UCLouvain

École polytechnique de Louvain

Mining for blockchains using commodity hardware

Author: **Adrien Ballet**
Supervisors: **Olivier Pereira, François-Xavier Standaert**
Readers: **Ramin Sadre, Itamar Levi**
Academic year 2018–2019
Master [120] in Computer Science and Engineering

Abstract

One of the most important goal of a blockchain is its decentralized aspect. The growing popularity of ASICs for solving common proof-of-work algorithms has led to the opposite effect: blockchain mining industries are now centralized in countries where energy is cheap. New cryptocurrencies based on memory-hard proof-of-works have been introduced to counteract the use of ASICs and favor commodity hardware. This work explores the implementation of such algorithms on FPGAs, in a context of reclaiming the overflow energy of solar panels. One of them, Equihash, is deeply studied and described. A comparison with an optimized CPU version is provided, as well as a feasibility analysis. The objective is to determine how well an FPGA performs in this context.

Acknowledgements

This dissertation concludes a five year-long journey at the Ecole Polytechnique de Louvain, on the road to becoming an engineer. I would like to thank every person that contributed to this enriching academical and personal experience. In particular, the persons who helped me to carry out this Master thesis.

First and foremost, I would like to express my gratitude to my supervisors, Olivier Pereira and François-Xavier Standaert, for their help and guidance throughout the year. They never stopped giving me the motivation I needed.

A special thank you goes to Itamar Levi, for his helpful advices. His availability and patient explanations about electronics were a key factor in the realization of this project.

Finally, I would like to thank all my loved ones, friends, family, and girlfriend, for their interest and support.

Contents

List of Figures

Glossary

ASIC Application Specific Integrated Circuit. 8

baud rate Number of symbols per second that occurs within a data transmission. 22

CPU Central Processing Unit. 8

DDR SDRAM Double Data Rate Synchronous Dynamic Random Access Memory. 10

DSP Digital Signal Processor. 10

FPGA Field-Programmable Gate Array. 8, 10, 21

GPU Graphics Processing Unit. 8

JTAG Joint Test Action Group. 21

PoW Proof of Work. 5

UART Universal Asynchronous Receiver Transmitter. iii, 19, 20, 22

USB Universal Serial Bus. 20

Chapter 1

Introduction

The blockchain was introduced with the first cryptocurrency, the Bitcoin, in 2008. In the famous paper, new revolutionary ideas were presented: a system that allows anyone to exchange money with anyone, over the Internet. A system that takes advantage of the community to secure the transactions. And primarily, a system that works in a decentralized way, without any third party.

Ten years and hundreds of new cryptocurrencies later, the decentralized aspect of the blockchain is in danger: instead of the community, industries are taking over the mining process. They aim for ever-increasing profitability, and therefore are looking for the best hardware, and the best place to operate.

A recent study has shown that more than 74% of the Bitcoin hashing power comes from China[16], where energy is cheap. Far from the original decentralized ideology of the blockchain. The mining takes place on dedicated hardware, as ASICs are faster and more energy-efficient.

On the other hand, far from the land of the rising sun, Belgians worry about a different problem. Many are those who installed solar panels to provide their home with renewable energy. But the government allowances for green energy are decreasing, and it is more difficult to give back the overflow energy on the electric grid.

What if we could connect these two issues? This is the context of this work: use the *free overflow energy* of solar panels, to empower back the blockchain community by decentralizing the mining again, in the homes of the people.

Some new cryptocurrencies have the very same purpose regarding the decentralization of their mining community. They are using a novel type of proof-of-works, called *memory-hard*, aimed at making the mining on ASICs an obstacle, or at least,

not much more efficient than commodity hardware. We will present these new algorithms, and choose one that could fit our needs.

Even if the overflow energy comes from a free renewable source, it should be used wisely. The kind of mining hardware chosen in this work is an *FPGA*, for his energy-efficient and reprogrammable properties. At the verge of dedicated and commodity hardware, an FPGA could be the best of both worlds.

Thereby, we will explore the implementation of a memory-hard proof-of-work on an FPGA, try to improve it, and review its efficiency compared to a more classic CPU version. Remembering the context, we will also determine if the chosen algorithm could lead to profitability.

Chapter 2

Background

The functioning of cryptocurrencies is briefly described, as well as its strong link with the blockchain proof-of-work, and hash functions.
The different hardware devices used to *mine for blockchain* are also reviewed, and some more detailed explanations are given about the functioning of the hardware of interest, the FPGA.
Memory-hardness is introduced, and the choice of the considered algorithm is argued.
Finally, the chosen algorithm is described.

Contents

2.1 Cryptocurrencies

The primal vision of a cryptocurrency, as described in the original Bitcoin paper[19], is the following:

"A purely peer-to-peer version of electronic cash [that] would allow online payments to be sent directly from one party to another without going through a financial institution."

In other words, we look for a method to exchange a currency in a decentralized and secure way. Satoshi Nakamoto, publisher of the Bitcoin paper, came up with the first description of the *blockchain* in 2008. An overview is provided in sections 2.1.1 and 2.1.2.

2.1.1 Decentralized transactions

Since there is no central authority to keep track of the transactions, the users of this payment system have to broadcast their spendings to each other, and save the history of trades themselves[1]. Transactions are typically signed using a cryptographic secret key/public key pair to prevent an unauthorized user to spend someone else's assets[13, 19].

A classic example of broadcast is the following : Alice intends to give 20$ to Charlie. She writes the transaction in her ledger, and notifies the other users. But what happens if Alice is adversarial, and sends another transaction of 20$ to Bob, while she only has 20$, and not the 40$ normally required for such a move?

This is known as the *double spending* problem. Clearly, we need a way to achieve consensus so that everyone can verify the transactions and be reasonably sure that they are legit.

2.1.2 Blockchain and proof-of-work

The idea to achieve consensus is the *blockchain*. The basic protocol is described in Nakamoto's paper[19]:

1. New transactions are broadcast to all nodes.
2. Each node collects new transactions into a block.
3. Each node works on finding a difficult proof-of-work for its block.

[1] In practice, one usually does not need to store the complete history of transactions, thanks to the use of Merkle trees[19].

4. When a node finds a proof-of-work, it broadcasts the block to all nodes.

5. Nodes accept the block only if all transactions in it are valid and not already spent.

6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

The key principle is to compact a fixed number of transactions in a block, and make the users *mine* that block, i.e. do a computationally intense task in order to validate the block. This is known as a *proof-of-work* (PoW).

Hash function

Such function is used to map an input message of any length to a fixed length n output[33].

$$h : \{0,1\}^* \to \{0,1\}^n$$

For an application as a proof-of-work, the hash function must have some properties: it must be deterministic, its output space must be uniformly distributed, and it must be resistant to collisions, i.e. it is hard to find a, b such that $h(a) = h(b), a \neq b$.

Proof-of-work

The *mining* process makes extensive use of a hash function. A random nonce is added alongside the transactions, and the whole block is fed to the cryptographic function.

The n-bits output of the hash function may be seen as an unsigned integer. This integer can then be compared to a *target value*. The goal of the miner is to find a nonce such that the output of the hash function is below a defined target value. If the comparison is not successful, a new random nonce is generated, and the process is repeated.

The target value is defined in such a way that each new block is mined after a fixed average time interval, usually a few minutes. As the number of miners increases, the target value decreases to make the proof-of-work harder and more time-consuming.

Blockchain

The blockchain uses the proof-of-work to ensure the integrity of the transactions[20]. Its principle is the following: the valid hash of the previous block is included in the next one, forming a chain (figure 2.1). If multiple miners find valid nonces at the

same time, there are multiple valid branches of the chain coexisting, until other miners complete the blockchain: sooner or later, one of the forks will be longer than the others, and it is always the longer one that is accepted by the community.

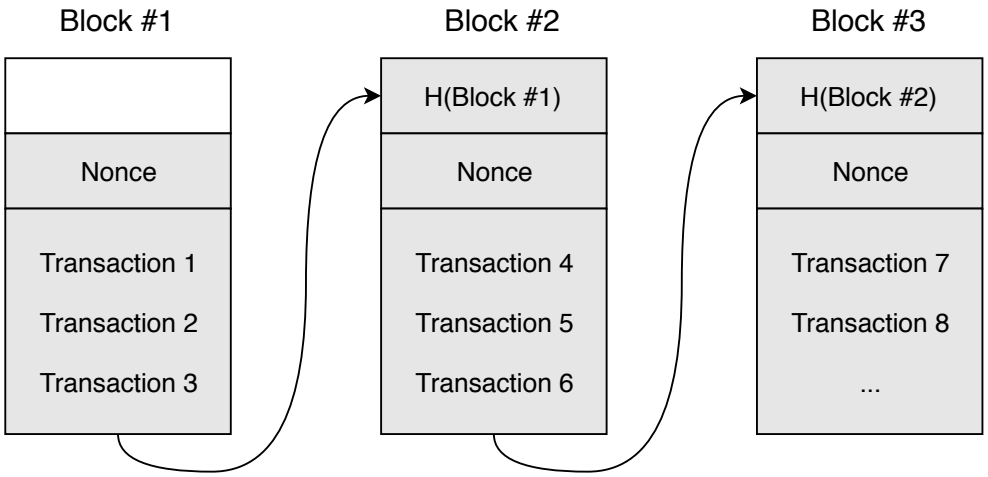

Figure 2.1: Simplified blockchain schematic

Should some miner want to modify a previous block, not only does he have to find a valid hash for that block, but also for all the blocks that were issued after that, because the new hash value would make them invalid.

This effectively prevents the *double spending problem*, as a malicious user has to have more processing power than half of the network to let the other users think that the corrupted fork is the genuine one. This is known as the *51% attack*. A newly issued transaction is only considered valid after a few new blocks are mined, because the probability of a user being able to forge the blockchain diminishes as the chain becomes longer.

Incentive to mine

The system is secure as long as a sufficient amount of users engage in the mining process. The last piece of the system is thus to add an incentive for mining. This can be achieved is two ways:

1. **Block reward:** few units of the cryptocurrency are given to the successful miner of a block. These units are not exchanged, they are created: this is how the number of units on the market increases. However, this number is usually capped, the cryptocurrency is a finite resource.

2. **Transaction reward:** as the miners chose which transactions are included in their current block, and the number of transactions per block is fixed,

the user that issues the transaction may add a small reward for the miner so that its block will be chosen as a priority. Once all the units of a given cryptocurrency are created, this will be the only way for a miner to be rewarded.

2.1.3 Applications

The blockchain is described here in its original context: cryptocurrencies. But its fundamental properties of storing and verifying information in a decentralized and secure way are interesting in many other domains. Here's a brief overview of potential other applications[21, 23]:

- **Smart contracts:** no more third party involved in ensuring that each stakeholder follows the terms, and that they know about the contract details.
- **Health system:** the data may be stored encrypted, and only a private key would be required to access it.
- **Degrees certification:** any company or university could easily verify that an applicant's degree is legit.
- **Internet of Things:** this is another version of the smart contracts: a network of devices could exchange information and trigger actions without any third party[29].

2.1.4 Limitations

This first version of the blockchain has some issues:

- The number of transactions per unit of time is limited. Indeed, the protocol enforces a fixed number of transactions per block, and adjusts the rate of mined blocks.
- The amount of data stored in a blockchain is limited. Because of its decentralized nature, there is no single source of storage for the blockchain, and broadcasting huge amount of data is impractical.
- The energetic cost of the computationally intensive proof-of-work. "The Bitcoin network can be estimated to consume at least 2.55 gigawatts of electricity currently, and potentially 7.67 gigawatts in the future, making it comparable with countries such as Ireland (3.1 gigawatts) and Austria (8.2 gigawatts)."[8].

- The size of the network: the robustness of the blockchain depends on its large and distributed miners base. And even with a large network, the 51% attack still exists: it's highly improbable, but not impossible.

There are a lot of research and experimentations currently conducted to overcome some of these problems. Some simply involve a simple change of protocol, e.g. to increase the rate of transactions. Others imply a completely different consensus scheme, such as *proof-of-stake*[5, 24].

2.2 Mining hardware

As explained in sections 2.1.2, 2.1.4, the blockchain's proof-of-work is computationally- and energy- intensive. Therefore, miners are constantly looking for the best hash-rate/energy ratio.

Figure 2.2 shows the spectrum of hardware used for mining. From left to right, the hardware becomes more dedicated to the mining purpose, and less suitable for general purpose computations.

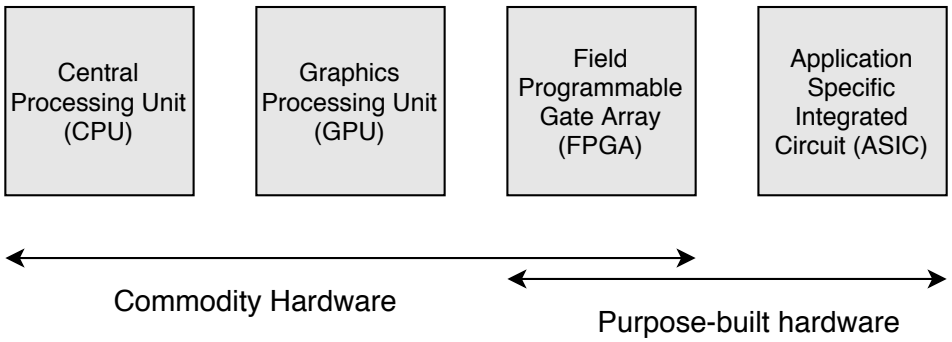

Figure 2.2: Blockchain mining hardware spectrum

The first implementation (January 2009) of a mining software for Bitcoin (and thus for cryptocurrencies in general) was meant to run on a CPU. Shortly after, in October 2010, a GPU implementation was released[30]. GPUs were the first milestone in terms of efficiency: as the difficulty to mine increased, so did the need for more dedicated hardware. By June 2011, FPGAs attracted the miners community, and soon enough, this led to the development of ASICs.

We distinguish *commodity hardware* from *purpose-built hardware*: "Commodity hardware is a computer device that is relatively inexpensive, widely available and

basically interchangeable with other hardware of its type. Unlike purpose-built hardware designed for a specific IT function, commodity hardware can perform many different functions."[28] CPUs and GPUs clearly belong to the first category.

ASICs are built for a very specific task, they belong to the second category. Well-engineered ASICs are also several orders of magnitude faster than CPUs, and require the least energy for a given task.

However, they are very costly to design, and obviously cannot be used for a different chore. ASICs are mostly used by the industry, rather than by individuals.

2.2.1 Field-Programmable Gate Array (FPGA)

One type of hardware stands at the edges of both categories: FPGAs. Such device is made of thousands of slices, each containing a few logic cells (figure 2.3)[37].

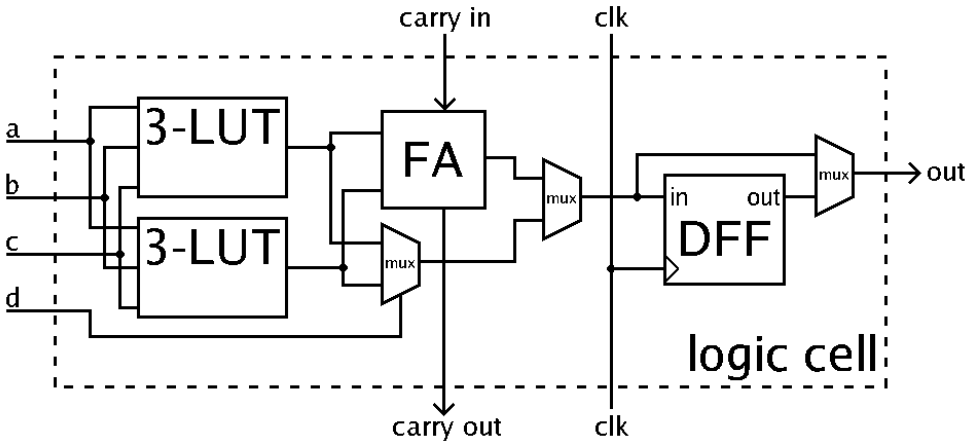

Figure 2.3: Schematic of a logic cell in an FPGA

A logic cell is typically made from:

- A n-input Look-Up Table
- Multiplexers
- Carry logic
- A Flip-Flop

By connecting thousands of these cells together, and modifying the values of the LUT, we can actually reroute the design and virtually implement any function as long as it fits in the hardware components.

This is why an FPGA stands in both categories: it can be seen as an editable circuit, designed for a defined task, but capable to be totally reprogrammed as desired for another chore.

FPGAs are not only made of logic cells, they usually also include some specific components such as BlockRAM to efficiently store and retrieve data, DSPs to help with signal processing applications, various LEDs, buttons, and switches. They may also have IO capabilities to interact with other pieces of hardware: DDR SDRAM, USB port, Ethernet port...

We understand here why FPGAs are worth of interest in the cryptocurrency-mining field: they borrow the efficiency of ASICs, but can be reprogrammed if necessary, and they are cheaper than the whole ASIC development cycle.

2.3 Memory-hardness in cryptocurrencies

The disparity between CPUs and ASICs exposed in the previous section is a threat for the blockchain model: if a well-designed ASIC is released on the market, chances are that mining industries will take advantage of that to increase their profit, and base their mining setup in a country where energy is cheap. As an example, the SHA-256 hash function used by Bitcoin is widely available as an ASIC, and 74% of the Bitcoin hashing power comes from China[16].

This is far from the original decentralized ideology of the blockchain. In order to loosen the advantages of the ASICs, new hash functions are explored. The key idea comes from the memory: fast and large storage hardware is expensive on ASICs. By making an extensive use of memory accesses in an algorithm, the efficiency of dedicated hardware should be reduced to the speed of the memory used.

2.3.1 Memory-bound algorithms

Assume $T(\alpha) = M(\alpha) + C(\alpha)$ is the total time taken by an algorithm to terminate for an input α, where M is the time spent in memory accesses, and C is the time needed to perform other computations. A *memory-bound* algorithm is an algorithm where

$$M(\alpha) >> C(\alpha), \forall \alpha$$

One may wonder about parallelism on ASICs. Indeed, computations might then be done in parallel, but they are negligible compared to memory accesses. Parallel

memory accesses are not a threat as long as the algorithm is devised in such a way that multiple workers cannot work with distinct zones of the memory. They will reach the maximum bandwidth of the memory, thus parallelism does not give a significant advantage.

With this in mind, memory-bound functions were introduced in [1]. The memory is written with a quite large array, that is later accessed in a pseudo-random way multiple times. The bandwidth of the memory would then become the bottleneck of the computation.

The *quite large* notion must be clarified: the memory needs must be large enough so that on-chip embedded memory should not be considered on ASICs. If both hardwares use the same kind of memory component, and the algorithm is bound by the maximum bandwidth, the performance of the controlling hardwares, whether they are CPUs or ASICs, should be comparable.

We only discussed the efficiency in terms of time. There is another aspect to bear in mind when comparing hardware: energy. ASICs maintain this advantage compared to commodity hardware: controlling the memory is still more energy-efficient on ASICs, unless the storage needs are such that the energy needed to power the memory makes the energy needed by the controller negligible. [18] shows that a recent 8GB DDR4 SDRAM chip needs 1.63W, while measurements on a 2.4 GHz Intel Core i5-4258U processor show a power consumption of 17.5W, with a possible peak up to 28W according to [14]. The amount of memory needed to significantly outgrow the consumption of the controller is very high, and this would be an obstacle to decentralization too. We can thus expect ASICs to maintain an advantage in terms of energy, even with memory-bound algorithms.

2.3.2 Memory-hard algorithms

Memory-bound algorithms seem convincing, but there is a flaw in the definition: what if we could decrease the memory needed, with only a small time tradeoff? ASICs could then progressively regain advantage. We need to constrain the time-space tradeoff with a new definition: *memory-hardness*, introduced in [22].

Memory-hard algorithms ensure a steep space-time tradeoff, i.e. reducing the memory needs demand a high time tradeoff (we are here talking about the total time, not only computation time). This is explained more formally in [4]:

Let $T_R(M)$ be the average running time of an algorithm, with memory M, and M_0 be the amount of memory needed on a standard implementation. Assume an

adversary uses only $\frac{M_0}{q}$ with $q > 1$. The running time growth can be expressed as:

$$C_R(q) = \frac{T_R\left(\frac{M_0}{q}\right)}{T_R(M_0)}$$

The time-space tradeoff is polynomial with steepness s if $C_R(q)$ can be approximated by a polynomial of q of degree s. The tradeoff is exponential if $C_R(q)$ can be approximated by an exponential of q. Notice the special case of a polynomial steepness with degree 1. The tradeoff is then linear: if the memory is divided by q, the time is multiplied by q.

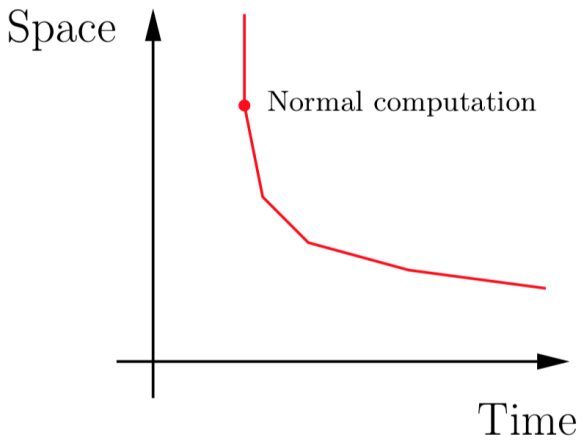

Figure 2.4: Visual representation of the time-space tradeoff

A memory-hard algorithm is thus also constrained by the space-time tradeoff, and a high steepness is desirable (at least linear to be considered memory-hard). Figure 2.4 show a visual example of a time-space tradeoff: decreasing the space implies an increase of the time, and the steeper the curve, the better, to prevent an attacker to efficiently use fewer memory than devised in the standard implementation.

2.3.3 Commodity-hardware friendly cryptocurrencies

Some cryptocurrencies have the goal of keeping their miners decentralized. They prioritize commodity hardware to make mining easily accessible, and try to exclude ASIC users from the mining community. This is were memory-hard algorithms come into play: since they reduce the efficiency of ASICs, the costly development

of such platform is discouraged.

Should an ASIC effectively be launched, a small change in the algorithm could break them instantly, where commodity hardwares simply need to be updated to keep working. An FPGA could use its IOs capabilities to interact with SDRAM for the high memory requirements, while being dedicated to this task and more energy-efficient, and its reprogrammable feature tackles a possible algorithm change in the future.

We are looking for a cryptocurrency that uses a memory-hard algorithm as a proof-of-work. The 50 most exchanged cryptocurrencies were identified on March 4, 2019. The complete comparison is provided in appendix A.1, under table A.1. Out of these 50 algorithms, we only consider the ones with a proof-of-work consensus scheme, and memory-hard algorithms:

- Litecoin (#5), Scrypt algorithm
- Monero (#11), Cryptonight algorithm
- Zcash (#17), Equihash algorithm
- Dogecoin (#22), Scrypt
- Bitcoin Gold (#23), Equihash
- Aeternity (#35), CuckooCycle algorithm

Four algorithms are thus yet to be compared.

Scrypt was introduced with the notion of memory-hardness in [22]. Its memory needs may be modified with parameters, allowing the user of the algorithm to define the best memory usage for its application. It was used as a proof-of-work when Litecoin launched in 2011. The creators of that cryptocurrency chose parameters such that the memory needs are 128KB[39]. This low value has an explanation: the verification process of the block needed the same space as the mining process. 128KB of memory seemed acceptable at the time, but ASICs designers were able to take advantage of this small amount of memory to devise dedicated hardware that was at least 100 times more efficient than GPUs. It is thus a failed attempt at avoiding ASICs, even with a theoretically memory-hard algorithm.

CuckooCycle [32] goal is to find a L-length cycle in a large bipartite graph, defined from a nonce. An interesting property is that although the search for such a cycle is memory-intensive, the verification is almost immediate. It is claimed to

be memory-bound, but memory-hardness is not deeply studied in [32]. The place of Aeternity in the chart also makes it less attractive.

Cryptonight's [9] idea was to use a memory scratchpad that could fit in the L3 cache of processors, around 2MB. It requires to fill the scratchpad with pseudo-random data, then perform read and write operations on deterministic addresses. Finally, the whole scratchpad is hashed to provide the final value. Instead of trying to disadvantage ASICs at first by using a huge amount of memory, here the algorithm benefits from the characteristics of CPUs. It was eventually broken as ASICs became available to mine this hash function. Monero implements the third iteration of CryptoNight, changing the internal functioning of the algorithm.

Equihash [4] is another promising memory-hard proof-of-work. It requires to find colliding strings based on Wagner's algorithm to solve the Birthday problem. It is asymmetric, in the sense that computation is time- and energy-demanding, but verification is almost instantaneous. Memory-hardness is studied in the publication. Its parameters can be easily modified to change its time and memory requirements. The cryptocurrencies using Equihash are in a good place in the chart, and Bitcoin Gold claims to change the parameters of the algorithm if needed to fight ASICs.

Scrypt in its current form is too vulnerable to ASICs, and CuckooCycle is not as well-spread as the two next algorithms. Cryptonight and Equihash were both valid candidates. The ease of verification with Equihash, its memory properties, and the convenient parameters are the reasons why this algorithm was chosen in this work. Equihash is described in section 2.4.

2.4 Equihash algorithm

2.4.1 In theory

Designed by Biryukov and Khovratovich in 2015 and described in [4], Equihash is an asymmetric memory-hard proof-of-work based on the generalized birthday problem, formulated as:

Given $2k$ lists L_j of n-bits strings $\{X_i\}$, find distinct $\{X_{i_j} \in L_j\}$ such that $x_{i_1} \oplus x_{i_2} \oplus ... \oplus x_{i_{2k}} = 0$.

Wagner proposed an algorithm to solve the problem in [35]. His idea is to apply a join operator on lists pair-wise, taking only into account the least significant bits l of each element. i.e. $L_{12} = L_1 \bowtie_l L_2$ is composed of the items of L_1 and L_2 that

collide on their first l bits (see figure 2.5).

1. Sort by least-significant bits $\frac{n}{k+1}$
2. Store XOR of collisions
3. Repeat for next $\frac{n}{k+1}$ bits.

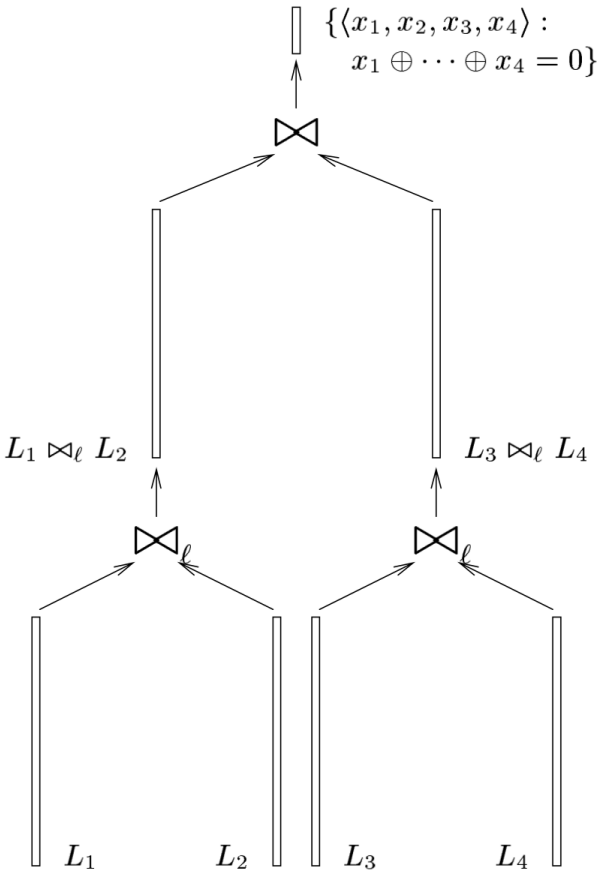

Figure 2.5: Example of Wagner algorithm for $k = 4$

Equihash defines a subtly different problem, with only one list, and parameters n and k:

Given a list of $2^{\frac{n}{k+1}+1}$ n-bits values, find 2^k items so that the resulting values XOR to zero, using Wagner's algorithm and taking $l = \frac{n}{k+1}$. The list is filled with

the n-bits output of a hash function h that takes as input a seed I, a random nonce V, and an integer:

$$L = \{h(I, V, x_i) | 1 < i < 2^{\frac{n}{k+1}+1}\}$$

The solution is then the list of 2^k integers x_i such that the output values of $h(I, V, x_i)$ collide.

The problem with this definition comes from the fact that one could easily choose a different value of l in the algorithm to obtain different solutions. Interestingly, the designers of Equihash found that the solution of the problem reveals how it was found. Indeed, a solution found with Wagner's algorithm shows a verifiable pattern since we can show that the first l least-significant bits that are colliding together (see figure 2.6).

$$\underbrace{\underbrace{H(x_1) \oplus H(x_2)}_{\text{equal in } \frac{n}{k+1} \text{ bits}} \oplus \underbrace{H(x_3) \oplus H(x_4)}_{\text{equal in } \frac{n}{k+1} \text{ bits}} \cdots \oplus H(x_{2^k})}_{\text{equal in } \frac{2n}{k+1} \text{ bits}} = 0.$$

Figure 2.6: Wagner's algorithm solution

The complete Equihash proof-of-work must thus verify three conditions (figure 2.7):

1. The given 2^k integers are valid, i.e. their hash values do effectively collide.
2. The hash of the whole solution passes a difficulty filter.
3. The solution shows the pattern of Wagner's algorithm.

If the solution does not pass the difficulty filter, or if no solution was found, the random nonce has to be modified and the algorithm must loop again to the list generation (figure 2.8).

Regarding the memory-hardness of the algorithm, it is shown in [4] that if we divide the memory by q, the running time growth can be expressed as:

$$C(q) \approx \frac{3q^{\frac{k-1}{2}} + k}{k+1}$$

Thus Equihash has a tradeoff that is polynomial with steepness $k - 1$.

$$H(I||V||x_1) \oplus H(I||V||x_2) \oplus \cdots \oplus H(I||V||x_{2^k}) = 0. \quad (1)$$

$$H(I||V||x_1||x_2||\cdots||x_{2^k}) = \underbrace{00\ldots0}_{q \text{ zeroes}} * * * * . \quad (2)$$

$$\underbrace{\underbrace{H(x_1) \oplus H(x_2)}_{\text{equal in } \frac{n}{k+1} \text{ bits}} \oplus \underbrace{H(x_3) \oplus H(x_4)}_{\text{equal in } \frac{n}{k+1} \text{ bits}}}_{\text{equal in } \frac{2n}{k+1} \text{ bits}} \cdots \oplus H(x_{2^k}) = 0. \quad (3)$$

Figure 2.7: Equihash conditions

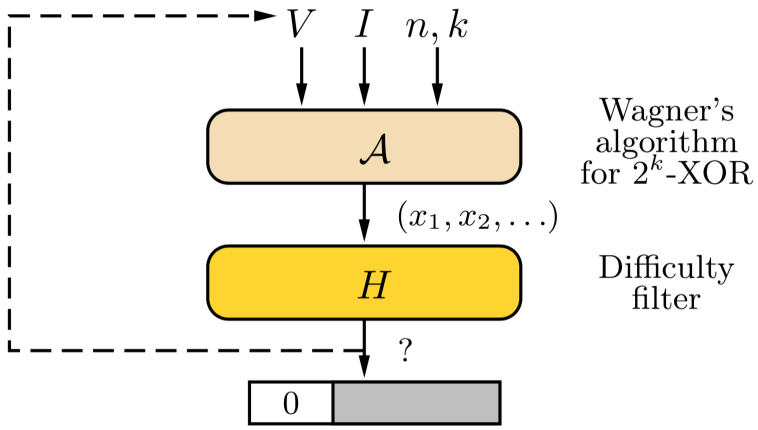

Figure 2.8: Equihash proof-of-work

2.4.2 In practice

There are three main steps in the Equihash algorithm: hash generation, sorting, finding collisions (figure 2.9).

The first hash generation could in theory use any hash function. In practice, the proof-of-work protocol has to define the function used, and Blake2b [3] was chosen for this step. Depending on the protocol of the chosen cryptocurrency, the hash generation may have a small difference with the original description of Equihash: a $2n$ string is generated with blake2b, and then split in two.

Then, the elements have to be sorted by their $\frac{n}{k+1}$ least-significant bits. The best traditional comparisons sorts have a known theoretical complexity of $O(m \log m)$ number of comparisons, with m the number of elements to be sorted. Depending on the algorithm used, a comparison requires at least a read access to the memory,

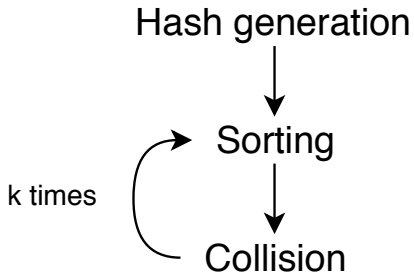

Figure 2.9: Steps of Equihash

that may be followed by a write access. Since the sorting step is repeated many times, we should reduce the number of memory accesses as much as possible.

We can take advantage of the fact than we may see these $\frac{n}{k+1}$ bits as unsigned integers, and that we know their bounds: they will always be in the range $[0; 2^{\frac{n}{k+1}} - 1]$. We will use the radix sort algorithm [25] that has interesting properties.

We first define a number of bits b as the radix. We will thus use 2^b different buckets in a separate zone of memory. Then, we sort the different items in those buckets according to their b least-significant bits. Notice how we have exactly one read access and one write access per element.

The procedure is repeated for the next b bits. Eventually, after the adequate number of passes, the whole data is sorted. We only need two memory zones: one to read the data, another one to write in the buckets. They can be swapped after each pass. Although the radix algorithm is not in place, it is not a drawback in this application: in any case, we need extra space to store the collisions. This is explained is section 3.3.6.

The number of passes p depends on the size of the data to be sorted, and the chosen value of b. In the Equihash setting, we have $p = \left\lceil \frac{n/k+1}{b} \right\rceil$ passes.

The number of memory access is thus twice the number of passes (one read access, one write access), multiplied by the number of elements: $2pm$. Since p is constant, the number of memory accesses with a radix sort is linear in time with respect to the number of elements.

One last particularity of the algorithm is the last collision step. From the above description, we could expect $\frac{n}{\frac{n}{k+1}} = k + 1$ loops of sorting/collision to fully process the n-bits strings. We actually have k steps, where the k^{th} collision takes place on the last $2\frac{n}{k+1}$ bits.

Chapter 3

Architecture of the system

The system relies on multiple parts communicating with each other. The broad picture is progressively zoomed in to reach a module-level description of the implementation. Some details about the communication between the board and the computer, and the FPGA and the memory, are also given.

Contents

3.1 Broad picture

Figure 3.1 shows the main two physical components of the system: a computer, and an evaluation board aimed at the developers to implement and test their design. Both units exchange data through the UART protocol.

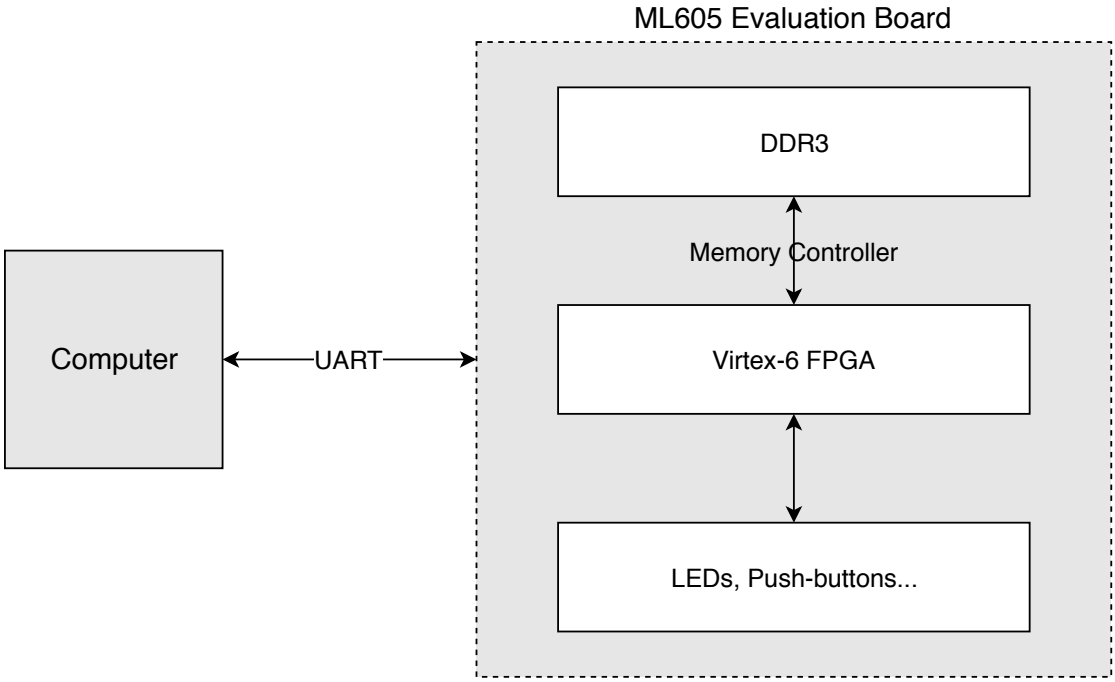

Figure 3.1: Broad architecture of the system

3.1.1 Computer

Since the heavy computation of the Equihash function is done on the board, the computer has very few requirements, and is agnostic regarding the brand or model. It just needs to be able to send an input seed, i.e. it must have:

- A serial port (via USB most likely)
- Any software capable of UART communication

3.1.2 Evaluation board

The design is implemented on the Virtex®-6 FPGA (XC6VLX240T-1FFG1156) ML605 Evaluation Kit (figure 3.2). The board includes the main basic components

that may be needed for a hardware design. Its key features for Equihash include [41]:

- A Virtex-6 FPGA, with the following main specifications
  - 301,440 flip-flops
  - 150,720 Look-Up Tables.
  - 14,976 kB of memory
- A soldered 200MHz differential clock
- 2GB DDR3 Memory SODIMM
- USB-to-UART Bridge
- USB-to-JTAG for configuration
- User I/Os (push-buttons, LEDs)

The board also provides lots of other features such as an LCD screen, a USB controller, a PCI Express port, an Ethernet port, a DVI port, etc. These are not used in this design.

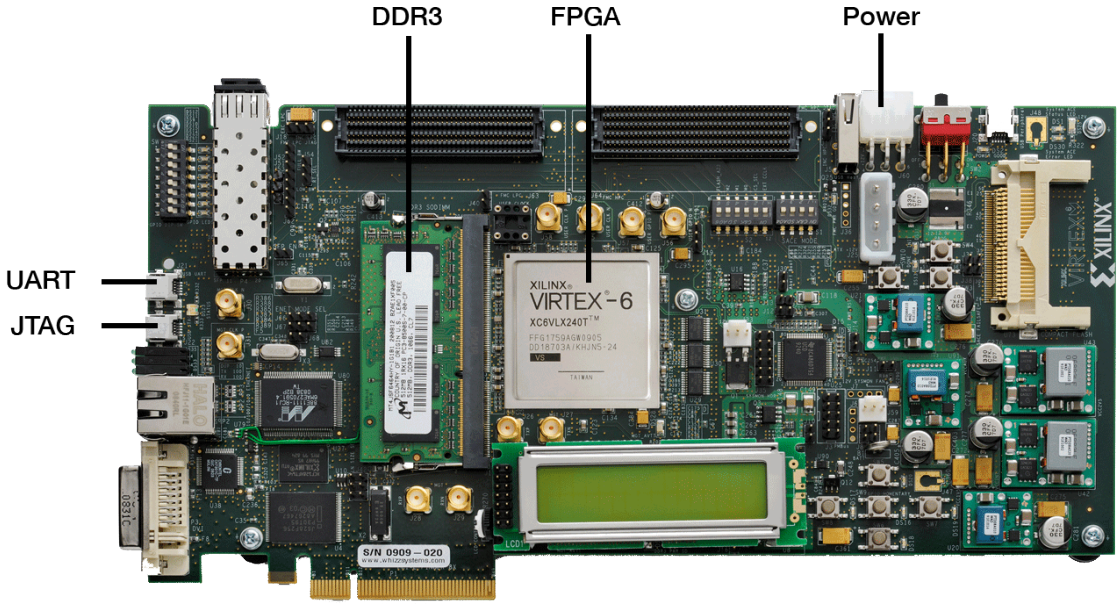

Figure 3.2: ML605 Evaluation Kit

3.2 Communication

3.2.1 UART protocol

The board computes the Equihash function, but it still needs some input parameters sent by a computer. The format of these parameters will be discussed later, the focus is now on the communication protocol between the PC and the evaluation board.

The key problem is that different hardware most likely run with different clock frequencies, and these frequencies are unknown to each device. Hence the connection cannot be synchronous. Since the area of the FPGA is also a concern, the protocol should be simple enough to implement.

The UART protocol (Universal Asynchronous Receiver Transmitter) has both benefits: simplicity and asynchronicity. The operation is described below.

As UART may send data in both directions, a device typically has two ports: `Tx` (transmitter) and `Rx` (receiver). For the sake of simplicity, figure 3.3 only shows a data flow in one direction. It is of course allowed to send data in both directions simultaneously, since the two streams are distinct.

The devices must agree on the **baud rate** [11, 36]: the number of symbols per second that occurs within a data transmission. In this case, the symbols are zero and one, hence it is equal to the bit rate, or number of bits exchanged per second. One of the more common baud rates, especially where speed is not critical, is 9600[1]. Other standard baud are 1200, 2400, 4800, 19200, 38400, 57600, and 115200 [15].

The baud rate should be chosen regarding the speed of transmission one is trying to achieve. However, there are some constraints with the frequency of the two communicating devices. While the clock frequency of the sender may be as low as the baud rate, the frequency of the receiver should be higher. The reason is that the receiver should try to sample each bit at the center of the oscillation. It is recommended that the clock frequency of the receiving device be 8 or 16 times higher than the baud rate to sample the data error-free [2].

A UART frame is typically composed of the following elements:

- A mandatory **start bit**: always 0.
- The mandatory **data**: its length may vary from 5 to 9 bits. Both devices must agree on the size of the data. Bits are sent little-endian style: from the least significant one, to the most significant one.

[1]For the longest stable period of modem technology after 1970, about 1984 to 1991, typical baud rate was 9600. It became the default value for serial equipments.

- An optional **parity bit**: to spot integrity problems.
- A mandatory **stop bit**: always 1. There may be multiple stop bits (e.g. 11).
- The logical **rest position** (when no data is transmitted or received) is 1.

Figure 3.3 shows a frame with 8 bits of data transmitted, and a 1-bit stop signal. Note that the start and stop bits are mandatory, the baud rate is slightly higher than the actual bit rate *in the application layer*[2]. Assume the baud rate is 9600, the data part of the frame is one byte, and there is no parity bit. With no interruption of transmission whatsoever, the bit rate *as seen by the application layer* will be

$$\text{baud rate} \times \frac{\text{data length}}{\text{frame length}} = 9600 \times \frac{8}{10} = 7680\text{bps}$$

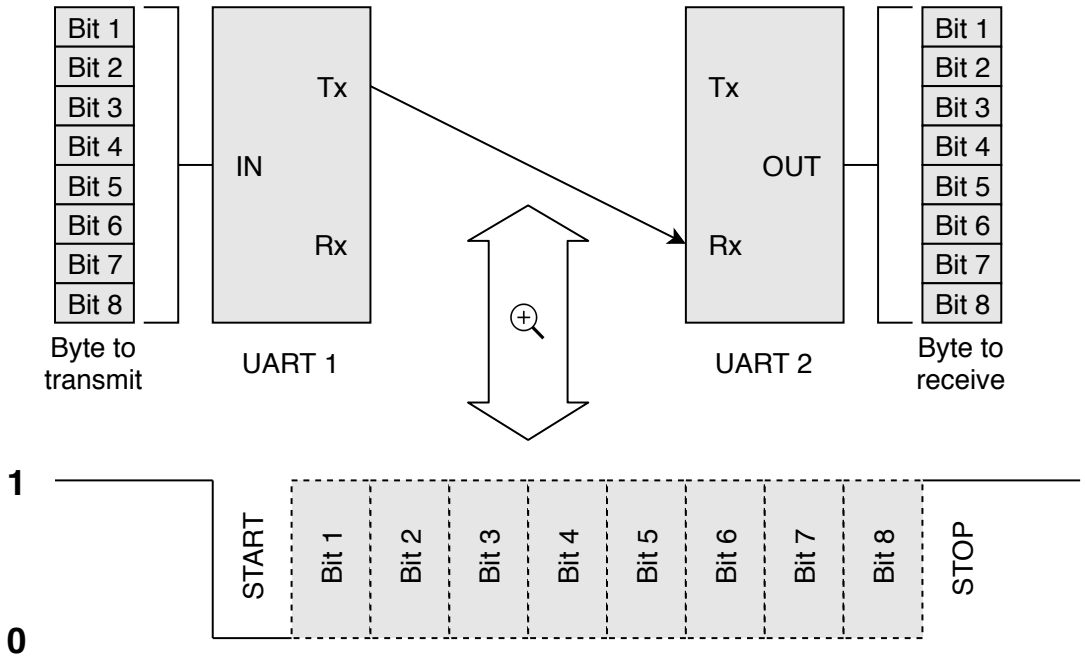

Figure 3.3: UART Protocol

3.2.2 Memory Controller

Xilinx provides some IP cores designed to work on its FPGAs. One of them is a memory controller for the DDR3 [40]. An example design is also supplied, that

[2]In this case, the Equihash core on the development board, and the software on the computer.

instantiates the memory controller as well as the required clock: while the main design runs at 200MHz, the memory controller requires a 400MHz clock.

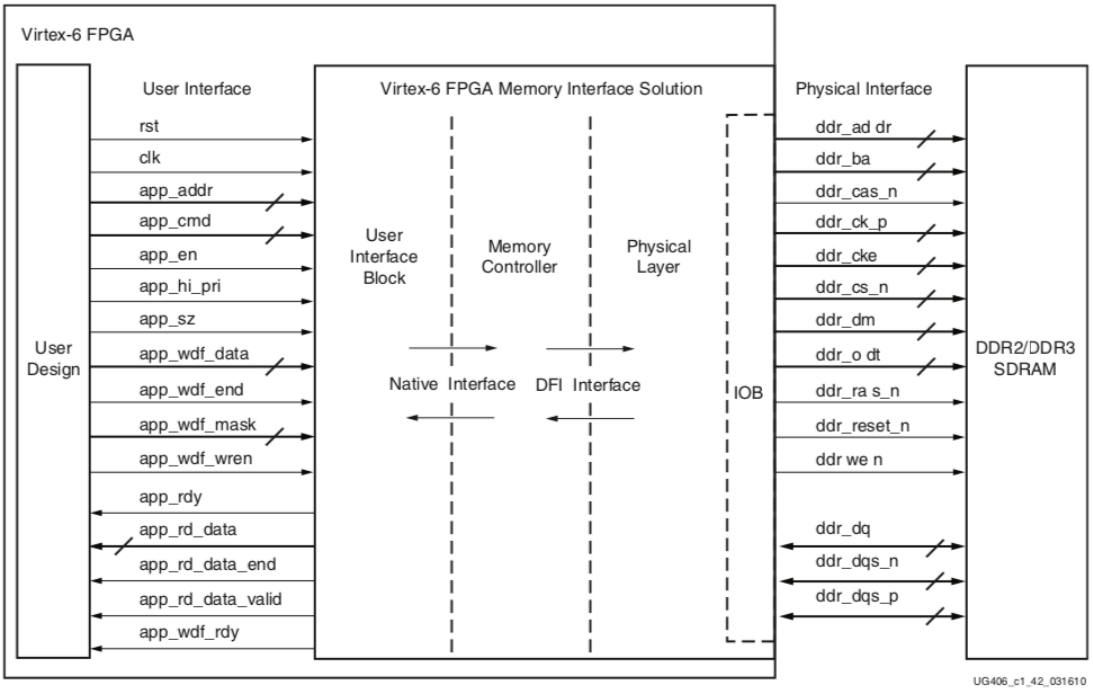

Figure 3.4: Memory controller user interface

Figure 3.4 shows the user interface provided with the core: it hides most of the logic to interact with the DDR3 SDRAM. The interface lets the user read and write words of 256 bits. An important aspect is that the words are 64-bits addressed, i.e. to read two consecutive words W_0 and W_1, the corresponding addresses are $A_0 = 0$ and $A_1 = 4$.

To write (figure 3.5), the user sends the WRITE command (`000`) in `app_cmd`, as well as the address in `app_addr`. `app_en` must be asserted. `app_rdy` must be high, and if the command is accepted, the signal stays high. The 256 bits data to write is given in `app_wdf_data`, and `app_wdf_wren` as well as `app_wdf_end` are asserted. If the data to write is taken into account, `app_wdf_rdy` stays high.

To read (figure 3.6), the user sends the READ command (`001`) in `app_cmd`, as well as the address in `app_addr`. `app_en` must be asserted. `app_rdy` must be high, and if the command is accepted, the signal stays high. When the data is available later on, the `app_rd_data_valid` signal becomes high and the data can be retrieved from `app_rd_data`.

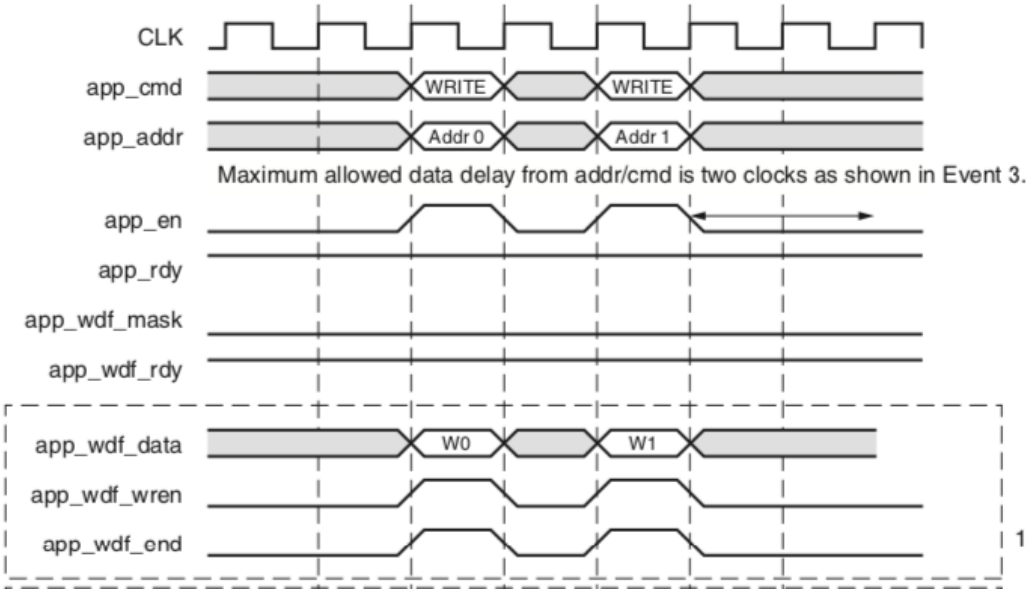

Figure 3.5: Write path of the user interface

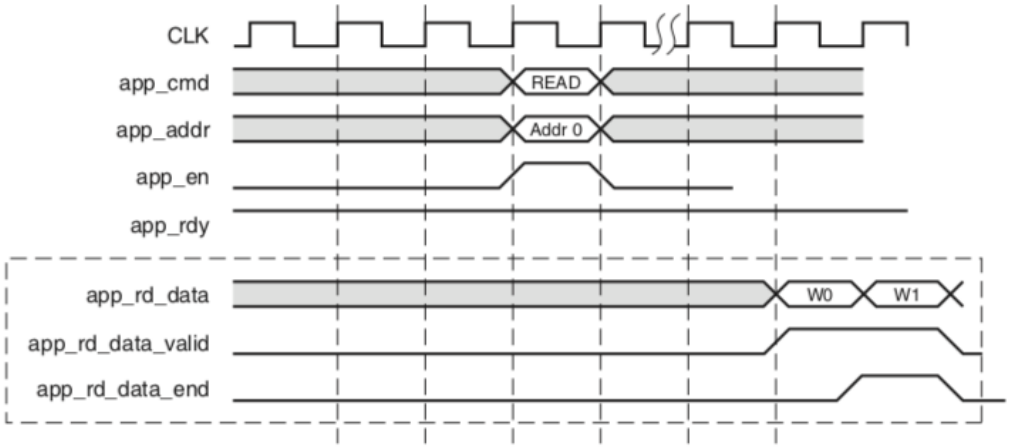

Figure 3.6: Read path of the user interface

3.3 Modules

The Equihash design is adapted from an open source implementation[10]. The chosen parameters were $n = 210, k = 9$, they were not modified. The design is divided in multiple modules; some of them achieve a specific step of the algorithm, while others are helper modules. Their working principles are described in this section.

3.3.1 `equihash_top`

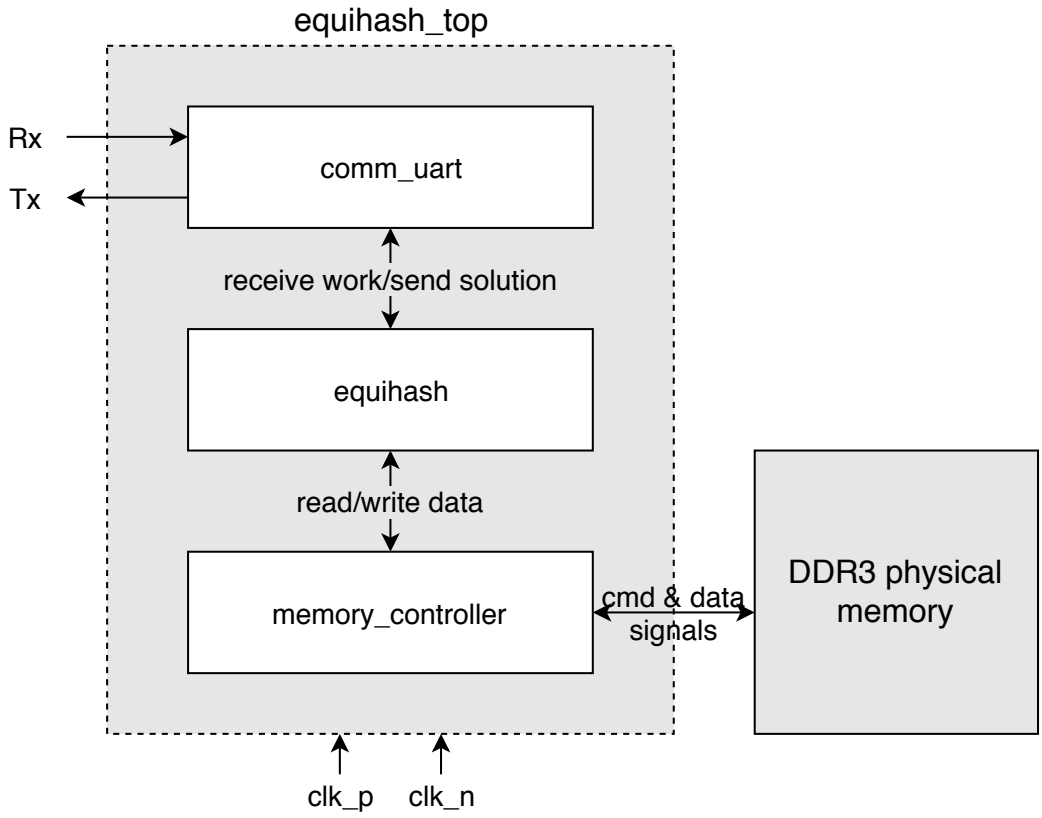

Figure 3.7: The `equihash_top` module

This is the top module of the design (see figure 3.7). It acts as a bridge between the DDR3 memory controller, the actual equihash algorithm, and the UART controller, and connects them together.

Its inputs and outputs are easy to understand:

- Two input signals for a **differential clock**.

- A **reset** signal (not shown on figure 3.7).
- The input `Rx` and output `Tx` signals required to communicate through the UART controller.
- Various outputs to **command the DDR3 chip**, and some inouts to **read and write data**.

Three modules are instantiated inside:

- `comm_uart`, the UART controller.
- `equihash`, the main Equihash module.
- `memory_controller`, the DDR3 memory controller interface.

3.3.2 `comm_uart`

The module is responsible for handling the UART communication with the host computer. Here are its IOs:

- A signal for the **input clock**.
- The input `Rx` and output `Tx` signals.
- An **output flag** to notify that a new job is available (i.e. new input parameters for the Equihash function).
- **Output busses** to carry the different parts of the new job.
- **Input busses** to send the solution back to the computer.

A few parameters are also expected :

- The **clock frequency**: 200,000,000 Hz.
- The **baud rate**: 115,200.

As explained in section 3.2.1, the baud rate is sufficiently small to allow the design to sample every bit in an efficient way. The clock will tick $\frac{200{,}000{,}000}{115{,}200} = 1,736$ times for each bit, more than enough to sample them correctly.

Receive new work

A small submodule `uart_receiver` is instantiated to accumulate the bits sent on the `Rx` wire. They are sampled at the mid-length of every bit transmitted. A flag is set to true whenever a new byte is available.

The computer sends the data as hexadecimal characters, followed by a line feed. For example, to send `0xDEADBEEF`, the computer sends the string `"DEADBEEF\n"`. Each character is encoded on a byte following the ASCII code.

On the `comm_uart` module level, each character is decoded back to a half-byte, and accumulated until the line feed character is encountered. The received data is then ready to be processed by the actual Equihash module, and a flag `tx_new_work` is raised to notify the module.

Send solution back

The solution, i.e. the 2^k indices, are hex-encoded in 8 characters (32 bits), followed by a line feed. Since the final solution may be found faster than the UART communication rate, a FIFO buffer is used to temporarily store the data to send.

3.3.3 `equihash`

The `equihash` module does no work by itself, it simply instantiates and connects together some submodules. Its IOs are:

- A **clock** signal.
- A **reset** signal.
- Inputs to **receive new work** from the UART controller.
- Outputs to **send the solution** back to the computer.
- The **`memory controller interface`** signals.

The instantiated submodules will be described in the next sections, and are just mentioned here for the sake of completeness:

- **`mem_gasket`**, a layer between the memory controller interface and the other modules.
- **`equihash_state`**, a module that orchestrate every stage of the algorithm.
- **`blake2b`**, the core that computes the eponymous hash function.

- `radix`, responsible for the sorting.
- `snoop`, a helper module for `radix`.
- `collision`, a module that finds the collisions between elements.

3.3.4 `mem_gasket`

The `mem_gasket` module is a layer between the memory controller interface, and the other modules that use the interface. Indeed, whether it is to write the blake2b hashes to memory, sort them, or find collisions, the external memory needs to be read or written. As multiple signals cannot be connected to a single input (e.g. one of the inputs of the memory controller), `mem_gasket` is going to correctly route the wires.

Writing in memory To write some data in memory, `mem_gasket` gives this interface (it will be referred to as *write interface*):

`waddr`	Address in memory where the data must be written
`wdata`	Data to write
`wvalid`	Boolean flag to confirm the write command

Reading in memory To read some data in memory, `mem_gasket` gives this interface (it will be referred to as *read interface*):

`rsend`	Boolean flag to send the read command
`raddr`	Address in memory where the data must be read
`rdata`	Bus to store the read data
`rvalid`	Boolean flag that indicates whether or not the reading operation was successful

Regarding the IOs of the module, we have:

- The classic **clock** and **reset** input signals.
- A 3-bits **state** input (more information in section 3.3.5).
- The usual **memory controller interface**.
- **Write interfaces** for the blake2b, radix, and collision steps. The signals `waddr`, `wdata`, and `wvalid` are simply repeated with a slight variation in the name to differentiate them (a special letter is added: `b` for blake2b, `r` for radix, and `c` for collision).

- **Read interfaces** for the radix and collision steps, as blake2b only needs to write in memory, not read it. The signals `rsend`, `raddr`, `rdata`, and `rvalid` are repeated and identified in the same way as for the write interfaces.

According to the state (i.e. whether the algorithm is in the blake2b, radix, or collision step), the module will give the right values to the memory controller interface based on the simplified *write and read interfaces*.

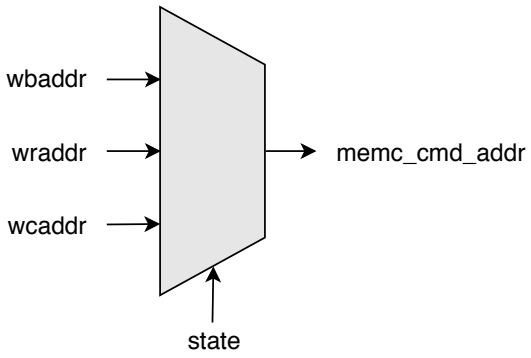

Figure 3.8: Example of a `mem_gasket` construction

On figure 3.8, an example of a multiplexer used by `mem_gasket` is shown. It allows to switch the source of the `memc_cmd_addr` according to the state. Note that an address must also be specified when reading.

3.3.5 `equihash_state`

The `equihash_state` module is the main controller of the Equihash design. It orchestrates every step of the algorithm. Its purpose is twofold:

1. Send a start signal to the other modules, and act upon their notification of completion.
2. Indicate the chunk of memory that each other module may use to read/write data in the DDR3 memory.

Here are the IOs of the module:

- A **clock** and a **reset** input signals.
- An output **state** signal.
- An output **stage** signal.
- Some output **start** signals to notify the other modules that they may begin.

- Some input **done** signals to let `equihash_state` know that they completed their work.
- Output busses that indicate the chunks of **memory available** for each state. The indication takes the form of pointers towards the start and end addresses of each chunk.

Here are the steps of the Equihash algorithm (see figure 3.9 for a visualisation of the state machine):

1. The algorithm begins by waiting for the DDR3 memory to be ready to receive commands (`init_done`), and for a new job sent by the host computer via UART (`uart_done`), this is the IDLE state. Next state is triggered when both signals are asserted.
2. Then, in the BLAKEB state, a `blake2b_start` flag is asserted, and the $2^{\frac{n}{k+1}+1}$ n-bits words are generated by the `blake2b` core, and written on memory. When the core is done with the generation, it raises the `blake2b_done` flag, and the machine enters the next state.
3. The next state, RADIX, sends a `radix_start` signal to launch the corresponding module. The words are sorted by their first $\frac{n}{k+1}$ bits, and the state machine is notified of the completion of this state when the `radix_done` flag is asserted.
4. Next, `collision_start` is set to 1, and the colliding words are identified in the COLLISION state. The machine waits for the `collision_done` signal. Steps 3 and 4 must be repeated for k stages. For this purpose, a `stg_ctr` is incremented after each successful COLLISION state, and compared to k to decide whether another stage is awaited, or if the final state may be entered.
5. After successfully completing the k stages of sorting and identifying collisions, the machine enters its last state, DONE, where the solution is sent to the computer, and `equihash_state_done` is finally asserted. The machine then goes back to the first IDLE state, and waits for its next work.

To generate the memory pointers, the module instantiates a submodule, `equihash_pointer`, described below in section 3.3.6.

3.3.6 `equihash_pointer`

As pictured on figure 3.10, the memory is divided in three zones: MEM_BUF0, MEM_BUF1, MEM_BUF2.

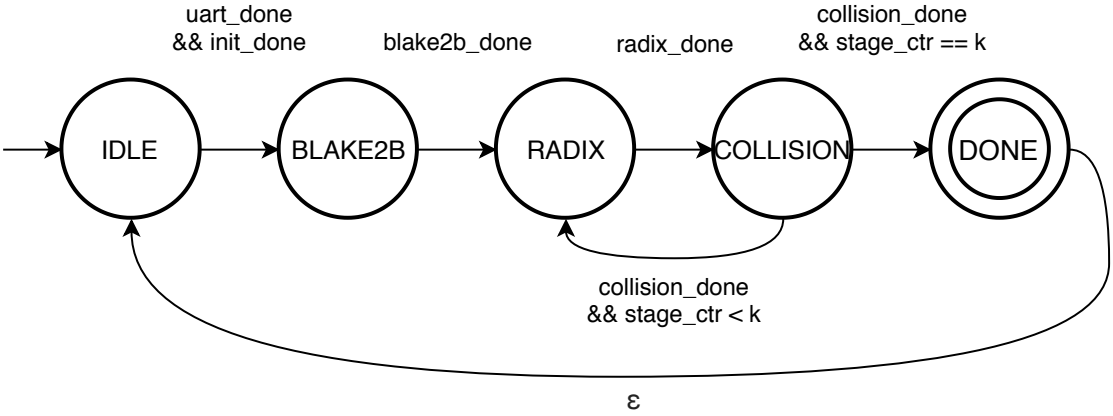

Figure 3.9: Main state machine of Equihash

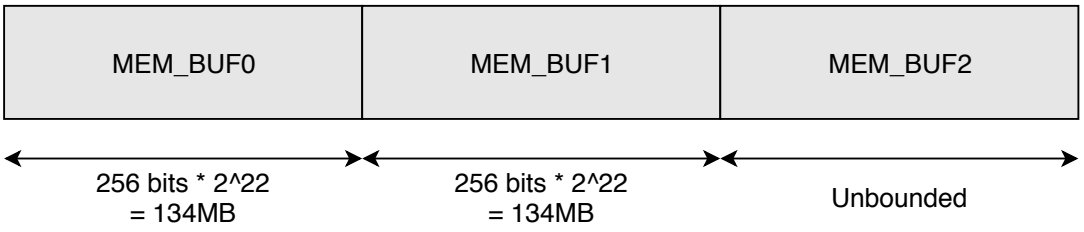

Figure 3.10: Memory organization

The first zone is used to store the hashes generated by Blake2b. Afterwards, the first and second zone are used to sort the data, swapping between the areas as needed to complete the sorting passes.

The collision step also need the two memory areas (this is why it was not a problem for the sort not to be in-place): one to read the data, and the other one to store the collisions.

The last area, MEM_BUF2 is used to store a binary tree of pointers to search for solutions.

The module outputs pointers towards the beginning and the end of the memory areas that may be used by the different steps. They are not constant: a start pointer may need to change following the number of passes of the radix step, and most of the end pointers are bound to the number of collisions found, which is different in each stage.

3.3.7 `blake2b`

The `blake2b` core is the first big step of the algorithm. Based on the input data of the Equihash function, it is going to generate $2^{\frac{n}{k+1}+1}$ words of n bits each. In practice, the algorithm specifies that Equihash must generate $2^{\frac{n}{k+1}}$ words of $2n$ bits, and then split each one in two parts to reach the expected amount of words.

The IOs of the module are the following:

- A **clock** and a **reset** input signals.
- The **`blake2b_start`** and **`blake2b_done`** signals from the `equihash_state` module.
- The **`blake2b_base_addr`** pointer from the `equihash_pointer` module.
- The **`uart_done`** and **`uart_rdata`** signals to receive the input data of Equihash
- A **`mem_cmd_full`** signal from the memory controller user interface, indicating when the module has to wait before sending more write commands.
- The **write interface** discussed in section 3.3.4.

The design uses an open source implementation of a blake2b core[27]. Its interface is described in table 3.1.

`init`	Initialize the core
`next_block`	Indicates that the new block may be digested
`block`	Block of 128 bytes, part of the input data.
`digest_len`	Desired length of hash in bytes.
`total_len`	Total length of data in bytes.
`ready`	Notify that the core is ready to digest another input.
`digest`	Output digest of the blake2b hash function.
`digest_valid`	Boolean flag that indicates that the input data is fully processed, and that the output hash may thus be used.

Table 3.1: Interface of `blake2b_core`

The module uses two state machines to process the blake2b step. The first one is in charge of the words generation, while the second one splits and writes the words in memory.

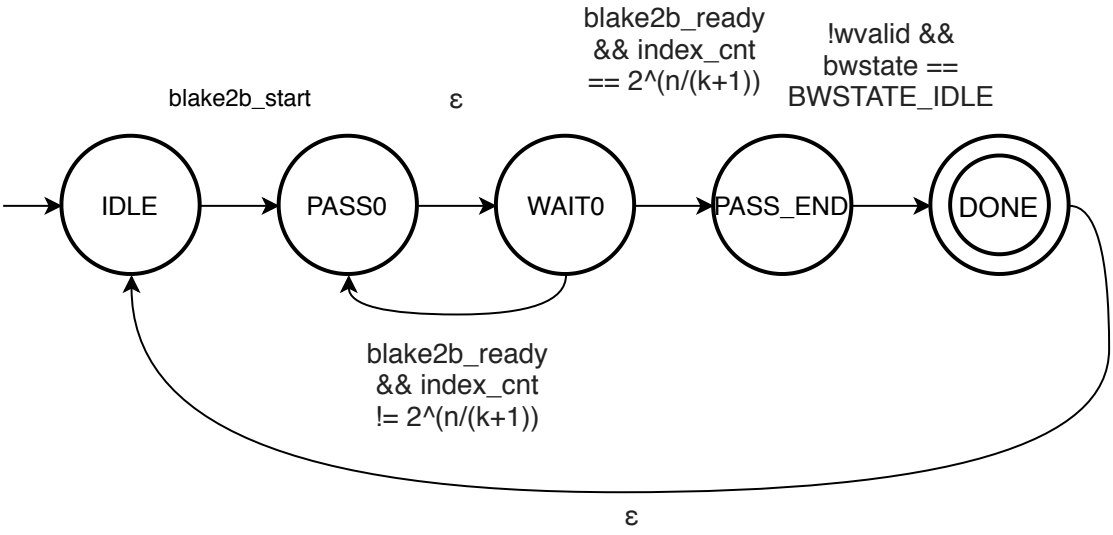

Figure 3.11: State machine of blake2b hash generation

Hash generation

A depiction of the generation state machine is presented on figure 3.11.

1. The algorithm enters the IDLE state where a counter `index_cnt` is initialized to 0. The counter keeps track of the number of hashes generated. It waits for the `blake2b_start` signal to be asserted before entering the next state.

2. The PASS0 state is juste an intermediate state that lasts for one clock cycle. It initiates the `blake2b_core` by asserting the `blake2b_init` signal, and sets the `blake2b_block` equal to the concatenation of the UART data and the value of `index_cnt`. The `blake2b_core` starts digesting the data.

3. The machine stays in state WAIT0 until `blake2b_ready` is asserted, indicating that the core is done with digesting the word, and ready to receive some new data to hash. Then we have two choices : either there are still hashes left to generate (i.e. `index_cnt` $\neq 2^{\frac{n}{k+1}}$), or the generation is complete, and the module can enter the next state.

4. In the PASS_END state, the machine waits for the last hashes to be written in memory, then skips to the DONE state.

5. In the DONE state, the module resets the counter, and notifies `equihash_state` that it completed its job by asserting `blake2b_done`. It then goes back to IDLE.

Hash splitting

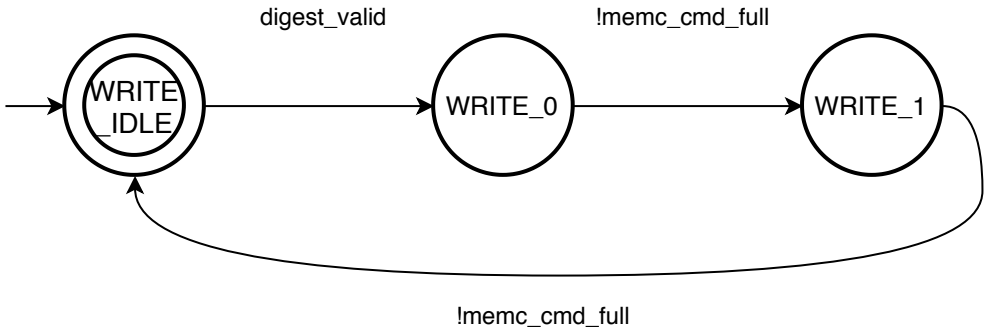

Figure 3.12: State machine of blake2b hash separation and memory writing

Concurrently to the generation of the hases, they are also split and written to the DDR3 memory. Two new counters are used: `write0_cnt`, initialized at 0, and `write1_cnt`, initialized to 1. Another state machine, depicted in figure 3.12, is used.

1. In the IDLE state, the machine simply waits for a new digest to be available. When `digest_valid` is asserted, it enters the next state.

2. In WRITE_0, the module first waits for the memory to be writable. Then it assigns the correct signals to the *write interface* so that the data written is the concatenation of `write_cnt0` and the n least-significant bits of the digest. We thus write the index of the hash as well as the hash itself. The address is the addition of the `blake2b_base_addr` and `write_cnt0`. Afterwards, the machine goes to the next state.

3. In WRITE_1, we mostly do the same as in the previous state, except that we use the `write_cnt1` and the n most-significant bits of the digest. The machine then returns to the IDLE state and waits for the next digest.

For every generated hash, the counters `write_cnt0` and `write_cnt1` are incremented by 2. The former keeps track of the even indexes, and the latter handles the odd indexes.

Each written item is prefixed by a 10-bits word filled with ones (juste before `write_0/1`). This is helpful in the collision step, explained in section 3.3.10.

3.3.8 `radix`

This module is responsible for sorting the words in memory. The choice of the radix algorithm is discussed in section 2.4.2. The most important steps of the algorithm are the following:

1. Read an element to sort from memory.
2. Write it in memory in the appropriate bucket, according to the value of its $[s \times r - 1; (s-1) \times r]$ bits, where s is the pass number, and r is the chosen radix.

The choice of the bucket is based on a clever pointer system to avoid wasting memory, a comprehensive explanation of which is given in section 3.3.9. On the basis of these elements, the IOs of the module are easy to understand:

- A **clock** and a **reset** input signals.
- The **`radix_start`** and **`radix_done`** inputs to interchange with `equihash_state`.
- The **`radix_base_addr`**, **`radix_scratch_addr`**, and **`radix_end`** pointers from `equihash_pointer`.
- The **`bucket[x]_base`** signals from `snoop`.
- The *read interface* and *write interface*, as explained in section 3.3.4.
- A **`memc_cmd_full`** signal from the memory controller interface, that notifies whether the DDR3 memory is or is not ready to receive new commands.
- Output **`snoop_pass`** and **`snoop_rst`** to indicate the number of the pass, and the reset signal, to the `snoop` module.

Two state machines rule the operation of the sort module. The state of the reading machine is stored in `radix_rstate`, and the state of the writing machine is in `radix_wstate`. The module has two parameters: RADIX_BITS and RADIX_PASS; the former indicates the number of bits chosen for the radix base, and the latter specifies the number of passes required to sort the data completely.

Since the radix algorithm is not in place, it needs two chunks of memory: one to read the elements, and another one to write the data in the appropriate buckets. The two chunks are swapped for each pass by using a simple selector: if the pass number is even, we read in `radix_base_addr` and write in `radix_scratch_addr`. If the pass is odd, it is the opposite.

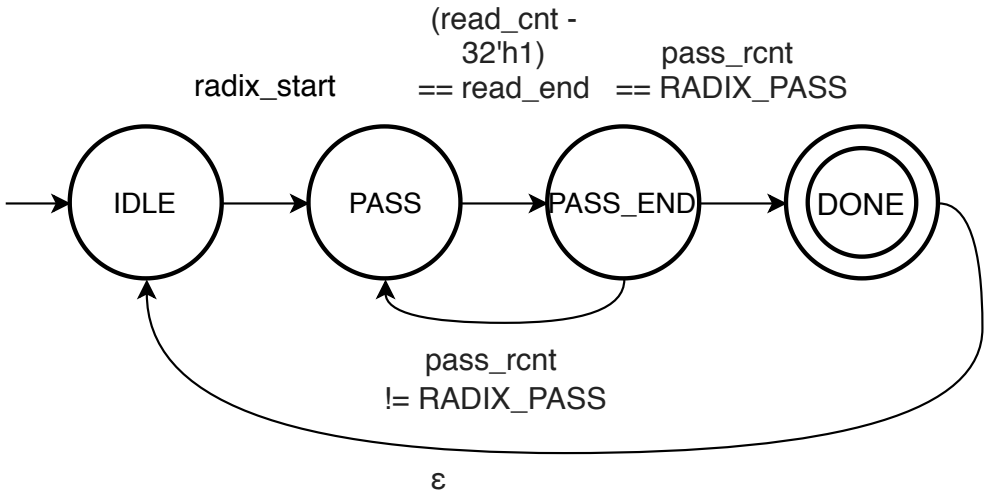

Figure 3.13: State machine of radix sort, reading in memory

Reading an element

The state machine ruling the reading operation is depicted on figure 3.13. There are two registers used to keep track of the reading process: `read_cnt` for storing the number of elements read per pass, and `pass_rcnt` to keep track of the number of passes.

1. In the IDLE state, `read_cnt` and `pass_rcnt` are reset to zero while the machine waits for the `radix_start` signal from `equihash_state`.

2. In the PASS state, the module increments `read_cnt`, sends a read command, waits for the result and repeats the process until every piece of data has been read (`read_end` is a pointer towards the very end of the data to sort). Then, since the machine has completed a read pass, it increments `pass_rcnt` and goes on to the next state.

3. In PASS_END, the machine first waits for the writing machine to finish its work in the current pass (`radix_wstate == WSTATE_PASS_END1`, not shown in the figure). Then, it compares `pass_rcnt` with the RADIX_PASS parameter. If they are equal, it means that the sort is complete, and the module switches to the final state. Otherwise, it resets `read_cnt` and goes back to the PASS state.

4. The DONE state simply resets `read_cnt`, and goes back to the IDLE state.

In this reading state machine, the address sent to the `mem_gasket` via the *read interface* is simply the value of `read_cnt`, since every element is read linearly, one after each other.

Once a new piece of data is read (i.e. `rvalid` is asserted), the values of `rvalid` and `rdata` are stored in temporary registers `rvalidq` and `rdataq` at the next clock cycle. This is necessary for the timing of the writing state machine.

Writing in the correct bucket

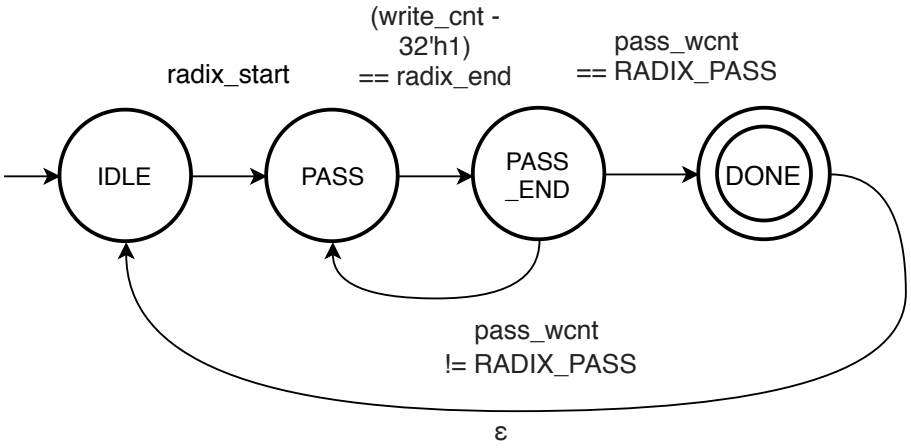

Figure 3.14: State machine of radix sort, writing in memory

Figure 3.14 shows the state machine for the writing part of the radix sort. There are two registers used to keep track of the writing process: `write_cnt` for storing the number of elements written per pass, and `pass_wcnt` to keep track of the number of passes. There are also some `bucket[x]_cnt` registers to store the number of elements contained in each respective bucket.

1. The machine starts in the IDLE state, where it resets the registers `write_cnt` and `pass_wcnt`. It waits for the `radix_start` signal before switching to the next state.

2. In the PASS state, the module waits for `rvalidq` to be true (i.e. a new piece of data has been read), and makes sure `memc_cmd_full` is de-asserted before sending a new write command. The module must define the bucket in which the data will be written. This is done in the previous clock cycle, when `rvalid` is asserted (and not yet `rvalidq`), by inspecting the appropriate bits of the `rdata`.
   Once the bucket is selected, the address is computed : it is the sum of `bucket[x]_base` (see section 3.3.9), `bucket[x]_cnt`, and `radix_base_addr` or `radix_scratch_addr` depending on the ongoing pass.
   Finally, `write_cnt` and `bucket[x]_cnt` are incremented.

If `write_cnt` reaches `radix_end`, every element has been written for the current pass, and the state is set to WSTATE_PASS_END. Otherwise, we repeat the whole bucket selection/address computation/writing procedure.

3. In PASS_END, the machine compares `pass_wcnt` with the RADIX_PASS parameter. If they are equal, it means that the sort is complete, and the module switches to the final DONE state. Otherwise, it goes back to the PASS state. The `write_cnt` register is reset to 0.

3.3.9 `snoop`

The radix algorithm has to distribute the elements to sort in multiple buckets, and repeat this step for multiple passes until the elements are fully sorted. Since the number of memory accesses are crucial in this memory-hard setting, dynamic memory allocations must be avoided at all cost.

The `snoop` modules helps to figure out the exact start address of each bucket, before the actual sorting takes place. This allows to completely avoid the need to move big chunks of memory, or to waste it.

Here are the IOs of the module:

- A **clock** and a **reset** input signals.
- The **`pass_cnt`** input that indicates the actual pass of `radix`.
- The **`wvalid`** and **`wdata`** signals from the *write interface*.
- The **`bucket[x]_base`** output signals, where `[x]` is the number of the bucket, that indicate their start addresses.

The module listens to `wvalid` and `wdata`. Whenever there is a new piece of data written to memory, the module infers which part of it will be useful for the next pass of radix sorting, and increments the pointers towards the buckets accordingly. Because this computation is done in parallel whenever new data is being written, the pointers are ready as soon as the sorting step starts and needs them.

An example is provided with figure 3.15. In this example, we have a radix base of two, that is four buckets. Assume a new word is written to memory, and the sorting did not start yet. `snoop` looks for the first two bits of the data, sees that it is equal to 1, and that is will thus be sorted in the BUCKET_1 when the sorting will begin. Therefore, the pointers towards BUCKET_2 and BUCKET_3 are both incremented by one to leave space in BUCKET_1.

This effectively ensures that no memory is wasted, and that enough space is provided for each bucket.

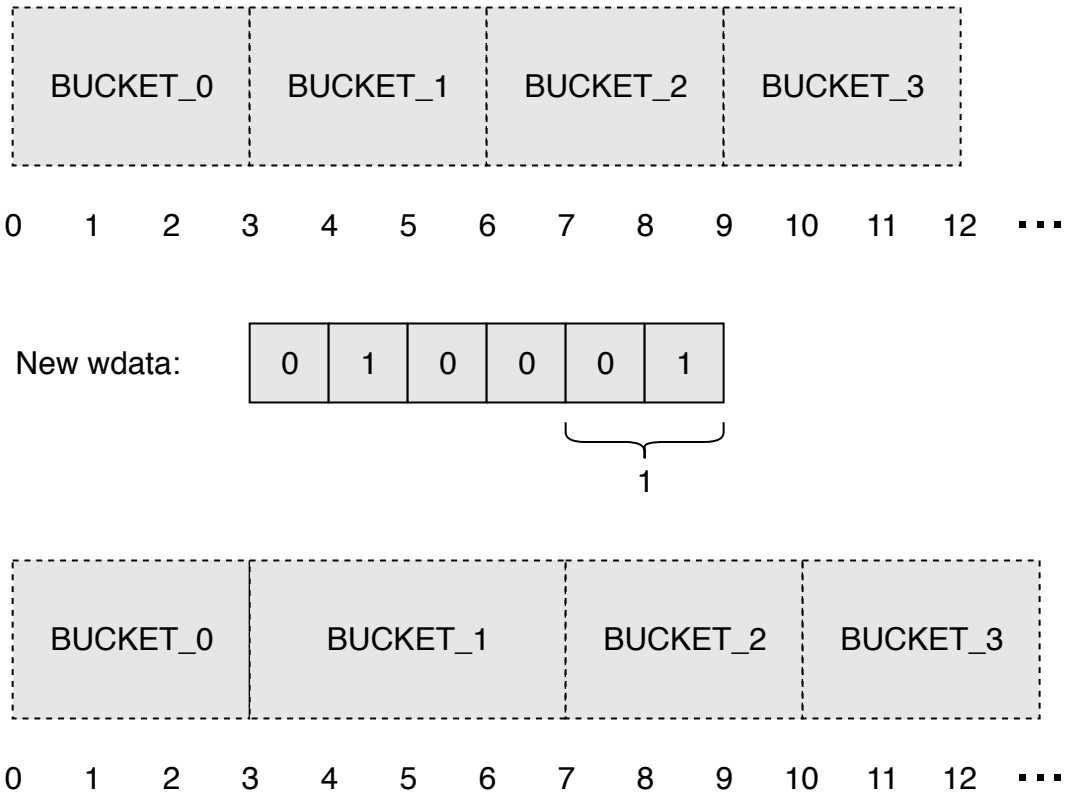

Figure 3.15: Example of the `snoop` module operation

The size of the radix base is very important. It should be large enough to decrease the number of required passes to complete the sorting. But as the size of the radix increases, the number of buckets and the logic to keep track of them also grows, *exponentially*.

The choice of the radix size must also be made wisely: an increase is size does not necessarily imply a drop in the number of passes. In the example of figure 3.15, a radix size of 2 involves three passes, and a size of 3 requires two passes, which is better time-wise. But a size of 4 or 5 also requires two passes. The overall time required to sort the data would no longer be affected, while the area needed for the logic would increase.

3.3.10 `collision`

Here are the IOs of the module:

- A **clock** and a **reset** input signals.
- The *read interface* and *write interface*.

- Various **pointers** to read the items and write the collisions.
- Output **UART signals** to send the solution back to the computer in the last stage.

The collision step is responsible for reading the sorted items, and finding collision between them on the $\frac{n}{k+1}$-bits part corresponding to the stage of the algorithm. A submodule `collision_store` is instantiated to actually store the collisions in memory. Figure 3.16 gives more insights about the functioning of both modules.

1. Sorted items are read from memory, and put in a FIFO list.
2. On the other side of the FIFO list, the `collision_store` module checks for collisions on the last $\frac{n}{k+1}$ bits. Colliding strings are stored internally, and matched pair-wise.
3. The colliding indexes and their special prefix (a 10-bits word filled with ones) are concatenated together, and written in MEM_BUF2 at address X (the `equihash_pointer` module keeps track of the address between each stage).
4. This address X and the actual collided bits shifted to the right are concatenated together (*without* the special prefix), and written in memory. The memory area is MEM_BUF1 (resp. MEM_BUF0) if items were read from MEM_BUF0 (resp. MEM_BUF1).
5. Once every sorted item is read and treated, the collision step ends.

Figure 3.16 provides an example that is worth explaining to fully understand the functioning of the module. The example happens in the first stage. Two strings, `123abc` and `456abc` were generated by blake2b, written with their respective indexes (`001` and `002`), and their special prefix (`111`). They are read from MEM_BUF0, then enter the FIFO, and happen to be in collision on their last bits. In MEM_BUF2, we will write their prefix and indexes at address X: `111|001|111|002`. In MEM_BUF1, we write the address X, and the collision shifted to the right: `AddrX|000575`, because $123 \oplus 456 = 575$.

When the last stage comes along, the solution must be returned back to the computer. We can notice that the strings in MEM_BUF2 are actually a binary tree (figure 3.17). When the last colliding strings are found in the last stage, a binary search occurs: the `collision` module will recursively navigate in the tree by reading the addresses, until it finds a leaf, recognizable by the special prefix. Once a leaf is found, its index part is sent back to the computer through UART.

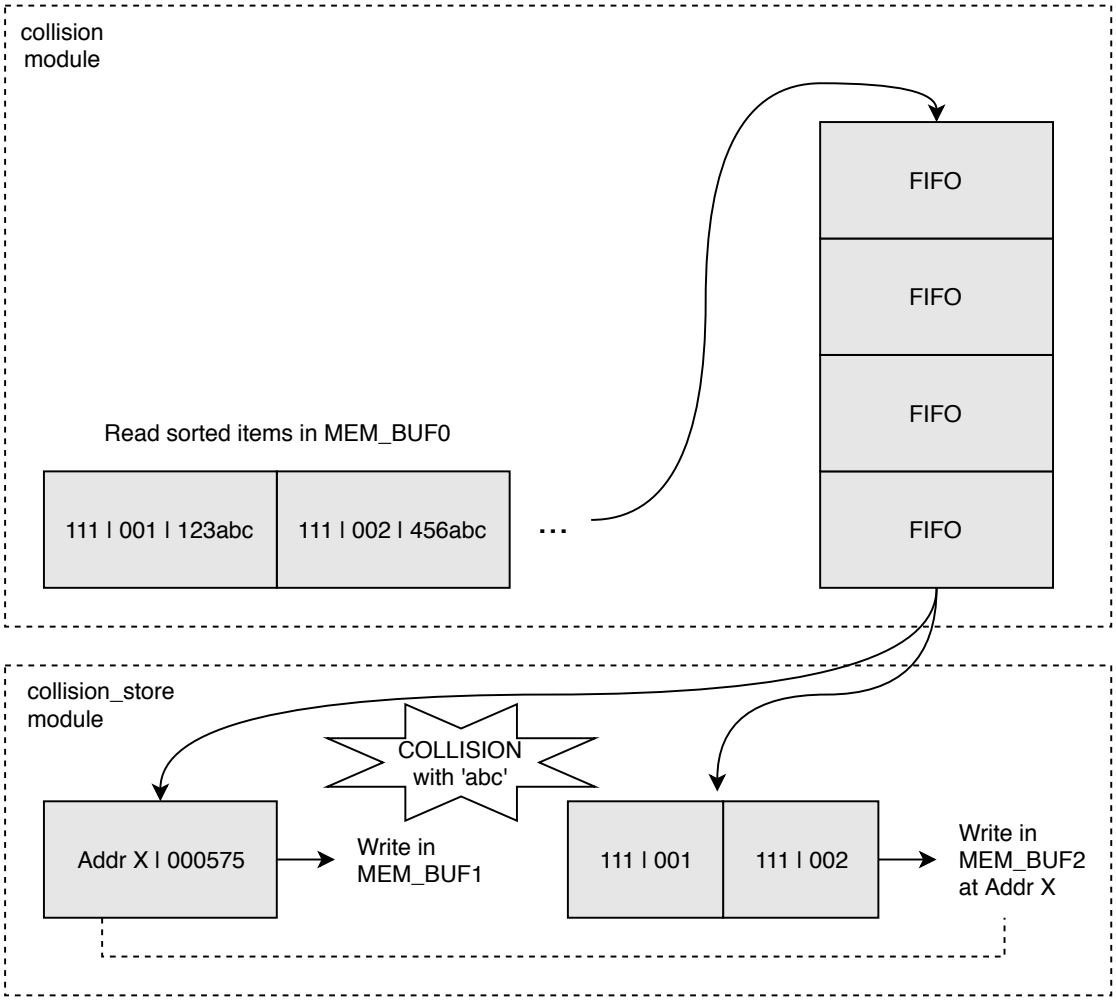

Figure 3.16: Example execution of the collision module

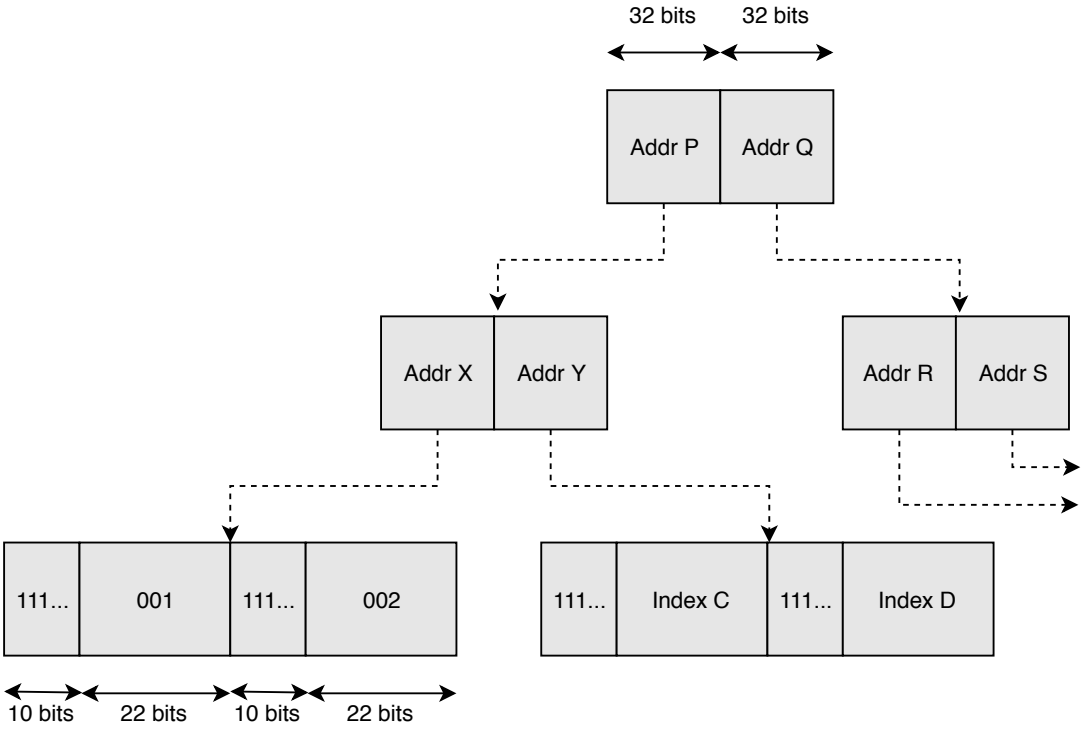

Figure 3.17: Binary tree of colliding indexes

Chapter 4

Improvements and measurements

A number of improvements were tried on the design to improve the efficiency. Results of the measurements are shown and discussed, and a comparison with a CPU implementation is given.

Contents

The timing information shown in this section has been measured by counting the number of cycles taken by each step in each stage, and outputting these results through UART. As the 200MHz embedded in the board should be stable, we can deduce the time by dividing the number of cycles by the frequency.

Power measurements were done statically by the XPower Analyzer tool in the Xilinx ISE suite. These are not live samples.

The number of components are shown in the Design Summary of Xilinx ISE. Since the different steps (Synthesis, Map, Place and Route) of the design implementation are not deterministic, small variations may occur.

4.1 Better use of resources

The `collision` module uses multiple FIFOs to store the elements read, or to help with the binary search. The implementation used an open source implementation from [26]. Unfortunately, Xilinx tools did not correctly infer a BlockRAM, which is initially intended. A translation to pure registers wasted resources and made routing harder. Instead, BlockRAMs should be used.

There are special Verilog primitives that can be used to instantiate FIFOs made of BlockRAMs on a Virtex-6: `FIFO18` and `FIFO36`. The maximum width of `FIFO18` is 36 bits, and 72 bits for `FIFO36`, both with a depth of 512. The depth is sufficient, but the width is not: complete words of 256 bits need to be stored in the `collision` module for example. To reach the needed width, FIFOs can be wired in parallel (figure 4.1): 4 `FIFO36` for a width of 256 bits.

The `blake2b` core had some timing problems: it needed a lower-frequency clock than the rest of the design to function properly. A 50MHz clock is instantiated on the basis of the 200MHz soldered one. Appropriate components and primitives were used to sync it at best.

The communication between the `blake2b` core and the higher-frequency other parts is ensured through FIFOs. Indeed, the `FIFO18` and `FIF36` allow asynchronous reads and writes, with different clock signals for reading and writing.

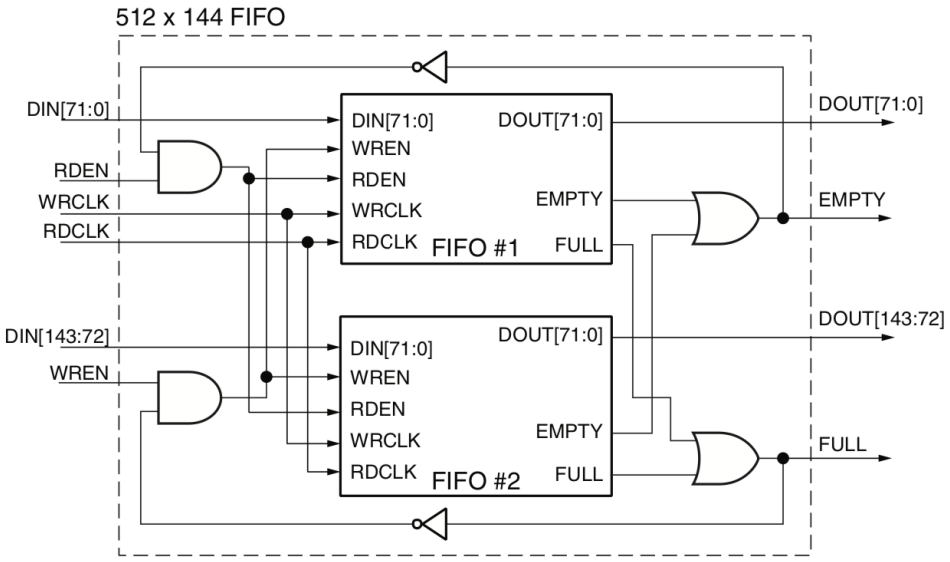

Figure 4.1: Parallel FIFO construction to increase width

4.2 Number of collisions

The number of supported collisions is important, since it limits the amount of pair-wise XOR operations possible. Indeed, let x be the number of contiguous collisions, we have $\frac{x!}{2(x-2)!} = \frac{x(x-1)}{2}$ possible configurations.

Figure 4.2 shows the timing of the algorithm obtained without any modification to the number of collisions supported (i.e. four contiguous collisions). We can observe that this number is not sufficient to obtain a solution at the end: the time taken by the *sorting* and *collision* steps decreases, meaning that there are less and less elements to treat. By step 6, there are almost no items left, and the algorithm stops at step 8.

On figures 4.3 and 4.4, other values are tried: 6 collisions for up to 15 pair-wise XORs, and 7 collisions for 21 XORed items. The observation of the timing is insightful: the logic for 6 collisions allows the algorithm to terminate, but loses some solutions in the process.

The last experiment with support for 7 collisions seems to be enough: we see very little to no change in the time taken to treat the data in each step. We can conclude that the number of elements stays almost constant, and allows for the full solution to be retrieved after step 9.

The full utilization and power consumption reports are shown in appendix A.2. Some trends stand out lightly: the power consumption of the collision module that increases slightly, from 8.3 to 12.1 mW. The number of LUTs increases by 6%, while the number of flip-flops stays almost constant. This is consistent with the augmentation of logic in the module.

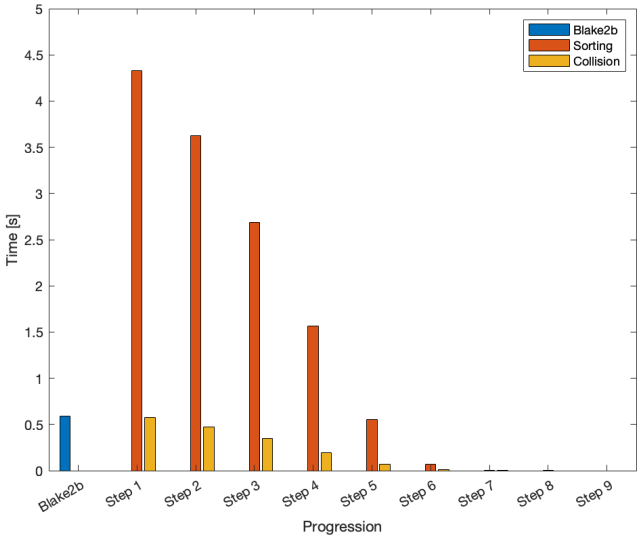

Figure 4.2: Execution of Equihash with support for 4 collisions

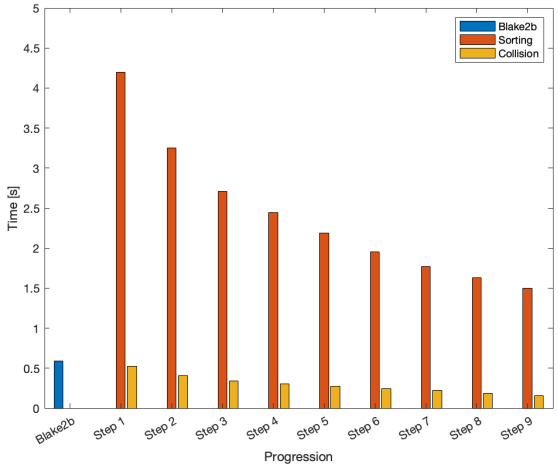

Figure 4.3: Execution of Equihash with support for 6 collisions

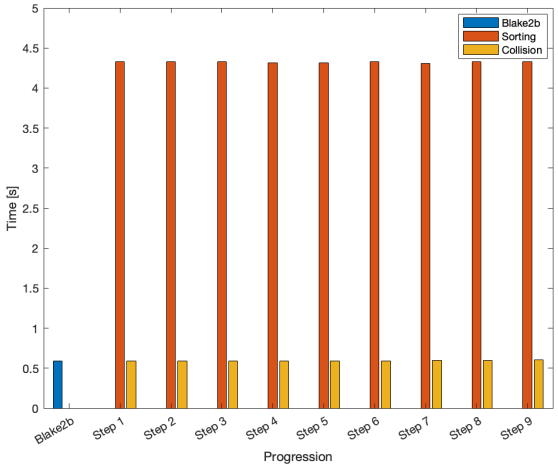

Figure 4.4: Execution of Equihash with support for 7 collisions

4.3 Radix sort

In the previous section4.2, we can observe that the radix part of the algorithm takes most of the execution time. More precisely, here is the total time taken by each part for the execution in figure 4.4:

Blake2b	**Sorting**	**Collision**
0.58 s	38.92	5.44

This is not surprising: in this state, the implementation uses 4 bits as the radix. Since the sorting takes place on $\frac{n}{k+1} = \frac{210}{9+1} = 21$ bits, we need 6 passes to sort the data entirely. To improve the timing, the obvious goal is to decrease the number of needed passes. Here's how the radix impacts the passes:

# radix bits	**Required buckets**	**# Passes**
4	16	6
5	32	5
6	64	4
7	128	3
8	256	3
9	512	3
10	1024	3
11	2048	2

Up to 7 bits, the number of passes decreases by one as the radix increases by one bit. Reaching a radix of 7 bits would allow the overall sorting timing to be divided by two with 3 passes instead of 6. 11 bits is the next step, allowing the data to be sorted in only two passes. Reaching an $O(n)$ sorting complexity with only one pass is unrealistic: it would mean keeping track of the same number of buckets as there are elements: 2^{21}.

As the number of buckets increases, so does the logic to keep track of the pointers towards them (achieved in the `snoop` module, see section 3.3.9). Let's see how far we can go.

Figures 4.5, 4.6, 4.7, 4.8 show the placement of the logic on the FPGA for respectively 5, 6, 7, and 8 radix bits. Notice how the yellow area grows: it includes the `radix` and `snoop` modules. Again, full results for power consumption and utilization are available in appendix A.3.

7 radix bits were achieved, effectively decreasing the needed time by a factor 2. For 8 bits, the placement completes (hence the availability of figure 4.8), but the

routing process is having a hard (and too long!) time to route the design: it had to be terminated manually, uncompleted.

The increasing logic can be observed on figure 4.9. We can see that the number of LUTs increases faster than the number of Flip-flops. The design needs more logic to perform the multiple additions at the correct pointers than it needs to store data. However, both increase exponentially according to the radix size, and thus linearly depending on the number of buckets, as expected.

Figure 4.10 shows the percentage of used slices, as well as the percentage of fully-used FF/LUT pairs that compose a slice. As the logic needs increase, we observe that the placement is more optimized: fully-used pairs increase. The utilization tops at 31%: there are still plenty of slices available, but the routing seems more complex: this is the limiting factor.

The improved timing reached with 7 radix bits can be observed on figure 4.11: the sorting step is twice as fast as before.

Finally, we can observe the energy used according to the size of the radix: the logic grows, and thus needs more power. However, the time needed for the algorithm to complete decreases. For an useful result, we need the time to decrease faster than the power. This is the case here: we observe a 39% decrease in the energy needed, from 245.66 J to 149.45 J.

# radix bits	**Power [mW]**	**Time to complete [s]**	**Energy [J]**
4	5469	44.918	245.66
5	5429	38.58	209.45
6	5649	31.95	180.48
7	5967	25.05	149.45

Figure 4.5: 5 radix bits. Magenta = Memory controller, Cyan = Blake2b, Green = collision, Yellow = Radix, Red = UART

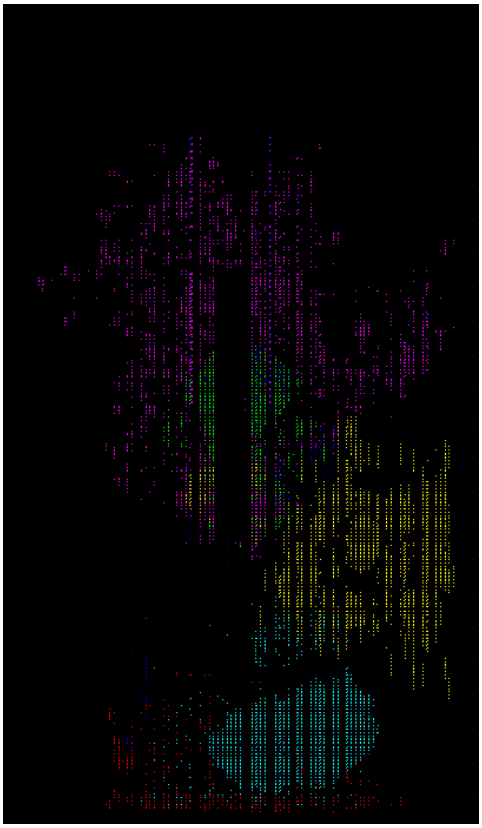

Figure 4.6: 6 radix bits. Magenta = Memory controller, Cyan = Blake2b, Green = collision, Yellow = Radix, Red = UART

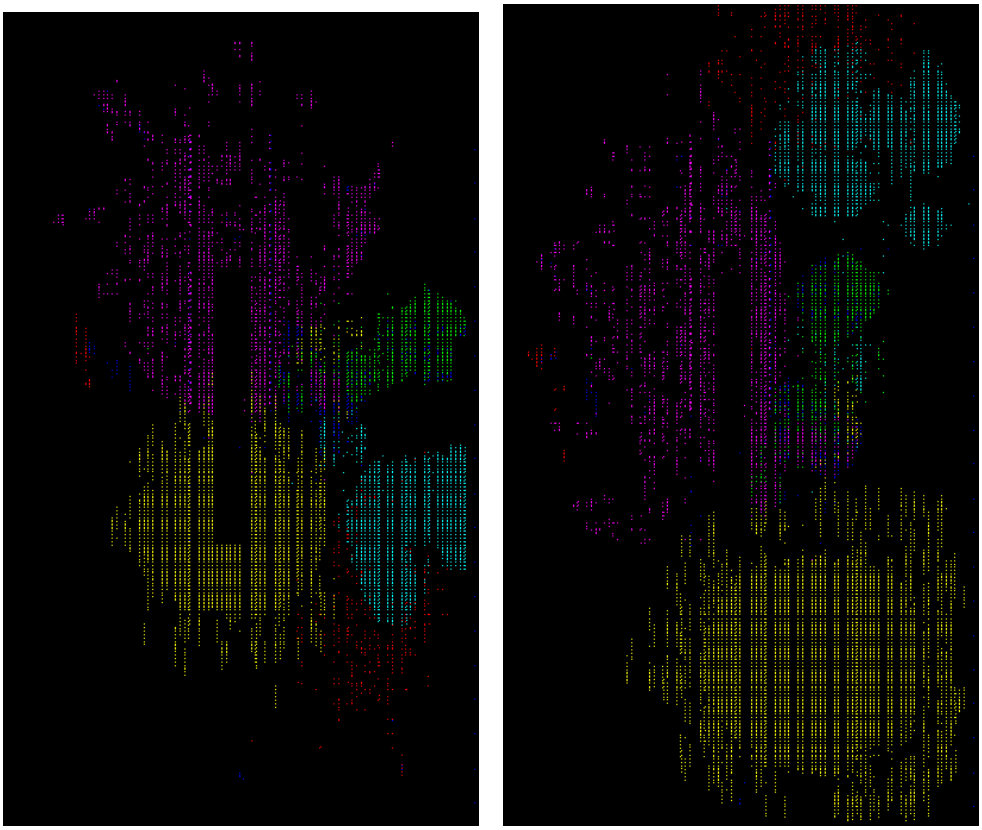

Figure 4.7: 7 radix bits. Magenta = Memory controller, Cyan = Blake2b, Green = collision, Yellow = Radix, Red = UART

Figure 4.8: 8 radix bits. Magenta = Memory controller, Cyan = Blake2b, Green = collision, Yellow = Radix, Red = UART

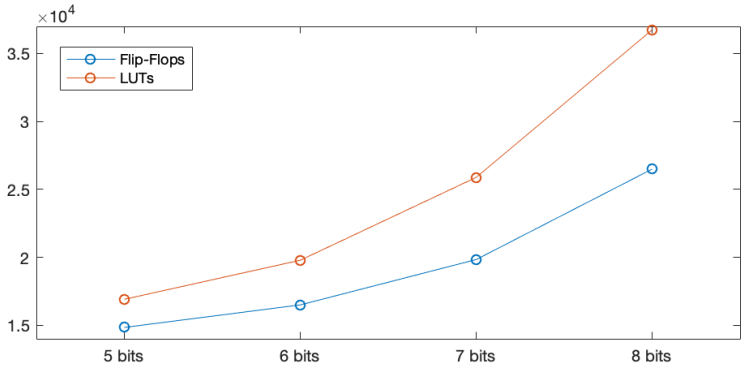

Figure 4.9: Number of Flip-flops and LUTs in relation with the size of the radix

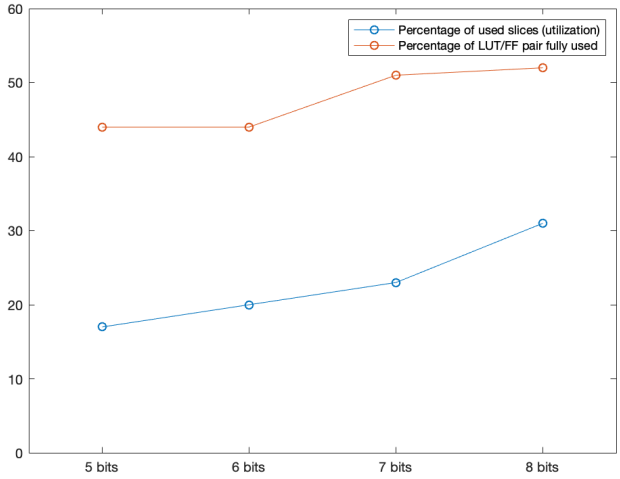

Figure 4.10: Evolution of the utilization

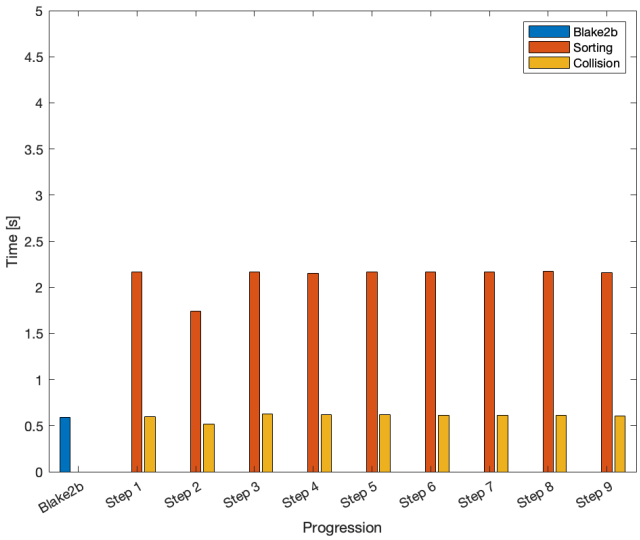

Figure 4.11: Execution of Equihash with 7 radix bits

4.4 Memory

We also tried to use the properties of DDR3, that introduces an 8-burst feature. In theory, this allows the controller to access the memory in a *burst* of 8 words [38], reducing the latency.

The words have an individual size of 64 bits. Since Xilinx provides a user interface of 256-bits wide, we only fetch 4 words at a time. An interesting speed bump could come from a 512-bits fetch, that is: 8 words. The idea is to group the data by packs of 512 bits, and then send two packets of 256 bits with the memory controller, so that it could perform a full burst of 8 words.

Unfortunately, no change in the timing was observed: the memory accesses seemed to have the same latency as before.

4.5 Comparison with a CPU implementation

This section presents a comparison between the FPGA implementation and a CPU version. The settings of the FPGA were presented earlier in chapter 3. We will take the most efficient version, Here are the specifications of the CPU and the memory:

- Processor Intel Core i5-4258u
- 2.40 GHz base frequency
- 2.90 GHz turbo frequency
- 28W maximal power
- 4GB 1600MHz DDR3

The version against which is tested the FPGA version is a state-of-the-art open source CPU implementation developed by [31]. Here are the time and power requirements of an execution of this CPU algorithm, as well as the previously obtained measurements for the FPGA:

	Power [W]	**Time to complete [s]**	**Energy [J]**
FPGA	5.967	25.05	149.45
CPU	18.47	2.104	38.86

A note on the measurements of the CPU: the time should be accurate enough, but the power is subject to variations. The sample was taken without any other program running, but the OS. Only a peak power consumption could be observed. The figures provided above are mean values taken over 20 samples.

As expected, the power consumption of the CPU is higher than the one of the FPGA (an increase of 209%). However, the CPU performs better according to the time (91 % better).

But the comparison is not fair: the memory controller of the FPGA runs at 400MHz, whereas the memory of the test computer runs at 1600MHz. The timing must be adjusted. Here is an estimation of the proportion of time spent by each step of the algorithm in computations, and in memory accesses (of course, many computations occur in parallel of the memory accesses; we only count here the ones that occur without any parallel memory access):

	Computations	**Memory accesses**	**Time [s]**
Blake2b	7.14%	92.86%	0.5871
Radix	11.63%	88.37%	19.0487
Collision	13.60%	86.4%	5.4109

If we weight the time spent in each of the three steps by the proportion of memory accesses, and divide this proportion by 4 ($\frac{1600}{400}$), we should obtain a fairer comparison since the memory seems under-utilized because of the memory-controller frequency.

$$\begin{aligned}
&0.5871 \times 0.9286 \times \frac{1}{4} + 0.5871 \times 0.0714 \\
&+ 19.0487 \times 0.8837 \times \frac{1}{4} + 19.0487 \times 0.1163 \\
&+ 5.4109 \times 0.864 \times \frac{1}{4} + 5.4109 \times 0.1360 \\
&= 8.506
\end{aligned}$$

This is an estimation of the time taken by the FPGA if it could operate the DDR3 with a 1600MHz clock like the computer. This leads to an energy consumption of 50.75 J per execution.

We see here that with equivalent memory properties, the FPGA gets closer to a state-of-the-art CPU version, but it is not yet as efficient. A deep dive into the memory controller mechanics could allow the 8-burst feature to work properly, giving to the FPGA the final needed boost to be more efficient than a CPU.

Chapter 5

Return on investment

Since we use this FPGA in the context of blockchains and cryptocurrencies, one last important part to explore is the financial aspect, and determine if the implementation could be profitable.

Contents

5.1 Costs

The initial development of the FPGA implementation must be taken into account. In a presentation of the FPGA version of Equihash for the AION Foundation, it is said that a team of three people worked during one month [17] to finish a proof-of-concept. Accounting the necessary time for an FPGA engineer to adapt it to a platform, improve it, and test it, and make it market-ready, we will add between one and three months.

Best case	**Average case**	**Worst case**
4 months	5 months	6 months

The team that develop the FPGA needs to be paid. Some salaries for FPGA Engineers may be found on [12] for different companies.

Best case	**Average case**	**Worst case**
2580€/month	3133€/month	4333€/month

The last cost is the FPGA itself. With minimal requirements for the number of logic cells, BlockRAMs available, and IOs, we can find hardware starting at 30€. Regarding the memory, a chip of 1GB DDR3 can be bought for less than 15€.

Best case	**Average case**	**Worst case**
≈45€	≈60€	≈75€

In this context, energy is free as it is the overflow of solar panels production. It is thus not taken into account.

5.2 Revenues

We will take the best performance achieved with the implementation on the FPGA: 25 seconds to achieve one execution. As Equihash produces 2 solutions in average, this gives a hash rate of $\frac{1}{25/2} = 0.08$ hash/s.

The number of hashes does not imply much: it must be compared to the difficulty to mine a cryptocurrency (i.e. the filter value under which the hash value must be), and the reward earned when a block is mined.

For the most popular Equihash-based cryptocurrency, Zcash, we can compute the estimated revenues by taking into account the hash rate, the difficulty, and the coin reward of a block.

Tools can compute this automatically [7]. The results are, at the very least, underwhelming: if we let the FPGA run continuously for a year, the revenues are... 0.001777€. Not enough to reimburse the cost of the hardware only in a lifetime, without speaking of the development costs. The other considered cryptocurrency, Bitcoin Gold, has a smaller difficulty, but also a smaller value. Since Zcash had a difficulty of 97.8×10^6 and a value of ≈47€, and Bitcoin Gold has a difficulty of 317×10^3 and a value of ≈13€, we can roughly compute the expected yearly gains of mining Bitcoin Gold: 0.18€.

A best case/worst case analysis was intended, but seeing these numbers, the results can be anticipated: even with the best case, none of these cryptocurrencies can be mined for profitability.

As we look at the market, it is not so surprising anymore. One of the most powerful and recent ASIC for Equihash, the Antminer Z11 [34], that has a power consumption of 1418W, generates 135,000 solutions per second, for a mere profit of 2.8€per day. It came out in April 2019, and in August 2019, it is already close to non-profitability.

CPUs are no competitors either: even with a slightly higher hashing rate, it could never reimburse its purchase price of 315€.

This small case study shows that even more important than the hardware used and the energy consumption, it really is the considered cryptocurrency that is a deciding factor in profitability: if it is too recent, it is easy to mine but the coin has little to no real value. And if it is too common, the difficulty filter makes it too hard to mine. The window between the two states moves fast; possibly too fast to develop and deploy a working version of a proof-of-work miner on a platform that is not widely used.

Chapter 6

Conclusion

We explained more deeply how the blockchain works, and how memory-hard proof-of-works are worth exploring to counter the growth of mining ASICs.

A large pool of interesting cryptocurrencies was reviewed. This led to the comparison, and the choice, of a memory-hard proof-of-work to further study: Equihash.

The algorithm was deeply described, to better understand how such an algorithm can be implemented in hardware, but also to consider possible improvements. We tried these improvement ideas, and explained their impact in terms of efficiency, as well as space utilization.

Compared to the open source implementation, we can now expect a constant number of solutions, as well as a near 100% boost in terms of time, with only a small effect the power consumption. Internal resources offered by the FPGA are used as much as possible. There is room for improvement with the better use of the DDR3 8-burst characteristic, which was ineffective in this design.

The comparison with the CPU implementation showed that the memory was indeed the bottleneck of such memory-hard algorithm. With an appropriate comparison, we observe that an FPGA comes close to the efficiency of a classic processor.

Finally, the small feasibility study showed that the cryptocurrencies market is cruel towards commodity hardware, even with memory-hard proof-of-works. Sooner or later, ASICs will find their way in the race. The real interest is then to know if there is enough time to develop an algorithm, and make profit from it. For Equihash, it is too late already.

On a personal note now, I learned a lot doing this work. As a computer

engineering student, working with a large design in hardware was new for me, and out of my comfort zone. But as it was maybe harder, this was also the opportunity to discover a lot about hardware design technologies and tools. It is impressive to observe how fast cryptocurrencies progress and adapt to the market. In the end, I enjoyed working on a practical case, from the choice of the algorithm, to its implementation and analysis.

Appendix A

In this appendix, various results are shown. They were not crucial to present in the main section, but are available here for the sake of completeness.

Contents

A.1 Top 50 cryptocurrencies

Table A.1 shows the top 50 cryptocurrencies exchanged on March 4, 2019. Data come from the CoinMarketCap website[6]. If a proof-of-work is used as a consensus protocol, the algorithm used is specified. Gray lines identify the memory-hard algorithms.

#	**Name**	**Consensus**	**Algorithm**	**MH?**	**Note**
1	Bitcoin	PoW	SHA-256		PoW = Proof of Work
2	Ethereum	PoW	Ethash		Imminent transition to PoS
3	Ripple				Protocol of payment, not minable
4	EOS	PoS			PoS = Proof of Stake
5	Litecoin	PoW	scrypt	Yes	ASICs exist
6	Bitcoin cash	PoW	SHA-256		
7	Stellar				Protocol of payment, not minable
8	TRON	PoS			
9	Bitcoin SV	PoW	SHA-256		
10	Cordano	PoS			
11	Monero	PoW	Cryptonight	Yes	
12	Iota				DAG based, no blockchain
13	Dash	PoW	X11		
14	Neo	dBFT			dBFT = Delegated Byzantine Fault Tolerance, derived from PoS
15	Ethereum Classic	PoW	Ethash		
16	NEM	PoI			PoI = Proof of Importance
17	Zcash	PoW	Equihash	Yes	
18	Waves	PoS			
19	Ontology	dBFT			
20	Tezos	PoS			
21	VeChain	PoA			PoA = Poof of Authority
22	Dogecoin	PoW	scrypt	Yes	
23	Bitcoin Gold	PoW	Equihash	Yes	
24	Qtum	PoS			

25	Decred	PoW+PoS	Blake		
26	Lisk	dPoS			dPoS = delegated Proof of Stake
27	ICON	dBFT	SHA-256		
28	Bytecoin	PoW	Cryptonight	Yes	
29	ABBC	PoW	X13		
30	Digibyte	PoW	SHA-256/scrypt		
31	Steem	PoS			
32	Bitshares	dPoS			
33	Nano	dPoS			
34	Bitcoin Diamonf	PoS			
35	Aeternity	PoW + PoS	CuckooCycle	Yes	
36	Komodo	dPoW			dPoW = delayed Proof of Work
37	Verg	PoW	Multiple		
38	Siacoin	PoW	Blake		
39	Bytom	PoW	Tensority		
40	Stratis	PoS			
41	Ravencoin	PoW	Mutiple		16 algos, dependent of previous blocks
42	Cryptonex	PoS			
43	Ark	dPoS			
44	Electroneum	PoW			ASIC friendly!
45	Ardor	PoS			
46	Factom	?			Infinite inflation
47	Hypercash	PoW/PoS	Blake		
48	MOAC	PoW	Ethash		
49	PIVX	PoS			
50	ProjectPai	PoW	SHA-256		

Table A.1: List of 50 mots exchanged cryptocurrencies, March 4, 2019

A.2 Full results for the experiment on the impact of the number of collisions supported

# Used slices	6013	15%
# LUT/FF pair fully used	8073	42%
# LUT/FF pairs with unused LUT	4216	22%
# LUT/FF pairs with unused FF	6636	35%
# LUTs	14709	9%
# FFs	13554	4%
# BRAMs	29	3%

Table A.2: Utilization of the FPGA resources for 4 collisions

# Used slices	6388	16%
# LUT/FF pair fully used	7996	38%
# LUT/FF pairs with unused LUT	5043	24%
# LUT/FF pairs with unused FF	7610	36%
# LUTs	15606	10%
# FFs	13565	4%
# BRAMs	29	3%

Table A.3: Utilization of the FPGA resources for 6 collisions

# Used slices	6145	16%
# LUT/FF pair fully used	7991	40%
# LUT/FF pairs with unused LUT	4445	22%
# LUT/FF pairs with unused FF	7454	37%
# LUTs	15445	10%
# FFs	13556	4%
# BRAMs	29	3%

Table A.4: Utilization of the FPGA resources for 7 collisions

Power by type [mW]		**Power by hierarchy [mW]**	
Clock	310	Memory controller	142
Logic	53	UART	3.1
Signals	54	Radix	32
BRAM	5	Blake2b	35.7
MMCM	123	Collision	8.3
IOs	1990	Equihash_state	2.6
Leakage	2928	Mem_gasket	2.6
Total	5458	Total	230.9

Table A.5: Power consumption for 4 collisions

Power by type [mW]		**Power by hierarchy [mW]**	
Clock	332	Memory controller	143
Logic	52	UART	3.4
Signals	59	Radix	32.2
BRAM	5	Blake2b	35.6
MMCM	123	Collision	11.6
IOs	1990	Equihash_state	2.8
Leakage	2928	Mem_gasket	2.7
Total	5485	Total	235.4

Table A.6: Power consumption for 6 collisions

Power by type [mW]		**Power by hierarchy [mW]**	
Clock	320	Memory controller	142
Logic	52	UART	2.9
Signals	55	Radix	30.4
BRAM	5	Blake2b	34.8
MMCM	123	Collision	12.1
IOs	1990	Equihash_state	2.6
Leakage	2928	Mem_gasket	3
Total	5469	Total	231

Table A.7: Power consumption for 7 collisions

A.3 Full results for the experiment on the number of radix bits

# Used slices	6465	17%
# LUT/FF pair fully used	9348	44%
# LUT/FF pairs with unused LUT	4175	19%
# LUT/FF pairs with unused FF	7561	35%
# LUTs	16909	11%
# FFs	14837	4%
# BRAMs	29	3%

Table A.8: Utilization of the FPGA resources for 5 radix bits

# Used slices	7569	20%
# LUT/FF pair fully used	10890	44%
# LUT/FF pairs with unused LUT	4421	18%
# LUT/FF pairs with unused FF	8894	36%
# LUTs	19784	13%
# FFs	16496	5%
# BRAMs	29	3%

Table A.9: Utilization of the FPGA resources for 6 radix bits

# Used slices	8667	23%
# LUT/FF pair fully used	15047	51%
# LUT/FF pairs with unused LUT	3553	12%
# LUT/FF pairs with unused FF	10836	36%
# LUTs	25883	17%
# FFs	19834	6%
# BRAMs	29	3%

Table A.10: Utilization of the FPGA resources for 7 radix bits

# Used slices	12014	31%
# LUT/FF pair fully used	21476	52%
# LUT/FF pairs with unused LUT	3809	9%
# LUT/FF pairs with unused FF	15269	37%
# LUTs	36744	24%
# FFs	26494	8%
# BRAMs	29	3%

Table A.11: Utilization of the FPGA resources for 8 radix bits

Power by type [mW]		**Power by hierarchy [mW]**	
Clock	290	Memory controller	143
Logic	80	UART	2.9
Signals	80	Radix	80.4
BRAM	5	Blake2b	36.2
MMCM	123	Collision	11.8
IOs	1991	Equihash_state	3.5
Leakage	2928	Mem_gasket	2.4
Total	5429	Total	283.5

Table A.12: Power consumption for 5 radix bits

Power by type [mW]		**Power by hierarchy [mW]**	
Clock	322	Memory controller	143
Logic	123	UART	3.2
Signals	108	Radix	149
BRAM	5	Blake2b	38.3
MMCM	123	Collision	11.7
IOs	2041	Equihash_state	2.8
Leakage	2932	Mem_gasket	2.7
Total	5649	Total	354

Table A.13: Power consumption for 6 radix bits

Power by type [mW]		**Power by hierarchy [mW]**	
Clock	342	Memory controller	145
Logic	215	UART	3.1
Signals	219	Radix	316.7
BRAM	5	Blake2b	35.2
MMCM	207	Collision	17.9
IOs	2040	Equihash_state	3.4
Leakage	2938	Mem_gasket	3.4
Total	5967	Total	646

Table A.14: Power consumption for 7 radix bits

Bibliography

[1] Martin Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. Moderately hard, memory-bound functions. *ACM Trans. Internet Technol.*, 5(2):299–327, May 2005. ISSN 1533-5399. doi: 10.1145/1064340.1064341. URL `http://doi.acm.org/10.1145/1064340.1064341`.

[2] atferrari. What clock frequency for a certain baud rate? Available at `https://forum.allaboutcircuits.com/threads/what-clock-frequency-for-a-certain-baud-rate.61514/`, November 2011.

[3] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. pages 119–135, 06 2013. doi: 10.1007/978-3-642-38980-1_8.

[4] Alex Biryukov and Dmitry Khovratovich. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. Cryptology ePrint Archive, Report 2015/946, 2015. Available at `https://eprint.iacr.org/2015/946`.

[5] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. 2019. Available at `https://arxiv.org/pdf/1710.09437.pdf`.

[6] CoinMarketCap. All cryptocurrencies. Available at `https://coinmarketcap.com/all/views/all/`, last viewed on March 4, 2019.

[7] CryptoCompare. Zcash mining profitability calculator. Available at `https://www.cryptocompare.com/mining/calculator/zec?HashingPower=0.08&HashingUnit=H%2Fs&PowerConsumption=0&CostPerkWh=0&MiningPoolFee=0`.

[8] Axel de Vries. Bitcoin's growing energy problem. *ScienceDirect*, 2018. Available at `https://doi.org/10.1016/j.joule.2018.04.016`.

[9] Monero Documentation. Cryptonight. Available at `https://monerodocs.org/proof-of-work/cryptonight/`.

[10] Earl Mai for the Aion Foundation. aion_epic_fpgaminer, 2018. Available at https://github.com/aionnetwork/aion_epic_fpgaminer.

[11] Lou Frenzel. What's the difference between bit rate and baud rate? *Electronic Design*, April 2012. Available at https://www.electronicdesign.com/communications/what-s-difference-between-bit-rate-and-baud-rate.

[12] Glassdoor. Fpga engineers. Available at https://www.glassdoor.fr/Emploi/fpga-engineer-emplois-SRCH_KO0,13.htm.

[13] Sourav Sen Gupta. *Blockchain*. John Wiley & Sons, Inc, 2017.

[14] Intel. Processeur intel core i5-4258u, technical sheet. Available at https://ark.intel.com/content/www/fr/fr/ark/products/75990/intel-core-i5-4258u-processor-3m-cache-up-to-2-90-ghz.html.

[15] jimblom. Serial communication. Available at https://learn.sparkfun.com/tutorials/serial-communication/all.

[16] Ben Kaiser, Mireya Jurado, and Alex Ledger. The looming threat of china: An analysis of chinese influence on bitcoin. *CoRR*, abs/1810.02466, 2018. URL http://arxiv.org/abs/1810.02466.

[17] Earl Mai. Presentation - aion hardware community: Equihash fpga. Available at https://youtu.be/A_sliZWWhfo.

[18] Micron. *TN-40-07: Calculating Memory Power for DDR4 SDRAM*. Micron. Available at https://www.micron.com/-/media/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf.

[19] Satoshi Nakamoto. 2008. Available at http://www.bitcoin.org/bitcoin.pdf.

[20] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016. ISBN 9781400884155.

[21] F.X. Olleros and M. Zhegu. *Research Handbook on Digital Transformations*. Research Handbooks in Business and Management series. Edward Elgar Publishing, 2016. ISBN 9781784717766. URL https://books.google.be/books?id=1_QCDQAAQBAJ.

[22] Colin Pericval. Stronger key derivation via sequential memory-hard functions. January 2009.

[23] Ameer Rosic. 17 blockchain applications that are transforming society, 2017. Available at https://blockgeeks.com/guides/blockchain-applications/.

[24] Fahad Saleh. Blockchain without waste: Proof-of-stake. 2019. Available at https://dx.doi.org/10.2139/ssrn.3183935.

[25] Karuna Sehgal. An introduction to bucket sort. Available at https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124, February 2018.

[26] Tony Storey. Fifo buffer module with watermarks (verilog and vhdl). Available at https://www.digikey.com/eewiki/pages/viewpage.action?pageId=20939499.

[27] Joachim Strömbergson. blake2b, 2018. Available at https://github.com/secworks/blake2.

[28] Suse. Definition commodity hardware. Available at https://susedefines.suse.com/definition/commodity-hardware/.

[29] CoinBundle Team. Blockchain & internet of things (iot), 2018. Available at https://medium.com/coinbundle/blockchain-internet-of-things-iot-be58703617c9.

[30] Greene Tristan. A brief history of bitcoin mining hardware. Available at https://thenextweb.com/hardfork/2018/02/02/a-brief-history-of-bitcoin-mining-hardware/.

[31] Tromp. Multi-parameter equihash proof-of-work multi-threaded c solvers. Available at https://github.com/tromp/equihash.

[32] John Tromp. Cuckoo cycle: A memory bound graph-theoretic proof-of-work. volume 8976, pages 49–62, 01 2015. ISBN 978-3-662-48050-2. doi: 10.1007/978-3-662-48051-9_4.

[33] Carmela Troncoso. Ecole polytechnique fédérale de lausanne, com301 computer security, lecture 4, 2018.

[34] ASIC Miner Value. Bitmain antminer z11. Available at https://www.asicminervalue.com/miners/bitmain/antminer-z11.

[35] David A. Wagner. A generalized birthday problem. In *CRYPTO*, 2002.

[36] Wikipedia. Baud. Available at https://en.wikipedia.org/wiki/Baud, February 2019.

[37] Wikipedia. Field-programmable gate array, 2019. Available at https://www.watelectronics.com/fpga-architecture-applications/.

[38] Wikipedia. Ddr3 sdram. Available at https://en.wikipedia.org/wiki/DDR3_SDRAM, AUgust 2019.

[39] Wikipedia. scrypt. Available at https://en.wikipedia.org/wiki/Scrypt#Litecoin_proof-of-work, May 2019.

[40] Xilinx. *Virtex-6 FPGA Memory Interface Solutions.* Xilinx, March 2011. Available at https://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf.

[41] Xilinx. *ML605 Hardware User Guide (UG534).* Xilinx, February 2019. Available at https://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**