

6.2050 Final Report

Bigbit532: A Modern Hardware-Accelerated Bitcoin Miner

Parker Spann
Department of EECS
Massachusetts Institute of Technology
pspann@mit.edu

Sean Huckleberry
Department of EECS
Massachusetts Institute of Technology
seanhuck@mit.edu

Abstract—Mining bitcoin involves computing up to 10^{23} hashes using the 256-bit Secure Hash Algorithm (SHA-256) to produce a result that satisfies a given target difficulty. The highly repetitive and resource-intensive nature of this task makes it well-suited to hardware acceleration through an FPGA. Here, we present an implementation of a bitcoin miner to break SHA-256. Given a correctly-formatted bitcoin block and a target, Bigbit532 computes hashes rapidly in parallel for maximum throughput. While initially tuned to the Xilinx Spartan-7 XC7S50-CSGA324 FPGA, Bigbit532’s parametrized design makes it highly scalable and easily transferable to other, more resourceful boards.

Index Terms—FPGA, bitcoin mining, SHA-256d, proof-of-work, hardware acceleration, cryptography

I. DESIGN OVERVIEW

Bigbit532’s design is intended to be simple and efficient, modular and parametrizable. Given the emphasis on performance, it is vital that the system avoids unnecessary overhead and enables effective fine-tuning. The high-level design is presented in Figure 1, which can be parsed as follows:

- 1) **Generating assignment:** *PC Host* is responsible for transmitting the block assignment to the miner over UART, given by $(block_header, target)$. Users may select a block to mine by specifying an 80-byte hex header when running the script.
- 2) **Receiving:** *Stacker* concatenates the bits as they arrive over the wire, handling endianness and formatting. The *Receiver* forwards the header $block_header$ to the *Coordinator* and $target$ to the comparator, and introduces a single-cycle high din_valid to begin the mining process.
- 3) **Producing candidate hashes:** *Coordinator* receives the assignment and operates as an FSM that coordinates the hash computation. It tries new hashes by requesting nonce values from the *Nonce Generator* and passing them into the $sha256d$ double-hash module. The first block is unique in that it computes an important *midstate*, which is reused for subsequent hash calculations. Parallel $sha256d$ modules enable the *Coordinator* to request and hash multiple nonces in parallel.
- 4) **Testing hashes:** *Comparator* checks hashes for validity.
- 5) **Returning valid hash:** When a valid hash is found, the *Coordinator* initiates a UART transmission back to

the *PC Host*, using an unstacker to send the nonce information for the valid hash byte-wise.

The technical challenge in mining bitcoin goes beyond building a functional prototype. To be worthwhile in the competitive mining space, we must maximize throughput while respecting the resource and timing constraints of the FPGA. Following a discussion of Bigbit532’s key modules, we discuss performance engineering in depth later in the report.

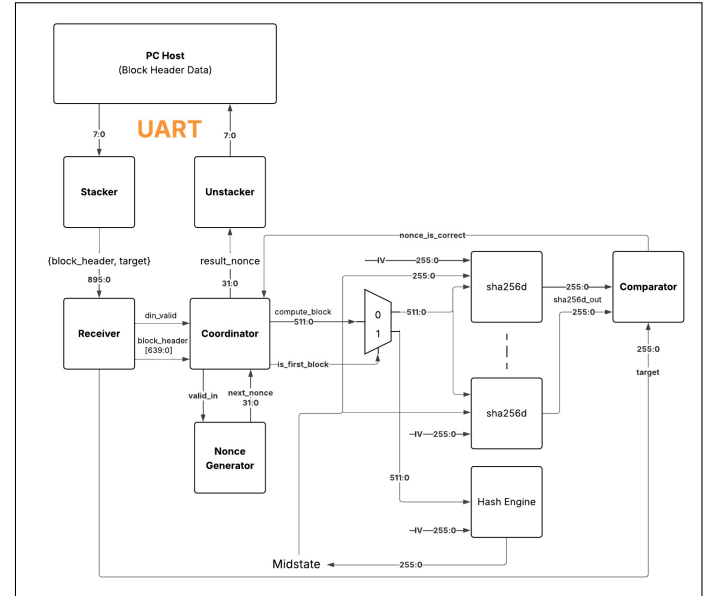


Fig. 1. System Block Diagram

II. THE BITCOIN PROTOCOL

Bitcoin is a consensus-based cryptocurrency that stores historical transactions on a publicly-available *blockchain*. To modify the blockchain, miners must find valid “proof-of-work” in the form of a SHA-256d hash lesser than a target value. Miners are rewarded monetarily for their contributions, typically through communal mining pools[1].

A. Standard Block Header

A bitcoin miner operates on a single block at a time, which is passed to the hardware as a block header. Miners modulate values within a given block header to find a set of parameters that, when passed through SHA-256d, produce a 256-bit output smaller than a given *target*.

As depicted in Figure 2, a standard block header takes the following form:

- **Version:** Indicates the protocol used to mine this block.
- **Previous Hash:** Hash of prior block on the blockchain.
- **Merkle Root:** Hash of all transactions.
- **Time:** Block timestamp.
- **Bits:** Encoded difficulty target.
- **Nonce:** Miner's primary varying value.
- **Padding:** Extra bytes for SHA-256 formatting.

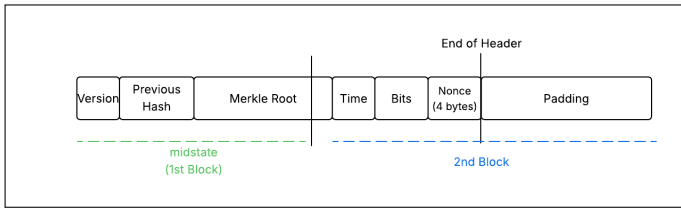


Fig. 2. Standard Block Header

B. Search

Miners canonically begin their search for a valid hash by varying the *nonce* (32 bits). If they are unsuccessful in producing a valid hash, they proceed to vary *extra_nonce* (96 bits), which indirectly changes the block by modifying the merkle root, and *time* (6-10 bits). See Algorithm 1 for the search pseudocode. In total, miners typically have around 138 bits of flexibility per block to produce a valid hash. In our implementation, we vary only *nonce*, which gives us enough hash capacity to meaningfully accelerate and evaluate performance without unnecessary complexity.

Algorithm 1 Mining Loop Structure

```

1: for  $t \leftarrow 0$  to TIME_LIMIT do
2:   update_time( $t$ )
3:   for  $extra\_nonce \leftarrow 0$  to  $2^{96} - 1$  do
4:     update_coinbase( $extra\_nonce$ )
5:     recompute_merkle_root()
6:     for  $nonce \leftarrow 0$  to  $2^{32} - 1$  do
7:       header  $\leftarrow$  assemble_header(time,  $extra\_nonce$ ,
        nonce)
8:       hash  $\leftarrow$  SHA256d(header)
9:       if hash < target then
10:        return (nonce,  $extra\_nonce$ , time)
11:      end if
12:    end for
13:  end for
14: end for

```

C. Secure Hash Algorithm (SHA)

SHA is at the core of the bitcoin protocol. It is commonly written as "SHA-256d," referring to the algorithm's double hashing of its inputs and producing a 256-bit output. Like other hash functions, SHA-256d is not reversible, so miners must brute-force inputs rather than computing a mathematical inverse. In the forward direction, as well, it is designed to prevent any simplifying mathematical reduction. Based on the published SHA standards, we present the pseudocode for a single SHA-256 compression in Algorithm 2, which can be applied twice to produce SHA-256d[2].

Algorithm 2 SHA256 (Single SHA-256 Compression)

```

1: function SHA256(message)
2:    $m \leftarrow \text{PAD}(\text{message})$ 
3:    $(a, b, c, d, e, f, g, h) \leftarrow H_0$ 
4:   for each 512-bit block  $C$  in  $m$  do
        $\triangleright$  Prepare message schedule
5:     for  $t = 0$  to 15 do
6:        $W_t \leftarrow C[t]$ 
7:     end for
8:     for  $t = 16$  to 63 do
9:        $W_t \leftarrow \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$ 
10:    end for
        $\triangleright$  Initialize working variables
11:     $(A, B, C, D, E, F, G, H) \leftarrow (a, b, c, d, e, f, g, h)$ 
        $\triangleright$  Main compression loop
12:    for  $t = 0$  to 63 do
13:       $T_1 \leftarrow H + \Sigma_1(E) + \text{Ch}(E, F, G) + K_t + W_t$ 
14:       $T_2 \leftarrow \Sigma_0(A) + \text{Maj}(A, B, C)$ 
15:       $H \leftarrow G$ 
16:       $G \leftarrow F$ 
17:       $F \leftarrow E$ 
18:       $E \leftarrow D + T_1$ 
19:       $D \leftarrow C$ 
20:       $C \leftarrow B$ 
21:       $B \leftarrow A$ 
22:       $A \leftarrow T_1 + T_2$ 
23:    end for
        $\triangleright$  Update hash state
24:     $a \leftarrow a + A$ 
25:     $b \leftarrow b + B$ 
26:     $c \leftarrow c + C$ 
27:     $d \leftarrow d + D$ 
28:     $e \leftarrow e + E$ 
29:     $f \leftarrow f + F$ 
30:     $g \leftarrow g + G$ 
31:     $h \leftarrow h + H$ 
32:  end for
33:  return concat( $a, b, c, d, e, f, g, h$ )
34: end function

```

III. TOP LEVEL

Bigbit532's *Top Level* module contains the UART RX and TX with corresponding stacker and unstackers, a single hash

engine for computing and storing the midstate, 10 parallel SHA-256d hash blocks, the coordinator, target comparator, and seven-segment display. In the sections that follow, critical modules are discussed in depth.

IV. UART DATA TRANSMISSION

A. RX

Block header and target value assignments are communicated from the *PC Host* and parsed by the on-FPGA *Receiver* module via UART. The 896-bit assignment is transmitted bitwise in larger byte-sized packets in little-endian order, as is the standard. A stacker is used to reconstruct the assignment.

B. TX

Transmission from the FPGA is also performed via UART, and is triggered by the *nonce_is_correct* logic that signals high only after the *Comparator* tests a valid hash. The 32-bit nonce is communicated back to the *PC Host*.

C. Receive Time

Using a baud rate of 115,200 for UART and the FPGA board's clock frequency of 100 MHz, it takes about 869 FPGA clock cycles to transmit a single bit over UART. Given the 896 bits of the block header and target being received along with start and stop bits for 112 bytes, the system sends over a total of 1120 bits. This leads to 973,280 FPGA clock cycles total needed to receive the block header and target, a significant delay which was considered when interfacing with the FPGA.

V. COORDINATOR

The *Coordinator* module (Figure 3) manages the control flow of the central hash computation loop, and is defined by a five-state FSM. By varying the nonce, it feeds new blocks into the hash engine until some input satisfies the given target.

- 1) **Rest:** The *Coordinator* starts at *Rest*, moving into *Prepare Hash* if it receives a valid *block_header* and *target* over UART, represented by the *din_valid* signal.
- 2) **Prepare Hash:** The *Coordinator* requests a nonce from *Nonce Generator*, and enters a *waiting* state while the hash engine and comparator complete computations. In the parallelized implementation, the *Coordinator* requests *NUM_PARALLEL_LINES* nonces, and sends one block per engine to process simultaneously.
- 3) **Waiting:** An idle state while the *Coordinator* waits for the output of the hash engines.
- 4) **Store Midstate:** For the very first hash output, we compute the hash of the upper 512 bits of the block header and store it as the midstate. Any subsequent hashes are computed as a function of this midstate and a nonce-varied version of the lower bits of the header. Any block hash that doesn't satisfy the comparator will return the *Coordinator* to the *Prepare Hash* state.
- 5) **Transmitting:** On the other hand, if the output is a valid proof-of-work, the *Coordinator* moves to the *Transmitting* stage, where it sends the corresponding nonce back to the PC host. Then, it re-enters the resting state until the next header is received.

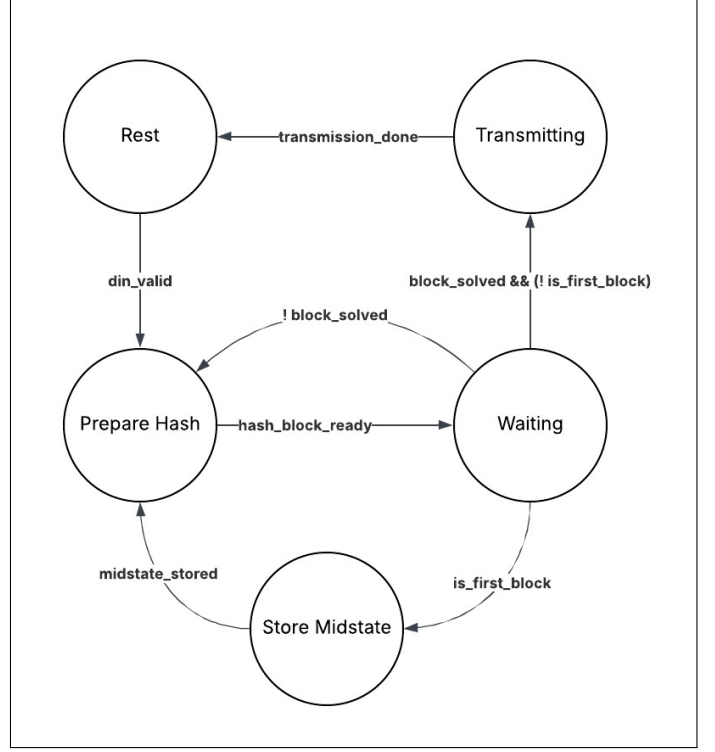


Fig. 3. Coordinator FSM

VI. HASH ENGINE AND SHA256D

The hash engine (Figure 4) is a composable module that directly implements the SHA256 pseudocode from Algorithm 2. It takes as input a *hash_block* and *state*, the latter of which is either the *midstate*, a result of SHA-256d hashing the first block for a given header, or the *initialization_vector* (IV), a standard byte sequence provided by the bitcoin protocol. The main computation involves a 64-round series of bitwise rotations, XORs, and additions called a *compression round*.

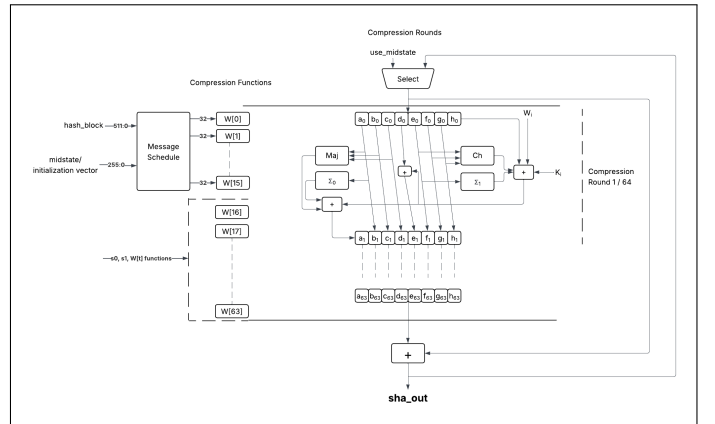


Fig. 4. SHA256 Hash Engine

The hash engine is wrapped in a higher-level *sha256d* module that performs the end-to-end double hash required by

the bitcoin protocol (Figure 5). This is exposed directly to the *Coordinator*, and is replicated when parallelizing the system.

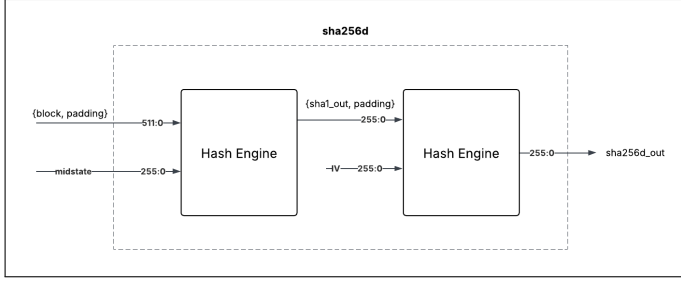


Fig. 5. sha256d Module

VII. SEVEN-SEGMENT DISPLAY

The board’s seven-segment display is programmed to display the nonce value (in hex) that the miner is currently computing, circumventing the UART bottleneck to provide a real-time visual progress-monitoring mechanism. The counter is only updated around every 1 million nonces, or 0x0100_0000. Thus the nonce is not precise beyond 24-bit resolution but provides a general sense of the system’s progress.

Two modules drive the seven-segment display: the *bto7s* hexadecimal-to-segment decoder and a digit-scanning controller. The scanner rapidly cycles through all eight digits, enabling only one digit at a time while feeding its corresponding nibble into the *bto7s* module. Since this multiplexing occurs at a high refresh rate, all digits appear continuously illuminated, and the display reliably shows the full 32-bit nonce value without missing updates.

VIII. PERFORMANCE ENGINEERING AND SCALING

A. Parallelization

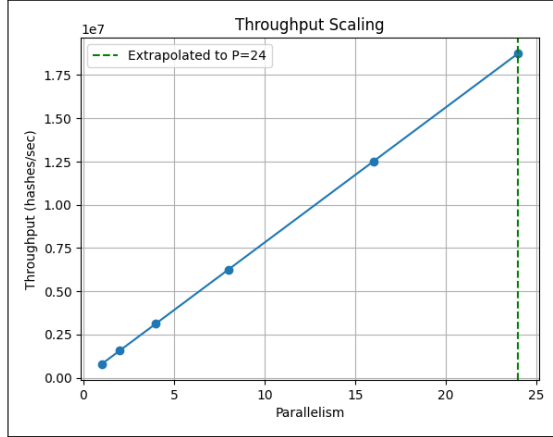


Fig. 6. Throughput Scaling w.r.t Number of Parallel Hash Engines

Our primary mechanism for boosting miner throughput was to perform the SHA-256d computations with multiple nonce values in parallel. To this end, we parametrize our *Top Level* with *NUM_PARALLEL_LINES*, and generate the corresponding number of parallel engines. The *Coordinator*

requests multiple nonce values from the *Nonce Generator* at a time, and each is passed into its own *sha256d* instance. Further downstream, each hash result is passed to the comparator to determine if any of the nonces are valid.

We optimized this value through empirical testing, landing on 10 parallel lines. Figure 6 demonstrates the scaling factor of 781,250 hashes/sec per additional parallel *sha256d* engine.

B. Pipelining

Canonically, the second main way to find a valid nonce quicker is by improving the throughput of the system through deeply pipelining within the *sha256d* module, and the two *Hash Engine* instances that it instantiates. We present pipelining optimizations from highest- to lowest-level:

1) *sha256d*: Recall that *sha256d* consists of 2 sequential *Hash Engines*. The final system greedily feeds new nonces into the *sha256d* as soon as the first *Hash Engine* finishes computing. This doubles the miner’s throughput by keeping both engines busy, instead of letting each remain idle for 128 cycles (the propagation delay of a single *Hash Engine*) while the other computes. Now in steady-state, the system will output the *sha256d* hash result of a new block header every 128 cycles compared to every 256.

2) *Hash Engine*: Within each *Hash Engine*, 64 compression rounds are required to compute the engine’s output. During development, we considered dividing these rounds into either two sequential *hash_engine_32* or four sequential *hash_engine_16* modules, enabling us to feed in a new block every 32 or 16 cycles, respectively. However, following discussions with course staff, we discerned that implementing these schema would incur identical cost to simply dedicating these resources to increasing the number of parallel *sha256d* blocks, while introducing routing and code complexity. As a result, given the existing parametrization of our *Top Level* module, we chose to not pursue this optimization.

3) *Compression rounds*: Another pipelining optimization considered was folding multiple compression rounds into a single pipeline stage. However, given the prohibitively long combinational path introduced by arithmetic within each round, we found this infeasible while meeting timing on a 100MHz clock. In fact, we had to do the opposite, breaking up each compression round itself into two cycles to meet timing. Due to dependencies between arithmetic stages, we were unfortunately not able to fully saturate the compression round pipeline, leading to a halving in throughput to satisfy timing. Even with intentional balancing of the amount of logic on either side of the pipeline, timing was tight.

C. Resource Utilization Scaling

The final implementation consists of the following: (1) 10 parallel *sha256d* blocks; (2) a single implicit pipeline stage between *Hash Engines* of our *sha256d* module; (3) no BRAM.

- **LUTS**: 24,318 / 32,600. The majority of LUT logic comes from the arithmetic within the *Hash Engine*’s compression functions. Thus, LUT usage increases with the number of parallel blocks.

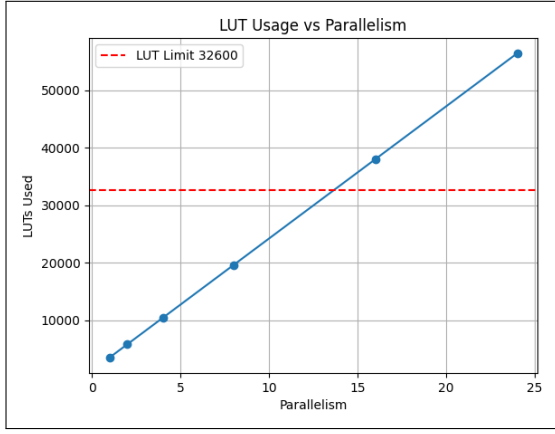


Fig. 7. LUT Scaling w.r.t Number of Parallel Hash Engines

Figure 7 demonstrates that LUTs scale linearly with parallel hash blocks. Our linear regression shows a baseline LUT usage of 3,478 for a single hash engine, with a scaling factor of 2,308 per additional parallel engine; this suggests a theoretical cap of 13 engines, contingent on other resource constraints being met.

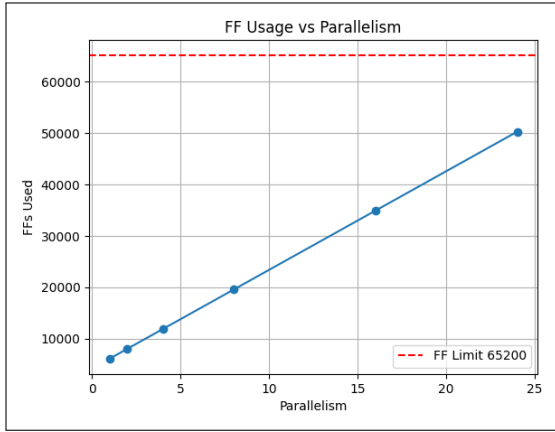


Fig. 8. Flip flop Scaling w.r.t Number of Parallel Hash Engines

- **Flip flops:** 23,692 / 65,200. Flip flops are primarily dedicated to pipelining within each compression round, breaking up long combinational adder chains. A single engine utilizes 6,040 flip flops. Regression suggests a scaling factor of 1,944 FF/engine, and a theoretical maximum of 31 engines. Flip flops are not our bottleneck.
- **Timing (WNS):** +0.025ns. To our surprise, timing was the key bottleneck in our design. For a single hash block, our WNS was comfortably positive at +0.512ns but, as we scaled the number of parallel blocks, we found that timing scaled inversely proportionally (see Figure 9). As a result, at 11 parallel blocks we were no longer able to make timing, with a slightly negative WNS of -0.007ns. We explored numerous options to reduce pressure on this bottleneck, though were unsuccessful to this end. Two ideas we considered were adding another pipeline

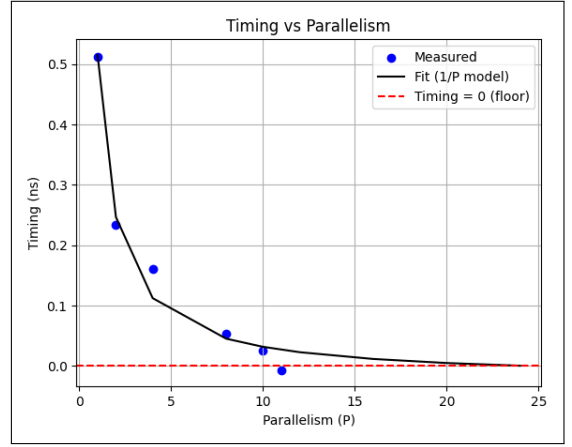


Fig. 9. WNS Scaling w.r.t Number of Parallel Hash Engines

stage within the *Hash Engine* or decreasing our clock speed, but found that the gains of 2-3 additional parallel hash blocks were not worth slowing down the rest of the existing system. Alternatively, we identified fan-out (e.g., the same midstate signal being sent to many parallel engines) as a potential perpetrator. Replicating registers affected by this fan-out was also not effective. Finally, we tested Vivado directives in various ways to mount parallel engines and arithmetic on DSPs instead of LUTs, but this did not improve WNS. As a result, while LUTs and flip flops offer the upside of up to 13 and 31 parallel blocks, respectively, timing restricted us to just 10. At 10 parallel engines, we consistently meet timing.

- **BRAM:** Initially, we intended to utilize BRAM to store bitcoin protocol constants and intermediate computations like the midstate. However, we found these constants to be small enough that they didn't apply significant enough pressure on our flip flops to warrant storing in BRAM.

IX. EVALUATION

A. Correctness

Correctness testing is performed both in hardware and software. The repo's README contains instructions for verifying correctness.

1) *Hardware:* Correctness is tested primarily by interfacing with the hardware through *pc_host.py*. This script takes an 80-byte block header in hex as argument. Subsequently, it computes the corresponding target in software and communicates to the FPGA over UART. Eventually, the FPGA will return a nonce value, which *pc_host* will validate by computing the forward hash and comparing it to the target.

The default behavior of *pc_host* is to run on block 125552, which has a correct nonce of 0x42a14695, and is well-known for having the lowest hash value at the time of mining[3]. We have also verified outputs on the genesis block. Given the zero probability of accidentally computing the correct nonce this result, in addition to verifying robustness of signals in sim, these results give us confidence in system correctness.

nonce_is_correct [9:0]	0...	001	000
uart_tx_data [7:0]	00		95
rest_nonce [23:0]	000000		42a146

Fig. 10. Correct UART Transmission Data

2) *Simulation*: Initial validation was done primarily in simulation. Sim tests began with unit testing (e.g., on the *Hash Engine*), and then grew towards a more end-to-end integration framework as we built out the rest of the system. *test_correctness.py* does exactly this: It simulates the process of sending an assignment to the miner, and receiving and verifying the result on the other end. Sim test analysis was done primarily through inspecting waveforms, allowing a more dynamic approach to verifying and debugging signals, values, and timing. Figure 10 shows the correct nonce of 0x42a14695 for block 125552 being loaded into the UART transmitter, following a single-cycle high of *nonce_is_correct*.

B. Throughput

With 10 parallel blocks, a single pipeline stage between the two *Hash Engines* in the *sha256d* modules, and a 128-cycle latency on each *Hash Engine*, the system currently achieves $\text{throughput} = \frac{\text{parallel_sha256d}}{t_{\text{clk}} * \text{cycles_per_sha256d}} = \frac{10}{(10 \text{ ns})(128)} = 7.81 \times 10^6$ hashes/sec. This matches almost exactly our empirical timing on block 125552: $\text{throughput} = \frac{0x42A14695 \text{ hashes}}{143s} = \frac{1117865621 \text{ hashes}}{143s} = 7.82 \times 10^6$. This is also about 10x faster than our benchmark C++ software implementation, which reaches 0.88×10^6 hashes/sec, demonstrating the effectiveness of our hardware accelerator.

Throughput was below our ambitious goal of 5×10^7 hashes/sec, particularly due to unforeseen timing bottlenecks that forced extra pipelining and restricted our parallel scaling.

C. Timing

As discussed in the performance engineering section, timing was an unforeseen bottleneck. That being said, by pipelining compression rounds and selecting 10 parallel hash blocks, we are able to consistently meet timing, making the system robust and performant.

D. Checklist, retro, and future work

1) *Checklist*: We fully meet our *commitment*. We fully meet an equivalent but revised version of our *goal*, scaling effectively to maximize throughput on the board, but focusing on parallel engines instead of pipelining stages for reasons discussed previously. Due to timing and feasibility constraints, we do not fully meet our *stretch goal*. That being said, we parametrize our system and perform a theoretical regression analysis so that, for a given board with known number of LUTs and FFs, we can scale intelligently. Furthermore, we implement the seven-segment display as an alternative to the other stretch goal features.

2) *Retrospective*: Correctly implementing the hash behavior was especially challenging due to the exact specification for the algorithm being difficult to find in full online. This cut into some of the time we had originally set aside for stretch goals. In retrospect, it would have been more efficient to fully understand these modules upfront by implementing them in a simpler language like Python. All things considered, we feel positively regarding our implementation and progress: We were able to get the miner working in sim and hardware, reason practically and theoretically about scaling, and implement a friendly user interface, meeting the majority of our goals.

3) *Future work*: First, we would attempt to relieve the timing bottleneck to fully scale our design as a function of LUTs on the board and meet our theoretical analysis presented in the performance engineering section. As a next step, we would like to try transferring our design to a larger board. Finally, we would also consider implementing the entire 138 bits of flexibility across *nonce*, *extra_nonce*, and *time* to make our design compatible with mining pools.

X. CONTRIBUTIONS

Parker drove the implementation of the *Top Level* and *Coordinator* modules, maintaining a high-level view of how the system was wired together. He was also responsible for parametrizing the system for easy scaling of parallel engines, the seven-segment display, and testing in sim. For the paper, Parker was primarily responsible for creating the module architecture diagrams. He also wrote the Design Overview, Bitcoin Protocol, UART, and Seven-Segment Display sections.

Sean focused on the implementation of the hash engine, pipelining, and pushing performance on hardware. The latter part required building the Python host, tackling UART transmission, and dealing with timing. For the paper, Sean designed the pseudocode for the mining and hashing algorithms, and wrote Python notebooks to generate scaling plots. He wrote the Abstract, Coordinator, Hash Engine, Performance Engineering and Scaling, and Evaluation.

We appreciate Joe Steinmeyer for his support and guidance during this project.

XI. SOURCE CODE

The code relevant to this project can be found here: <https://github.mit.edu/6205F25/fa25-6205-team54.git>.

REFERENCES

- [1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: <https://bitcoin.org/bitcoin.pdf>.
- [2] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS) – FIPS PUB 180-4," 2015. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.
- [3] "Bitcoin block 125552," CryptoID Bitcoin Blockchain Explorer, 2011. [Online]. Available: <https://btc.cryptoid.info/btc/block.dws?125552.html>.