AION FPGA Digital Design Architecture

Earl Mai
emai@epicblockchain.io
June 6th, 2018

Revision 0.6

Scope

The document provides a detailed digital design architecture of an Equihash solver covering the major design blocks targeted for Field Programmable Gate Array (FPGA) usage.

The design sacrifices some complexity for an easier understanding of the design which is more suitable as a learning tool. The reader should be able to understand the motivation behind the design without having a deep knowledge in digital design.

Equihash

https://www.cryptolux.org/images/b/b9/Equihash.pdf
https://en.wikipedia.org/wiki/Equihash
https://github.com/aionnetwork/aion_miner/wiki/Aion-equihash_210_9--specification-and-migration-guide.

Equihash is an asymmetric Proof-of-Work (PoW) algorithm based on the generalized birthday problem. Please see the linked documents above for a detailed description.

Parameters

The design will allow for flexible parameters to allow quicker simulation and verification of the design. A list of valid and tested parameters will be supplied along with the Aion specific parameters. Smaller parameters will be used in testbenches for quicker verification and simulation.

n	***k***
96	5
200	9
210	9

Parameter *n*

The message size produced from the blake2b (b2b) used as the input data to the Equihash algorithm. For Aion's specific use case *n* will be 210.

Parameter *k*

The number of stages -1 when solving the Equihash algorithm. For Aion's specific use case *k* will be 9 (10 stages).

Parameter *N*

The number of elements used as the input data when solving the algorithm. *N* is defined as:

$$N = 2^{(\frac{n}{k+1})+1}$$

The number of indices for Aion's specific use case is 2^{22} or ~ 4 million elements.

Parameter *d*

The size of message used to find collisions in each stage is defined as:

$$d = \frac{n}{k+1}$$

The message size for Aion's specific use case is 21 bits.

Sorting Algorithm

The Equihash algorithm requires collision detection of subsets of the message. There are advantages in sorting at the cost of complexity.

Various sorting algorithms were reviewed for use with performance and memory consumption as the defining criteria. What is also important is to understand that software sorting algorithms do not necessarily translate well into hardware implementations due to the complexity, lack of parallelism and the use of dynamic memory structures. Algorithms that require memory intensive swapping are also not suitable due to Equihash being a memory bound algorithm.

Bubble Sort

Bubble sort is a logically simple design with a large penalty in performance and bandwidth due to constant memory swapping. With the addition of a data cache before the memory controller and sequential nature in access patterns bandwidth utilization can be reduced but performance is lacking.

Bitonic Sort

https://ganges.usc.edu/svn/pg/pubs/preprint/ren-fpga-2015.pdf

Bitonic sort is a good candidate for a hardware implementation due to the algorithm being parallel in nature. By creating a pipelined sorting network and instantiating multiple blocks with crossbars, the design could support streaming data at the cost of FPGA area. The dedication of area to sorting may be beneficial since the bulk of time in Equihash is spent generating input (blake2b) and finding pairs (sorting) but to fully sort a variety of FPGA sizes, there are simpler methods that can achieve similar performance.

Radix / Bucket Sort

The sorts are very similar in nature and offer the best balance between logic complexity and performance. The bucket approach allows for several stages of sorting by breaking up the message size *d* into smaller segments which dictate the number of buckets required. By snooping the data being written into memory in previous stages, we are able to predetermine bucket sizes to avoid the dynamic allocation of memory.

Field Programmable Gate Array (FPGA)

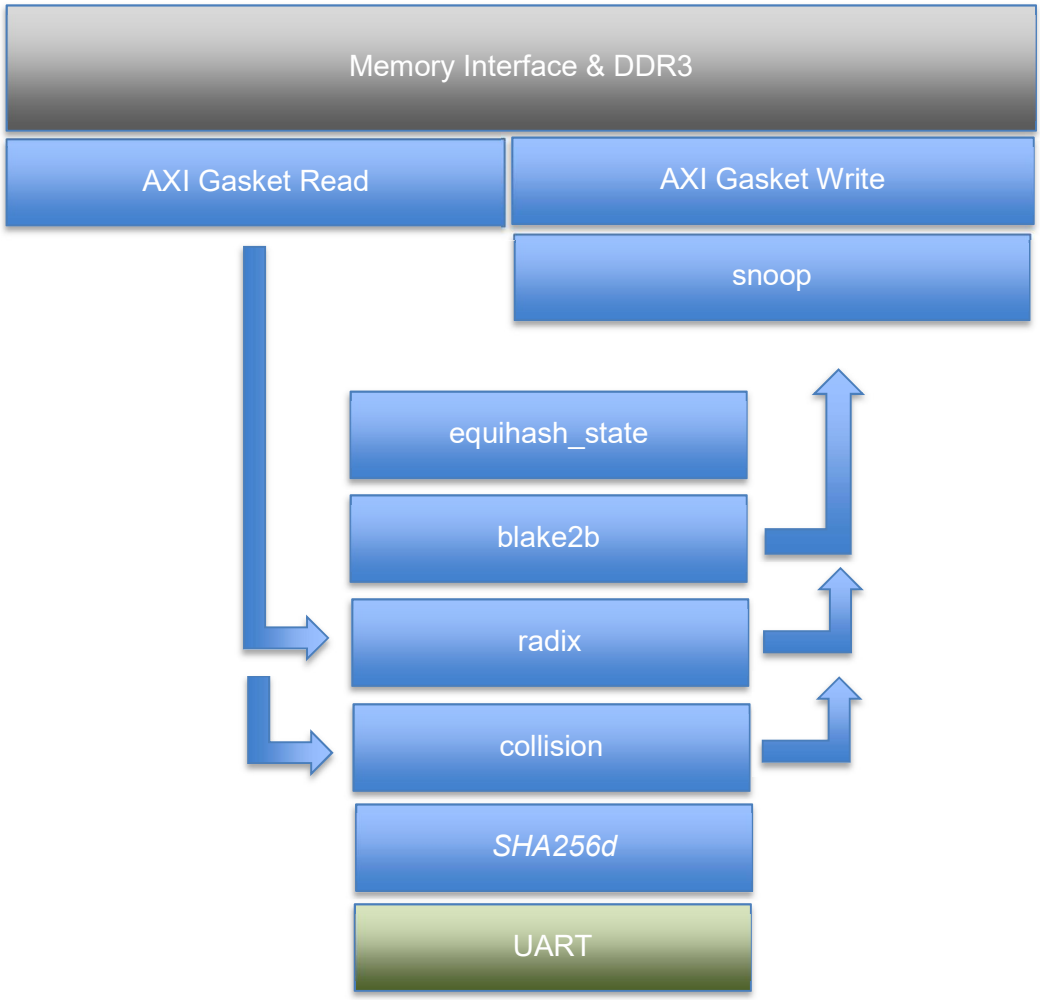
https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html

The device ePIC will use for development is the KC705 Evaluation Kit from Xilinx. The key features include:

- Kintex 7 FPGA (XC7K325T-2FFG900C)
    - 326, 080 Logic cells
    - 840 DSP Slices
    - 16, 020 Kb Block RAM
- 1GB DDR SODIMM @ 800MHz
- UART to USB Bridge
- PCIx8 Express Edge Connector

Top Level Architecture

A diagram of the top level modules can be found below. In the subsequent sections, details of each module will be described to allow understanding of the design methodology.



The design contains three clock domains, memory clock (grey), hash clock (blue) and communication clock (green).

blake2b block

https://blake2.net/blake2.pdf
https://github.com/secworks/blake2
https://github.com/mikalv/blake2-1

Please refer to the above documentation for detailed description of the algorithm.

The blake2b module can generate a hash output every 27 clocks. Due to the input size of 140 bytes the message is processed in two passes making the final hash output valid after 54 clocks.

Data output from the module will be split and written to the Intermediate Work (current) space using the index as its address. A signal will also be sent to allow the snoop block to increase its counters allowing sorting to position data efficiently without dynamic allocation of memory.

Note that increasing the number of blake2b cores can decrease the amount of time it takes to generate the initial data at the cost of area. Investigating midstate like optimizations from can lead to performance doubling due to the nonce bits residing at the bottom of the header.

Aion specific implementation uses a key which is defined as “AION0PoW2109"

Memory block

The memory subsystem contains the bulk of the design logic consisting of the hash core, memory controller and bucket counters used to aid the sorting algorithm.

DDR3 Memory Controller subblock

The memory controller used for this project will be created using Xilinx Memory Interface Generator (MIG). Depending on data width and bank configuration of the memory, several configurations may be possible to increase bandwidth using multiple controllers.

Initially the native MIG interface was used due to its simplicity but came at a price of having a fixed data bus width of 512 bits. Currently the design is being updated to support the AXI protocol which allows for data bus width of 256 bits but adds complexity to the driving logic.

For the KC705 evaluation kit will follow be created using the following steps from the Xilinx MIG example design:
https://www.xilinx.com/support/documentation/boards_and_kits/kc705/2014_3/xtp196-kc705-mig-c-2014-3.pdf

DDR3 Memory

The memory on board every device will vary in size and type ranging from DDR-DDR4 as well as HBM. The Equihash algorithm is primarily limited by memory bandwidth and as such one can typically find the maximum solve rate based on memory bandwidth.

For the KC705 evaluation kit comes with 1GB DDR3 from Micron with a part number MT8JTF12864HZ-1G6 with a rated peak bandwidth of 12.8GB/s. https://www.micron.com/~/media/documents/products/data-sheet/modules/sodimm/ktf8c128_256_512x64hz.pdf

The memory controller will run at 4 times the design clock allowing a 256bit interface for memory accesses. This fits inline with most of the accesses that are required due to the 210bit message concatenated with addresses. The pairs binary tree will require 64bit accesses so the memory bus will be qword enabled.

AXI Gasket

The memory interface from all blocks utilize the simple RTS/RTR protocol to send and receive memory. This is done for portability to other memory controllers but to utilize the MIG AXI controller, a gasket needs to be added supporting AXI-4.

The write interface is simpler once the data has been sent out, there is no waiting for any return. The memory controller ensures coherency so that data is written into memory before subsequent reads. The interface will support as many writes until the memory controller is saturated.

The read interface requires data to be returned after an address request thus requiring more logic. Only one read request will be supported at a time which is inefficient. Therefore AXI burst will be supported so that one request can return multiple data. Further changes can be added to fully support tagging which can allow out of order read returns to occur.

snoop block

The snoop block is a parameterized module containing counters to determine the bucket sizes for sorting. By marking every write of message data to memory (including initial blake2b data), the snoop block will increment counters until a stage has finished. The resulting counters will be used to divide buckets in memory for the radix sort.

Two sets of counters will be used and ping ponged back and forth for the current stage and next stage buckets.

The parameters include:

- Number of buckets
- Size of counters (based on N)

Due to verilog limitations, one can either specify output as an array or pre-process the verilog to account for the differing number of bucket output pointers.

equihash_state block

The main state machine of the design which contains all control signals controlling the blocks in the Equihash digital design.

The block triggers and start and finish of every subblock and maintains count as the algorithm progresses.

Typical control flow:

- Wait for UART input
- Start Stage 0
    - Generate blake2b data
    - Start Radix Sort block
    - Start Collision block
- End Stage 0
- Start Stage 1
    - Start Radix Sort block
    - Start Collision block
- End Stage 1

…

…

- Start Stage k-1
    - Start Radix Sort block
    - Start Collision block
- End Stage k-1
- Start Stage *k*
    - Start Radix Sort block
    - Start Collision block
    - Start Binary Search
- End Stage *k*
- Wait for UART output

Pointer subblock

This module contains the base memory pointers to all the various buffers in memory and recycles buffers as data becomes stale. A few factors are involved when deciding how to organize memory due to the bit width of the system. Storing in multiples of the bit width allows for fewer accesses as well less wasted space.

Due to sorting, when Intermediate Memory is sorted to find collisions, a pointer to the pairs used to generate the work must be stored as well.

All spaces are accessed in 256bit addresses while the Pairs Memory Pointers are addressed on a 64bit bases due to the compacting of two 32bit addresses.

Blake Memory Pointer

The initial dataset generated by the blake2b module is kept for verification purposes and also used as the leaf nodes of the binary tree. The data generated by this block is roughly 256MB and thus this sets the buffer size for all the remaining buffers.

Intermediate Memory Pointers

The Intermediate Memory contains the XOR work for each stage and can be discarded as the data becomes stale. Data is always prepended with the indice or pairs pointer when written to memory.
There are 2 Intermediate Memory spaces, one for current work and one for next work.
Current work gets discarded and next work becomes current as the stages progress. Data that overflows these memories are discarded as well thus leaving a small possibility of missed solutions.

Pairs Memory Pointers

The collision pairs are stored as a binary tree and saved for every stage. Each entry is a node in a binary tree, hence contains the address to two children. Since the initial blake2b generated dataset will be kept, leaf nodes can be found by checking the address is in the Blake Memory space.

Radix Sort block

The Radix Sort block reads the current Intermediate Memory containing the original blake2b data or XOR work data. The data is then written back into the corresponding buckets based on the bucket counters and iterated depending on the block parameters.

The message size d is divided into buckets depending on the desired number of iterations through elements N. Smaller number of buckets require more passes through memory so varying these parameters have a direct impact on performance.

For Aion's $(n,k) = (210,9)$ the message size is 21bits can be sorted with 8 buckets and 7 passes or 128 buckets and 3 passes with a large increase in the number of bucket counters required. These are just examples but after analysis 16 buckets and 5 passes allows for better timing, area and performance trade offs.

Collision block

The core functionality of this block is to compare the message of the XOR work and determine if there are collisions. Data is read from the sorted Intermediate Memory (current) and pairs are written out to Pairs Memory for the next stage. Resultant XOR work is written out to Intermediate Memory (next).

The block buffers multiple XOR work items in case of multiple collisions up to a maximum of 5. This value is parameterized and can be increased or decreased at the cost of finding all solutions.

A streaming buffer (FIFO) is used for the input and controlled with a watermark always allowing for the collision logic to have work. When the buffer is depleted below the watermark, the buffer is filled ensuring that it is never empty. Sizing of this buffer is configurable and should be set a depth to balance out memory read latency. With the additional use of AXI burst reads and known memory latency, the user can specify the depth to ensure starvation never occurs.

When collisions are detected, pairs and resultant XOR data is written back to memory in preparation for the next stage. As higher number of data elements collide, the streaming buffer reads are stalled to allow for adequate cycles to write out into memory.

During the final stage of processing ($k+1$) the complete pair solution needs to be recovered for outputting to SHA256 or UART. This is achieved by recursively reading the search tree until leaf nodes are reached. To preserve ordering, the 64bit pair data is split into 32bit addresses and the first half is processed while the second half is pushed into the FILO. This buffer is required sized to ensure the full tree can be read. While this process is running in parallel, continuing operation of the collision block occurs until a second full solution is found in which case the streaming buffer will be stalled. Since 2 solutions are expected to be found, the odds of this are low and there is no need to support more than one recursive search.

Note Pairs data does not require a counter since they are referenced via a binary tree but Intermediate Memory is sorted so the size must be known.

Double SHA256 block

https://github.com/secworks/sha256

The final stage of the Equihash algorithm takes the initial input message from blake2b, size of pairs and resultant pair list and runs it through a double SHA256. The module will be able to buffer up to 3 solutions using a lightly pipelined double SHA256 core to save area. The difficulty comparator will be check SHA256 output and final values will be sent off to the UART module if conditions are satisfied.

This block is optional since verification of difficulty threshold is better suited to be done on the host. This allows all the pairs of data to be transmitted back to the host along with nonce for verification as well as difficulty checking.

Universal Asynchronous Receiver-Transmitter (UART) block

This module is the sole interface between the FPGA and host. The design runs on a separate clock domain and fifos are used to cross clock domain between the UART and main design. The maximum transfer speed of 115200bps can be increased if necessary outside the standard baud rates but for the current design, communication will not be the bottle neck.