Bit by Bit: An FPGA-assisted Bitcoin Miner

Sabiyyah Ali
6.2050
MIT
Cambridge, USA
sabi@mit.edu

Joshua Herrera
6.2050
MIT
Cambridge, USA
jih@mit.edu

***Abstract*—Mining cryptocurrency is particularly well suited to hardware tasks because it requires quickly computing many hash functions which can be parallelized. We present a system for mining Bitcoin accelerated by an FPGA. We constructed a pipelined and optimized implementation of the SHA256d algorithm on a Nexys A7 FPGA Development Module. The module communicates with a computer over a USB-RS232 connection, sending mined nonce values and receiving the latest blockchain block headers. We simulate a hypothetical blockchain by providing the FPGA with block headers with a known correct nonce value, and compare the response from the FPGA with the known solution. We are able to compute approximately 43.75 Million hashes per second, which allows us to search through 2^{32} nonce values in just over 98 seconds! All of the source code for our project can be found at [8].**

***Index Terms*—Bitcoin, FPGA, RS232, SHA256**

I. Introduction

Hardware assisted mining can speed up the process of mining a Bitcoin block considerably. We present an approach to mining hardware acceleration using a Nexys A7 FPGA Development Module clocked at 100 MHz. Every time the blockchain grows, the FPGA must parse and hash the latest block header using the SHA256d algorithm [1]. Our system implements a SHA256d hashing engine with optimizations that take advantage of invariants in the Bitcoin Protocol, allowing us to compute 16 hashes per FPGA clock cycle. After mining a block, we need to transmit the 32 bit nonce value that was used to mine the block to an external computer so that it can submit the mined block to a Bitcoin node. We created a bidirectional communication channel with a local computer over USB-UART using the USB-RS232 functionality built into the FPGA development module. Universal asynchronous Receiver-Transmitter (UART) is an open ended protocol which specifies how data is sent over two wires. The development module includes a chip which allows us to use this protocol over USB, which we use to send data over the cable that powers our FPGA.

Section II explains the process of Bitcoin mining and the necessary components of the Bitcoin protocol used in the report. Section III explains how the hashing engine works in detail. Section IV provides insight into the choices we made for our communication stack.

II. The Bitcoin Protocol

The Bitcoin blockchain is a directed acyclic graph where each node is called a ”block”. This graph can grow by adding ”mined” blocks. At any given point in time, only blocks that exist on the longest path in the graph are considered valid, mined blocks. The protocol is designed so that obtaining a mined block is computationally difficult. It only makes sense to mine blocks that are added to the end of the longest chain in the blockchain.

A given block consists of a header and a body. In order for a block to be considered mined, its header's hash value must be less than some known target value. Bitcoin uses the SHA256d hashing algorithm, which does not have a computationally efficient inverse function. Thus, miners must change parts of the block header and recompute its hash many times until they get lucky and find a block header with a small enough hash value.

The body of a block contains transactional information about how bitcoins are exchanged, while the header of the block contains the following fields:

- **Version**: This describes the protocol version being used. Valid mined blocks cannot use an old version.
- **Previous Block Hash**: This determined where on the blockchain the block is being added. Normally, this value is the hash of the latest known block on the longest chain in the blockchain.
- **Merkle Root**: This can be thought of as an iterated hash of the contents of the block. This is what links a given block header to the body. A full explanation of how to compute the Merkle Root from the block body can be found on Wikipedia [2].
- **Time Stamp**: The time the block was mined
- **Target**: Specifies the difficulty of mining the block. The hash of the block header must be smaller than this value in order for the block to be considered ”mined”. This is recalculated every 2016 blocks on the blockchain.
- **Nonce**: This is an arbitrary value that is used to add entropy to the block header. Changing the nonce will change the hash of the block header, which allows a miner to cycle through different hashes.

In order to change the block header, a miner can change the Merkle Root (changing the block body changes the Merkle Root), time stamp (this shouldn't be off by more than 2 hours from the actual time the block was mined), and nonce fields. Together, these provide about 40 bytes of entropy.

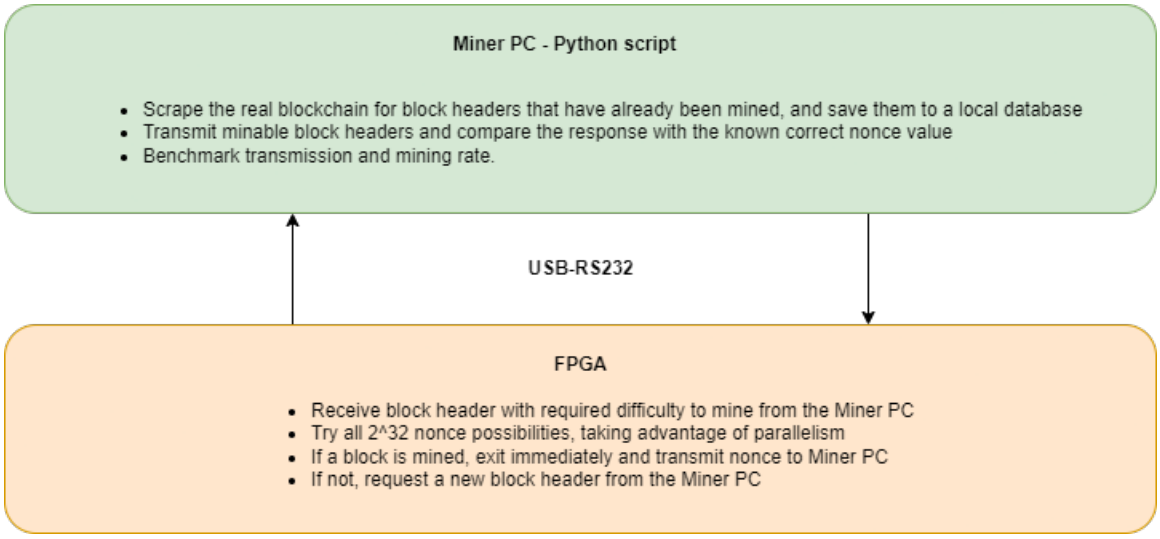

Fig. 1. High Level system diagram

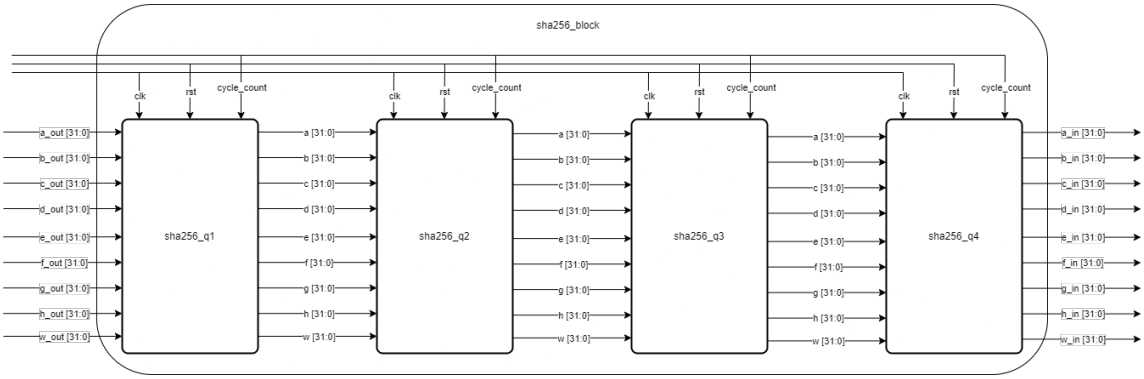

Fig. 2. SHA256 hashing engine block diagram.

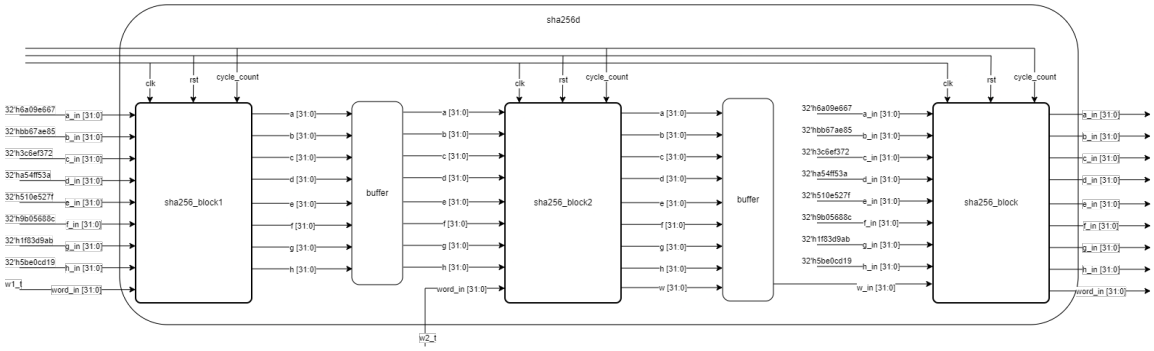

Fig. 3. SHA256 hashing engine block diagram.

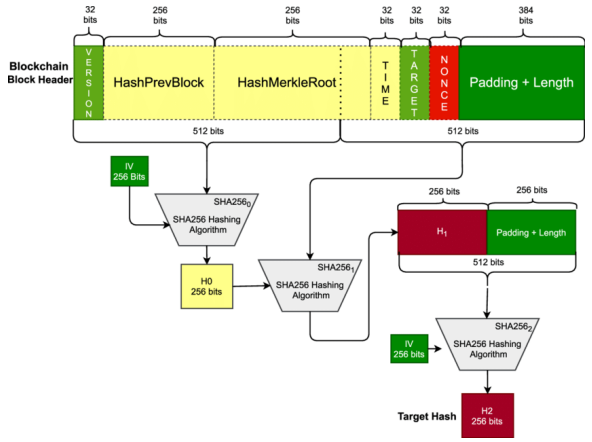

Fig. 4. SHA256d algorithms block diagram. Adopted from [3]

III. Hashing Engine

A. The SHA256d algorithm

The SHA256 hash function is specified by [1]. The input can only hash a message with a size divisible by 512 bits. A block header is 640 bits, so 384 bits of padding must be added. Each 512 bit block is processed by a block module, which is shown in Figure 3. This happens serially, such that the output of one block module is used as an input by the next block module; meanwhile, the first block module uses known constants as its input. This module corresponds to the trapezoidal gray blocks in Figure 4. The Bitcoin protocol uses SHA256d, which calls for hashing the block header with SHA256, and then hashing the result once more (a "double-hash"). Since the output of SHA256 is 256 bits wide, the second hashing step only requires a single 512 bit block to be processed. This is outlined in Figure 4. Below we go over the SHA256 algorithm itself and the corresponding modules for its hardware implementation.

The first block never changes so we can compute its message schedule and intermediate hash values $a...h$ and store them in a BRAM buffer. When we change the nonce value, we only need to compute the hash values for the second block using the stored values from the first block. The outline for this process is shown in Figure 2.

B. The sha256_q module

The sha256_q module performs the innermost loop of the sha256 algorithm computation, including both the message schedule computation and the shuffling procedure. The original SHA256 specification [1] describes a procedure where, for each block, the algorithm loops 64 times. Our implementation splits this 64-loop into four 16-loops as shown in the pseudocode from Algorithm 1. Each sha256_q module computes an intermediate 16-loop of the algorithm, and four are chained together to construct a single block module. Each sha256_q module computes both the message schedule and the intermediate hash values and passes them to the next module down the line. The message schedule computation

Algorithm 1: SHA256 Pseudocode

Data:
B_i = The ith input 512 bit block
N = The number of input blocks
a, b, c, d, e, f, g, h are initialized to known constants
W_i **for** $i \leftarrow 1$ **to** 16 is initialized to the first 16 words (32 bit) from the first 512 bit block
H_i **for** $i \leftarrow 1$ **to** 8 is initialized to known constants
F_1, F_2, F_3 are known functions composed of simple bitwise arithmetic operations
begin
  for $i \leftarrow 1$ **to** N **do**
    for $j \leftarrow 1$ **to** 4 **do**
      `/* Compute the next 16 words of the message schedule using the past 16 */`
      for $k \leftarrow 1$ **to** 16 **do**
        $W_k \leftarrow F_1(W_{16j+k-16}...W_{16j+k})$
      `/* Shuffle variables around */`
      for $k \leftarrow 1$ **to** 16 **do**
        $T_1 \leftarrow F_2(a, b, c, d, e, f, g, h, W_k)$
        $T_2 \leftarrow F_3(a, b, c, d, e, f, g, h, W_k)$
        $a \leftarrow T_1 + T_2$
        $b \leftarrow a$
        $c \leftarrow b$
        $d \leftarrow c$
        $e \leftarrow d + T_1$
        $f \leftarrow e$
        $g \leftarrow f$
        $h \leftarrow g$
    $H_1 \leftarrow H_1 + a$
    $H_2 \leftarrow H_1 + b$
    $H_3 \leftarrow H_1 + c$
    $H_4 \leftarrow H_1 + d$
    $H_5 \leftarrow H_1 + e$
    $H_6 \leftarrow H_1 + f$
    $H_7 \leftarrow H_1 + g$
    $H_8 \leftarrow H_1 + h$
  return $H_1||H_2||H_3||H_4||H_5||H_6||H_7||H_8$

(function F_1 from the pseudocode) requires remembering the last 16th, 15th, 7th, and 2nd message schedule words previously computed, which we keep in a distributed RAM pipeline. All of the operations happen in a single clock cycle, so each loop iteration completes in sixteen cycles.

C. The sha256_block module

The sha256_block module puts 4 sha256_q modules together to compute one 64 cycle iteration of the algorithm for a single input block. Since the four sha256_q modules are pipelined, it can compute a hash once every 16 clock cycles. An additional two cycles of latency are added to each sha256_q module in order to break up combinational logic between cycles. Finally, an additional cycle of latency

is added to the sha256_block module. This result is a total of $(16 + 2) * 4 + 1 = 73$ clock cycles of latency. Our FPGA is clocked at 100 MHz. For our purposes, latency is not a concern since any delay incurred is dwarfed by the amount of time it takes to search through all possible nonce values.

D. The sha256d module

Recall that the Bitcoin protocol calls for a double-hash of the block header. This function, which we implement in the sha256d module, is described by the block diagram in Figure 4. Figure 2 reveals the steps we make to Bitcoin-specific optimizations to maintain a tight 16-step pipeline. The 640 bit block header is padded into two 512 bit blocks, but the first block never changes (the nonce value which we change is found in the second block). We can precompute the intermediate hash values for the first block and store them in a buffer to use for all subsequent hash calculations. If we recomputed these values every time, it would take 32 clock cycles to compute the first block's hash values, which would halve the throughput of our system.

The second block changes as we change the block header nonce value. This means we still have to compute the intermediate hash values for the second block with every new nonce. The first block module therefore expects a 32 bit word from the second block every cycle, so that the entire block is transmitted over 16 cycles. The output of the first SHA256 hash is kept in a buffer so that it can be fed into a subsequent sha256_block module which is needed to compute the second hash. Figure 3 outlines the entire process. Apart from the intermediate hash values $a, b, ..., h$, each block module takes in a 32 bit "word_in" input as well as a cycle_count input. The word_in input is for the actual data being hashed, and is passed in one word per clock cycle over the course of 16 cycles. The cycle_count signal is a counter that increments from 0...15 which is used to keep the pipeline synchronized across cycles.

IV. UART Communication Stack

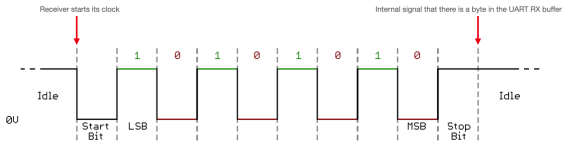

Fig. 5. UART Timing Diagram. Adopted from [4]

A. Overview

We can transceive block header and nonce values one byte at a time using an RS232 interface. Both the receiver and transmitter must agree upon components of the transmission like the clock frequency, data width, presence of a parity bit, and the number of stop bits. We are using an 8-bit data width without a parity bit or flow control. We chose a clock rate of 115200 Hz, which is one of many standard values for UART baudrates. Figure 5 displays a timing diagram of a single byte of information being sent over a wire in one direction.

B. Application layer protocol

We transmit information one byte at a time using the UART protocol [5]. All transmissions are prefixed with a "command byte", which both parties agree on ahead of time. The receiver will know the size and content of the information transmitted depending on the value of this first byte. For example, for a block header transmission, the FPGA is expecting exactly 80 bytes. A table of commands we use is listed in Table I. The incoming block header is kept in a BRAM module so that it can be reused by other modules down the pipeline.

C. Denoising

UART protocol is prone to noise on the RX line, since any drop to logic 0 indicates a start bit and therefore the presence of a transmission. In order to make our system resilient to single-cycle fluctuations in the RX signal, we maintain a history of 8 RX values sampled uniformly over the last 64 clock cycles. We then take the denoised signal to be the majority bit of the samples. This incurs a small latency penalty of $\sim$32 clock cycles, but also grants immunity to any fluctuations in the RX signal that might be caused by environmental interference or a bad quality USB cable.

D. Clock Domain Crossing

The FPGA development board we use provides a base clock signal at 100 MHz which we use across all of our modules. The UART baudrate we chose is 115200Hz, which means that there are 868 FPGA clock cycles in one period of the UART clock. In order to send and receive UART signals, we maintain a counter which emulates a secondary clock by counting from 0 to 868. Though the frequency is not exactly 115200Hz, it is well within 1% of the correct clock speed, which is actually sufficient for a successful communication channel to be established.

It is crucial that the UART module, shown in in Figure 6, resets this counter every time an incoming byte is detected. If the clock was never reset, then the the receiver and sender clocks would desynchronize, drifting apart, and potentially cause the receiver to either miss a bit or double count a bit in the transmission. If even a single bit is changed in the block header, the output hash will be completely different, defeating the point of the system.

E. Input and Output buffers

In order to make the system more robust, we place 1-byte input and output buffers for data bytes sent to and from the UART module. This allows us to run the communication modules alongside any blocking computation that may inhibit the communication module from consuming incoming data as soon as it is received. Both the send and receive modules are wrapped in a common UART module which includes the input and output buffers while also using the AXI [6] protocol to manage communication with other system modules.

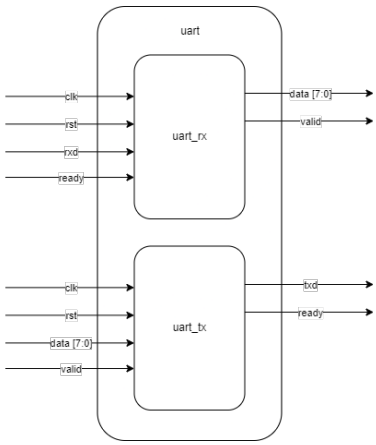

Fig. 6. UART Module Block Diagram

TABLE I
UART COMMAND PROTOCOL

Command	Description	Recipient	Subsequent Data Size
8'b00000000	Requests latest block header	Computer	0 Bytes
8'b00000001	Provides latest block header	FPGA	80 Bytes
8'b00000010	Provides successful nonce	Computer	4 Bytes
8'b00000011	Request nonce retransmission	FPGA	0 Bytes

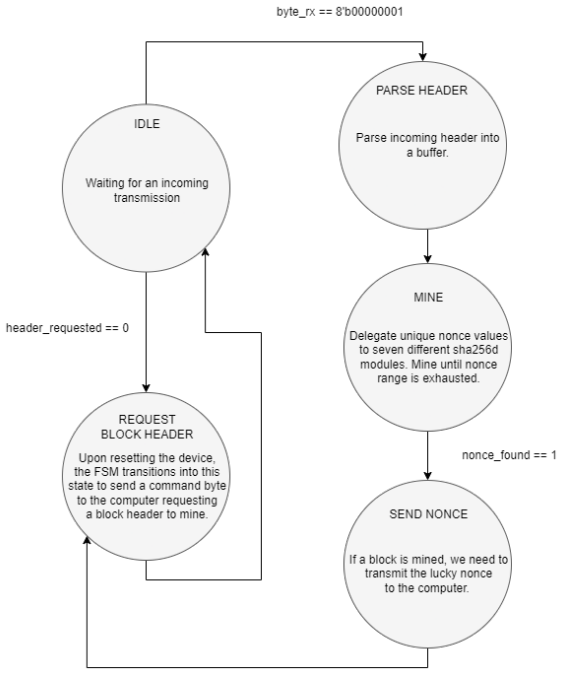

Fig. 7. Top Level State Machine

V. TOP LEVEL

The top level module for our system maintains a state machine which glues together the communication stack with the hashing engine. When idling, it will request a new block header. Upon receiving the block header, it will preprocess the header, computing and storing the intermediate hash values for the first block. It will also add the 384 bit padding to the stored header so that it can be consumed directly by the hashing engine. It will also extract the target from the block header so that it can be easily compared to the output hash of the hashing modules. Once the input is preprocessed, the top level FSM enters the MINE state, where seven duplicate hashing modules are assigned unique nonce values across the range of all nonces. We only managed to synthesize seven duplicate modules before the FPGA ran out of logic resources (the 8th module required $\sim$1000 more than the remaining 9500 LUTs). If a working nonces is found, the state machine transitions to the SEND NONCE state and sends the golden nonce to the computer via our UART module. The computer is expected to compute a unique block header every time a new one is requested, which can by done by changing the Merkle Root or the timestamp field.

A. Optimizations and Future Work

We found that the bottleneck in our design was LUT utilization, which we exhausted after duplicating the hashing modules in the hashing engine. In particular, slice logic took up the largest percentage of resources ($\sim$98%) according to the post-placement synthesis report. In order to reduce slice LUT utilization, we removed unnecessary pipeline registers where possible and used BRAM to store large constants (the SHA256 algorithm has 1024 bits of constants) and intermediate algorithm state, such as the message schedule. Unfortunately, some of our BRAM modules were inferred as distributed RAM instead. Future work might consider using a dedicated BRAM IP block to guarantee BRAM usage.

As mentioned in Section III, we piplined our hashing engine so that it has a total throughput of 1 hash per 16 clock cycles which equals 6.25 MHashes/s. We duplicated our module seven times, for a combined hash rate of 43.75 MHashes/s. To explore all 2^{32} nonce values, this would take, theoretically, 98.171 seconds.

We also took advantage of invariants in the Bitcoin protocol to reduce the computation required for our system. Recall, for example, that the first block of the block header doesn't contain the nonce value and therefore does not change. By preprocessing it with the sha256_block module, we can double our throughput from $\frac{1}{32}$ hashes per cycle to $\frac{1}{16}$ hashes per cycle.

VI. Evaluation

Unfortunately, testing our system on the real blockchain would be infeasible. The profitability of a mining node is directly related to its hashrate. Bitcoin's hash function has allowed ASICs to emerge and dominate the mining space. Modern ASICS are thousands of times faster than anything we might hope to develop on our FPGA, pushing the mining difficulty of the network higher and higher. At the time this report was written, the mining difficulty of the latest block was 34,244,331,613,176. At a hash rate of 43 MHashes/s, it would take, in expectation, 38,909,599,034 days to mine a single block by using a single FPGA. Clearly, we need another way to benchmark our system.

In order to evaluate our system, we emulated a miner node by scraping block headers from the real blockchain which are known to have a valid hash below their target value. We also generated dummy block headers with arbitrarily high target values so that we could test that the FPGA could actually find the correct nonce. Once we confirmed that our mining accelerator was working correctly, we timed the amount of time it took to explore all nonce values. We used the pyserial python API to read and write to the FPGA [7]. We sent and received information using the commands in Table I.

Experimentally, the end-to-end delay for exploring all 2^32 nonce values was 98.192 seconds, which is 0.02 seconds more than what we calculated theoretically. We attribute the extra delay to the time it takes to communicate over RS232, as well as any delay incurred by the Python script and the operating system (Ubuntu in our case). We found that communication over RS232 was very reliable. We performed a stress test that sent thousands of different block headers with known correct nonce values (which means tens of thousands of bytes of data were being transmitted to the FPGA), and the FPGA returned the correct value 100% of the time.

VII. Team Contributions

As a whole, we felt that we both contributed to this project equally. Joshua focused on reimplementing the SHA256 algorithm and adding the Bitcoin-specific optimizations to our FPGA. Sabbiyyah focused on developing and testing the communications stack and developed most of the top level code as well. Every part of the project and paper saw contributions from both persons in one way or another. The portions that we worked on correspond with the portions that we wrote for the paper.

References

[1] FIPS PUB 180-4. "Secure Hash Standard (SHS)," https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
[2] https://en.wikipedia.org/wiki/Merkle_tree
[3] Naik Rahul, "Optimising the SHA256 Hashing Algorithm for Faster and More Efficient Bitcoin Mining 1" 2013
[4] Adams, V. Hunter. https://vanhunteradams.com/Protocols/UART/UART.html
[5] https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
[6] https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface
[7] https://pyserial.readthedocs.io/en/latest/index.html
[8] https://github.mit.edu/jih/6.2050-sabi-jih-final-project-submission