FPGA Based Bitcoin Mining

By

Philip Dotemoto

Senior Project

Electrical Engineering Department

California Polytechnic State University

San Luis Obispo

June, 2014

Table of Contents

List of Figures and Tables

Acknowledgements

Introduction

Bitcoin is an experimental peer-to-peer digital currency based on public key cryptography. It was introduced by Satoshi Nakamoto in 2009 as a version of electronic cash that would allow payments to be sent from one party to another without going through a financial institution [1]. Traditionally, financial institutions, such as banks, are trusted to store and protect a customer's currency. The bank will handle the transfer of money between its customers and clients, but there are several disadvantages in this system. Electronic transfers between banks can be costly since there is usually a transaction fee, they can be slow taking several days to complete, and transfers cannot be made anonymously. Other payment processors such as Visa, MasterCard, and PayPal also charge fees that can cost several percent of the transaction. Bitcoin is a system of owning and transferring currency that omits these trusted third parties and instead relies on a peer-to-peer network to validate transactions and prevent double-spending.

The Bitcoin Network

Bitcoin relies on cryptographic proof instead of trusted third parties. Public key cryptography is used to make and verify digital signatures that users use to send payments. Let's suppose Alice and Bob are two users in the bitcoin network. Alice and Bob each have an address which is similar to a bank account number and tracks the number of bitcoins they have. The address is also associated with a public and private key. The private key is used to sign transactions when sending bitcoins while the public key can be used by anyone to validate the transaction signature.

Now, suppose Alice wants to send bitcoins to Bob.

1. Bob sends his address to Alice.
2. Alice adds Bob's address and the amount of bitcoins to a 'transaction' message.
3. Alice then signs the transaction message with her private key and announces her public key for signature verification.
4. Alice broadcast the transaction on the bitcoin network where all users can see the message.

All users on the Bitcoin network that know the transaction addresses belong to Alice and Bob can see that Alice has transferred bitcoins to Bob.

Later, Bob decides to transfer the same bitcoins to Charlie. Bob now repeats the steps Alice performed to send her bitcoin to Bob.

1. Charlie sends his address to Bob.
2. Bob adds Charlie's address and the amount of bitcoins to a 'transaction' message.
3. Bob signs the transaction message with his private key and announces his public key for signature verification.
4. Bob broadcast the transaction on the bitcoin network.

Another user, Eve cannot try to steal these bitcoins by replacing Bob or Charlie's address with her own. The transfers were signed with Alice and Bob's private key instructing that the coins were transferred

from Alice to Bob and then Bob to Charlie. Once Charlie accepts the coins, he also accepts that the coins were first passed from Alice to Bob, and then from Bob to him.

This record of transactions between Alice, Bob, and Charlie is added to a constantly growing chain of blocks that contains the record of all transactions on the bitcoin network. The record of transactions is maintained by the bitcoin network, and each block is validated with proof of work before it is accepted into the chain. Valid blocks are chained together so that the transfer of bitcoins can be tracked, and if one block is modified, all the following blocks will need to be recomputed with proof of work. Once the block containing Alice's transaction to Bob is added to the block chain, Bob can be confident that the transaction has been accepted by other computers in the network and permanently recorded. This prevents Alice from trying to send the same coins to another user and double spending her coins. The bitcoin network generates blocks every 10 minutes which would require Bob to wait at least this amount of time to be able to verify the transaction.

Only the single longest and fastest-growing block chain is considered valid to protect the bitcoin network from malicious attacks. The block chain is constantly growing since new blocks are validated every 10 minutes and a malicious user would need to control more than 50% of the network's computing power to be able to modify transactions. Without a significant portion of the network computing power, it's unlikely to be able to try and branch off from the valid block chain creating a separate malicious chain since the network will only accept the longest and fastest growing chain.

Proof of Work

The bitcoin network requires each block in the chain to include proof of work to ensure its validity. Proof of work is a piece of data that is difficult to produce due to being costly or time-consuming. Hashcash is the proof of work function used by the bitcoin network and uses two iterations of the secure-hash-algorithm-256 (SHA-256). Cryptographic hashes are designed to be hard to invert. It's simple to compute y from x, when y=H(x), but it's very difficult to find x only given y.

SHA-256

A hash function maps a message of an arbitrary length to a string of a fixed length, called the 'message hash' or 'message digest'. The compression processes of mapping the original message to the new fixed length message is known as 'hashing'.

The proof of work difficulty is adjusted to limit the rate of new block generation to every ten minutes. Since it is very difficult and improbable to completely reverse a secure hash in ten minutes, the hash must instead have a value less than the current target difficulty.

Example:
Let's take the hash of "Hello, world!". The target is to find a variation of the hash with a value beginning with '000'. The string is varied by adding an integer value to the end called a nonce and incrementing it until the target is satisfied.

```
"Hello, world!0" => 1312af178c253f84028d480a6adc1e25e81caa44c749ec81976192e2ec934c64
"Hello, world!1" => e9afc424b79e4f6ab42d99c81156d3a17228d6e1eef4139be78e948a9332a7d8
"Hello, world!2" => ae37343a357a8297591625e7134cbea22f5928be8ca2a32aa475cf05fd4266b7
...
"Hello, world!4248" => 6e110d98b388e77e9c6f042ac6b497cec46660deef75a55ebc7cfdf65cc0b965
"Hello, world!4249" => c004190b822f1669cac8dc37e761cb73652e7832fb814565702245cf26ebb9e6
"Hello, world!4250" => 0000c3af42fc31103f1fdc0151fa747ff87349a4714df7cc52ea464e12dcd4e9
```

Figure 1: Hash target example

In this example, the target is satisfied after 4251 hashes. Figure 1 shows the first three iterations of the hash, and the target is satisfied by the hash "Hello, world!5240". The difficulty can be increased by increasing the number of zeroes in the target since most computers can achieve millions of hashes per second. The current bitcoin network target at the time of this writing is shown below.

$$0.00000000000000000001980402864596927449480706603286383040$$

The National Institute of Standards and Technology (NIST) published the Secure Hash Standard [2] in 2002 that outlined three new Secure Hash Algorithms SHA-256, SHA-384, and SHA-512. SHA-224 was later added to form the SHA-2 family of hash functions. The SHA-256 hash algorithm produces a 256-bit message hash and consists of three stages. The first stage is message padding and parsing where a binary message is appended with '1' and padded with zeroes until its length is equal to 448 mod 512. The original message length is then appended as a 64-bit binary number. The padded message is parsed into N 512-bit blocks, denoted $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$. Each of the 512-bit blocks is then passed to the second stage, message expansion. The SHA-256 algorithm operates on 32-bit words, and each 512-bit $M^{(i)}$ block is broken down into 16 32-bit blocks denoted $M_t^{(i)}$, for $0 \leq t \leq 15$. The Message expander also expands each $M^{(i)}$ block into 64 32-bit W_t blocks, according to the equations:

$$\sigma_0(x) = ROT_7(x) \oplus ROT_{18}(x) \oplus SHF_3(x)$$

$$\sigma_1(x) = ROT_{17}(x) \oplus ROT_{19}(x) \oplus SHF_{10}(x)$$

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

$ROT_n(x)$ is a circular rotation of x by n positions to the right.

$SHF_n(x)$ is a right shift of x by n positions.

The final stage of SHA-256 is message compression. The W_i words from the expansion stage are input to the SHA compression function. The compression function has 8 32-bit working variables A, B, …, H, that are initialized to the first 32-bits of the fractional parts of the square roots of the first 8 primes ($H_0^{(0)}$-

$H_7^{(0)}$) at the start of each call to the hash function. The compression function is then iterated sixty-four times and outlined by:

$$T_1 = H + \sum\nolimits_1 (E) + Ch(E, F, G) + K_t + W_t$$
$$T_2 = \sum\nolimits_0 (A) + Maj(A, B, C)$$
$$H = G$$
$$F = E$$
$$D = C$$
$$B = A$$
$$G = F$$
$$E = D + T_1$$
$$C = B$$
$$A = T_1 + T_2$$

Where,

$$Ch(x, y, z) = (x \; AND \; y) \oplus (\bar{x} \; AND \; z)$$

$$Maj(x, y, z) = (x \; AND \; y) \oplus (x \; AND \; z) \oplus (y \; AND \; z)$$

$$\sum\nolimits_0 (x) = ROT_2(x) \oplus ROT_{13}(x) \oplus ROT_{22}(x)$$

$$\sum\nolimits_1 (x) = ROT_6(x) \oplus ROT_{11}(x) \oplus ROT_{25}(x)$$

The K_t inputs are 64 32-bit constants initialized from an array of the first 32 bits of the fractional parts of the cube roots of the first 64 primes. After sixty-four iterations of the compression function, an intermediate hash value $H^{(i)}$ is calculated:

$$H_0^{(i)} = A + H_0^{(i-1)}$$

$$H_1^{(i)} = B + H_1^{(i-1)}$$

$$H_2^{(i)} = C + H_2^{(i-1)}$$

$$H_3^{(i)} = D + H_3^{(i-1)}$$

$$H_4^{(i)} = E + H_4^{(i-1)}$$

$$H_5^{(i)} = F + H_5^{(i-1)}$$

$$H_6^{(i)} = G + H_6^{(i-1)}$$

$$H_7^{(i)} = H + H_7^{(i-1)}$$

The SHA-256 compression algorithm then repeats on the next 512-bit block from the padded message until all N data blocks are processed. The final 256-bit output, $H^{(N)}$, is formed by concatenating the final hash values:

$$H^N = H_0^{(N)} \& H_1^{(N)} \& H_2^{(N)} \& H_3^{(N)} \& H_4^{(N)} \& H_5^{(N)} \& H_6^{(N)} \& H_7^{(N)}$$

SHA-256 hash example

1 . Pad message to be hashed in a way that the result is a multiple of 512 bits long

    a. With message M of length, in bits, L, append "1" bit to the end of the message. Then, append k zero bits, where k is the smallest non-negative solution to L+1+k = 448 mod 512. Finally, append the 64 bit block that is equal to the number L in binary

    b. Example, (8 bit ASCII) message "abc" is shown in Table 1.

    length, L = 8*3 = 24

    L+1+k = 24+1+k = 448

    k = 448 – (24+1) = 423 zero bits

<p align="center">Table 1: Padded message "abc"</p>

01100001	01100010	01100011	1	00…0	00…011000
a, 8 bits	b, 8 bits	c, 8 bits	"1" bit pad	Pad 423 bits	L, 64 bits

    Padded message length is a multiple of 512 bits.

2. Parse the massage into 512 bit message blocks $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$

3. Processes message blocks one at a time beginning with a fixed initial has value $H^{(0)}$, sequentially compute

$$H^{(i)} = H^{(i-1)} + C_{M(i)} (H^{(i-1)}), \text{ for i = 1, 2, …, N}$$

C is the SHA-256 compression function

+ is a word-wise mod 2^{32} addition

$H^{(i)}$ is the hash of the block $M^{(i)}$

Initial hash values $H^{(0)}$ are the fractional parts of the square roots of the first eight primes.

Block Header

The block header is constantly hashed to generate bitcoins. A block header, shown in Table 2 and Figure 2Figure 3, contains the following fields.

Table 2: Block header fields

Field	Purpose	Updated When	Size (Bytes)
Version	Block version number	When software is upgraded, a new version is specified	4
hashPrevBlock	256-bit hash of the previous block header	A new block comes in	32
hashMerkleRoot	256-bit hash based on all of the transactions in the block	A transaction is accepted	32
Time	Current timestamp as seconds since 1970-01-01T00:00 UTC	Every few seconds	4
Difficulty	Current target in compact format	The difficulty is adjusted	4
Nonce	32-bit number (starts at 0)	A hash is tried (increments)	4

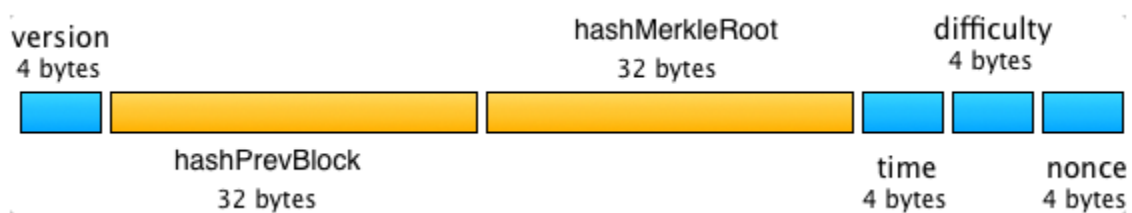The block header is an 80 byte value.

Figure 2: Block header size

Mining

Mining is the process of spending computation power to secure bitcoin transactions against reversal and to introduce new bitcoins to the system. The incentive to mine bit coins and validate transactions is currently driven by the possibility of receiving new bitcoins when a block is validated. The reward serves the purpose of distributing new coins in a decentralized way and to motivate bitcoin users to keep securing transactions on the network. The mining reward is 25 bitcoins, but this value is halved ever 210,000 blocks to control the currency supply.

The rate of block creation is constant at six per hour. This rate is controlled by the difficulty of hashing and the number of bitcoins generated per block is set to decrease geometrically, with a 50% reduction every four years. This limits the maximum number of bitcoins in the system to 21 million. The 50% reduction algorithm is assumed to be based on the approximate rate at which other commodities, such as gold, are mined. The 21 million ($2.1*10^{15}$) is also close to the maximum value of a 64-bit floating point number. There are concerns about deflation with the fixed monetary base, but bitcoins can be divided down to eight decimal places allowing 0.00000001 quantities of BTC to be traded. The bitcoin protocol can also be modified to handle smaller amounts in the future [6].

Mining involves hashing the block header until a hash value is found to be less than the current target. When a hash value is found, this proof of work validates the new block and the miner gets newly generated bitcoins. If a valid hash is not found, the miner tries a new nonce, and recalculates the hash.

A simplified mining algorithm is shown in Figure 3.

```
block_header = <version + prev_block + merkle_root + timestamp + bits>
nonce = 0
hash = 1
target = 0.000123456789

while ( hash > target )
{
        hash = SHA256( SHA256( nonce + block_header ) )
        nonce++
}
```

Figure 3: Simplification of mining algorithm

Field Programmable Gate Array

A field programmable gate array (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing. The FPGA configuration is specified by a hardware description language (HDL) to implement custom logical functions similar to any application specific integrated circuit (ASIC). FPGAs have the advantage of being updatable with new designs and having low non-recurring development cost relative to ASICs. FPGAs disadvantage is their higher per unit cost relative to ASICs when used in large quantities.

FPGAs have advantages in bitcoin mining due to their lower power usage and higher levels of customization when compared to other commercial off the shelf (COTS) hardware. When bitcoin was first introduced, central processing units (CPUs) from Intel and AMD were used as miners, but they were quickly replaced by graphics processing units (GPUs) from Nvidia and AMD. CPUs have relatively few arithmetic logic units (ALUs) and are designed to run more general executive and decision making software. GPUs have the ability to perform lots of repetitive work because they contain large numbers of ALUs designed to increase their ability to calculate the mathematical formulas to drive pixels on a screen. These same ALUs can be repurposed to repeatedly try different hashes, and the number of ALUs has a direct effect on the hash output. FPGAs can be configured to compute the SHA-256 algorithm with even more efficiency since their hardware is developed for this task.

Objectives

The main objectives of this project are as follows.

1. Learn about the bitcoin network and payment system
2. Study bitcoin mining algorithm and SHA-256
3. Compare the advantages of implementing bitcoin mining in hardware versus software
4. Test performance of the open source FPGA Bitcoin miner on Altera DE2-115 development board (Cyclone IV EP4CE115F29C7)
5. Mine block data from the bitcoin network
6. Identify areas for improvement to increase the miner's hash rate

Requirements

The main function of the miner is to run the SHA-256 algorithm on the block header to produce a valid proof of work. Table 3 breaks down the functional requirements derived from this objective.

Table 3: Miner requirements and specifications

Marketing Requirements	Engineering Specifications	Justification
1	The miner shall be able to retrieve header information from the bitcoin network and submit valid proof of work.	The miner will need to be able to retrieve block header information to hash and send valid hashes back to the bitcoin network to receive any rewards.
1	The bitcoin mining algorithm (double SHA-256 hash) shall be implemented on a commercial off the shelf (COTS) FPGA.	An FPGA with enough resources to implement the complete mining algorithm needs to be chosen.
2	The miner shall use standard hardware interfaces and connectors.	The miner will need to be easily setup by most users and those without extensive knowledge of the system.
2	The miner shall not require the user to interface with the system at the hardware description level for basic setup.	Users without prior knowledge of FPGAs should be able to use the miner.
Marketing Requirements		
1. The system shall implement a bitcoin miner on an FPGA.		
2. The system shall be easy to interface and setup.		

Design

The Open-Source FPGA Bitcoin Miner was used as the foundation for this project since it is already supported by other users in the bitcoin community [3]. The open source miner also supports solo mining or pools and both Xilinx and Altera devices. This project's goal is to understand the open source miner implementation and identify areas for improvement.

Software

The Open-Source FPGA Bitcoin Miner includes scripts to program supported FPGAs and interfaces to the bitcoin network. This software primarily retrieves work from the bitcoin network and sends it to the FPGA miner.

Hardware

The Altera DE2-115, Digilent ZYBO Zynq-7000, and Digilent Nexys-2 development boards were considered based on their capabilities and cost. The Digilent Nexys-2 and Altera DE2-115 development boards were available from their use in prior coursework. The Digilent ZYBO Zynq-7000 was also a possible alternative since it has a XILINX All Programmable System-on-Chip (AP SoC) that integrates a dual-core ARM Cortex-A9 processor with Xilinx series 7 FPGA logic. The Altera DE2-115, shown in Figure 4, was chosen because the Cyclone IV has the highest number of programmable logic elements. Development boards based on newer and higher density FPGAs were considered, but their high cost prevented their use in this project. Table 4 shows a breakdown of the development boards considered for this project.

Table 4: Development board comparison

Development Board	FPGA / Processor	Logic Count	Memory	Cost
Terasic Altera DE2-115	Cyclone IV EP4CE115F29C7	114,480 Logic Elements (LEs)	128 MB SDRAM, 2 MB SRAM, 8 MB Flash	$299 (Academic) $595 (Commercial)
Digilent ZYBO Zynq-7000	XILINX 650Mhz dual-core Cortex-A9 + Equivalent reprogrammable logic to Atrix-7 FPGA	28K logic cells	512MB x32 DDR3 RAM, 128Mb Serial Flash	$125 (Academic) $189 (Commercial)
Digilent Nexys-2	XILINX Spartan3E-500 FG320	500K gates	16MB SDRAM, 16MB Flash ROM	$129 (Academic) $169 (Commercial)
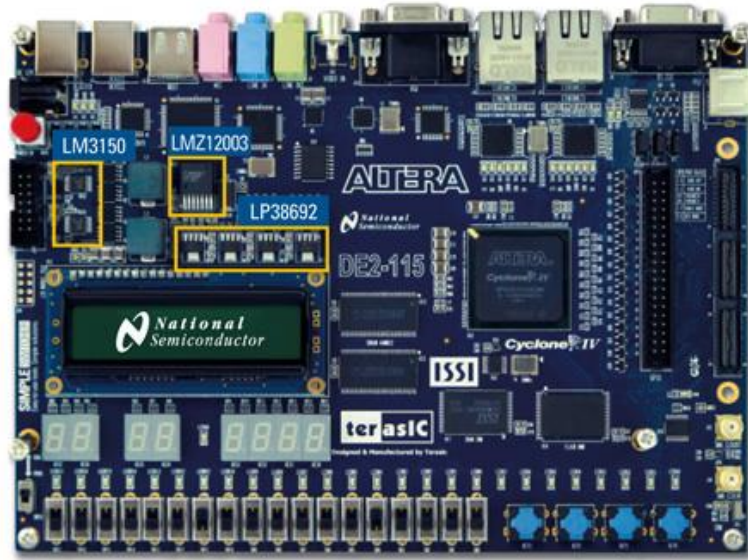
Figure 4: Altera DE2-115

fpgaminer_top

The fpgaminer_top pads the block header to be hashed so that it is a multiple of 512 bits for the SHA-256 algorithm. The nonce is the only field that is updated for each hash so the first 64 bytes of the block header remain constant. The constant portion of the header is hashed once and stored as the midstate value for subsequent hash iterations. The remaining 16 bytes of the header contain the nonce and are constantly hashed since the nonce is incremented each iteration. The midstate_buf and state registers holds the initial midstate value while the data_buf register holds the rest of the 16 byte header information.

The fpgaminer_top also includes two serial sha256_transform modules that perform the double SHA-256 hash as specified in the Hashcash proof of work function. The state and data register values are passed into the first sha256_transform module. The nonce is then updated while the hashes are being performed. When the nonce is incremented, it replaces the previous nonce value in the data register. This ensures that the miner is constantly trying new nonce values until it finds a hash result that satisfies the target. When the second SHA-256 hash is completed, its output is compared with the difficult and if the required number of trailing zeroes match, the is_golden_ticket register is set. Since the nonce is updated while the SHA-256 hashes are being performed, the current nonce value will be larger than the one that actually produced the target hash. An offset is subtracted from the current nonce value to get the golden_nonce that resulted in the target hash. A block diagram of the fpgaminer_top is shown in Figure 5.
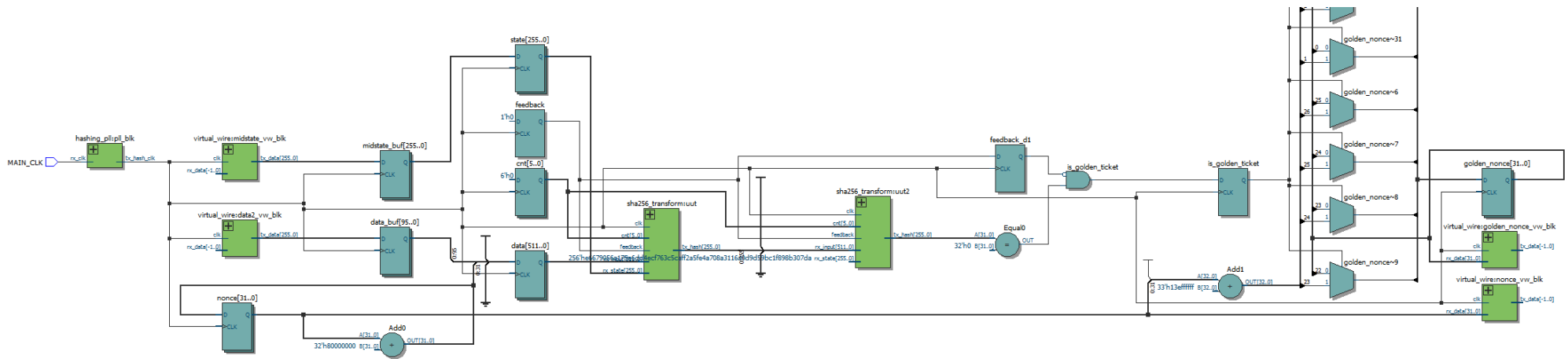
Figure 5: FPGA miner top

SHA-256 compression function

The SHA-256 compression function is run for 64 iterations on each 512-bit block for the padded message. Figure 6 shows one iteration of the SHA-256 compression function.
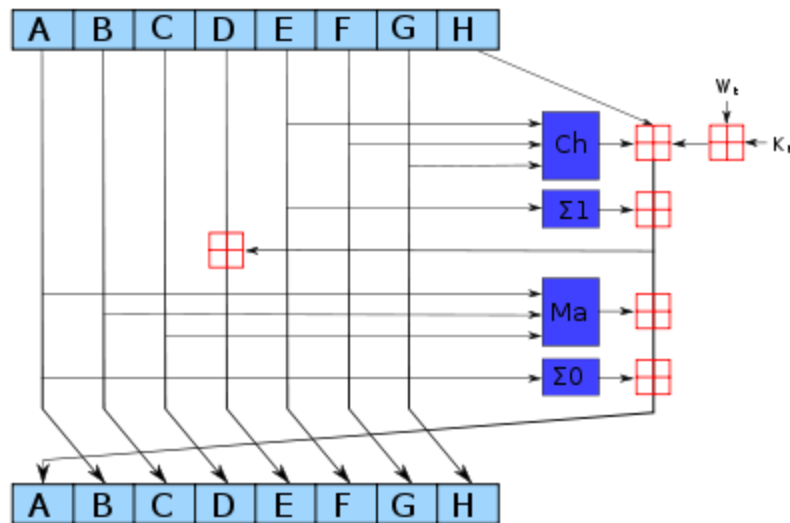
Figure 6: SHA-2 compression function

Red ⊞ is addition modulo 2^{32}

Ch Block

Ch(E,F,G) = (E AND F) XOR (!E AND G)

Figure 7 shows the ch block module implementation. Table 5 is the associated truth table for the Ch function and shows (E AND F) XOR (!E AND G) is equivalent to G XOR (E AND (F XOR G)).

```verilog
module ch (x, y, z, o);

    input [31:0] x, y, z;
    output [31:0] o;

    assign o = z ^ (x & (y ^ z));

endmodule
```

Figure 7: Ch block Verilog implementation

assign o = G XOR (E AND (F XOR G))

18

E	F	G	Ch
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Ma Block

Ma(A,B,C) = (A AND B) XOR (A AND C) XOR (B AND C)

Figure 8 shows the Ma block module implementation. Table 6 is the truth table for the Ma function and shows (A AND B) XOR (A AND C) XOR (B AND C) is equivalent to (A AND B) OR (C AND (A OR B)).

```
module maj (x, y, z, o);

      input [31:0] x, y, z;
      output [31:0] o;

      assign o = (x & y) | (z & (x | y));

endmodule
```

Figure 8: Ma block Verilog implementation

assign o = (A AND B) OR (C AND (A OR B))

Table 6: Ma Truth Table

A	B	C	Ma
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$\sum_0$ *Block*

>>> is a logical right rotate

$\sum_0(A)$ = (A >>> 2) XOR (A >>> 13) XOR (A >>> 22)

Figure 9 shows the $\sum_0$ block implementation.

```verilog
module e0 (x, y);

     input [31:0] x;
     output [31:0] y;

     assign y = {x[1:0],x[31:2]} ^ {x[12:0],x[31:13]} ^ {x[21:0],x[31:22]};

endmodule
```

Figure 9: $\sum$0 block Verilog implementation

The assign statement in Figure 9 is equivalent to the following

$$\text{assign } y = (x >>> 2) \text{ XOR } (x >>> 13) \text{ x OR } (x >>> 22)$$

$\sum_1$ Block

$\sum_1(E) = (E >>> 6) \text{ XOR } (E >>> 11) \text{ XOR } (E >>> 25)$

Figure 10 shows the $\sum_1$ block implementation.

```verilog
module e1 (x, y);

     input [31:0] x;
     output [31:0] y;

     assign y = {x[5:0],x[31:6]} ^ {x[10:0],x[31:11]} ^ {x[24:0],x[31:25]};

endmodule
```

Figure 10: $\sum$1 block Verilog implementation

The assign statement in Figure 10 is equivalent to the following

$$\text{assign } y = (x >>> 6) \text{ XOR } (x >>> 11) \text{ XOR } (x >>> 25)$$

s0 Block

s^n = right rotation by n bits
R^n = right shift by n bits
$\sigma_0(x) = s^7(x) \text{ XOR } s^{10}(x) \text{ XOR } R^3(x)$

Figure 11 shows the s0 block implementation.

```verilog
module s0 (x, y);

     input [31:0] x;
     output [31:0] y;

     assign y[31:29] = x[6:4] ^ x[17:15];
     assign y[28:0] = {x[3:0], x[31:7]} ^ {x[14:0],x[31:18]} ^ x[31:3];

endmodule
```

Figure 11: s0 Block Verilog Implementation

s1 Block

$\sigma_1(x) = s^{17}(x)$ XOR $s^{19}(x)$ XOR $R^{10}(x)$

Figure 12 shows the s1 block implementation.

```verilog
module s1 (x, y);

    input [31:0] x;
    output [31:0] y;

    assign y[31:22] = x[16:7] ^ x[18:9];
    assign y[21:0] = {x[6:0],x[31:17]} ^ {x[8:0],x[31:19]} ^ x[31:10];

endmodule
```

Figure 12: s1 Block Verilog Implementation
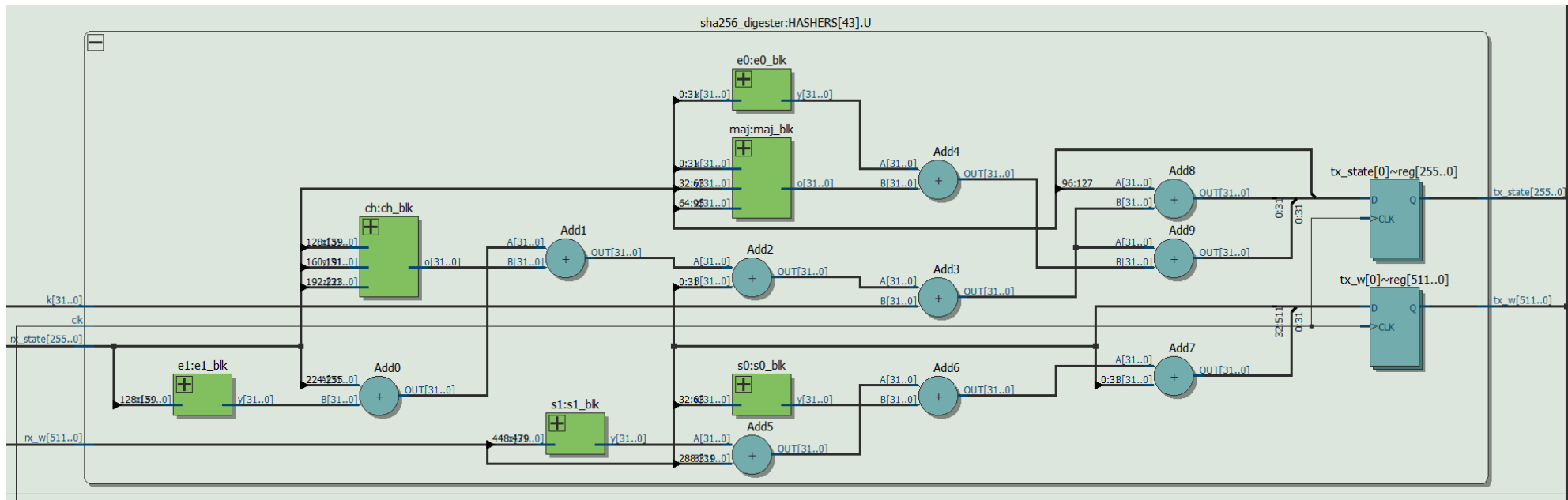
Figure 13: SHA-256 compression block

Figure 15 is the block diagram for the SHA-256 compression function.

The IDX(x) function, shown in Figure 14, gets the 32-bit word at index x.

Example:
IDX(0) = [31:0]
IDX(1) = [63:32]

```
// A quick define to help index 32-bit words inside a larger register.
`define IDX(x) (((x)+1)*(32)-1):((x)*(32))
```

Figure 14: IDX function

sha256_digester module

The sha256_digester module, shown in Figure 15, implements the compression function in Figure 6.

```verilog
module sha256_digester (clk, k, rx_w, rx_state, tx_w, tx_state);

        input clk;
        input [31:0] k;
        input [511:0] rx_w;
        input [255:0] rx_state;

        output reg [511:0] tx_w;
        output reg [255:0] tx_state;


        wire [31:0] e0_w, e1_w, ch_w, maj_w, s0_w, s1_w;


        e0      e0_blk (rx_state[`IDX(0)], e0_w);
        e1      e1_blk (rx_state[`IDX(4)], e1_w);
        ch      ch_blk (rx_state[`IDX(4)], rx_state[`IDX(5)], rx_state[`IDX(6)], ch_w);
        maj     maj_blk        (rx_state[`IDX(0)], rx_state[`IDX(1)], rx_state[`IDX(2)], maj_w);
        s0      s0_blk (rx_w[63:32], s0_w);
        s1      s1_blk (rx_w[479:448], s1_w);

        wire [31:0] t1 = rx_state[`IDX(7)] + e1_w + ch_w + rx_w[31:0] + k;
        wire [31:0] t2 = e0_w + maj_w;
        wire [31:0] new_w = s1_w + rx_w[319:288] + s0_w + rx_w[31:0];


        always @ (posedge clk)
        begin
                tx_w[511:480] <= new_w;
                tx_w[479:0] <= rx_w[511:32];

                tx_state[`IDX(7)] <= rx_state[`IDX(6)];
                tx_state[`IDX(6)] <= rx_state[`IDX(5)];
                tx_state[`IDX(5)] <= rx_state[`IDX(4)];
                tx_state[`IDX(4)] <= rx_state[`IDX(3)] + t1;
                tx_state[`IDX(3)] <= rx_state[`IDX(2)];
                tx_state[`IDX(2)] <= rx_state[`IDX(1)];
                tx_state[`IDX(1)] <= rx_state[`IDX(0)];
                tx_state[`IDX(0)] <= t1 + t2;
        end

endmodule
```

Figure 15: sha256_digester module

Inputs

k is a constant initialized from an array of the first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311. The values are in hexadecimal and shown in Figure 16.

```verilog
        // Constants defined by the SHA-2 standard.
        localparam Ks = {
            32'h428a2f98, 32'h71374491, 32'hb5c0fbcf, 32'he9b5dba5,
            32'h3956c25b, 32'h59f111f1, 32'h923f82a4, 32'hab1c5ed5,
            32'hd807aa98, 32'h12835b01, 32'h243185be, 32'h550c7dc3,
            32'h72be5d74, 32'h80deb1fe, 32'h9bdc06a7, 32'hc19bf174,
            32'he49b69c1, 32'hefbe4786, 32'h0fc19dc6, 32'h240ca1cc,
            32'h2de92c6f, 32'h4a7484aa, 32'h5cb0a9dc, 32'h76f988da,
            32'h983e5152, 32'ha831c66d, 32'hb00327c8, 32'hbf597fc7,
            32'hc6e00bf3, 32'hd5a79147, 32'h06ca6351, 32'h14292967,
            32'h27b70a85, 32'h2e1b2138, 32'h4d2c6dfc, 32'h53380d13,
            32'h650a7354, 32'h766a0abb, 32'h81c2c92e, 32'h92722c85,
            32'ha2bfe8a1, 32'ha81a664b, 32'hc24b8b70, 32'hc76c51a3,
            32'hd192e819, 32'hd6990624, 32'hf40e3585, 32'h106aa070,
            32'h19a4c116, 32'h1e376c08, 32'h2748774c, 32'h34b0bcb5,
            32'h391c0cb3, 32'h4ed8aa4a, 32'h5b9cca4f, 32'h682e6ff3,
            32'h748f82ee, 32'h78a5636f, 32'h84c87814, 32'h8cc70208,
            32'h90befffa, 32'ha4506ceb, 32'hbef9a3f7, 32'hc67178f2};
```

Figure 16: Ks, round constants array

rx_w is initialized to zero and updated based on the next nonce when new data is input into the hasher.

rx_state is initialized to the initial hash values. The initial hash values, shown in Figure 17, are the first 32 bits of the fractional parts of the square roots of the first 8 primes 2…19.

$H_1(0)$ = 6a09e667 → register a
$H_2(0)$ = bb67ae85 → register b
$H_3(0)$ = 3c6ef372 → register c
$H_4(0)$ = a54ff53a → register d
$H_5(0)$ = 510e527f → register e
$H_6(0)$ = 9b05688c → register f
$H_7(0)$ = 1f83d9ab → register g
$H_8(0)$ = 5be0cd19 → register h

```verilog
.rx_state(256'h5be0cd191f83d9ab9b05688c510e527fa54ff53a3c6ef372bb67ae856a09e667),
```

Figure 17: Initial hash values

Outputs

tx_w contains the message blocks.

tx_state contains the updated registers a through h after the SHA-256 compression function is applied.

SHA-256 Compression Function

The input to e0_blk is the first 32-bit word of rx_state which is initialized to $H_1(0)$ = 6a09e667 = register a.

The output of e0_blk is e0_w. Figure 18 shows the e0_blk assignment statement.

24

```
e0      e0_blk (rx_state[`IDX(0)], e0_w);
```
Figure 18: e0 block output

The input to e1_blk is the fifth 32-bit word of rx_state which is initialized to $H_5(0)$ = 510e527f → register e.

The output of e1_blk is e1_w. Figure 19 shows the e1_blk assignment statement.

```
e1      e1_blk (rx_state[`IDX(4)], e1_w);
```
Figure 19: e1 block output

The inputs to ch_blk are the fifth, sixth, and seventh 32-bit words of rx_state. These are initially $H_5(0)$ = 510e527f = register e, $H_6(0)$ = 9b05688c = register f, and $H_7(0)$ = 1f83d9ab = register g respectively.

The output of ch_blk is ch_w. Figure 20 shows the ch_block assignment statement.

```
ch      ch_blk (rx_state[`IDX(4)], rx_state[`IDX(5)], rx_state[`IDX(6)], ch_w);
```
Figure 20: ch block output

The inputs to maj_blk are the first, second, and third 32-bit words of rx_state. These are initially $H_1(0)$ = 6a09e667 = register a, $H_2(0)$ = bb67ae85 = register b, and $H_3(0)$ = 3c6ef372 = register c respectively.

The output of maj_blk is maj_w. Figure 21 shows the maj block assignment statement.

```
maj     maj_blk      (rx_state[`IDX(0)], rx_state[`IDX(1)], rx_state[`IDX(2)], maj_w);
```
Figure 21: maj block output

The input to s0_blk is the second 32-bit word of rx_w.

The output of s0_blk is s0_w. Figure 22 shows the s0_blk assignment statement.

```
s0      s0_blk (rx_w[63:32], s0_w);
```
Figure 22: s0 block output

The input to s1_blk is the fifteenth 32-bit word of rx_w.

The output of s1_blk is s1_w. Figure 23 shows the s1_blk assignment statement.

```
s1      s1_blk (rx_w[479:448], s1_w);
```
Figure 23: s1 block output

t1 is the sum of register H, the Ch block, s1 block, the first 32-bit word of rx_w, and k. Figure 24 shows the t1 assignment statement.

```
wire [31:0] t1 = rx_state[`IDX(7)] + e1_w + ch_w + rx_w[31:0] + k;
```
Figure 24: t1 intermediate sum

t2 is the sum of the $\sum_0$ block and the Ma block. Figure 25 shows the t2 assignment statement.

```
wire [31:0] t2 = e0_w + maj_w;
```
Figure 25: t2 intermediate sum

The tx_w output is the sum of the s1 block, tenth 32-bit word of rx_w, s0 block, and first 32-bit word of rx_w. Figure 26 shows the tx_w assignment statement.

```verilog
wire [31:0] new_w = s1_w + rx_w[319:288] + s0_w + rx_w[31:0];
```
Figure 26: tx_w output

Registers a through h are updated. Figure 27 shows the Verilog implementation.

h ← g
g ← h
f ← e
e ← d + t1
d ← c
c ← b
b ← a
a ← t1 + t2

```verilog
always @ (posedge clk)
begin
        tx_w[511:480] <= new_w;
        tx_w[479:0] <= rx_w[511:32];

        tx_state[`IDX(7)] <= rx_state[`IDX(6)];
        tx_state[`IDX(6)] <= rx_state[`IDX(5)];
        tx_state[`IDX(5)] <= rx_state[`IDX(4)];
        tx_state[`IDX(4)] <= rx_state[`IDX(3)] + t1;
        tx_state[`IDX(3)] <= rx_state[`IDX(2)];
        tx_state[`IDX(2)] <= rx_state[`IDX(1)];
        tx_state[`IDX(1)] <= rx_state[`IDX(0)];
        tx_state[`IDX(0)] <= t1 + t2;
end
```
Figure 27: SHA-256 compression function register update

sha256_transform module

```verilog
// Perform a SHA-256 transformation on the given 512-bit data, and 256-bit
// initial state,
// Outputs one 256-bit hash every LOOP cycle(s).
//
// The LOOP parameter determines both the size and speed of this module.
// A value of 1 implies a fully unrolled SHA-256 calculation spanning 64 round
// modules and calculating a full SHA-256 hash every clock cycle. A value of
// 2 implies a half-unrolled loop, with 32 round modules and calculating
// a full hash in 2 clock cycles. And so forth.
module sha256_transform #(
        parameter LOOP = 6'd4
) (
        input clk,
        input feedback,
        input [5:0] cnt,
        input [255:0] rx_state,
        input [511:0] rx_input,
        output reg [255:0] tx_hash
);

        // Constants defined by the SHA-2 standard.
        localparam Ks = {
                32'h428a2f98, 32'h71374491, 32'hb5c0fbcf, 32'he9b5dba5,
                32'h3956c25b, 32'h59f111f1, 32'h923f82a4, 32'hab1c5ed5,
                32'hd807aa98, 32'h12835b01, 32'h243185be, 32'h550c7dc3,
                32'h72be5d74, 32'h80deb1fe, 32'h9bdc06a7, 32'hc19bf174,
                32'he49b69c1, 32'hefbe4786, 32'h0fc19dc6, 32'h240ca1cc,
```

```verilog
                32'h2de92c6f, 32'h4a7484aa, 32'h5cb0a9dc, 32'h76f988da,
                32'h983e5152, 32'ha831c66d, 32'hb00327c8, 32'hbf597fc7,
                32'hc6e00bf3, 32'hd5a79147, 32'h06ca6351, 32'h14292967,
                32'h27b70a85, 32'h2e1b2138, 32'h4d2c6dfc, 32'h53380d13,
                32'h650a7354, 32'h766a0abb, 32'h81c2c92e, 32'h92722c85,
                32'ha2bfe8a1, 32'ha81a664b, 32'hc24b8b70, 32'hc76c51a3,
                32'hd192e819, 32'hd6990624, 32'hf40e3585, 32'h106aa070,
                32'h19a4c116, 32'h1e376c08, 32'h2748774c, 32'h34b0bcb5,
                32'h391c0cb3, 32'h4ed8aa4a, 32'h5b9cca4f, 32'h682e6ff3,
                32'h748f82ee, 32'h78a5636f, 32'h84c87814, 32'h8cc70208,
                32'h90befffa, 32'ha4506ceb, 32'hbef9a3f7, 32'hc67178f2};


        genvar i;

        generate

                for (i = 0; i < 64/LOOP; i = i + 1) begin : HASHERS
                        wire [511:0] W;
                        wire [255:0] state;

                        if(i == 0)
                                sha256_digester U (
                                        .clk(clk),
                                        .k(Ks[32*(63-cnt) +: 32]),
                                        .rx_w(feedback ? W : rx_input),
                                        .rx_state(feedback ? state : rx_state),
                                        .tx_w(W),
                                        .tx_state(state)
                                );
                        else
                                sha256_digester U (
                                        .clk(clk),
                                        .k(Ks[32*(63-LOOP*i-cnt) +: 32]),
                                        .rx_w(feedback ? W : HASHERS[i-1].W),
                                        .rx_state(feedback ? state : HASHERS[i-1].state),
                                        .tx_w(W),
                                        .tx_state(state)
                                );
                end

        endgenerate

        always @ (posedge clk)
        begin
                if (!feedback)
                begin
                        tx_hash[`IDX(0)] <= rx_state[`IDX(0)] + HASHERS[64/LOOP-6'd1].state[`IDX(0)];
                        tx_hash[`IDX(1)] <= rx_state[`IDX(1)] + HASHERS[64/LOOP-6'd1].state[`IDX(1)];
                        tx_hash[`IDX(2)] <= rx_state[`IDX(2)] + HASHERS[64/LOOP-6'd1].state[`IDX(2)];
                        tx_hash[`IDX(3)] <= rx_state[`IDX(3)] + HASHERS[64/LOOP-6'd1].state[`IDX(3)];
                        tx_hash[`IDX(4)] <= rx_state[`IDX(4)] + HASHERS[64/LOOP-6'd1].state[`IDX(4)];
                        tx_hash[`IDX(5)] <= rx_state[`IDX(5)] + HASHERS[64/LOOP-6'd1].state[`IDX(5)];
                        tx_hash[`IDX(6)] <= rx_state[`IDX(6)] + HASHERS[64/LOOP-6'd1].state[`IDX(6)];
                        tx_hash[`IDX(7)] <= rx_state[`IDX(7)] + HASHERS[64/LOOP-6'd1].state[`IDX(7)];
                end
        end


endmodule
```

Figure 28: sha256_transform module

Figure 28 shows the complete sha256_transform module that initializes the double SHA-256 hash.

Inputs

feedback is initialized to zero and controls the .rx_w and .rx_state output.

cnt is also initialized to zero.

27

rx_state is initialized to the initial hash values. These are the first 32 bits of the fractional parts of the square roots of the first 8 primes 2…19.

rx_input is the data (message/block header) to be hashed and is based on the current nonce.

Outputs
tx_hash is the hashed output.

Generate hashers
The for loop in the sha256_transform module generates the sha256_digester modules.

Quartus II settings

Figure 29 shows the Quartus II settings that were used to set the clock rate to 50Mhz.
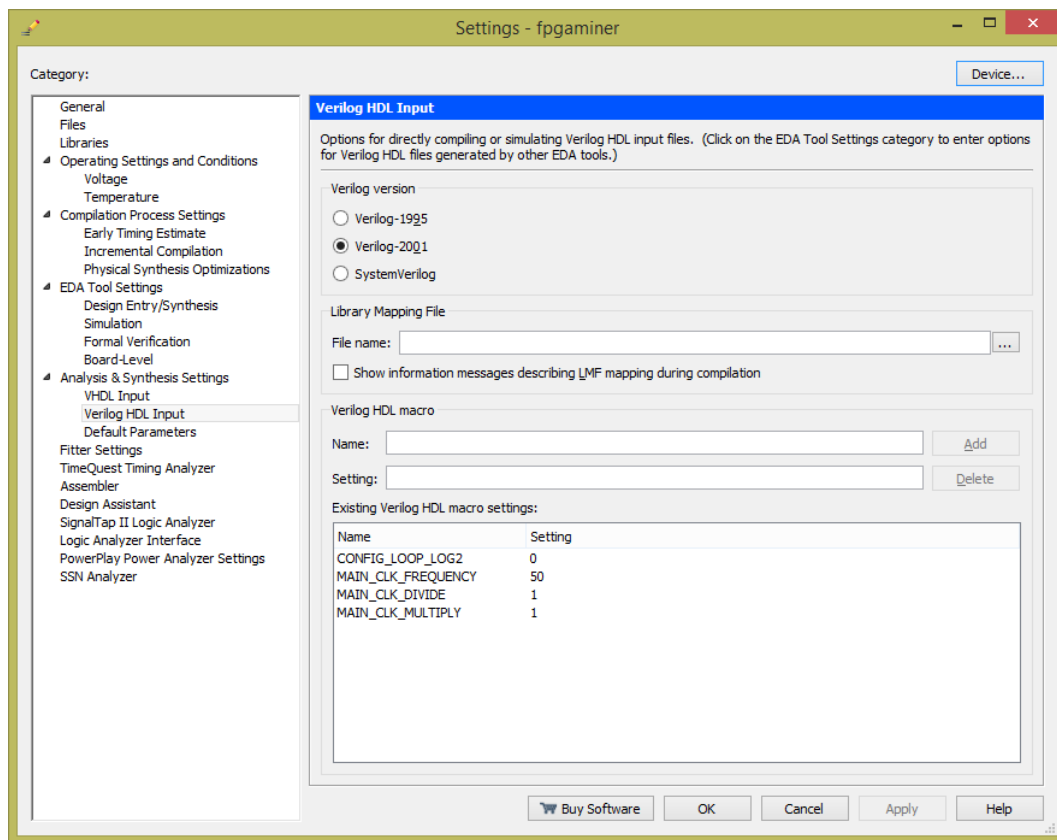
Figure 29: Verilog HDL Inputs

CONFIG_LOOP_LOG2 = 0

The CONFIG_LOOP_LOG2 parameter determines how unrolled the SHA-256 compression calculations are. A setting of 0 is completely unrolled, resulting in 128 rounds and a larger, but faster design. A setting of 1 will reduce to 64 rounds with half the size and speed. A setting of 2 will reduce to 32 rounds with a quarter of the size and speed. The valid range is 0 to 5.

MAIN_CLK_FREQUENCY = 50MHz

> The clock frequency is set to 50MHz to balance the speed and cooling of the Cyclone IV EP4CE115F29C7.

MAIN_CLK_DIVIDE = 1

MAIN_CLK_MULTIPLY = 1

Testing

The Open-Source FPGA Bitcoin Miner was tested on the Altera DE2-115 development board to verify that it could successfully mine bitcoins from the bitcoin network.

Miner instructions

The following instructions are included with the Open-Source FPGA Bitcoin Miner

1) Connect the DE2-115 Development Kit to your PC through USB, connect its power, and turn it on.
2) Ensure that the DE2-115's drivers have been installed successfully on your PC. Consult the DE2-115 User Guide for more information on setting up the DE2-115.
3) Navigate to 'scripts/program' and run 'program-fpga-board.bat'.
4) Follow the instructions provided by the program-fpga-board script. Select the correct cable and programming file. Once programming has succeeded, the DE2-115 is now ready to mine!
   1. *Note: This script sometimes fails immediately upon execution. Please try running it again.*
5) Run 'mine.bat'
6) If working correctly, 'mine.bat' will leave a console window open where it reports hashing rate, estimated hashing rate and accepted/rejected share information.
7) Profit!

The Open-Source FPGA Bitcoin Miner was tested with pooled mining since bitcoins are only created 25 at a time, and the race to validate a block and get the 25 BTC reward is very competitive. It can take a long time before a single user could expect to make a return on their mining if any.

 Pooled mining instead offers smaller, more frequent, and steadier payouts. Bitcoin pools give users blocks of lower difficult to solve and each solution found is counted as one 'share'. Occasionally, a solution may also meet the full target difficulty requirements of the bitcoin network and the pool will be rewarded the 25 BTC reward. The 25 BTCs are then divided among users based on their number of shares. Once a reward is paid out, a new round is started and users in the pool will work for new shares of the next block reward. In this way, users in a mining pool get more frequent payouts since they are not required to find the hash that satisfies the network target. On the other hand, a user is required to share the 25 BTC reward among the mining pool even if they are the user that found the successful hash.

This project initially tried to mine with btcguild.com, but there were problems connecting the Open-Source Bitcoin Miner to the pool. The mining.bitcoin.cz was tried next, and it uses a Stratum mining protocol instead of the getwork specification that the open source miner is built upon. To fix this problem, a Stratum mining proxy is available that bridges the older HTTP/getwork protocol and the Stratum mining protocol. The Stratum mining protocol is used since it is not bound by miner performance. Once the connection to the mining pool was established, the miner could successfully earn shares as shown in Figure 30.

Figure 30: Mining console

The performance of the open source miner, shown in Figure 30, is around 50MH/s which is relatively low compared to other ASIC miners. Butterfly Labs is currently taking pre-orders for a 600GH/s miner, but it has a price tag of $2,196 and uses 350W. In comparison, the DE2-115 is estimated to use around 4.7 watts and cost $300 for academic users. The Butterfly Labs miner still offers exponentially better performance at 273MH/$ versus the DE2-115 at 0.16MH/$. To try and speed up the performance of the open source miner the SHA-256 hashes could be run in parallel to double the throughput of the miner and increase the chance of finding a valid hash.

The current open source miner has a utilization of 75,049 / 114,480 logic elements or about 66% of the Cyclone IV's resource total. Figure 31 shows the logic element usage in the Quartos synthesis summary. Figure 32 also shows that the logic element usage is relatively evenly distributed over the entire FPGA. It seemed possible that there might be enough spare area to implement a pipelined design, but the new design ended up trying to use 168,104 logic elements or 147% of the available resources. The Quartus II design software failed to route the pipelined project because there were not enough logic elements.
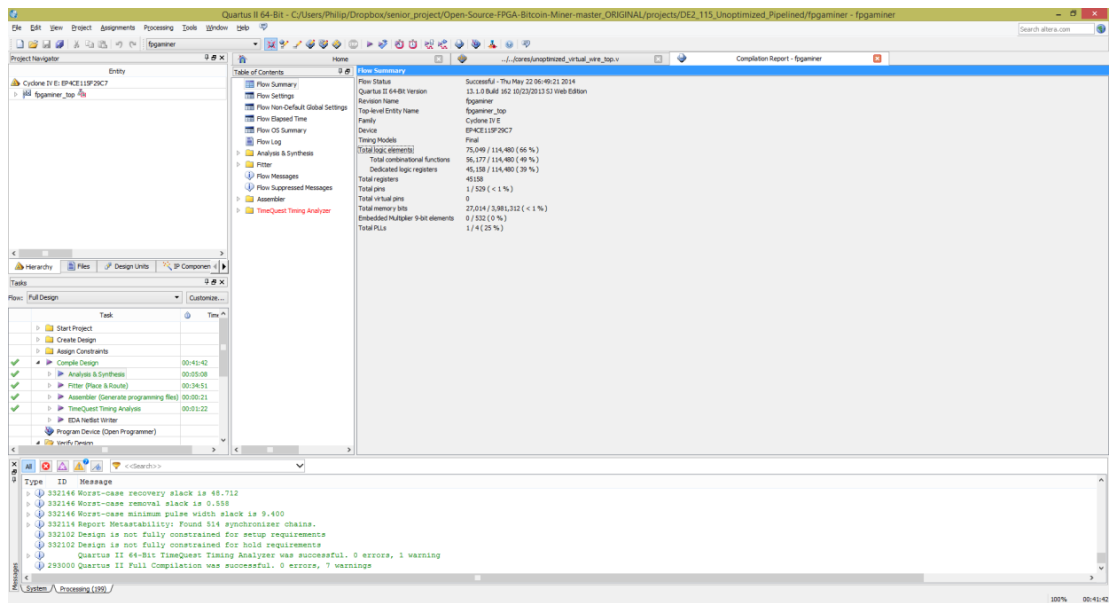
Figure 31: Quartus synthesis summary, unpipelined
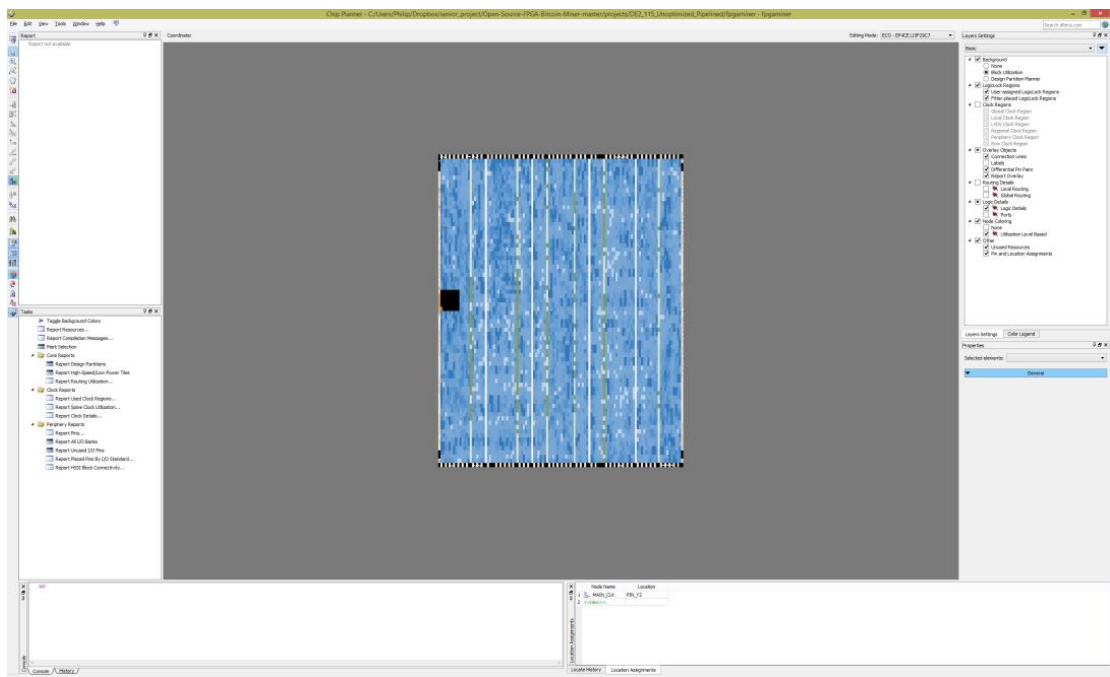
Figure 32: Quartus chip planner, unpipelined

Another consideration to take into account is the data rate between the bitcoin mining network pool and the miner. The miner needs to request block header information to hash which is 80bytes. The DE2-115 development board connects to a computer using USB 2.0 which is rated at 480Mb/s (60MB/s).

Data rate limitation

60MB/s / 80bytes = 750,000 per second.

The USB 2.0 transfer rate limits the number of block headers that can be sent to the miner to 750,000 per second so the miner could potentially have a max hash rate of 750MH/s. Further speed increases could make use of the gigabit Ethernet connection (1000Mb/s or 125MB/s).

Gigabit Ethernet limitation

125MB/s / 80bytes = 1,563,000 per second.

A gigabit connection could have a max has rate around 1.56GH/s.

Conclusion and Recommendations

The Open-Source FPGA Miner offers acceptable performance on the Altera DE2-115, but could be further improved by pipelining the design and using a higher tier FPGA with more logic elements. Another simple performance enhancement would be to increase the clock rate of the miner, but 50MHz was used because higher clock rates would require actively cooling the Cyclone IV to prevent damage. A final recommendation for this project would be to update the interface between the bitcoin network mining pools and the miner to support the newer Stratum protocol.

While FPGAs currently offer an increase in mining performance over other COTS options, the future of bitcoin mining is moving towards ASICs. One of the current dilemmas with bitcoin mining is balancing the cost of mining with the potential rewards or received bitcoins from validating blocks. It's a relatively costly investment to purchase mining hardware and pay for the electricity to run it with the volatility of the bitcoin market. One of the biggest reasons CPUs and GPUs are no longer used as miners is because the electricity to run them often cost more than the amount of bitcoins received from mining. The bitcoin network is built upon the idea that other users on the network will validate transactions for potential bitcoin rewards. ASICs offer the best performance per watt, but FPGAs may still have a place in the future of crypto currency as a platform to test develop new mining algorithms. The biggest advantage of FPGAs in crypto currency mining is that they are not limited to one currency such as bitcoin. Other competing crypto currencies such as litecoin or dogecoin could be mined with an FPGA because it could be reprogramed to run scrypt instead of SHA-256.

Bibliography

[1] Nakamoto, Satoshi. "Bitcoin: A Peer-to-Peer Electronic Cash System." November 2008.

[2] "Descriptions of SHA-256, SHA-384, and SHA-512."

[3] "Fpgaminer/Open-Source-FPGA-Bitcoin-Miner." *GitHub*. N.p., n.d. Web. 10 June 2014.

[4] "Official Open Source FPGA Bitcoin Miner (Last Update: April 14th, 2013)."*Official Open Source FPGA Bitcoin Miner (Last Update: April 14th, 2013)*. N.p., n.d. Web. 10 June 2014.

[5] McEvoy, Robert P., Crowe, Francis M., Murphy, Colin C., Marnane, William P. "Optimisation of the SHA-2 Family of Hash Functions on FPGAs."

[6] "Main Page." *Bitcoin*. N.p., n.d. Web. 10 June 2014.

[7] "Home." *BTC Guild*. N.p., n.d. Web. 11 June 2014.

[8] "Stratum Mining Protocol - Mining.bitcoin.cz." *Stratum Mining Protocol - Mining.bitcoin.cz*. N.p., n.d. Web. 11 June 2014.

Appendix A: Source Code

```verilog
/*
*
* Copyright (c) 2011-2012 fpgaminer@bitcoin-mining.com
*
*
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*
*/


// Top-level module that uses the unoptimized mining core and Virtual Wire
// external interface.


`timescale 1ns/1ps

module fpgaminer_top (
        input MAIN_CLK
);

        // The LOOP_LOG2 parameter determines how unrolled the SHA-256
        // calculations are. For example, a setting of 0 will completely
        // unroll the calculations, resulting in 128 rounds and a large, but
        // fast design.
        //
        // A setting of 1 will result in 64 rounds, with half the size and
        // half the speed. 2 will be 32 rounds, with 1/4th the size and speed.
        // And so on.
        //
        // Valid range: [0, 5]
`ifdef CONFIG_LOOP_LOG2
        localparam LOOP_LOG2 = `CONFIG_LOOP_LOG2;
`else
        localparam LOOP_LOG2 = 0;
`endif

        // No need to adjust these parameters
        localparam [5:0] LOOP = (6'd1 << LOOP_LOG2);
        // The nonce will always be larger at the time we discover a valid
        // hash. This is its offset from the nonce that gave rise to the valid
        // hash (except when LOOP_LOG2 == 0 or 1, where the offset is 131 or
        // 66 respectively).
        localparam [31:0] GOLDEN_NONCE_OFFSET = (32'd1 << (7 - LOOP_LOG2)) + 32'd1;


        ////
        reg [255:0] state = 0;
        reg [511:0] data = 0;
        reg [31:0] nonce = 32'h00000000;


        //// PLL
        wire hash_clk;
```

```verilog
        hashing_pll # (
                .INPUT_FREQUENCY (`MAIN_CLK_FREQUENCY),
                .DIVIDE_BY (`MAIN_CLK_DIVIDE),
                .MULTIPLY_BY (`MAIN_CLK_MULTIPLY)
        ) pll_blk (
                .rx_clk (MAIN_CLK),
                .tx_hash_clk (hash_clk)
        );


        //// Hashers
        wire [255:0] hash, hash2;
        reg [5:0] cnt = 6'd0;
        reg feedback = 1'b0;

        sha256_transform #(.LOOP(LOOP)) uut (
                .clk(hash_clk),
                .feedback(feedback),
                .cnt(cnt),
                .rx_state(state),
                .rx_input(data),
                .tx_hash(hash)
        );
        sha256_transform #(.LOOP(LOOP)) uut2 (
                .clk(hash_clk),
                .feedback(feedback),
                .cnt(cnt),
                .rx_state(256'h5be0cd191f83d9ab9b05688c510e527fa54ff53a3c6ef372bb67ae856a09e667),
                .rx_input({256'h00000100000000000000000000000000000000000000000000000080000000,
hash}),
                .tx_hash(hash2)
        );


        //// Virtual Wire Control
        reg [255:0] midstate_buf = 0, data_buf = 0;
        wire [255:0] midstate_vw, data2_vw;

        virtual_wire # (.OUTPUT_WIDTH(256), .INSTANCE_ID("STAT")) midstate_vw_blk (.clk
(hash_clk), .rx_data (), .tx_data (midstate_vw));
        virtual_wire # (.OUTPUT_WIDTH(256), .INSTANCE_ID("DAT2")) data2_vw_blk (.clk (hash_clk),
.rx_data (), .tx_data (data2_vw));

        //// Virtual Wire Output
        reg [31:0] golden_nonce = 0;

        virtual_wire # (.INPUT_WIDTH(32), .INSTANCE_ID("GNON")) golden_nonce_vw_blk (.clk
(hash_clk), .rx_data (golden_nonce), .tx_data ());
        virtual_wire # (.INPUT_WIDTH(32), .INSTANCE_ID("NONC")) nonce_vw_blk (.clk (hash_clk),
.rx_data (nonce), .tx_data ());


        //// Control Unit
        reg is_golden_ticket = 1'b0;
        reg feedback_d1 = 1'b1;
        wire [5:0] cnt_next;
        wire [31:0] nonce_next;
        wire feedback_next;

        assign cnt_next =  (LOOP == 1) ? 6'd0 : (cnt + 6'd1) & (LOOP-1);
        // On the first count (cnt==0), load data from previous stage (no feedback)
        // on 1..LOOP-1, take feedback from current stage
        // This reduces the throughput by a factor of (LOOP), but also reduces the design size by
the same amount
        assign feedback_next = (LOOP == 1) ? 1'b0 : (cnt_next != {(LOOP_LOG2){1'b0}});
        assign nonce_next = feedback_next ? nonce : (nonce + 32'd1);


        always @ (posedge hash_clk)
        begin
                midstate_buf <= midstate_vw;
```

```verilog
            data_buf <= data2_vw;

            cnt <= cnt_next;
            feedback <= feedback_next;
            feedback_d1 <= feedback;

            // Give new data to the hasher
            state <= midstate_buf;
            data <=
{384'h000002800000000000000000000000000000000000000000000000000000000000000000000000000000080000000, nonce_next, data_buf[95:0]};
            nonce <= nonce_next;


            // Check to see if the last hash generated is valid.
            is_golden_ticket <= (hash2[255:224] == 32'h00000000) && !feedback_d1;
            if(is_golden_ticket)
            begin
                    // TODO: Find a more compact calculation for this
                    if (LOOP == 1)
                            golden_nonce <= nonce - 32'd131;
                    else if (LOOP == 2)
                            golden_nonce <= nonce - 32'd66;
                    else
                            golden_nonce <= nonce - GOLDEN_NONCE_OFFSET;
            end
        end

endmodule
```

```verilog
/*
 *
 * Copyright (c) 2011-2012 fpgaminer@bitcoin-mining.com
 *
 *
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 */


module virtual_wire # (
        parameter INPUT_WIDTH = 0,
        parameter OUTPUT_WIDTH = 0,
        parameter INITIAL_VALUE = " 0",
        parameter INSTANCE_ID = "NONE"
) (
        input clk,
        input [INPUT_WIDTH-1:0] rx_data,
        output [OUTPUT_WIDTH-1:0] tx_data
);

        altsource_probe          altsource_probe_component (
                                 .probe (rx_data),
                                 .source_clk (clk),
                                 .source (tx_data)
```

```verilog
                                // synopsys translate_off
                                ,
                                .clrn (),
                                .ena (),
                                .ir_in (),
                                .ir_out (),
                                .jtag_state_cdr (),
                                .jtag_state_cir (),
                                .jtag_state_e1dr (),
                                .jtag_state_sdr (),
                                .jtag_state_tlr (),
                                .jtag_state_udr (),
                                .jtag_state_uir (),
                                .raw_tck (),
                                .source_ena (),
                                .tdi (),
                                .tdo (),
                                .usr1 ()
                                // synopsys translate_on
                                );
        defparam
                altsource_probe_component.enable_metastability = "YES",
                altsource_probe_component.instance_id = INSTANCE_ID,
                altsource_probe_component.probe_width = INPUT_WIDTH,
                altsource_probe_component.sld_auto_instance_index = "YES",
                altsource_probe_component.sld_instance_index = 0,
                altsource_probe_component.source_initial_value = INITIAL_VALUE,
                altsource_probe_component.source_width = OUTPUT_WIDTH;

endmodule
```

```verilog
// Generate a clock to be used by the hashing cores.

module hashing_pll # (
        parameter INPUT_FREQUENCY = 50,
        parameter DIVIDE_BY = 1,
        parameter MULTIPLY_BY = 1
) (
        input rx_clk,
        output tx_hash_clk
);

        wire [4:0] clks;

        assign tx_hash_clk = clks[0];

        altpll altpll_component (
                .inclk ({1'b0, rx_clk}),
                .clk (clks),
                .activeclock (),
                .areset (1'b0),
                .clkbad (),
                .clkena ({6{1'b1}}),
                .clkloss (),
                .clkswitch (1'b0),
                .configupdate (1'b0),
                .enable0 (),
                .enable1 (),
                .extclk (),
                .extclkena ({4{1'b1}}),
                .fbin (1'b1),
                .fbmimicbidir (),
                .fbout (),
                .fref (),
                .icdrclk (),
```

```verilog
        .locked (),
        .pfdena (1'b1),
        .phasecounterselect ({4{1'b1}}),
        .phasedone (),
        .phasestep (1'b1),
        .phaseupdown (1'b1),
        .pllena (1'b1),
        .scanaclr (1'b0),
        .scanclk (1'b0),
        .scanclkena (1'b1),
        .scandata (1'b0),
        .scandataout (),
        .scandone (),
        .scanread (1'b0),
        .scanwrite (1'b0),
        .sclkout0 (),
        .sclkout1 (),
        .vcooverrange (),
        .vcounderrange ());
defparam
        altpll_component.bandwidth_type = "AUTO",
        altpll_component.clk0_divide_by = DIVIDE_BY,
        altpll_component.clk0_duty_cycle = 50,
        altpll_component.clk0_multiply_by = MULTIPLY_BY,
        altpll_component.clk0_phase_shift = "0",
        altpll_component.compensate_clock = "CLK0",
        altpll_component.inclk0_input_frequency = (1000000 / INPUT_FREQUENCY),
        altpll_component.intended_device_family = "Cyclone IV E",
        altpll_component.lpm_hint = "CBX_MODULE_PREFIX=main_pll",
        altpll_component.lpm_type = "altpll",
        altpll_component.operation_mode = "NORMAL",
        altpll_component.pll_type = "AUTO",
        altpll_component.port_activeclock = "PORT_UNUSED",
        altpll_component.port_areset = "PORT_UNUSED",
        altpll_component.port_clkbad0 = "PORT_UNUSED",
        altpll_component.port_clkbad1 = "PORT_UNUSED",
        altpll_component.port_clkloss = "PORT_UNUSED",
        altpll_component.port_clkswitch = "PORT_UNUSED",
        altpll_component.port_configupdate = "PORT_UNUSED",
        altpll_component.port_fbin = "PORT_UNUSED",
        altpll_component.port_inclk0 = "PORT_USED",
        altpll_component.port_inclk1 = "PORT_UNUSED",
        altpll_component.port_locked = "PORT_UNUSED",
        altpll_component.port_pfdena = "PORT_UNUSED",
        altpll_component.port_phasecounterselect = "PORT_UNUSED",
        altpll_component.port_phasedone = "PORT_UNUSED",
        altpll_component.port_phasestep = "PORT_UNUSED",
        altpll_component.port_phaseupdown = "PORT_UNUSED",
        altpll_component.port_pllena = "PORT_UNUSED",
        altpll_component.port_scanaclr = "PORT_UNUSED",
        altpll_component.port_scanclk = "PORT_UNUSED",
        altpll_component.port_scanclkena = "PORT_UNUSED",
        altpll_component.port_scandata = "PORT_UNUSED",
        altpll_component.port_scandataout = "PORT_UNUSED",
        altpll_component.port_scandone = "PORT_UNUSED",
        altpll_component.port_scanread = "PORT_UNUSED",
        altpll_component.port_scanwrite = "PORT_UNUSED",
        altpll_component.port_clk0 = "PORT_USED",
        altpll_component.port_clk1 = "PORT_UNUSED",
        altpll_component.port_clk2 = "PORT_UNUSED",
        altpll_component.port_clk3 = "PORT_UNUSED",
        altpll_component.port_clk4 = "PORT_UNUSED",
        altpll_component.port_clk5 = "PORT_UNUSED",
        altpll_component.port_clkena0 = "PORT_UNUSED",
        altpll_component.port_clkena1 = "PORT_UNUSED",
        altpll_component.port_clkena2 = "PORT_UNUSED",
        altpll_component.port_clkena3 = "PORT_UNUSED",
        altpll_component.port_clkena4 = "PORT_UNUSED",
        altpll_component.port_clkena5 = "PORT_UNUSED",
        altpll_component.port_extclk0 = "PORT_UNUSED",
        altpll_component.port_extclk1 = "PORT_UNUSED",
```

```verilog
                altpll_component.port_extclk2 = "PORT_UNUSED",
                altpll_component.port_extclk3 = "PORT_UNUSED",
                altpll_component.width_clock = 5;


endmodule
```

```verilog
/*
*
* Copyright (c) 2011 fpgaminer@bitcoin-mining.com
*
*
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*
*/


`timescale 1ns/1ps

// A quick define to help index 32-bit words inside a larger register.
`define IDX(x) (((x)+1)*(32)-1):((x)*(32))


// Perform a SHA-256 transformation on the given 512-bit data, and 256-bit
// initial state,
// Outputs one 256-bit hash every LOOP cycle(s).
//
// The LOOP parameter determines both the size and speed of this module.
// A value of 1 implies a fully unrolled SHA-256 calculation spanning 64 round
// modules and calculating a full SHA-256 hash every clock cycle. A value of
// 2 implies a half-unrolled loop, with 32 round modules and calculating
// a full hash in 2 clock cycles. And so forth.
module sha256_transform #(
        parameter LOOP = 6'd4
) (
        input clk,
        input feedback,
        input [5:0] cnt,
        input [255:0] rx_state,
        input [511:0] rx_input,
        output reg [255:0] tx_hash
);

        // Constants defined by the SHA-2 standard.
        localparam Ks = {
                32'h428a2f98, 32'h71374491, 32'hb5c0fbcf, 32'he9b5dba5,
                32'h3956c25b, 32'h59f111f1, 32'h923f82a4, 32'hab1c5ed5,
                32'hd807aa98, 32'h12835b01, 32'h243185be, 32'h550c7dc3,
                32'h72be5d74, 32'h80deb1fe, 32'h9bdc06a7, 32'hc19bf174,
                32'he49b69c1, 32'hefbe4786, 32'h0fc19dc6, 32'h240ca1cc,
                32'h2de92c6f, 32'h4a7484aa, 32'h5cb0a9dc, 32'h76f988da,
                32'h983e5152, 32'ha831c66d, 32'hb00327c8, 32'hbf597fc7,
                32'hc6e00bf3, 32'hd5a79147, 32'h06ca6351, 32'h14292967,
                32'h27b70a85, 32'h2e1b2138, 32'h4d2c6dfc, 32'h53380d13,
```

```verilog
                    32'h650a7354, 32'h766a0abb, 32'h81c2c92e, 32'h92722c85,
                    32'ha2bfe8a1, 32'ha81a664b, 32'hc24b8b70, 32'hc76c51a3,
                    32'hd192e819, 32'hd6990624, 32'hf40e3585, 32'h106aa070,
                    32'h19a4c116, 32'h1e376c08, 32'h2748774c, 32'h34b0bcb5,
                    32'h391c0cb3, 32'h4ed8aa4a, 32'h5b9cca4f, 32'h682e6ff3,
                    32'h748f82ee, 32'h78a5636f, 32'h84c87814, 32'h8cc70208,
                    32'h90befffa, 32'ha4506ceb, 32'hbef9a3f7, 32'hc67178f2};


        genvar i;

        generate

                for (i = 0; i < 64/LOOP; i = i + 1) begin : HASHERS
                        wire [511:0] W;
                        wire [255:0] state;

                        if(i == 0)
                                sha256_digester U (
                                        .clk(clk),
                                        .k(Ks[32*(63-cnt) +: 32]),
                                        .rx_w(feedback ? W : rx_input),
                                        .rx_state(feedback ? state : rx_state),
                                        .tx_w(W),
                                        .tx_state(state)
                                );
                        else
                                sha256_digester U (
                                        .clk(clk),
                                        .k(Ks[32*(63-LOOP*i-cnt) +: 32]),
                                        .rx_w(feedback ? W : HASHERS[i-1].W),
                                        .rx_state(feedback ? state : HASHERS[i-1].state),
                                        .tx_w(W),
                                        .tx_state(state)
                                );
                end

        endgenerate

        always @ (posedge clk)
        begin
                if (!feedback)
                begin
                        tx_hash[`IDX(0)] <= rx_state[`IDX(0)] + HASHERS[64/LOOP-
6'd1].state[`IDX(0)];
                        tx_hash[`IDX(1)] <= rx_state[`IDX(1)] + HASHERS[64/LOOP-
6'd1].state[`IDX(1)];
                        tx_hash[`IDX(2)] <= rx_state[`IDX(2)] + HASHERS[64/LOOP-
6'd1].state[`IDX(2)];
                        tx_hash[`IDX(3)] <= rx_state[`IDX(3)] + HASHERS[64/LOOP-
6'd1].state[`IDX(3)];
                        tx_hash[`IDX(4)] <= rx_state[`IDX(4)] + HASHERS[64/LOOP-
6'd1].state[`IDX(4)];
                        tx_hash[`IDX(5)] <= rx_state[`IDX(5)] + HASHERS[64/LOOP-
6'd1].state[`IDX(5)];
                        tx_hash[`IDX(6)] <= rx_state[`IDX(6)] + HASHERS[64/LOOP-
6'd1].state[`IDX(6)];
                        tx_hash[`IDX(7)] <= rx_state[`IDX(7)] + HASHERS[64/LOOP-
6'd1].state[`IDX(7)];
                end
        end


endmodule


module sha256_digester (clk, k, rx_w, rx_state, tx_w, tx_state);

        input clk;
        input [31:0] k;
        input [511:0] rx_w;
```

```verilog
        input [255:0] rx_state;

        output reg [511:0] tx_w;
        output reg [255:0] tx_state;


        wire [31:0] e0_w, e1_w, ch_w, maj_w, s0_w, s1_w;


        e0      e0_blk  (rx_state[`IDX(0)], e0_w);
        e1      e1_blk  (rx_state[`IDX(4)], e1_w);
        ch      ch_blk  (rx_state[`IDX(4)], rx_state[`IDX(5)], rx_state[`IDX(6)], ch_w);
        maj     maj_blk (rx_state[`IDX(0)], rx_state[`IDX(1)], rx_state[`IDX(2)], maj_w);
        s0      s0_blk  (rx_w[63:32], s0_w);
        s1      s1_blk  (rx_w[479:448], s1_w);

        wire [31:0] t1 = rx_state[`IDX(7)] + e1_w + ch_w + rx_w[31:0] + k;
        wire [31:0] t2 = e0_w + maj_w;
        wire [31:0] new_w = s1_w + rx_w[319:288] + s0_w + rx_w[31:0];


        always @ (posedge clk)
        begin
                tx_w[511:480] <= new_w;
                tx_w[479:0] <= rx_w[511:32];

                tx_state[`IDX(7)] <= rx_state[`IDX(6)];
                tx_state[`IDX(6)] <= rx_state[`IDX(5)];
                tx_state[`IDX(5)] <= rx_state[`IDX(4)];
                tx_state[`IDX(4)] <= rx_state[`IDX(3)] + t1;
                tx_state[`IDX(3)] <= rx_state[`IDX(2)];
                tx_state[`IDX(2)] <= rx_state[`IDX(1)];
                tx_state[`IDX(1)] <= rx_state[`IDX(0)];
                tx_state[`IDX(0)] <= t1 + t2;
        end

endmodule
```

```verilog
/*
*
* Copyright (c) 2011-2012 fpgaminer@bitcoin-mining.com
*
*
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*
*/

// Double pipelined module

`timescale 1ns/1ps

module fpgaminer_top (
        input MAIN_CLK
);
```

```verilog
        // The LOOP_LOG2 parameter determines how unrolled the SHA-256
        // calculations are. For example, a setting of 0 will completely
        // unroll the calculations, resulting in 128 rounds and a large, but
        // fast design.
        //
        // A setting of 1 will result in 64 rounds, with half the size and
        // half the speed. 2 will be 32 rounds, with 1/4th the size and speed.
        // And so on.
        //
        // Valid range: [0, 5]
`ifdef CONFIG_LOOP_LOG2
        localparam LOOP_LOG2 = `CONFIG_LOOP_LOG2;
`else
        localparam LOOP_LOG2 = 0;
`endif

        // No need to adjust these parameters
        localparam [5:0] LOOP = (6'd1 << LOOP_LOG2);
        // The nonce will always be larger at the time we discover a valid
        // hash. This is its offset from the nonce that gave rise to the valid
        // hash (except when LOOP_LOG2 == 0 or 1, where the offset is 131 or
        // 66 respectively).
        localparam [31:0] GOLDEN_NONCE_OFFSET = (32'd1 << (7 - LOOP_LOG2)) + 32'd1;


        ////
        reg [255:0] state = 0;
        reg [511:0] data = 0;
        reg [255:0] state2 = 0;
        reg [511:0] data2 = 0;
        reg [31:0] nonce = 32'h00000000;
        reg [31:0] nonce2 = 32'h00000000;


        //// PLL
        wire hash_clk;

        hashing_pll # (
                .INPUT_FREQUENCY (`MAIN_CLK_FREQUENCY),
                .DIVIDE_BY (`MAIN_CLK_DIVIDE),
                .MULTIPLY_BY (`MAIN_CLK_MULTIPLY)
        ) pll_blk (
                .rx_clk (MAIN_CLK),
                .tx_hash_clk (hash_clk)
        );


        //// Hashers
        wire [255:0] hash, hash2, hash3, hash4;
        reg [5:0] cnt = 6'd0;
        reg [5:0] cnt2 = 6'd500000;
        reg feedback = 1'b0;
        reg feedback2 = 1'b0;

        sha256_transform #(.LOOP(LOOP)) uut (
                .clk(hash_clk),
                .feedback(feedback),
                .cnt(cnt),
                .rx_state(state),
                .rx_input(data),
                .tx_hash(hash)
        );
        sha256_transform #(.LOOP(LOOP)) uut2 (
                .clk(hash_clk),
                .feedback(feedback),
                .cnt(cnt),
                .rx_state(256'h5be0cd191f83d9ab9b05688c510e527fa54ff53a3c6ef372bb67ae856a09e667),
                .rx_input({256'h00000010000000000000000000000000000000000000000000000000080000000,
hash}),
                .tx_hash(hash2)
        );
```

```verilog
// Added
        sha256_transform #(.LOOP(LOOP)) uut3 (
                .clk(hash_clk),
                .feedback(feedback2),
                .cnt(cnt2),
                .rx_state(state2),
                .rx_input(data2),
                .tx_hash(hash3)
        );
        sha256_transform #(.LOOP(LOOP)) uut4 (
                .clk(hash_clk),
                .feedback(feedback2),
                .cnt(cnt2),
                .rx_state(256'h5be0cd191f83d9ab9b05688c510e527fa54ff53a3c6ef372bb67ae856a09e667),
                .rx_input({256'h00000100000000000000000000000000000000000000000000000080000000,
hash3}),
                .tx_hash(hash4)
        );
//

        //// Virtual Wire Control
        reg [255:0] midstate_buf = 0, data_buf = 0;
        wire [255:0] midstate_vw, data2_vw;

        reg [255:0] midstate_buf2 = 0, data_buf2 = 0;
        wire [255:0] midstate_vw2, data2_vw2;

        virtual_wire # (.OUTPUT_WIDTH(256), .INSTANCE_ID("STAT")) midstate_vw_blk (.clk
(hash_clk), .rx_data (), .tx_data (midstate_vw));
        virtual_wire # (.OUTPUT_WIDTH(256), .INSTANCE_ID("DAT2")) data2_vw_blk (.clk (hash_clk),
.rx_data (), .tx_data (data2_vw));

        virtual_wire # (.OUTPUT_WIDTH(256), .INSTANCE_ID("STAT")) midstate_vw_blk2 (.clk
(hash_clk), .rx_data (), .tx_data (midstate_vw2));
        virtual_wire # (.OUTPUT_WIDTH(256), .INSTANCE_ID("DAT2")) data2_vw_blk2 (.clk (hash_clk),
.rx_data (), .tx_data (data2_vw2));

        //// Virtual Wire Output
        reg [31:0] golden_nonce = 0;

        virtual_wire # (.INPUT_WIDTH(32), .INSTANCE_ID("GNON")) golden_nonce_vw_blk (.clk
(hash_clk), .rx_data (golden_nonce), .tx_data ());
        virtual_wire # (.INPUT_WIDTH(32), .INSTANCE_ID("NONC")) nonce_vw_blk (.clk (hash_clk),
.rx_data (nonce), .tx_data ());


        //// Control Unit
        reg is_golden_ticket = 1'b0;
        reg feedback_d1 = 1'b1;
        wire [5:0] cnt_next;
        wire [31:0] nonce_next;
        wire feedback_next;

        reg is_golden_ticket2 = 1'b0;
        reg feedback_d1_2 = 1'b1;
        wire [5:0] cnt_next2;
        wire [31:0] nonce_next2;
        wire feedback_next2;


        assign cnt_next =  (LOOP == 1) ? 6'd0 : (cnt + 6'd1) & (LOOP-1);
        // On the first count (cnt==0), load data from previous stage (no feedback)
        // on 1..LOOP-1, take feedback from current stage
        // This reduces the throughput by a factor of (LOOP), but also reduces the design size by
the same amount
        assign feedback_next = (LOOP == 1) ? 1'b0 : (cnt_next != {(LOOP_LOG2){1'b0}});
        assign nonce_next = feedback_next ? nonce : (nonce + 32'd1);

        assign cnt_next2 =  (LOOP == 1) ? 6'd0 : (cnt2 + 6'd1) & (LOOP-1);
        // On the first count (cnt==0), load data from previous stage (no feedback)
```

```verilog
        // on 1..LOOP-1, take feedback from current stage
        // This reduces the throughput by a factor of (LOOP), but also reduces the design size by
the same amount
        assign feedback_next2 = (LOOP == 1) ? 1'b0 : (cnt_next2 != {(LOOP_LOG2){1'b0}});
        assign nonce_next2 = feedback_next2 ? nonce : (nonce + 32'd1);


        always @ (posedge hash_clk)
        begin
                midstate_buf <= midstate_vw;
                data_buf <= data2_vw;

                midstate_buf2 <= midstate_vw2;
                data_buf2 <= data2_vw2;

                cnt <= cnt_next;
                feedback <= feedback_next;
                feedback_d1 <= feedback;

                cnt2 <= cnt_next2;
                feedback2 <= feedback_next2;
                feedback_d1_2 <= feedback2;

                // Give new data to the hasher
                state <= midstate_buf;
                data <=
{384'h0000028000000000000000000000000000000000000000000000000000000000000000000000000000000800
00000, nonce_next, data_buf[95:0]};
                nonce <= nonce_next;

                state2 <= midstate_buf2;
                data2 <=
{384'h0000028000000000000000000000000000000000000000000000000000000000000000000000000000000800
00000, nonce_next2, data_buf[95:0]};
                nonce2 <= nonce_next2;

                // Check to see if the last hash generated is valid.
                is_golden_ticket <= (hash2[255:224] == 32'h00000000) && !feedback_d1;
                is_golden_ticket2 <= (hash4[255:224] == 32'h00000000) && !feedback_d1_2;
                if(is_golden_ticket)
                begin
                        // TODO: Find a more compact calculation for this
                        if (LOOP == 1)
                                golden_nonce <= nonce - 32'd131;
                        else if (LOOP == 2)
                                golden_nonce <= nonce - 32'd66;
                        else
                                golden_nonce <= nonce - GOLDEN_NONCE_OFFSET;

                end
                else if(is_golden_ticket2)
                begin
                        if (LOOP == 1)
                                golden_nonce <= nonce2 - 32'd131;
                        else if (LOOP == 2)
                                golden_nonce <= nonce2 - 32'd66;
                        else
                                golden_nonce <= nonce2 - GOLDEN_NONCE_OFFSET;
                end
        end

endmodule
```

Appendix B: Analysis of Senior Project Design

Project Title:
FPGA Based Bitcoin Mining

Student Name:
Philip Dotemoto

Student Signature:

Advisor Name:
John Oliver

Advisor Initials:

Summary of Functional Requirements:

Marketing Requirements	Engineering Specifications	Justification
1	The miner shall be able to retrieve header information from the bitcoin network and submit valid proof of work.	The miner will need to be able to retrieve block header information to hash and send valid hashes back to the bitcoin network to receive any rewards.
1	The bitcoin mining algorithm (double SHA-256 hash) shall be implemented on a commercial off the shelf (COTS) FPGA.	An FPGA with enough resources to implement the complete mining algorithm needs to be chosen.
2	The miner shall use standard hardware interfaces and connectors.	The miner will need to be easily setup by most users and those without extensive knowledge of the system.
2	The miner shall not require the user to interface with the system at the hardware description level for basic setup.	Users without prior knowledge of FPGAs should be able to use the miner.
Marketing Requirements		
1. The system shall implement a bitcoin miner on an FPGA.		
2. The system shall be easy to interface and setup.		

Primary Constraints:
The project constraints in this project are time and limitations with the available FPGA hardware. This project was limited in scope to studying the bitcoin mining protocol and verifying the operation of the Open-Source Bitcoin miner with the goal of identifying potential areas for improvement. With additional time, this project could be extended to try further modifications to the open source miner. The other main constraint in this project was the chosen Altera DE2-115 development board. While the Cyclone IV EP4CE115F29C7 has enough resources to implement the open source miner, it did not have enough

spare resources to place and route a double pipelined configuration. An FPGA with more logical elements would have been useful to test the pipelined design.

Economics:

While bitcoin mining has the potential to be profitable, the Altera DE2-115 development board does not offer enough performance for its price. The DE2-115 has an academic price of $300, and this cost would never realistically be recouped when considering the time and power the DE2-115 requires while mining. The DE2-115 does have numerous peripheral connections and devices that are not required for bitcoin mining. A simplified FPGA miner that includes only an FPGA, clock source, programmer, and power supply could be profitable. The price breakdown estimate is Cyclone IV EP4CE115F29C7: $469.37 for a quantity of 36 = $13.03, support circuitry estimated: $10, printed circuit board: $30, assembly: $20. Parts and assembly alone could cost around $75. Selling an FPGA miner for around $100 might be profitable for the manufacture, but not the end user looking to make a profit mining bitcoins.

Manufacturability:

This project utilized the DE2-115 development board, but a comparable dedicated FPGA miner could be manufactured in large scale.

Sustainability:

The FPGA mining software is unsustainable when considering the energy and computation power required to mine bitcoins is wasted on validating transactions. The SHA-256 hash is desirable for a crypto currency, such as bitcoin, because it serves as a proof of work for valid blocks, but the computation power of the bitcoin network could potentially be repurposed to solve more useful computations.

Ethical:

One of the ethical concerns with bitcoins is their ability to be transferred anonymously between users. This has made bitcoin a popular payment for illegal substances and goods. Recently, the FBI shut down the Silk Road anonymous market because it was primarily a site to buy and sell drugs. Bitcoins were the only currency accepted on Silk Road because it is difficult to trace individuals with transactions. While bitcoin has its foundations in being a unregulated currency, its usage in illegal transactions highlights some of the disadvantages. Another case is the Mt. Gox bitcoin exchange which was a website for users to trade bitcoins for other currencies. When Mt. Gox shutdown and filed for bankruptcy many users were unable to withdraw bitcoins they had stored with Mt. Gox. Since bitcoins are unregulated, there is no government deposit insurance and many users lost their bitcoins.

Development:

The development of this project was based on the Open-Source Bitcoin Miner. Future improvements to the open source miner include pipelining the design to increase the hash rate, updating the getwork protocol to Stratum, and implementing an option for a gigabit Ethernet instead up USB 2.0 to increase throughput to the miner.