James Lovejoy & Jonathan Harvey-Buschel
6.111 Final Project Proposal

Bitcoin Script Validation Accelerator

Abstract

Bitcoin uses the secp256k1 Koblitz elliptic curve for cryptographic operations like signature validation, and Bitcoin transactions are scripts that are written in a custom bytecode language that runs in a simple virtual machine. A majority of the machine time spent when synchronizing to the Bitcoin network is used for signature checking and script validation; our goal is to accelerate this process by developing a dedicated core for ECC operations. The accelerator will communicate with a host machine and offload ECC operations to the FPGA; we plan to use the DSP blocks to construct the multiplier, adder, and subtractor modules involved in ECC operations. Our stretch goal is to implement the full script evaluation virtual machine alongside the cryptographic accelerator.

Submodules

Our design involves three main subsystems; the cryptography library running on the host that interfaces with the FPGA, the math cores used for ECC operations, and a control unit to sequence the use of the math cores for the different ECC operations.

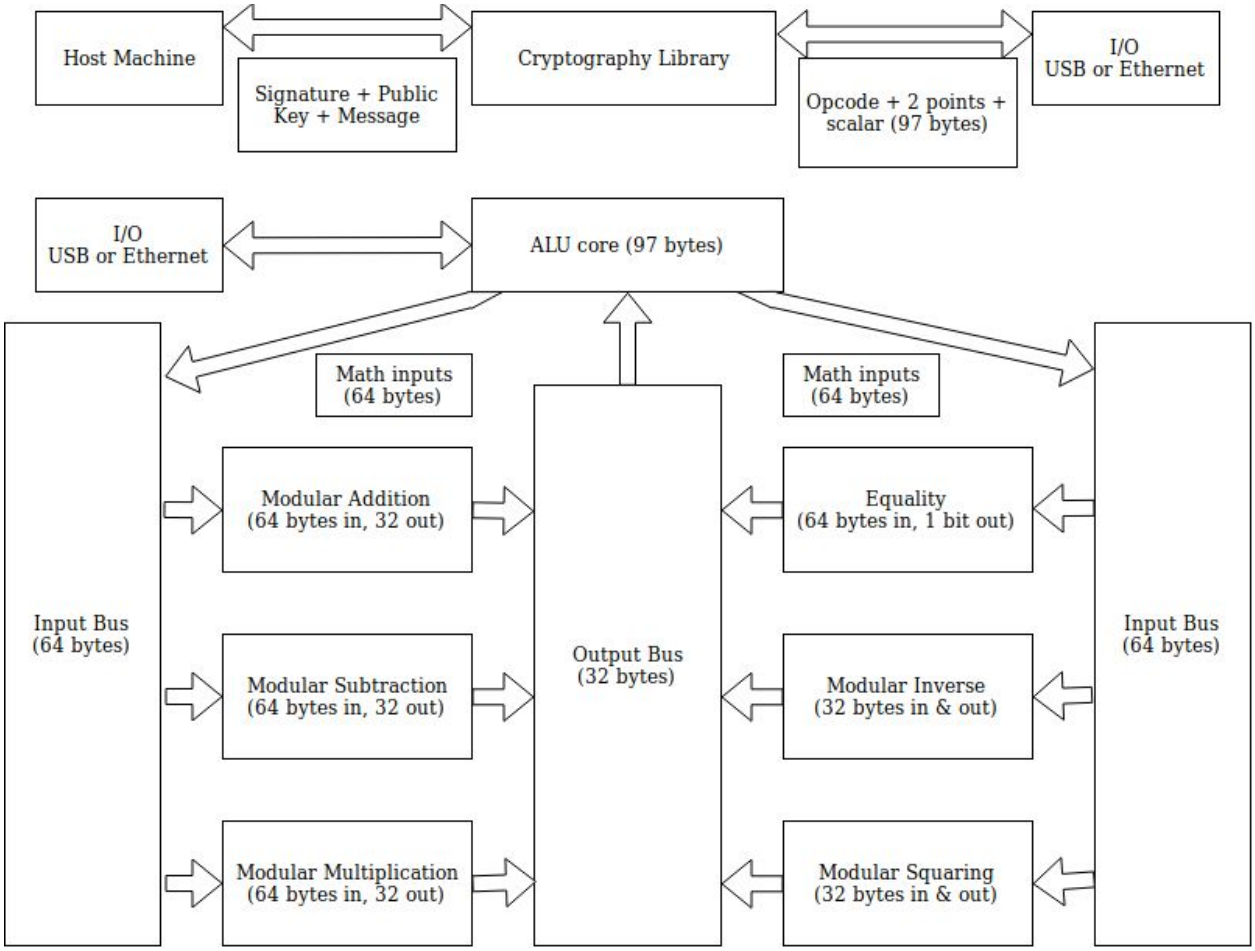
James will work on the cryptography library and testing infrastructure, and Jonathan will work on developing the math cores using the DSP blocks.

For each math core, the inputs are two 256 bit unsigned integers; the control unit takes both of these inputs as well as an opcode to signal which ECC operation should be computed over the inputs.

Metrics

The primary metric we will use to judge the success of our project is number of ECC operations per Watt. The utility of this accelerator is to allow low-performance hardware on par with single-board computers such as the Raspberry Pi to sync to the Bitcoin network at speeds only found on full desktop systems. Given this goal, making an accelerator with power consumption less than or comparable to a full desktop is ideal. To measure this we can benchmark our accelerator against boards like the Raspberry Pi running with stock settings, as well as measure power consumption at the wall with tools like the Kill-A-Watt.

System Architecture



Performance

Our strategy for implementation is to focus on designing working math modules, and then improving the algorithms used in the complex modules like multiplication, as well as develop more performant scheduling for the ALU core. The starting point is to write each math core using the naive algorithm, and without any explicit use of DSPs. The initial ALU core will pass data between math cores sequentially, and not accept new data until the final result is returned from the last math core.

Inevitably the more complicated math cores will have a higher latency than simpler cores like the equality checker. The ALU core cannot take advantage of this, but it can saturate the pipelines for each math core. For those that are implemented using DSPs, there is a minimum pipeline depth of 1 to 4 cycles depending on the operation. To do this the data and opcode need to be linked while progressing through the system, as multiple pieces of data may be in the same pipeline but attached to different opcodes.

Using the DSP blocks will allow us to save on total LUT count, which is essential given how large our data buses are. As an example, we tried to synthesize a 256-bit modulo-multiply unit,

and it used 225 DSPs and ⅓ of the LUTs available on the Nexys 4 DDR board. To help alleviate this problem we will be using the Nexys 4 Video board, and also choosing algorithms that allow use to use deeper pipelines with less DSPs. For I/O we plan to use Ethernet, since UART over USB is constrained to a few Mb/s and even a Raspberry Pi would be able to saturate that link.

Subsystems

Elliptic curve operations take place on big numbers that are 256-bits in size. Thus, we will need to design a big number ALU that is capable of handling operations on such wide integers. Additionally, each of the operations take place modulo the order of elliptic curve so each math operation also contains a modulo operation as the final step. The low level math operations are then composited by FSMs that handle the higher level EC point operations that each use multiple low-level math stages with intermediate values. Finally, the high-level EC point operations are sequenced to validate an ECDSA signature. Validation requires a SHA256 hash operation on the message being signed, so a module for that will be required as well.

Big number ALU

All input and output operands are 256-bit unsigned unless otherwise stated. $order$ is a constant that is defined by the particular elliptic curve in use.

Operation	**Pseudocode Equivalent**	**I/O sizes**
mod_add	$c = (a + b) \% order$	a, b, c: 256-bit
mod_sub	$c = (a - b) \% order$	a, b, c: 256-bit
mod_div	$c = (a / b) \% order$	a, b, c: 256-bit
mod_mul	$c = (a * b) \% order$	a, b, c: 256-bit
mod_inverse	Find x such that: $ax \% order = 1$ Uses extended euclidean algorithm	a, x: 256-bit
mod_sq	$c = a^2 \% order$	a, c: 256-bit
eq	$c = (a == b)$	a, b: 256-bit c: 1-bit

EC point ALU

point_eq

Pseudocode: $c = (eq(a_y, b_y) \;\&\&\; eq(a_x, b_x))$

I/O sizes: a_y, b_y, a_x, b_x : 256-bit
c : 1-bit

point_get_lambda

Pseudocode:

```
if point_eq(a, b):
      return mod_div(mod_mul(mod_sq(a_x), 3), mod_mul(a_y, 2))
else:
      return mod_div(mod_sub(b_x, a_x), mod_sub(b_y, a_y))
```

I/O sizes:
a_y, b_y, a_x, b_x : 256-bit
$output$: 256-bit

point_add

Pseudocode:

```
lambda = point_get_lambda(a, b)
c_x = mod_sub(mod_sub(mod_sq(lambda), a_x), b_x)
c_y = mod_sub(mod_mul(mod_sub(a_x, c_x), lambda), a_y)
return (c_x, c_y)
```

I/O sizes:
a_y, b_y, a_x, b_x : 256-bit
$output$: 512-bit

point_mul

Pseudocode:

```
res = a
for i in range(m-1):
      res = point_add(res, a)
return res
```

I/O sizes:
a: 512-bit
m: 256-bit
$output$: 512-bit

point_on_curve

Pseudocode:

```
return eq(mod_add(mod_mul(mod_sq(a_x), a_x), 7), mod_sq(a_y))
```

I/O sizes:
a: 512-bit
$output$: 1-bit

Testing and Validation

Initially we will implement the above subsystems using the OpenSSL C library's big number operations and compare the outputs to those produced by the high-level OpenSSL "EC_POINT" functions to ensure correctness of our algorithms before implementing them in Verilog. This code will also be useful for generating correct testbench inputs and outputs that can be used to write Verilog testbenches for the various low-level modules as well as end-to-end validation of the higher level EC point operations. Furthermore, the C code will serve as our reference implementation from which we gauge any performance improvement that the FPGA implementation provides. We can utilise the onboard BRAM to store testing inputs and outputs on the FPGA before dealing with computer interface using Ethernet I/O. It will be possible to use the C code to generate thousands of test cases in a few seconds making it fairly easy to get good test coverage.

Host code

In order to interface with the FPGA we plan to use the on-board Gigabit Ethernet port so that we can get sufficient I/O bandwidth to make full use of the potential speed improvement. This will require a computer program to be written that understands the communication protocol with the device, collates and feeds in instructions and data to the FPGA and retrieves the results. This will be separate from the testbenches as it will be required to interface with other user programs (ie Bitcoin Core) in order to retrieve useful validation inputs and returns the outputs to the program.