UNIVERSITY OF CALIFORNIA
SANTA CRUZ

HARDWARE ACCELERATION FOR MULTI-SCALAR MULTIPLICATION IN ZERO-KNOWLEDGE PROOFS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

BACHELOR OF SCIENCE

in

COMPUTER ENGINEERING

by

Connor C. Masterson

March 2022

The Dissertation of Connor C. Masterson
is approved:

Professor Scott Beamer, Chair

Professor Matthew Guthaus

Professor Jose Renau

Peter F. Biehl
Vice Provost and Dean of Graduate Studies

Copyright © by

Connor C. Masterson

2022

Table of Contents

List of Figures v

List of Tables vi

Abstract vii

Dedication ix

Acknowledgments x

1 Introduction 1
- 1.1 Secure Communication vs. Secure Computation . . . 1
- 1.2 Thesis Overview . . . 3

2 Background 5
- 2.1 Cryptography Basics . . . 5
- 2.2 Elliptic Curve Cryptography . . . 7
- 2.3 zk-SNARKs . . . 9
  - 2.3.1 What are zk-SNARKs . . . 9
  - 2.3.2 From Arbitrary Computation to zk-SNARK . . . 10

3 Design 14
- 3.1 Computation Overview . . . 14
- 3.2 Module Hierarchy . . . 17
  - 3.2.1 Modular Multiplicative Inverse . . . 17
  - 3.2.2 Point Addition . . . 17
  - 3.2.3 Point Multiplication . . . 18
  - 3.2.4 Point Addition Reduction . . . 20
- 3.3 Overall Architecture . . . 21

4 Evaluation and Results 24
- 4.1 Testing Environment and Methodology . . . 24

4.2 Results . . . 25

5 Related Work **29**
5.1 PipeZK . . . 29
5.2 Buntterfly . . . 30

6 Conclusion **31**
6.1 Future Work . . . 31
6.1.1 Projective Coordinates . . . 31
6.1.2 Pippenger's Algorithm . . . 32
6.1.3 Time vs. Space Tradeoff in Point Add Reduction Module . . . 32
6.1.4 Load Balancing in Top Level . . . 32
6.2 Final Remarks . . . 33

Bibliography **34**

List of Figures

2.1 Visualization of Elliptic Curve Point Addition operation. 9

3.1 MSM Computation Algorithm 15
3.2 Point Addition Algorithm 15
3.3 Block Diagram of Point Addition Module. The 'Point Calculation' block contains the arithmetic described in Figure 3.2 18
3.4 Block Diagram of Point Multiplication Module 20
3.5 Block Diagram of Point Reduction Module 21
3.6 Block Diagram of Top Level MSM Module. Thick arrows indicate busses comprised of `requestsize` many points or scalars. 22

4.1 Performance of MSM module (in terms of cycle count) with varying degrees of parallelization. 26

List of Tables

4.1 Workloads used for evaluation. 25
4.2 Execution times of Scala model. 27
4.3 Estimated runtime of our design on FPGA at 100MHz. as well as speedup over performance of our functional Scala model. 28

Abstract

Hardware Acceleration For Multi-Scalar Multiplication In Zero-Knowledge Proofs

by

Connor C. Masterson

As more and more aspects of life begin to rely on the internet, secure communication and secure computation are becoming extremely important. Zero-Knowledge Proofs are cryptographic protocols that allow one party to prove a statement to another party in a way that doesn't allow for the extrapolation of any additional information that the prover does not explicitly want the verifier to know. These protocols have many applications in blockchains, verifiable outsourcing of work, digital signatures, or proving statements about private pieces of data. These proofs, while effective, put enormous pressure on CPU's as the generation of these proofs is rather non-trivial.

One computationally demanding phase of Zero-Knowledge Proof generation is multi-scalar multiplication (MSM) on elliptic curves. In this work, we accelerate MSM with a custom hardware accelerator designed in the Chisel Hardware Description Language. This higher level HDL allows for easy parameterization so we can easily explore the design space. Our design shows 4.5-13x speedups over our functional model of Multi-Scalar Multiplication. We are also able to scale up the parallelism of the design to observe a reduction in cycle count. With custom hardware to handle this time-consuming operation, these zero-knowledge proofs can become more feasible for wide

scale adoption. Adoption of these protocols would increase privacy between parties and help foster collaboration between parties that would otherwise not trust each other.

To my mom,

who inspires me to be a better person every day.

Acknowledgments

I want to start by thanking my advisor and Chair of this thesis committee, Dr. Scott Beamer. Professor Beamer allowed me to join his research group the summer after my sophomore year and I have been working with him ever since. While I didn't have an exact idea of what I wanted to work on, I knew I wanted to work on something related to computer architecture. He allowed me to get started working on graph processing before moving on to this thesis. This work has allowed me to learn and explore so many topics outside the classroom and for that I am incredibly grateful. These experiences have given me invaluable experience in performance minded code and hardware design. I do not take these opportunities likely and am grateful to have been given them.

I am also grateful to committee members Dr. Matthew Guthaus and Dr. Jose Renau. Their time and effort spent reviewing my work is greatly appreciated.

I would also like to thank my family. To my parents and my sister, I am proud to call you my family and as much as I love being in Santa Cruz, I always look forward to coming home and seeing you all again. Facetiming you when things are difficult has always helped me get through hard times. Lastly, I would like to thank my friends and colleagues. I am grateful that I have had the privilege to have spend these last four years with you all and I look forward to seeing what we all accomplish in the years to come!

Chapter 1

Introduction

1.1 Secure Communication vs. Secure Computation

Cryptography is a field that aims to help make the things users do on computers more secure. This allows users to have greater confidence that their sensitive information will remain secure. Traditionally, cryptography has been used to ensure secure communication. However, it can also ensure secure computation. This is an important distinction to make. Secure communication is about ensuring that if Alice's message to Bob is intercepted, the attacker won't be able to decipher that message. Secure computation, is about securing the inputs to a function to allow Alice or Bob to perform work on behalf of the other without knowing the inputs. Zero-Knowledge Proofs are tools that make secure computation possible.

A classic example of a simple ZKP involves the game Where's Waldo[1]. If one

[1] This is a visual game where elaborate pictures are drawn with many many crowded characters and it is up to the person playing to find one specific character among many, that character being Waldo.

party wanted to prove that they know where Waldo is without actually revealing where Waldo was, there is a simple ZKP that can be performed. To do this, the prover could take a large piece of paper, much larger than the original picture containing Waldo, and cut a small hole in this piece of paper. Then, the original picture can be placed behind this large piece of paper such that only Waldo is visible through the small cut out. This way, anyone observing this would be able to decide that yes indeed the prover does know where on the page Waldo is. But, if given the original picture again, they would have to find Waldo themselves as the proof they observed provided no context or information other than that which was explicitly stated by the prover. While real world ZKP are much more complicated, the idea is the same. A prover wants to prove a statement without revealing any extra information to the verifier.

ZKP can help to foster more collaboration between parties that may have otherwise not been able to work together. For example, say a machine learning startup wants to secure some funding by proving their framework is the best there is. But, they don't want to reveal how it works because that is their trade secret. ZKP would allow this startup to instead construct a proof that states the benefits of their product without actually revealing how their product works. This way anyone who reads the proof will know that their product really is the best but have no idea how it works. Anyone reading the proof would be unable to learn anything about the framework that the startup didn't explicitly want them to know in the first place.

ZKP has the ability to be implemented in many privacy concerned applications such as secure outsourcing of work (Verifiable Computation), blockchain verification,

anonymous credentials, and more. However, because the construction of these proofs is computationally intensive, they have not yet seen wide scale adoption. We propose an accelerator to speed up one important phase in the generation of these proofs: Multi-Scalar Multiplication on Elliptic Curves.

1.2 Thesis Overview

Chapter 2 of this work provides a brief overview of modern cryptography, its primitives, and its uses. We also explain how Elliptic Curves act as cryptographic primitives. Finally, this chapter will cover what zk-SNARKs are and how they are constructed at a very high level.

Chapter 3 of this work details the hardware design which this work is mainly about. It covers a summary of the targeted workload and its characteristics as well as what design choices were made and why. This accelerator takes in two equal length vectors: one of elliptic curve points and one of scalars. The MSM module then computes a dot product of these two vectors resulting in one final elliptic curve point. This code is open source and can be found at github.com/ucsc-vama/multi-scalar-multiplication.

In Chapter 4, we evaluate the design and present the results of our testing. This includes an explanation of the evaluation methodology as well as the results themselves presented within the context of the current state of the field.

Related work is presented in Chapter 5. This includes discussion about collaborations at UC Santa Cruz as well as similar outside work targeting the same workloads.

Finally, in Chapter 6, we conclude the work. This includes discussion of future work and conclusions to be drawn as a result of this work.

Chapter 2

Background

This section aims to provide a background on the information necessary for understanding Multi-Scalar Multiplication to appreciate the design choices we make. We present the actual design in later chapters.

2.1 Cryptography Basics

Cryptography today relies on "one-way" or "trapdoor" functions. These are problems that are simple to verify, but extremely difficult to undo. A classic example is the idea that it is very easy to calculate $ab = x \pmod{p}$, but extremely difficult to calculate b if only a, x, and p are given and a, x, and p are all very large. This is referred to as the Discrete Logarithm Problem or DLP for short.

When the numbers a, x, and p are very large, there are a vast number of possibilities for what b could be. The larger these numbers are, the more secure an application or protocol is said to be. The only way to try and solve for b is to try many

combinations over and over which is intractable. However, if we are not careful, we may not get the full benefit (that benefit being security) of a large finite set. It is important that p must be prime. If p is not prime, we may end up in a cyclic subgroup of the larger finite group. These finite groups often have what is referred to as a *generator* which is an element of the group. From the generator, all other elements can be generated through repeated applications of the $\bullet$ operation. If we don't start with the generator and instead start with an arbitrary point, we may end up not being able to generate all other elements. When this happens the security is significantly decreased because the size of our set of elements has decreased. We are often able to avoid this possibility by ensuring that p is prime. Because of this, it is always recommended to use specific elliptic curves or specific algorithms that have been verified by cryptographers.

Note above the use of modular arithmetic. We actually cannot encrypt (or "construct a hiding") for a value if we allow ourselves access to all possible integers. We must constrain computation to some kind of finite group. The larger this group is, the better; but it cannot be infinite. These groups have a very important property: There exists an operation denoted $\bullet$ between two elements of this finite group for which the result is another element of the group. This property is referred to as closure. This property is exhibited by modular arithmetic as well as elliptic curves defined over finite fields, which will be discussed in the following subsection.

2.2 Elliptic Curve Cryptography

Elliptic Curves also have a DLP called the ECDLP which will be discussed shortly. Before discussing the ECDLP, we need to understand what these curves are under the hood. These elliptic curves take the form $y^2 = x^3 + ax + b \pmod{p}$ in which a and b are constant coefficients that define the curve which exists in a finite field of prime order[2]. A finite field F_p is a square p x p matrix of coordinates. We say a curve is "defined over" a certain finite field if the coefficients a and b that define that curve are within the field. In practice, p will always be a very large (128-768 bit) prime number. As an example we can still understand what is going on with a smaller prime number. To ensure a point is actually on a curve and within a field, we can simply try computing $y^2 - x^3 - ax - b \pmod{p} == 0$. If the inequality does not hold, we know that the point is not on the curve.

These curves can be thought of as the collection of the points made up of integers on the specified curve over F_p. We say an elliptic curve exists over a finite field when its coefficients are elements of that field. It is this collection of points that make up our finite group of elements. Since we are in a finite field, operations on these points must satisfy three properties: closure, identity, and associativity. In this case, anytime we add points together on this curve, or multiply them by a scalar, we end up with another point on the elliptic curve, i.e. we always end up with another element of our finite group of points.

We'll start with a small example curve defined by $a = 0, b = 7, p = 17$. To

ensure that our operation $\bullet$ on a point on our elliptic curve always results in another point on our elliptic curve, we need to rethink how we add points together. To add points on an elliptic curve, we can't simply do something like the following. Given:

$$P_1 = (12, 1) \quad P_2 = (3, 0) \quad \Rightarrow \quad P_1 + P_2 = (15, 1)$$

This resulting point was calculated by simply adding the x and y coordinates respectively. This resulting point is not on the elliptic curve because $12 - 153 - 7$ (mod 17)! $=$ 0. We must define Point Addition so that our result is also a point on the same curve. Elliptic curves have a property that states that the addition of points on a line together are equal to zero. In this case, zero is represented by a special "point at infinity" which we denote as $\mathcal{O}$. (Later sections will address how this is handled in hardware.) This property ensures that if we add three points on a straight line $P_1 + P_2 + P_3 = \mathcal{O}$. We can rearrange this equation to get $P_1 + P_2 = -P_3$. We now have a way to add points on our curve. While this equation looks simple, it is important to remember what this equation actually means. It means we can find a straight line that intersects two points on our curve (P_1 and P_2, the points we want to add together) and we follow that line until it intersects a third point on our curve. We then negate that point. This can be done by negating just the y coordinate. Figure 2.1 visually shows what this operation looks like. Once addition is defined, scalar multiplication can also be defined as repeated additions. If we need to multiply a scalar S and a point P, we can perform $P + P + P \ldots + P$.

Adding a point to itself is one corner case to we need to address. In this case, instead of drawing a line that connects our two operand points, we follow the tangent line from the one given point until it reaches the curve and then negate the point to find our resulting point.

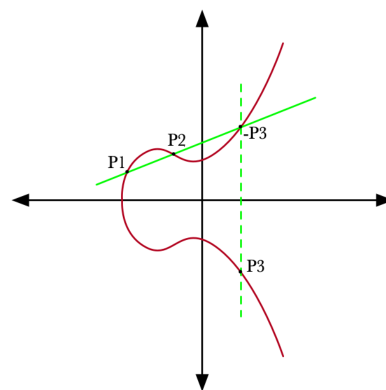

Figure 2.1: Visualization of Elliptic Curve Point Addition operation.

These operations allow us to form groups of points. These groups are the basis of elliptic curve cryptography systems. With these tools, we can now utilize the Elliptic Curve Discrete Logarithm Problem. If provided with a point on a curve and a number of Point Additions to perform, it is easy to figure out the end-point. However, if provided only the start and end points, finding the number of Point Additions required to get to the end point is intractable when the number of Point Additions is sufficiently large. Thus, this number of Point Additions is the number that gets encrypted.

2.3 zk-SNARKs

2.3.1 What are zk-SNARKs

zk-SNARK stands for Zero Knowledge Succinct Non-Interactive Argument of Knowledge. This is a rather large acronym, so we will tackle it one part at a time. The Zero Knowledge part refers to the fact that the verifier of the proof will not be able to

extrapolate any information about the proof that the prover did not explicitly want the verifier to know. The Succinct part means that the proof is reasonably small and can be verified quickly. The Non-Interactive part means that the prover and verifier do not need to communicate back and forth, so the prover can simply construct the proof and the verifier will accept it if they determine it was properly constructed. The last part is the Argument of Knowledge and this refers to the actual information that the prover is arguing that they know to be true.

In zk-SNARK systems there is always a prover who is trying to prove a statement about a piece of data that that prover does not want to leak information about as well as a verifier who can easily verify the validity of the proof. These proofs satisfy three properties[5]. The first and most important is *soundness*. This means that the verifier can, with a very high degree of certainty, determine when a proof was constructed in good faith. The second property is *completeness*. This means that the proof is sufficient (it is complete) to convince the verifier. The last property is *zero-knowledge*. This is the idea that the prover can prove their statement about a secret without actually revealing any kind of information about the data. When the verifier verifies the proof, they do not learn anything about the secret that the prover does not want them to know.

2.3.2 From Arbitrary Computation to zk-SNARK

To preface, this background will not cover everything necessary to understand zk-SNARKS from the ground up. The full pipeline from "statement to prove" to zk-SNARK is incredibly complicated and this it is not the aim of this work to provide

insight on all the necessary math that makes this possible. This section will do a high level summary and focus on the part that is relevant to the targeted workload. Consider reading references for more detailed information on the incredible math that makes these proofs possible[5][9][4][3].

zk-SNARKs begin with a statement about a computation. This statement as well as the computation can be arbitrarily complex. The more complex the statement, the longer proof generation will be. The first step of proof generation is to take a function F that the prover wishes to encode into an Algebraic Circuit. This is essentially a directed acyclic graph where the nodes represent multiplications in the computation from F. The next step is to construct what is called a Rank 1 Constraint System (R1CS). This takes our graph based representation of the function F and turns it into a series of three vectors. These vectors are often called L, R, and O. These stand for left, right, and output and they represent the left and right inputs to our gates in our Algebraic Circuit. The output vector represents the outputs of our gates. Now it may be easier to see that allowing arbitrary complexity of our original function F can allow these vectors to become extremely large.

The next step is to convert this R1CS into a Quadratic Arithmetic Program (QAP). This essentially means we will again transform our representation of the original function F. This time from vectors into polynomials. This set of polynomials is the QAP that now represents our original function F. While this may seem needlessly complex, polynomials have predictable forms. A degree d polynomial is of the form $c_d x + \ldots + c_1 x + c_0 = y$. The only thing that changes from one polynomial to another is the

coefficients $c_0 ... c_d$. Thus we can say to 'know' a polynomial P is to know its coefficients $c_0 ... c_d$. This series of coefficients (scalars) will be how the presented hardware module sees these polynomials.

There are two main computations that the prover must perform. The first involves calculating the coefficients of a degree p polynomial. This is done through a series of Number Theoretic Transforms (NTT) and Inverse Number Theoretic Transforms (INTT)[6][8]. These are essentially Fast Fourier Transforms but instead over a finite field. This is an important part of the workload but is not addressed by this work[1]. The second part of the prover's computation is the Multi-Scalar Multiplication (MSM) that this work is focused on. This involves as input two vectors of the same length. One vector is made up of scalars which represent the coefficients of a polynomial. The other is a series of points on a predetermined elliptic curve. The prover then must multiply each point P_i by the corresponding scalar scalar S_i. This computation is more properly articulated with:

$$\mathbf{Q} = \sum_{i=1}^{n} P_i S_i$$

Where P_i is point on the elliptic curve and S_i is a coefficient of a polynomial. The results of these multiplications are then added together to produce one final point, referred to above by $\mathbf{Q}$.

In this work, we present a design for a hardware accelerator for Multi-Scalar

[1]Another project also aimed at accelerating ZKP at UC Santa Cruz addresses this part of the computation. This work is being undertaken by Jason Vranek and is called Buntterfly.

Multiplication between Elliptic Curve points and scalars, a major component of Zero-Knowledge Proof generation. ZKP construction specifically requires many EC point multiplications on the coefficients of large polynomials. While this work was made with ZKP in mind, any workload that needs to perform many EC point multiplications or additions could also utilize our design and modules.

Chapter 3

Design

This section will explain the overall architecture of the accelerator. We begin with an overview of the computation. Then, we introduce the smallest module and slowly build up the hierarchy until we have an understanding of the full architecture from the bottom up.

3.1 Computation Overview

As explained in the last section, the MSM computation to accelerate is given by the equation:

$$\mathbf{Q} = \sum_{i=1}^{n} P_i S_i$$

where **Q** is a one final point on the given elliptic curve. Pseudo-code for this can be seen in Figure 3.1. One thing to note about this is that if this were written in code, the loop could be considered *embarrassingly parallel* if each iteration did not write to a shared variable for a reduction. However, the work performed in each iteration is indeed

MSM-Computation(S, P)

Q $\leftarrow$ 0

for s, p in $\vec{S}, \vec{P}$ **do**

$Q \leftarrow$ Q + s $\bullet$ p

Figure 3.1: MSM Computation Algorithm

Point-Addition(P1, P2)

$\lambda \leftarrow (P_{2,y} - P_{1,y})/(P_{2,x} - P_{1,x})$

$R_x \leftarrow \lambda^2 - P_{1,x} - P_{2,x}$

$R_y \leftarrow \lambda(P_{1,x} - R_x) - P_{1,y}$

return Point(R_x, R_y)

Figure 3.2: Point Addition Algorithm

independent of all other operations so we can still parallelize this computation if we are careful about how we sum up our Point Multiplications. There is previous work that has dealt with accelerating a single Point Multiplications[1], but we must handle many. By using Chisel to construct hardware generators, we can parameterize the degree to which we parallelize this computation in order to trade off space (resource usage) and performance.

It is also important to remember that $\bullet$ in Figure 3.1 represents scalar point multiplication and not traditional scalar multiplication. We defined this scalar point multiplication earlier as repeated point additions. Point addition involves finding a straight line between two points (and a tangent line if adding a point to itself,) following

that line until intersecting a third point, and then negating that third point.

$$P_1 + P_2 + P_3 = \mathcal{O} \implies P_1 + P_2 = -P_3$$

This can be computed with the algorithm seen in Figure 3.2 and is the heart of the computation. In practice, to avoid doing division, lambda is often calculated instead with:

$$\lambda = (3P1_x^2 + a) \times \ modinv(2P1_y, p)$$

where a and p are constants from the definition of the elliptic curve. It is this formula containing modular inverse that we implement in our design. To construct a Point Multiplication module, we must first began with a modular inverse module. This can then be used to construct a Point Addition module, which is then used to construct a Point Multiplication module.

There a number of benefits that our architecture has. Most importantly it is flexible. The number of EC Point Multipliers can be scaled up and down to perform more or less Point Multiplications at a time. Whether mapping to an FPGA or implementing as an ASIC, we will always have the option to change how large the data path is. CPU's have to perform these operations as a series of many instructions on only a small handful of cores. This architecture could potentially have a very wide data path if implemented in a system with the memory bandwidth to support it.

3.2 Module Hierarchy

3.2.1 Modular Multiplicative Inverse

The Modular Multiplicative Inverse module is the first and smallest module in the design. It takes as input two signed integers (a, p) and a load signal. It outputs a single signed integer which is the modular multiplicative inverse of a mod p and a valid signal indicating when the result is complete. It starts with a value n equal to zero. This value is increased by one each clock cycle until the condition $a \times n \pmod{p} == 1$ is true, at which point the valid signal is asserted and n is sent to the output. If we arrive at a state in which $n = p$ then the modular multiplicative inverse does not exist.

3.2.2 Point Addition

The Point Addition module involves the most arithmetic. This is where we leverage our modular multiplicative inverse module and add two points together. This module takes as input (x, y) coordinates for two points, constants a, p from the definition of the elliptic curve, and a load signal. It should be noted that a and p are not stored in registers. This is because they are constant throughout the entire course of the computation and will not change. In any given zk-SNARK protocol, the elliptic curve is chosen ahead of time by the programmer and does not change.

When the load signal is asserted, the coordinates for both points are latched into registers. These values, along with a and p are then used to calculate λ and the coordinates of the result point. When the instantiated modular multiplicative inverse

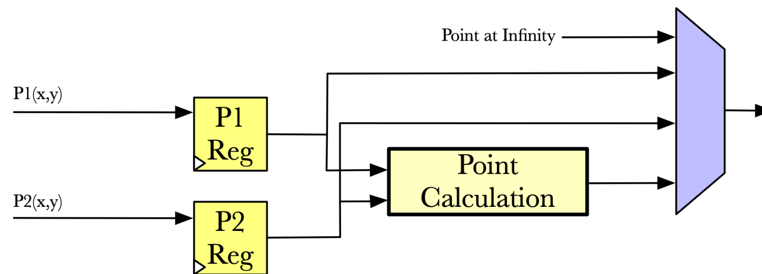

Figure 3.3: Block Diagram of Point Addition Module. The 'Point Calculation' block contains the arithmetic described in Figure 3.2

module asserts its valid signal, the module can use that result to compute its result and assert its valid signal on the next cycle.

There are two special cases to check for here that allow this module to complete within one cycle. If one of the input points is $\mathcal{O}$ (point at infinity), then the output is set to the other point. If the two input points are inverses of each other, the output is set to $\mathcal{O}$. A schematic of this module can be see in Figure 3.3.

This computation is rather dense and could potentially be pipelined in the future. However, many of the issues that stem from this module may mitigated by switching to Projective Coordinates, and we discuss this in further detail in Chapter 6.

3.2.3 Point Multiplication

At this level of the hierarchy, we begin to concern ourselves more with control flow than raw computation. This module takes as input a signed scalar s, the (x, y) coordinates of a point, a and p constants, and a load signal. This functional unit instantiates one Point Addition Module and passes its a and p inputs. From a high

level, this module adds the given point to itself s times by way of its instantiated Point Addition (PAdd) Module.

This module also has a few registers to keep track of intermediate values over the course of the computation as well as registers in which to latch its inputs when the load signal is asserted. There are two registers which hold the result of each intermediate addition which are enabled by the valid signal of the PAdd module, so, each time the PAdd module finishes an addition it saves the result. The overall output of the Point Multiplication module comes from these registers.

One of the point inputs (P1) to the Point Addition Module is always fixed since we are adding one point onto a running total. However, the second input (P2) is one that changes on each iteration. If it is the very first addition, the input is the given (x, y) pair itself. For all the additions after, the inputs for P2 are the outputs from the previous iteration. We are able to avoid a combinational loop here because the Point Addition Module latches the intermediate values into registers.

Since the input scalar s is latched into a register, it is also utilized as the counter which determines when the computation is complete. That value is decremented until it is equal to one. The valid signal can be asserted once this counter register equals one and the valid signal from the PAdd Module goes high again.

This Point Multiplication (PMult) module is controlled by a simple state machine composed of three states: `idle`, `specialcases`, and `working`. The module starts in the `idle` state and remains there until the load signal goes high. When this happens, if we are dealing with a special case, the module moves to the `specialcases` state and

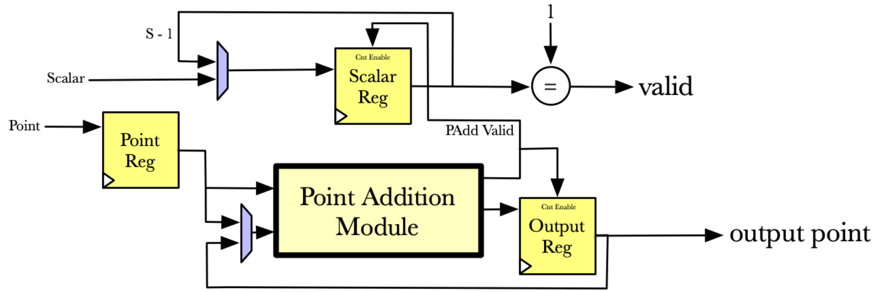

Figure 3.4: Block Diagram of Point Multiplication Module

handles that case accordingly. If we are not dealing with one the these cases, the modules moves to the `working` state until its computation is complete and it transfers back to the `idle` state.

3.2.4 Point Addition Reduction

The Point Addition Reduction module takes as input a vector of EC Points. In Chisel, this must be implemented with the `Vec` construct to support hardware indexing by the control logic. These Points are then summed up over time. Similar to the PMult module this is accomplished with a pair of registers to hold the x and y coordinates of an intermediate sum. This sum is connected directly to the output and is deemed valid when the count is equal to the number of points minus one and the valid signal from the PAdd module goes high again. The additions are performed by a single Point Addition Module. While using a single PAdd module can be slower, it also ensures good utilization of that module. If we were to have $log_2 n$ PAdd modules, the addition would be faster at the cost of lower utilization. A block diagram of this can be seen in

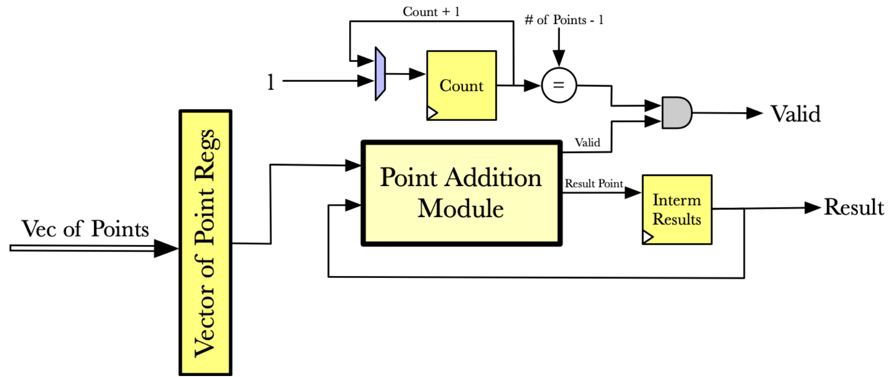

Figure 3.5: Block Diagram of Point Reduction Module

Figure 3.6.

This module is similar to the Point Multiplication module in that it instantiates one Point Addition Module and performs repeated addition. This module differs in that it must also contain many registers to hold all the points that will be used as operands over the course of the computation.

This Point Addition Reduction (PReduc) module is also controlled by a simple state machine, this time with just two states: `idle` and `working`. The module remains in the `idle` state by default and moves to the `working` state when the load signal goes high and moves back to the `idle` state when the computation is complete.

3.3 Overall Architecture

The top level module takes in one parameter, which is the length of the point and scalar vector inputs. Given that the total size of the Point and Scalar vectors in a zk-SNARK proof construction can be extremely large, this parameter will be much

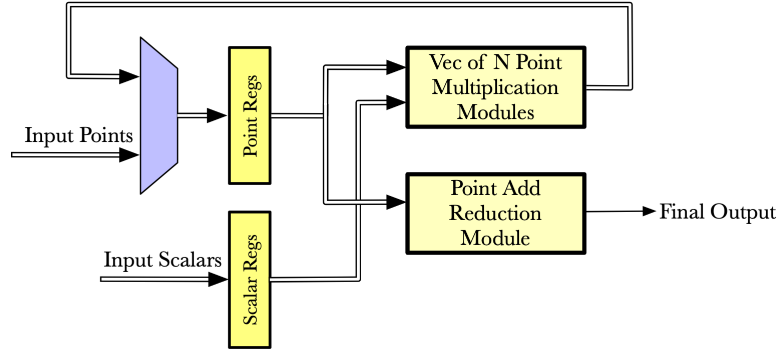

Figure 3.6: Block Diagram of Top Level MSM Module. Thick arrows indicate busses comprised of `requestsize` many points or scalars.

smaller. This parameter is called `requestsize` and can be tuned at compile time. As input, the module also takes a load signal and three vectors, all of size `requestsize`. These represent the x coordinates, the y coordinates, and scalars to be used in the computation. As output, there is a valid signal and a point comprised of x and y coordinates. A block diagram for this module can be seen in Figure 3.6.

The parameter `requestsize` is also the number of Point of Multiplication Modules. Each scalar S_i and point P_i is given as input to one of the PMult modules. Once all the point multiplications are complete, they are given to a PReduc module. This module, as explained in the last section, sums up a list of given points. In this case, we are summing up the results of all the Point Multiplications. The valid signal from the reduction module serves as the valid signal for the overall design. Once this reduction is complete, the set of multiplications is complete.

It is expected that the number of multiplications in any given ZKP proof construction is extremely large. It would be impractical to have one million PMult modules each perform one operation. This would make the module too large and leave

too much hardware underutilized. It is expected that data will be predictably streamed into this module `requestsize` many elements at a time by the CPU. Our design does not aim to take in and compute the entire sequence of multiplications all at the same time. Instead, after each set of `requestsize` multiplications is complete, the module will assert its valid signal. The host system can then store this output and stream the next `requestsize` many points and scalars into our design. It is up to the host system to store each intermediate result point and accumulate the sum on the CPU as the MSM module continues to work on the next set of multiplications.

Chapter 4

Evaluation and Results

This chapter explains the testing environment, evaluation methodology, and motivation behind the chosen tests. It also discusses the subsequent results.

4.1 Testing Environment and Methodology

Our tests will focus on the performance of the MSM Module relative to itself as well as a CPU baseline as we tweak parameters in our Chisel generator. We perform experiments on a dual socket Intel Xeon Platinum 8260 Cascade Lake CPU's at a 2.4GHz base clock. The system also has 768 GB of DDR4 DRAM. The host system is running Ubuntu 20.04 LTS using Linux Kernel 5.4. The Chisel RTL was written and compiled with Chisel 3.4.3. Our design was verified by first creating a functional model of the behavior in pure Scala. We implement a small Elliptic Curve (EC) library and then create functions that perform EC scalar point multiplication. This allows us to easily integrate our Chisel tests with our functional model to perform tests.

Table 4.1: Workloads used for evaluation.

Workload	Curve Parameters (a, b, p)	Number of Points and Scalars
1	0, 7, 17	12,000
2	0, 7, 17	12,000

4.2 Results

Our evaluation computes a series of Multi-Scalar Multiplications on different variations of the accelerator with different degrees of parallelization. That is, with a different number of Point Multiplication modules instantiated in the Top Level module. Our initial hypothesis is that when increasing parallelization, performance will increase. We should be able to observe a decrease in cycle count as the module is able to take in more data at a time. Writing Chisel generators instead of static RTL allows us to quickly generate variations on our design and explore the design space.

We measure the performance of our accelerator by tracking the number of clock cycles it takes to compute a series of multiplications. We only consider the cycles in which the multi-scalar multiplication is being computed. Future applications of this design must also consider data loading and unloading times. For this work, we assume ideal data flow with predictable DRAM access times.

The workloads used for evaluation can be seen in Table 4.1. These workloads consist of two equal length vectors of EC points and scalars. Both workloads are the same length and utilize the same curve but are comprised of different points and scalars. These vectors are streamed into an MSM module `requestsize` elements at a time. The curves in these workloads are defined by small parameters whereas the curves used in

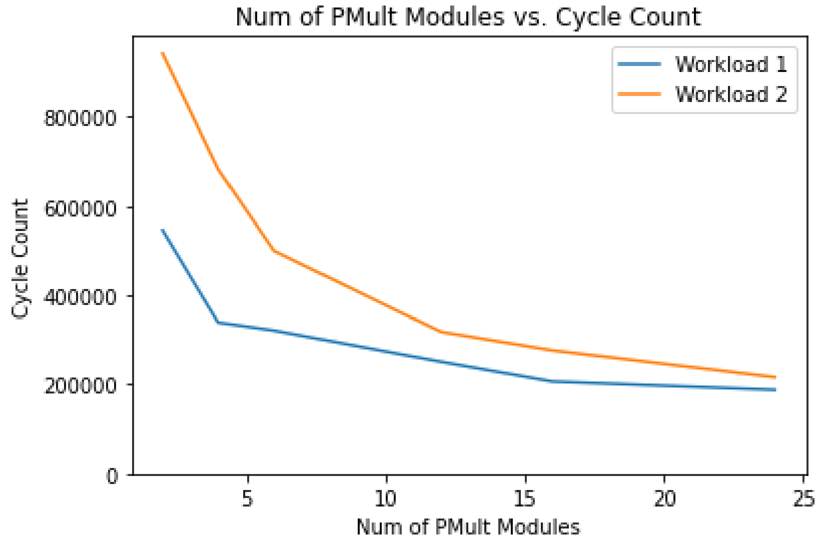

Figure 4.1: Performance of MSM module (in terms of cycle count) with varying degrees of parallelization.

real cryptographic protocols have 128, 256, or even as high as 700+ bit numbers defining them. The reason for this is that our PAdd module takes in affine coordinates which need a modular inverse to calculate resultant points. This modular inverse is calculated by continually computing $a \times n \pmod{p} == 1$ where n starts at 0 and is increased until the inequality is true. This process can take a tremendous number of cycles to complete when p is large. Normally, this can be solved by using projective coordinates, but our design does not currently support them. To ensure that our cycle count measurements are not dominated by time spent solving for modular inverses, we restrain ourselves to small curves.

We measure performance on the same workloads on different instances of our MSM modules, each with an increasing number of Point Multiplication modules. As can be see in Figure 4.1, performance with just two Point Multiplication modules is

Table 4.2: Execution times of Scala model.

Workload	Number of Points/Scalars	Exec. Time Avg. (Sec.)
1	12,000	0.029
2	12,000	0.038

not very good as it makes barely any attempt to exploit any of the parallelism in the given workload. As the number of Point Multiplication modules increases, there is a clear gain in performance as cycle times decrease. However, these performance gains do level off as we run into the effects of Amdahl's law and the non-parallelizable portion of the workload begins to comprise a larger and larger percentage of the overall workload. Even as the number of PMult modules increases the designs continue to instantiate just one Point Reduction module. This means that as there is more data to reduce on each iteration. The variations of the design are decreasing the time spent performing point multiplications, but increasing the time spent performing point reductions. This can be seen in the non-linear scaling in Figure 4.1 as the number of PMult modules increases.

Cycle count improvements level off at around 16 multipliers. This seems to indicate that this is the limit imposed by our Point Reduction module. The Point Reduction module could be altered to work in $\log_2 N$ time by using $\log_2 N$ Point Addition modules but that remains in the future work category for now. This optimization would lower this lower bound on cycle count as it would make the reduction a smaller fraction of the overall workload.

We also compare performance of our design with our functional Scala model[1].

[1]We recognize that Scala is a managed language with a garbage collector and a virtual machine. This means that there is variability in performance. To get more accurate numbers when benchmarking our CPU's performance, we run tests many times and take an average on the performance.

Table 4.3: Estimated runtime of our design on FPGA at 100MHz. as well as speedup over performance of our functional Scala model.

Workload	Speedup over Scala model (2 - 24 Multipliers)
1	5.3x - 15.4x
2	4.02x - 17.5x

We use timers inserted into our Scala model to measure CPU performance. These numbers are presented in Table 4.3. We assume our design will run on an FPGA at 100MHz. We use this number to calculate an estimated run time of our design with the following formula:

$$Exec.\ Time\ (Sec.) = Cycle\ Count\ /\ (1 \times 10^8\ Cyc./Sec.)$$

Taking the results for cycle count seen in Figure 4.1, when compared to the execution time of our Scala model, our design ranges from 4.02x faster with just 2 multipliers and 17.5x faster when using 24 multipliers. These numbers show that our design has a clear benefit when it comes to cycles taken to complete an MSM computation. The wide data path that our design allows more than makes up for the slower clock frequency of an FPGA. Even in the smallest variation, perhaps more suited to a smaller FPGA, we are still able to provide a speedup over the CPU implementation. If we were to instead implement this design as an ASIC, we believe performance could be increased by an order of magnitude from the increase to clock frequency alone. A 1GHz clock rate is a reasonable assumption for an ASIC. Given that this application is not memory bound, we believe that our ASIC would be well utilized and see full benefit of an increased clock frequency.

Chapter 5

Related Work

This section covers similar work that has taken place in the Verifiable Computing hardware design space. This includes other work from other people at UC Santa Cruz and Microsoft.

5.1 PipeZK

PipeZk[10] is a collaborative work from Microsoft Research, Tsingua University, and more. This work presents a hardware accelerator for zero-knowledge proof generation. PipeZK is a full end to end system that handles both MSM and NTT (Number Theoretic Transform) and heavily influenced this work.

PipeZK makes a number of different design choices for its MSM acceleration. PipeZK only has a handful of Point Addition modules in the entire design, but shares them all system wide. This comes with a fair bit of extra control logic but it means that at any time, whether an addition needs to be performed for a reduction or multiplication,

it is sent to one of a few centralized FIFO's where it can get dispatched to any adder. This also means that these additions need to be tagged so the control logic knows where to send the result. This approach certainly helps to solve the resource utilization problem.

PipeZK also converts to projective coordinates before performing its MSM multiplications to avoid calculating a modular inverse. This allows the point addition modules to have a more predictable latency. This helps to ensure that the FIFO's which feed into the shared adders don't overflow.

PipeZK shows reasonable improvements over CPU/GPU software implementations. It is, to the best of our knowledge, the first end to end accelerator for ZKP proof generation.

5.2 Buntterfly

Buntterfly is another ZKP hardware accelerator under developement at UC Santa Cruz by Jason Vranek. This project is attempting to handle the NTT portion of the ZKP proof construction workload. In the future, we hope to build an interface between this work and his to create a fully functioning open source ZKP accelerator.

Chapter 6

Conclusion

6.1 Future Work

6.1.1 Projective Coordinates

As has been mentioned a few times before in this work, switching to projective coordinates would be helpful for constraining cycle count of point additions[7][10]. Projective coordinates represent what normally takes two values with affine coordinates (x, y) and represents them with three coordinates (X, Y, Z). These coordinates are derived through $x \rightarrow X/Z \quad y \rightarrow Y/Z$. These coordinates remove the need for modular inverse and instead introduces the extra work on converting back to affine coordinates at the end of the multiplication.

6.1.2 Pippenger's Algorithm

Pippenger's Multi-Product[3] algorithms are efficient ways to perform many multiplications at once. For this use case, it would involve instantiating many smaller multipliers instead of larger ones. This involves radix sorting the λ-bit scalars into buckets, each $\lambda/N = B$ bits wide[10]. This results in 2^B buckets. For each bucket numbered 0 to 2^B, the points are sorted into the buckets whose number matches their scalars λ bits. This converts the multiplications into a set of smaller additions. These additions then have to be weighted by $2^{N\times B}$. These smaller additions allow more of the multiplication to be done in parallel as the computation is broken down into smaller independent parts. This is also implemented in PipeZK.

6.1.3 Time vs. Space Tradeoff in Point Add Reduction Module

The current Point Addition Reduction module performs one addition at a time with one instantiated PAdd module. If given N input points, it will take N-1 point additions to complete. This module could instantiate $\log_2 N$ PAdd modules and complete the reduction with Log_N point additions. This module could also instantiate any number of PAdd modules in between. This would be a worthwhile design space exploration to find a good balance between performance and resource usage.

6.1.4 Load Balancing in Top Level

The Top Level module has two main stages of work: multiplying, reducing. The current design suffers from poor resource utilization. When the multipliers are working,

the point reduction module is sitting idle and vice versa. With some more control logic, these could become two pipeline stages to increase utilization. The PipeZK model could also be adopted which instead decouples the two stages and uses FIFO's to handle work between the two main stages of computation.

6.2 Final Remarks

This work presents a novel design for a Multi-Scalar Multiplication accelerator designed in Chisel. This design is parameterizable and allows for the number of multiplications performed at a time to be increased or decreased. This allows for easy tradeoffs to be made between resource usage and performance. While this design is smaller then some others, it had the benefit of being written in Chisel to it can be improved quickly. In the future, we hope to integrate it with Buntterfly as well as other ZKP related acceleration functional units. Additionally, it is also open source and can be found at github.com/ucsc-vama/multi-scalar-multiplication. Being open source, this design can be improved by anyone at any time. It will benefit from the architecture community and make zero-knowledge proof systems much faster in the future.

Bibliography

[1] Hamad Alrimeih and Daler Rakhmatov. Fast and flexible hardware support for ecc over multiple standard prime fields. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2661–2674, 2014.

[2] Mohammad Anagreh, Eero Vainikko, and Peeter Laud. Accelerate performance for elliptic curve scalar multiplication based on naf by parallel computing. In *ICISSP*, 2019.

[3] Ryan Henry, David R. Cheriton, and Northeastern Ontario. Pippenger's multiproduct and multiexponentiation algorithms (extended version). 2010. [Online] Available: http://www.cacr.math.uwaterloo.ca/techreports/2010/cacr2010-26.pdf.

[4] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2013. Best Paper Award.

[5] Maksym Petkus. Why and how zk-snark works. *CoRR*, abs/1906.07221, 2019.

[6] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: high-performance

architecture for computation on homomorphically encrypted data in the cloud. *CoRR*, abs/1909.09731, 2019.

[7] Nagaraja Shylashree and V. Sridhar. Hardware realization of fast multi-scalar elliptic curve point multiplication by reducing the hamming weights over gf(p). *International Journal of Computer Network and Information Security*, 6:57–63, 09 2014.

[8] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398, 2019.

[9] Hartwig und Paola Mayer. zk-snark explained: Basic principles. 2017. [Online] Available: https://blog.coinfabrik.com/zk-snarks-explained-basic-principles/.

[10] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *48th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, June 2021.