

# Introduction

The efficiency of an implementation of an algorithm is dependent on the architecture and the instruction set used. Serial (sequential) instructions, and vector (parallel) instructions can be used depending on the task that needs to be done. Our presentation is about the work that we have been doing with a multiplication algorithm (namely schoolbook multiplication), and how we have been working to implement it in the z/Architecture with vector instructions using a DSL called COCONUT written in Haskell.

## Objectives

A major objective of the project was to create algorithmic implementations in Z. We were also tasked with testing numerous algorithms that were previously implemented in COCONUT. Our goal was to find bugs and ensure code reliability, in both our own and others' work. Since the speed of cryptography depends heavily on the bottleneck of multiplication, our work focused on Z-implementations of it. However, for simplicity, our poster mentions mainly the algorithmic ideas behind our work, along with some C code.

## Subjects and Methods

## Case Study: Schoolbook Multiplication

Your traditional algorithm used at school.

$\begin{array}{c} (a_3a_2a_1a_0) \\ \times (b_3b_2b_1b_0) \end{array}$							
			$a_3b_0$	$a_2b_0$	$a_1b_0$	$a_0b_0$	
			$a_3b_1$	$a_2b_1$	$a_1b_1$	$a_0b_1$	
	$a_3b_2$	$a_2b_2$	$a_1b_2$	$a_0b_2$			
$a_3b_3$	$a_2b_3$	$a_1b_3$	$a_0b_3$				
				$a_3b_0l_0$	$a_2b_0l_0$	$a_1b_0l_0$	$a_0b_0l_0$
			$a_3b_0h_0$	$a_2b_0h_0$	$a_1b_0h_0$	$a_0b_0h_0$	
			$a_3b_1l_1$	$a_2b_1l_1$	$a_1b_1l_1$	$a_0b_1l_1$	
	$a_3b_1h_1$	$a_2b_1h_1$	$a_1b_1h_1$	$a_0b_1h_1$			
	$a_3b_2l_2$	$a_2b_2l_2$	$a_1b_2l_2$	$a_0b_2l_2$			
$a_3b_3h_2$	$a_2b_3h_2$	$a_1b_3h_2$	$a_0b_3h_2$				
$a_3b_3l_3$	$a_2b_3l_3$	$a_1b_3l_3$	$a_0b_3l_3$				
$a_3b_3h_3$	$a_2b_3h_3$	$a_1b_3h_3$	$a_0b_3h_3$				
$\text{ans}_h$				$\text{ans}_l$			

## Implementations

Schoolbook multiplication is used in order to multiply values of arbitrary size. The maximum permitted size of a value is dependant on the language being used.

Therefore, data structures are used to represent larger data types. For example, in C, an array can be used to store numbers larger than the 64-bit built-in long.

```
long* mul(long* a, long* b, long* c)
{
    for(int i = 0; i < 4; i++)
    {
        for(int j = 0; j < 4; j++)
        {
            c[i+j] += a[i]*b[j];

            if(c[i+j] >= 4294967296)
            {
                c[i+j+1] += 1;
                c[i+j] = 0;
            }
        }
    }
    return c;
}
```

## Using Distributivity

Each individual term in the first set of brackets is multiplied by each term in the second set of brackets.

For example, the multiplication of  $\{1x, 2y, 3z, 4w\}$  and  $\{4x, 3y, 2z, 1w\}$  would proceed as  $1x*4x, 1x*3y, 1x*2z, 1x*1w, 2y*4x$ , and so on. This allows for two

sets of 4 size  $x$  ( $x$  here is arbitrary size) elements to be multiplied, using variables to represent increments of  $2^{(\text{size } x)}$ . The output would, in turn, represent 8 size( $x$ ) elements.


## Faster Computers

For years, RSA encryption — based around the difficulty of integer factorization — was the gold standard in asymmetric cryptography. A recent major development in cryptography was the integration of elliptic curves. These curves are polynomials of the form  $y^2 = x^3 + ax + b$  with special properties that require less computation for encryption and decryption, as evidence suggested they were virtually uncrackable with much smaller values than RSA required to be so. In 1994, Peter Shor's work showed that conventional encryption schemes dependent on the difficulty of integer factorization, notably RSA, were vulnerable to quantum brute force attacks. It's also been shown that elliptic curve cryptography (ECC) is especially vulnerable to quantum attacks. Staying ahead of quantum computing provides an immense challenge for researchers, and faster operations mean we can hold off the effects of quantum supremacy longer.

## Montgomery Multiplication

## Montgomery multiplication is a modular multiplication

**Example. Use Montgomery Multiplication to solve the following:  $7 * 15 \text{ module } 17$ ,  $R = 100$**

$a = 7$ $b = 15$ $N = 17$ $R = 100$ $R' = 8$		<i>Final answer:</i> $= a * \underline{b} * R' \bmod 17$ $= 3 * 4 * 8 \bmod 17$ $= 96 \bmod 17$ $= 11$
<i>To convert to montgomery space:</i> $\underline{a} = a * R \bmod N$ $\underline{a} = 7 * 100 \bmod 17$ $= 700 \bmod 17$ $= 3$	$\underline{b} = b * R \bmod N$ $\underline{b} = 15 * 100 \bmod 17$ $= 1500 \bmod 17$ $= 4$	

algorithm, with notable applications in cryptography. It transforms values into a Montgomery representation, which allows for multiplication with bit shifts and addition, instead of costlier division and subtraction operations.

## Results

Previously, an algorithm was designed with the intent of adding arbitrary-sized integers for ECC. However, our group uncovered that in most cases (with integers  $\geq 64$  bits) this implementation produced erroneous results. Our discovery led to it being patched, and a working implementation of the algorithm.

We used **property-based testing** to identify errors in existing algorithms. This played a key role in implementing efficient computations. We create general properties that must be satisfied, and use Haskell's QuickCheck to run through a large set of values that expose the property.

QuickCheck: <https://www.youtube.com/watch?v=MmerAYuGVLQ>

## Conclusion

As the limits of technology are pushed ever further, we must strive to maintain ways to provide fast, secure communications and services. As adversaries gain access to better hardware, we need to use stronger encryption. To prevent security from being compromised, we need to continuously improve the efficiency and scale of our encryption schemes, which is the long term aim of our project.

## Acknowledgements