Stand-Alone ASIC Bitcoin Miner

Final Report

Submission Date: December 13, 2017
Thursday 2:30 (Lab Section 5)
TA: Mochen

Prepared by:
Yash Bharatula
Michael Toner
Chinar Dhamija
Rahul Patni

1.Executive Summary

The underlying foundation of bitcoin mining utilizes the concept of hashing functions, a method of compressing input data to a block of fixed size. The bitcoin transaction record must be hashed into a block that must be verified. The notion of mining bitcoin simply refers to the validation of these blocks, using a double SHA256 encoding algorithm. This algorithm can be implemented in an ASIC with high efficiency, as many concurrent processes can be accomplished at the same time.

Bitcoin and other cryptocurrencies have the potential to dominate the financial environment in the future and competition to validate bitcoin blocks is always increasing. ASIC designs of bitcoin mining represent the most efficient approach leading innovation currently. ASIC implementations of these algorithms are faster than other approaches, such as GPU and CPU mining. The concepts necessary to realize the bitcoin mining design using the SHA256 encoding algorithm, involves: an understanding of bitcoin and its block representations; the bitcoin implementation of its hashing algorithm; and knowledge regarding dynamic data transfer.

A successful implementation would require the following resources:

- ➢ Verilog HDL Simulation and Design Synthesis Tool Chain
- ➢ USB bus documentation
- ➢ Reference Standard Cell Simulation Library for Mapped Design Verification
- ➢ Reference Standard Cell Technology Library for Final Design Layout Verification
- ➢ Bitcoin SHA256 Algorithm Documentation
- ➢ Bitcoin block description

The remainder of the proposal will be subdivided into sections regarding topic, expressing a higher level of detail of different aspects of the design. Towards the end of the document we include estimates of the area and timing parameters of the design and a general schedule we will follow to complete the design.

2. Design Specifications

2.1. System Usage

2.1.1. System Usage Diagram

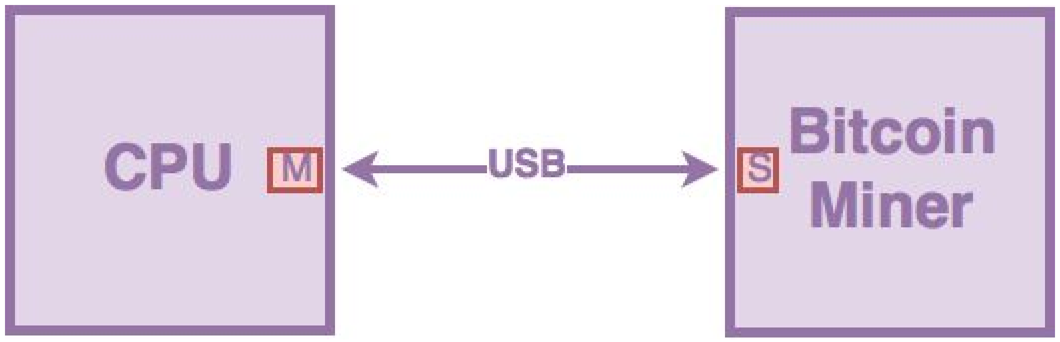


Figure 1: System Usage Diagram for Bitcoin Miner

The high-level block diagram in Figure 1 illustrates the intended use of the bitcoin miner in a system. The CPU will talk to the miner via a USB interface. The CPU will wake the bitcoin miner up to begin computation of a new block header on a blockchain. The CPU will send the entire block header in just 2 USB packets. It will then wait for the miner to calculate a valid hash of the header. The CPU will need to resend the block occasionally to make sure the time data on the header is up to date. Once a correct hash for the block is found, the miner will then send this hash back to the CPU to transmit on the blockchain. The CPU can also interrupt the miner at any point by sending an interrupt packet that will tell the bitcoin miner to stop computation of the current block. This will usually happen if another miner has already ‘solved’ the current block.

2.1.2. Implemented Standards and Algorithms Overview

- Bitcoin Standard
    - Double SHA-256 Hashing Algorithm
    - Cryptographic Nonce
    - Merkle Trees
    - Proof-of-work difficulty target
- USB 1.1 Interface
    - Bulk Transfer
    - NRZi Encoding
    - 12 MHz Speed

2.1.3. Design Pinout

Table 1: Miscellaneous Pinout Table

Signal Name	**Type (In/Out/Bidir)**	**Number of Bits**	**Description**
n_rst	In	1	Asynchronous Reset. (Active Low)
clk	In	1	System clock
vcc	Power	1	Power Pin
gnd	Ground	1	Ground Pin

Table 2: USB Mapped Slave Interface Pins

Signal Name	**Type (In/Out/Bidir)**	**Number of Bits**	**Description**
d_plus	Bidirectional	1	Computer Data line
d_minus	Bidirectional	1	Computer Data line
vcc	Power	1	Power Pin
gnd	Ground	1	Ground Pin

2.2. Operational Characteristics

2.2.1 *Normal Usage Design:*

1. CPU sends block packet to miner
2. The miner calculating valid hash of the block
3. CPU sends new block every few seconds to make sure the time is updated
4. Wait for bitcoin miner to compute a valid hash
5. A valid hash is found is ready to transmit
6. The CPU asks the miner to transmit the valid header back
7. At any point during this the CPU can send an interrupt packet to stop all computation of the current block header

2.2.2. SHA 256 Algorithm Description

[Source: National Institute of Standards and Technology]

The following section is a detailed description of how we are using the SHA-256 algorithm to hash. In this documentation a word is 32 bits of data.

Input: Block header to encrypt (80 bytes)

Output: SHA-256 bit hash

Step 1: Split Input Block

The Block/ Block chain will be split into 512-bit chunks. begin with the original header of length L bits and append a single '1' bit
append K '0' bits, where K is the minimum number >= 0 such that L + 1 + K + 64 is a multiple of 512. Append L as a 64-bit big-endian integer, making the total post-processed length a multiple of 512 bits.

Step 2: Copy chunk into the message schedule array

Copy chunk into first 16 words w[0..15] of the message schedule array. The message schedule array is 64 words long (2048 bits long).

Step 3: Initialize hash values

The hash array is an array of 8 32 bit wide words (256 bits wide in total). If we are hashing the first chunk, initialize the hash values (h[0:7]) to first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19:

```
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19
```

Step 4: Initialize array of round constants

Initialize array k[0:64] (64x32 bits wide) to the first 32 bits of the fractional parts of the cube roots of the first 64 primes:

```
k[0..63] :=
  0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4,
0xab1c5ed5,
  0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,
0xc19bf174,
  0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc,
0x76f988da,
  0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351,
0x14292967,
  0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e,
0x92722c85,
  0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585,
0x106aa070,
  0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f,
0x682e6ff3,
  0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7,
0xc67178f2
```

Step 5: Extend first 16 words of the message schedule array

Extend the first 16 words into the remaining 48 words w[16..63] of the message schedule array:

```
   for i from 16 to 63
        s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
        s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightshift 10)
        w[i] := w[i-16] + s0 + w[i-7] + s1
```

Step 5: Initialize computation variables (a through h) to current hash values

```
       a := h0
       b := h1
       c := h2
       d := h3
       e := h4
       f := h5
       g := h6
       h := h7
```

Step 6: Perform compression algorithm

```
for i from 0 to 63
        S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
        ch := (e and f) xor ((not e) and g)
        temp1 := h + S1 + ch + k[i] + w[i]
        S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
        maj := (a and b) xor (a and c) xor (b and c)
        temp2 := S0 + maj

        h := g
        g := f
```

```
f := e
e := d + temp1
d := c
c := b
b := a
a := temp1 + temp2
```

Step 7: Add compressed chunks into current hash values

```
h[0] := h[0] + a
h[1] := h[1] + b
h[2] := h[2] + c
h[3] := h[3] + d
h[4] := h[4] + e
h[5] := h[5] + f
h[6] := h[6] + g
h[7] := h[7] + h
```

Step 8: Verify if the hash has a proper proof of work

The final hashed value, contains all separate hashes in one variable. This variable will keep updating with every hashed chunk. Once all chunks are done hashing, we check whether there are the same number of leading zeros as the “proof of work”. The “proof of work” can be found in the header of the block.

For more description on the bitcoin proof of work see the developer reference on bitcoin.org

2.2.3 Block Header Description

Table 3: Block Header Description Header

Field	**Purpose**	**Data Type**	**Field Size**
Version	**Block version number**	**Int32**	**4**
hashPrevBlock	**Hash of the previous block header**	**char array**	**32**
hashMerkleRoot	**Hash based on all of the transactions in the block**	**char array**	**32**
Time	**Timestamp in UTC for block creation**	**uint32**	**4**
Difficulty	**Current target in compact format, number of zeros to match**	**uint32**	**4**
Nonce	**32-bit number, various values with which the hash is tried, starts at 0**	**uint32**	**4**

This is the description of the block header. Block headers are always 80 bytes. This is what is the actual data that gets hashed with the hashing function. This header is hashed and the leading zeros in the hash of this header is counted and compared with the “Difficulty” that is part of the header. If they match, it is a valid block. If they don’t match, the nonce is incremented and the header is re-hashed. This process is repeated until there is a match. This serves as the proof-of-work that establishes that this is a valid block.

2.3.4. Supported USB Transfer Modes

2.3.4.1. Bulk Transfers

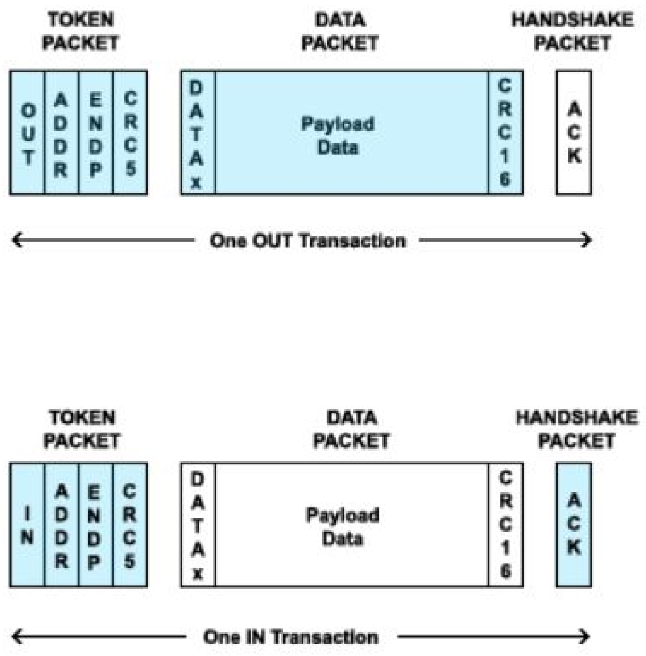


Figure 2: Bulk Transfer Packet Order

Data can be transmitted or received serially through the data line. The bulk transfer protocol allows for the ability to transfer large amounts of data with error free delivery. The USB bus serves as a communication protocol that bridges the CPU with the device. The Bulk transfer mode capabilities consist of in and out transactions. Each transaction can be represented by the successful transference of three packets. An output transaction begins with a token packet that specifies that it is an out packet. It is then followed by the payload data packet. As the full speed USB bus interface runs at full speed, or 12 MHz, the maximum size of the payload packet is 64 bytes. As bitcoin block header is 80 bytes, it must be sent into two separate parts. The final packet of the transaction is the ACK signal sent from the device. An in transaction consists of a similar token packet, but receives a data payload rather than sending it. The USB protocol then concludes the IN transaction with a handshake packet (ACK). Every instance that a packet is sent, a sync byte must be sent to prevent the clock from drifting away from the bus clock. This byte synchronizes the clock. The host either requests an In or Out token depending on the type of information that the host requires. The protocol replies with the corresponding data packet if successful or an error signal if an instance fails. The bulk transfer also conducts error detection through the usage of cyclic redundancy checks (CRCs), resulting in a linear shift register where certain bits are Xored together. The timing waveform analysis can be found later in section **3.2.3.3.**

2.4. Requirements for Design

The first constraint for our design comes from power consumption. In order for this module to be profitable we need to estimate how many hashes per second our design should do to offset the power consumption of the design. Using a current price of bitcoin at 7286 $/BTC, the cost of electricity in West Lafayette at 8.48 ¢/kWh, and a high estimate that our module uses 2 watts of energy, we would need to calculate 3.344×10^9 Hashes per second. This would force our clock rate to be .03 nanoseconds. This rate is completely unfeasible for any design and is impossible to compute even a single addition. We feel it is more realistic to set our clock period to be 10 nanoseconds or 100 MHz. This is the minimum speed needed to run the USB interface at full speed. We can justify this change because it is not feasible for our miner to even be profitable because the price of bitcoin is so volatile right now. This change would allow us to compute a total of 750,000 hashes per second which is very high considering the fabrication technology of .5 microns we are using. The other way to combat this would be to increase the number of hashing modules by a significant amount. This would work but would need a completely unrealistic amount of space making the cost of this design very expensive to fabricate and unprofitable.

As mentioned, space is in an important factor that must be optimized when nearing the completion of the design. The placement of registers and components must be taken into account, along with the wiring associated with the components. For simplicity the entire block header to be hashed is stored inside an array and is then redirected to each hashing module. A block is a standard 640 registers, and as such this takes up a significant portion of the overall design. This is simpler however, than to store the block in every module and only requires that there is a lot of wiring in our project. Because of the nature of SHA-256 the size of the wiring will also be a significant portion of our design. Most of the algorithm is wire rearranging between the 2048 wide input to the output for the algorithm. Wires are hard to estimate the size of until we make a full layout of the design and should be calculated more precisely later on.

Pipelining was considered as a portion of the hash calculation as a way to increase the throughput of the design but it is not a real feasible way to increase speed. It would take an additional 2048 registers to pipeline the design at any point which would almost double the area of a hashing module which would completely ruin the purpose of pipelining and add unneeded complexity to the design. It would be much more efficient to just add more hashing modules to increase throughput. We may revisit pipelining if we come realize we do not need all 2048 registers to pipeline the design later.

3. Design Implementation

3.1 Design Architecture

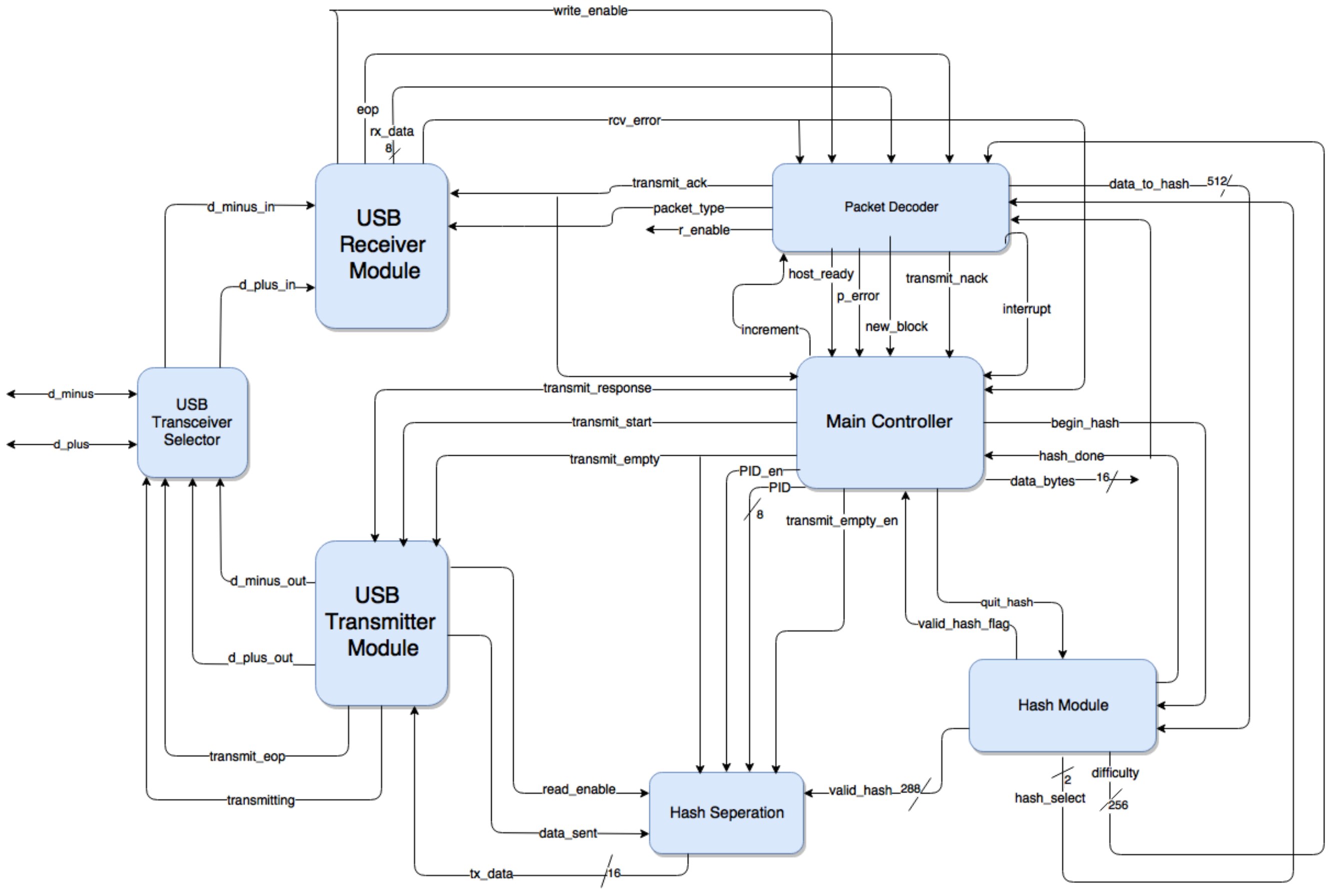


Figure 3: High-Level Design Architecture for Bitcoin Miner

The architecture of this project is shown in the image above. The first main module is the USB bus interface. This module will read and write to the CPU and gets all necessary data to the other modules. This module will be woken up and will read a bulk transfer packet. This packet will be sent to the packet decoder which will tell the main controller what type of packet is being sent and will get ready to send data to the hash module. The main controller will then be initialized with all the parameters to begin hashing. The controller will start hashing of the block as it is being read in from the USB bus. If the hash module computes a valid hash it will put the hash in the output hash storage and will be ready to send it back to the CPU to place onto the block chain. At any point in this process 2 things will happen. The CPU could send a new block with an updated time parameter and the hash module will have to restart the calculation. The other thing that could happen is that another user found a valid block first and the CPU will tell the miner to halt all calculations and go into an idle mode.

3.2 Functional Block Diagrams

3.2.1.1 Hashing Module Block Diagram

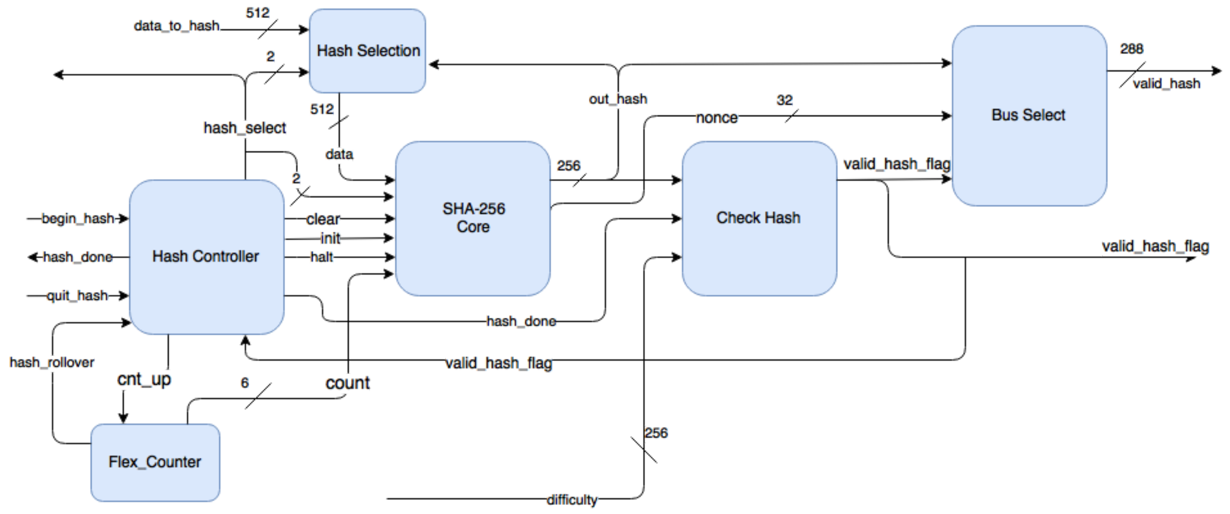


Figure 4: Functional Diagram for the Hashing Module

How the algorithm works (brief):

- The total data is 640 bits wide
- Hash the first 512 with predetermined w values of length 256
- Hash the second 256 bits (with padded 256 bits) with the previous hash replacing the predetermined w values
- If the output of the second hash has appropriate difficulty (leading zeros) send a valid_hash_flag pulse
- Therefore, the bitcoin algorithm is a actually SHA256(SHA256(Block)).

The core of the hashing module is to compute a SHA-256 hash on the input data. The input, data_to_hash, is directed the hash selection block. The hash selection block determines whether to choose the previous hash or the new data to hash depending on the stage of algorithm. The clear signal is used in the SHA256 block to tell it to use the default value for prev_hash and not the previous hash. This must be asserted when we calculate the first instance of any SHA256 algorithm.

This selected data is sent into the core block and and the SHA-256 block computes its hash. The check hash block compares this hash to the difficulty to see if it is a valid hash. It is valid then the valid_hash signal is set high and tells the main controller to be ready to transmit this hash back to the host. This means after we have computed the hash we have to send out_hash back into the input data to calculate the hash on that hash.
Once we have a valid hash, we send out the hash through valid_hash which is appended with the successful nonce value, hence valid_hash is 256+32 wide: 288 bits.

3.2.1.2 Hashing Module Timing Waveform

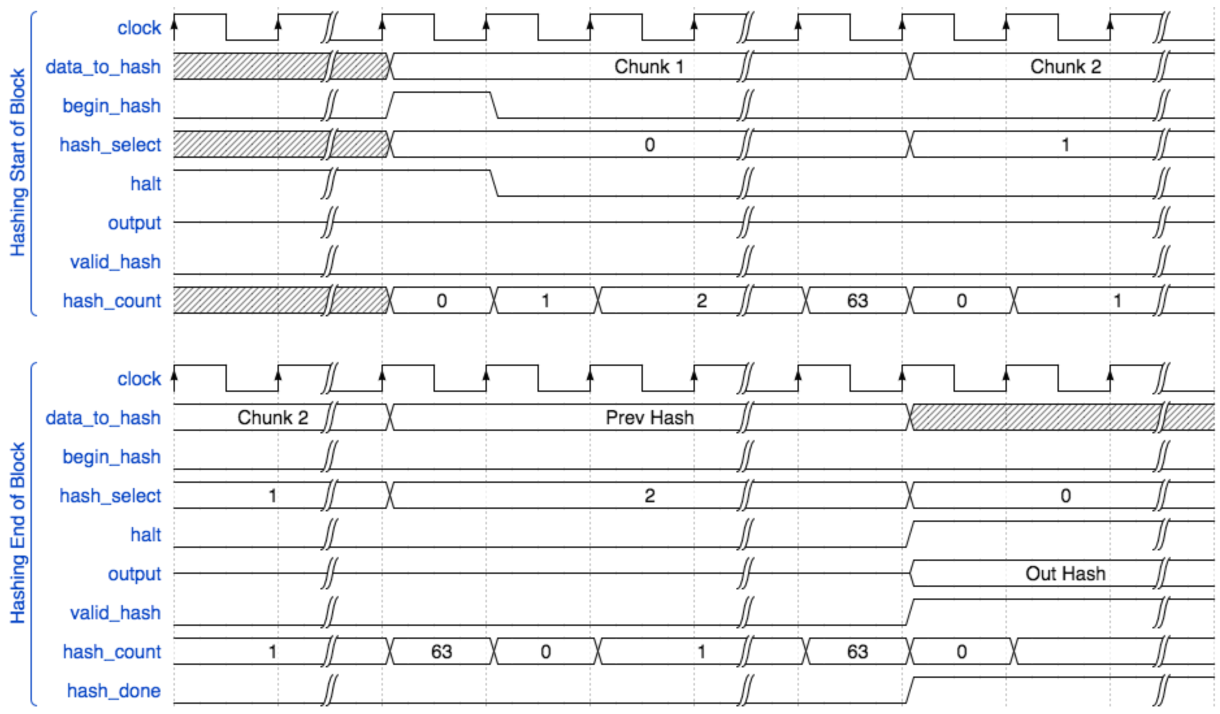


Figure 5: Hashing Module Timing Waveform

The hashing module initially receives data in chunks from the chunk decoder. It takes 64 clock cycles to hash one chunk. When hash select is 0 and 1, chunks 1 and 2 are hashed respectively. When hash select equates to two, the previous data re-enters the hashing module and goes through the hashing algorithm again. As soon as the SHA-256 algorithm is run again, halt is asserted to signify that no hashing is currently being conducted. If the hash is valid, valid hash is asserted and it is put on the output line bus.

3.2.1.3 Hashing Module RTL Diagrams
3.2.1.3.1 Hash Selection RTL Diagram

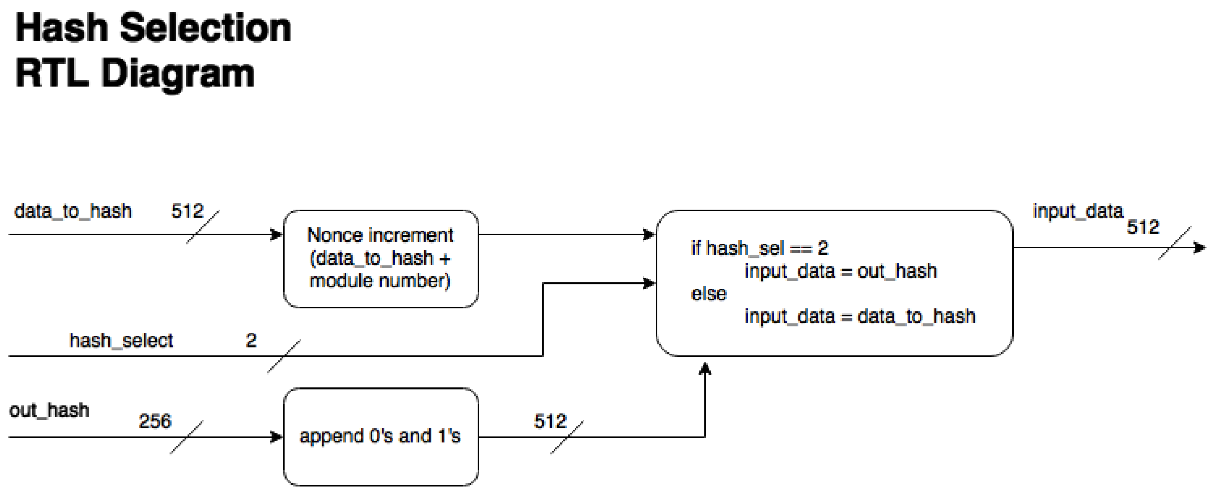


Figure 6: Hash Selection RTL Diagram

Table 4: Hash Selection Area Estimation Table

Component	**Type**	**# of Gates/Flip-Flops**	**Area (um^2)**	**Description**
Nonce Increment	3-bit Full adder	12	9000	8 bit flex counter
Hash Select	512- bit 2-way Multiplexer	2048	1536000	Combinational Logic

The Hash Selection block essentially serves as a 512-bit 2-way multiplexer and can increment the nonce value. The hash_select signal determines which hash to send out as the input_data. This determines whether the block sends the first chunk of data or the second. The multiplexer will ultimately equate to roughly 2048 gates, or approximately 15360000 um^2 in terms of area. The nonce increment is a 3-bit full adder which roughly comprises of 12 gates. This block will take about 9000 um^2.

3.2.1.3.2 SHA-256 RTL Diagram

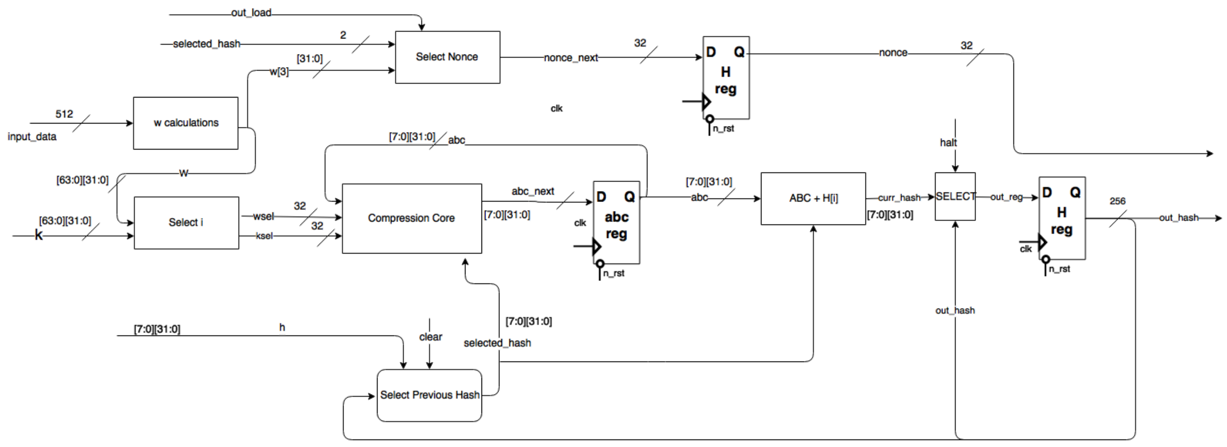


Figure 7: SHA-256 RTL Diagram

Table 5: SHA-256 Area Estimation Table

Component	Type	# of Gates/Flip-Flops	Area (um^2)	Description
W – Full Adders	Combinational	4608 * 8 = 36864	27648000	W – Full Adders
Byte Registers	Register	4096	8192000	Byte Registers
Compression Full Adders	Combinational	224	168000	Compression Full Adders

The SHA-256 block serves as the most space consuming aspect of our design. The block that calculates w, is just a series of full adders that ultimately contribute to roughly 27648000 um^2, with eight hashing modules. In addition, 8 hashing modules will produce around 4096 byte registers and 224 compression adders. These components take 6144000 um^2 and 168000 um^2 of space respectively. A major area of potential optimization arises from the w calculations.

3.2.1.3.3 Check Hash RTL Diagram

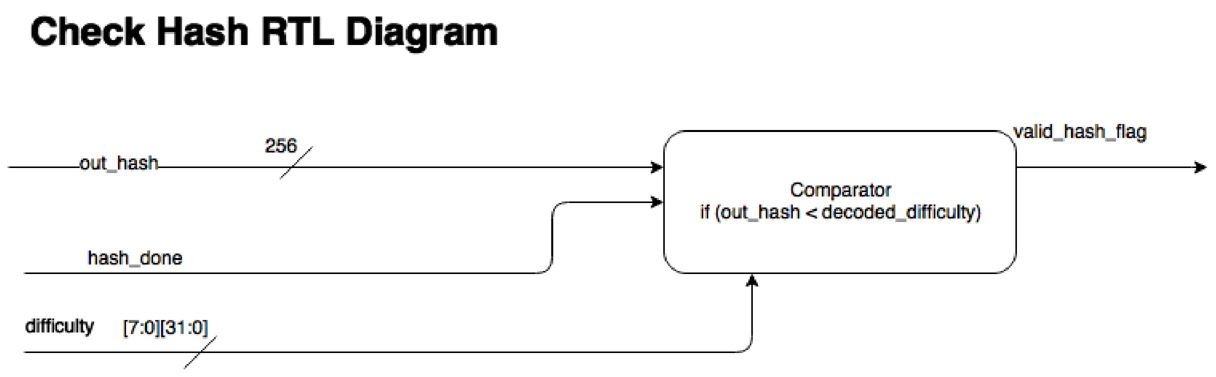
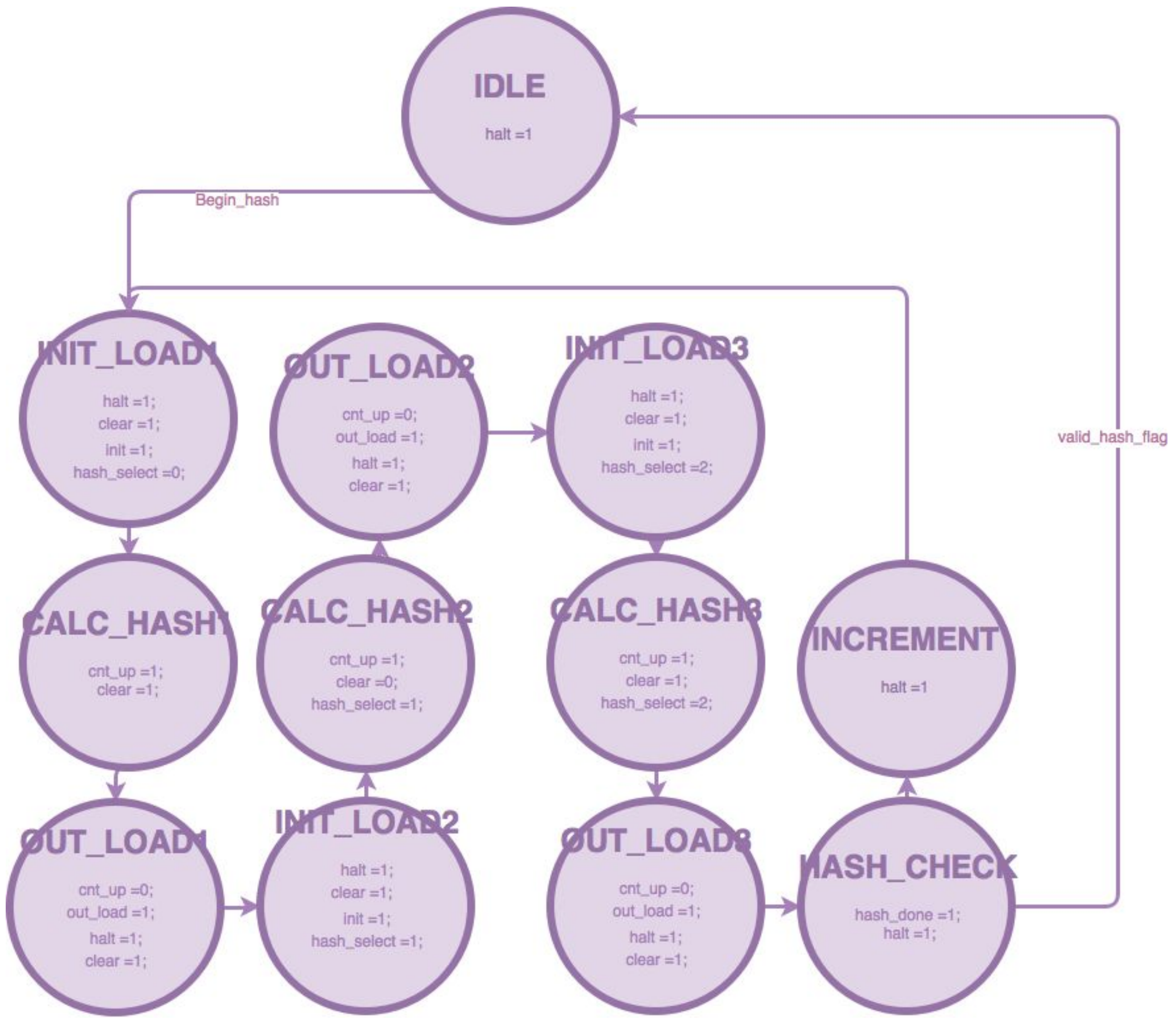


Figure 8: Check Hash RTL Diagram

Table 6: Check Hash Area Estimation Table

Component	**Type**	**# of Gates/Flip-Flops**	**Area (um^2)**	**Description**
Comparator	Combinational	256*8 = 2048	1536000	Comparator Block

3.2.1.4 Hashing Module Controller



The module manages the hashing algorithm by dividing the hashing process into a series of sequential chunks. A state machine seemed like the ideal tool for a sequential process such as this.

3.2.2.1 Packet Decoder Block Diagram

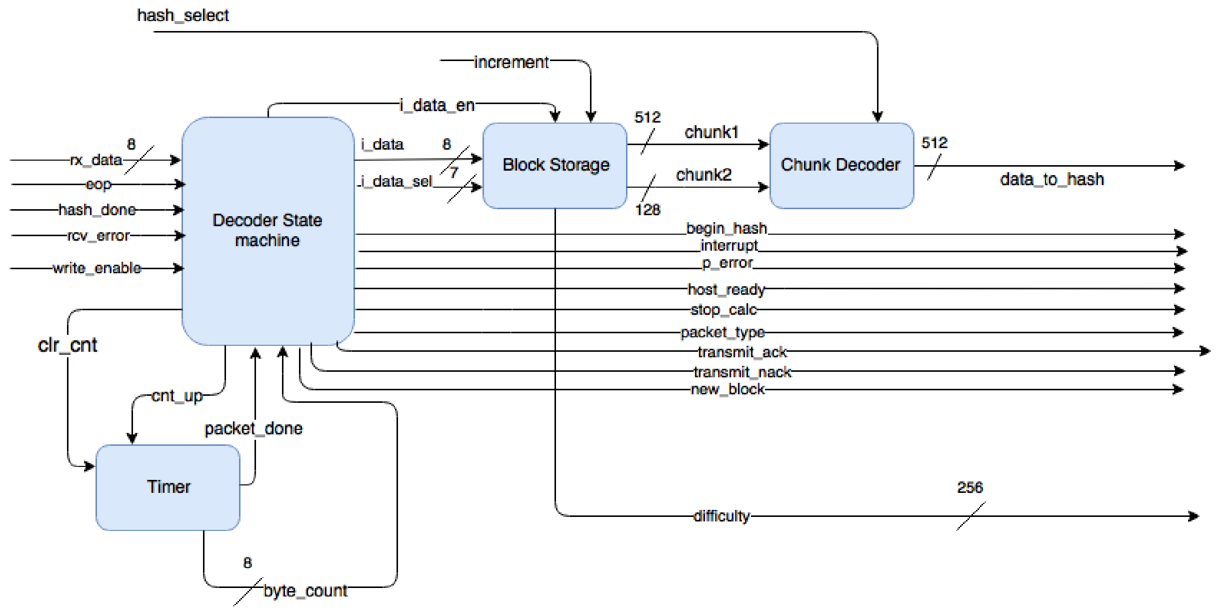


Figure 9: Functional Block Diagram for the Packet Decoder

The packet decoder consists of a main state machine, utilizing other derivative decoder modules and storage modules. In addition, the main state machine requires a timer block to count the number of bytes to write. The timer block receives a cnt_up signal from the main decoder state machine, which serves as a count enable for the counter, and is used throughout the state machine. The state machine will read the contents of the rx_data in the usb interface, and store them into an array of registers, represented in the diagram as block storage. The i_data_sel signal selects where in memory to store in every byte. The difficulty decoder takes in the target difficulty and performs a mathematical operation to determine the number of leading zeros, or the difficulty of the block. The new_block signal from the state machine signals the difficulty decoder to load a new target_difficulty from the block storage. The chunk decoder takes in the bytes stored in the block storage and sends blocks of the hash in sections to the hash module. It also implements nonce in the second chunk before sending the block to the hashing module.

3.2.2.2 Packet Decoder State Transition Diagram

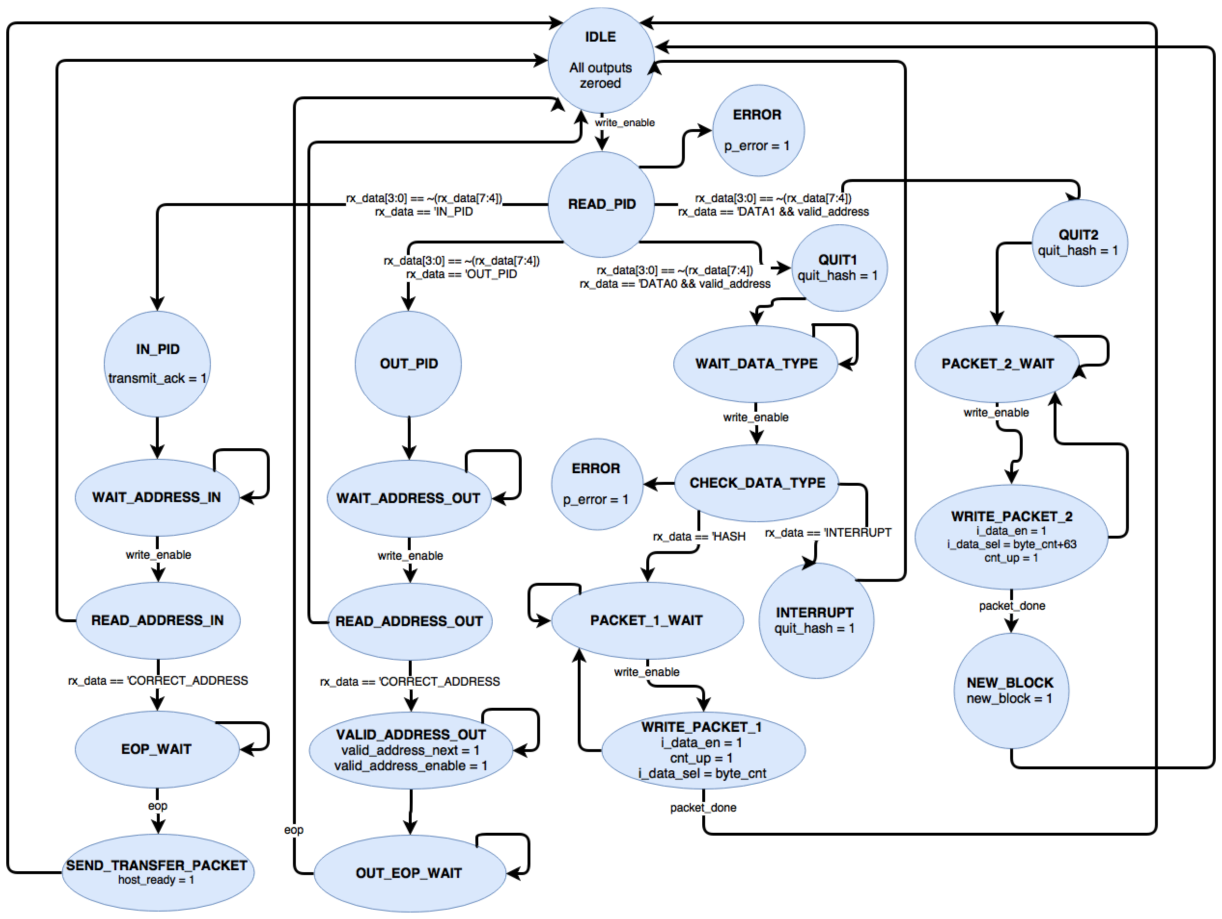


Figure 10: Packet Decoder State Transition Diagram

Initially, all the output signals in the packet decoder are zero, indicating the IDLE state. The next step is to determine the type of USB transaction, a IN or an OUT transaction. If the signal is a IN transaction, the decoder asserts the host_ready signal and returns to IDLE. If the signal is an OUT transaction, then a series of steps results in the data being written into storage. The first step is to read the PID of the data packet, which can signify data 0 or data 1. Then the decoder reads the data type which serves as an interrupt. If the data type is not a hash, the state machine asserts stop_calc and then returns to idle. If the data type is a hash, it will enable i_data_en and continuously send data to storage until the data is finished. When all the data has been stored, the state machine pulses the new_block signal and returns to IDLE.

3.2.2.3 Packet Decoder Timing Waveform

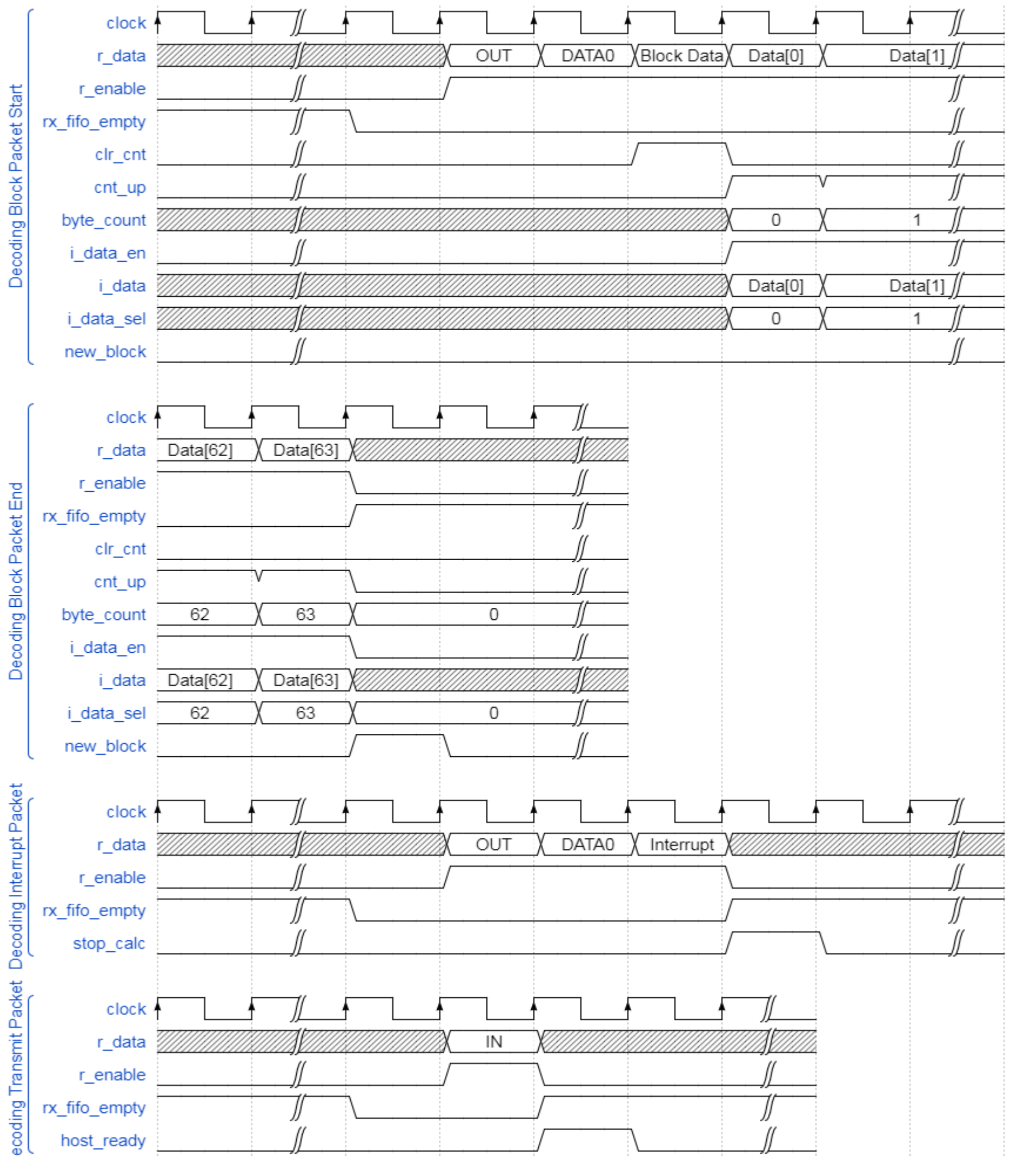


Figure 11: Timing Waveform for Packet Decoder

This timing diagram represents the 4 main operations the packet decoder does as it reads data from the USB module. Packet decoder will make decisions based on the data the USB receives. The first two diagrams are a description of reading the block header. The first three bytes it reads determines that the current transfer in and OUT transfer with the block header data. The next 64 bytes of data is the first chunk of the block header. Not shown here is a second bulk transfer that sends the last 16 bytes of data. Once all of the data is read in then the new block flag will trigger for one cycle, telling the main controller a block is ready.

The third waveform is an interrupt packet. The decoder will read in the 3 bytes of data to determine that it is an interrupt packet and will then trigger the stop_calc signal to tell the main to stop all hash calculations.

The last diagram shows how a transmission packet is read by the decoder. For this waveform the decoder needs to only read the first byte of the packet to determine that this packet will require a transmission and will send the host_ready signal to the main controller. This signal will allow the main to send back either a correct hash for the block header or an empty packet, signifying that a hash has not been found yet.

3.2.2.4 Packet Decoder RTL Diagrams

3.2.2.4.1 Decoder State Machine RTL Diagram

Table 7: Packet Decoder Area Estimation Table

Component	**Type**	**# of Gates/Flip-Flops**	**Area (um^2)**	**Description**
State Counter	Register w/ Reset	4	8000	4 bit to decode 9 states
Output Logic	Combinational	10	10000	Combinational Logic
Next State Logic	Combinational	10	10000	Combinational Logic

The decoder block is just a simple state machine with 11 states. In order to implement this state machine, a 4 bit state counter register is required, combined with an output and next state combinational block. Referencing previous labs, these combinational blocks contain roughly 10 gates. It is assumed that a flip-flop takes a minimum of 1000 um^2 and these values are doubled in order to allow for routing and wiring.

3.2.2.4.2 Timer RTL Diagram

Table 8: Packet Decoder Timer Area Estimation Table

Component	**Type**	**# of Gates/Flip-Flops**	**Area (um^2)**	**Description**
7 bit flex counter	Combinational	90	90,000	7 bit flex counter

3.2.2.4.3 Block Storage RTL Diagram

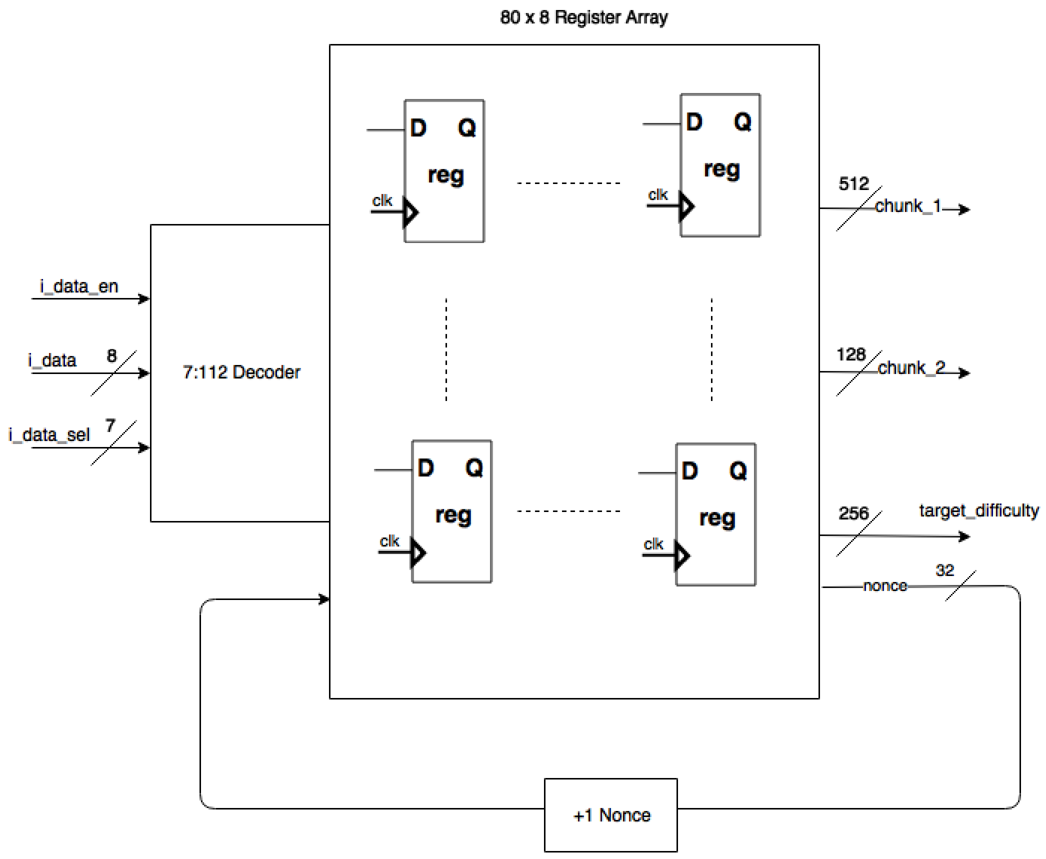


Figure 12: Block Storage RTL Diagram

Table 9: Block Storage Area Estimation Table

Component	**Type**	**# of Gates/Flip-Flops**	**Area (um^2)**	**Description**
Decoder	Combinational	640	480,000	4 bit to decode 9 states
Register Memory	Register	640	960000	An array of registers to store bytes of data

This design will take a significant portion in terms of size. The decoder involves roughly 640 gates and the array of registers will contain 640 registers. Assuming that a register without a reset and gate will have an area of 900 and 500 um^2 respectively, and that these values are doubled to account for routing, the decoder and the array will take up 480,000 and 960,000 um^2 respectively.

3.2.2.4.5 Chunk Decoder RTL Diagram

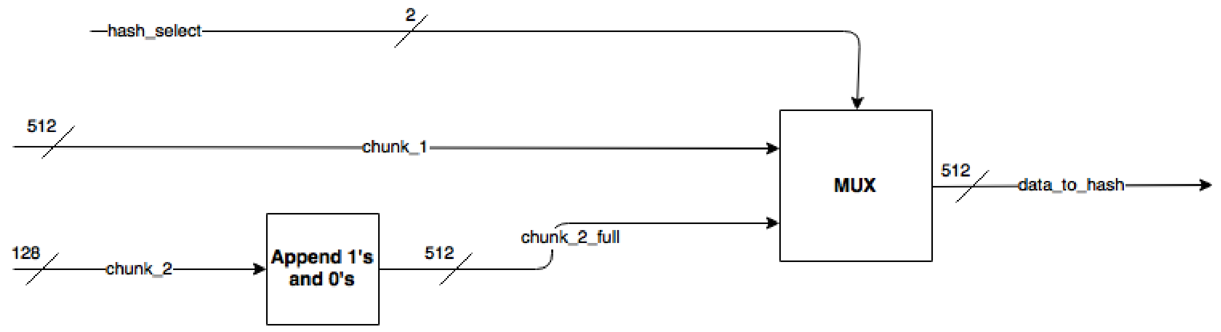


Figure 13: Chunk Decoder RTL Diagram

Table 10: Chunk Decoder Area Estimation Table

Component	**Type**	**# of Gates/Flip-Flops**	**Area (um^2)**	**Description**
Flex Counter	Register w/ Reset	100	100000	8 bit flex counter
Output Logic	Combinational	10	10000	Combinational Logic
Next State Logic	Combinational	10	10000	Combinational Logic

The Chunk Decoder will either utilize chunk_1 or chunk_2 depending on hash_select. One input to the multiplexer is chunk_1 which is 512 bits and the other input is chunk_2 which is 128 bits. Zeroes and ones will be appended to chunk_2 to make it 512 bits wide and the end a flex counter will be used to add 8 to the nonce. The Chunk Decoder takes up approximately 120 gates giving a grand total of 90,000 um^2 for the area estimate.

3.2.2.5.1 Hash Separator RTL Diagram

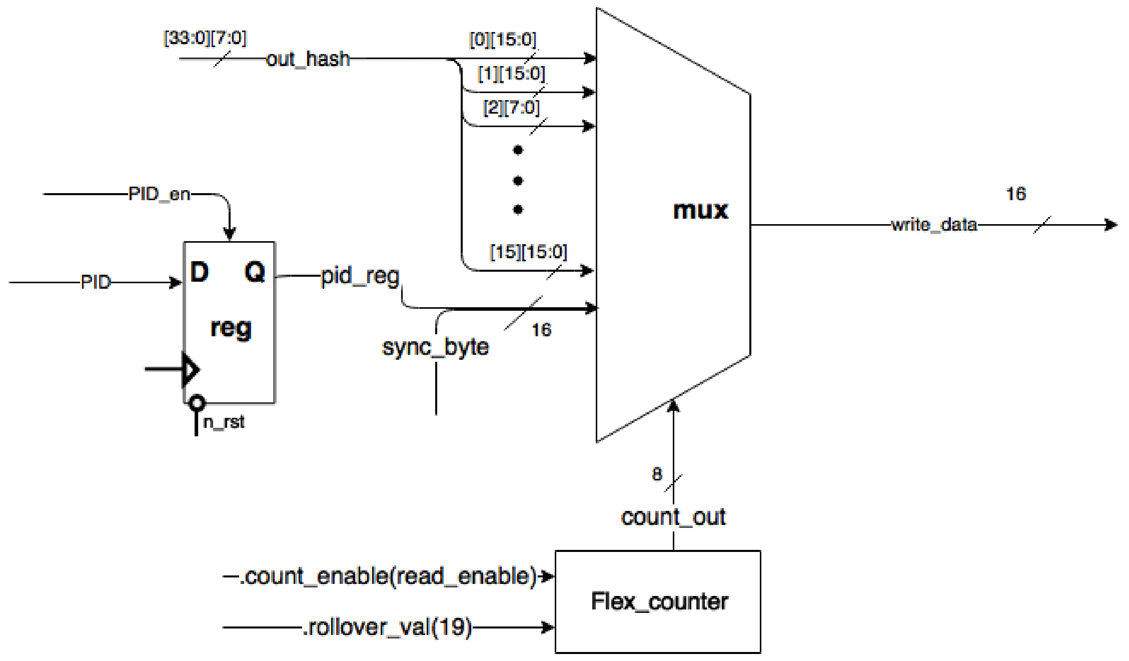


Figure 14: Hash Separator RTL Diagram

Table 11: Hash Separator Estimation Table

Component	Type	# of Gates/Flip-Flops	Area (um^2)	Description
Hash Separator Multiplexer	Combinational	1088	816000	19 way 16-bit Multiplexer
Counter	Combinational	64	48000	5 bit flex counter

This block takes a valid out_hash calculated by the hashing modules and writes it into the USB tx_data when read_enable is high. Read enable will enable the counter here 19 times order to write the hash, nonce, a sync byte and the proper PID which make up an entire USB packet. This is done with a large 19 way 16-bit wide multiplexer and a simple flex counter as the select line to pick which word to write to the transmitter shift register

3.2.3.1 USB Interface Block Diagram

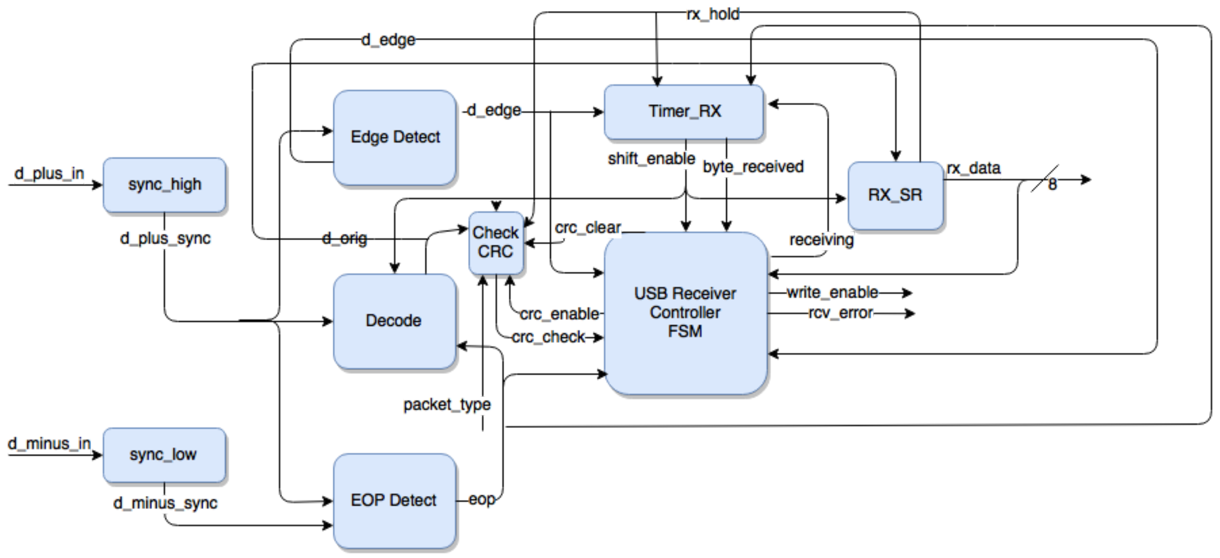


Figure 15: USB Receiver Interface Block Diagram

The USB consists of two lines, d+ and d-,which are inverses of each other. Both of these lines must go through a synchronizer in order to ensure a stable output. The d+ line goes through the edge detect, decode, and EOP detect blocks. The edge detect block just recognizes a transition in d+, and sends that signal to the receiver timer and the controller state machine. The decode block serves as the NRZi decoder and outputs the data serially from d+. It sends this information to the receiver shift register, which pushes the data into the block storage within packet decoder. The EOP detect block also takes in the d- line and asserts a EOP signal when both d+ and d- are held low, signifying an end of packet. This block sends this signal to the decoder and to the USB controller. The USB controller state machine takes in the signals and outputs the correct signals to a variety of blocks. A more comprehensive description of the state machine can be found in section 3.2.3.2.

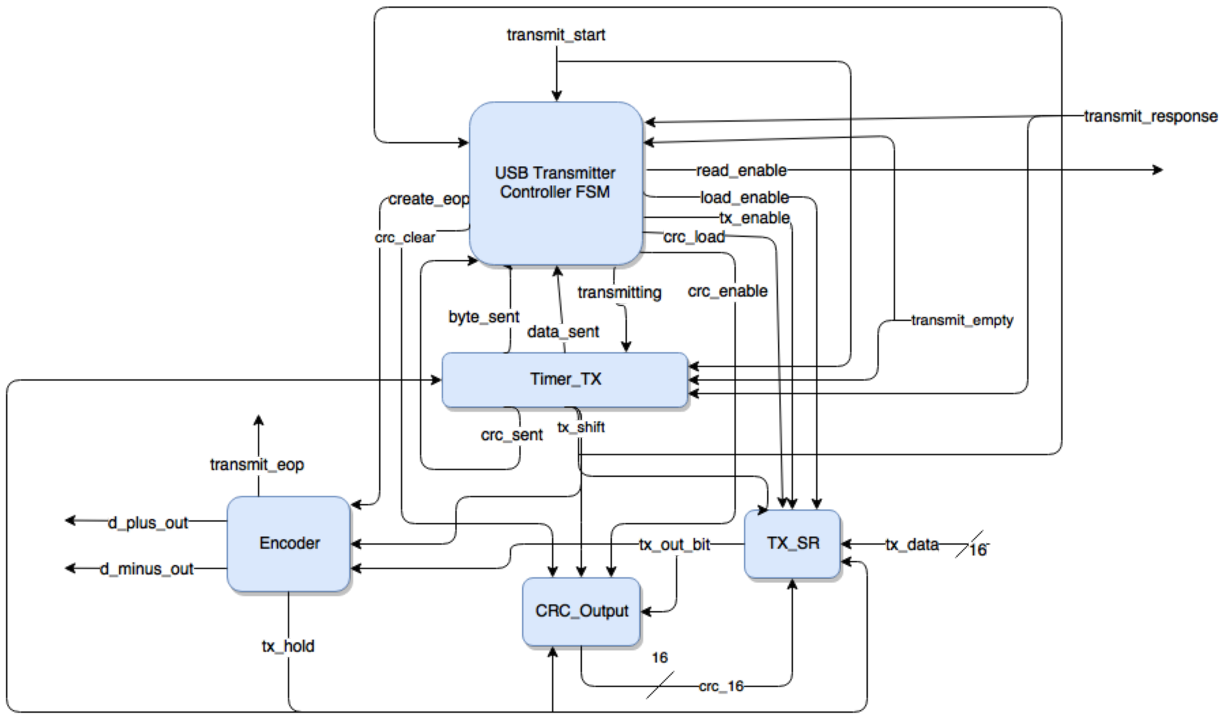


Figure 15: USB Transmitter Interface Block Diagram

Packet decoder handles sending data to the transmitter, which takes in data from another module and pushes it through a parallel to serial shift register to obtain a serial signal tx_out. This signal is then encoded through the NRZi encoder to obtain d_plus_out and d_minus_out. The transmitter is dependent on the block storage in packet decoder which stores all the data. The decoded output goes through the check CRC block and go through a linear feedback register with xor gates. The two types of CRC include CRC5 and CRC16, represented by the polynomials x^5 + x^2 + x^0 and x^16 + x^15 + x^2 + x^0 respectively.

3.2.3.2 USB Controller State Transition Diagram

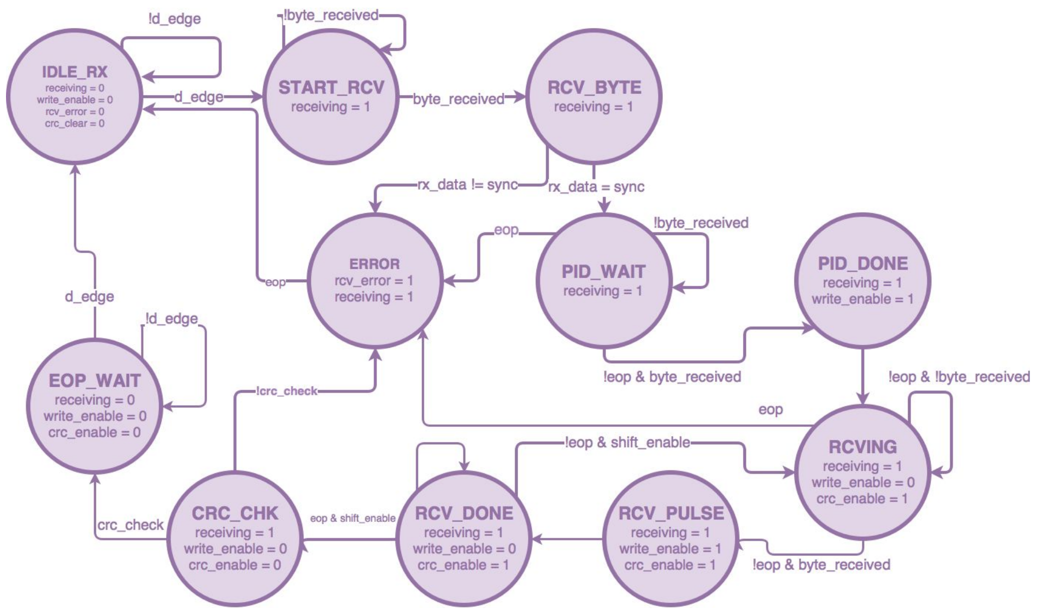


Figure 16: State Transition Diagram for USB Receiver Controller

If the d+ line has a rising edge transition, as represented by the d_edge signal, the bus is receiving data to write. The signal receiving is asserted at this state and stays high till an end of packet. As soon as a byte has been received, the state machine transitions to the next state. The sync byte is checked at this stage. If the data does not match the sync byte, then an error signal is asserted and the machine goes back to idle. If the data does match the sync byte, the state machine progresses. In the PID_WAIT state, an eop signal also takes the machine to the error state. A byte received signal with not an end of packet allows it to progress. The signal write_enable is asserted in the PID_DONE state and goes low in RCVING state and it will continue to the next state when another byte_received with no end of packet is encountered. The last thing that happens to complete the receiving cycle is to go through a cyclic redundancy check (CRC). CRC is a final check to ensure that all the non-PID fields and data packets do not encounter any errors during transmission. Afterwards, goes into the EOP_WAIT state and waits for another rising transition in the d+ line.

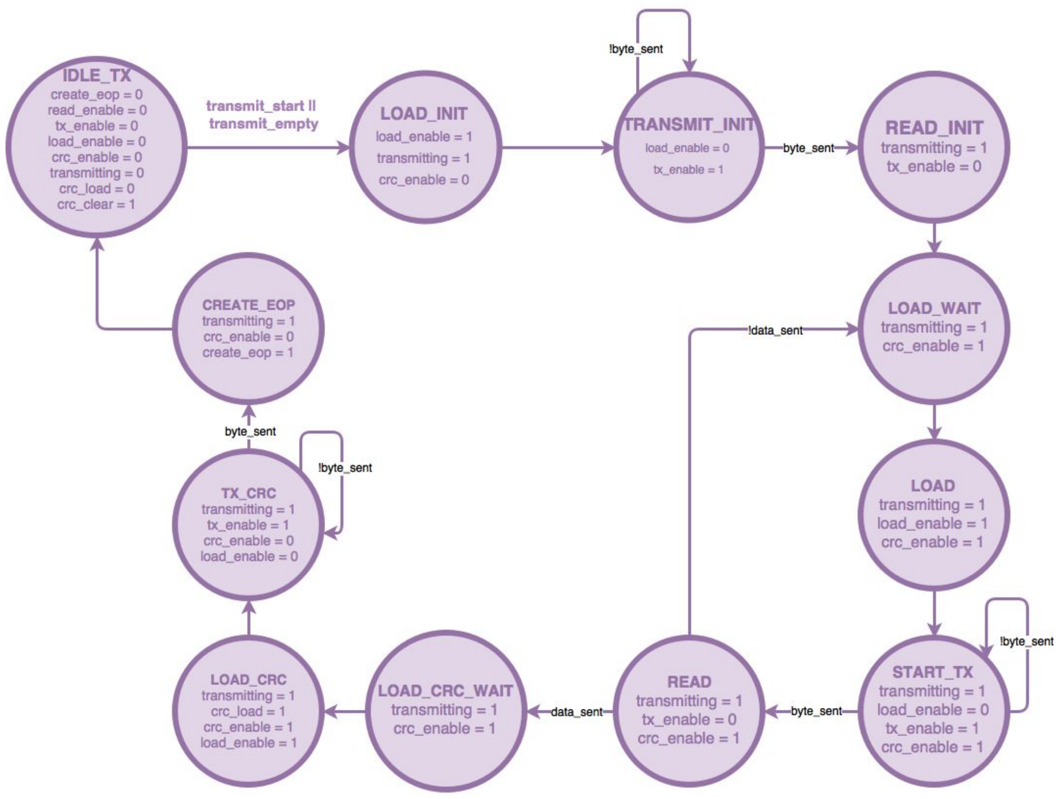


Figure 16: State Transition Diagram for USB Transmitter Controller

If the transmit_start or transmit_empty signal is asserted while the current state is IDLE the transmission begins. While on this path, transmitting is set to one and remains one until the state becomes IDLE again. Load enable is turned on for a pulse so the data can be loaded. Afterwards, tx_enable is pulsed to transmit data. Then the READ state is looped 5 times if transmit_empty was asserted or 34 times if transmit_empty was asserted. Once data_sent is asserted, the CRC is checked and until an entire byte is sent. After the CRC is checked for bit stuffing, an end of packet is created and transitions back to IDLE.

3.2.3.3 USB Interface Waveform Diagram

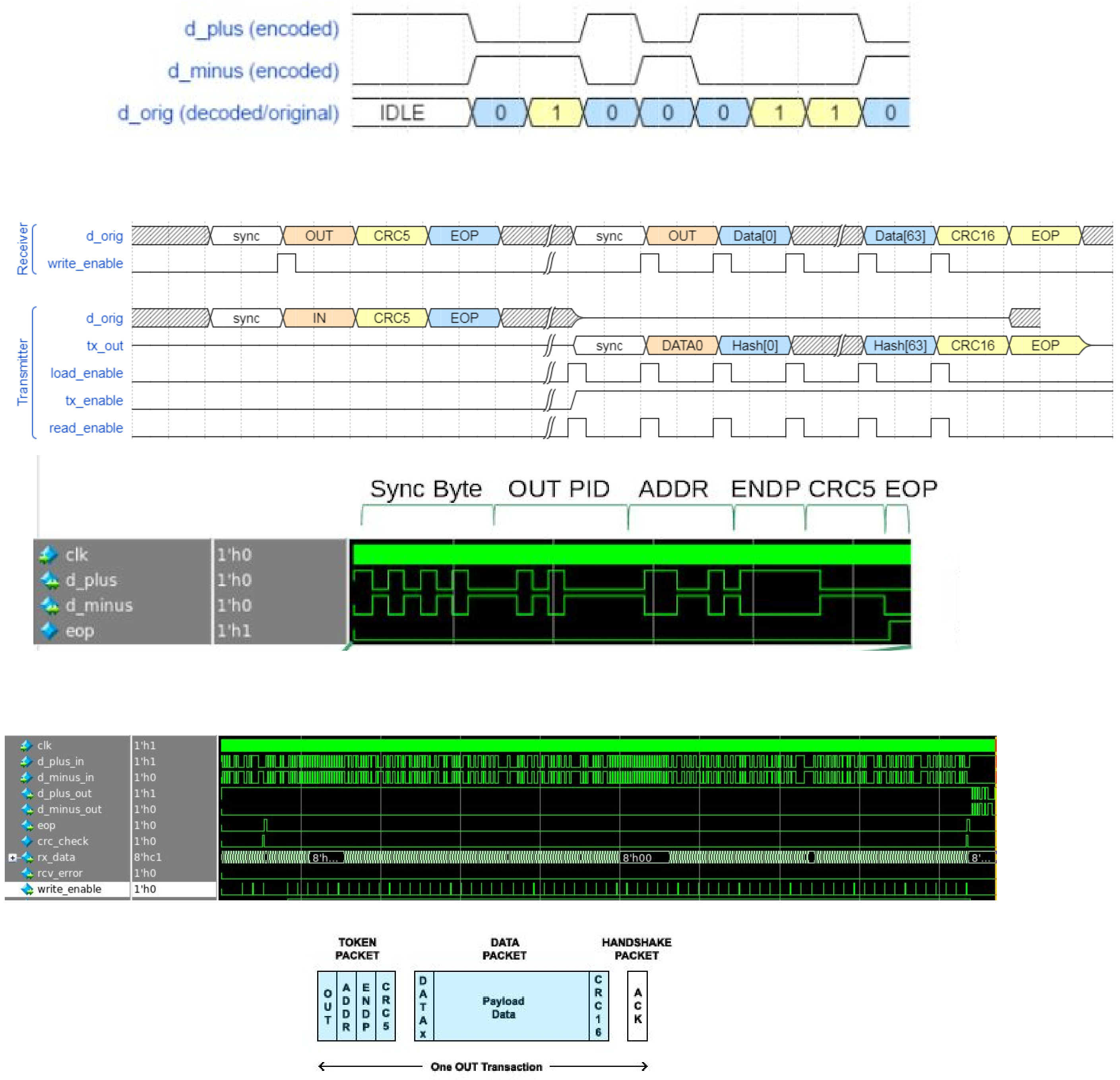


Figure 17: USB Interface Waveform Diagram

The first timing diagram describe how a USB decodes the lines d-plus and d-minus into 0’s and 1’s. NRZi decoding involves the transitions of the data line rather than the current value. When a transition is detected, the corresponding decoded state is a zero.

If no transition is detected, the state is represented as a one. In order to ensure that the clock does not drift too much, a zero is inserted after six consecutive ones. This condition is denoted as bit stuffing. The second timing diagram illustrates a token packet followed by a data packet. Both packets begin with a sync byte which synchronizes the clock. When acting as a receiver, write enable is pulsed every time a byte is received. When the USB bus is acting as a transmitter, tx_enable is asserted high as soon as the first byte of the data packet is received. In addition, load_enable and read_enable is pulsed initially at the same instance that tx_enable is asserted, and every consecutive byte till the end of packet. The third waveform image is from the top level test bench which illustrates a token out packet. The token packet consists of a sync byte, an out PID, the device address and endpoint, the CRC5, and an EOP. This packet signifies that a data packet will be received, which begins with a sync byte, the data, and an eop. In response, the USB transmitter will send an ack packet.

3.2.3.4 USB Interface RTL Diagrams

3.2.3.4.1 Edge Detect RTL Diagram

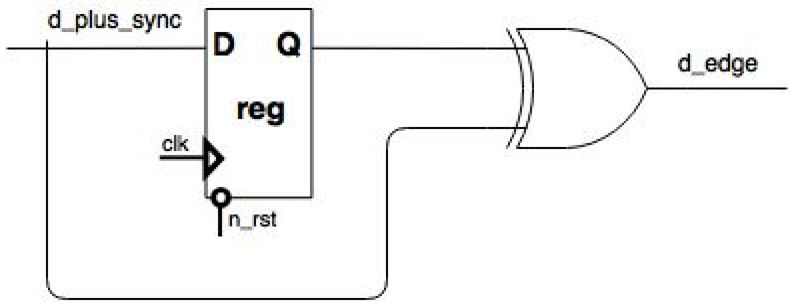


Figure 18: Edge Detect RTL Diagram

Table 12: Edge Detect Estimation Table

Component	Type	# of Gates/Flip-Flops	Area (um^2)	Description
D_plus_sync Register	Register w/ Reset	1	1500	Register to store the value of d_plus_sync
X-OR Gate	Combinational	1	750	X-OR gate to detect transitions

The edge detect component represents a very small portion of the design. It merely consists of a register and an X-OR gate. The size of a flip-flop and gate was assumed to be 1000 um^2 and 500 um^2 respectively and the size was doubled to allow for greater estimation of error after routing is included.

3.2.3.4.2 End of Packet (EOP) Detect RTL Diagram

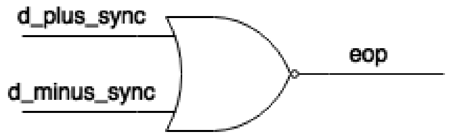


Figure 19: EOP RTL Diagram

Table 13: EOP Estimation Table

Component	Type	# of Gates/Flip-Flops	Area (um^2)	Description
Nand Gate	Combinational	1	750	NOR gate to identify the state where d_plus_sync and d_minus_sync are both low, signifying an end of packet

The end-of_packet (EOP) detector consists of a single NOR gate that takes two signals as an input. The size of a gate was assumed to be 500 um^2 and the size was doubled to allow for greater estimation of error after routing is included.

3.2.3.4.3 Decoder RTL Diagram

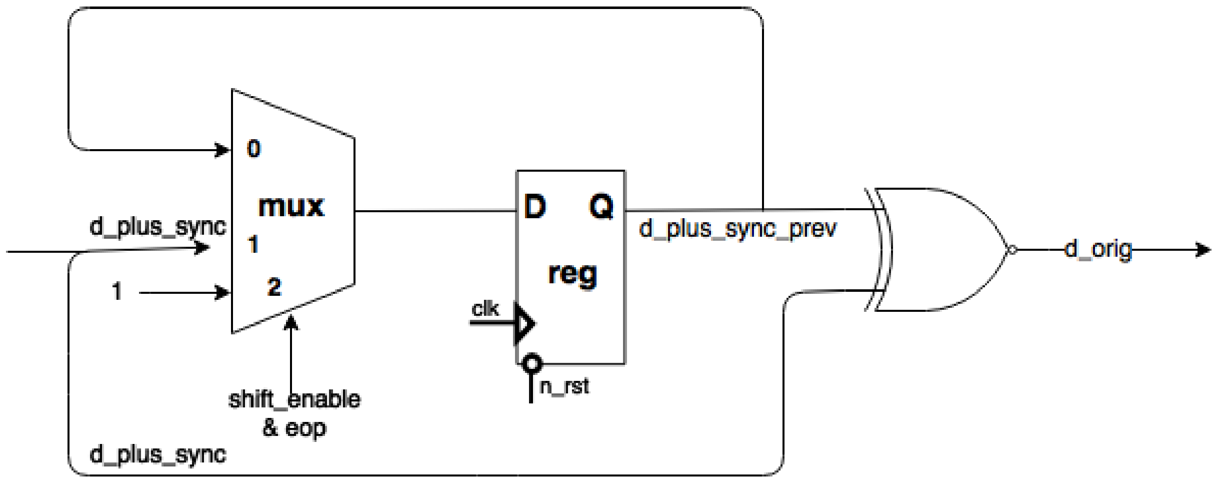


Figure 20: Decoder RTL Diagram

Table 14: Decoder Estimation Table

Component	Type	# of Gates/Flip-Flops	Area (um^2)	Description
Shift enabled multiplexer	Combinational	1	750	2:1 Multiplexer with enable, shift_enable.
D_plus_sync Register	Register w/ Reset	1	1500	Stores the value of the output of the shift enabled multiplexer
X-NOR Gate	Combinational	1	750	X-NORs the current and previous states of d_plus_sync
Final decoded output multiplexer	Combinational	2	1500	2:1 Multiplexer with enable, shift_enable && eop.

The decoder consists of an initial 2:1 multiplexer that outputs the value of d_plus_sync into a register. Then additional combinational logic is implemented in order to decode the value. The total area of this block is relatively small. The size of a flip-flop and gate was assumed to be 1000 um^2 and 500 um^2 respectively and the size was doubled to allow for greater estimation of error after routing is included.

3.2.3.4.4 USB Controller Finite State Machine (FSM) RTL Diagram

Table 15: USB Controller Estimation Table

Component	Type	# of Gates/Flip-Flops	Area (um^2)	Description
State Counter	Register w/ Reset	4	6000	4 bit to decode 9 states
Output Logic	Combinational	10	7500	Combinational Logic
Next State Logic	Combinational	10	7500	Combinational Logic

The USB controller is a finite state machine, so the block consists of three components: a state counter register, an output logic block, and a next state logic block. There are thirteen states to the controller, so a 4 bit register is required. The estimation for the output and next state logic was referenced from previous labs.

3.2.3.4.5 Timer_RX RTL Diagram

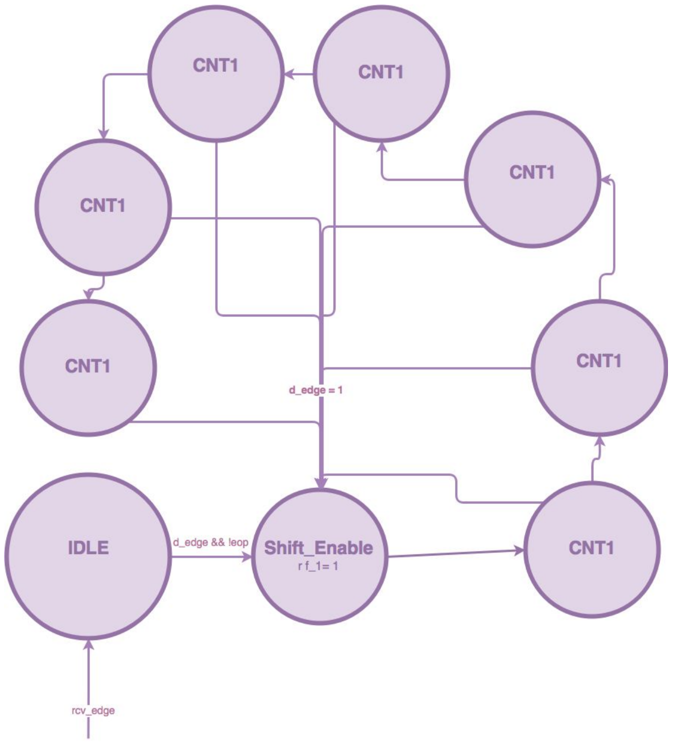


Figure 21: Receiver Timer RTL Diagram

Table 16: Receiver Timer Estimation Table

Component	Type	# of Gates/Flip-Flops	Area (um^2)	Description
Shift Enable Flex Counter	Combinational	50	37500	4 bit flex counter is roughly 50 gates
Combinational Logic Block	Combinational	2	1500	Inverter and AND gate to represent output
Byte_received Flex Counter	Combinational	50	37500	4 bit flex counter is roughly 50 gates
Bit stuffing Flex Counter	Combinational	50	37500	4 bit flex counter is roughly 50 gates

The timer_RX block is represented by multiple flex counters with a combinational logic block to assign the correct value of shift_enable. Each of the two 8 bit flex counters roughly consist of 120 gates, whereas the 6 bit flex counter, accounting for bit stuffing, roughly contains 90 gates. The combinational block is simple as it consists of two gates: an AND gate and an inverter.

3.2.3.4.6 Timer_TX Diagram

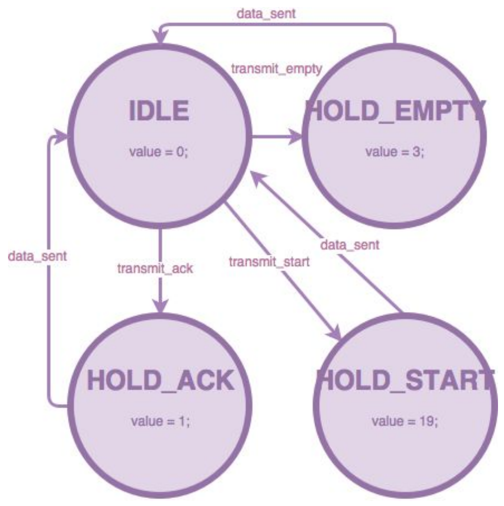


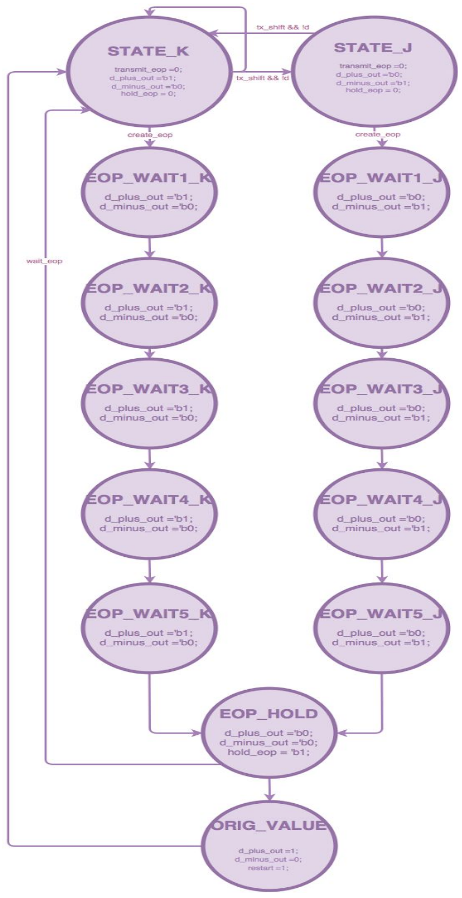
Figure 22: Transmitter RTL Diagram

Table 17: Transmitter Estimation Table

Component	**Type**	**# of Gates/Flip-Flops**	**Area (um^2)**	**Description**
Shift Enable Flex Counter	Combinational	50	37500	4 bit flex counter is roughly 50 gates
Byte_received Flex Counter	Combinational	64	48000	5 bit flex counter is roughly 64 gates
Transmit Empty Multiplexer	Combinational	1	750	Counter value determination

The timer_TX block is similar to the timer_RX block, with less components. This design only contains two flex counters chained together. One 4 bit counter connected to a 5 bit counter. These counters consist of 50 and 64 gates respectively.
The timer counts up to 34 and 5 depending on whether transmit_empty is enabled or not. When transmit_empty is high that means we must send a packet telling the host we are not ready with a valid hash and we transmit 5 bytes with only one byte of data. When transmit_empty is low we are transmitting 34 bytes of data to send a hash packet which is 32 bytes of data and 2 bytes of some metadata.

3.2.3.4.7 Encoder Diagram



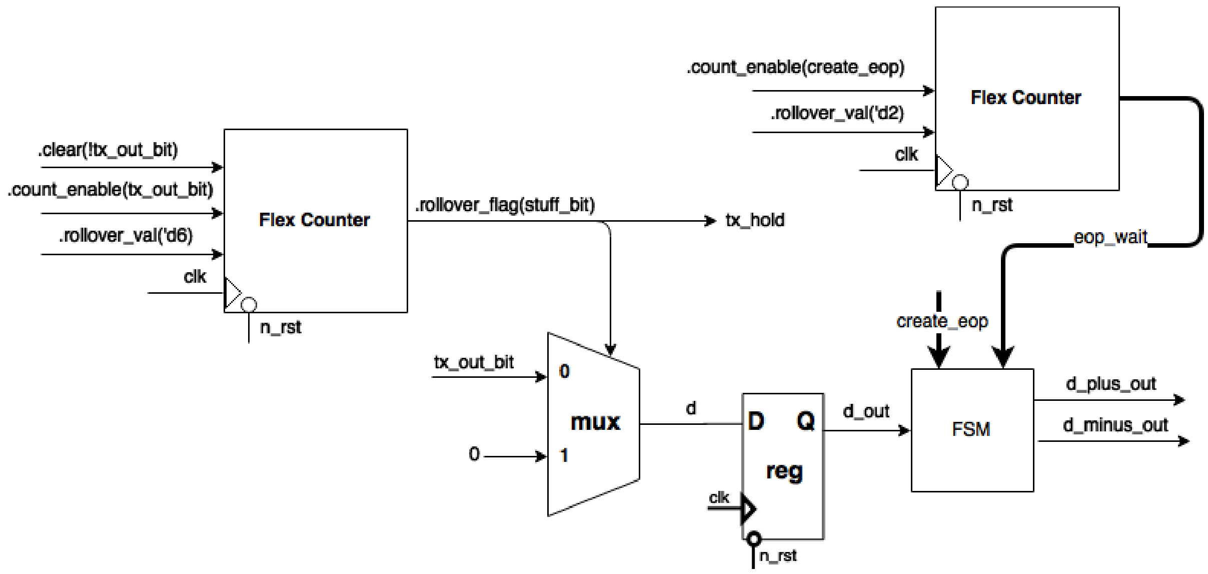


Figure 23: Encoder RTL Diagram

Table 18: Encoder Estimation Table

Component	**Type**	**Gates/Flip-Flops**	**Area (um^2)**	**Description**
Tx_hold Flex Counter	Combinational	50	37500	4 bit flex counter is roughly 50 gates
Tx_out Multiplexer	Combinational	1	750	2:1 multiplexer to select tx output
Tx_out Register	Register w/ Reset	1	1500	Storing the value from the multiplexer output
Output Logic Block	Combinational	5	3750	Outputs the encoded data in the USB data bus.
D_plus_out Register	Register w/ Reset	1	1500	Stores the value for D_plus_out
Inverter	Combinational	1	750	Inverts the d_plus_out

The biggest portion of this design is the 6 bit flex counter, accounting for roughly 90 gates. Most of the design is simple register and combinational logic, consisting of multiplexers, storage registers, and an inverter. There is an output logic combinational block that will assign the correct values to the USB data line.

3.2.4.1 Main Controller State Transition Diagram

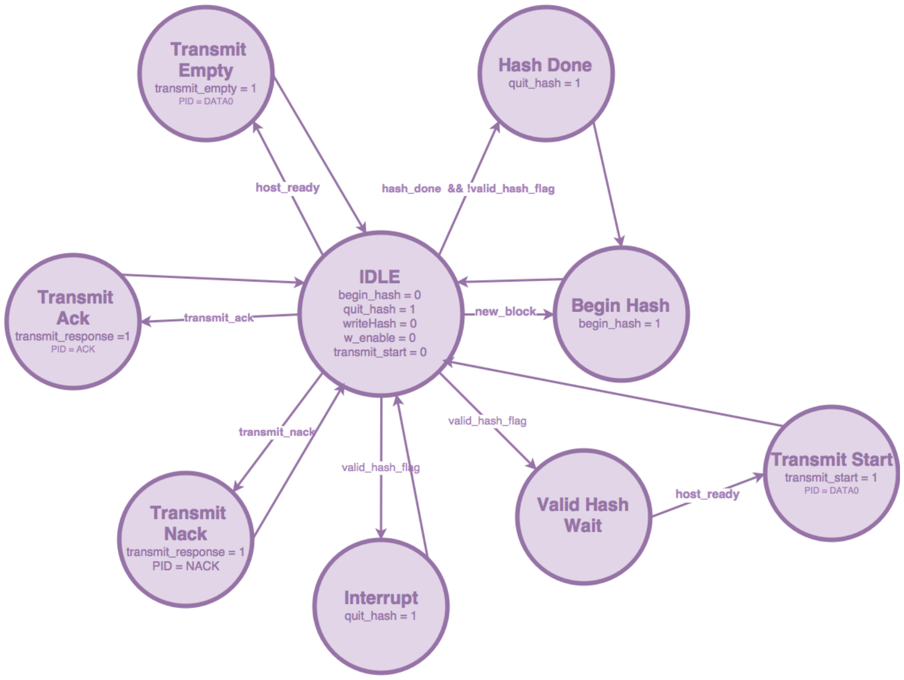


Figure 24: State Transition Diagram for Main Controller

The main controller synthesizes a large portion of logic connecting the USB interface, the packet decoder, and the hash module. The controller can be represented as a finite state transition diagram, as shown above. The initial IDLE state enumerates the varying necessary output signals, with all but quit_hash initialized to zero. The main controller manages different transmit signals: transmit empty, transmit ack, and transmit nack. Transmit empty will tell the USB to transmit an empty packet, whereas transmit ack and transmit nack lead to a transmission of ack or nack packets, respectively, The main controller will stop the calculation of the hash if it receives an interrupt signal. Furthermore, the main controller is also responsible for beginning the hashing process and signifying the end of the hashing algorithm.

3.3 Design Timing Analysis

Top 5 most critical paths Estimate

Table 19: Core Area Estimation Table

Starting Component	Propagation Delay (ns)	Combinational Logic	Propagation Delay (ns)	Ending Component	Setup Time or Propagation Delay (ns)	Total Path Delay (ns)	Target Clock Period (ns)
ABC registers in SHA-256	**0.4**	**Hashing function for-loop 2**	**63.84**	**ABC registers in SHA-256**	**0.2**	**64.44**	**10**

Input_data from hash selection	**0.4**	**Set up of w based on input data and random hashed values**	**15.84**	**W data inside SHA-256**	**0**	**15.84**	**10**
Out_Hash	**0.4**	**Hash Separator mux**	**2.8**	**Tx_sr in USB interface**	**.2**	**4.08**	**10**
Input Data Storage	**0.4**	**Difficulty Decoder**	**2.4**	**Target difficulty**	**0**	**2.8**	**10**
USB Tx_out Register	**.4**	**Output logic used**	**.5**	**USB D_plus_out Register**	**.2**	**1.32**	**10**

3.3.1 Path descriptions:

1. These are the components of the delay:
    a. 64-way 32-bit wide mux = 10 gates: 10 * 0.2ns = 2ns
    b. 4 32-bit full adders:
        i. Code:
            1. temp1 := h **+** S1 **+** ch **+** k[i] **+** w[i]
            2. e := d **+** temp1
        ii. Each full adder has 64 gate critical path: 64 * 0.2 ns = 12.8 ns
        iii. Total: 4 x 12.8 ns: 51.2 ns
    c. Propagation and setup time for flip-flops: 0.6 ns
    d. Wire and capacitive load is 53.8 ns * 1.2 = 63.84
    e. With setup and propagation time = 64.44

    This is the main constraint to our entire design and is the core of the calculation in the hashes. This is how we set the overall time constraint for our design as it allows us to calculate up 500,000 hashes per second. This could be reduced if the adders used are not ripple-carry but are instead carry-lookahead adders. We are not sure if this is possible in verilog currently.
2. These are the components of the delay:
    a. 2 32-bit full adders:
        i. Code:
            1. w[i] := w[i-16] **+** s0 **+** w[i-7] **+** s1
        ii. Each full adder has 64 gate critical path: 12.8 ns
        iii. Total with capacitive load = 15.36 ns
    b. 2 XORs
        i. Code:
            1. s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightshift 10)
        ii. Each xor: 0.2ns
        iii. The rightrotate just involve re-wiring, so no propagation delay

        iv. Total with capacitive load: 0.48ns
    c. Total: 15.84ns

This path is not from register to register because it is only combinational logic that needs to be done before other calculations can be made. Currently we are just giving this portion one clock cycle to complete instead of adding it to the first timing path so that it doesn't increase the first path's timing analysis.

3. Difficulty calculation

    This consists of a 24-bit barrel shifter to compute a variable number of bit shifts based on the input data. There are a total of 110 2-1 multiplexers in this barrel shifter but there is a depth of $\log_2(24)$, or 5 when rounded up. 5 multiplexers of propagation with each one taking 2 gates this is a propagation time of 2 nanoseconds and when accounting for capacitance this is 2.4 ns. Well under our desired clock time. There is no output register for this calculation because this data also just needs to be ready once the output data is ready from the first timing path.

4. Hash Separator

    This separator is using a 19-way 16 bit mux to select which of these bytes it needs to write to on the transmit line. This can be accomplished by 7 tiers of 8-bit 2-way muxes. Each mux would take 2 gates to propagate through and all 7 tiers would take 14 gates in total. With capacitance loading and setup times this adds up to 4.08 ns of total delay. This is well under the timing constraints.

5. USB tx_out

    This timing path is just through some of the most complicated portion of the USB interface module. Most of this module is sequential and does not have large timing paths. This path was included to show that nothing in USB will be a big timing constraint for our design. This path in particular goes through the output logic of the main state machine inside of the USB controller and is a rough estimate of the number of gates that it will need to go through.

3.3.2 Actual Timing Analysis vs Estimate

The actual critical path that the compiler recognized was from the current state register in our main controller through the selection blocks in packet decoder that determine which chunks will be the input to the hashing module. The data that was selected is then propagated through to the ABC registers in SHA-256. This is the critical path that we predicted during our estimation. Our estimate forgot to take into account the signals that select that data initially but this is still the same path.It takes significantly shorter than we estimated originally because we gave the compiler a compilation target of 10 ns to meet when designing this block. In the report file (Bitcoin_miner/reports/bitcoin_miner.rep), the timing analysis for that path was estimated to be at 9.97 ns. This seems close to the actual clock rate of 10 ns but in reality the design does not come close to hitting that much delay. The process in which this path is used is given extra clock cycles because it must wait for other data to be ready. Whenever we change the chunk from the main controller, the hash controller waits an extra initialization state to load in the new data before doing any calculations on this, this was necessary anyway to make the design work regardless of the critical path. This was the only path that got close to the 10 ns cut off and shows that we met our target clock period of 10 ns we set out for.

3.4. Area Estimation

Table 20: Hash Separator Estimation Table

Core Area Calculations				
Name of Block / Module	**Category**	**Gate/FF Count**	**Area (um^2)**	**Area (mm^2)**
USB Interface	**Mixed**	**869**	**1043250**	**1.04325**
Packet Decoder Module	**Mixed**	**2646**	**2786000**	**2.786**
Hashing Module	**Mixed**	**49388**	**47929000**	**47.929**
Total Core Area		**52903**	**52525500**	**52.5255**
Total Chip Area			**54573657.2**	**54.5736571**

3.5. Estimated Area vs Actual Area

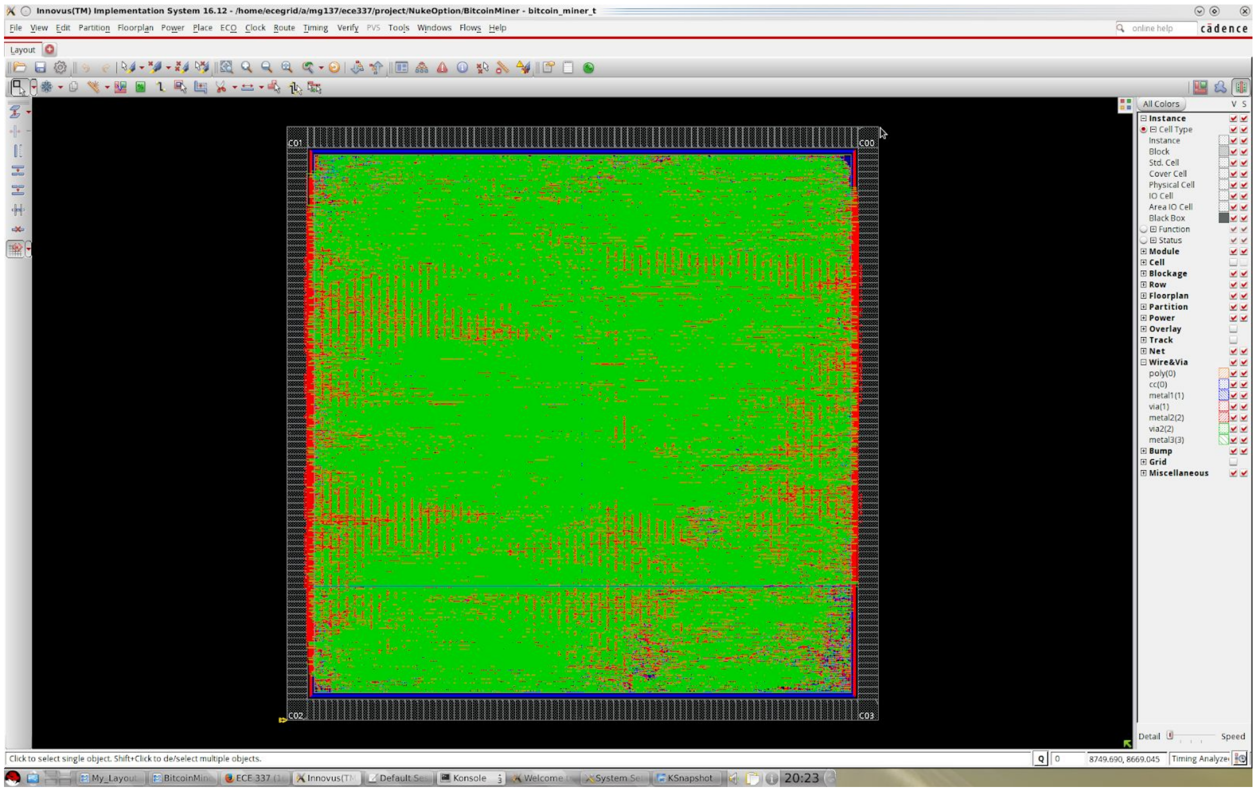


Figure 25: **Design Layout**

Area: 8728 um^2 x 8665 um^2 = 75.63 mm^2
The actual design layout was larger than expected, considering the synthesized design estimated it at 22 mm^2. A large portion of this project involved rewiring and obtaining the correct data representation. In order to change the form of the data, such as switching from little to big endian, modules were required to accomplish this task, which was uncounted for originally. In addition, wiring took a lot more space than expected, as the synthesized design estimated a size of 22mm^2 compared to the actual size of 75.63 mm^2, implying that the wiring and physical component placement took about 3.5 times the estimated value. We used an assumption that the wiring would be about 2 times the estimated values, and set our design criteria using that information. Also because of the size of the input data to the hashing module (512 bit), any small conditional statements, lead to huge increases in area. For example, if we forgot to count one if statement in that block, it would result in a 512-bit multiplexer taking up to 2048 gates and ~2 mm^2 of area. These small adjustments added up and pushed our design over our limit for the area

4. Success Criteria

4.1.1 Fixed Criteria

1. (2 points) Test benches exist for all top-level components and the entire design. The test benches for the entire design can be demonstrated or documented to cover all of the functional requirements given in the design specific success criteria. **(COMPLETED)**
2. (4 points) Entire design synthesizes completely, without any inferred latches, timing arcs, and, sensitivity list warnings. **(COMPLETED)**
3. (2 points) Source and mapped version of the complete design behave the same for all test cases. The mapped version simulates without timing errors except at time zero. **(COMPLETED)**
4. (2 points) A complete IC layout is produced that passes all geometry and connectivity checks. **(COMPLETED)**
5. (2 points) The entire design complies with targets for area, pin count, throughput (if applicable), and clock rate. The final targets for these parameters will be determined by course staff based on your design review. Failure to reach any of the targets will result a score of 1 out of 2 provided that you are within 50% on area, 10% on pin count, and 25% on throughput. Doing worse in any category will result in a score of 0 out of 2. **(COMPLETED)**
    a.) Area: 55 mm^2
    b.) Pinout: 6 (2-pin USB and power, clock, reset, and ground)
    c.) Clock Period: 10 ns

4.1.2 Design Specific Success Criteria

1. (2 points) Demonstrate by simulation of Verilog test benches that the Bitcoin Miner design is able to successfully find a valid nonce for a block and returns a hashed header. The nonce value can be confirmed by utilizing an online SHA256 calculator. **(COMPLETED)**
2. (2 points) Demonstrate by simulation of Verilog test benches that the Bitcoin Miner design is able to hash a block header via $(\text{SHA-256})^2$ algorithm. **(COMPLETED)**
3. (2 point) Demonstrate by simulation of Verilog test benches that the Bitcoin Miner design is able to receive an interrupt packet and halt the computation. **(COMPLETED)**
4. (1 point) Demonstrate by simulation of Verilog test benches that the Bitcoin Miner design is able to receive a block packet via bulk transfer USB. **(COMPLETED)**
5. (1 point) Demonstrate by simulation of Verilog test benches that the Bitcoin Miner design is able transmit a hashed packet via USB. **(COMPLETED)**

4.2. Explanation of Success Criteria

4.2.1. Fixed Criteria

1. See appendix for file names and locations for all test benches created for verification of existence
2. The log folder containing the verification of no latches is in the directory “mg138/Bitcoin_miner/docs/bitcoin_miner.log”
3. This fixed criteria was proven during the demonstration with Dr. Johnson and Mochen. Running sim_full_source and sim_full_mapped from our included makefile will also demonstrate this fixed criteria.
4. Refer to images in “mg138/Bitcoin_miner/docs/bitcoin_miner/bitcoin_layout.png” and “mg138/Bitcoin_miner/docs/bitcoin_miner/connectivity.png” for screenshots regarding the design layout and connectivity issues. The layout was generated on Yash Bharatula’s (mg137) account, which is why there are no layout generation files in the mg138 main folder.
5. See section 3.5 for discussion regarding estimated area and actual area. See section 3.3.2 regarding the timing analysis of our top level file.

5. Design Verification

5.1 Design Verification Overview

Table 21: Design Verification Plan

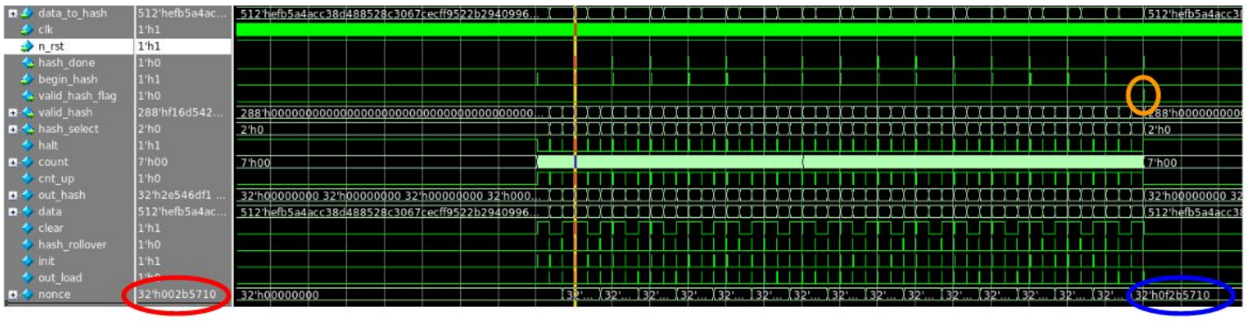
What to Verify	Design Module Involved	Verification Procedure Summary	DSSC(s) Proved	Comments
Find a valid nonce and return valid hash header successfully	Hash Module & Packet Decoder	Compare hash output and nonce values of certain blockchains as described in the official bitcoin website: https://blockchain.info	DSSCs 1 & 2	All values in a blockchain header with the correct nonce values of all transactions are provided on this website
Correctly hash a block header via (SHA-256)^2 algorithm	Top Level	Compare output with known output hash from the given block: this can be acquired from the same website the block came from (https://blockchain.info)	DSSC 2	Choose block header from the website and compare our algorithm's output to the title of the block.
USB interface successfully receives an interrupt packet and halts computation	Top Level	Interrupt packet test (check process of hashing once interrupt signal is asserted)	DSSC 3	Test bench sends an interrupt and main controller sends an interrupt signal to the hash module
USB interface correctly receives a block packet via bulk transfer	USB Receiver Module & USB Transceiver Selector	Compares received block with the actual block values Check CRC of that data with crccalc.com	DSSC 4	The test bench will send the data and then compare the received information with what was sent
Successfully transmit a valid hashed packet via USB	USB Transmitter Module & Hash Separator & USB Transceiver Selector	Test bench sends valid hashed header Check CRC of that hash with crccalc.com	DSSC 5	The test bench will provide a valid hash to the Hash Separator and then validate the transmitted output

5.2 Test Scenario Breakouts

Find Valid Nonce & Return Valid Hash Header

- Test Bench Expectations/Requirements:
    - Represent data from block # 100,000 on blockchain.info in the correct format: ensure appropriate endianness, order of bits, etc. to make sure that the input for the hashing module gets the data in the expected format
    - Transmit the input block via USB to the receiver
    - Then use the waveform explorer to verify that the miner found the correct hash at the right time
- Main Verification Test Steps:
    1. Download the data from Block 100,000 (including prev hash, merkle root, difficulty, nonce, etc.)
    2. Ensure the data is in the correct format, call the flip_endian task appropriately so that the input matches up correctly for what the hashing module expects
    3. Select a nonce value slightly lower than the required nonce
    4. Send the packet via USB to the receiver and store it in packet decoder
    5. The hashing module will begin calculating the current header
    6. The nonce is initially incorrect and will be incremented by the packet decoder and hashing module
    7. Onec the nonce is correct the valid hash flag will be set high and we can verify that it is indeed correct
- Proof of verification

Please refer to figure 26 for all description of test bench waves.
The screenshot below is of the hashing module showing it compute 16 invalid hashes until it calculates one valid one. The initial test input to this module was the 640 bit block header,
640'h0100000050120119172a610421a6c3011dd330d9df07b63616c2cc1f1cd00200000000006657a9252aacd5c0b2940996ecff952228c3067cc38d4885efb5a4ac4247e9f337221b4d4c86041b002b5710,
This header is almost a valid header but the nonce portion of this header is the last 32 bits, 0x002b5710. The correct nonce should be 0x0f2b5710. In our test bench you can see the initial nonce in the red circle is loaded with 0x002b5710. Then 16 rounds of (SHA-256)^2 are computed. This is shown by the 16 hash_done and begin hash signals being pulsed high. Then at the end the valid_hash_flag is set high, shown in the orange circle. At that time the out_hash register stays at its current value and you can see in the blue circle that the valid nonce 0x0f2b5710 is the nonce register. This nonce can be confirmed to be the true nonce of block #100,000 by going to the website blockchain.info and seeing the nonce is 274148111. That number is 0x0f2b5710 when converted to hexadecimal and written in little endian. It is too long to be seen on the waveform but in the out_hash register is also the correct hash, 000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1afd33e506.



```
data_to_hash     512'hefb5a4ac...
clk              1'h1
n_rst            1'h1
hash_done        1'h0
begin_hash       1'h1
valid_hash_flag  1'h0
valid_hash       288'hf16d542...
hash_select      2'h0
halt             1'h1
count            7'h00
cnt_up           1'h0
out_hash         32'h2e546df1 ...
data             512'hefb5a4ac...
clear            1'h1
hash_rollover    1'h0
init             1'h1
out_load         1'h0
nonce            32'h002b5710
```

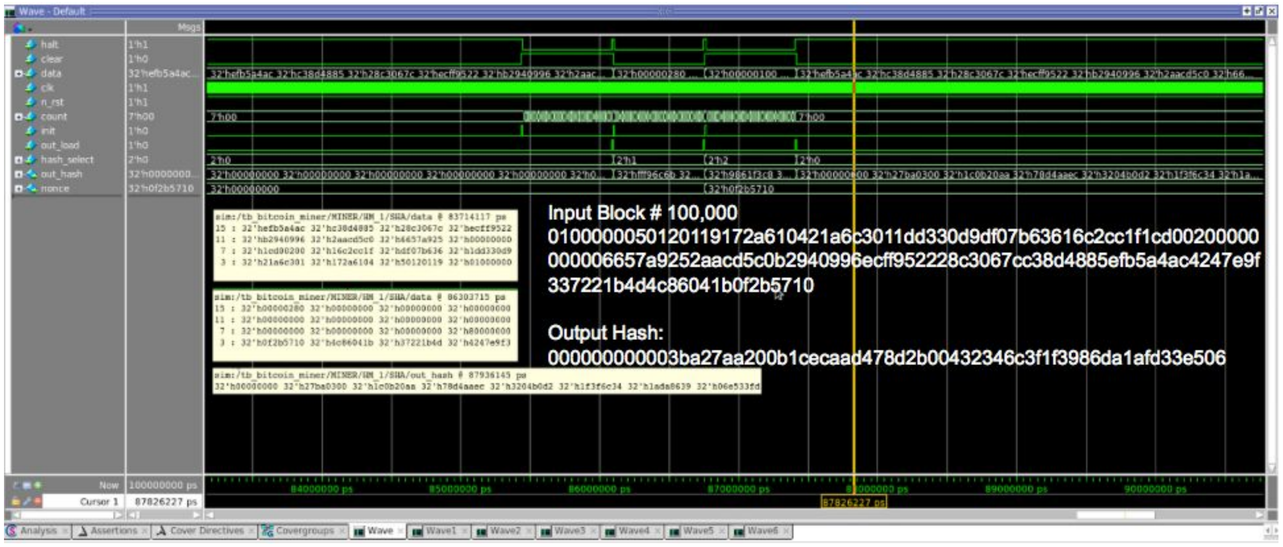
512'hefb5a4acc38d488528c3067cecff9522b2940996...
512'hefb5a4acc3
288'h00000000000000000000000000000000000000...
288'h00000000
2'h0
2'h0
7'h00
7'h00
32'h00000000 32'h00000000 32'h00000000 32'h000...
32'h00000000 32
512'hefb5a4acc38d488528c3067cecff9522b2940996...
512'hefb5a4acc3
32'h00000000
32'h0f2b5710

Figure 26: Incrementing Nonce and finding a valid hash test

Hash Block Header via (SHA-256)^2 Algorithm

- Test Bench Expectations/Requirements:
  - Input block header number #100,000
  - Read output Hash
- Main Verification Test Steps:
  1. Refer to blockchain.info and block #100,000 for valid input block header of 0x0100000050120119172a610421a6c3011dd330d9df07b63616c2cc1f1cd00200000000006657a9252aacd5c0b2940996ecff952228c3067cc38d4885efb5a4ac4247e9f337221b4d4c86041b002b5710
  2. The corresponding output header should be 0x000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1afd33e506

As shown in the screenshot in figure 26, the input to the block is the same as block 100,000 on the bitcoin blockchain. The 2 yellow boxes on the left show the header cut up into 2 512 bit chunks written in little endian. The second chunk is padded according to the SHA algorithm. The screenshot also shows that three main tasks in the middle. This because the (SHA-256)^2 algorithm for a block header takes 3 rounds of SHA. 2 for the input header which much be split up and another one for the second round of SHA. at the end of the three rounds the correct out hash is shown on the out_hash bus for this block. Both the input and output can be double checked via the website blockchain.info or any other blockchain tracking website.



```
sim:/tb_bitcoin_miner/MINER/HM_1/SHA/data @ 83714117 ps
15 : 32'hefb5a4ac 32'hc38d4885 32'h28c3067c 32'hecff9522
11 : 32'hb2940996 32'h2aacd5c0 32'h6657a925 32'h00000000
 7 : 32'h1cd00200 32'h16c2cc1f 32'hdf07b636 32'h1dd330d9
 3 : 32'h21a6c301 32'h172a6104 32'h50120119 32'h01000000

sim:/tb_bitcoin_miner/MINER/HM_1/SHA/data @ 86303715 ps
15 : 32'h00000280 32'h00000000 32'h00000000 32'h00000000
11 : 32'h00000000 32'h00000000 32'h00000000 32'h00000000
 7 : 32'h00000000 32'h00000000 32'h00000000 32'h80000000
 3 : 32'h0f2b5710 32'h4c86041b 32'h37221b4d 32'h4247e9f3

sim:/tb_bitcoin_miner/MINER/HM_1/SHA/out_hash @ 87936145 ps
32'h00000000 32'h27ba0300 32'h1c0b20aa 32'h78d4aaec 32'h3204b0d2 32'h1f3f6c34 32'h1ada8639 32'h06e533fd
```

Input Block # 100,000
0100000050120119172a610421a6c3011dd330d9df07b63616c2cc1f1cd00200000
000006657a9252aacd5c0b2940996ecff952228c3067cc38d4885efb5a4ac4247e9f
337221b4d4c86041b0f2b5710

Output Hash:
000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1afd33e506

Figure 26: **Hash Block Header via (SHA-256)^2 Algorithm**

USB Interface Receives an Interrupt Packet & Halts Computation

- Test Bench Expectations/Requirements:
  - Test vector containing interrupt packet
- Main Verification Test Steps:
  1. Emulates interrupt packet from test vector
  2. Checks status of calculation
  3. The purple circle in the figure below represents the data packet and the yellow circle is the transmitted ack packet. The hash done signal will keep asserting, signaling that the hashing algorithm is working. This signal stops once the interrupt signal is asserted, signifying a halt in computation. When the interrupt signal is pulsed high, the hashing module stops hashing the data.

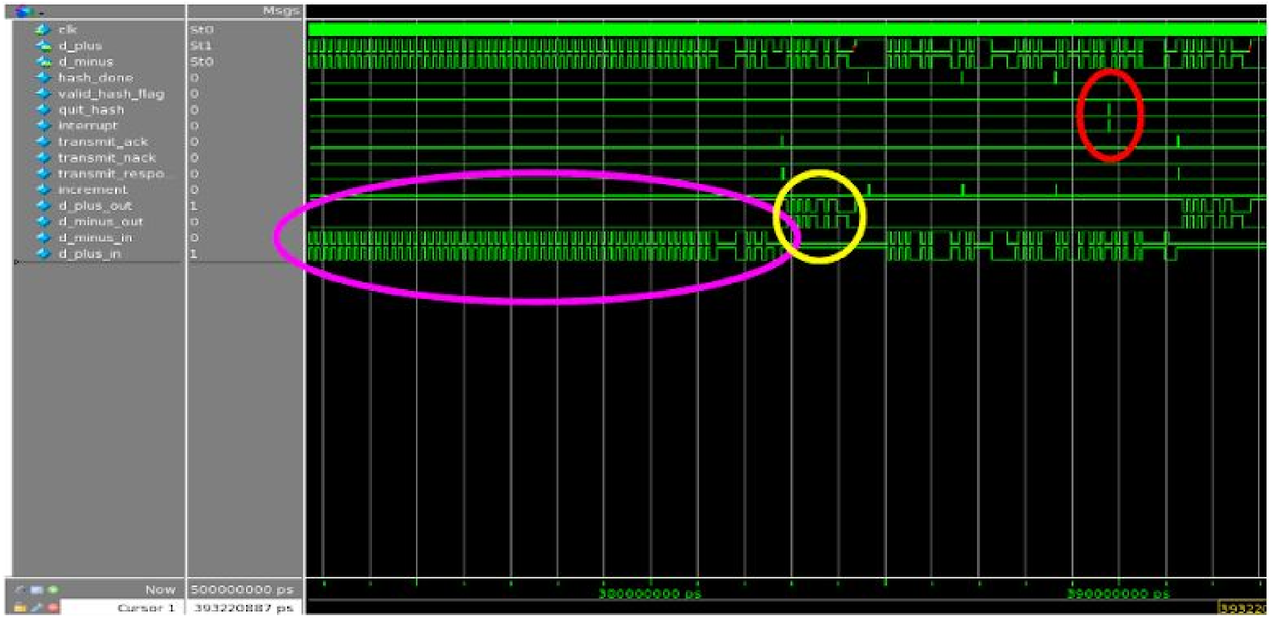
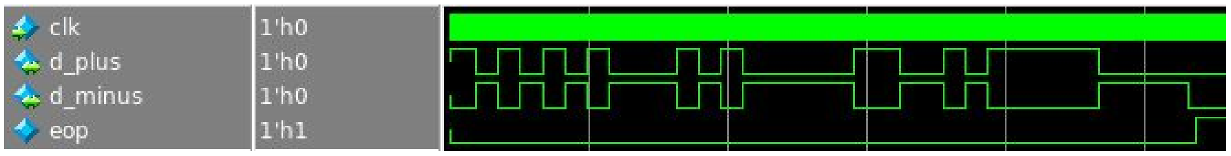


***Figure 27:* Interrupt halts hash calculation**

USB Interface Receives Block Packet via Bulk Transfer

- Test Bench Expectations/Requirements:
  - Valid hashed header sent on data line
  - Sample Crc5 bytes: 1011010000010101111101111, 100001110011101000111101
  - Sample Crc16 data: 11000011000000001000000001000000110000001111011101011110, 11010010110001001010001011100110100100010111000000111000
  - Sample crc data comes from "CYCLIC REDUNDANCY CHECKS IN USB"
  - Represent data from block # 100,000 on blockchain.info in the correct format: ensure appropriate endianness, order of bits, etc. to make sure that the input for the hashing module gets the data in the expected format

- Main Verification Test Steps:
  1. Send sample crc5 bytes
  2. Check if the correct crc was obtained
  3. Send sample crc16 bytes
  4. Check if the correct crc was obtained
  5. Send in consecutive blocks and verify crc via online crc calculator.
  6. Verify that crc check is asserting at the correct time.
  7. The token packet is represented by the initial red circle in timing waveform in figure 27. The transmitted ack packet is represented by the other red circle. The data in the receiving line between these two packets is the data packet containing the block header



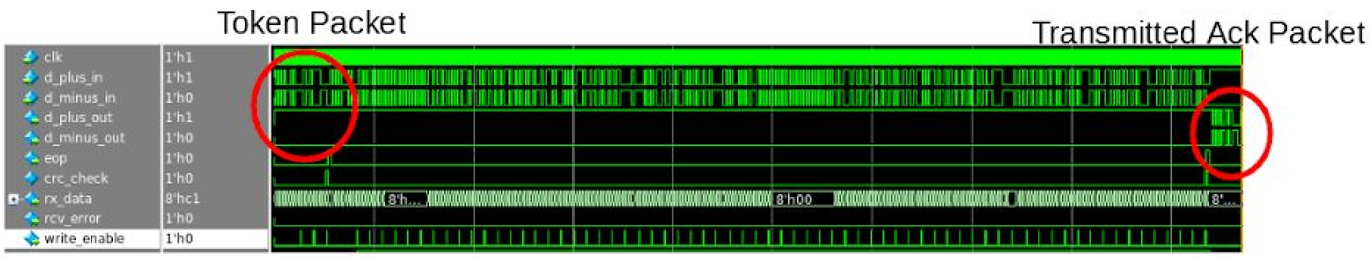


Figure 27: USB Receiving Block Packet via Bulk Transfer

Transmit Valid Hashed Packet via USB

- Test Bench Expectations/Requirements:
  - Valid hashed header sent on data line
  - Sample Crc5 bytes: 1011010000010101111101111, 100001110011101000111101

- Sample Crc16 data: 1100001100000000010000000010000001100000011110111010111110, 110100101100010010100010111001101001000101110000001110000
- Sample crc data comes from “CYCLIC REDUNDANCY CHECKS IN USB”
- Represent data from block # 100,000, #99,999, and #100,001 on blockchain.info in the correct format: ensure appropriate endianness, order of bits, etc. to make sure that the input for the hashing module gets the data in the expected format
- Connect Transmitter to USB Receiver to make make process of sending large amounts of data simpler.
- Data sent: 000000000000080b66c911bd5ba14a74260057311eaeb1982802f7010f1a9f090h9bcc8940, 000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1afd33e50610572b0f, 000000000002d01c1fccc21636b607dfd930d31d01c3a62104612a1719011250380388B2

- Main Verification Test Steps:
  1. Send in sample CRC data to verify that the transmitter is transmitting the correct checksum
  2. Sending in blocks and concurrently check if the transmitter is receiving the correct data using a fork.
  3. Test Bench verifies that the correct data is being transferred.
  4. See document at “mg138/ece337/Bitcoin_miner/docs/tb_USB_tx_top_level TestBench Results” for verification of the checking script.

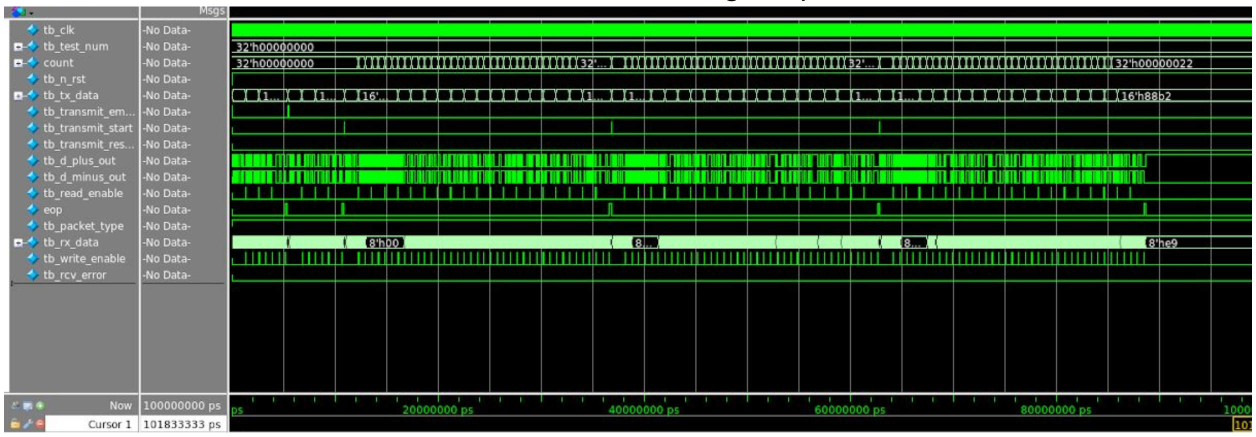


Figure 28: USB Transmitting Hashed Packet

6. Project Management

Per member list of major design component/module responsibilities:

1. Michael: Packet Decoder, Main Controller, Hashing Module

Michael primarily developed the hashing module and the main controller. In addition, he was able to help debug most aspects of the design when the final design top level file was being tested. He implemented the test benches for his assigned modules.

2. Chinar: USB Receiver, USB Transmitter

Chinar helped develop the USB receiver and USB transmitter modules, creating the initial diagrams and helping debug the code. She also created test benches for multiple blocks within different modules to ensure foundational functionality.

3. Yash: USB Receiver, USB Transmitter, Packet Decoder

Yash primarily developed the USB receiver and transmitter modules, creating the test benches associated with them. He was also able to debug most aspects of the design when testing the final design top level file.

4. Rahul: Hashing Module, Packet Decoder

Rahul worked on the hashing module with Michael. He created multiple helper functions which served to be very useful in testing and creating other modules. He also helped create Packet Decoder.

7. Appendix A

7.1 Verilog Code Structure and Test Benches

All verilog source code files and test benches can be found in:
mg138/ece337/Bitcoin_miner/source

7.1.1) Main Controller & Overall Top Level (Bitcoin Miner)

Top Level: bitcoin_miner.sv
Top Level Test Bench: tb_bitcoin_miner.sv

Other top level files that do not have sub modules

- main_controller.sv
- USB_transceiver_selector.sv
    - tb_USB_transceiver_selector.sv
- PD_hash_separation.sv
    - tb_PD_hash_separation.sv

7.1.2) Hashing Module

Top Level:HM_top_level.sv
Top Level Test Bench: tb_HM_top_level.sv

Component Files:

- HM_SHA_256.sv
- HM_bus_select.sv
- HM_check_hash.sv
- HM_controller.sv
- HM_hash_selection.sv
- HM_timer.sv

Other testbenches:

- tb_HM_SHA_256.sv

7.1.3) Packet Decoder

Top Level:	PD_top_level.sv
Top Level Test Bench:	tb_PD_top_level.sv

Component Files:

- PD_block_storage.sv
- PD_chunk_decoder.sv
- PD_controller.sv

- PD_hash_separation.sv
- PD_timer.sv
- PD_top_level.sv

Other testbenches:
- tb_PD_block_storage.sv
- tb_PD_chunk_decoder.sv
- tb_PD_hash_separation.sv

7.1.4) USB Receiver Files

Top Level:	USB_rx_top_level.sv
Top Level Test Bench:	tb_USB_rx_top_level.sv

Component Files:
- USB_sync_high.sv
- USB_sync_low.sv
- USB_eop_detect.sv
- USB_edge_detect.sv
- USB_decoder.sv
- USB_crc_16.sv
- USB_crc_5.sv
- USB_crc_rx.sv
- USB_rx_controller.sv
- USB_rx_counter.sv
- USB_timer_rx.sv
- USB_rx_sr.sv

Other test benches:
- tb_USB_rx_controller.sv
- tb_USB_timer_rx.sv
- tb_USB_crc_16.sv
- tb_USB_crc_5.sv
- tb_USB_decoder.sv

7.1.5) USB Transmitter Files

Top Level:	USB_tx_top_level.sv
Top Level Test Bench:	tb_USB_tx_top_level.sv

Component Files:
- USB_tx_controller.sv
- USB_tx_sr.sv
- USB_timer_tx.sv
- USB_crc_tx.sv
- USB_encoder.sv

Other Test Benches:
- tb_USB_encoder.sv
- tb_USB_timer_tx.sv

7.1.6 Other Helper Files:
- flex_counter.sv
- flex_counter_fix.sv
- flex_pts_sr.sv
- flex_stp_sr.sv
- Flip_endian.sv

7.2 Report Files
Reports for all top level modules are included in mg138/ece337/Bitcoin_miner/reports folder.

7.3 Datasheets
The only datasheets we used are referenced in our references section. These included USB standard packet descriptions and SHA-256 pseudocode and test vectors.

References

USB Made Simple. (2008). [Online] Available at: http://www.usbmadesimple.co.uk/index.html.

"Block #100000," *Blockchain*, 29-Dec-2010. [Online]. Available at: https://blockchain.info/block/000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1afd33e506

A. Isakov, *Online CRC Calculator*, 2015. [Online]. Available at: http://www.crccalc.com.

"CYCLIC REDUNDANCY CHECKS IN USB," *USB.org*. [Online]. Available at: http://www.usb.org/developers/docs/whitepapers/crcdes.pdf.

"Cryptographic Standards and Guidelines," *NIST*, 29-Dec-2016. [Online]. Available: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values.

"Hash Functions," *NIST*, 04-Jan-2017. [Online]. Available: https://csrc.nist.gov/Projects/Hash-Functions.