# Zcash FPGA acceleration engine

Version 1.4.2 release

Ben Devlin
bsdevlin@gmail.com

GitHub repo: https://github.com/bsdevlin/zcash-fpga/

# Version history

- v1.1
    - First major release of the code, includes many reusable logic cores, along with the equihash engine, secp256k1 signature verification engine, and bls12-381 coprocessor with Fp and $Fp^2$ point multiplication (pairing to be implemented in v1.2)
    - Top level module for the Zcash acceleration engine
    - Top level board files for both Bittware VVH and Amazon AWS EC2 F1 FPGAs
        - bls12-381 coprocessor so far has only been tested on AWS
    - Document still missing content for some sections, will be completed in v1.2
- v1.2
    - bls12-381 coprocessor
        - Added optimal ate pairing module (instruction ATE_PAIRING)
        - Removed some instructions that were not used for control
        - Updated diagrams and performance numbers
- v1.3
    - bls12-381 coprocessor
        - Added instructions for MILLER_LOOP and FINAL_EXP to allow for multi-pairing operations
        - Added $Fp^{12}$ to the MULT instruction
        - Removed FPOINT_MULT instructions as they can all be covered by POINT_MULT
    - Changes to systemverilog to improve timing
    - Added AWS builds
        - Added sample output of test program
- v1.4
    - Added accum_mult_mod block to /ip_cores/ and modified bls12-381 coprocessor to use this
        - Modular multiplier using carry save adders for multiplication and RAM tables for reduction, 3X performance compared to previous Karatsuba multiplier
- v1.4.1
    - Fix bug with AWS build not working due to files not getting read properly, added a parameter to bls12-381 top level “USE_KARATSUBA” and zcash_fpga_pkg.sv “BLS12_381_USE_KARATSUBA” so that the old multiplier can optionally be used
    - Added comment that for simulating on AWS this parameter should be changed to “YES”
    - Added comment that Python 3 is needed for the accum_mult_mod script
- v1.4.2
    - Fixed bug with accum multiplier RAM loading in bls12-381 coprocessor (if USE_KARATSUBA==NO), added to testbench
    - Added AFI image using new multiplier (100% BRAM and 0% URAM so we can avoid loading RAM contents at runtime - URAM does not support init files for RAM contents - updated python generation script for accum_mult_mod multiplier to reflect this)

# Terms used

| | | | |
|---|---|---|---|
| FP (Field point) | FE (Field element) | JB (Jacobian) | AF (Affine) |
| FPGA (Field programmable gate array) | EC (Elliptic curve) | SW (Software - generally meaning what runs on the CPU) | AXI (Advanced eXtensible Interface) |
| Non-adjacent form (NAF) | RAM (Random Access memory) | BRAM (Block RAM) - on Xilinx FPGAs | URAM (Ultra RAM) - on Xilinx FPGAs |
| zk-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) | ECDSA (Elliptic Curve Digital Signature Algorithm) | FFF (Fast Fourier transform) | |

# Overview

## Zcash FPGA project

Zcash FPGA acceleration engine is a FPGA system used to accelerate the Zcash network. The **first phase** is focused on accelerating verification components of the blockchain, and the **second phase** is focused on zk-SNARK acceleration and elliptic curve operations required. All code developed is written in system verilog and open source under the GPL 3.0 license, intended to be modular and parameterizable for reuse, and can be found at the GitHub repo linked on the first page of this document.

FPGA acceleration allows us to offload work to a chip that is configured at the gate level to do specific hardware functions, and can bring several **advantages** over a CPU implementation:

1. Can be configured for large parallelism - e.g.you could configure an FPGA to do 1000x 32bit multiplications all at the same time allowing for large throughputs
2. Specialized functions that an x86 processor takes many instructions to implement could be implemented as a single instruction on an FPGA
3. Low latency direct access to data - e.g. you could develop custom TCP/IP hardware on an FPGA bypassing a NIC card / having a CPU make decisions

But also has **disadvantages**:

4. Clock speed is much slower on FPGA (100MHz - 300MHz depending on logic implemented) compared to a CPU (3GHz+) with multiple cores
5. Getting data in and out of the FPGA from the CPU takes roughly ~300ns(PCIe roundtrip) which translates to ~1000 clock cycles on a CPU even before we start processing
    a. This is for an optimized core - AWS FPGAs used in this experiment take 1us+ roundtrip
6. Development cycle is much slower compared to CPU and not as easily accessible to a SW engineer

The goal of this work is not only to develop open source FPGA acceleration code for various Zcash systems and that can be of use to the wider community, but also to investigate/research the direction for future development (i.e. what cases are good candidates for acceleration and what cases are better left to SW).

## Interfaces and FPGA hardware

The FPGA engine is designed to either be implemented on a Bittware board (VU37P FPGA w/ 8GB HBM, 16GB DDR4) or run on an Amazon AWS EC2 F1 FPGA instance (VU9P w/ 64GB DDR4). Both FPGAs are the same generation and speed grade, but depending on the board clock rates on FPGA might have to be scaled so that timing closure can be met (AWS FPGAs require extra “glue” logic and seem to not meet timing as easily as the VU37P). I have tried to use non-vendor specific blocks where possible (i.e. BRAMs, core logic, is mostly written from scratch in systemverilog), but in case cases I have used Xilinx IP for simplicity (mainly in the AWS top level, where the .xci files are included in the /ip folder). It would not take much work to implement the same code on an Altera FPGA or older generation Xilinx FPGA.

Communication to FPGA is split into two main methods:

1. Based on commands that are formatted with a header, followed by optional data (inputs for the command). FPGA sends replies to SW after a command is completed or in the case of any errors. These are sent over an AXI4-stream interface.

2. Using an instruction memory and data register approach, SW has direct access to FPGA memory and can configure more complex logic flows. Interrupt commands can be implemented so FPGA will send data to SW without required polling of FPGA memory. This is used for the bls12-381 coprocessor in phase 2. These are sent over an AXI4-lite interface.

Depending on the FPGA board used communication is either exposed to SW through a C++ library over PCIe (when using AWS), or over USB-UART (when using the Bittware board). There are wrappers that convert the communication method to the internal FPGA AXI-lite and AXI-stream interfaces.

# Project goals

At a high level the FPGA architecture currently comprises several engines for dedicated tasks to handle the commands from SW, where more engines are to be added as development continues:

- Blake2b hash
- SHA256 hash
- Equihash verification engine
- Transparent signature verification engine (accelerate point multiplication on the secp256k1 curve)
- BLS12-381 coprocessor (accelerate EC operations on the bls12-381 curve such as point multiplication and pairing)

## Phase 1

Phase 1 is focused on offloading various aspects of verifying the Zcash block chain onto the FPGA. These will include:

1. A equihash verification engine, which can take in a block header + solution and verify it is correct, as well as other fields in the block header that require processing (such as hashing)
2. Verifying transparent transaction in the block chain, which will be done by implementing a secp256k1 engine that can take in signatures and verify their correctness.

## Phase 2

Phase 2 is focused on accelerating zk-SNARK operations.

1. This will be implemented a BLS12-381 coprocessor, where software can write instruction memory on the FPGA that will allow for chaining of multiple commands without having to send data in and out of the FPGA. This coprocessor will implement Fp, $Fp^2$, $Fp^6$, $Fp^{12}$ arithmetic over the bls12-381 curve, as well as several higher level operations such as inversion, calculating powers, calculating frobenius map, miller loop, final exponentiation, and optimal ate pairing. Software can read and write both data and instruction memory to poll the current status of the coprocessor, or interrupt instructions can be used to send interrupts back to SW when certain commands complete.

The main goals for acceleration using this coprocessor*:

- Generate a shielded Zcash (Sapling) transaction with acceleration from the coprocessor
- Sign a shielded Zcash (Sapling) transaction with acceleration from the coprocessor

*The processor has been implemented on FPGA but requires some work in Zcash's Rust code to correctly interface with the FPGA and utilize, which has been made a lower priority at the moment for this project. A Rust wrapper around the cpp FPGA library has been developed, but still requires work before it could be released into production and used with an AWS F1 FPGA instance.

# Implementation

## Overview

- **FPGA:**
  - Bittware XUPVVH dev board w/ Virtex UltraScale+ VU37P HBM VCU128-ES1 (8GB HBM, 16GB DDR4)
    - Interface to hose over UART (USB)
  - AWS EC2 F1 FPGA instance UltraScale+ VU9P (64GB DDR4)
    - Interface to host over PCIe
- **Software API:**
  - C++ library for AWS boards over PCIe - this is in the github repo aws/cl_zcash/software/runtime/zcash_fpga.hpp
    - A rust interface is in development and should be released in a later version, to allow the Zcash client to run on an AWS instance to utilize FPGA acceleration
  - USB-UART for Bittware boards using Python - this is in the github repo bittware_xupvvh/software/zcash_fpga.py

## FPGA Memory Map

The FPGA has 2 main methods of sending and receiving data, these are:

1. The AXI4 stream interface, which is used to send and receive commands and can be used with larger amounts of data (detailed in the next section).
2. The AXI4 lite memory map interface, mainly used for configuration, debug, instruction, and data memory. This is done via individual 32 bit writes and reads. The memory space of the FPGA is organized as:

| Name | Address range |
|---|---|
| Top level control and configuration | ** Not currently present in version v1.3 |
| Stream control module (only present on AWS builds) | 0x0 to 0xFFF |
| BLS12-381 coprocessor | 0x1000 to 0x4FFF |

(each regions memory section is detailed in the architecture section)

## Streaming commands

The streaming interface data is streamed from SW to FPGA with a 16 byte header at the very start, and then depending on the command or reply type from FPGA there can be a sub-header and additional inputs / outputs. All values here are little endian and length (len) is specified in bytes. The format of the header is:

```
typedef __packed__ struct {
```

```
  uint32_t cmd_type;    // This is the command type (given below) either from SW or
from FPGA
  uint32_t len;         // This is the total length in bytes of the packet either from
SW or from FPGA
} fpga_header_t;
```

Commands are capable of being sent back-to-back in the same stream, but the start of a new command must be aligned to an 8 byte boundary.

## SW to FPGA

These are the commands the FPGA is capable of receiving from SW.

### reset_fpga

cmd_type: 0x00000000
len: 8 (no additional data follows the header)

This command resets the FPGA internal logic logic to its initial state. This should be called when first connecting to the FPGA, or if any errors happen and the FPGA is unresponsive (if this command does not fix the problem you will need to reprogram the FPGA). The FPGA will send a **reset_fpga_rpl** to SW after it has been reset.

### get_fpga_status

cmd_type: 0x00000001
len: 8 (no additional data follows the header)

This command asks the FPGA to reply with the current status using a fpga_status_rpl message.

### verify_equihash

cmd_type: 0x00000100
len: 8 + 8 + length of block header (CBlockHeader) (1487 for N=200, K=9)

This command takes takes a block header and will verify the equihash solution is correct, according to Zcash protocol doc, and passes the difficulty filter. The FPGA will send a **verify_equihash_rpl** back to SW with the result of the check along with the index from the command so that it can be matched (in the case of multiple concurrent operations).

```
typedef __packed__ struct {
  fpga_header_t hdr;
  uint64_t      index;        // This index is returned with the result
  CBlockHeader  block_header; // Serialized data of block header class from Zcash code
block.h
} verify_equihash_t;
```

### verify_secp256k1_sig

cmd_type: 0x00000101

len: 8  + 8 + 160
This command verifies the signature used in a transparent transaction over the EC **secp256k1**.
Inputs are the hash H(m) of the message m, the signature (comprised of two values - s and $r_x$), and Q (public key of signer uncompressed). P is the base point of secp256k1 and stored on the FPGA. The FPGA then decodes this command into a series of instructions for the secp256k1 ECDSA core. An index is also given that it returned with the result to track multiple concurrent commands.

```
typedef __packed__ struct {
  fpga_header_t hdr;
  uint64_t      index;         // This index is returned with the result
  uint256_t     s;             // Signature
  uint256_t     r;             // Signature
  uint256_t     hash;          // Hash of message that was signed to be verified
  uint512_t     Q;             // Signers public key (uncompressed form)
} verify_secp256k1_sig_t;
```

## FPGA to SW

These are the replies the FPGA is capable of sending to SW.

### reset_fpga_rpl

cmd_type: 0x80000000
len: 8 (no additional data follows the header)

This tells SW that the FPGA has been reset successfully. After this a get_fpga_status message should be sent to the FPGA to confirm it is in a good state.

### fpga_status_rpl

Cmd_type: 0x80000001
len: 8 + 36

This reply tells SW the current status of the FPGA, the build information, what commands it is capable of running, and any error flags or extra debug information that might be useful.

```
typedef __packed__ struct {
  fpga_header_t hdr;
  uint32_t      fpga_version;     // e.g. 0x00_01_00_00 (v 1.0.0, format
major.minor.patch)
  uint64_t      fpga_build_date;  // String of build date FPGA image was built
  uint64_t      fpga_build_host;  // String of machine name FPGA image was built
  uint64_t      fpga_cmd_cap;     // Bitmask of what commands are capable to run on
this FPGA build
  uint64_t      fpga_state;       // What the FPGA state is in and any error flags
} fpga_status_rpl_t;
```

### fpga_ignore_rpl

Cmd_type: 0x80000002
len: 8 + 8

This reply tells SW that the FPGA received a message it was unable to decode (either did not have the capability or some error in the message, for example incorrect length), and is ignoring it.

```
typedef __packed__ struct {
  fpga_header_t hdr;
  fpga_header_t ignored_command;  // This is the command that the FPGA ignored
} fpga_ignore_rpl_t;
```

### verify_equihash_rpl

cmd_type: 0x80000100
len: 8 + 8 + 1

This command from FPGA gives the result of a **verify_equihash** command, along with the index and resulting bitmask for any errors found (will be all 0 if it verifies correctly).

```
typedef __packed__ struct {
  fpga_header_t hdr;
  uint64_t      index;
  uint8_t       result_mask;  //   [0] == DIFFICULTY_FAIL, [1] == XOR_NON_ZERO, [2] ==
BAD_IDX_ORDER, [3] == BAD_ZERO_ORDER;
} verify_equihash_rpl_t;
```

### verify_secp256k1_sig_rpl

cmd_type: 0x80000101
len: 8  + 8 + 1
This command replies to SW with the result of the verification check for a secp256k1 signing. We return the result of the verification along with the index. The result passed if none of the result_mask bits are set.

```
typedef __packed__ struct {
  fpga_header_t hdr;
  uint64_t      index;
  uint8_t       result_mask; // []0] == R_OUT_OF_RANGE, [1] == S_OUT_OF_RANGE, [2] ==
X_INFINITY, [3] == SIGNATURE_VERIFICATION_FAILED
} verify_secp256k1_sig_rpl_t;
```

### bls12_381_interrupt_rpl

cmd_type: 0x80000200

len: 8  + 4 + N
This command replies to SW when an interrupt instruction is hit by the bls12-381 coprocessor, along with the data that was pointed to by the instruction. The length in the header will for up to the data[N], to know how much data to process in this message you need to parse the date_type.

```
typedef __packed__ struct {
  fpga_header_t hdr;
  uint32_t      index;  // Custom value from user in the instruction
  uint8_t       data_type // What type of data (e.g. Affine point, scalar, JB point)
  uint8_t       data[N] // Depending on data in slot this will be from 48 to 576 bytes
long
} bls12_381_interrupt_rpl_t;
```

# FPGA command capability register

This is the bit mask returned from the fpga_status_rpl_t message. If a command is sent to the FPGA for something it has no capability to run, it will reply with a “fpga_ignore_rpl_t”.

| Bit | Capability | Note |
|---|---|---|
| 0 | verify_equihash with N= 200, K = 9 | Only one of these can be enabled per FPGA build |
| 1 | verify_equihash with N= 144, K = 5 | |
| 2 | verify_secp256k1_sig | Verify a secp256k1 signature |
| 3 | BLS12-381 coprocessor enabled | Used to accelerate zk-SNARK |
| | | |

# FPGA Architecture

## Overview of blocks in the system

These are the blocks in the system, build-time parameters can control which optional blocks are included in the FPGA build (e.g. you might disable those that aren’t used so the system fits on a smaller FPGA). Depending on if all blocks are enabled or not, the internet clock speed to FPGA might need to be lower to take into account that the FPGA will have a harder time to close timing constraints.

- Top level board
  - Control block **(required)**
  - Equihash verification engine (optional)
    - Verify pow
    - Find solution (mine)
    - Blake2b for generating XORs
    - SHA256 for difficulty check
    - Hash Map for checking duplicates
    - Order checker of indexes
  - Transparent Signature Verification Engine (secp256k1 ECDSA core) (optional)

        - 256b Scalar multiplier mod p / mod n
        - 256b Scalar inversion mod p / mod n
        - High speed 256b integer multiplier with mod reduction stage of either n or p
        - Point add
        - Point double
        - Point multiply
        - Resource arbitrator (to share 256b multiplier core)
    - BLS12-381 Coprocessor (zk-SNARK accelerator) (optional)
        - Resource arbitrator sharing
            - 381b integer multiplier mod p
            - 381b integer adder mod p
            - 381b integer subtractor mod p
        - Dual mode Fp / $Fp^2$ point operations on bls12-381
            - Point add / double
            - Point multiply
        - Instruction memory
        - Data memory
    - Interface module **(required)**
        - UART (For Bittware board)
        - PCIe (For Amazon AWS)

This section talks in more detail about the architecture of each main engine on the FPGA, along with performance results.

## Interface module

### AWS (Amazon)

The AWS top level has a wrapper cl_zcash_aws_wapper.sv which maps the data coming in over PCIe 512 bits wide to the 64 bits wide expected by the Zcash internal logic. It is also responsible for mapping to the streaming interface. The top level parameter “USE_AXI4” controls if AXI4 or AXI4-lite will be used for the streaming interface.

### VHH (Bittware)

This top level has a wrapper to generate the required clocks, and to provide an interface from the USB-UART into the Zcash internal logic.

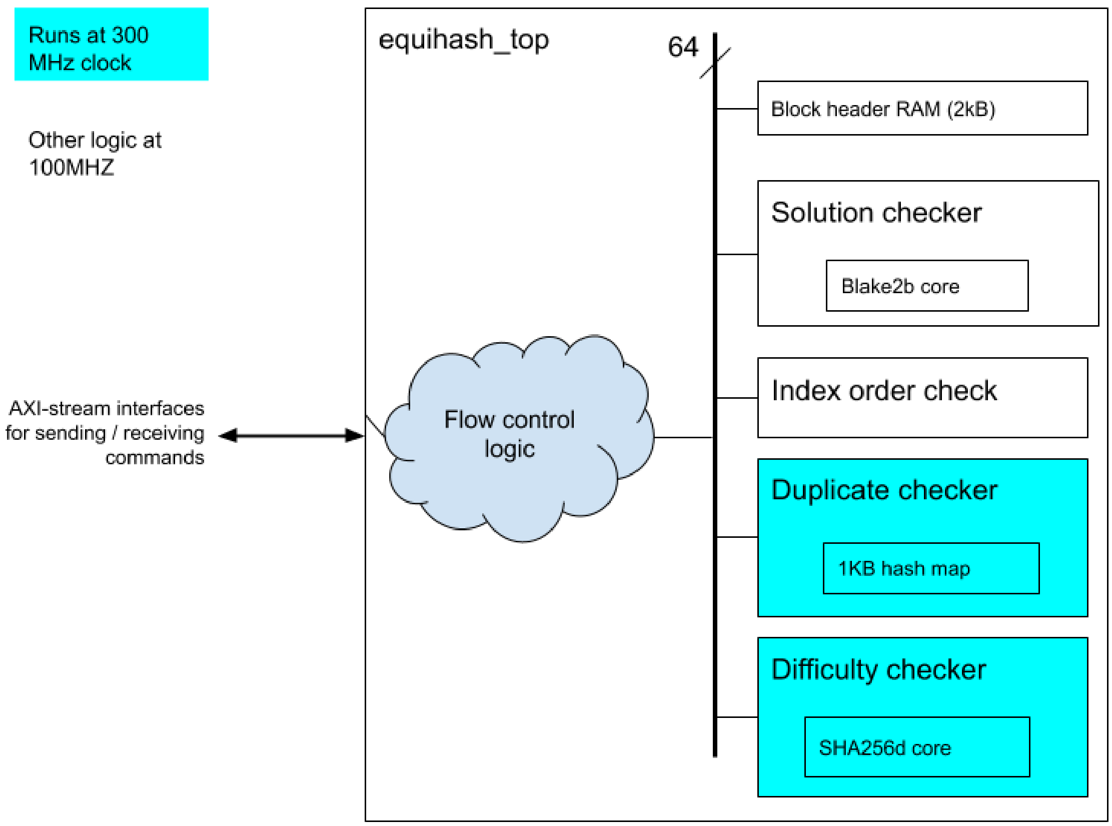
## Equihash Verification Engine

### Overview

The equihash engine takes in a block header and then stores the data in global memory, and each sub-block is given the data required for it to check, each which will set a single bit in the resulting block mask. The blake2b block is fully unrolled and running at 200MHz, meaning it takes 64 clock cycles to get a single result, but after that each clock cycle is a new result. This allows the hash of the 512 XOR strings in the equihash solution to be computed at very high throughput. This is more important for parameters (n=200,k=9) than the proposed (n=144, k=5) as there are less hases to be performed. The duplicate checker is a hash map and can run at a higher frequency of 300MHz. All the checks run in parallel so the slowest check will determine the performance, currently the duplicate check and difficulty check. This could be improved by moving both to a higher clock frequency.

The Blake2b core is able to generate a new hash output after an initial delay of 2 + ceil(input bytes/128)*24, so for the solution checker here (140B input, 512 hashes), we achieve 177M hash/s. Maximum performance would be at 5G hash/s.

## Block diagram




## Performance evaluation

### FPGA resources

Percentages reported for the VU37P

| LUT | FF | DSP | BRAM |
|---|---|---|---|
| 87914 (3%) | 54362 (3%) | 0 | 6 (0.2%) |

### Clock cycles

| | FPGA clock cycles | FPGA throughput | CPU cycles | 3GHz CPU throughput |
|---|---|---|---|---|
| Solution check | 600 @100MHz | | | |
| Index order check | 356 @100MHz | | | |
| Duplicate check | 1443 @300MHz | | | |
| Difficulty check | 1068 @300MHz | | | |

| | | | | |
|---|---|---|---|---|
| **Equihash solution verification** | 1068 @300MHz | 207K op/s | ~2868040 | ~1K op/s |

Here performance on FPGA is 207X faster, likely due to high performance Blake2b core, as well as all checks being done in parallel.

## Transparent Signature Verification Engine (secp256k1 ECDSA core)

### Overview

This engine handles all the operations for the curve secp256k1. This block at a top level supports point multiplication with a top level state controller, point multiplication, point addition, point doubling, point inversion, integer multiplication, and integer modulo reduction blocks. Blocks are shared via a resource arbitrator.
We optionally can use the endomorphism of secp256k1 to split the k in *X=kQ* into two smaller half-size *k1* and *k2*, by instantiating a “endomorphism decom block”, which gives close to a 2x improvement in throughput, at the cost of having 2 more multiplication engines.

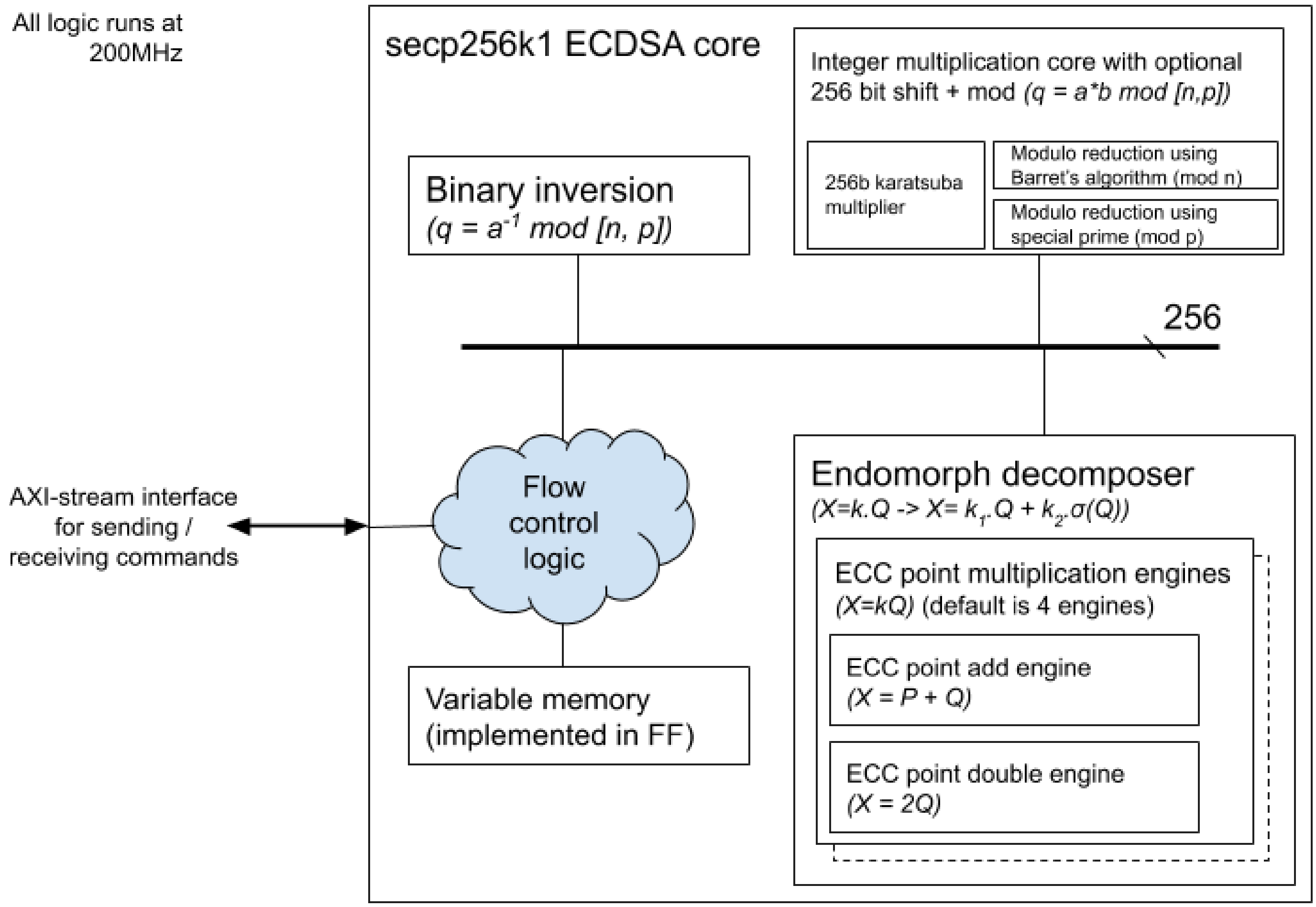
We create two ECC point multiplication modules which run in parallel to calculate $X = u_1P + u_2Q$ required for signature verification. These run in parallel, but due to the pipelined integer multiplication core we have both point multiplication modules share this.

The algorithm used for point multiplication is the double and add method, but we take advantage of the FPGA parallelism and do the double and add at the same time. Since doubling is faster than adding, we start the next double if we have an unfinished add in progress, improving performance. Each point double takes 54 clock cycles and each point addition takes 104 clock cycles.

The integer multiplication core is implemented with the karatsuba algorithm (2 levels) and each level is pipelined over 3 flip-flops (for timing @ 200MHz), so that a result is valid after 6 clock cycles, with a new result every clock cycle. On the output we can optionally bit shift (used for endomorph decomposition), reduce the result mod p (taking 2 clock cycles as it takes advantage of the prime form), or reduce the result mod n which takes longer as it uses barrett's algorithm. Mod n operations are only required at the start and very end so this path does not need to be optimized too much.

Binary inversion uses the gcd algorithm and takes roughly 708 clock cycles, so this is avoided as much as possible by: 1) Converting to jacobian coordinates for point multiplication and 2) the signature result can be checked without converting back to affine coordinates using the same method as in Zcash’s git source code.

### Block diagram




### Performance evaluation

Average performance of the core is shown below for signature verification (which will depend on the number of adds/doubles required). This is also compared to the same function from zcash's git running on a 3GHz processor (measured using average of CPU cycle counts). I did not try to optimize by using non-adjacent form . (NAF) window methods / Shamir's trick, as on the FPGA we run the calculations truly in parallel and might not benefit from these techniques, although this could be a point for future exploration. Improving the equations used for point double and point add would also improve performance. The FPGA was successfully meeting timing at a 200MHz clock. FPGA throughput could be improved by instantiating more cores.

#### FPGA resources

Percentages reported for the VU37P

| | **LUT** | **FF** | **DSP** | **BRAM** |
|---|---|---|---|---|
| secp256k1 ECDSA core (without endomorph enabled) | 57697 (4.4%) | 31751 (1.2%) | 144 (1.6%) | 2 (0.1%) |
| secp256k1 ECDSA core (with endomorph enabled) | 98792 (7.5%) | 61909 (2.1%) | 144 (1.6%) | 2 (0.1%) |

#### Clock cycles

| | **FPGA clock cycles** | **FPGA throughput** | **CPU cycles** | **3GHz CPU** |
|---|---|---|---|---|

| | | | | throughput |
|---|---|---|---|---|
| Point double mod p | 54 | 3.7M op/s | | |
| Point add mod p | 104 | 1.9M op/s | | |
| Inversion mod n | 708 | 282K op/s | | |
| secp256k1 ECDSA core (without endomorph enabled) | 20224 | 9.9K op/s/core | 223350 | 13.4K op/s |
| **secp256k1 ECDSA core (endomorph enabled)** | 10100 | 20K op/s/core | | |

FPGA performance is 1.5X compared to a 3GHz CPU. The FPGA could instantiate multiple ECC engines to run in parallel.

### Future Optimizations

Investigating the impact using NAF has on performance would be the next possible optimization.

## BLS12-381 Coprocessor (zk-SNARK accelerator)

### Overview

This coprocessor is used to accelerate zk-SNARKS as the majority of elliptic arithmetic used during proving and verifying is run on top of the bls12-381 curve.
Unlike previous cores, the coprocessor can be configured by writing to instruction memory rather than accepted hard coded commands. This is to allow more flexibility in how the co-processor is used. SW can either poll registers on the FPGA coprocessor or use interrupt instructions so that the FPGA will send data to SW.

The coprocessor has instruction memory that can be written to, after a reset command the entire memory is initialized to NOOP-WAIT. The coprocessor has a memory bank with addressable data slots each 64 bytes wide per address for variables that can be used with instructions, example sizes for variables are:

- Scalar integer takes 1 slot
- Point in Fp takes 3 in jacobian coordinates (2 in affine)
- Point in $Fp^2$ take 6 in jacobian coordinates (4 in affine)
- $Fp^{12}$ element takes 12 slots

Each data slot only uses 48 bytes on the FPGA (64 bytes of address space is used in SW to simplify the mapping of memory to slot index). The first 381 bits of a slot store that elements data, the remaining 3 bits are used as a format for the type of element stored (more bits can be added if needed).

| | |
|---|---|
| 0 | Scalar |
| 1 | Fp element |
| 2 | $Fp^2$ element |
| 3 | $Fp^{12}$ element |

| | |
|---|---|
| 4 | Fp point AF |
| 5 | Fp point JB |
| 6 | $Fp^2$ point AF |
| 7 | $Fp^2$ point JB |

### Instructions

Instructions are 8 bytes each (1 byte for op-code, and then the rest is used to address variables).

Interrupts are sent by using the SEND-INTERRUPT instruction which can be used to send the result of a calculation to SW. SW will have a method of registering a callback function that would be called when an interrupt is detected, the function will take a pointer to memory that will hold the data sent from FPGA.

Montgomery form is not used in any of the operations (as we can use RAM lookup table technique for the modular reduction).

All point operations can be given inputs in affine or jacobian coordinates, but outputs will be in JB unless otherwise stated. There is not a specific instruction for converting to affine coordinates because you can get the same result by multiplying the point element (Fp or $Fp^2$) by INV-ELEMENT(MUL-ELEMENT(Z, Z)).

| **Instruction** | **Description** |
|---|---|
| NOOP_WAIT (0x0) | Coprocessor waits at this command and does nothing (used to stall or after a reset) |
| COPY_REG(0x1, a, b) | Copy contents of register b = a |
| JUMP(0x2, a) | Jump instruction pointer to location a |
| JUMP_IF_EQ(0x4, a, b, c) | Jump instruction pointer to location a if b == c, else go to next instruction (b and c are limited to the lower 64 bits) |
| JUMP_NONZERO_SUB(0x5, a, b) | If b != 0 then jump to a and b--, otherwise go to next instruction (b is limited to the lower 64 bits) |
| SEND_INTERRUPT(0x6, a, b) | Send an interrupt to SW along with the data in slot a. Amount of bytes sent will depend on data type stored in slot. 16 bit value of b will be appended to the interrupt message header (see streaming commands for bls12_381_interrupt_rpl_t) |
| | |
| MUL_ELEMENT (0x10, a, b, c) | Do Fp / $Fp^2$ / $Fp^{12}$ field element multiplication, c = a x b |
| ADD_ELEMENT (0x11, a, b, c) | Do Fp / $Fp^2$ field element addition, c = a + b |
| SUB_ELEMENT (0x12, a, b, c) | Do Fp / $Fp^2$ field element subtraction, c = a - b |
| INV_ELEMENT(0x13, a, b) | Calculate the inverse of a Fp / $Fp^2$ field element a and store in b |

| | |
|---|---|
| POINT_MULT(0x20, a, b, c) | Do a Fp / $Fp^2$ point multiplication using scalar a and Fp / $Fp^2$ affine point b, and store result jacobian point in c. c = a x b |
| [1]MILLER_LOOP(0x21, a, b, c) | Do a miller loop of the G1 Fp affine point in a and G2 $Fp^2$ affine point in b, and store result $Fp^{12}$ field element in c |
| [1]FINAL_EXP(0x22, a, b) | Do a final exponentiation of the $Fp^{12}$ field element in a and store result $Fp^{12}$ field element in b |
| ATE_PAIRING(0x23, a, b, c) | Do an optimal ate pairing of the G1 Fp affine point in a and G2 $Fp^2$ affine point in b, and store result $Fp^{12}$ field element in c |

**Notes**:
[1]The purpose of these commands is to allow for faster multi-pairing operations - you can call the MILLER_LOOP instruction on multiple points, and then only a single FINAL_EXP instruction.

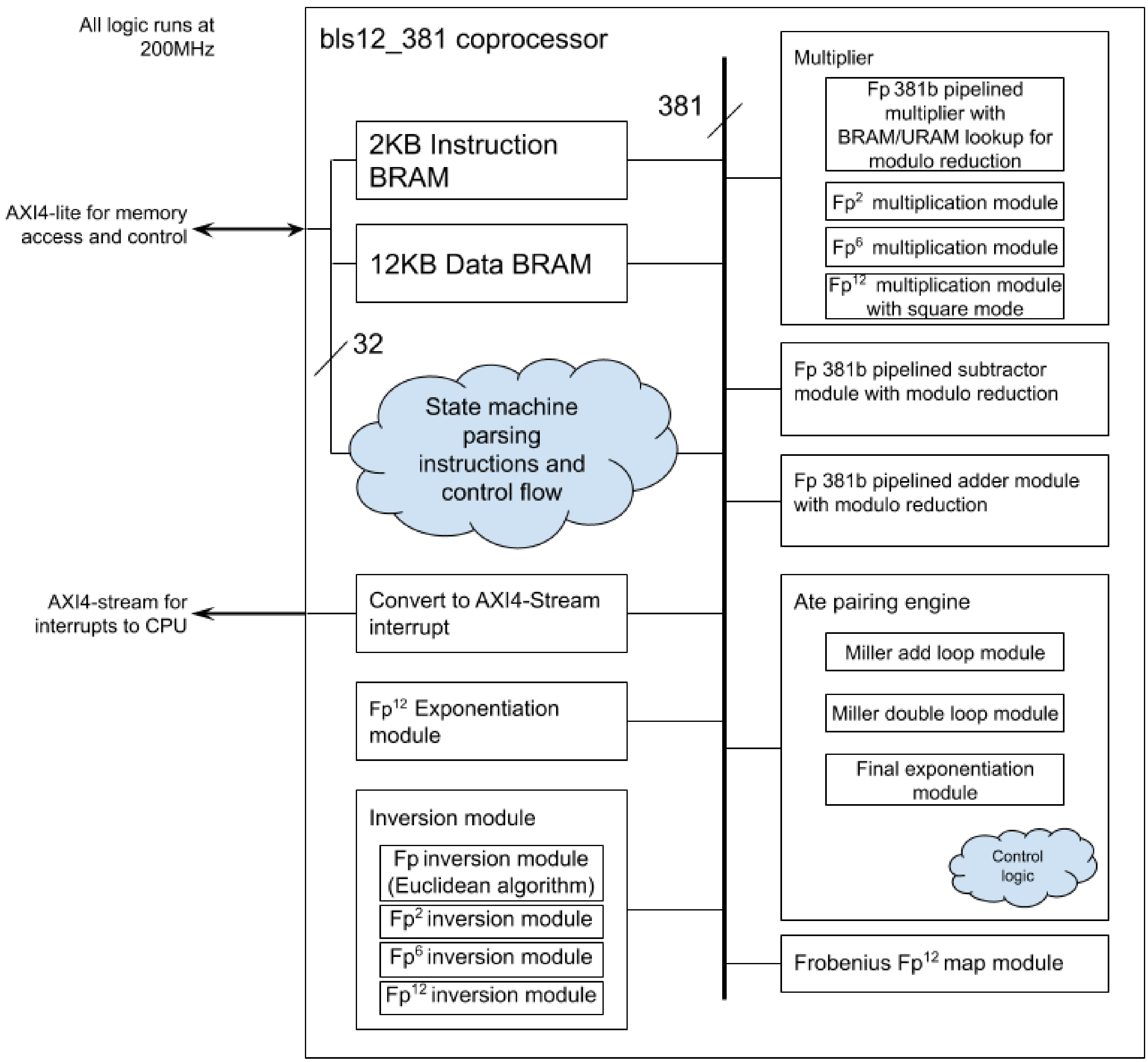
## Memory Map

This is the AXI-lite portion of the core that can be used for configuration, as well as writing/reading instruction/data memory.

| **Register Name** | **Address** | **Access** |
|---|---|---|
| Instruction memory offset / reset control | 0x0 | Read: returns the memory offset where instruction memory begins<br>Write: A ‘1’ to bit[0] will reset the instruction memory, a ‘1’ to bit[1] will reset the data memory |
| Data memory offset | 0x4 | Read only: returns the memory offset where data memory begins |
| Data memory size | 0x8 | Read only: returns the power of 2 number of data memory slots (i.e. 8 => 256 slots) |
| Instruction memory size | 0xc | Read only: returns the power of 2 number of instruction memory slots |
| Current instruction pointer | 0x10 | Read: returns current instruction memory pointer<br>Write: sets the instruction memory pointer (will wait until current operation finishes) |
| Last instruction cycle count | 0x14 | Read only: returns the number of clock cycles the last instruction took to complete |
| Data for v1.4 multiplier (reduction RAM) | 0x18 | A write here will load 32 bits onto the reduction RAM data line required for 1.4v multiplier |
| Control for v1.4 multiplier (reduction RAM) | 0x1c | A write to bit[1] will enable a shift of data written to location 0x18. |

| | | |
|---|---|---|
| | | A write to bit[0] will enable a write to the reduction RAM address (this should be done after shifting 381bits of data via 0x18 and writes to bit[1]). |

## Architecture

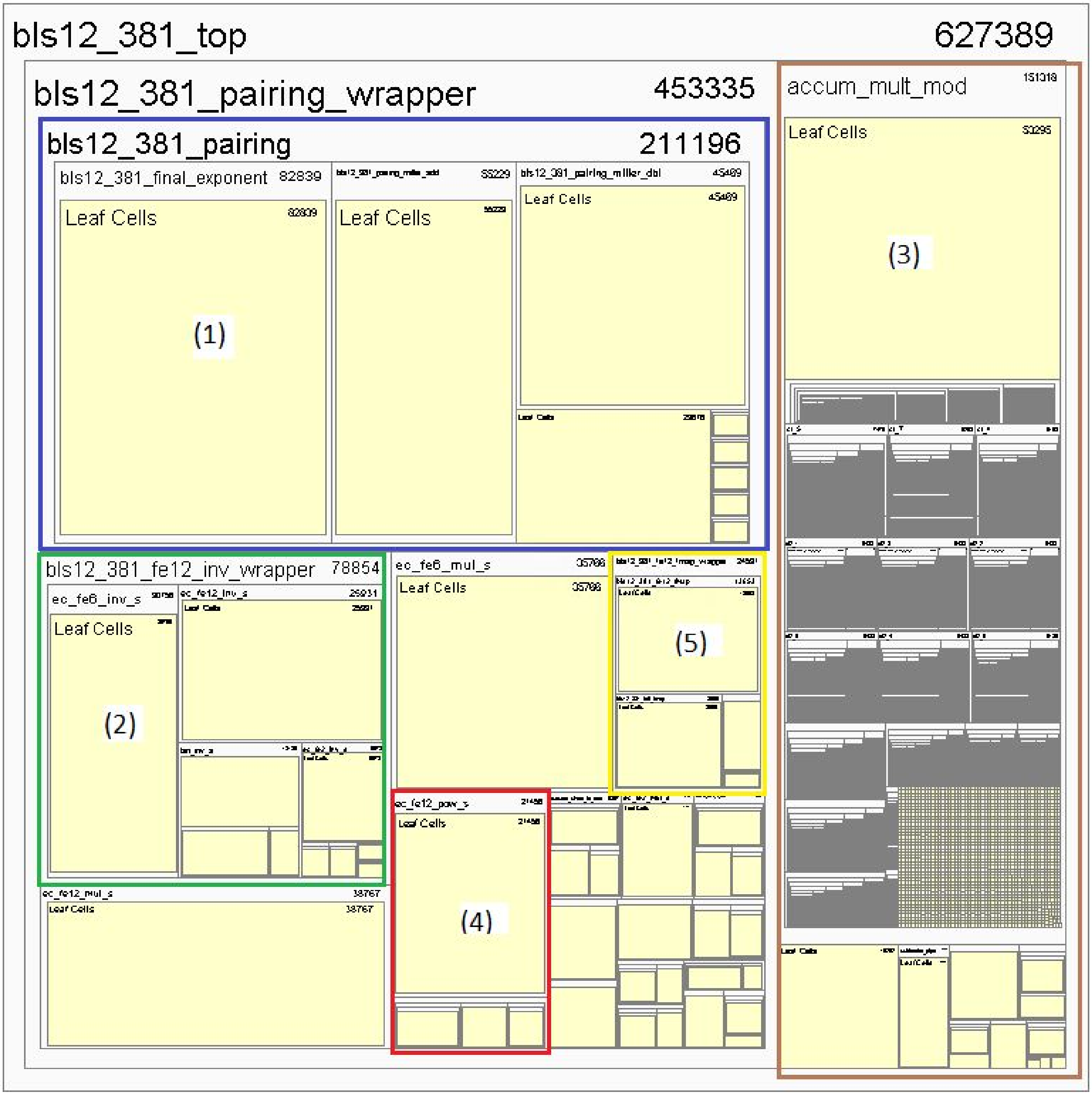



The coprocessor operates on a shared 381 bit bus with a main state machine with pointers into a data and instruction memory (implemented using Xilinx Ultra RAM on the FPGA). The top level multiplier, adder, and subtractor are all fully pipelined (so a new result each clock) and are resource shared with the entire coprocessor (so inversion block, dual mode point multiplier, pairing engine,... all use this).
The multiplier used is sized so to take advantage of the FPGA DSPs - 27x17 bit wide multipliers. We perform all multiplications in parallel, followed by a tree of carry-save adders. After this we use a RAM lookup technique

for the modular reduction. We then propagate carries, and perform another stage of modular reduction, and final stage of checking if we need to subtract the modulus again.
Point multiplication is supported by either placing a dedicated point multiplication module (same as secp256k1), or by reusing the point double and add modules in the miller loop block which is used for the pairing calculation. Reusing the miller loop logic saves FPGA resources, but results in a slower point multiplication.

Below is a diagram showing the hierarchy of modules and their respective resource usage on the FPGA taken from Vivado. The different main blocks are:

1. The pairing engine, which consists of the Miller loop and final exponentiation.
2. The $Fp^{12}$ inversion module
3. The multiplier - in v1.4 we changed this multiplier to a version that uses RAM lookup, so most of the space is occupied by the carry save adder trees.
4. The $Fp^{12}$ exponentiation module
5. The frobenius map $Fp^{12}$ module




## Performance Comparison

### FPGA resources

Percentages reported for the VU37P.

| LUT | FF | DSP | RAM |
|---|---|---|---|
| 327k (25.1%) | 226.6k (8.7%) | 345 (3.8%) | 133 URAM (13.8%), 231 BRAM (11.4%), 14164 LUTRAM (2.3%) |

### Clock cycles

Here performance was benchmarked vs the Rust bls12_381 crate on a 32GB, 3.7GHz i5-9600K CPU. FPGA is running at 200MHz. Although we have lower throughput for individual operations in Fp, we are able to take advantage of parallel operations inside the higher order $Fp^{12}$ operations, and see a 2.9x speedup in the final ate pairing. A large amount of time is spent on the final exponentiation, which could be a target for optimization - or moving to the weil pairing which does not require the final exponentiation (instead requires two miller loop iterations).

| | **FPGA clock cycles** | **FPGA throughput (op/s)** | **3.7GHz CPU throughput (op/s)** |
|---|---|---|---|
| Fp inversion | 2685 | 74.5K | 109K |
| $Fp^{12}$ inversion | 3565 | 56K | 60.5K |
| Fp multiplication + modulo reduction | 9 | 22M | 20.8M |
| $Fp^{12}$ multiplication + modulo reduction | 270 | 740K | 228K |
| Fp point multiplication | 49800 (dedicated $Fp^2$ point mult block) | 4016 | 4926 |
| $Fp^2$ point multiplication | 62064 (dedicated $Fp^2$ point mult block) | 3222 | 1499 |
| Optimal  Ate pairing miller loop stage | 38844 | 5148 | 1747 |
| Optimal  Ate pairing final exponentiation stage | 87800 | 2277 | 854 |
| Optimal Ate pairing total | 126644 | 1580 | 553 |

### Future Optimizations

- Investigating the impact of NAF on point multiplication
- Pre-computation for the G2 double / add values used in the miller loop (useful if we are doing multiple pairings)
- Implement sparse multiplication for $Fp^{12}$ for the miller loop
- Modify architecture to use redundant polynomial form, then addition / subtraction / multiplication could be done faster as we don't need to propagate carries.

# User Guide

This section goes over example usage of the system.

## Running Simulations

### Module level simulations

Most modules have a corresponding "_tb.sv" in the tb/ folder, and are self checking so can be added to local copy of Vivado and ran, and will print a message that all tests passed for that module if there are no problems. A top level simulation that tests all functions and emulates the Bittware VVH top level is here https://github.com/bsdevlin/zcash-fpga/blob/master/zcash_fpga/src/tb/zcash_fpga_top_tb.sv
The easiest way is to start a new Vivado project, and add all .sv and .xci (ip) files to the project, then run the _tb file you want. A good place to start is
https://github.com/bsdevlin/zcash-fpga/blob/master/zcash_fpga/src/tb/bls12_381_top_tb.sv or https://github.com/bsdevlin/zcash-fpga/blob/master/zcash_fpga/src/tb/zcash_fpga_top_tb.sv.
When simulating the top modules, they include the multiplier with RAM lookup for modular reduction, so you need to run this script otherwise simulation / building will produce an error:
https://github.com/bsdevlin/zcash-fpga/blob/master/ip_cores/accum_mult_mod/scripts/generate_files.py
(This is only for the bls12-381 coprocessor, if you want to disable this you can set the top level param )
**Note**: This needs to be Python 3

It is also recommended defining FASTSIM, otherwise the adder module will consume a lot of time to simulate (https://github.com/bsdevlin/zcash-fpga/blob/master/ip_cores/accum_mult_mod/src/rtl/compressor_tree_3_to_2.sv).

### AWS Board level

The simulation test case for the AWS board are in the repo folder https://github.com/bsdevlin/zcash-fpga/tree/master/aws/cl_zcash/verif/tests and can be run by:

1. cd */home/centos/aws-fpga/hdk/cl/developer_designs/cl_zcash/verif/scripts*
2. make TEST=test_zcash FASTSIM=YES

This will compile and run the test cases, they are all self checking, so if not ERRORs are printed and the simulation finishes then there are no problems. If something unexpected happens you can run xsim and look at the waveforms. At the moment there is just one test (test_zcash.sv) that will test all of the top level block functions.
**Note**: Currently there is an issue with the reduction RAM based multiplier when simulating on AWS, so for simulations to pass chang this parameter to "YES":
https://github.com/bsdevlin/zcash-fpga/blob/master/zcash_fpga/src/rtl/top/zcash_fpga_pkg.sv#L43
(Simulating the RAM based multiplier can be done in Vivado project mode)
The bls12-381 part of the simulation can take a while (10min) as it does a multi pairing.

## Usage with a local FPGA board

If the board is local, it can be configured over USB-UART (note this is very low bandwidth and just mainly used for proof of concept / testing).

Commands can be called from the python script: bittware_xupvvh/software/zcash_fpga.py

# Usage on AWS

AWS runs over PCIe and has a higher bandwidth, but due to timing a slower clock is used (as there is more glue logic on the FPGA).
At the time of writing this these were the versions used in the AWS toolchain:

| Developer Kit Version (HDK) | Tool Version Supported (Vivado) | Compatible FPGA developer AMI Version |
|---|---|---|
| 1.4.8-1.4.X | 2018.3 | v1.6.0 (Xilinx SDx 2018.3) |

## Building the FPGA image

If you make changes to the code or want to build a new image, you can follow the steps below. If you do not want to do this, you can skip to the next section "Loading FPGA image" and use one of the pre-built images listed in "Existing AFIs". When building the image it is recommended to change the parameters here to only enable the blocks required - https://github.com/bsdevlin/zcash-fpga/blob/3a8c799a742061760d9c1deaaaebd72a60792ca9/zcash_fpga/src/rtl/top/zcash_fpga_pkg.sv#L32 , as enabling everything will make the build take longer and might not meet timing.

1. Start an AWS instance and load it with the FPGA Developer AMI (https://aws.amazon.com/marketplace/pp/B06VVYBLZZ)
    a. This should be a f1 instance (e.g. f1.2xlarge) so you have access to an FPGA
    b. If you just want to build the FPGA image you can use a cheaper instance like r5.xlarge (just need at least 32GB RAM)
2. Clone the zcash git repo
3. Clone the aws-fpga repo
    a. *git clone https://github.com/aws/aws-fpga.git*
4. Copy the folder *zcash-fpga/aws/cl_zcash* to the AWS folder
    a. *cp -r /home/centos/zcash-fpga/aws/cl_zcash /home/centos/aws-fpga/hdk/cl/developer_designs/*
5. Copy the folder */home/centos/aws-fpga/hdk/cl/examples/common*
    a. *cp -r /home/centos/aws-fpga/hdk/cl/examples/common /home/centos/aws-fpga/hdk/cl/developer_designs/*
6. Run the hdk_source.sh script to setup the AWS environment
    a. *cd /home/centos/aws-fpga; source hdk_setup.sh*
    b. Note: If you get an error with Vivado not being present, it might be due to locale issue, try:
        i. *export LC_ALL="en_US.UTF-8"*
7. Set the variables for Zcash scripts:
    a. *export CL_DIR=/home/centos/aws-fpga/hdk/cl/developer_designs/cl_zcash; export ZCASH_DIR=/home/centos/zcash-fpga/*
8. Generate the FPGA IP files

a. cd */home/centos/aws-fpga/hdk/cl/developer_designs/cl_zcash/ip/* ; *./run_cl_sde_ip_flow*

9. Start building the FPGA image
    a. cd */home/centos/aws-fpga/hdk/cl/developer_designs/cl_zcash/build/scripts* ; *./aws_build_dcp_from_cl.sh -clock_recipe_a A0 -clock_recipe_b B1*
    b. Note: AWS clock recipes are here: https://github.com/aws/aws-fpga/blob/master/hdk/docs/clock_recipes.csv , a higher performance version of the core can use "*-clock_recipe_a A1 -clock_recipe_b B0*", a slower version (but easier to build and meet timing) could use "*-clock_recipe_a A2 -clock_recipe_b B1 -strategy BASIC*"
        i. Note: this will not work with an ILA debug core since the clock speed (15MHz) is too slow compared to JTAG frequency
    c. You can check progress by looking at "/home/centos/aws-fpga/hdk/cl/developer_designs/cl_zcash/build/scripts/last_log"

The build should run and will take several hours, depending on the instance type / clock recipe. If there are no problems, the output will be in /home/centos/aws-fpga/hdk/cl/developer_designs/*cl_zcash*/build/checkpoints/to_aws/*.tar and needs to be uploaded to an Amazon S3 bucket. The bucket used in this project is "zcash-fpga-west". From here you can follow the standard flow detailed on the AWS FPGA github: https://github.com/aws/aws-fpga/blob/master/hdk/README.md#step3 .

After this you should have an agfi-ID that can be used to program the FPGA.

## Loading FPGA image

To load an FPGA image you need it's agfi-ID, either from the previous step or from the table in the following section "Existing AFIs".

Run this commands to load the FPGA:

1. sudo fpga-load-local-image -S 0 -I -F *agfi-ID*

**Note**: You can check for errors / metrics by running the command "sudo fpga-describe-local-image -S 0 --metrics". If you see all 0's then there is no problem, but if you see some timeouts like this:

ocl-slave-timeout-addr=0x2001
ocl-slave-timeout-count=4

You should reload the FPGA image (step 1 above). There is a known issue with AWS where the first load will sometimes show this problem, but reloading FPGA fixes it.

## Rust interface

A rust interface has been developed to allow the Zcash client to utilize the FPGA acceleration.

## Startup test program

A simple program is in /home/centos/aws-fpga/hdk/cl/developer_designs/*cl_zcash*/software/runtime/ (test_zcash)

1. Run the sdk_source.sh script to setup the software AWS environment
    a. *cd /home/centos/aws-fpga; source sdk_setup.sh*

Run the make file and then the test program (using sudo), and check there are no errors. The test program connects to the FPGA and programs the bls12-381 coprocessor to do a multi-pairing and then send the result back to SW, and then tests the secp256k1 ECDSA verification (will first check these are enabled in the build). Expected output would be this (taken from v1.3.0 build):

```
[centos@ip-172-31-15-165 runtime]$ sudo ./test_zcash
INFO: AFI PCI  Vendor ID: 0x1d0f, Device ID 0xf000
INFO: Read 0x280000 from ISR register.
WARNING: Expected 0x01D00000.
INFO: Read 0x1fc from TDFV register.
INFO: Read 0x0 from RDFO register.
INFO: AXI4 mode is set DISABLED
INFO: write_stream::Wrote 8 bytes of data
INFO: Read FIFO shows 37 bytes waiting to be read from FPGA
INFO: Read 40 bytes from read_stream()
INFO: FPGA version: 0x10300, built on 0x6c6444617465227d
INFO: FPGA capability register: 0xc [ENB_VERIFY_EQUIHASH_200_9: 0,
ENB_VERIFY_EQUIHASH_144_5 0, ENB_VERIFY_SECP256K1_SIG 1, ENB_BLS12_381 1]
INFO: Finished initializing FPGA.
INFO: Testing secp256k1 core...
INFO: write_stream::Wrote 176 bytes of data
INFO: Read FIFO shows 19 bytes waiting to be read from FPGA
INFO: Read 20 bytes from read_stream()
INFO: verify_secp256k1_sig_rpl.hdr.cmd = 0x80000101
INFO: verify_secp256k1_sig_rpl.bm = 0x0
INFO: verify_secp256k1_sig_rpl.index = 0xa
INFO: verify_secp256k1_sig_rpl.cycle_cnt = 0x2de5
INFO: Testing bls12_381 coprocessor...
INFO: Resetting instruction memory
INFO: Resetting data memory reset
INFO: Set BLS12_381 current instruction slot to 0 (was 7)
INFO: Set BLS12_381 current instruction slot to 1 (was 0)
INFO: Read FIFO shows 64 bytes waiting to be read from FPGA
INFO: Read 64 bytes from read_stream()
INFO: Read FIFO shows 592 bytes waiting to be read from FPGA
INFO: Read 592 bytes from read_stream()
INFO: BLS12_381 current instruction slot is 7
INFO: Data slot is now 7
slot 0, pt: 0,
data:0x00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000a
slot 1, pt: 3,
data:0x04fb0f149dd925d2c590a960936763e519c2b62e14c7759f96672cd852194325904197b0b19c6b52
8ab33566946af39b
```

```
slot 2, pt: 3,
data:0x185ef728cf41a1b7b700b7e445f0b372bc29e370bc227d443c70ae9dbcf73fee8acedbd317a286a5
3266562d817269c0
slot 3, pt: 3,
data:0x03a3734dbeb064bf4bc4a03f945a4921e49d04ab8d45fd753a28b8fa082616b4b17bbcb685e455ff
3bf8f60c3bd32a0c
slot 4, pt: 3,
data:0x1409cebef9ef393aa00f2ac64673675521e8fc8fddaf90976e607e62a740ac59c3dddf95a6de4fba
15beb30c43d4e3f8
slot 5, pt: 3,
data:0x1692a61ce5f4d7a093b2c46aa4bca6c4a66cf873d405ebc9c35d8aa639763720177b23beffaf522d
5e41d3c5310ea333
slot 6, pt: 3,
data:0x081abd33a78d31eb8d4c1bb3baab0529bb7baf1103d848b4cead1a8e0aa7a7b260fbe79c67dbe41c
a4d65ba8a54a72b6
slot 7, pt: 3,
data:0x0900410bb2751d0a6af0fe175dcf9d864ecaac463c6218745b543f9e06289922434ee446030923a3
e4c4473b4e3b1914
slot 8, pt: 3,
data:0x113286dee21c9c63a458898beb35914dc8daaac453441e7114b21af7b5f47d559879d477cf2a9cbd
5b40c86becd07128
slot 9, pt: 3,
data:0x06d8046c6b3424c4cd2d72ce98d279f2290a28a87e8664cb0040580d0c485f34df45267f8c215dcb
cd862787ab555c7e
slot 10, pt: 3,
data:0x0f6b8b52b2b5d0661cbf232820a257b8c5594309c01c2a45e64c6a7142301e4fb36e6e16b5a85bd2
e437599d103c3ace
slot 11, pt: 3,
data:0x017f1c95cf79b22b459599ea57e613e00cb75e35de1f837814a93b443c54241015ac9761f8fb20a4
4512ff5cfc04ac7f
slot 12, pt: 3,
data:0x079ab7b345eb23c944c957a36a6b74c37537163d4cbf73bad9751de1dd9c68ef72cb21447e259880
f72a871c3eda1b0c
INFO: All tests passed!
```

## FPGA debug

Debug instructions can be found here:
https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md
There is a parameter in cl_zcash.sv, USE_ILA = “NO” which can be changed to “YES” to enable a build with the debug logic. You can change the connections as needed.

## Latest AFIs

These are the latest bug-free (no known bugs) public AFIs that exist and can be used on an AWS F1 instance.

| agfi-ID | afi-ID | Notes |
|---|---|---|
| agfi-0f7d033fb78c698e0 | afi-024814af3da043e90 | v1.4.2, using new multiplier (all BRAM so powers up with correct values), contains bls12-381 coprocessor, secp256k1 engine, equihash engine |
| agfi-0d0aeee105030594a | afi-081b60a4044e3db15 | v1.3, Contains secp256k1 sig core and bls12-381 coprocessor |

## Old (unused) AFIs

These are listed here for tracking purposes but not intended to be used as they are mostly debug / have bugs.

| agfi-ID | afi-ID | Notes |
|---|---|---|
| agfi-0528daff45454ed7c | afi-09056704c94b5280b | v1.0.0 First test version used for testing AWS flow, will not work with test program. |
| agfi-05561b352d56b5f57 | afi-0c8109482d730073c | v1.0.1 Test version |
| agfi-0fa84678db6b2752f | afi-07ec21206df23e398 | v1.1.0, Has all modules enabled but on a slow clock recipe for testing. BLS12_381 core has Fp and Fp2 fpoint instructions |
| agfi-019c2736fd0141219 | afi-0b891a8fc9644f1a0 | v1.1.0_150, only has BLS coprocessor enabled but running at 125MHz, uses AXI4 as PCIe interface |
| agfi-05468e41c302eb331 | afi-06a4b56d6e4bfd896 | v1.1.1, contains all cores @ 125MHz, uses AXI-lite as PCIe interface |
| agfi-0fce4c1ad9e0c6c43 | afi-0da67f631a2573656 | v1.1.2 contains all cores @ 125MHz |
| agfi-0c4a39d7638bc6010 | afi-0bcef9f0c08bee7c1 | v1.1.2 contains all cores @ 15MHz |
| agfi-0abc260b651d87d41 | afi-0075820f5d00bd799 | v1.1.3 Bug fixes to BLS12_381 core, 125MHz |
| agfi-07ae22f20d6e90559 | afi-0e49dd7ef17fda51a | v1.1.4, bug fix for multiple back to back interrupts, 125MHz |
| agfi-0db37e1358c1d885f | afi-0907df570f7dc7b2b | Debug version of v1.1.4 above (15MHz) |
| agfi-06d033b207d8f65c5 | afi-07177e176d04aa84b | v1.1.5, debug version 125MHz, contains bug fix for inverter, fp^12 logic |

# Conclusions

We were able to realize the main goals of this project:

- Accelerate blockchain verification
  - We developed an equihash verification engine that is able to take an input block header, verify the equihash solution is correct, matches the required difficulty, ordering requirement, and index uniqueness. With the current equihash parameters of N=200, K=9 we were able to achieve 207x speedup compared to the current Zcash SW client. The parallelism of the FPGA was able to be exploited fully here, as we can calculate many Blake2b hashes in parallel required for the solution verification, as well as doing the other checks at the same time. The engine takes parameters for the equihash values of N and K so can be adapted to other settings (e.g. N=144, K=5).
  - For verifying transparent transactions on the Zcash blockchain we developed a secp256k1 ECDSA core. This core is able to verify the signature used on a transparent transaction, which is the same as a transaction in Bitcoin. We were able to achieve 1.5x speedup when compared to the current Zcash SW client. The main reason the speedup is less when compared to the equihash verification is the lack of parallelism that we can exploit for a single signature verification.
- Accelerate zk-SNARKs
  - We developed a bls12-381 coprocessor which is able to perform curve operations that are required for zk-SNARKs. The coprocessor was designed with a simple instruction set so that it can be programmed from SW and is flexible in the flow of operations it can perform. All EC operations have also been implemented as SystemVerilog software models in the bls12_381_pkg.sv file, allowing for testbench verification and ease of implementation. We achieved 3x speedup compared to a 3.7GHz processor.
- Develop an open source FPGA code base for benefiting the wider community
  - All the code developed has been made open source and released under the GNU General Public License v3.0. It has all been developed in SystemVerilog (the most modern hardware description language), and uses parameters where possible so that the code can easily be reused and can be of benefit to the greater community (other zk-SNARK projects, other crypto coins, research projects…). Even after this project finishes it is expected that the code will continue to be worked on and improved in the open source community.

We decided not to implement FFT acceleration on the FPGA as after talking with Zcash engineers it was decided the benefit to zk-SNARKs would be minimal when compared to accelerating EC pairing operations. Although this could be a future project as FPGAs have historically been used to calculate FFTs and there exists a large amount of reference IP and code.

When developing code for operations on the bls12-381 curve it was evident the main consumer of FPGA clock cycles was the Fp multiplication operation followed by the modular operation over a prime where we could not use any special tricks to modular reduce in HW. Time taken to optimize this area of the code, or different prime selection would greatly speedup the overall performance of operations.

We accelerated the optimal ate pairing, but we found the time to calculate the final exponentiation was more than 2x the time for a miller loop on the FPGA (could be due to not enough optimization in Fp / $Fp^{12}$ multiplication), but this points that possibly a different pairing could be fast - such as the weil pairing where we have no final exponentiation stage, but require multiple miller loops.

The bls12-381 coprocessor uses a Fp -> $Fp^2$ -> $Fp^6$ -> $Fp^{12}$ towering, as the Zcash bls12-381 Rust create code was used as a reference when developing the FPGA code, but it would be worth investigating if different towerings would be faster on FPGA - it’s proposed Fp -> $Fp^2$ -> **$Fp^4$** -> $Fp^{12}$ towering might be faster.

# Appendix

## Example decoding Zcash block #346

Hash 0x000000eff179fb1e47b7aa8667ad4d8e1ef3dbb0d79144030482bf93b5e6339f

Hex dump of block (CBlock):

```
0    04 00 00 00 13 d6 d1 a4 10 51 42 19 f7 2f f3 a0 df d5 c3 8b 62 1c c2 c6 68 78 4d 2f d6 fd 10 8f
20   48 00 00 00 30 16 31 55 23 12 34 9d d5 3b 6b 9e 23 1d f8 bc b8 c2 d3 32 64 cc 02 f5 cd d9 a9 69
40   fb 93 80 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60   00 00 00 00 3f 85 13 58 bf c3 03 1e 1b b2 b5 50 a4 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80   00 00 00 00 00 00 00 00 00 00 00 03 fd 40 05 00 9d fa 04 89 e1 18 99 dc 5e 50 5d 91 24 57 44 49
a0   28 12 b3 f6 0e 31 04 e2 1e 98 b2 7d 80 38 c4 41 82 ca de 1a e2 ec dc e2 77 10 4f 9f a6 5d d4 b6
c0   a1 b6 ab 44 66 24 ef 6a 0a c2 a8 5e 2e a3 32 19 7c d3 cd 51 b6 e8 a3 31 d4 04 d4 68 bc ed 6b e6
e0   19 e4 8f 0b 8f c4 3b f9 dd 44 b2 f1 05 b4 7c b8 e7 e5 eb 21 96 bd 12 89 0e df 36 07 41 1e 55 81
100  b9 14 c2 91 b2 a7 1f 27 19 79 7c bc 49 13 42 34 62 bd 11 fb d7 b8 00 31 85 01 31 4f 2b 4e 15 1a
120  87 f6 40 16 9f d2 19 ef 51 bd 9c 19 94 38 7c 69 88 bf 68 77 7d 69 e3 06 2f dc 61 0e 4a 43 99 04
140  b7 d1 f6 26 78 fb e8 a9 2f f1 2a 38 0c b0 5b 29 33 4b 37 7c c5 30 11 e5 db a4 80 01 30 56 b5 a5
160  71 1b 10 e6 35 1e 5d 72 f5 4b 93 63 3b 0e 5d 4c 0b 12 ff 9b d6 20 31 84 5f 47 fb 90 23 af db 3c
180  15 bd 4a 51 ab b9 9a d8 0d 4c ef 21 b5 c9 da e9 a3 a5 61 a8 97 74 c2 ff 4e 3d 89 92 20 94 37 b7
1a0  63 f8 9d 22 61 6b 01 15 62 12 f7 40 47 ba a3 43 6a a0 5e bf 3a 25 d5 b9 df f7 d9 d0 b7 e0 ba 43
1c0  83 9c 1d 00 b2 3c 01 d1 e9 a8 42 95 06 8f 65 20 fb 53 59 d2 f7 c9 b2 60 44 ab 0f 0f de de 8a 02
1e0  45 62 8d 43 4a 64 bd 96 8d 93 8f 22 6f a9 75 32 ec b7 a0 af 27 06 0a aa 7f 97 3a 2d b2 95 83 20
200  35 de d2 92 4b 08 bc 6a 4a 06 f8 b1 d4 db b8 55 c1 f0 37 01 db ba a7 55 52 93 c4 3a 86 9c 23 3e
220  3f 2c c7 50 14 bd 2c ef 23 aa ad e5 1b 3e d9 08 fc 7b 1a 03 c7 a2 d8 71 8d 16 37 97 28 52 af 95
240  64 23 21 c5 57 7d 14 80 14 fd 68 e0 a0 96 87 03 c7 7a d1 8b 7a ad 29 99 a6 78 d6 0f 63 04 8f 33
260  30 ff d3 1c bb 75 3c c6 66 c8 35 1f 35 cf ac 76 46 93 0b b7 1c 17 8f 86 05 ff 7e 6f a1 94 71 c8
280  e1 09 cc 59 13 61 62 07 8b 17 e5 e5 e7 4f db 49 01 c4 6a 17 2e 25 15 6d bd 35 43 87 39 f3 a4 da
2a0  ec 96 ea dc fc 78 a4 77 9a dd 07 26 70 f6 5f 6d d1 0c 74 96 5c f3 8b f2 f2 d6 85 42 b4 54 99 d4
2c0  58 f5 2d c9 25 63 35 9c 87 47 48 90 f6 dd 47 61 d8 24 76 6e f6 4f 07 fd 5b 5c 38 12 ed 9c b4 4d
2e0  85 69 47 e0 c2 b2 02 f4 b9 fa 7d ce c3 da 05 03 53 6d a5 1d 65 99 92 19 72 25 96 2c b6 63 2c c1
300  c9 ff 91 35 e2 20 a3 d9 33 ff 8d fa 2b 24 61 12 93 ad ae 45 99 76 1b 2e 0e 32 2a 36 7c a3 ea f5
320  44 33 da 78 95 27 53 6d d5 6a 26 c7 f9 5f b7 01 cf 9e 2f 00 52 68 11 70 fa 95 50 ad 69 bd 5e 15
340  f6 9c 81 5f 1b c7 f7 79 fa 18 30 47 dd 86 f4 61 b1 a3 e3 3b 97 ec 3d 59 b3 17 c4 8d 36 de ba 7d
360  8d fc d6 e3 71 a8 d9 32 1e 7e d7 79 c0 a4 44 66 44 16 15 2c ad f5 e1 17 64 ba f0 5f 11 79 cb 8f
380  fa 4c 42 0a d3 5f b5 d8 f4 39 73 b9 c7 33 da e1 e5 55 1a 57 00 14 fc 03 4f 08 ff 76 4c 64 b5 e1
3a0  c9 7d 75 d7 a2 40 49 7b 01 66 9f d3 e8 25 55 69 f9 64 4a 2f 5f 7d 82 36 1a 08 d7 dd 46 35 8f 79
3c0  47 3e 6b 5d 65 c0 37 66 5e 7b c0 94 69 30 84 ff 7b 1f 76 60 dd 77 e8 03 fa 95 75 e0 5d 3d 43 fb
3e0  e3 d0 74 0c 11 ee 51 eb f1 af 9b 47 08 98 f7 1f 75 4a 7a d9 bf 5e f1 7a f2 14 4c dc 95 4e 4f 69
400  e8 13 b0 0f 5f e9 4e 93 1d b2 b3 37 cd 10 44 c3 e7 50 e0 9b 68 b2 18 e1 41 5e 25 54 4c b9 52 83
420  65 96 0b e4 bf 02 62 c3 5e 6d f3 0f 35 85 5e 5e 2f 09 63 8a 14 61 20 1b 0d 53 1e 53 42 96 ba 19
440  12 dc 73 d0 5d a3 de 37 9e f4 b2 c2 40 3b 2b 41 e6 57 d6 45 37 11 03 09 ad e0 1b 40 78 fe d6 c2
460  da cc 31 05 3e 9d 28 ff cf a4 13 db 62 8a 68 2e 95 1f 88 23 63 9a a7 d1 1b 9d 79 60 b1 ac 35 04
480  4f bb c8 3e d4 5f 2e a6 9c b4 4b 1c a5 f9 89 fa 9e ba fc 23 2e 44 45 0a c8 55 44 9b aa 53 d6 f2
4a0  39 f3 a6 5a 1f 59 d3 3f 06 1e c4 14 35 db 63 48 cf ef df d4 0b 4c 42 20 6f 16 63 5a 82 b6 25 9b
4c0  52 d9 ec 0f 0f 9f 0a fb 85 6a 2c e0 6f fa 23 29 9e b4 0f 05 db 50 74 02 83 28 6b a9 ba 71 b4 20
4e0  bd 47 c5 18 c3 af 7c eb 15 a9 05 3f 26 d7 de a7 89 31 11 9b 1c 58 64 ae a4 96 3a 55 6b 06 50 84
500  36 6b 8a cc 2d 36 7a d2 2c 7f 5a ec d2 2d 1c d1 c3 57 2a 2e 52 bf 26 cb 46 00 e0 d7 05 85 ae 38
520  a7 12 94 78 78 d4 38 07 8c 59 a0 1d 5f 34 f3 6c 08 c1 87 97 5e 98 b4 a7 9b b3 93 37 12 16 72 d6
540  ef cf 39 6b f8 33 12 d8 9b 51 b4 15 d7 71 3c f3 5b 19 ea e7 ae 71 4b 50 93 7e ee 11 a1 9e 38 58
560  a8 98 0a 4c 1b 52 33 24 b9 9f 08 e0 a2 d1 a2 2b 93 47 e2 43 fb ad a5 38 1a fe 0a 09 40 fc ca b0
580  ca 34 52 c2 6f 15 b3 82 f3 67 bb 23 89 7e fe fd 19 30 f8 db 53 9a ec d9 32 ea c6 46 32 c1 d2 4a
5a0  61 42 de 11 6d 49 3d 7d 1c 33 14 b0 37 56 cc 07 0d 53 3b cc 62 6d 2f bf 38 a2 59 d4 33 f2 cb 5b
5c0  52 d1 65 66 f6 a9 f5 79 bb 87 18 bf 7b d5 db 01 01 00 00 00 01 00 00 00 00 00 00 00 00 00 00
5e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ff ff ff ff 04 02 5a 01 00 ff ff
600  ff ff 02 20 fa 07 01 00 00 00 00 23 21 02 7a 46 eb 51 35 88 b0 1b 37 ea 24 30 3f 4b 62 8a fd 12
620  cc 20 df 78 9f ed e0 92 1e 43 ca d3 e8 75 ac 88 fe 41 00 00 00 00 00 17 a9 14 7d 46 a7 30 d3 1f
640  97 b1 93 0d 33 68 a9 67 c3 09 bd 4d 13 6a 87 00 00 00 00
```

**Header:**

Version:

`04 00 00 00`

Previous block hash:

```
13 d6 d1 a4 10 51 42 19
f7 2f f3 a0 df d5 c3 8b
62 1c c2 c6 68 78 4d 2f
d6 fd 10 8f 48 00 00 00
```

Merkle Root hash:

```
30 16 31 55 23 12 34 9d
d5 3b 6b 9e 23 1d f8 bc
b8 c2 d3 32 64 cc 02 f5
cd d9 a9 69 fb 93 80 50
```

Final sapling root hash:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

Time:

`3f 85 13 58`

Bits (Difficulty):

`bf c3 03 1e`

Nonce:

```
1b b2 b5 50 a4 01 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 03
```

Equihash solution (the 0xfd4005 here is used to decode the length of the array of bytes, 0xfd means the size is stored as a 2 byte integer 0x4005 == 1344 bytes):

`fd 40 05 .. 7b d5 db(1344 bytes until address 0x5ce)`

**Transactions:**

Transaction input array size (one transaction):

`01`

Version (only 4 bytes here as is not overwinter):

`01 00 00 00`

Input to transaction array size (one input):

`01`

OutPoint:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
ff ff ff ff
```

Script (first byte is length, 4 bytes long):

```
04
02 5a 01 00
```

Sequence:

`ff ff ff ff`

Transaction output array size (two transactions):

`02`

1st transaction output amount (17300000, 0.173 ZEC):

```
20 fa 07 01 00 00 00 00
```

1st transaction output script (first byte is length, 35 bytes long):

```
23
21 02 7a 46 eb 51 35 88
b0 1b 37 ea 24 30 3f 4b
62 8a fd 12 cc 20 df 78
9f ed e0 92 1e 43 ca d3
e8 75 ac
```

2nd transaction output amount (4325000, 0.04325 ZEC):

```
88 fe 41 00 00 00 00 00
```

2nd transaction output script (first byte is length, 23 bytes long):

```
17
a9 14 7d 46 a7 30 d3 1f
97 b1 93 0d 33 68 a9 67
c3 09 bd 4d 13 6a 87
```

Locktime:

```
00 00 00 00
```