CHAIR OF COMPUTER ARCHITECTURE AND PARALLEL SYSTEMS

Efficient FPGA Implementation of Blockchain Operations

Summer Semester 2020 - Winter Semester 2020/21

Eke Timur Schapeler Nicolas Starnecker Christoph Weiser Florian

Contents

1 Introduction

In recent years blockchain technologies and especially their most common use case, cryptocurrencies, have gained more and more public attention. One of the first implementations of blockchain was the cryptocurrency Bitcoin. It not only represents an opportunity for exchanging digital money but also for earning money by providing the computing power that keeps the underlying blockchain running. This money is earned in form of an incentive for solving a certain computational problem. Since this problem is very narrow, people started using not only CPUs but also GPUs, FPGAs, and ASICs to solve it more efficiently.

FPGAs provide the possibility to create hardware that is on the one hand more specialized than a GPU or CPU but on the other hand more flexible than an ASIC. This allows to implement further blockchain algorithms in the future and hence to switch to the most lucrative one easily. Therefore it is an interesting challenge to now implement a circuit capable of working efficiently on the Bitcoin blockchain, which is currently the most common one.

In the following, we will describe our outcomes from working on such an implementation for a given FPGA board. First, we give a short introduction to blockchain, Bitcoin, and the SHA-256-Algorithm. Second, we will depict our base architecture and the communication within the FPGA and to the host. Based on this architecture, we will explain which different approaches we used to increase our performance. We further depict how to show the correctness of our implementation. Eventually, we have a look at the effects of different optimizations on the performance and compare our FPGA-based architecture to other implementations.

2 Background

In this section, we will introduce the basic ideas of the Bitcoin blockchain and particularly the main algorithm SHA-256, which we will implement. A basic understanding of these concepts is necessary to comprehend the underlying considerations of our architecture and optimizations.

2.1 Blockchain and Bitcoin

The fundamental idea behind blockchain is the cryptographic linking of chunks of data, which are the blocks, in order to make the contained timestamp and data unalterable. Therefore, every subsequent block contains not only the next part of the data but also a cryptographic hash digest of the previous block [2].

This makes it difficult for attackers to modify the blocks afterwards, as altering a given block leads to a different hash digest[1]. To keep the chain valid this new hash digest would have to be included in the next block, which would consequently alter that block as well. As a result, all of the blocks following the modified block have to be recalculated upon a change. If this recalculation is not possible in a fast way by the attacker, the regular chain will outpace the modified chain. The modified chain is then not recognized as real and ignored by the network, effectively mitigating the attack.

One of the first realizations of a blockchain was introduced by Satoshi Nakamoto in form of the cryptocurrency Bitcoin [3]. The currency uses a blockchain as a so called *distributed ledger* to keep track of all transactions to avoid double spending and retroactive changes of the transaction history. As depicted in figure 1, the linked part of the chain are the so-called block headers.

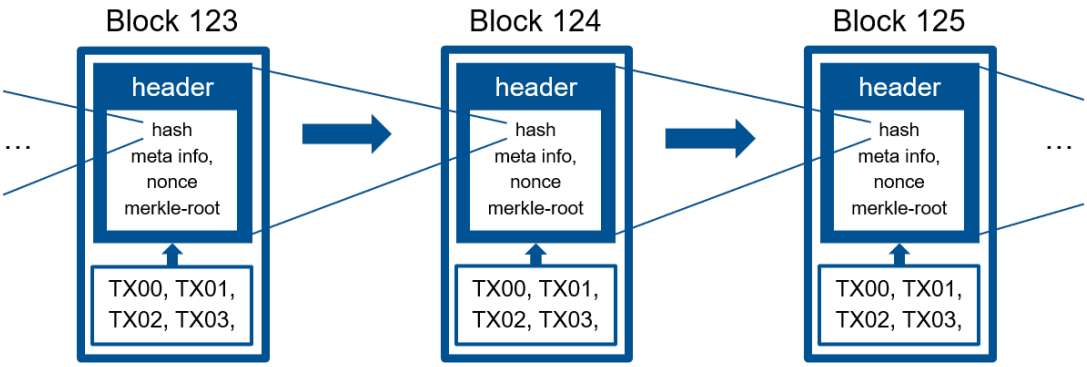

Figure 1: Schematic outline of the Bitcoin chain

The headers are linked through the cryptographic hash digest of the previous block header. Furthermore, they include meta data describing the respective block as laid out in figure 2. The transactions are integrated into the block's header via the *merkle root*, which is a cryptographic hash digest summarizing them.

Figure 2: Format of a Bitcoin block header

In order to verify the blocks and transactions within them, the Bitcoin protocol uses a *decentralized consensus system*. Every new block needs a so called *Proof-of-Work* (PoW) in

[1] Although it is theoretically possible that we actually get the same hash digest again this is due to the currently unbroken collision resistance of SHA-256 very unlikely.

order to get accepted by the Bitcoin peer-to-peer network and finally to get appended to the chain. In the case of Bitcoin, these proofs are the characteristic leading zero bits in the double-applied SHA-256 (SHA-256d) hash digest of the header, or, in more technical terms, a digest below a certain target value. The action of searching for such a digest is called *Bitcoin mining*.

Since the SHA-256 algorithm is designed in such a way that we cannot force to get certain properties in a hash digest, the only alternative is pure brute-force, which consumes a lot of calculation power. Therefore each PoW guarantees that, on average, a certain amount of computational power was put into the creation of a given block. If that amount is so high that even all network participants combined need several minutes per block, it is very hard for an individual attacker to be fast enough to rewrite the history of blocks as described earlier.

The main part of the mining procedure is the search for the hash digest. Because a hash function returns the same value for the same data every time, we have to change our header if the hash digest is not below the current target. Such a change can be achieved by either modifying the merkle root, the timestamp or the nonce. The other fields (version, previous hash and target) are fixed for the current block. Changing the merkle root is achieved by either including other transactions or changing a special field in the coinbase transaction, the first transaction which includes the incentive for finding a block. Both ways are quite costly since the merkle root has to be (partly) recalculated. Simpler ways of changing the header are direct increments of the dedicated 32-bit *nonce* field or the timestamp in the header. These options are the ones we will use later.

Finally to guarantee that the PoW still works even if a lot of miners join the Bitcoin network, the target value is regularly adjusted to keep the rate of blocks constant. Therefore, we cannot hard-code the target value in this implementation and must require it to be read out from the header.

2.2 SHA-256

SHA-256 is the core algorithm of the Bitcoin proof-of-work and therefore the main algorithm we implement on our FPGA. It was officially introduced in the U.S. federal standard FIPS 180-2 in 2002 as part of the SHA-2 hash function family. It is a collision and preimage resistant hash function able to hash messages with a length of up to 2^{64} bit into 256-bit message digests [4].

Preimage resistance means that it is infeasible to find a message for a given digest. This property makes the algorithm suitable for the verification of blocks in the Bitcoin protocol because the only way to find a valid block hash is through brute force, as already mentioned earlier.

Now we will have a look at the details of the hash digest calculation according to the FIPS standard [4]. The algorithm can be split into two major parts, namely preprocessing

and hash computation.

Preprocessing

During preprocessing the message has to be prepared to fit the algorithm. To make the subsequent subdivision into 512-bit chunks possible, the message is first padded to have a length in bits divisible by 512. That is achieved by adding a 1 bit to the end of the message M and filling it with zeros until the length l' of the result matches the equation $l' \equiv 448 \mod 512$. Lastly, the length of the original message M is appended as a 64-bit binary representation.

The subdivision into processable units is done by splitting the padded message M' into N 512 bit chunks $(M^{(1)}, \ldots, M^{(N)})$. Each of the chunks can also be seen as 16 32-bit words $(M_0^{(i)}, \ldots, M_{15}^{(i)})$.

Before starting the calculation it is also important to set the initial hash values $H_1^{(0)}, \ldots, H_7^{(0)}$ according to equation (1).

$$\begin{aligned} H_0^{(0)} &= \texttt{6a09e667} \\ H_1^{(0)} &= \texttt{bb67ae85} \\ H_2^{(0)} &= \texttt{3c6ef372} \\ H_3^{(0)} &= \texttt{a54ff53a} \\ H_4^{(0)} &= \texttt{510e527f} \\ H_5^{(0)} &= \texttt{9b05688c} \\ H_6^{(0)} &= \texttt{1f83d9ab} \\ H_7^{(0)} &= \texttt{5be0cd19} \end{aligned} \tag{1}$$

Hash computation

The hash computation always processes one chunk at a time, so the following steps are executed for each of the N blocks.

1. Extend the chunk $M^{(i)}$ to 64 message schedule words $(W_0, \ldots, W_{63})$ according to equation (2).

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

2. Set the working variables $a, \ldots, h$ to the last hash values $H_1^{(i-1)}, \ldots, H_7^{(i-1)}$ respectively.

3. In round t from 0 to 63, process message schedule word W_t according to equations (3) using functions (4) and (5) as well as constants K_t (6).

$$\begin{aligned}
T_1 &= h + \sum\nolimits_1(e) + Ch(e,f,g) + K_t + W_t \\
T_2 &= \sum\nolimits_0(a) + Maj(a,b,c) \\
h &= g \\
g &= f \\
f &= e \\
e &= d + T_1 \\
d &= c \\
c &= b \\
b &= a \\
a &= T_1 + T_2
\end{aligned} \tag{3}$$

$$\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \quad \oplus (x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \quad \oplus (x \wedge z) \oplus (y \wedge z)
\end{aligned} \tag{4}$$

$$\begin{aligned}
\sum\nolimits_0(x) &= \mathrm{ROTR}^2(x) \quad \oplus \mathrm{ROTR}^{13}(x) \quad \oplus \mathrm{ROTR}^{22}(x) \\
\sum\nolimits_1(x) &= \mathrm{ROTR}^6(x) \quad \oplus \mathrm{ROTR}^{11}(x) \quad \oplus \mathrm{ROTR}^{25}(x) \\
\sigma_0(x) &= \mathrm{ROTR}^7(x) \quad \oplus \mathrm{ROTR}^{18}(x) \quad \oplus \mathrm{SHR}^3(x) \\
\sigma_1(x) &= \mathrm{ROTR}^{17}(x) \quad \oplus \mathrm{ROTR}^{19}(x) \quad \oplus \mathrm{SHR}^{10}(x)
\end{aligned} \tag{5}$$

Constants K_t (from left to right)

```
428a2f98  71374491  b5c0fbcf  e9b5dba5  3956c25b  59f111f1
923f82a4  ab1c5ed5  d807aa98  12835b01  243185be  550c7dc3
72be5d74  80deb1fe  9bdc06a7  c19bf174  e49b69c1  efbe4786
0fc19dc6  240ca1cc  2de92c6f  4a7484aa  5cb0a9dc  76f988da
983e5152  a831c66d  b00327c8  bf597fc7  c6e00bf3  d5a79147
06ca6351  14292967  27b70a85  2e1b2138  4d2c6dfc  53380d13
650a7354  766a0abb  81c2c92e  92722c85  a2bfe8a1  a81a664b
c24b8b70  c76c51a3  d192e819  d6990624  f40e3585  106aa070
19a4c116  1e376c08  2748774c  34b0bcb5  391c0cb3  4ed8aa4a
5b9cca4f  682e6ff3  748f82ee  78a5636f  84c87814  8cc70208
          90befffa  a4506ceb  bef9a3f7  c67178f2
```
(6)

4. After processing all words of a chunk the result is added to the old hash values:

$$\begin{aligned} H_0^{(i)} &= a + H_0^{(i-1)} \\ H_1^{(i)} &= b + H_1^{(i-1)} \\ H_2^{(i)} &= c + H_2^{(i-1)} \\ H_3^{(i)} &= d + H_3^{(i-1)} \\ H_4^{(i)} &= e + H_4^{(i-1)} \\ H_5^{(i)} &= f + H_5^{(i-1)} \\ H_6^{(i)} &= g + H_6^{(i-1)} \\ H_7^{(i)} &= h + H_7^{(i-1)} \end{aligned} \tag{7}$$

Lastly, after processing all N chunks, the hash digest is the concatenation of the hash values:

$$\text{message digest } = H_0^{(N)} \left\| H_1^{(N)} \right\| H_2^{(N)} \left\| H_3^{(N)} \right\| H_4^{(N)} \left\| H_5^{(N)} \right\| H_6^{(N)} \| H_7^{(N)} \tag{8}$$

3 Basic architecture

In this section, we present our approach to implementing the Bitcoin Mining FPGA. We will derive the base architecture of the system from the target algorithm, as well as present the method of internal communication between the components of the system. Finally, the connection to the host will be depicted.

3.1 SHA-256 Implementation

For the purposes of work-sharing, modularity, and testing, it is imperative to divide the computation of a SHA-256 hash among different components. While the preprocessing is done in a single component, we split the hash computation into two parts based on the functionality and the data being processed.

Padder

The preprocessing, which consists of the padding, is done in a component appropriately named padder. The output of the padder ($M_0^{(i)}, \ldots, M_{15}^{(i)}$) is only dependent on the input length, so we hardcode the specific padding logic for both possible input lengths for the SHA-256d calculation. The two possible lengths are 640 bit for the first hash of a block header and 256 bit for the second hash of a SHA-256 hash digest. Furthermore, we implement two different padder components, each specialized on an input size to

```
padded_message(511 downto 255) <= hash_buffer & '1';
padded_message(254 downto 9) <= (others => '0');
padded_message(8 downto 0) <= "100000000";
```

Figure 3: Code of the padder for a 256-bit message

eliminate the need for any control logic. This, along with known input lengths, allows both padders to be implemented with concurrent statements as in figure 3, making them clock-less, simple, and efficient.

The padded message is then divided into chunks, which are processed sequentially by the next component, the extender. Each chunk is now interpreted as 16 32-bit words.

Extender

The task of the extender is to generate 64 message schedule words ($W_0, \ldots, W_{63}$) from the padded chunk. The first 16 words are directly adopted from the received chunk, while each subsequent word is generated using the preceding 16 words. As we will discuss later, data dependencies allow for two words to be generated concurrently.

Compressor

The compressor holds the working variables, which get transformed for 64 rounds using message schedule words as specified in the previous section. As each round depends on the previous one, the compressor cannot be parallelized and is thus the main performance-critical part of the system.

To eliminate the need for a large buffer for the message schedule words generated by the extender, we do not precompute them in favor of synchronous calculation while the compressor is running. Furthermore, we already add the constants K_t to W_t in the extender prior to passing W_t to the compressor. This removes the need for the compressor to keep track of the current round number, reducing its complexity.

To compute a Bitcoin block hash, the block header has to be hashed twice. While it is possible to sequentially compute both hash digests on a single set of components, this introduces additional control logic and discourages stage-specific optimizations (e.g. hard-coding the padder for 640 and 256 bits). Therefore we decided to chain two sets of components needed for the computation of SHA-256, as seen in figure 4.

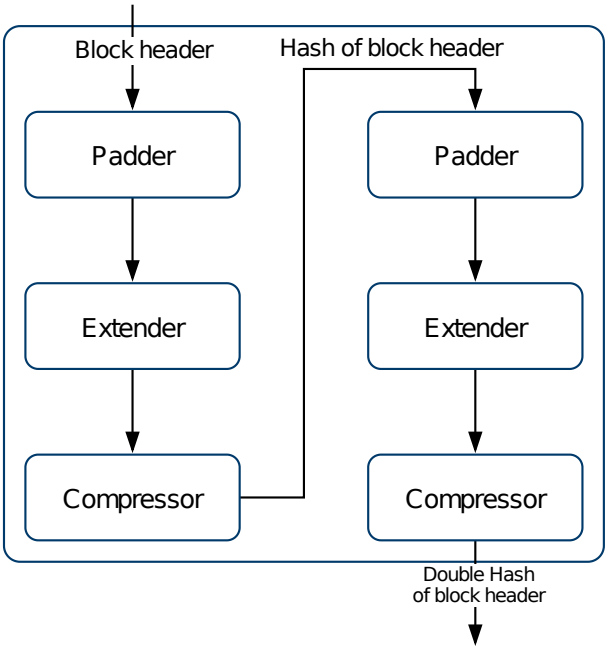

Figure 4: Components for the computation of a SHA-256d hash

3.2 Internal Communication

In order that the padder, extender, and compressor components work together seamlessly, a common communication protocol is required. Such a protocol should include the intermediate hash value calculations as well as additional signals to synchronize the transfer of information.

First Approach: Master-Slave Communication Protocol with Feedback

Our first approach to synchronize the data transfer between the SHA-256 components was to implement a feedback-based master-slave protocol. The master sends a chunk of data to the slave and waits until it confirms that the data chunk has been processed before sending the next one. For this, the master possesses an outgoing "enable" signal to inform the slave about new data. Additionally, the slave component has an outgoing "ready" signal to the master which communicates if it is ready to receive new data. This approach is visualized in figure 5. The main advantage of this communication protocol is that components don't need to know the execution times of other components and thus are more independent from each other.

However, this communication protocol isn't well suited for a speed optimized FPGA design because the duration between setting the ready signal and receiving the next

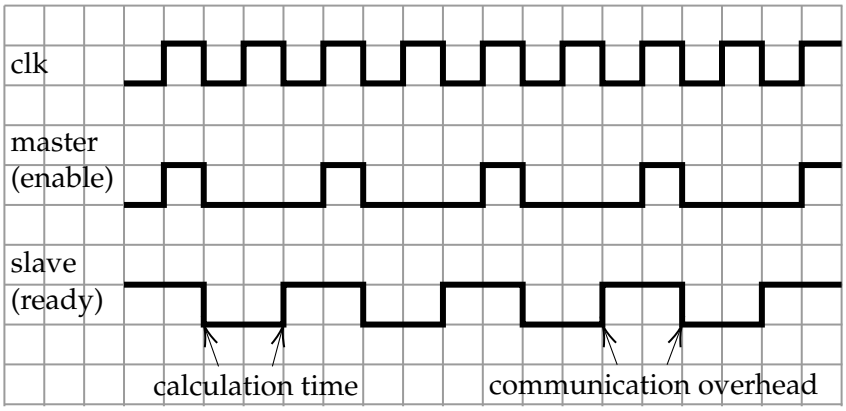

Figure 5: Rising edge clocked master slave communication protocol

enable signal cannot be used by the child component to perform calculations.

Better Approach: Individual Communication Protocol for Each Component

The most efficient way to reduce the aforementioned communication overhead is to make use of the knowledge about other component's execution times, more specifically the number of clock cycles a master component has to wait before sending the next data to its slave. We'll demonstrate this by taking a closer look at the padder-extender and extender-compressor connections using this improved approach. In general, the bottlenecks of our SHA-256 implementation are the extender and compressor components, as they have to perform internal calculations for each of the 64 rounds. Thus, we will focus on removing the communication overhead in these two components.

As for the padder-extender connection, one can e.g. implement a slightly modified master-slave communication protocol. We already know that the padder only needs one cycle to generate a new padded chunk and that the extender needs considerably more cycles to process it. To let the extender work without a communication overhead, the extender can send the ready signal two cycles before finishing its own calculations. After one cycle, the padder receives the ready signal from the extender and after another clock cycle, the next padded chunk is available for the extender. Consequently, the extender can immediately continue with calculating the next block header.

The extender-compressor connection can be realized through three one-bit signals from the extender to the compressor in addition to the data vector. This allows the extender to notify the compressor about relevant events: when data is sent, when a new block header is processed, and when the last piece of information belonging to the current block header is sent. The compressor doesn't need to give feedback to the extender after calculating a hash because both components have the same duration of execution.

3.3 External Communication

In order to send the data to work on to the FPGA and to receive the result, we need a connection between the FPGA and host. Due to our very specialized task, the FGPA just has to understand two communication requests:

1. Receive a block header.
2. Send the current nonce and whether it leads to a valid hash digest.

If we look at figure 2, we can see that all information necessary (header and target) for computing hashes and comparing these against the target value is already included in the header. Therefore, no further information has to be sent for the calculation. Technically we do not need to pass the nonce, which is the last 32 bits of header, as this value is meant to be changed for each hash. For easier testing, however, it is still included and used as the base value that the FPGA begins computing hashes from. Once a new header is sent, the FGPA is reset and immediately starts with the new calculation. This is justified because when we send a new header to the FPGA, it is either due to detecting that the previous header would lead to an invalid block or already having found a valid hash for the previous header. It would then be of no use to further work on it. One of these situations occurs every time we or another miner found a block which includes some transactions included in the header we are currently computing hashes for.

Upon first inspection, it appears quite obvious that we should return the calculated hash along with the nonce and a bit showing whether the search was successful for the response. The problem with this approach is the resource consumption of the logic gathering the hash and sending it back to the host. Since finding a valid nonce happens quite rarely, this part of the board would be unused most of the time. Therefore, we only send back the current nonce and a found bit and opt to calculate the SHA-256d hash on the host for the case where the nonce is valid.

Another point we have to care about is keeping the stall cycles of the FPGA as low as possible. That is to say, ensuring the FPGA starts with the next header as soon as possible after finding a valid block or finishing all possible nonces. At first some sort of header queue comes to ones mind. However, this would consume resources to store another 640 bit header on the FPGA board and is therefore not optimal.

In our case, there is a much simpler solution, but the following considerations only work based on the assumption that the time for trying all 2^{32} nonces is significantly greater than one second. For the highest achieved nonce rate of 50.0 *MHash/s*, we need $2^{32} Hash / 50.0\ MHash/s \approx 85.9s \gg 1s$ to try out all different nonces. After all nonces for a given header have been tried out without success, we have to change another part of the header. The simplest change that can be done is the increment of the timestamp. Since the time is represented in UNIX Epoch time, an increment by one represents a time increment by one second. Here our assumption assures that we do not surpass the

current real-time and always calculate with valid timestamps[2].

Both request are sent to the FPGA via a Python script using UART. On the FPGA we implemented a dedicated component for accepting requests that acts as an interface to the mining components.

4 Optimizations

If one were to implement a Bitcoin Mining FPGA as described earlier, though functional, it would be highly inefficient both in terms of speed and utilization of the logic slices available, approximately computing one Bitcoin double hash every 192 cycles[3], if optimally implemented. In the following three subsections we aim to give more insights into the optimizations used to increase the rate at which the FPGA computes the Bitcoin double SHA-256 and methods used to more efficiently employ the logic slices available to us. The first subsection details the way our implementation handles parallelization, the second describes the usage of pipelining in this use case, and the final section catalogs the optimizations we applied to the individual components.

4.1 Parallelization

The first optimization analyzed will be parallelization, where the main motivation is to more effectively utilize the resources available to us.

4.1.1 Mining Core Component

In order to parallelize our system, we implemented the here-called mining core. This component needs to fulfill three demands in order to be useful for our parallelization efforts:

1. Compute hashes for a fixed set of nonces.
2. Evaluate whether it has found a valid nonce.
3. Be resettable.

[2]The Bitcoin network accepts timestamps within 2 hours of the network adjusted time. In the unlikely case that no valid hash is found during this time, the timestamp needs to be updated externally.

[3]The initial header is 640 bits long, padded to 1024 bits. The first stage of the Bitcoin double SHA-256 takes two iterations, one for each 512-bit chunk. Of these two iterations, each takes 64 rounds for extending the chunks into the message array and applying the compression function, adding up to $2 \cdot 64 = 128$ cycles (if one round is calculated in one cycle). The second stage of the Bitcoin double SHA-256 receives the result of the first one, a 256-bit message, which is then padded to 512 bits. This leads to one iteration, meaning 64 rounds of the extender and compressor. Summing these values up, a total of $128 + 64 = 192$ cycles is needed per Bitcoin double hash, without accounting for the time needed for padding and synchronization of the components.

To realize these, this component is given a certain nonce generation scheme, threshold, and header with which it continuously computes SHA-256d hashes. It does this by appending the current nonce generated through the externally provided algorithm to the header for which a valid hash is desired, computing a hash, and repeatedly regenerating the next nonce through the previously named scheme until a hash satisfying the given threshold is found. This functionality is executed by extending our previous SHA-256 implementation through prepending a nonce generator and appending a comparator component. The nonce generator computes the next nonce with which the header is to be padded according to the given technique and the comparator verifies if the computed hash matches the provided constraints, setting an output signal in the case where it does. Finally, the mining core is resettable through an input signal that can be set by an external actor, which is asynchronously redirected to all subcomponents of the mining core, thereby interrupting all computation inside of it and reseting all subcomponents to their initial state. Through these three extensions, we have now created a controllable, multipliable unit capable of computing SHA-256d.

4.1.2 Distribution of Work

The next aspect to realizing this parallel system is to define the above named nonce-generation algorithm that is to be used for assigning groups of nonces to a mining core. Due to the only aspect of a given nonce being influential on the time of computation being its length (which is constant across nonces) and SHA-256's preimage resistance all nonces are equally valuable. Since no nonce leads to a faster computation or greater chance of success, the only demand our algorithm has to meet is to avoid duplicate computations using the same nonce.

To achieve this, we define the following algorithm:

1. Given n mining cores $\{c_0, ..., c_{n-1}\}$, compute its modulo remainders $\{0, ..., n-1\}$.
2. Assign each mining core c_i their corresponding modulo remainder i.
3. A mining core computes their first hash using i as a nonce. If this hash is not valid, it keeps generating nonces and hashing with them by continuously adding n to their i.

Since each natural number a (and the nonce can be interpreted as a natural number at no penalty in this case) can be written as $a = i + b \cdot n$ where i is a modulo remainder of n and b is some natural number, this algorithm ensures all possible nonces are hashed. Additionally, since this expression is unique for each nonce (as each nonce can only have one modulo remainder) the algorithm also ensures no two mining cores compute using the same nonce.

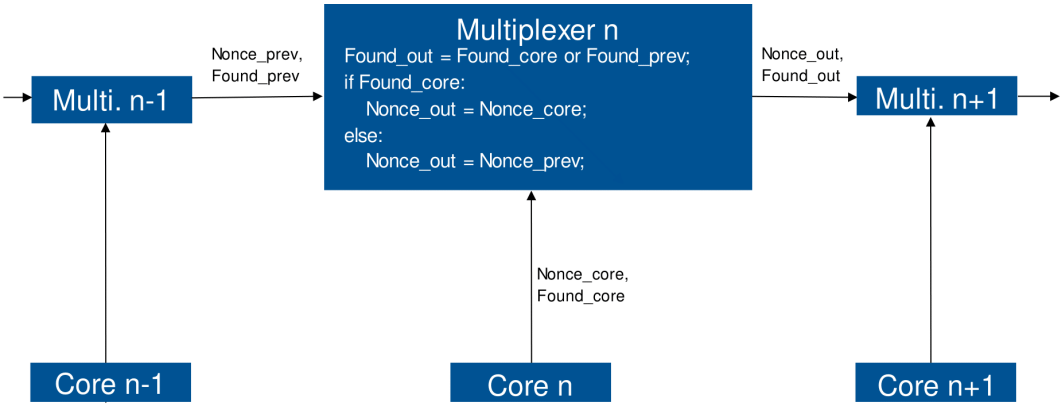

Figure 6: Architecture of our system including multiplexer for result reduction

4.1.3 Mining Core Synchronization

Finally, we need to deduce a methodology to reduce all results we get from each mining core while keeping performance penalization to a minimum. To do this, we create a corrsponding multiplexer component for each mining core, as shown in figure 6. where the last multiplexer points to the receiver of the reduced information, in our case a component that communicates this information to an external actor, as described in Section 3.3.

The setup above yields a few advantages:

First, due to its simplicity, it can be implemented on hardware using just a few logic gates, making it rather small. This is relevant, as keeping this component small allows for more space to place mining cores which increase performance.

Next, evaluating this chain of multiplexers does not require waiting on the presence of any signals and can therefore be evaluated fully concurrently, meaning the impact on performance is absolutely minimal. The reason there is no need to wait on the presence of any signals is due to the design of the mining core: The found signal outgoing from each core is set to 0 by default, however, the nonce signal is set as soon as the next nonce is generated by the nonce generator. This premature setting of the nonce is not an issue, as the chain of multiplexers filters out nonce signals whose corresponding found signals are not set. The advantage of the just-described configuration is that this way in the case where we find a valid nonce, by the time the found signal is set, the nonce signal will already have been defined for the whole duration of the SHA-256d hash computation. This ensures we avoid the case where a valid found signal is sent prior to the updated nonce arriving.

An additional property of this structure is the way it handles the (extremely unlikely) case that two cores both compute a valid hash at the same time using two different nonces. As can be seen from figure 6, this case is handled by selecting the nonce from

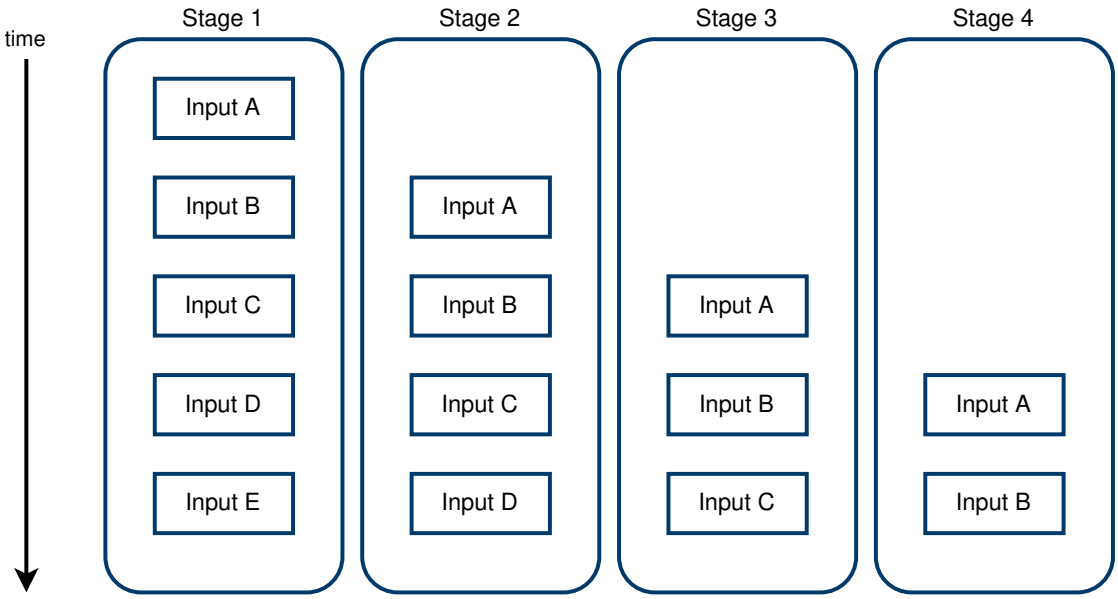

Figure 7: Filling of a pipeline

the core with the greater index, which in this case serves no direct purpose. The reason we can do this, is due to the fact that the value of a nonce is entirely irrelevant to the Bitcoin Network, as long as its hash concatenated behind its header is lower than the mining target at the time of its mining.

4.2 Pipelining

In general, pipelining does not reduce the duration of computation but it improves the throughput, allowing for greater utilization of the logic cells. The computation is divided into stages so that the hardware of each stage can process different inputs with the stages working concurrently. To ensure the optimal throughput, each stage has to take the same amount of time, else the throughput will be defined by the slowest stage, as others must wait for it to complete. Additionally, the pipeline has to be 'filled' to function the best, which means that n computations have to be ongoing for n stages. This concept is illustrated in figure 7. In our case, as resets happen infrequently, the pipeline will almost always be filled.

4.2.1 Simple Pipelining

For the mining core, the division into stages is apparent: the first and the second SHA computations become the two stages of the pipeline. Without this pipelining, the first set of components previously had to wait for the second hash to complete before beginning

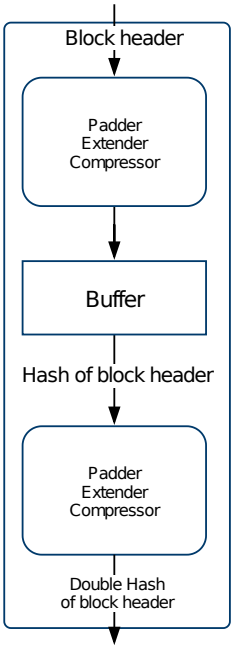

Figure 8: Pipelined mining core

to process the next block header. Now, after the first hash is computed, it is cached in a buffer between the stages, meaning the first stage can proceed to process the next block header, while the second stage works with the cached hash of the current block header, as seen in figure 8.

Upon closer examination, we determine the durations of the stages: stage 1 takes $2 \cdot 64$ compression rounds, while stage 2 takes only 64 rounds. So, with the pipeline filled, the throughput is 1 block header / $2 \cdot 64$ rounds, compared to 1 / $3 \cdot 64$ rounds without pipelining.

4.2.2 Use of Length Extension Vulnerability of SHA-256

Upon closer inspection, however, we can do even better: In the first stage of SHA-256d, we want to compute **SHA-256**(*header*), which we achieve by running the extension and compression function over the first 512 bits of the header, saving this state for the compression function, padding the remaining 128 bits of the header, and running the extension and compression function over these. Notice, however, how the first 512 bits contain the version of the block, the previous block hash, and part of the merkle root hash. All are values that remain constant between different mining cores and over time for a given header, as the only values that are changed are the nonce and the timestamp, both of which are contained in the last 128 bits.

We can take advantage of this by slightly changing the structure and functionality of our mining core: We designate a single mining core with an additional out-port that outputs a 256-bit vector which is then fed into the other mining cores. The content of this buffer will be the hash of the first 512 bits of the block header. Due to the length extension attack vulnerability SHA-256 [4] has, we can feed this resulting hash into the initial state of each compressor of the other mining cores and thereby skip computing the hash of the first 512 bits of our header for the remainder of the time we mine using it. Although this solution has the downside that for the time needed to compute the hash of the first 512 bits of the block header all but one mining core are not working, the upside in comparison is huge: For each nonce we compute SHA-256d for, we skip the amount of time we waited at the beginning for the designated mining core to compute the hash of the first 512 bits.

One may ask themselves why we opted for having one mining core compute this initial hash and distribute it to the others instead of having each core compute the initial hash themselves (which would come at no speed penalty, as this computation would be running on each core in parallel and with no time difference between cores). While the just-named solution provides the advantage of eliminating the communication overhead between cores, it requires an additional 256-bit buffer in which each mining core would need to store this hash.
In our implementation with the hash distribution, the compiler of the VHDL code appears to be smart enough to realize that all mining cores are reading out the same hash and therefore can optimize the space needed for this by presumably reusing the route connecting the computed hash to each mining core. In the case of having each mining core compute the hash for themselves, the compiler does not have as much opportunity for optimization. Therefore, though in most cases our implementation is likely more space-optimal, the choice for which design-choice of said length-extension attack is ultimately compiler-specific.

4.2.3 Compressor pipelining

Another interesting optimization idea is to further roll out the pipeline for each of the two hashing stages. For this, both stages would be split into 64 sub-stages, each of which correspond to one round of the SHA-256 algorithm. As a result, each mining core would only contain one nonce generator, padder and comparator, but $2 \cdot 64$ extenders and $2 \cdot 64$

[4] If given **Hash**(m_1) and the length of m_1, an attacker can compute **Hash**($m_1 \mid m_2$) where m_2 is some message controlled by the attacker, the hashing algorithm is vulnerable to length-extension attacks. SHA-256 is a special case, as an attacker can compute **Hash**($m_1 \mid m_2$) using only **Hash**(m_1) without knowing the length of m_1. The reason for this is that **Hash**(m_1) is the state of the compression function after running over the final chunk of m_1. If m_a were defined as $m_1 \mid m_2$, the hash for m_a could be computed by taking **Hash**(m_1) as the state for the compression function and running the compression and extension function over the added chunks from m_2. To be clear, an attacker did not need to know the value of m_1 in the previous example to compute **Hash**(m_a), knowing **Hash**(m_1) was sufficient. In our case, m_a can be viewed as the header, m_1 as its first 512 bits, and m_2 as its last 128 bits respectively.

compressors.

The main reason speaking against this optimization is that there aren't enough logic cells on the target FPGA board. This becomes clear by estimating the maximum saved area compared to our final implementation. As the two extender and two compressor components jointly take up around 85% of a mining core's logic cells, completely rolling out the pipeline while not changing the hash components themselves would increase the size of a single core $0,85 \cdot 63 + 1 \approx 55$ times. However, the constants of the compressor K and the compressor buffer, described in section 4.2.2, are only needed once per core and could be optimized in a rolled out pipeline. Removing them would decrease the size of a mining core by approximately 10%, still making it $0,9 \cdot 0,85 \cdot 63 + 1 \approx 49$ times larger than our final implementation. Given that our final implementation includes 16 cores, a single "rolled-out" core, which is 49 times larger, doesn't fit on the target board.

4.3 Optimization of Single Components

Aside from parallelization and pipelining, we applied hardware-specific timing and area optimizations to single components to do more work in one cycle and to allow for more mining cores to be deployed on the FPGA board.

4.3.1 Timing

In FPGA design, timing closure is a vital task and can be achieved by reducing the length of critical paths. As for our mining cores, using only the rising clock edge and transferring as few data outside of the mining core entities as possible solved most timing issues at first. The longest critical path after timing optimizations, however, is located in the compressor component, making it the main limitation for a mining core's throughput. In our final implementation, the compressor already caused some timing violations with a 100 MHz clock, but our deployment tests showed that these violations weren't significant enough to influence the correctness of the calculated hashes. Yet, we weren't able to raise the clock frequency to 200 MHz because of these timing violations.

Compressor Unrolling

To increase the amount of work done per clock cycle, we attempted to perform multiple compressor rounds in a single cycle.

Due to the structure of the internal state transformations, as depicted in figure 9, four rounds per cycle is the most straightforward unrolling of the compressor. Sadly, this version introduces a much longer critical path than is possible within a clock cycle. Additionally, the massive size increase of the components through this optimization

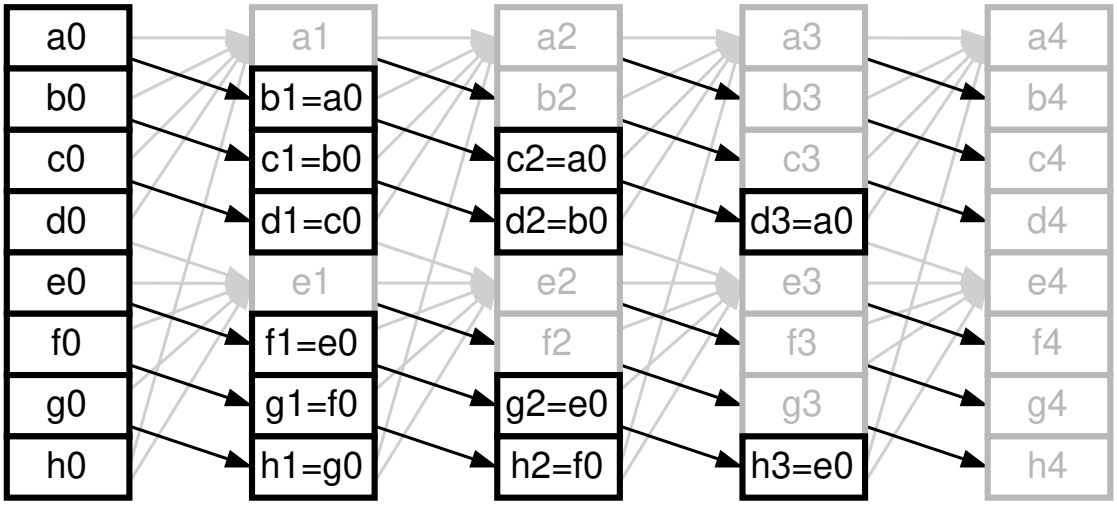

Figure 9: Permutation of state words in four compression rounds

means less cores fit on the FPGA board and therefore removes the possibility of managing the increased critical path by lowering the clock frequency, as this would impact performance more than the original optimization improves it. Due to these reasons, we didn't pursue this variant any further.

With an unrolling factor of two (rounds per cycle), a single mining core runs without timing issues on our standard clock speed. This is in part due to the two message digest words being able to be computed concurrently in the extender (as there are no data dependencies between them). Additionally, the total size increase of a mining core is smaller than two, which means that we would observe a net gain in performance for 12 or more functional cores (compared to 23 non-unrolled cores). We found the most problematic critical path to be during the cycle when the mining cores first receive the buffered hash of the first chunk of a header. Already for 14 cores, these critical paths are too long for our clock speed and cause the calculations to fail. While 13 cores would already increase our performance, 17 was our target, bounded by the number of logic cells available. To reach that goal, we reverted our changes to the shared buffer in hopes of reducing the critical path by buffering the first header hash inside each mining core. While this change did increase the size of the components, we could get 16 cores to run without timing issues.

4.3.2 Area

By decreasing the size of each mining core, more mining cores can fit on the FPGA board, and consequently the overall performance increases. In this chapter, we will take a look at different techniques to optimize the area occupancy of our Bitcoin mining implementation. It is important to notice beforehand, that the logic cells of the VHDL

entities are sometimes counted towards different components by the Vivado synthesis. Thus, the total number of logic cells belonging to a mining core is a more reliable measurement than those of a single component. In the following, all quantified logic cell optimizations of a component will assume that the other components didn't change in size unless noted otherwise.

First of all, some components of our implementation contained buffers, which could be removed without affecting the correctness of the Bitcoin mining algorithm. For example, the padder between the compressor of the first SHA-256 stage and the extender of the second stage doesn't need an internal 256-bit vector to store the intermediate hash as the compressor's output hash isn't changed before the next hash is calculated. Removing the redundant buffer from the padder reduced its size from 354 cells to 2 cells. Analogously, the internal comparator buffer could be removed by directly using the output of the second compressor. This decreased the comparator size from about 260 cells to just 7 cells.

Another method to reduce the number of logic cells per mining core is to lower the number of bits transferred between components. As an example, a mining core only transmits the found nonce (32 bits) to the main entity instead of the calculated hash digest (256 bits) or the whole block header (640 bits).

Extender Size Optimization

Out of all component size optimizations, reducing the extender size resulted in the most significant area enhancement as the two extenders initially were the largest components within each mining core. The vast majority of logic cells occupied by the extender was used by the 64-entry message schedule array.

In section 2.2 we already learned that the first step of the SHA-256 hash computation is to prepare the message schedule using the 16 padded message blocks. This task is assigned to the extender component in our implementation. Using formula 2, it can be seen that the first 16 values of the message schedule are directly copied from the padded chunk, and calculating W_t for $16 \leq t \leq 63$ only depends on the 2nd, 7th, 15th and 16th preceded message schedule array entries as demonstrated in figure 10. Since previous message schedule entries are exclusively used within the extender and only the most recent value is important for the compressor calculations at a time, it is possible to reduce the message schedule size from 64 to 16 entries by accessing all indices modulo 16. Algorithm 1 implements this idea.

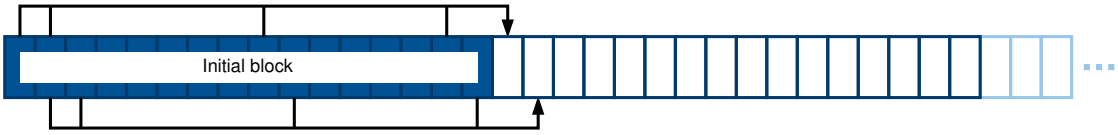

Figure 10: Data dependencies of extender

Algorithm 1 Pseudocode of the simplified extender with a 16-entry message schedule array

Require: functions σ_0, σ_1 and padded chunk $M[0 \dots 15]$
1: ARRAY $W[0 \dots 15]$
2: **for** $t \leftarrow 0$ to 63 **do**
3: **if** $t < 16$ **then**
4: $W[t] = M[t]$
5: **else**
6: $W[t \mod 16] = \sigma_1(W[(t-2) \mod 16]) + W[(t-7) \mod 16] + \sigma_0(W[(t-15) \mod 16]) + W[t \mod 16]$
7: **end if**
8: SEND_NEXT_WORD_TO_COMPRESSOR($W[t \mod 16]$)
9: **end for**

The FPGA hash rate can be further improved by swapping out the extender array into the block RAM or - if available on the FPGA board - into the distributed RAM. The block RAM typically consists of dedicated SRAM cells on the FPGA board, whereas the distributed RAM is generated with the logic cells' lookup tables. Block RAMs are generally well suited for large memories that does not occupy any logic cells. Distributed RAMs, in contrast, are ideal for smaller memories and need significantly fewer logic cells to store data compared to flip flops. For both memory types, the write operation is synchronous. The read operation on distributed RAMs is asynchronous, unlike the synchronous read on block RAMs.

The use of a RAM type can either be instantiated manually by the HDL programmer or automatically by tools such as the Vivado synthesis tool. We opted to let Vivado choose the best RAM type for the extender array, as suggested by the Vivado synthesis documentation [8]. This keeps the VHDL source code flexible and portable. To let the extender array be automatically memory-optimized by Vivado, we had to remove the assignment during an asynchronous reset to make sure writing into the array only takes place synchronously. Through using the distributed RAM, the extender size decreased from around 1500 logic cells to just 300 logic cells.

Employing the distributed RAM for the extender with two rounds per cycle is a bit more complex because depending on the exact FPGA board, the distributed RAM might only have a single write port, but two entries need to be stored per clock cycle. To fix this problem, we decided to split the internal 16-entry extender array into two 8-entry

arrays, one for the former even indices and the other one for the odd indices. Algorithm 2 specifies this approach in greater detail. As a result, the extender size dropped from 2100 logic cells to 1000 logic cells.

Algorithm 2 Pseudocode of the extender with two rounds per cycle and two 8-entry message schedule arrays

Require: functions σ_0, σ_1 and padded chunk $M[0 \dots 15]$
1: ARRAY $W_A[0 \dots 7]$
2: ARRAY $W_B[0 \dots 7]$
3: **for** $t \leftarrow 0$ to 31 **do**
4: **if** $t < 8$ **then**
5: $W_A[t] = M[2 * t]$
6: $W_B[t] = M[2 * t + 1]$
7: **else**
8: $W_A[t \mod 8] = \sigma_1(W_A[(t-1) \mod 8]) + W_B[(t-4) \mod 8] + \sigma_0(W_B[t \mod 8]) + W_A[t \mod 8]$
9: $W_B[t \mod 8] = \sigma_1(W_B[(t-1) \mod 8]) + W_A[(t-3) \mod 8] + \sigma_0(W_A[(t-7) \mod 8]) + W_B[t \mod 8]$
10: **end if**
11: SEND_NEXT_WORDS_TO_COMPRESSOR($W_A[t \mod 8], W_B[t \mod 8]$)
12: **end for**

5 Correctness

While we implement different optimizations, it is important to verify that the correctness of the computation is maintained. This is done through a suite of automated and manual tests, which are run for each version.

5.1 Unit Testing

To test the correctness of single components, we run unit tests as GHDL test-benches to simulate common scenarios for those components. Those can be run manually, but are also executed as part of our CI/CD pipeline. For example, the compressor is tested with message schedule words equal to zero, and the output is compared to the correct output from a verified implementation. The extender is tested similarly, by verifying message schedule words for a given chunk. The comparator is tested to correctly detect hashes within a given threshold according to the Bitcoin specification. We have to note that the testing is performed only in simulation, so problems with the timing (like those of the four-round rolled-out compressor) are not tested at this stage.

5.2 Integration Testing

While Unit testing just shows the correctness for common scenarios for the individual components, integration testing allows us to check the whole chain of components and software used for mining.

Therefore the following aspects have to be checked

1. Are all mining cores running and able to find a block?
2. Communication: Host <-> Miner (Simulation/FPGA)

Regarding the first point, we test whether for several real block headers from the actual blockchain the correct nonces are found. However, this is not enough to show that all our mining cores are actually working. Due to the work distribution described earlier, the hash for the start nonce passed within the header will always be tried out on the first mining core. In order to show that all cores are running we have to force the miner to find the hash on a specific core. We do this by taking real Bitcoin block headers and decrementing the "correct" nonces of the headers by an offset. If, for example, we decrement the nonce by 2 and send it to the implementation, the passed nonce will be calculated on the first core, the next nonce (in this case start nonce + 1) on the second core and therefore the correct nonce (in this case start nonce + 2) on the 3rd core. By increasing this offset up to $\textit{number of cores} - 1$ we can show the correct functionality of all cores.

We run the described tests not only against the FPGA but also a GHDL simulation. This allows us to detect major problems already prior to the time intensive synthesizing step. With this technique we are limited to problems regarding the logic behind the miner. Since the simulation does not know anything about the FPGA, we are not able to find physical issues regarding timing and insufficient resources on the FPGA board. Such issues are either detected by a failing synthesizing step or if the tests ran after deployment do not work against the FPGA.

The correctness of the communication is indirectly shown by the tests above, because during execution the miner has to handle all of two request (see section 3.3).

6 Performance Analysis

Having introduced all these optimizations, it is now of interest to measure the performance improvements they yield. To achieve this, we begin by establishing meaningful measures of performance for our specific field. Next, we will record how our optimizations specifically influence these metrics. Finally, we will use these measurements to assess our architecture in comparison to other implementations.

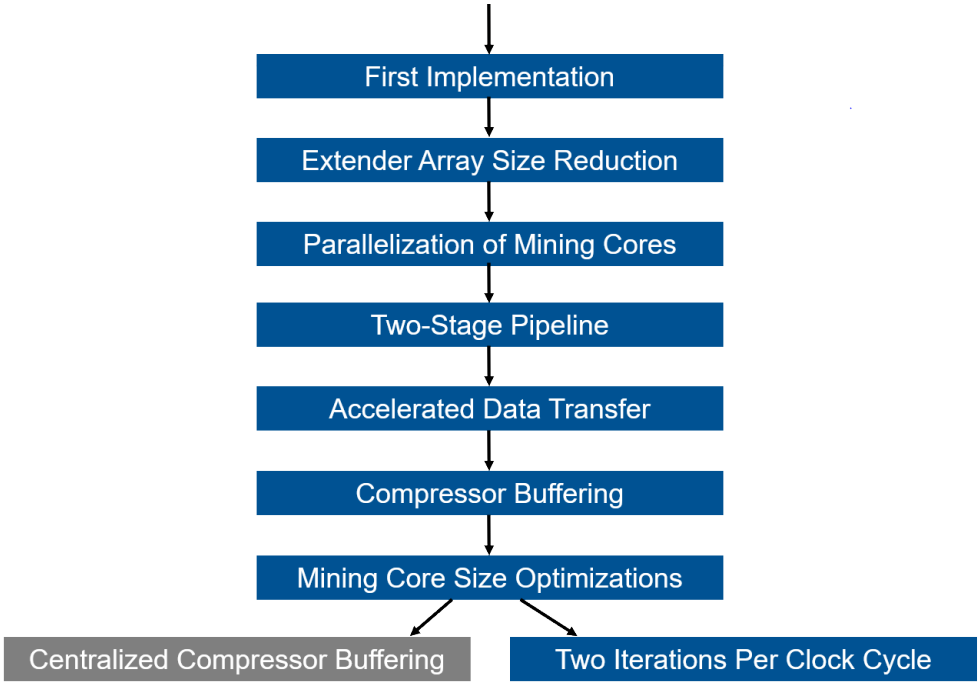

Figure 11: Conducted optimizations

6.1 Performance Definition

When comparing hardware for Bitcoin mining, several different metrics appear. The most common one being a measure of the calculation capabilities of the hardware in *Hashes/second*(H/s). Because we are in a very specialized field, the hashes are not actually SHA-256 digests, but SHA-256d block header digests.

However, fast calculation is not the only thing that makes mining profitable. Particularly the power consumption of the miner is crucial for the profitability. Therefore it is also worth looking at the miner's efficiency in terms of *Hashes/Joule* (H/J or H/Ws).

6.2 Performance Enhancements Through Optimizations

This section aims to evaluate the performance increase caused by the optimizations, which were introduced in chapter 4. As the optimizations were conducted during different stages of the project and have dependencies among each other, it is often difficult to determine to what extent an optimization increased the final hash rate when viewed in isolation. Therefore, each optimization is evaluated based on its predecessor in the order of implementation. Figure 11 gives an overview of the optimizations presented in this chapter and the conflicts between them.

The most important measurement to evaluate an optimization is the total hash rate, calculated as the product of the hash rate per core and the number of cores that fit on the board. We measured and optimized our implementation on a Digilent Artix-7 Arty A7-100T FPGA. Some features of this board are listed in table 1.

Logic Cells	Logic Slices	Flip-flops	Block RAM	DSP Slices
101,440	15,850	126,800	4,860 Kbits	240

Table 1: Features of the Arty A7-100T FPGA [1]

In general, the expected hash rate H of optimization can be calculated using the formula $H = \frac{F \cdot N}{C}$, where F is the clock frequency, N the number of mining cores on the FPGA board and C the number of cycles a core needs to calculate another hash digest. The first version of our Bitcoin mining algorithm was sequential and needed 394 clock cycles to calculate a new hash digest. These 394 cycles consist of $2 \cdot 3 \cdot 64 = 384$ cycles to pass the 64 rounds of the three SHA-256 stages when one round is calculated in two clock cycles, plus 10 cycles for internal communication overhead. With a 100 MHz clock frequency, this resulted in a total hash rate of $\frac{10^8 Hz}{394 \frac{1}{Hash}} \approx 0.25 MHash/s$.

Extender Array Size Reduction

The size reduction of the internal extender array from 64 entries to 16 entries, as described in section 4.3.2, decreased the size of a single extender from around 6300 logic cells to only about 1900 cells. As the extender was the largest component within the mining core, downsizing the extender array reduced the total implementation size from around 16400 occupied logic cells to about 7600 logic cells. This didn't immediately influence the hash rate as we only had one mining core so far, but it later enabled us to deploy twice as many mining cores on the FPGA board than before and thus theoretically doubled the hash rate.

Parallelization of Mining Cores

Through implementing logic to distribute the nonces between mining cores, the cores were able to work in parallel (see section 4.1.1). As seen in table 1, our FPGA board contains a bit more than 100000 logic cells. The Vivado reports showed that all components for the external communication combined had a size of 6000 logic cells and the cores themselves had a size of 7600 logic cells each. Consequently, 12 cores could be deployed on our target FPGA board, leading to an overall hash rate of 3.05 MHash/s, a twelvefold increase.

Two-Stage Pipeline

So far, a mining core always waited for the previous hash calculation to finish before starting with the next hash. As a result, only one of the two SHA-256 hashing stages was active at a time. The next conducted optimization was the implementation of a two-stage pipeline for the two SHA-256 hashing stages in a mining core. This optimization is explained in greater detail in section 4.2.1. As the first stage always had to process two padded chunks and the second stage just one, the first stage now reached 100% utilization and the second one 50%. As a result, each core was able to calculate a new hash digest every 264 cycles while the number of occupied logic cells and hence the mining core count stayed constant. This raised the total hash rate by 50% to 4.55 MHash/s.

Accelerated Extender-Compressor Data Transfer

By switching from the old communication protocol to a faster one, which doesn't wait for feedback (see section 3.2), the extender and compressor were able to transmit data every clock cycle instead of only every second clock cycle. This optimization reduced the number of clock cycles needed by a mining core to calculate a hash digest from 264 to 136 cycles, while not significantly changing the size of a core. Consequently, still with 12 mining cores on the FPGA board, the overall hash rate climbed to 8.82 MHash/s.

Compressor Buffering

Adding a buffer to the first stage of a mining core makes it possible to skip the calculation of the first padded chunk for all nonces except the first through a length extension attack (see section 4.2.2). As a result, the first stage now only had to calculate one padded chunk instead of two. After having additionally removed the communication overhead, the mining core was able to calculate a new hash digest every 64 cycles instead of every 136 cycles. The added buffer, however, increased the size of the first compressor by 700 logic cells from 1800 to 2500 cells. The total mining core size rose to about 9000 cells per core, leaving space for 11 cores on the board. The total hash rate went up to 17.2 MHash/s.

Mining Core Size Optimizations

This subsection summarizes the effects of various area optimization strategies presented in chapter 4.3.2. First, removing the buffers from the padder and comparator reduced the components' sizes from 350 to 2 cells and 260 to 7 cells. Second, swapping out the extender array into the distributed RAM cut the extender size down from 1500 cells to 300 cells. In addition, several smaller optimizations, for example, simplifying the

compressor, only returning the nonce for a found hash instead of the hash digest itself, and reducing the size of the external communication components further reduced the area of a mining core by almost 1500 logic cells combined. All in all, area optimizations decreased the size of a single mining core from 9100 to 4800 logic cells, enabling 21 mining cores to be deployed on an FPGA board instead of just 11. Consequently, the total hash rate grew to 32.8 MHash/s.

Centralized Compressor Buffering

Another optimization idea is to have a special first mining core, which is different from the other mining cores and responsible for the compressor buffering (see section 4.2.2). Subsequently, the other mining core's generators only receive the second 512-bit chunk of the block header and wouldn't need their own compressor buffer, which reduces the total occupied area. Having a shared compressor buffer for all mining cores indeed reduced the mining core size by about 300 logic cells to 4500 cells. Trying to reduce the size of the nonce generator, however, didn't influence the occupied area. As a result, we were able to deploy 23 cores on the FPGA board and the hash rate grew to 35.9 MHash/s.

Two Rounds per Clock Cycle

Calculating two rounds per clock cycle (see section 4.3.1) doubled the hashing speed of a single mining core, but had different effects on the mining core's components:

- The compressor size dropped from 1500 to 1400 logic cells, although now calculating two rounds per cycle, because some logic was moved into the extender. This helped to improve some slack violations.
- The size of the extender increased from 300 to 1000 logic cells due to the additional logic to calculate two rounds per cycle.

The effectiveness of this optimization, however, was also limited by the fact that it wasn't compatible with the shared nonce buffering because of timing problems. Implementing a buffer for each mining core again increased the occupied area slightly, but reduced the longest critical path length and fixed the timing issues. Thus, the total mining core size of this optimization increased from 4500 to 5900 logic cells. We were able to deploy 16 cores, that calculate a new hash digest every 32 cycles, leading to a overall hash rate of 50.0 MHash/s. Compared to the implementation with only one round per clock cycle but with shared buffering, this optimization achieved a higher overall hash rate.

The performance enhancements caused by the described optimizations are visualized in figure 12. All optimizations combined lead to a 200 times better performance. The final

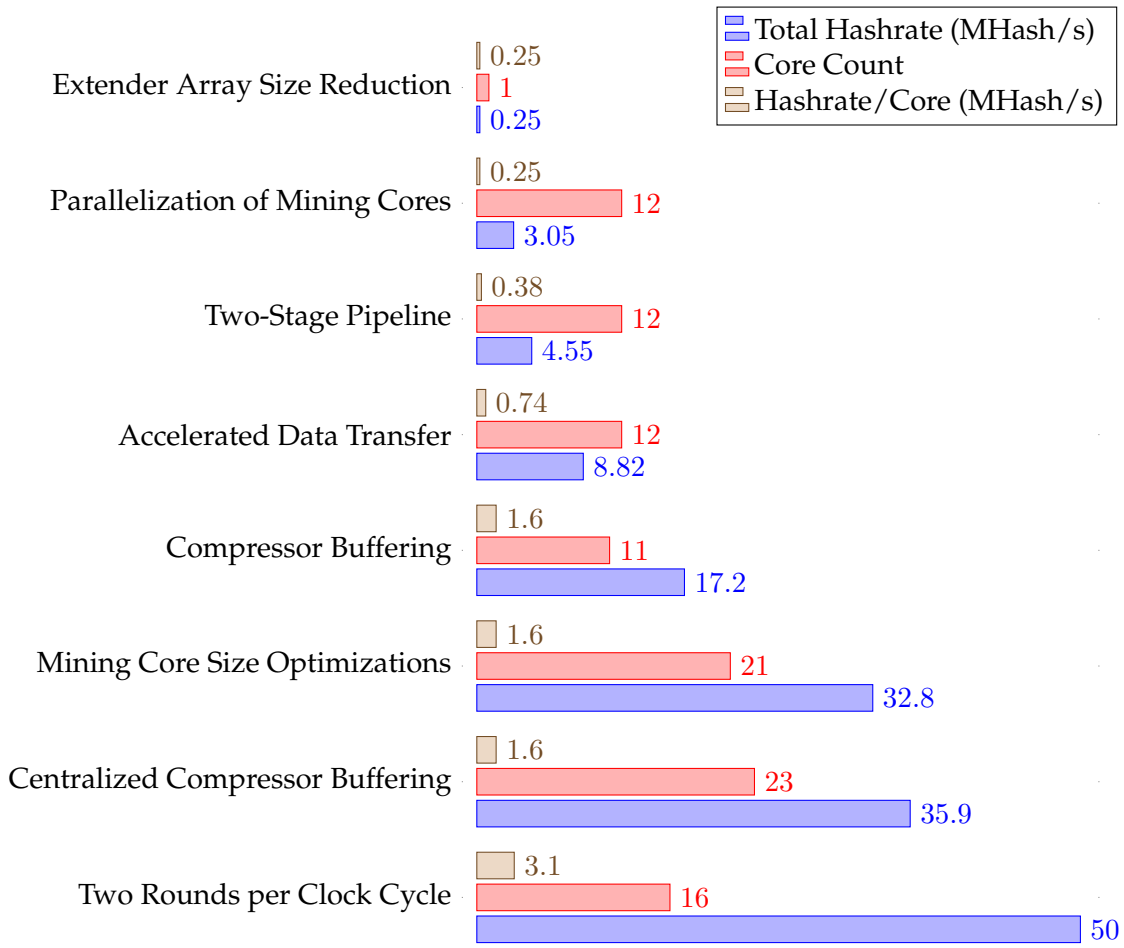

Figure 12: Comparison of optimizations

hash rate of 50.0 MHash/s could be verified in physical tests on the FPGA board with an accuracy of more than 99.8%[5].

6.3 Comparison With Other Implementations

In order to assess the significance of our implementation, it is crucial to compare its performance to other hardware. For this purpose we used a list provided by the bitcoin wiki[6] [7]. As one can see from figure 13 below, this paper shows, our implementation is capable to compete with CPUs and potentially GPUs with a larger FPGA board to host it. Unfortunately, this hashrate is minimal compared to that of an ASIC[6], as shown

[5]The inaccuracy was mainly caused by the time measurement of our Python test script.

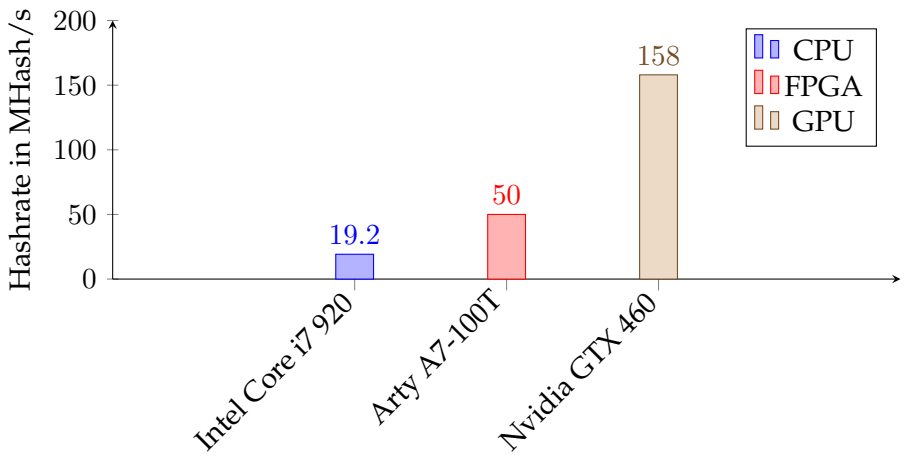

Figure 13: Comparison of hashrate of different hardware

in figure 14, shedding strong doubt on the potential of FPGA implementations being able to compete with ASICs regarding cryptocurrency mining.

Another aspect to account for in Bitcoin mining is the power consumption of the specified hardware in relation to the hashrate generated. Upon first inspection in figure 15, our implementation shines, with nearly 30 times the efficiency of a GPU. Once again, however, the here described implementation cannot compete with the efficiency of an ASIC as depicted by figure 16.

Additionally, it must be noted that the here-described implementation consumes 2.46 Watt regardless of whether it is computing useful hashes or calculating needless values while waiting for a new header. The power consumption was obtained from the static report generated by Vivado. The fact that the FPGA board might calculate needless hashes is acceptable in our use case, as the Bitcoin network has an uptime of 99.985% and was last down in 2013. Therefore, we can rationally assume that a header will always be available to mine for the FPGA, making the time in which it is not calculating minimal.

Now that we have estimated this implementation's electric efficiency, a final calculation that is of interest is the resulting economic efficiency. Therefore, we need the Bitcoin mining difficulty, which gives an indication of the difficulty to find a hash digest for the current target[7]. The average time needed to find a block with a given hash rate H and mining difficulty D is $\frac{D \cdot 2^{32}}{H}$. At the time of this writing, the Bitcoin mining difficulty is $20.6 \cdot 10^{12}$. Using this and our hash rate of 50.0 MHash/s, we can simply estimate the time needed to find one valid block hash [5]:

[6]This also explains the difficulty in finding mining data related to hardware more recent than the one presented here - due to the huge discrepancy in mining power between ASICs and other hardware, very few people still mine using CPUs and GPUs.

[7]More information on the Bitcoin mining difficulty can be found at [5]

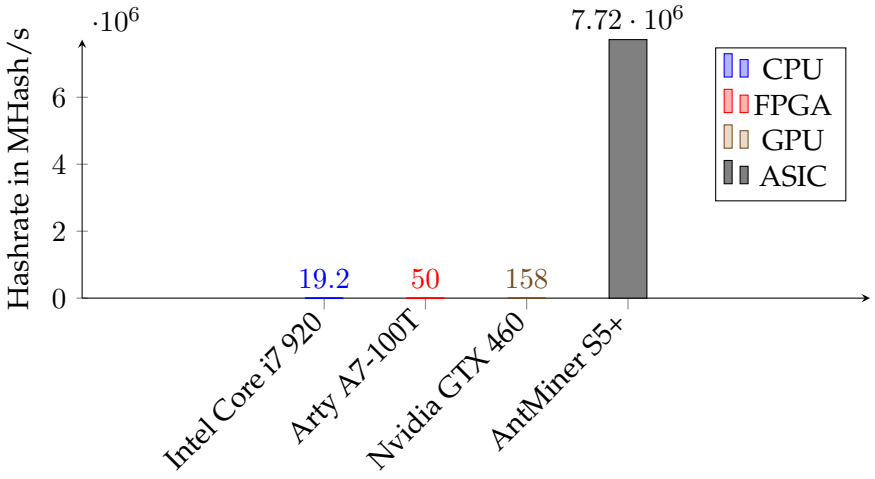

Figure 14: Comparison of hashrate of different hardware including ASIC

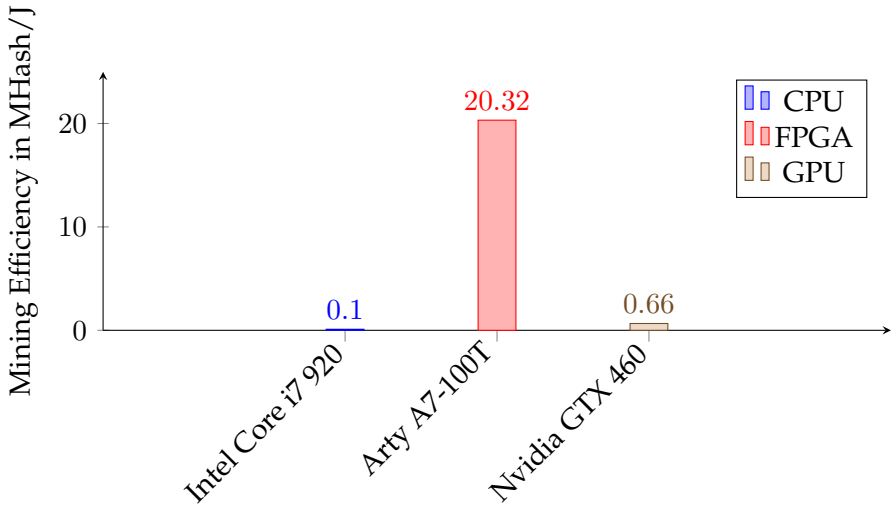

Figure 15: Comparison of mining efficiency of different hardware

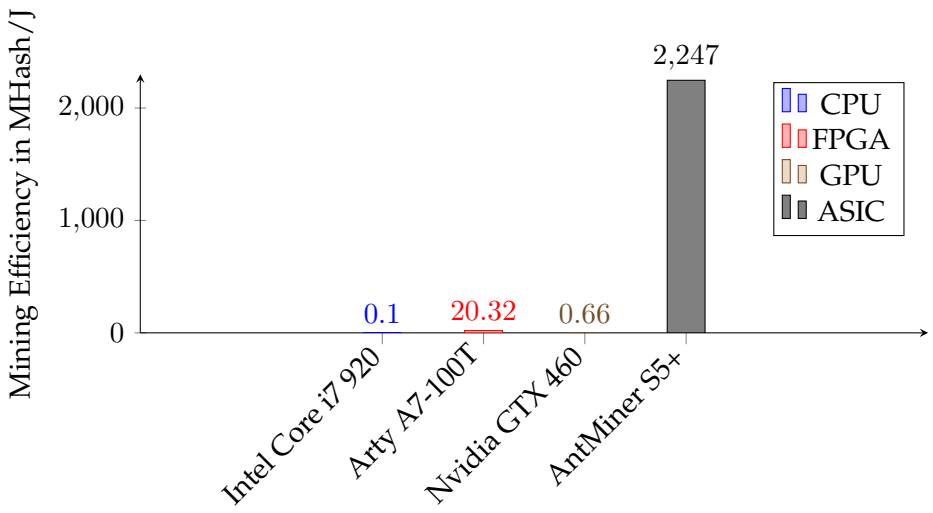

Figure 16: Comparison of mining efficiency of different hardware including ASIC

$t = \frac{20.6 \cdot 10^{12} \cdot 2^{32}}{5.00 \cdot 10^{7}}$ seconds/block $\approx 1.77 \cdot 10^{15}$ seconds/block $\approx 5.61 \cdot 10^{7}$ years/block

At the time of this writing, the Bitcoin network rewards the miner with 6.25 Bitcoin for mining a block. This means this implementation mines at a rate of $\frac{5.61 \cdot 10^{7}}{6.25}$ years/bitcoin $\approx 8.98 \cdot 10^{6}$ years/bitcoin $= 0.0898$ years/satoshi[8]. Alternatively written, one can expect to make approximately 11.13 satoshi/year. At the current Bitcoin price of 36.300 USD, this is equal to approximately 0.004 USD/year. However, these numbers do not account for the cost of electricity. Factoring this in, our implementation burns through $365.25 \cdot 24 \cdot 2.46$ kWh/year ≈ 21.55 kWh/year (with the power consumption of this implementation being $2.46W$) if running full-time. The profitability of this implementation in dependence of the electricity price is visualized in graph 17. Thus, the electricity price at which we break even is $1.86 \cdot 10^{-4}$ USD/kWh.

7 Summary

To sum up, we have seen that Bitcoin makes use of blockchain technology to avoid double spendings and unwanted modifications of the transaction history. The Bitcoin blockchain itself uses a SHA-256d hash which has to fall below a specified target as Proof-of-Work. Consequently, the purpose of Bitcoin mining is to calculate the hashes of slightly altered block headers with brute-force until one of these hashes drops below the target. Our project aimed to implement such a Bitcoin mining algorithm on an FPGA board and optimize its efficiency.

[8]Satoshi is the smallest piece to which a bitcoin can be broken down, with one bitcoin being equal to 100 million satoshi.

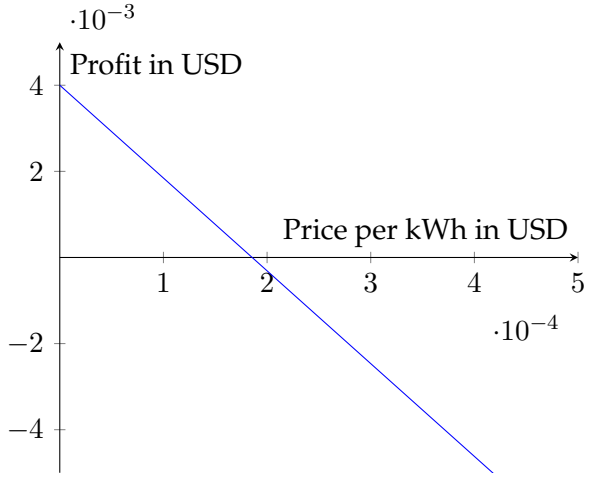

Figure 17: Profitabiliy depending on the price of electricity

To do so, we structured our implementation in mining core entities, that can calculate hashes in parallel. The 64 rounds of the SHA-256 algorithm are performed by padder, extender and compressor components. A mining core entity combines two of these SHA-256 stages with a nonce generator and a comparator. Our optimizations mostly focused on the extender and compressor components, as they were the bottleneck in terms of throughput and occupied the largest area on the FPGA board.

In the previous sections we have seen various optimizations that increased the performance of a Bitcoin Proof-Of-Work implementation significantly. On the algorithmic side, we were able to calculate two rounds per clock cycle and to exploit the length extension vulnerability of SHA-256. On the hardware-specific side, we improved the timing to speed up the hash rate of a single mining core and improved the area utilization to fit more mining cores onto the FPGA board. Some of the performed optimizations had to be weighed against others because they weren't compatible.

Before measuring the effectiveness of optimizations, it is especially important to validate their correctness. Our CI pipeline executes simulation unit tests for the hash components and for the whole mining core entity. Furthermore, automatic deployment tests are conducted, which check that no timing-related problems occur.

All in all, we were able to reach a hash rate of 50.0 MHash/s on a Diligent Arty A7-100T board, compared to 0.25 MHash/s without any optimizations. 50 MHash/s is better than most CPUs and not too far below the hash rate of most GPUs. When compared to CPUs and GPUs, our implementation especially shines in terms of its low power consumption, resulting in considerably better mining efficiency. However, the hash rate of an FPGA - as expected - is substantially less than that of a Bitcoin-mining ASIC, making FPGA mining quite inefficient. Thus, even connecting a larger FPGA board to the Bitcoin network wouldn't be economically viable.

The described optimizations are nevertheless relevant because most of them can also be applied when designing a Bitcoin mining ASIC. Additionally, we have shown that iterative hashing algorithms can have a significantly better hashing efficiency on FPGA boards than on CPUs and GPUs. This might be particularly interesting for smaller or new Proof-Of-Work based cryptocurrencies, for which ASIC miners aren't available.

References

[1] Digilent. Arty a7 reference manual. https://reference.digilentinc.com/reference/programmable-logic/arty-a7/reference-manual (last visited on 22.01.2021).

[2] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 437–455, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (last visited on 17.01.2021), October 2008.

[4] National Institute of Standards and Technology (NIST). FIPS 180-2: Secure Hash Standard (SHS). https://csrc.nist.gov/CSRC/media/Publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf (last visited on 29.12.2021), August 2002.

[5] Bitcoin Wiki. Bitcoin mining difficulty. https://web.archive.org/web/20201224080905/https://en.bitcoin.it/wiki/Difficulty (last visited on 18.01.2021), December 2020.

[6] Bitcoin Wiki. Bitcoin non-specialized hardware comparison. https://web.archive.org/web/20201112023816/https://en.bitcoin.it/wiki/Non-specialized_hardware_comparison (last visited on 18.01.2021), November 2020.

[7] Bitcoin Wiki. Bitcoin mining hardware comparison. https://web.archive.org/web/20210111232459/https://en.bitcoin.it/wiki/Mining_hardware_comparison (last visited on 18.01.2021), January 2021.

[8] Xilinx. Vivado design suite user guide - synthesis. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf (last visited on 22.01.2021), November 2016.

List of Tables

List of Figures