from the output due to the changes in byte ordering between the input and the output. With doing this we can see that the calculated values between the two are the same, and that we have effectively hashed the block header as expected by Bitcoin’s implementation of the Hashcash proof of work algorithm.

The team then worked on creating a test driver that worked with the 56 byte input or less data on the FPGA. This technique followed very similarly to the design for the Modelsim example, but instead of creating a Modelsim testbench, the team created a verilog driver for the board. This uses the communication module and block parser to send the correct data to the hash module, and to receive the calculated hash from the board. The team also created a driver program for this functionality in c, that reads in the binary information from a binary file before sending it to the board. The team needed to create another program to create the binary information, and for this Python was used to write the binary data to a file. With all of this done the team was able to run this unit test on the hashing functionality.

![Fig 12: Result of online hash calculation](image)

![Fig 13: Raw hex data used to send to FPGA](image)

![Fig 14: Result of hash as result from FPGA](image)
From this unit test the team can see that the hash is being calculated successfully on the board in the same manner that it was in the simulation. As with the results that were calculated on the simulation, the output here is swapped in 4 byte chunks. The final design of the hash module will take this into account, but for the unit test the output here is as expected.

The team then created another unit test to ensure that a full block was able to be successfully hashed on the FPGA. This means that the team needed to calculate the midstate on the FPGA and patch it back into the hash module to ensure that the result of the 80 byte block was found properly. This driver was created in verilog and created similarly to the previous unit test such that data could be sent and received through the communication module. As with the previous design, the raw data was read in by the software driver and sent to the board, but the padding was done by the hardware on the FPGA instead of via software. The team used a previous block’s header information to test this functionality.

Fig 15: Result of hash from online calculation

Fig 16: Raw hex data used to send to the FPGA
Final Design:

The final design of the proof of work based miner varied in multiple ways from the proposed development the team created with the modular breakdown. The modules that remained as proposed were the communication module, block parser module, hash module, and result check module.

The design choices that the team made to utilize the maximum throughput configuration of the hash cores negated the use of the dispatcher module. The project focused on parallelism.
within the singular hash core instead of parallelism between multiple hash cores. The project initially planned on using a much larger and more powerful FPGA, but due to restrictions from Covid-19, the team ended up utilizing the Altera DE2-115 FPGA, which only featured enough space for one full instance of the hash module.

The team also removed the clock module from the final design due to size constraints. While the final design has space available with regards to logic elements, the clock module was unable to be fitted by the FPGA onto the final design. The team decided to utilize a software based approach to updating the time on the device instead. The final design was not running at a speed where clock updates needed to be every second, so the software approach updates the time every five seconds.

Due to restraints on doing hardware design fully remote, the team did not have physical access to any of the hardware. Because of this the team took extreme caution with the speed of the final design on the device. In order to reach the maximum speed of the device, the clock needs to be raised, but for the final testing the team downclocked the clock to 5Mhz to avoid any issues that could have arisen with heat from the device. If the team had physical access to the device, the team would have pushed the clock rate to the maximum and provided ample cooling solutions for the FPGA. The team could have likely run the final result at 50Mhz, but did not want to take risks since observations on temperature could not be made.

The final hash module ended up being built directly into the top level entity. This was initially for debugging purposes, but also ended up helping to simplify the driving logic for the hashing functionality. If this project were done on a larger board, this would easily be able to be encapsulated in a module so that it could be parallelized between multiple instantiations. The final result could run at a theoretical maximum of around 75MHz with the use of a phase locked loop (PLL) to speed up the clock on the board. This number is based on time constraints for the chosen hash module. With physical access and additional cooling it could be possible to get this value higher. This could be a continuation of this project after the Covid-19 pandemic.

The final entity, miner_top_level, instantiates all of the modules highlighted in the final design, and includes the logic for the hash module functionality. Whenever new data is received on the board, the board calculates the midstates before filling the pipeline with data. When the first successful outcomes begin to appear, the board enables the result checking functionality. The difficulty, nonce, and time needed to be delivered to the board in big endian for their functionality to work as expected. All values are expected to be in little endian order for the Bitcoin Hashcash hash, so these values are rearranged before hashing is done.
Flow Status: Successful - Tue Apr 20 2014:57:56 2021
Revision Name: fpga_miner
Top-level Entity Name: miner_top_level
Family: Cyclone IV E
Device: EP4CE115F29C7
Timing Models: Final
Total logic elements: 80,088 / 114,480 (70%)
Total registers: 4893
Total pins: 3 / 529 (< 1%)
Total virtual pins: 0
Total memory bits: 29,504 / 3,981,312 (< 1%)
Embedded Multiplier 9-bit elements: 56 / 532 (11%)
Total PLLs: 0 / 4 (0%)

Fig 19: Compilation results for top level entity

<table>
<thead>
<tr>
<th>Resource</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Estimated Total logic elements</td>
<td>98,012</td>
</tr>
<tr>
<td>3 Total combinational functions</td>
<td>63616</td>
</tr>
<tr>
<td>4 Logic element usage by number of LUT inputs</td>
<td></td>
</tr>
<tr>
<td>1 - 4 input functions</td>
<td>1819</td>
</tr>
<tr>
<td>2 - 3 input functions</td>
<td>54572</td>
</tr>
<tr>
<td>3 &lt;=2 input functions</td>
<td>7225</td>
</tr>
<tr>
<td>6 Logic elements by mode</td>
<td></td>
</tr>
<tr>
<td>1 -- normal mode</td>
<td>26534</td>
</tr>
<tr>
<td>2 -- arithmetic mode</td>
<td>37082</td>
</tr>
<tr>
<td>8 Total registers</td>
<td>48938</td>
</tr>
<tr>
<td>1 -- Dedicated logic registers</td>
<td>48938</td>
</tr>
<tr>
<td>2 -- I/O registers</td>
<td>0</td>
</tr>
<tr>
<td>9 I/O pins</td>
<td>3</td>
</tr>
<tr>
<td>11 Total memory bits</td>
<td>29504</td>
</tr>
<tr>
<td>13 Embedded Multiplier 9-bit elements</td>
<td>56</td>
</tr>
<tr>
<td>15 Maximum fan-out node</td>
<td>CLK</td>
</tr>
<tr>
<td>16 Maximum fan-out</td>
<td>51190</td>
</tr>
<tr>
<td>17 Total fan-out</td>
<td>312455</td>
</tr>
<tr>
<td>18 Average fan-out</td>
<td>2.63</td>
</tr>
</tbody>
</table>

Fig 20: Compilation resource usage for top level entity
In the compilation results for the top level entity we can see that the board is approaching being full. The team also experienced this in compilation, where the clock module, and other logic could not be fit into the final design. The design used a total of three pins, the serial in, serial out, and 50Mhz clock, which makes this design easy to port from one FPGA to another.

The team created two separate integration tests to test that the final design of the FPGA works as intended. The first integration test checks that the FPGA is able to successfully validate previous correct blocks header data from the blockchain (Miner Top Level Integration Test 1) [13]. The team received this raw data and wrote it to a binary file using the Python script. This integration test then writes the data from the binary file to the FPGA after ordering it properly for the board. The software then polls the FPGA every second to determine a solution to the Hashcash algorithm has been found. If a solution is found, it is read from the FPGA, and ordered in the proper manner to be written back out to a file and printed to the terminal.

![Fig 21: Miner top level first integration test](image)

Here the miner instantly finds the correct result and returns this information in the exact order that would be seen on any other mining node. Using this integration test the team can calculate the difference in nonce values and time and see that the board is running at the expected speed.

The second integration test featured on simulating a real blockchain environment (Miner Top Level Integration Test 2). This test runs the real first 10 Bitcoin “genesis” blocks. These blocks use the lowest difficulty, with $2^{32}$ average iterations required for a block to be mined. The test board used by the team running at its downclocked speed of 5Mhz should find a solution on average every $2^{32} / (5 \times 2^{20}) = 819.2/60 = 13$ minutes. This is similar to the time required for an original genesis block to be mined, and at the faster clock rate this would take on average 52 seconds per run as opposed to 13 minutes. This integration test reads the data from a binary file including all ten block headers. The data is read and the time is overwritten with the current Linux epoch time. This makes it so that the board is finding a realistic solution to this Proof of Work algorithm. Since the difficulty of the network in this example is so low, blocks can be
erratic in the time that they take to be calculated. Our example output from this finds results within two minutes of each other, but as difficulty rises in the network these times become much more stable.

```
bash: 4.2$ /miner_top_level_stream
Init serial port:
Reading block information:
    Header: 0x91000000e5dc82a3a6e51a6117635622a956296ce064d9007f974fb71fa47b27a
C72c8e67769f976776b3a75a3fa9f8a4a4be4e9576f2631d101c0c2b7c
Sending block information to miner: done

block mined:
    Hash: 0x000000008e5dc82a3a6e51a6117635622a956296ce064d9007f974fb71fa47b27a
    Header: 0x91000000e5dc82a3a6e51a6117635622a956296ce064d9007f974fb71fa47b27a
C72c8e67769f976776b3a75a3fa9f8a4a4be4e9576f2631d101c0c2b7c
    Sending block information to miner: done

Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
inary time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Updating header time...
Block mined:
    Hash: 0x000000008e5dc82a3a6e51a6117635622a956296ce064d9007f974fb71fa47b27a
    Header: 0x91000000e5dc82a3a6e51a6117635622a956296ce064d9007f974fb71fa47b27a
C72c8e67769f976776b3a75a3fa9f8a4a4be4e9576f2631d101c0c2b7c
    Sending block information to miner: done
```

Fig 22: Beginning of miner top level stream

Fig 23: Activity in simulation of miner top level stream

Here the test finds two results right next to each other, and the team can tell that these are valid by looking at the amount of preceding zeros on the hash. With this the design of the FPGA
Based proof of work miner was finished. Additional projects in the future could hook this up to networking to allow the device to actually function as a portion of the Bitcoin network. This could be done through the use of a mining pool, or by downloading the entire blockchain and validating nodes on your host computer. This is a less common technique because fully mining a block is extremely rare with the current hash rate of the network, so sharing rewards in a mining pool often is the most efficient. Due to the late start of the project and Covid-19 the team did not have the time and resources available to do this.

**Evaluation:**

The team created a FPGA based mining implementation on the DE2-115 that is capable of an estimated 75Mh, as hash rate in the Bitcoin network is measured as double hash throughput. The DE2-115 development board has a maximum power consumption of 15 watts [7]. While the project likely did not utilize the full power of the DE2-115, the team will assume that it did due to the inability to measure the exact power consumption of the design. There are unused features on the board that would increase the power consumption that were unused, such as a secondary FPGA, and external circuitry. The cost of the DE2-115 board is $595 for non-education purposes and $309 for educational purposes. Since this board would be utilized for non-educational purposes as a miner, the team assumes the $595 price point [8]. With this information the performance of the DE2-115 can be compared to other types of architectures, with regards to performance, power consumption, and price. It is also important to note that the DE2-115 development board is not a new FPGA, and while the formal release date is not listed, copyright information dates back to 2003 for the device [9]. This means that when comparing with modern hardware, it is important to note that modern FPGA devices would likely be better in all three evaluation categories. As well as this, the use of central processing units (CPUs) and graphical processing units (GPUs) has fallen in popularity since the rise of ASICs, so the available information on these devices will also be out of date. The team took this into account with determining the data for comparison [10] [11].

<table>
<thead>
<tr>
<th>Device:</th>
<th>Type:</th>
<th>Hash Rate: (Mh)</th>
<th>Power Consumption (J):</th>
<th>Price ($)</th>
<th>MH/j</th>
<th>MH/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia GTX 680</td>
<td>GPU</td>
<td>127.3</td>
<td>100</td>
<td>$549.00</td>
<td>1.27</td>
<td>0.23</td>
</tr>
<tr>
<td>Nvidia GTX 580</td>
<td>GPU</td>
<td>156.6</td>
<td>244</td>
<td>$499.00</td>
<td>0.64</td>
<td>0.31</td>
</tr>
<tr>
<td>Nvidia GTX 295</td>
<td>GPU</td>
<td>117.3</td>
<td>289</td>
<td>$500.00</td>
<td>0.41</td>
<td>0.23</td>
</tr>
<tr>
<td>Radeon 7970x3</td>
<td>GPU</td>
<td>1950</td>
<td>750</td>
<td>$1,650.00</td>
<td>2.60</td>
<td>1.18</td>
</tr>
<tr>
<td>Radeon 5870</td>
<td>GPU</td>
<td>421</td>
<td>215</td>
<td>$379.00</td>
<td>1.96</td>
<td>1.11</td>
</tr>
<tr>
<td>Radeon 5970</td>
<td>GPU</td>
<td>820</td>
<td>347</td>
<td>$600.00</td>
<td>2.36</td>
<td>1.37</td>
</tr>
<tr>
<td>Radeon 6870</td>
<td>GPU</td>
<td>322</td>
<td>170</td>
<td>$239.00</td>
<td>1.89</td>
<td>1.35</td>
</tr>
<tr>
<td>Type</td>
<td>Avg MH/j</td>
<td>Avg MH/$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>----------</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU</td>
<td>1.59</td>
<td>$0.83</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>0.13</td>
<td>$0.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPGA</td>
<td>13.76</td>
<td>$0.83</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASIC</td>
<td>2,558.06</td>
<td>$1,929.62</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first table provides insight on the performance of specific components from different architectures. Here the team notes that FPGAs are much more efficient than both GPUs and CPUs in terms of power consumption, but close to GPUs in terms of performance per dollar. With this information FPGAs would be the preferred choice over GPUs and CPUs for proof of work based mining. This data is also useful to the team in comparing the performance of the finished DE2-115 based FPGA miner with comparison to other FPGA implementations. The power consumption estimate of 15 watts was set knowing that it may be overestimating the power, and these numbers are reflected in the lowest Mh/w of the group. These numbers are still
higher than the related GPUs and CPUs in the group. The main drawback to FPGA based mining here is the cost of FPGA boards. FPGAs are a relatively new technology still and have high price tags associated with components.

The performance in comparison to ASIC miners is drastically different. ASIC miners are substantially more efficient than their FPGA counterparts however ASICs take a significant amount of time to reach the market due to development and fabrication costs. During this development process FPGAs offer the greatest efficiency of all conventional options, while still offering similar levels of flexibility as provided by the more general purpose options such as GPUs and CPUs.

It is also important to note that some cryptocurrency networks are migrating to ASIC resistant proof of work algorithms. This is generally done to cut down on the waste created by development of ASIC devices, however it does mean that the traditionally most efficient option is not necessarily viable for a miner who wishes to change networks.

Despite the limited time frame and lack of resources, the team was able to design and implement a fully functional proof of work hasher based on the Bitcoin Hashcash proof of work algorithm. The final implementation was both highly efficient and cost effective when compared to its counterparts on general purpose devices. Though an ASIC implementation offers the greatest hash rates and efficiency when applicable, FPGAs stand as an obvious middle ground for when either an ASIC implementation is not possible or while an ASIC implementation is still under development.

Appendix:

Communication Module:

```verilog
// Frederich Stine, Alex Spaulding, and Jo Morrison
// UAH CPE495/496 Senior Design - FPGA Based Proof of Work Miner
// Verilog implementation - 2021

// Communication Module
// this module instantiates both the receive and transmit serial
// blocks into a single interface for communication
module communication_module(CLK_50, SER_IN, SER_OUT, block,
hash_in, header_in, ready_to_send, data_received);

// Wires and registers used by the module
input CLK_50;
input SER_IN;
input ready_to_send;
output SER_OUT;

output wire [639:0] block;
output wire data_received;
input [255:0] hash_in;
input [639:0] header_in;

// Instantiation of serial receive module
// This is available in the serial_receive block diagram
serial_receive(.CLK_50(CLK_50), .SER_IN(SER_IN), .block(block),
.data_received(data_received));

// Instantiation of serial transmit module
// This is available in the serial_transmit block diagram
serial_transmit(.CLK_50(CLK_50), .SER_IN(SER_IN), .header_in(header_in),
.ready_to_send(ready_to_send), .hash_in(hash_in), .SER_OUT(SER_OUT));

endmodule
```

Difficulty Calculator:

```verilog
// Frederich Stine, Alex Spaulding, and Jo Morrison
// UAH CPE495/496 Senior Design - FPGA Based Proof of Work Miner
// Verilog implementation - 2021

// Diff Calculator Module
// This module expands the big endian representation of the
```
"bits" field of the Bitcoin blockchain header into a comparable hash representation.

The difficulty is packed in a custom floating point format that is expanded by the module below.

```
module diff_calculator(diff_update, diff, CLK_50, diff_target);
// Wires and registers used by the module
input wire [31:0] diff;
input wire diff_update;
output reg [255:0] diff_target;
input wire CLK_50;

// Whenever new data appears, floating point representation is expanded
always @(posedge CLK_50) begin
    if(diff_update) begin
    end
end
```

Block Parser:

```
Frederich Stine, Alex Spaulding, and Jo Morrison
UAH CPE495/496 Senior Design - FPGA Based Proof of Work Miner
Verilog implementation - 2021

Block Generator Module
This module takes in a raw chunk of data and returns the different Bitcoin block header data in a parsed manner. Also includes functionality for information update on hardware clock timer update.
```

```
module block_parser(block, version, hash_prev, hash_merkle, epoch_time, difficulty, nonce, data_valid, CLK_50, data_received, clock_time, clock_valid);
// Wires and registers used by module
input [639:0] block;
input [31:0] clock_time;

output reg [31:0] version;
output reg [255:0] hash_prev;
output reg [255:0] hash_merkle;
```
output reg [31:0] epoch_time;
output reg [31:0] difficulty;
output reg [31:0] nonce;
output reg data_received;
input data_valid;
input clock_valid;
input CLK_50;

// Parsing block information whenever new data occurs
// or hardware clock updates
always @(posedge CLK_50) begin
    if(data_valid) begin
        version[31:0] <= block[31:0];
        hash_prev[255:0] <= block[287:32];
        hash_merkle[255:0] <= block[543:288];
        epoch_time[31:0] <= block[575:544];
        difficulty[31:0] <= block[607:576];
        nonce[31:0] <= block[639:608];
        data_received <= 1;
    end
    if(clock_valid) begin
        epoch_time <= clock_time;
        data_received <= 1;
    end
    if(data_received) begin
        data_received <= 0;
    end
end
endmodule

Miner Top Level:

// Frederich Stine, Alex Spaulding, and Jo Morrison
// UAH CPE495/496 Senior Design - FPGA Based Proof of Work Miner
// Verilog implementation - 2021

// Top level miner module
// This module is the final design for the FPGA based
// Proof of Work miner.
// This module incorporates the difficulty check module
// as well as the hash module.
module miner_top_level(CLK_50, SER_IN, SER_OUT);
// Wire and registers used by module
input CLK_50;
input SER_IN;
output SER_OUT;

reg ready_to_send = 0;
wire [31:0] version;
wire [255:0] hash_prev;
wire [255:0] hash_merkle;
wire [31:0] epoch_time;
wire [31:0] difficulty;
wire [31:0] initial_nonce;
wire [639:0] block;
wire [255:0] diff_target;
wire result_valid;
wire data_received;
wire data_parsed;
reg [255:0] hash_in;
wire [31:0] clock_time;
wire clock_valid;
reg [639:0] header_in;

wire [31:0] little_difficulty;
wire [31:0] little_epoch_time;
wire [31:0] little_nonce;
wire [639:0] raw_data;
wire [639:0] ordered_data;
reg [31:0] nonce;

// Instantiation of communication module
communication_module(.CLK_50(CLK_50), .SER_IN(SER_IN), .SER_OUT(SER_OUT),
.header_in(header_in),
.block(block), .hash_in(hash_in), .ready_to_send(ready_to_send),
.data_received(data_received));

// Instantiation of block parser module
block_parser(.block(block), .version(version), .hash_prev(hash_prev),
.hash_merkle(hash_merkle), .epoch_time(epoch_time),
.difficulty(difficulty), .nonce(initial_nonce),
.data_valid(data_received), .CLK_50(CLK_50),
.data_received(data_parsed), .clock_time(clock_time),
.clock_valid(clock_valid));

// Instantiation difficulty calculator module
diff_calculator diff_calc(.diff_update(data_parsed), .diff(difficulty),
.CLK_50(CLK_50), .diff_target(diff_target));

// Data rearrangement used by module
These rearrangements ensure that all data is properly formatted for
the specific ordering that is used by the hash module chosen,
as well as the endianness used by Bitcoin mining.

wire [255:0] little_hash_result;

assign little_nonce[7:0] = nonce[31:24];
assign little_nonce[15:8] = nonce[23:16];
assign little_nonce[23:16] = nonce[15:8];
assign little_nonce[31:24] = nonce[7:0];

assign little_difficulty[7:0] = difficulty[31:24];
assign little_difficulty[15:8] = difficulty[23:16];
assign little_difficulty[23:16] = difficulty[15:8];
assign little_difficulty[31:24] = difficulty[7:0];

assign little_epoch_time[7:0] = epoch_time[31:24];
assign little_epoch_time[15:8] = epoch_time[23:16];
assign little_epoch_time[23:16] = epoch_time[15:8];
assign little_epoch_time[31:24] = epoch_time[7:0];

// recreating full block
assign raw_data[31:0] = version[31:0];
assign raw_data[287:32] = hash_prev[255:0];
assign raw_data[543:288] = hash_merkle[255:0];
assign raw_data[575:544] = little_epoch_time[31:0];
assign raw_data[607:576] = little_difficulty[31:0];
assign raw_data[639:608] = little_nonce[31:0];

// Reorder data for hashing algorithm -- may need to change this around
assign ordered_data[31:0] = raw_data[31:0];
assign ordered_data[63:32] = raw_data[287:256];
assign ordered_data[95:64] = raw_data[255:224];
assign ordered_data[127:96] = raw_data[223:192];
assign ordered_data[159:128] = raw_data[191:160];
assign ordered_data[191:160] = raw_data[159:128];
assign ordered_data[223:192] = raw_data[127:96];
assign ordered_data[255:224] = raw_data[95:64];
assign ordered_data[287:256] = raw_data[63:32];
assign ordered_data[319:288] = raw_data[543:512];
assign ordered_data[351:320] = raw_data[511:480];
assign ordered_data[383:352] = raw_data[479:448];
assign ordered_data[415:384] = raw_data[447:416];
assign ordered_data[447:416] = raw_data[415:384];
assign ordered_data[479:448] = raw_data[383:352];
assign ordered_data[511:480] = raw_data[351:320];
assign ordered_data[543:512] = raw_data[319:288];
assign ordered_data[575:544] = raw_data[575:544];
assign ordered_data[607:576] = raw_data[607:576];
assign ordered_data[639:608] = raw_data[639:608];

// Swapping hash result endianness for use with result check
assign little_hash_result[7:0] = hash_result[31:24];
assign little_hash_result[15:8] = hash_result[23:16];
assign little_hash_result[23:16] = hash_result[15:8];
assign little_hash_result[31:24] = hash_result[7:0];

assign little_hash_result[39:32] = hash_result[63:56];
assign little_hash_result[47:40] = hash_result[55:48];
assign little_hash_result[55:48] = hash_result[47:40];
assign little_hash_result[63:56] = hash_result[39:32];

assign little_hash_result[71:64] = hash_result[95:88];
assign little_hash_result[79:72] = hash_result[87:80];
assign little_hash_result[87:80] = hash_result[79:72];
assign little_hash_result[95:88] = hash_result[71:64];

assign little_hash_result[103:96] = hash_result[127:120];
assign little_hash_result[111:104] = hash_result[119:112];
assign little_hash_result[119:112] = hash_result[111:104];
assign little_hash_result[127:120] = hash_result[103:96];

assign little_hash_result[135:128] = hash_result[159:152];
assign little_hash_result[143:136] = hash_result[151:144];
assign little_hash_result[151:144] = hash_result[143:136];
assign little_hash_result[159:152] = hash_result[135:128];

assign little_hash_result[167:160] = hash_result[191:184];
assign little_hash_result[175:168] = hash_result[183:176];
assign little_hash_result[183:176] = hash_result[175:168];
assign little_hash_result[191:184] = hash_result[167:160];

assign little_hash_result[199:192] = hash_result[223:216];
assign little_hash_result[207:200] = hash_result[215:208];
assign little_hash_result[215:208] = hash_result[207:200];
assign little_hash_result[223:216] = hash_result[199:192];

assign little_hash_result[231:224] = hash_result[255:248];
assign little_hash_result[239:232] = hash_result[247:240];
assign little_hash_result[247:240] = hash_result[239:232];
assign little_hash_result[255:248] = hash_result[231:224];

// Registers and wires used by the hash module Logic
reg CLK;
reg [4:0] div = 0;
reg new_data;
reg [9:0] count;
reg [255:0] state;
reg [511:0] data;
wire [255:0] hash;
wire [255:0] hash_result;
reg result_enable = 0;
reg [31:0] nonce_out;

// Driving logic for the hash module
// This takes care of calculating the midpoint of the first hash value
// when new data arrives, as well as checking the result and ensuring
// all output values are correct.
always @(posedge CLK_50) begin
    // Clock divider for remote testing and heat purposes
    if(div == 9) begin
        CLK = 1;
        div = 0;
    end
    else begin
        CLK = 0;
        div = div + 1;
    end

    // Checking for new data from the serial module on-full clock
    if(data_parsed) begin
        new_data = 1;
        nonce = initial_nonce;
    end

    // Driver action happens on the slower clocks rising edge
    if(CLK) begin
        // Resetting when new data arrives
        if(new_data == 1) begin
            new_data = 0;
            count = 0;
            result_enable = 0;
            state = 256'h5be0cd191f83d9ab9b05688c510e527fa54ff53a3c6ef372bb67ae856a09e667;
            data = ordered_data[511:0];
        end
        // Setting midstate hash value
        else if(count == 65) begin
            state = hash;
        end
    end
end
data = {384'h0000028000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000008000000, ordered_data[639:512]};
    count = count + 1;
    // Enabling result check
    else if(count == 194) begin
        result_enable = 1;
        count = count + 1;
        data = {384'h0000028000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000008000000, ordered_data[639:512]};
        nonce[31:0] = nonce[31:0] + 1;
    end
    // Normal operation
    else if(count == 195) begin
        data = {384'h0000028000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000008000000, ordered_data[639:512]};
        nonce[31:0] = nonce[31:0] + 1;
    end
    // Nonce value incrementation after midstate and normal operation calculations
    else if(count > 65) begin
        data = {384'h0000028000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000008000000, ordered_data[639:512]};
        nonce[31:0] = nonce[31:0] + 1;
        count = count + 1;
    end
    // Counter increment until midstate calculated
    else begin
        count = count + 1;
    end
    // Result checking
    // This is not done with the result module due to simplification
    // This checks the result with no cycle delay and can output the data and ready to send signals as needed.
    if(result_enable) begin
        if(little_hash_result < diff_target) begin
            header_in = raw_data;
            hash_in = hash_result;
            ready_to_send = 1;
        end
else begin
    ready_to_send = 0;
end
end

// First instantiation of hasher
sha256_transform #(1) hash1(
    .clk(CLK),
    .feedback(1'b0),
    .cnt(6'b0),
    .rx_state(state),
    .rx_input(data),
    .tx_hash(hash)
);

// Second instantiation of hasher
sha256_transform #(1) hash2(
    .clk(CLK),
    .feedback(1'b0),
    .cnt(6'b0),
    .rx_state(256'h5be0cd191f83d9ab9b05688c510e527fa54ff53a3c6ef372bb67ae856a09e667),
    .rx_input({256'h0000010000000000000000000000000000000000000000000000000000000000
              ,hash}),
    .tx_hash(hash_result)
);
endmodule

Miner Top Level Integration Test 1:

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include "fpga_serial_source_probe.h"
#include <time.h>

// dev/ttyS1
#define COM_PORT "/dev/ttyS1"
```c
#define BAUD (speed_t) B115200
#define time_pos 68
#define diff_pos 72
#define nonce_pos 76

// functions for data ordering
void swap_endian(unsigned char*);
void swap_endian_hash(unsigned char*);
void int_to_barray(int integer, unsigned char*);
int barray_to_int(unsigned char*);
void rearrange_hash(unsigned char*);

int main(int argc, char** argv)
{
    // initialize serial port with com port driver
    // Location & baud rate
    init_serial_port(COM_PORT, BAUD);
    printf("Init serial port:\n");

    // variables for data to be sent and received
    unsigned char header[80];
    unsigned char header_out[80];
    unsigned char hash[32];
    unsigned char char_in;

    // reading in block information
    FILE* fhandle;
    FILE* fhandle2;

    fhandle = fopen("block_header_stream", "rb");
    fhandle2 = fopen("block_output_stream", "wb");

    // initializing arrays to zero
    for(int i=0; i<80; i++)
    {
        header_out[i] = 0;
    }

    for(int i=0; i<32; i++)
    {
        hash[i] = 0;
    }

    // Resetting board
    fpga_strobe_put((unsigned char) 0);
}
for(int i=0; i<10; i++)
{
    fread(header, sizeof(header), 1, fhandle);

    printf("Reading block information: \n");
    // Outputting header data being sent to board
    printf("Header: 0x");
    for(int i=0; i<80; i++)
    {
        printf("%02x", header[i]);
    }
    printf("\n\n");
    // Rearranging data for the board
    swap_endian(&header[0]);
    swap_endian_hash(&header[4]);
    swap_endian_hash(&header[36]);

    int epochtime;
    epochtime = (int)time(NULL);
    int_to_barray(epochtime, &header[68]);

    // Sending block information with fpga_header_put
    printf("Sending block information to miner: ");
    fpga_header_put((unsigned char) 0, header);
    printf(" done\n\n");

    // polling every second for response from FPGA
    int ready = 0;
    unsigned int endtime;
    unsigned int updatetime = time(0) + 5;
    while(ready == 0)
    {
        // probing ready
        ready = fpga_byte_get((unsigned char) 0);
        endtime = time(0) + 1;
        if(updatetime < time(0))
        {
            printf("Updating header time... \n");
            epochtime = (int)time(NULL);
            int_to_barray(epochtime, &header[68]);
            fpga_header_put((unsigned char) 0, header);
            updatetime = time(0) + 5;
        }
        while(time(0) < endtime);
    }
// Read routine for FPGA
fpga_strobe_put((unsigned char) 0);
fpga_hash_get((unsigned char) 1, hash);
fpga_header_get((unsigned char) 2, header_out);
ready = 0;

// Reordering data so that it is stored in Little endian and printed in Big endian order
swap_endian(&header_out[0]);
swap_endian_hash(&header_out[4]);
swap_endian_hash(&header_out[36]);
swap_endian(&header_out[68]);
swap_endian(&header_out[72]);

// Subtracting offset from the nonce value used for correct header data
int nonce;
swap_endian(&header_out[76]);
nonce = barray_to_int(&header_out[76]);
nonce = nonce - 130;
int_to_barray(nonce, &header_out[76]);

// Rearranging the hash value that we get out of the hash module
rearrange_hash(hash);
swap_endian_hash(hash);

printf("Block mined:\n");

// Printing hash in Big endian
printf("Hash: ");
printf("0x");
for(int i=31; i>=0; i--)
    printf("%02x", hash[i]);
printf("\n");

// Printing header in Little endian
printf("Header: ");
printf("0x");
for(int i=0; i<80; i++)
    printf("%02x", header_out[i]);
printf("\n");

}
fclose(fhandle);
fclose(fhandle2);
function to convert integer into byte array that can be sent and received by our FPGA in the correct orientation
pass this function the integer to be converted and the address of an element in an array where the value should be placed

```c
void int_to_barray(int integer, unsigned char* barray)
{
    barray[0] = integer&0xFF;
    barray[1] = (integer>>8)&0xFF;
    barray[2] = (integer>>16)&0xFF;
    barray[3] = (integer>>24)&0xFF;
}
```

function to convert byte array back into an integer that is sent back from out FPGA in the correct orientation
pass this function the address of an element to be converted in the byte array and the int value is returned

```c
int barray_to_int(unsigned char* barray)
{
    int convertint;
    return convertint;
}
```

void swap_endian(unsigned char* barray)
{
    unsigned char swap;
    swap = barray[0];
    barray[0] = barray[3];
    barray[3] = swap;
    swap = barray[1];
    barray[1] = barray[2];
    barray[2] = swap;
}

void swap_endian_hash(unsigned char* barray)
{
    unsigned char swap;
    for(int i=0; i<16; i++)
    {
        swap = barray[i];
        ...
```c
barray[i] = barray[31-i];
barray[31-i] = swap;
}

void rearrange_hash(unsigned char* barray)
{
    unsigned char swap1, swap2, swap3, swap4;
    for(int i=0; i<4; i++)
    {
        swap1 = barray[(4*i)];
        swap2 = barray[(4*i)+1];
        swap3 = barray[(4*i)+2];
        swap4 = barray[(4*i)+3];

        barray[(4*i)] = barray[31-(4*i)-3];
        barray[(4*i)+1] = barray[31-(4*i)-2];
        barray[(4*i)+2] = barray[31-(4*i)-1];
        barray[(4*i)+3] = barray[31-(4*i)];

        barray[31-(4*i)-3] = swap1;
        barray[31-(4*i)-2] = swap2;
        barray[31-(4*i)-1] = swap3;
        barray[31-(4*i)] = swap4;
    }
}
```

Miner Top Level Integration Test 2:

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include "fpga_serial_source_probe.h"
#include <time.h>

#define COM_PORT "/dev/ttyS1"
#define BAUD (speed_t) B115200
#define time_pos 68
#define diff_pos 72
```
#define nonce_pos 76

// functions for data ordering
void swap_endian(unsigned char*);
void swap_endian_hash(unsigned char*);
void int_to_barray(int integer, unsigned char*);
int barray_to_int(unsigned char*);
void rearrange_hash(unsigned char*);

int main(int argc, char** argv)
{
    // initialize serial port with com port driver
    // Location & baud rate
    init_serial_port(COM_PORT, BAUD);
    printf("After init_serial_port\n");

    // variables for data to be sent and received
    unsigned char header[80];
    unsigned char header_out[80];
    unsigned char hash[32];
    unsigned char char_in;

    // reading in block information
    FILE* fhandle;
    fhandle = fopen("block_header_1", "rb");
    fread(header, sizeof(header), 1, fhandle);
    fclose(fhandle);
    printf("Reading block information: \n");
    printf("\n");

    // initializing arrays to zero
    for(int i=0; i<80; i++)
    {
        header_out[i] = 0;
    }
    for(int i=0; i<32; i++)
    {
        hash[i] = 0;
    }

    // Outputting header data being sent to board
    printf("0x");
    for(int i=0; i<80; i++)
{  printf("%02x", header[i]);
}
printf("\n\n");

// Rearranging data for the board
swap_endian(&header[0]);
swap_endian_hash(&header[4]);
swap_endian_hash(&header[36]);

// Resetting board
fpga_strobe_put((unsigned char) 0);

// Sending block information with fpga_header_put
printf("Sending block information to miner:");
fpga_header_put((unsigned char) 0, header);
printf(" done\n\n");

// polling every second for response from FPGA
int ready = 0;
unsigned int endtime;
while(ready == 0)
{
   // probing, ready
   ready = fpga_byte_get((unsigned char) 0);
   endtime = time(0) + 1;
   while(time(0) < endtime);
}

// Read routine for FPGA
fpga_strobe_put((unsigned char) 0);
fpga_hash_get((unsigned char) 1, hash);
fpga_header_get((unsigned char) 2, header_out);
ready = 0;

// Reordering data so that it is stored in little endian and
// printed in big endian order
swap_endian(&header_out[0]);
swap_endian_hash(&header_out[4]);
swap_endian_hash(&header_out[36]);
swap_endian(&header_out[68]);
swap_endian(&header_out[72]);

// Subtracting offset from the nonce value used for correct header data
int nonce;
swap_endian(&header_out[76]);
nonce = barray_to_int(&header_out[76]);
nonce = nonce-130;
int_to_barray(nonce, &header_out[76]);

// Rearranging the hash value that we get out of the hash module
rearrange_hash(hash);
swap_endian_hash(hash);

printf("Block mined: \n");

// Printing hash in big endian
printf("Hash: ");
printf("0x");
for(int i=31; i>=0; i--)
    printf("%02x", hash[i]);
printf("\n");

// Printing header in little endian
printf("Header: ");
printf("0x");
for(int i=0; i<80; i++)
    printf("%02x", header_out[i]);
printf("\n");

} // function to convert integer into byte array that can be sent and
// received by our FPGA in the correct orientation
// pass this function the integer to be converted
// and the address of an element in an array where the value should
// be placed
void int_to_barray(int integer, unsigned char* barray)
{
    barray[0] = integer&0xFF;
barray[1] = (integer>>8)&0xFF;
barray[2] = (integer>>16)&0xFF;
barray[3] = (integer>>24)&0xFF;
}

// function to convert byte array back into an integer
// that is sent back from out FPGA in the correct orientation
// pass this function the address of an element to be converted in the
// byte array and the int value is returned
int barray_to_int(unsigned char* barray)
{
    int convertint;
return convertint;

void swap_endian(unsigned char* barray)
{
    unsigned char swap;

    swap = barray[0];
    barray[0] = barray[3];
    barray[3] = swap;
    swap = barray[1];
    barray[1] = barray[2];
    barray[2] = swap;
}

void swap_endian_hash(unsigned char* barray)
{
    unsigned char swap;

    for(int i=0; i<16; i++)
    {
        swap = barray[i];
        barray[i] = barray[31-i];
        barray[31-i] = swap;
    }
}

void rearrange_hash(unsigned char* barray)
{
    unsigned char swap1, swap2, swap3, swap4;

    for(int i=0; i<4; i++)
    {
        swap1 = barray[(4*i)];
        swap2 = barray[(4*i)+1];
        swap3 = barray[(4*i)+2];
        swap4 = barray[(4*i)+3];

        barray[(4*i)] = barray[31-(4*i)-3];
        barray[(4*i)+1] = barray[31-(4*i)-2];
        barray[(4*i)+2] = barray[31-(4*i)-1];
        barray[(4*i)+3] = barray[31-(4*i)];

        barray[31-(4*i)-3] = swap1;
        barray[31-(4*i)-2] = swap2;
        barray[31-(4*i)-1] = swap3;
        barray[31-(4*i)] = swap4;
    }