Blockchain Acceleration Using FPGAs - Elliptic curves, zk-SNARKs, and VDFs

Ben Devlin

September 30th 2019

Presentation Outline

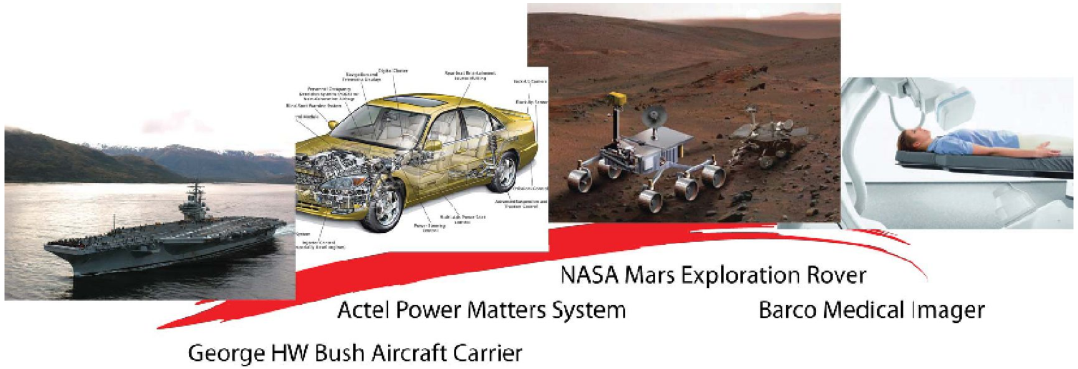
- Introduction
    - Self introduction
    - FPGAs
- FPGA acceleration project
    - Equihash
    - Elliptic curve cryptography
        - secp256k1 engine (transparent transactions)
    - zk-SNARKs
        - bls12-381 pairing coprocessor (shielded transactions)
    - Future direction
- VDFs
    - Overview, example use case
- Conclusion
- Acknowledgments

Self introduction

- Graduated from Massey University, New Zealand with BE in computer systems
- University of Tokyo, Japan in EE, with Ph.D. in Electronic Engineering, focus was on asynchronous circuits design and new FPGA architectures, RSA acceleration
- First worked at Xilinx in FPGA architecture research group in California, afterwards at Tower Research (HFT) in New York
- Currently on one year non-compete
- Was able to work on Zcash grant project for FPGA acceleration

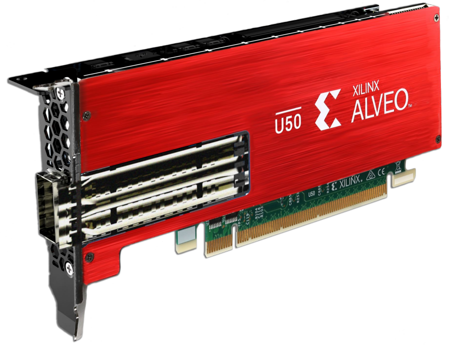
Field Programmable Gate Array

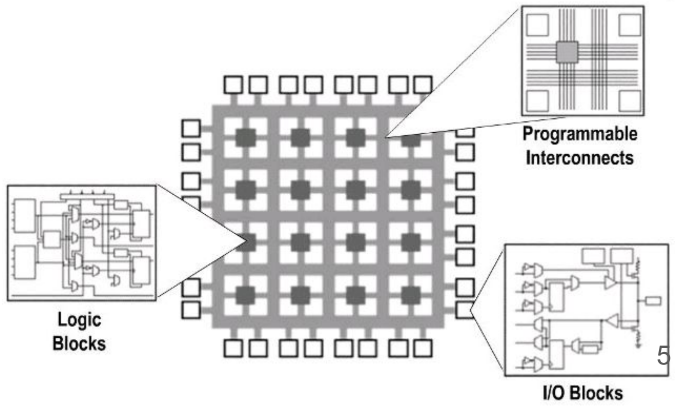
- "Field Programmable" - Integrated chip that can be configured after manufacturing - as opposed to application specific integrated circuit (ASIC)
- Wide usage in many applications
  - Often not consumer visible
- Recently has become more available
  - Amazon AWS F1 instances, ~$1.65 an hour to rent
  - High level synthesis (HLS), SDK driven flow


NASA Mars Exploration Rover
Actel Power Matters System
Barco Medical Imager
George HW Bush Aircraft Carrier

FPGAs

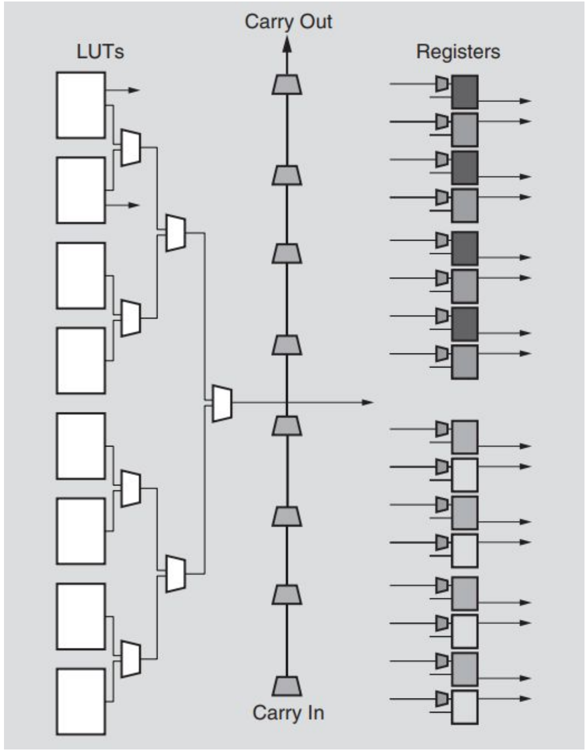
- Consists of array of logic gates and routing interconnects, and configuration memory
- Programs written in hardware-description language (SystemVerilog, VHDL) and then compiled (synthesis, place & route) into a chip image (bitstream)
- Can implement complex circuits
  - 8.938M Logic Cells, 1,976 IO pins, 165MB on chip RAM, 20B+ transistors (Xilinx VU19P)
  - Access 16nm, 7nm technology
  - Will have hard digital signal processors (DSP), ARM CPU cores, PCIe gen4 controllers, artificial intelligence engines

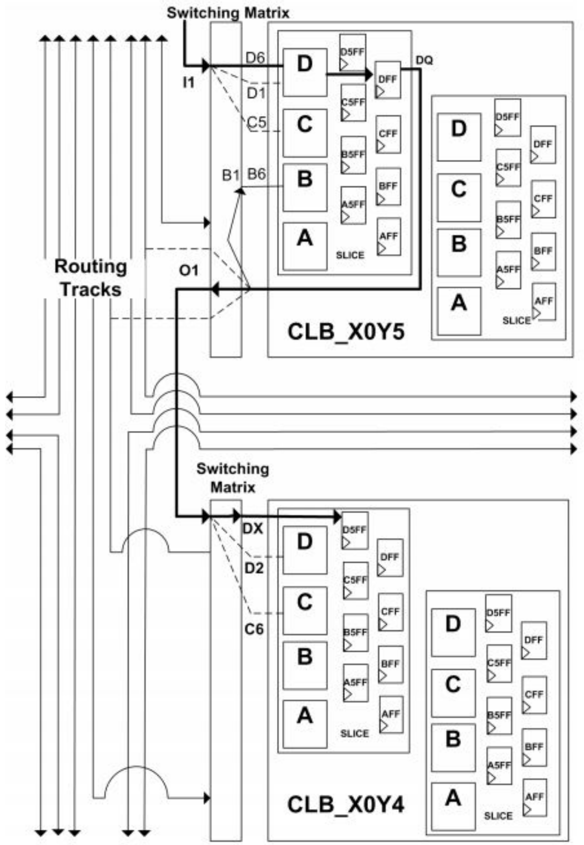




FPGAs

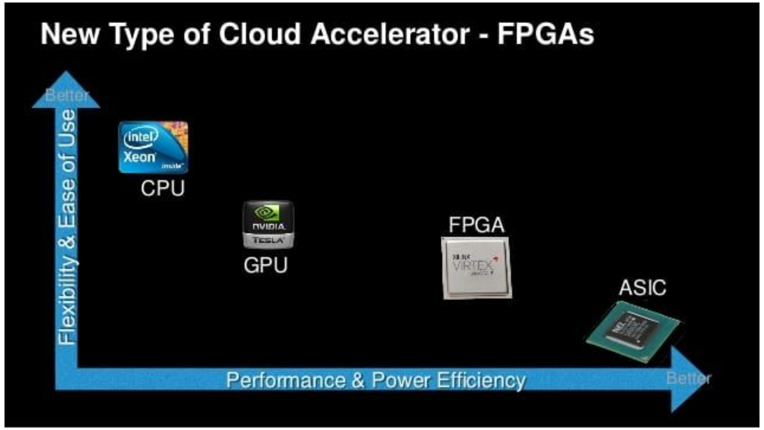
- Circuits are created by connecting routing wires to blocks
- Performance will depend on the worst-case timing between flip-flops (registers)
- Usually 100's - 400's MHz
  - Much slower than 3GHz+ modern CPU





FPGA Acceleration Targets

- FPGA acceleration is ideal for tasks that
  - Can be parallelized
    - FPGA can instantiate 1000’s of modules in parallel
  - Contain custom logic not easily implemented on a CPU
    - i.e. 381 bit data path, hash mixer functions
  - Not memory bandwidth bound
    - FPGAs will have 100’s of megabits on board
    - HBM / DDR allows for GB’s of memory but latency starts to take a hit
  - Does not require a lot of communication from SW to FPGA
    - PCIe can be 100’s to 1000’s of ns


New Type of Cloud Accelerator - FPGAs
Better
Flexibility & Ease of Use
CPU
GPU
FPGA
ASIC
Performance & Power Efficiency
Better

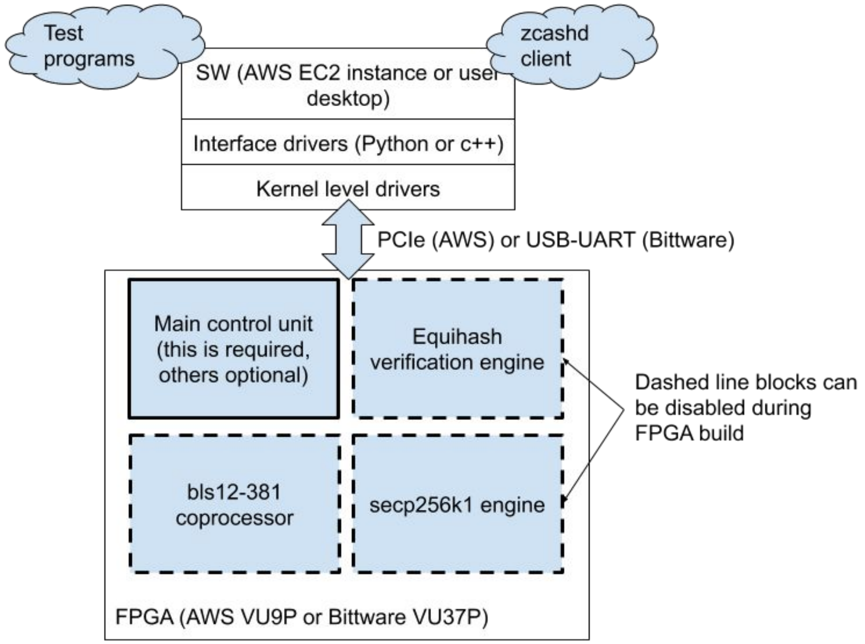
FPGA acceleration project

- Goal to develop FPGA code to accelerate Zcash blockchain
  - Focus on improving blockchain (not developing a miner)
  - Develop open source code code that doesn't exist
  - Research
- Three main goals
  - Equihash
    - Block header verification
  - Transparent transaction verification
    - ECDSA on secp256k1
  - Shielded transaction verification (zk-SNARK)
    - Pairings on bls12-381

Zcash

Top level of FPGA design

- Top level of FPGA has control unit for interfacing with SW
  - Developed interfaces USB-UART + python lib (Bittware board) and PCIe + cpp library (for AWS)
  - Uses messages with common header and custom payload to communicate with SW
- FPGA has optional accelerators that are enabled or disabled during build

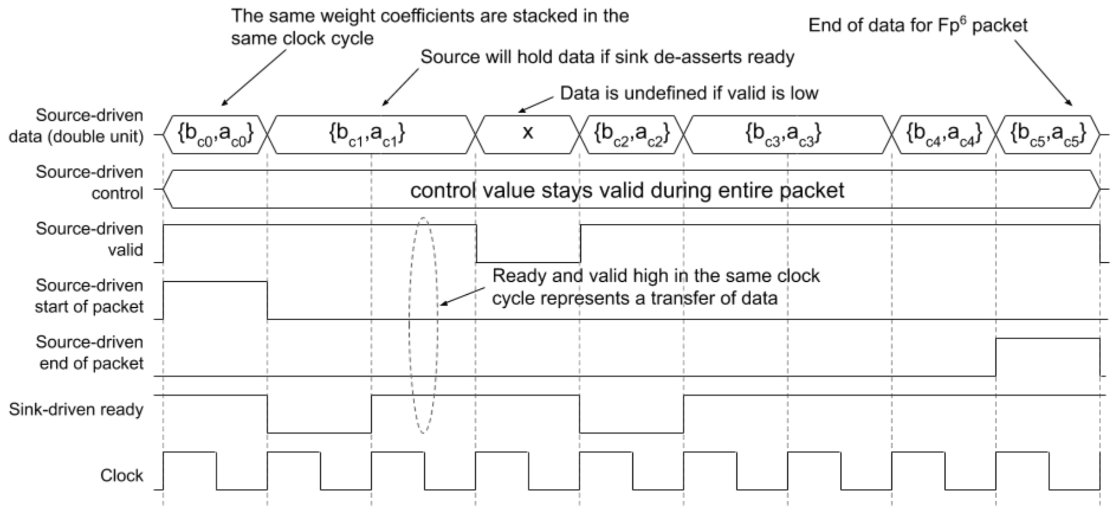

Test programs
zcashd client
SW (AWS EC2 instance or user desktop)
Interface drivers (Python or c++)
Kernel level drivers
PCIe (AWS) or USB-UART (Bittware)
Main control unit (this is required, others optional)
Equihash verification engine
bls12-381 coprocessor
secp256k1 engine
Dashed line blocks can be disabled during FPGA build
FPGA (AWS VU9P or Bittware VU37P)

```
typedef __packed__ struct {
  uint32_t cmd_type;
  uint32_t len;
} fpga_header_t;
```

```
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;
```

FPGA internal connections use SystemVerilog interfaces

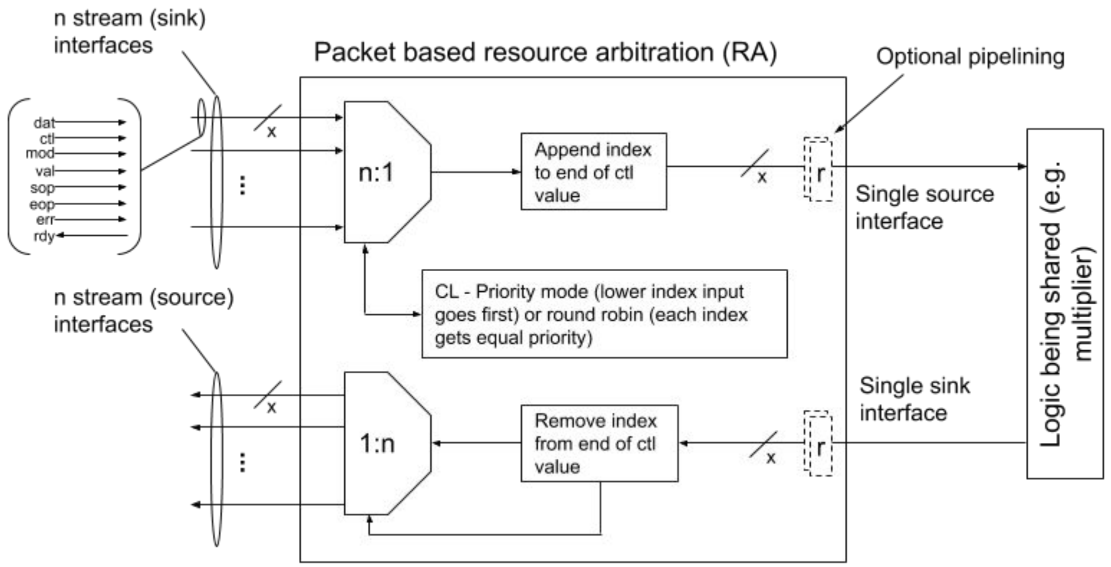
- All connections on the FPGA use interfaces
  - Allows for back pressure
  - Source holds data when receiver ready is low, receiver only takes data when valid is high
  - Similar to AXI stream
- Easy to add resource arbitrators and have common blocks shared
- Example "packet" shown



The same weight coefficients are stacked in the same clock cycle
Source will hold data if sink de-asserts ready
Data is undefined if valid is low
End of data for Fp^6 packet
Source-driven data (double unit)
{b_{c0},a_{c0}}
{b_{c1},a_{c1}}
x
{b_{c2},a_{c2}}
{b_{c3},a_{c3}}
{b_{c4},a_{c4}}
{b_{c5},a_{c5}}
Source-driven control
control value stays valid during entire packet
Source-driven valid
Ready and valid high in the same clock cycle represents a transfer of data
Source-driven start of packet
Source-driven end of packet
Sink-driven ready
Clock

FPGA supports modularity with resource arbitrators

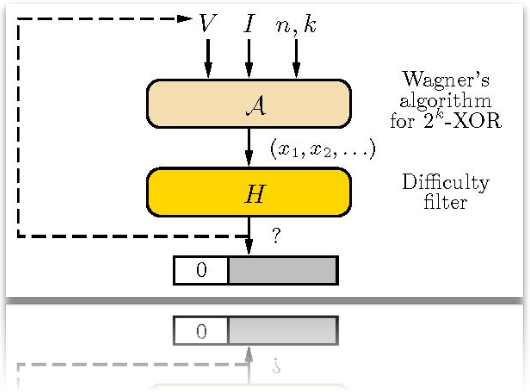
- Control signal grows by ceil_log2(n) as the index is appended by control logic
- Optional pipelining registers so so provide better performance
  - Otherwise ready signal from sink to source needs to be combinatorial (line wire delay)


n stream (sink) interfaces
dat
ctl
mod
val
sop
eop
err
rdy
x
Packet based resource arbitration (RA)
Optional pipelining
n:1
Append index to end of ctl value
x
r
Single source interface
Logic being shared (e.g. multiplier)
CL - Priority mode (lower index input goes first) or round robin (each index gets equal priority)
n stream (source) interfaces
x
1:n
Remove index from end of ctl value
x
r
Single sink interface

Equihash

Equihash proof of work

- Zcash proof of work, generalization of the birthday problem and requires finding colliding hash values
- Has parameters n and k to adjust the algorithm memory and time requirements. Currently Zcash uses n=200, k=9
- Algorithm flow:
  - Generate hashes of the block header, increasing index, and a nonce
  - Need $2^{n/(k+1)+1}$ hashes (2^{20} x 25B which is ~50MB RAM), was originally thought to be ASIC resistant, but ASICs exist now. Has been talked about a plan to move to n=144, k=5 which would require 2^{25} x 25B hashes which is ~800MB RAM
  - Then you need to find 2^k hashes so that the XOR equals 0, as well as some index ordering and duplicate restrictions
  - Resulting block header with solution is run through SHA256d and checked with difficulty filter

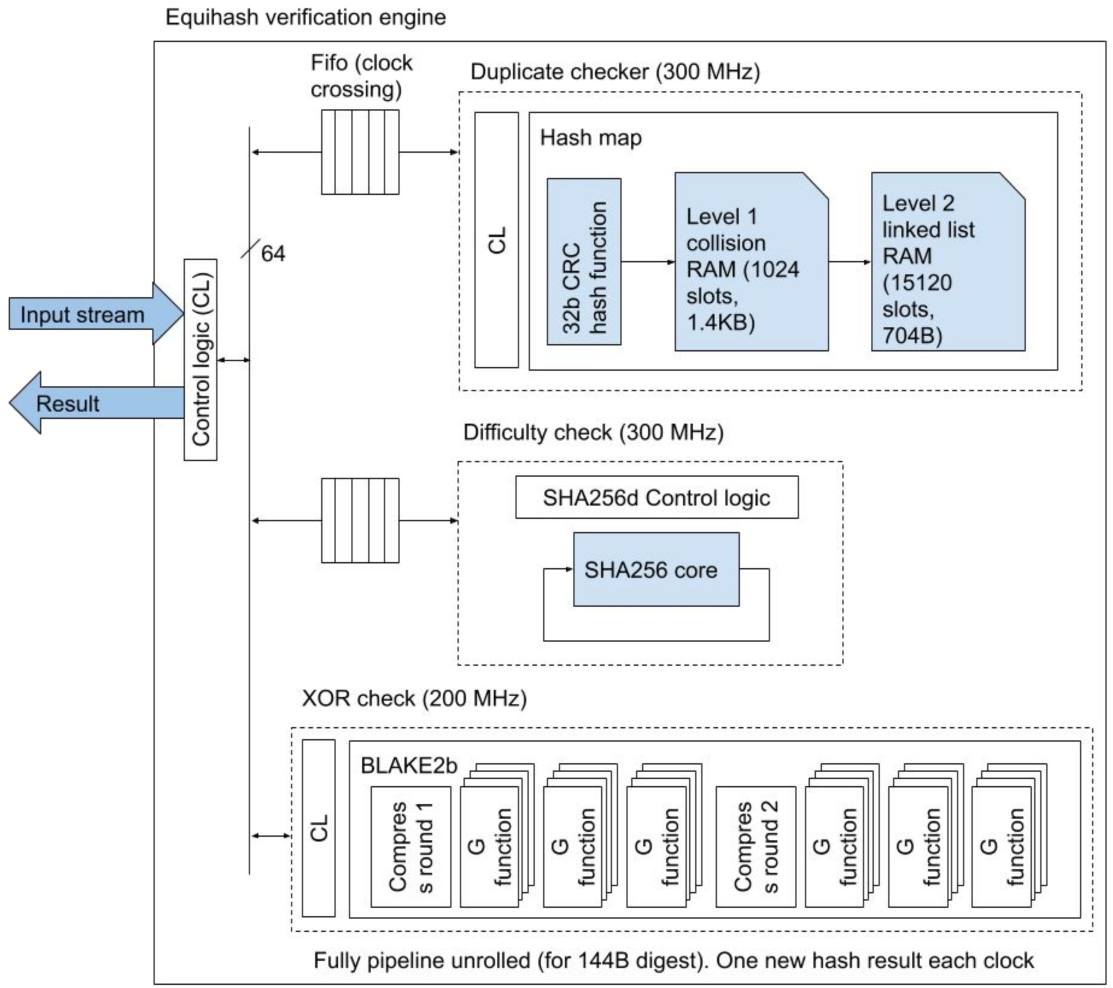


Accelerating Equihash verification

- Block header contains miners Equihash solution, we want to design a FPGA circuit that can check the conditions:
  - Hash list XORs to zero (solution check)
  - Order is followed (tree nodes have n bits 0)
  - No duplicate indexes
  - Difficulty filter passes
- Can be easily parallelized
- Need to implement
  - High performance BLAKE2b core (hash function used in Zcash)
  - Hash map
  - SHA256d
  - Control logic

Equihash engine

- Input is streamed in on 64bit bus and split into three
- Using 200MHz and 300MHz clock domains
- Hash map uses two RAMs and 32bit CRC for hash function
- XOR check decodes solution list and performs BLAKE2b hashes
  - 64 clock cycles for a hash result (two full rounds), new result each clock cycle



Equihash engine results

- 207x speedup compared to 3GHz CPU (benchmark via cycle counter) running Zcashd code
  - Able to exploit parallelism in BLAKE2b (64 calculations at once)

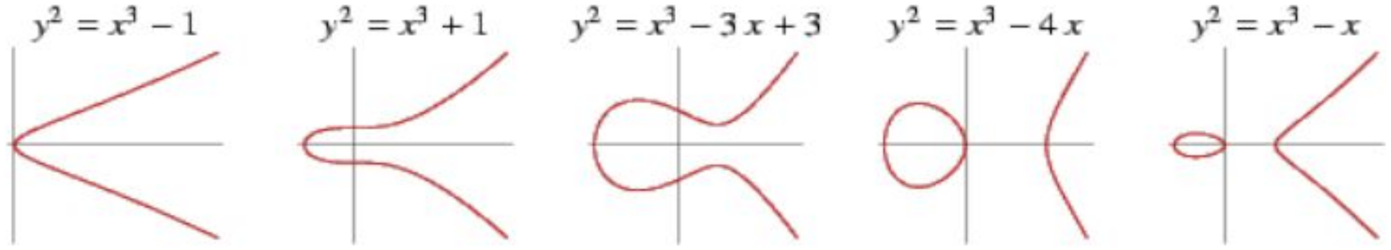
LUT	**FF**	**DSP**	**BRAM**
87914 (3%)	54362 (3%)	0	6 (0.2%)

	FPGA clock cycles	**FPGA throughput**	**CPU cycles**	**3GHz CPU throughput**
Solution check	600 @200MHz			
Index order check	356 @200MHz			
Duplicate check	1443 @300MHz			
Difficulty check	1068 @300MHz			
Equihash solution verification	1068 @300MHz	207K op/s	~2868040	~1K op/s

Elliptic curve cryptography

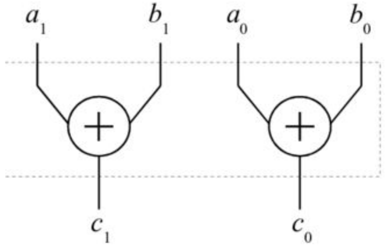
Elliptic curve cryptography (ECC)

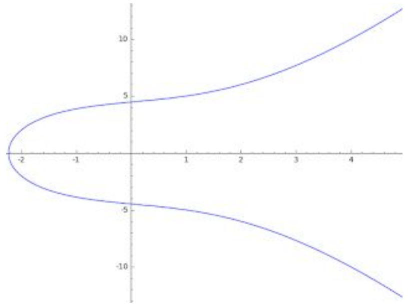
- An approach to public-key cryptography first proposed in 1985, based on elliptic curves over finite fields
  - Two keys, one public Q, (Q = x.G) and one private (x). G is the generator or “base” point
- Relies on the fact that it is easy to calculate Q if you know x and G, but very difficult to calculate x only knowing Q and G
- The advantage of ECC compared to RSA is that it requires smaller key sizes (bits) for equivalent levels of security
  - e.g. 2048 bit key sizes when using RSA in ECC only requires 224 bit keys



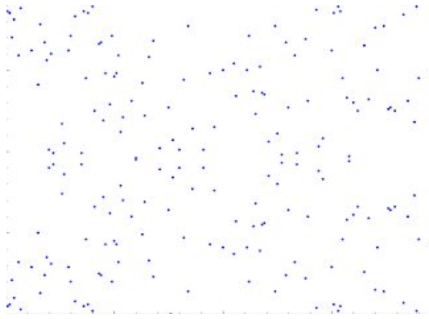
Elliptic curves over fields

- We will operate on integer values on a curve modulo a prime p
  - This is F(p) or Fp
- Also exists elliptic curves over binary finite fields
  - F(2^m)
  - This makes the hardware implementation much faster (as there is no carry propagation), but are less well studied and possibly not as secure as prime field curves
  - Both curves we accelerate in this project are over prime fields





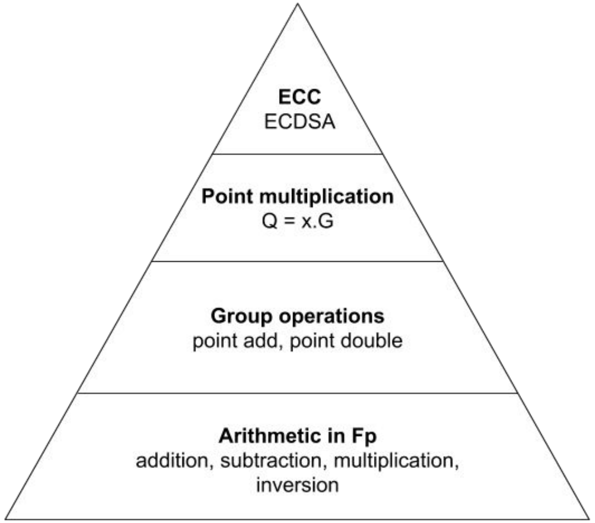
$y^2 = x^3 + 4x + 20$



$y^2 = x^3 + 4x + 20 \text{ mod } p$

Curve operations & ECDSA

- RSA relies on modular exponentiation (c^d mod n), which can be calculated by repeated multiplications and squarings
- ECC relies on point multiplications (Q = x.G)
  - x is an integer scalar, G is a generator point for the curve (so has x and y coordinates), Q is the resulting point on the curve
  - These can be calculated by repeated point doubling and additions.
- Elliptic Curve Digital Signature Algorithm (ECDSA) allows us to use ECC to securely verify a person's signature of a hashed message

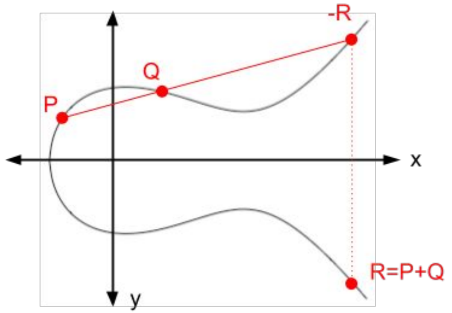


1. $w = s^{-1} \bmod n$
2. $u_1 = (h * w) \bmod n$
3. $u_2 = (r * w) \bmod n$
4. $(x_2, y_2) = (u_1.G + u_2.Q) \bmod n$

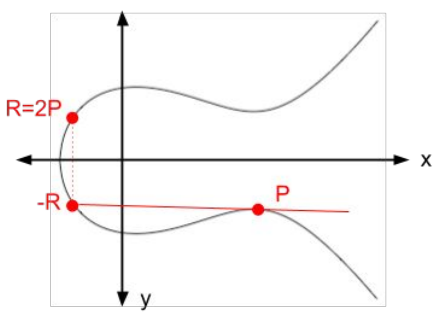
ECC multiplication = point double and point add

- To calculate Q = x.G we use repeated point_add and point_double depending on the binary values in x
- Point addition and doubling formulas can using affine coordinates involve expensive divisions (require an inversion), so we use Jacobian coordinates (X, Y, Z), $x = X/Z^2$ and $y = Y/Z^3$
    - This way we don't have to do any divisions

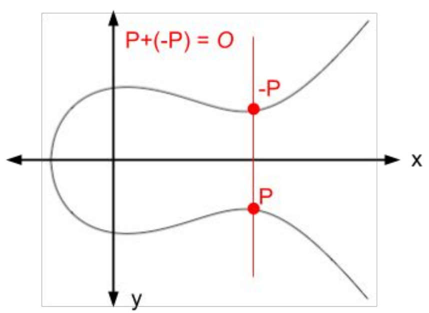
```
N ← G
Q ← 0
for i from 0 to m do
    if $x_i$ = 1 then
        Q ← point_add(Q, N)
    N ← point_double(N)
return Q
```



(a) Point addition



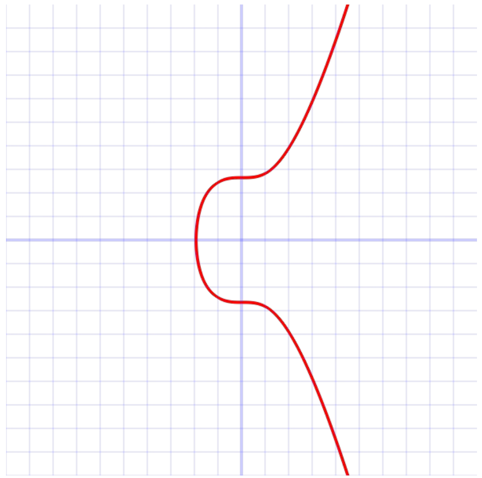
(b) Point doubling



(c) Point at Infinity

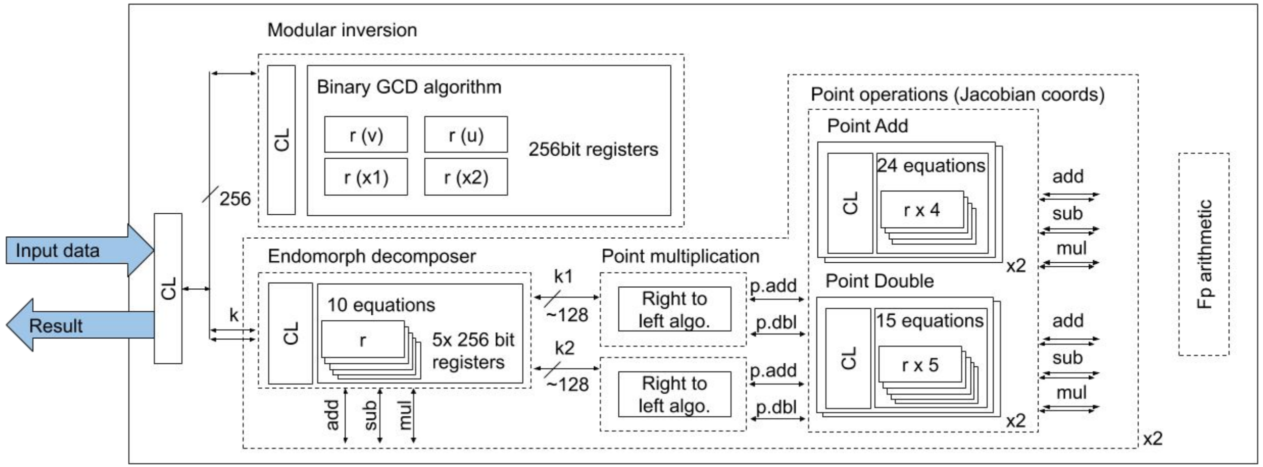
Secp256k1

- secp256k1 is an elliptic curve with 256 bit prime modulus
- Originally used by Bitcoin, Zcash also uses for transparent transaction signature
- Several hardware friendly features
  - $p = 2^{256} - 2^{32} - 2^{9} - 2^{8} - 2^{7} - 2^{6} - 2^{4} - 1$
    - Can do modulo reduction with bit shifts and additions, do not need to use Barret or Montgomery reduction for arithmetic in Fp
  - Has an efficient endomorph
    - A point multiplication Q = x.G can be split into two half size multiplications which can be done in parallel, $x = x_1 + x_2\lambda$ (mod n)



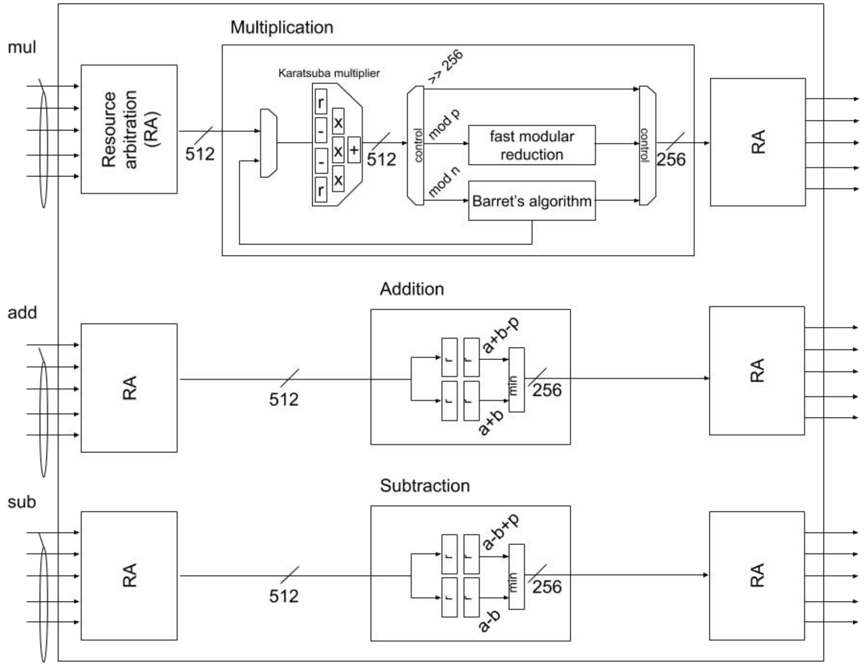
FPGA secp256k1 engine

- Top level has control logic (CL) for parsing ECDSA messages and returning results
- For calculating inverses we use the binary GCD algorithm which only involves additions, subtractions, and bit shifts
- We use the endomorph and run the 4 point multiplications for ECDSA



Secp256k1 Fp arithmetic unit

- We have a single multiplication, addition, and subtraction block
  - Shared via resource arbitration
- Multiplier uses 2 stages of Karatsuba algorithm recursions, each recursion takes 3 clock cycles
  - $x.y = z_2B^{2m} + z_1B^m + z_0$
    - $z_2 = x_1.y_1$
    - $z_1 = (x_0-x_1).(y_0-y_1) + z_2 + z_0$
    - $z_0 = x_0.y_0$
- Modular reduction is either
  - mod p = fast reduction only using shifts and additions, 3 clocks
  - mod n = Barrets algorithm, requires 2 more multiplications and 2 subtractions
  - Simple 256 bit shift (used during endomorph calculation)
- Subtraction and adder use double pipeline to check if we need to add or subtract modulus



Secp256k1 engine results

- Achieved 1.5x speedup
  - Not as great likely due to only 4x parallelism or heavily optimized SW

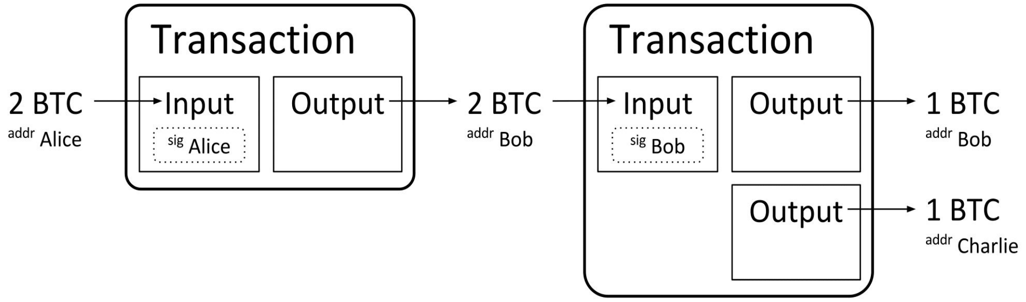
	LUT	FF	DSP	BRAM
secp256k1 ECDSA core (without endomorph enabled)	57697 (4.4%)	31751 (1.2%)	144 (1.6%)	2 (0.1%)
secp256k1 ECDSA core (with endomorph enabled)	98792 (7.5%)	61909 (2.1%)	144 (1.6%)	2 (0.1%)

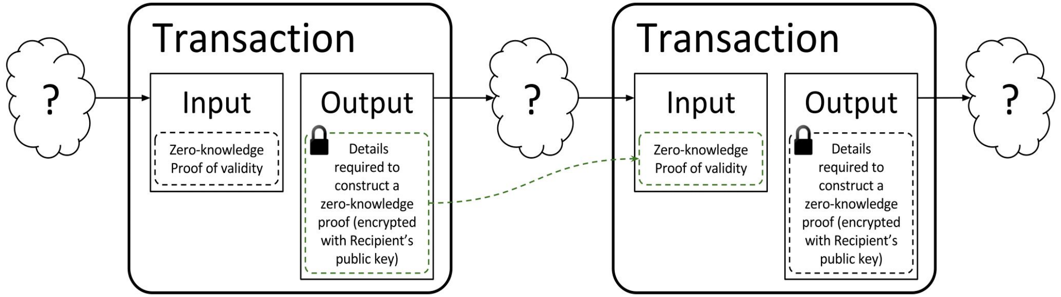
	FPGA clock cycles	FPGA throughput	CPU cycles	3GHz CPU throughput
Point double mod p	54	3.7M op/s		
Point add mod p	104	1.9M op/s		
Inversion mod n	708	282K op/s		
secp256k1 ECDSA core (without endomorph enabled)	20224	9.9K op/s/core	223350	13.4K op/s
secp256k1 ECDSA core (endomorph enabled)	10100	20K op/s/core		

zk-SNARKs

zk-SNARKs

- Stands for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge
- Allows you to create a proof that you know something without revealing any knowledge
- Allows for shielded transactions on the Zcash blockchain
- Different proving systems, complexity is in ECC operations (point mult, pairing), FFT calculations



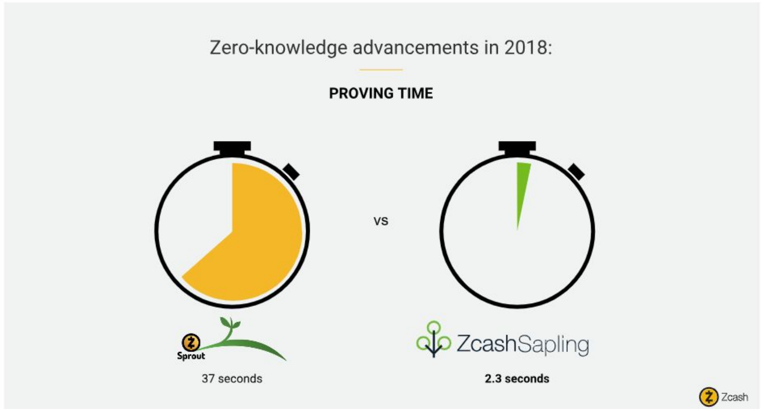


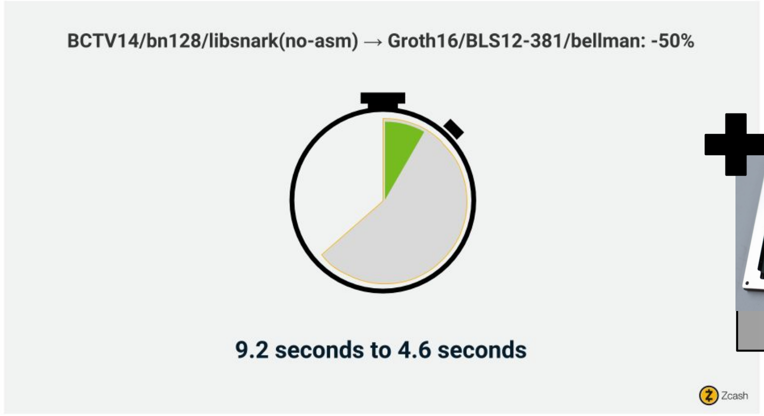
?
Transaction
Input
Zero-knowledge Proof of validity
Output
Details required to construct a zero-knowledge proof (encrypted with Recipient's public key)
?
Transaction
Input
Zero-knowledge Proof of validity
Output
Details required to construct a zero-knowledge proof (encrypted with Recipient's public key)
?

https://electriccoin.co/blog/zsl/

zk-SNARKs used by Zcash blockchain

- In October 2018 Zcash had a network upgrade “Sapling” (client v2)
- Greatly improves performance of shielded transactions (zk-SNARKs)
- Uses Groth16 proving system, over bls12-381 curve
  - Verification pairing operations
  - Creating a proof is multi-exp and FFTs
- We decided to accelerate the bls12-381 curve (pairings and multi-exp)

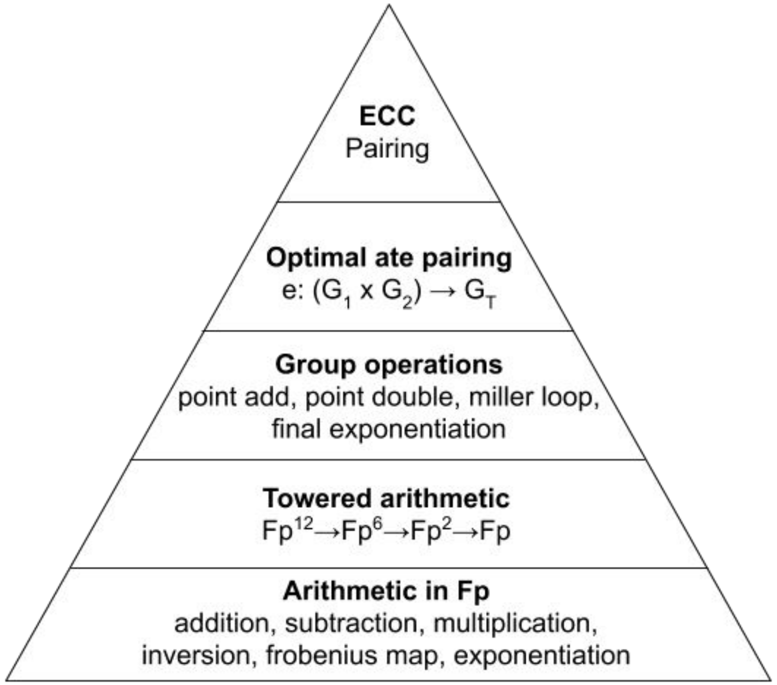




https://electriccoin.co/blog/reducing-shielded-proving-time-in-sapling/

Pairings on elliptic curves

- Compared to previous operations, we now add a lot more complexity
- We are operating on points in G_1 and G_2 which are in fields Fp and Fp^2, with the pairing producing an element in Fp^{12}
- In addition to point addition and doubling, we need to run Miller loop and final exponentiation algorithms
- We tower the extension fields for efficiency
- Need to be able to perform more complex arithmetic on all extension fields


ECC
Pairing
Optimal ate pairing
$e: (G_1 \times G_2) \rightarrow G_T$
Group operations
point add, point double, miller loop, final exponentiation
Towered arithmetic
$Fp^{12} \rightarrow Fp^6 \rightarrow Fp^2 \rightarrow Fp$
Arithmetic in Fp
addition, subtraction, multiplication, inversion, frobenius map, exponentiation

Towered arithmetic

- Adopting a tower of extensions (such as $Fp^{12} \rightarrow Fp^{6} \rightarrow Fp^{2} \rightarrow Fp$) has advantage that we can perform Karatsuba multiplication methods rather than schoolbook at each level
  - For example a schoolbook multiplication of a single Fp^{12} element would be 144 Fp multiplications, but using Karatsuba algorithms and towering we only need 54
- We implement towered arithmetic using equations similar to point addition and doubling
- Each block on FPGA contains interfaces to lower extension field arithmetic
- Field elements are stored as streams on the FPGA, where each coefficient is streamed in ascending order so we never have busses wider than the base field element size (e.g. 381 or 762 bits)
  - e.g. Fp^{6} is streamed $\longrightarrow\{c_0, c_1, c_2, c_3, c_4, c_5\}\longrightarrow$
  - Arithmetic with multiple elements is streamed $\longrightarrow\{a_0b_0, a_1b_1, a_2b_2, a_3b_3, a_4b_4, a_5b_5\}\longrightarrow$

ECC point double on extension field

- Example of point double formula, same as formula for base group Fp, just all multiplications / etc are in Fp^2
- FPGA streaming architecture allows for point operations in any field

```
function fp2_jb_point_t dbl_fp2_jb_point(input fp2_jb_point_t p);
  fe2_t I_X, I_Y, I_Z, A, B, C, D, X, Y, Z;

  I_X = p.x;
  I_Y = p.y;
  I_Z = p.z;
  A = fe2_mul(I_Y, I_Y);
  B = fe2_mul(fe2_mul(4, I_X), A);
  C = fe2_mul(fe2_mul(8, A), A);
  D = fe2_mul(fe2_mul(3, I_X), I_X);
  X = fe2_mul(D, D);
  X = fe2_sub(X, fe2_mul(2, B));

  Y = fe2_mul(D, fe2_sub(B, X));
  Y = fe2_sub(Y, C);
  Z = fe2_mul(fe2_mul(2, I_Y), I_Z);

  return {X, Y, Z};
endfunction
```

Group operations

- In order to compute the optimal ate pairing on FPGA, need to be able to:
    - Miller loop (result is a Fp^{12} element)
        - Point doubling in Fp^2 (these can be reused to implement point multiplication)
        - Point addition in Fp^2
        - Addition, Subtraction, Multiplication in Fp^2
        - Squaring in Fp^{12}, sparse multiplication in Fp^{12}
    - Final exponentiation, $(p^{12}-1)/r$ (result is a Fp^{12} element)
        - Inversion in Fp^{12}, Fp^6, Fp^2, Fp
        - Exponentiation of Fp^{12}
            - Repeated Fp^{12} squarings and Fp^{12} multiplications
        - Frobenius map
            - Multiplications in Fp^2, Fp
        - Subtraction in Fp^{12}, multiplication in Fp^{12}

Bls12-381 curve

- The elliptic curve Zcash uses for zk-SNARKs
- 381 bit prime, embedding degree k=12
- Pairing-friendly curve
  - z-value of bls curve has low hamming weight, makes Miller loop more efficient
  - k is not too large
    - k is the smallest integer such that r divides q^k-1
- Security level ~128 bits, due to subgroup order r ~ 2^{255}
- Prime modulus is not any special form we can take advantage of, so arithmetic needs to use Barret / Montgomery / Lookup table algorithms for reduction
- Rust implementation: https://github.com/zkcrypto/bls12_381

Bls12-381 coprocessor

- Decided to implement coprocessor than has a custom instruction set, data memory, instruction memory
  - Can be programmed by SW to run more complex flows
- Data memory is in slots, one slot is 381 bit + metadata
- Coprocessor can send interrupts to SW
- Both pairings, multi-pairings, and multi-exp are possible

NOOP_WAIT (0x0)
COPY_REG(0x1, a, b)
JUMP(0x2, a)
JUMP_IF_EQ(0x4, a, b, c)
JUMP_NONZERO_SUB(0x5, a, b)
SEND_INTERRUPT(0x6, a, b)

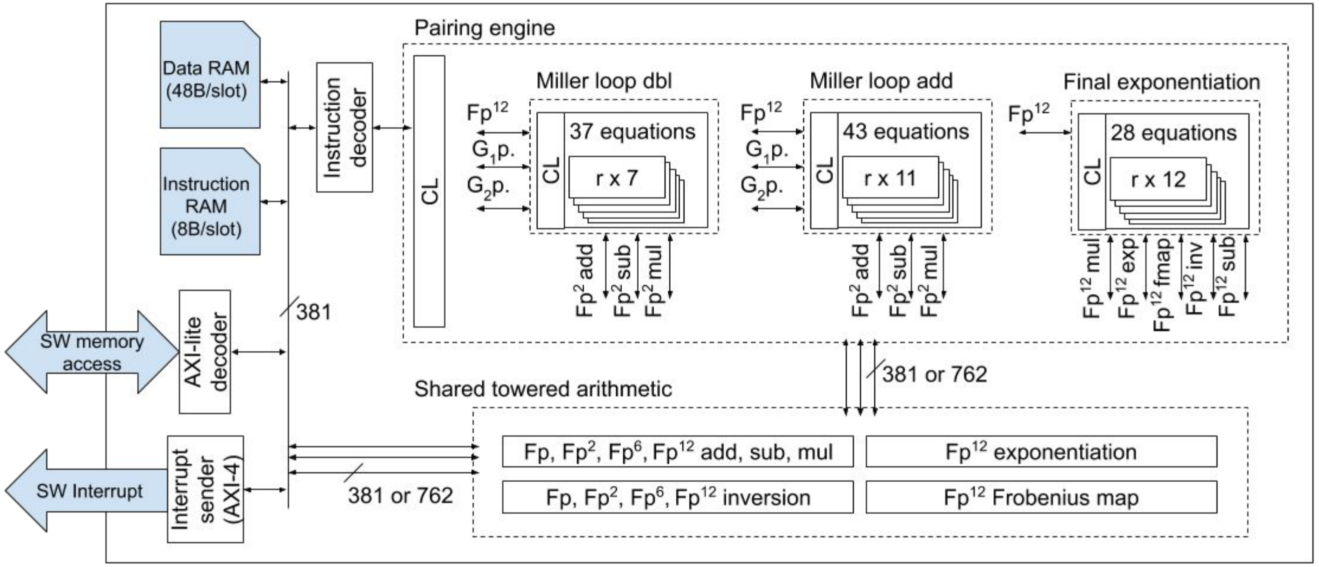
MUL_ELEMENT (0x10, a, b, c)
ADD_ELEMENT (0x11, a, b, c)
SUB_ELEMENT (0x12, a, b, c)
INV_ELEMENT(0x13, a, b)

POINT_MULT(0x20, a, b, c)
MILLER_LOOP(0x21, a, b, c)
FINAL_EXP(0x22, a, b)
ATE_PAIRING(0x23, a, b, c)

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

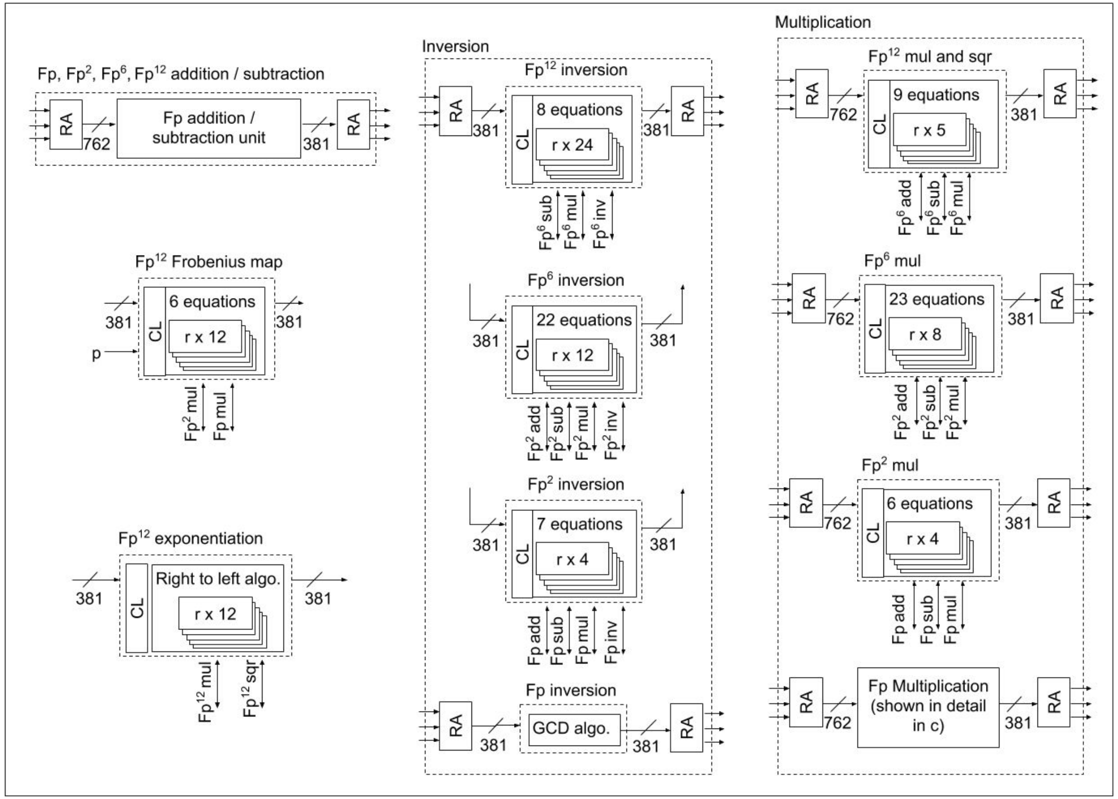
Bls12-381 coprocessor diagram

- SW can access entire coprocessor memory region via AXI-lite interface
- FPGA can initiate sends to SW via interrupt instructions on AXI-4
- Towered arithmetic is shared
- Point multiplication instruction reuses Miller loop blocks



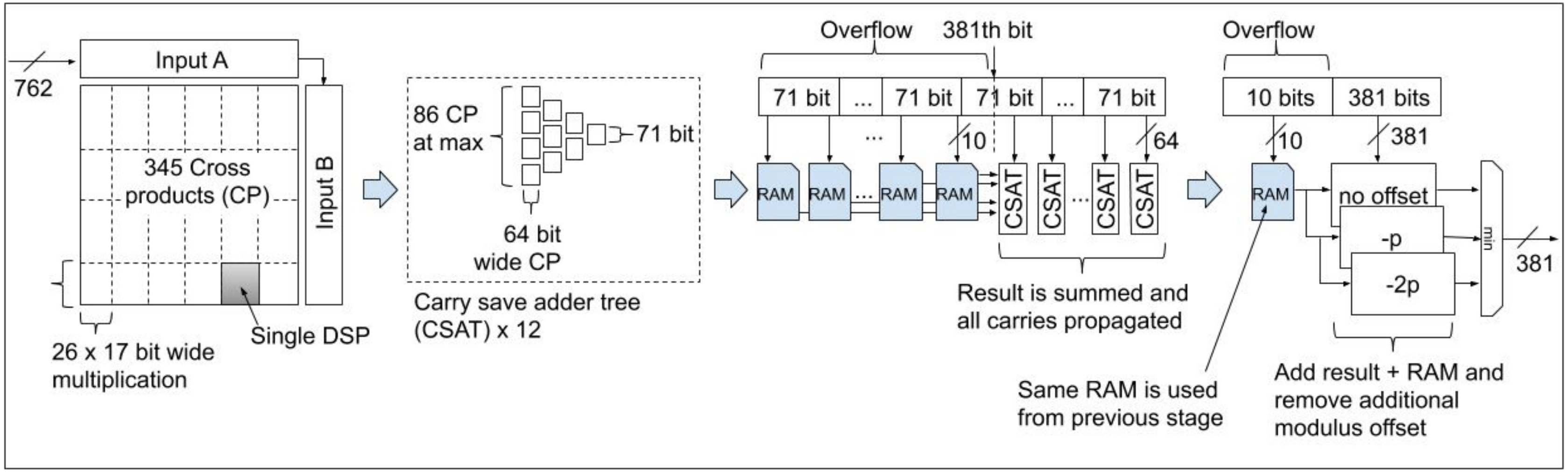
Bls12-381 towered arithmetic

- Can represent towered arithmetic by equations on lower levels
- Addition and subtraction don't need anything special, just add/sub each coefficient
- Fp multiplier is main bottleneck



Bls12-381 Fp multiplier

- Originally we used Karatsuba multiplier with Barret reduction algorithm
  - But this was taking ~30 clock cycles
- Replaced with parallel multiplier + carry look-ahead trees + RAM for reduction
  - Takes 9 clock cycles (3x speedup)
  - Could be better if we could use redundant form



Bls12-381 coprocessor results

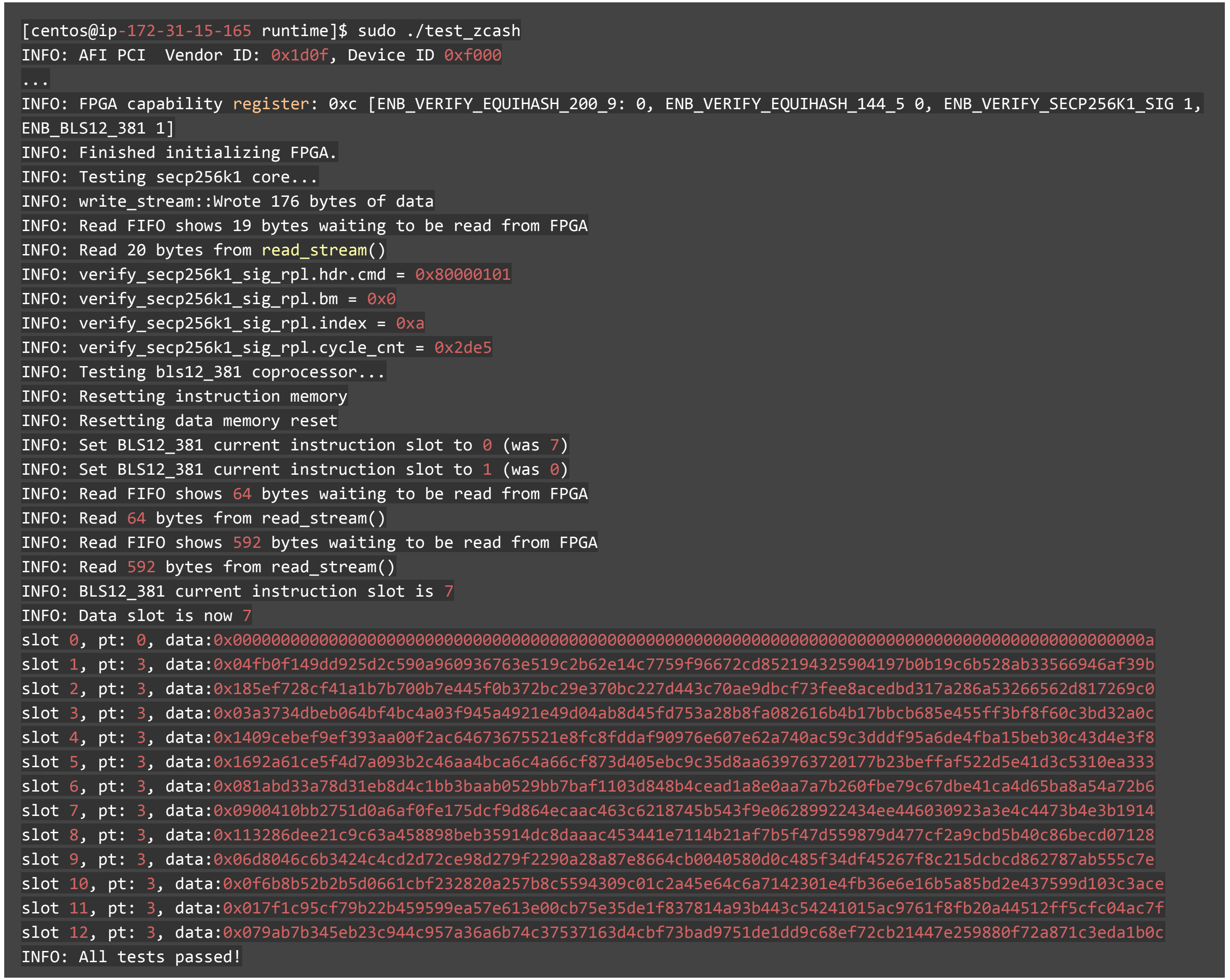
- Achieved 2.9x compared to benchmarked Rust code on 3.7GHz CPU

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%)

	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

AWS F1 FPGA Instance

- Developed and tested flow to simulation / build the Zcash FPGA on AWS
- Created public AFI (FPGA image)
- All documented in user guide in GitHub



```
[centos@ip-172-31-15-165 runtime]$ sudo ./test_zcash
INFO: AFI PCI  Vendor ID: 0x1d0f, Device ID 0xf000
...
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:0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000a
slot 1, pt: 3, data:0x04fb0f149dd925d2c590a960936763e519c2b62e14c7759f96672cd852194325904197b0b19c6b528ab33566946af39b
slot 2, pt: 3, data:0x185ef728cf41a1b7b700b7e445f0b372bc29e370bc227d443c70ae9dbcf73fee8acedbd317a286a53266562d817269c0
slot 3, pt: 3, data:0x03a3734dbeb064bf4bc4a03f945a4921e49d04ab8d45fd753a28b8fa082616b4b17bbcb685e455ff3bf8f60c3bd32a0c
slot 4, pt: 3, data:0x1409cebef9ef393aa00f2ac64673675521e8fc8fddaf90976e607e62a740ac59c3dddf95a6de4fba15beb30c43d4e3f8
slot 5, pt: 3, data:0x1692a61ce5f4d7a093b2c46aa4bca6c4a66cf873d405ebc9c35d8aa639763720177b23beffaf522d5e41d3c5310ea333
slot 6, pt: 3, data:0x081abd33a78d31eb8d4c1bb3baab0529bb7baf1103d848b4cead1a8e0aa7a7b260fbe79c67dbe41ca4d65ba8a54a72b6
slot 7, pt: 3, data:0x0900410bb2751d0a6af0fe175dcf9d864ecaac463c6218745b543f9e06289922434ee446030923a3e4c4473b4e3b1914
slot 8, pt: 3, data:0x113286dee21c9c63a458898beb35914dc8daaac453441e7114b21af7b5f47d559879d477cf2a9cbd5b40c86becd07128
slot 9, pt: 3, data:0x06d8046c6b3424c4cd2d72ce98d279f2290a28a87e8664cb0040580d0c485f34df45267f8c215dcbcd862787ab555c7e
slot 10, pt: 3, data:0x0f6b8b52b2b5d0661cbf232820a257b8c5594309c01c2a45e64c6a7142301e4fb36e6e16b5a85bd2e437599d103c3ace
slot 11, pt: 3, data:0x017f1c95cf79b22b459599ea57e613e00cb75e35de1f837814a93b443c54241015ac9761f8fb20a44512ff5cfc04ac7f
slot 12, pt: 3, data:0x079ab7b345eb23c944c957a36a6b74c37537163d4cbf73bad9751de1dd9c68ef72cb21447e259880f72a871c3eda1b0c
INFO: All tests passed!
```

VDFs

- A Verifiable Delay Functions (VDF) is a function that takes some medium-large quantity of non-parallelizable work to compute, but can be verified very quickly
- Good points for FPGA acceleration
  - Memory is not bottleneck
  - Custom data path
  - Not a lot of communication required between FPGA and SW
  - Compared to CPU can get 10-20x
    - CPU took 3 years vs FPGA 2 months to solve 1999 MIT's time capsule problem (~80 trillion repeated squarings) - was expected to take 30 years originally
- Compared to ASIC maybe ~10x slower (but not 100x)
  - Due to fact this can't be parallelized, so ASIC just has advantage of no programmability → smaller delays → faster clock
- Useful for proof of history, guaranteeing some bound of runtime

VDFs- randomness on the blockchain in Ethereum 2.0

- Being able to have a trustworthy source of reliable random values on the blockchain is a big plus
  - Can be used for a proof of stake system where you need to random select a block to be added to the blockchain
- RANDAO is a scheme that can be used to generate random numbers by having participants each submit their own random number (can't be trusted), but then a central system combines them to produce a trusted random number
  - But the problem is if a dishonest user withholds their number after seeing everyones else because they know it can affect the randomness
- Can be solved with VDFs - you specify the maximum time a participant has to submit their random number (e.g. 5 min), and then the actual random number to be published is the result of a 100min VDF
  - We can produce a reliable bound on the run time even when compared to ASIC

Conclusions

- Developed three main projects on the FPGA
  - **Equihash** verification → 207x speedup
    - Can exploit parallelism in BLAKE2b and required checks 💚
  - Transparent transactions (**secp256k1 engine**) → 1.5x speedup
    - Less opportunity for parallelism, 256 bit data pipeline is not so far from native implementation on CPU (already 15x disadvantage) ❌
    - Multiple cores could increase FPGA advantage 💚
  - Shielded transactions (**bls12-381 coprocessor**) → 2.9x speedup
    - Pairing operation has more parallelism 💚
    - 381 bit data pipeline favours more custom approach 💚
    - Able to implement high performance modulo reduction 💚
  - All modules enabled uses 58% LUT, 17% DSP, 27% BRAM
- All code is open source and released under GPL-3 license
  - Designed to use interfaces and equations, so can be easily ported to other curves or used in other projects

Future direction

- zk-SNARKs also use FFTs for creating proofs which would be good candidates for FPGA acceleration
  - Focus on parallelisms in multi-exp for greater speedups
- Other curves
  - Zcash’s Halo or Bitcoin’s Bulletproof are emerging technologies, don’t use pairings but do use ECC point multiplications
- Optimizing underlying arithmetic on FPGA
  - Use redundant form or RNS

Acknowledgements

We would like to thank the Zcash Foundation for supporting this work with a grant, and especially show our gratitude for all the valuable feedback and guidance provided from Josh Cincinnati, Ian Munoz, George Tankersley, Henry de Valence, Deirdre Connolly, Jack Grigg, and Sonya Mann.