AN EXTENSION AND IMPLEMENTATION OF THE MOD WITHOUT MOD ALGORITHM TO EFFICIENTLY COMPUTE THE MODULUS OF A NUMBER IN HARDWARE

By

RYAN SWANN

Bachelor of Science in Electrical Engineering
Oklahoma State University
Stillwater, Oklahoma
2019

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2020

AN EXTENSION AND IMPLEMENTATION OF THE MOD WITHOUT MOD ALGORITHM TO EFFICIENTLY COMPUTE THE MODULUS OF A NUMBER IN HARDWARE

Thesis Approved:

Dr. James E. Stine, Jr.

Thesis Adviser

Dr. Keith Teague

Dr. John Hu

ACKNOWLEDGMENTS

I would like to sincerely thanks my adviser Dr. James E. Stine, Jr., for the tremendous amount of time and support he has given me during this research as well as providing a good role model for an ethical and fun outlook towards education and design.

I would like to thank my Parents, Rayce and Laila Swann for their never ending support in my endeavors and instilling within me the drive to do better.

I would like to thank my grandparents, Calvin and Louise Ellis and Pat and Tony Valentino for instilling within me good moral values and a positive outlook on the world.

I would like to express my thanks to my sister, Trayce Swann for keeping me fashionable and sane along my journey

I would like to express my immense appreciation to my long time friends Dustin Caples, Blake Loftin, Brady Loftin, Trevor Taylor, Jeff Lee, and John Allen for never giving up on me, regardless of how many times I tell them I'm busy.

I would like to thank Alex Underwood, Teo Ene, and Brett Mathis for keeping every day fun.

And to those I love, thanks for sticking around.

Acknowledgments reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: RYAN SWANN

Date of Degree: DECEMBER, 2020

Title of Study: AN EXTENSION AND IMPLEMENTATION OF THE MOD WITHOUT MOD ALGORITHM TO EFFICIENTLY COMPUTE THE MODULUS OF A NUMBER IN HARDWARE

Major Field: ELECTRICAL ENGINEERING

Abstract: This thesis discusses a hardware implementation of modulo that does not require a multiplication. This implementation is based on the algorithm proposed in Mark A. Will's "Mod without mod" in which the an algorithm is presented to calculate the modulus of large values using shifting and adding. This allows our implementation to be comparable in clock cycles to other implementations without the need for a multiplier's delay. This algorithm is compared with others, such as Barret reduction, Montgomery reduction, and fast modular reduction. Our implementation of this modulo algorithm is shown to be faster in many cases. This paper proposes both a hardware implementation of this algorithm as well as synthesis results in soi12s0 45nm IBM Multi-threshold CMOS (MTCMOS) technology and ARM-based standard cells.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Modular reduction is an important piece of arithmetic in modern designs the cornerstone of modular reduction is the modulo operation in which we find the remainder of a value that is difficult to reverse. The modulo operation's increase in importance in recent years is primarily due to the use of the operation in modern encryption techniques. Generally this operation has been accomplished by devices that are close in size and design to a full divider. This causes inefficient usage of resources that could be regained through use of more efficient algorithms.

As the modulo operation merits more usage due to it's increased occurrence in modern encryption algorithms that nearly all modern processor designs have speedup instructions for. Unfortunately, as these processors gain higher speed they can also better crack some encryption algorithms so we are forced to increase the key and operation size to keep up with the higher demand for security. One such algorithm that makes a lot of use of the modulo operations is Elliptic curve cryptography [1] in which modular arithmetic is used in operations of both point addition and point doubling which are the cornerstone operations of Elliptic Curve Cryptography.

Elliptic curves work based on the principle of "Finite Fields". Finite or Galois fields are fields that contain a finite number of elements. These fields, especially the galois fields that have an order of 2^n have interesting quirks that we can take advantage of to create encryption systems which are more efficient with smaller keysizes and footprint. These key systems use Finite fields to adequately create a public private key system through use of the difficulty of reversal in the finite field operation. Systems like RSA try to take advantage of the fact that it is difficult to factor a large integer that is comprised of prime factors but systems like elliptic curve cryptography better take advantage of more modern mathematical techniques to create these systems without sacrificing speed or security. The security and usability of any specific elliptic curve is commonly based on the base parameters that are selected to generate the curves. Generally when we are using Elliptic Curve Cryptography in any modern implementation we will use one of the standardized curves from places such as NIST. These curves are generally generated by a prime polynomial, many of the results for this thesis were generated based on the use of the NIST curve P384. This curve provides a modern level of security while also being able to accomplish the operation at a realistic speed. One important point is that modular arithmetic is also used in a newer technique known as "Supersingular isogeny key exchange" which has been touted as an adequate modern solution to the issues presented by quantam computers. This technique seems to be one of the better solutions for post quantam security as it is an algorithm that does not have a complexity that is significantly affected by the way that quantam computers operate. This operation, like ECC, also requires significant usage of the modulo operation. [2].

To summarize the importance of cryptography, essentially cryptography is what keeps data that is leaving any given piece of technology a near gurantee that it won't be intercepted and interpreted by an outside third party. One of the difficulties with this is that as more powerful processors become mass-available it is becoming more realistic for basic and widely available general purpose processors to be able to crack into some of the more archaic methods of securing our data. The difficulty is that as we try to combat the increased processing speed of would-be hackers we must also increase the size of the key that we are using for cryptography. The difficulty with this is that if we don't re-adapt our algorithms then the key size will continue to grow. For this purpose, algorithms have been developed that take advantage of interesting mathematical properties in order to create a more secure system without the need for immensely large key sizes. This fear was further accentuated by the introduction of quantam computing into the playing field. Quantum computers have approaches to cracking cryptography that can significantly reduce the total search space and complexity. For this purpose algorithms have been developed which do not have this quirk but do still require modular arithmetic such as the use of supersingular isogeny graphs which do not have the same flaw that allows quantam computers to take advantage of other such algorithms. Through these endeavors we developed algorithms such as AES and ECC which allow users to obtain higher security without having to compute values with significantly larger integer sizes.

Modular arithmetic, is considered to be a system of arithmetic that ”wraps around” once it reaches the modulus value [3]. This form of arithmetic is very common to many forms of cryptography and important as a common case for modern designs in the field of computer architecture due to the increasing focus on processor security and security speedup.

While computation of the remainder of a division without doing the actual division seems simple at first glance it can actually be quite mathematically complex. There are many methods proposed to compute the remainder on it’s own, but it is important to create evaluate th efficiency of a hardware implementation of these algorithms to avoid wasteful energy and time dependence. Although there are methods that are quite efficient utilizing Montgomery multiplication, they tend to be more complicated and area/energy intenstive [4, 5]. Other methods, such as double-add-reduce (DAR) methods are simplistic and slow [6].

One method that was discovered during the research for this thesis is ”Mod Without Mod” [7] which utilizes only shifting, adding, and subtracting in order to accomplish the remainder calculation. While the algorithm in [7] is promising, the original work does not propose any sort of hardware specific implementation or consideration.

This thesis proposes an architecture that accomplishes the algorithm proposed in [7] as well as extensions that can be made to the base algorithm in order to take advantage of some quirks available only to hardware design. Simulation and synthesis results are presented that show the performance of the proposed design against it's other contemporaries. These results are presented for 14nm, 32nm, and 45nm CMOS technologies using a standard-cell library and industry standard tools to calculate the critical path, area, and energy performance of the designs.

These results show that the proposed reducer has an approximately 10% decrease in critical path delay over it's contemporaries as well as a cycles wise comparison of the proposed reducer with it's contemporaries. To conclude, the newly designed reducer is shown to be a contender against other design options as well as an immensely viable design choice alongside in larger designs alongside other hardware with it's ability to reach speeds upwards of 2GhZ.

CHAPTER II

BACKGROUND

Modular arithmetic is a long standing and widely used form of arithmetic in which the numbering system "wraps around". The modulo is the process of finding the remainder from a given division operation. For example, if you have 100 divided by 9 in quotient and remainder form, which is 11 remainder 1, then the result of the modulo operation is 1 [9]:

$$100\, mod\, 9 \;=\; 1$$

Modular arithmetic is a great match for the world of cryptography for a couple of reasons [10]. First, when performing modular arithmetic we can keep our values in a certain range which is useful when we have a limited hardware size such as 64 bit for many modern day systems. Therefore, through the use of specific values for the modulus a "limit" can be placed on the number of binary digits required to represent a number. Second, the complicated computation of the modulo operation is difficult to reverse especially when it is done many times through an implementation of a modular arithmetic system in something like an elliptic curve encryption system.

An elliptic curve encryption system is able to provide a public private key system that is more secure per bit than other more common algorithms such as RSA. When calculating the encrypted output of an elliptic curve operation the arithmetic operation which gets the most use is the modulo operation. The use of this operation comes from the point addition and point doubling operations that make up the majority of the elliptic curve encryption system. While this is one example of a cryptography operation that uses modulo there are many other alternative encryption systems that also are heavily reliant on the modulo operation.

One type of reduction that is quite simple is the 2^{k-a} reduction [11]. The 2^{k-a} reducer is very simple in hardware but requires quite a few cycles and uses hardware components that require a lot of time for calculation. The pseudocode below shows the basic execution principle of the 2^{k-a} reduction.

Algorithm 1 2^{k-a} reduction [11]

```
$a = 2^k - m$
$r = x \ mod \ 2^k$
$q = int(\frac{x}{2^k} - 2)$
while $q \neq 0$ do
    while $q \neq 0$ do
        $r = r + (q * a) \ mod \ 2^k$
        $q = int(\frac{q*a}{2^k})$
    end while
    $q = int(\frac{r}{2^k})$
    $r = r \ mod \ 2^k$
end while
return $r$
```

As shown by the pseudocode in Algorithm 1, the algorithm operates primarily on r and q throughout the execution of 2^{k-a} reduction. The first thing that you may notice is that this algorithm uses *mod* 2^k within the while loop, which is a useful function when using binary arithmetic by which modulo by a power of 2^k is equivalent to dropping the bits more significant than index k. This allows us to have this as an operation with little logic in a hardware implementation. In this algorithm two loops are seen, an outer and an inner, that both operate on r and q. Within the inner loop we generate a number that is less than x and still gives the "correct" answer when a mod m operation is performed. We then, using the outer loop, run the inner loop multiple times to achieve $x\ mod\ m$. The calculation of $x\ mod\ m$ using $2k - a$ reduction also requires the use of a multiplier which is one of the slower operations in hardware arithmetic and thus will require approximately $n(n - k + 1)T_{mult}$[11] where T_{mult} is the amount of time required for a calculation using the multiplier, n is the size of the input, and k is the size of the reduced output.

Another great reduction algorithm is Barrett reduction, which is considered to be the gold standard for fast reduction [10, 12]. With Barrett reduction, there is interesting mathematical quirks that are utilized in order to calculate $x\ mod\ m$ without the need for looping which allows designers to make a reduction module that takes a small number of clock cycles. Unfortunately, the underlying arithmetic operations required for the calculation of $x\ mod\ m$ using this form of reduction are quite complicated and require the use of a multiplier as shown in Algorithm 2.

Algorithm 2 Barrett reduction [11]

```
if $b = 2$ then
    $t = 2$
else
    $t = 1$
end if
$c = int(\frac{b^n}{m})$
$y = int(\frac{x}{b^{k-1}})$
$w = y * c$
$q = int(\frac{w}{b^{n-k+1}}\ mod\ b^{k+t})$
$r = (x\ mod\ b^{k+t}) - (q * m\ mod\ b^{k+t})\ mod\ b^{k+t}$
```

The Barrett reduction algorithm is one of the best there is when it comes to number of clock cycles needed to achieve the result. Unfortunately, the calculation of $x\ mod\ m$ using Barrett reduction uses complicated operations. As we can see from

Figure 2.1, even the most basic implementation of Barrett reduction requires a multiplier, which is generally one of the slower arithmetic operations in an implementation. The Barrett reducer is a reducer with great performance in terms of number of cycles per operation but at the cost of a longer critical path through the multiplier.

The Barrett reducer shown makes use of both a multiplier and subtractor in order to calculate the modulus of the input value x. A multiplier implementation, especially one of sizes such as 256 and 384 bits, require a large amount of logic, area, routing, and delay in order to place and route into a design. The multiplier, as shown in the results section, bloats the area and power dissipation of the barrett reducer quite considerably and leads to a slower critical path delay and large energy usage than the proposed design, discussed later in this work.

When discussing real world applications of modulo it is hard to get around the use of modular multiplication. Modular multiplication is the most commmon process used in the aforementioned encryption techniques where a number is multiplied and then the modulo is taken of this post-multiplication value multiplied by the scalar

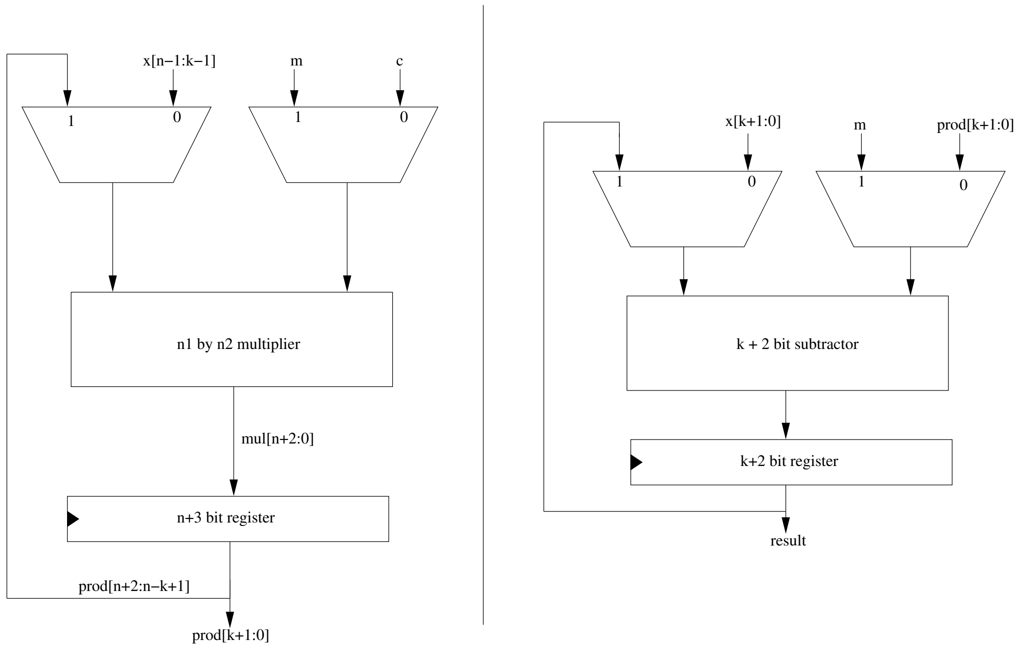

Figure 2.1: Example Implementation of Barrett Reducer

then it's modulus taken [13]. As a result of the modular multiplication's importance, there are a few interesting implementations.

One modular multiplication unit, the Double, add, and reduce (DAR) modular multiplier is an algorithm that accomplishes the multiplication and reduction at the same time. It accomplishes this through the use of modular addition inside the multiplication process.

In algorithm 3 the pseudocode for modular addition is shown. The DAR modular multiplication process using the modular addition function is then shown in algorithm 4.

Algorithm 3 Modular addition (x,y,m,k) [11]

```
# This method will add x and y mod m
$z_1 = x + y$
$z_2 = (z_1\ mod\ 2^k) + (2^k - m)$
$c_1 = int(\frac{z_1}{2^k})$
$c_2 = int(\frac{z_2}{2^k})$
if $c1 = 0$ and $c2 = 0$ then
    return $z1\ mod\ 2^k$
else
    return $z2\ mod\ 2^k$
end if
```

Within a modular addition implementation, hardware can take advantage of some interesting mathematical quirks by dropping the highest-order bit. This operation is the equivalent of a modulus by 2^k where k is the location of the dropped bit. Through use of this helpful piece of binary arithmetic $x + y\ mod\ m$ is easily calculated using this algorithm, as shown in Algorithm 4.

Through examination of the DAR modular multiplication operation, not only does it accomplish the calculation in relatively few operations but it also does not

Algorithm 4 DAR modular multiplication ($x * y \; mod \; m$) [14]

```
m = modulus
k = size of the reduced output

$p = 0$
$i = 0$

# x[i] is binary digit in x at position i
for x[i] in binary(x) do
    # Double operation ($p + p \; mod \; m$)
    $p = modAdd(p, p, m, k)$

    if $x[k - i - 1] = 1$ then
        # Add operation ($p + y \; mod \; m$)
        $p = modAdd(p, y, m, k)$
    end if

    $i = i + 1$
end for
```

require the use of a multiplier. This is great because a multiplier is a relatively slow component so the DAR modular multiplier can operate on a somewhat higher clock speed.

One of the most commonly used and amazing forms of modular multiplication is Montgomery reduction. Montgomery reduction has many different versions and implementations that could not possibly be covered in the scope of this thesis. Montgomery reduction is commonly used in many implementations of encryption, the primary reason for this is that Montgomery reduction implements a scalar multiplication

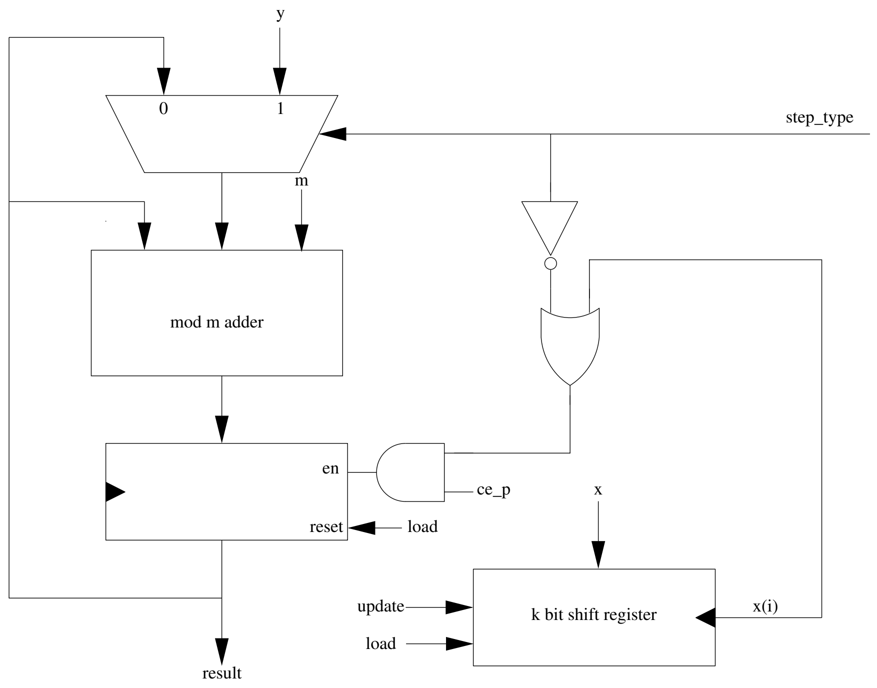

Figure 2.2: Example implementation of a DAR modular multiplier

naturally and, therefore, does not require a multiplier to multiply by a scalar [5]. This feature allows for an implementation that accomplishes both the multiplication and reduction at the same time. In algorithm 5 a simplified form of montgomery reduction is shown As can be seen from the simplified form of the Montgomery algorithm shown in Algorithm 5. The reason that this algorithm is commonly used is that there are many methods in both software and hardware that can make use of this algorithm. This allows Montgomery multiplication to be both cycle and time efficient. The primary issue with montgomery reducers is that they are complicated and still often have a large critical path delay as well as a high complexity and area [5]. Consequently, there is a need for something better than both Barrett and DAR methods as shown in this thesis.

The background for the implementation proposed in this thesis is primarily based on the algorithm originally proposed in [7] in which an algorithm is discussed by the authors that is based primarily on shifting the initial value and adding a pre-

Algorithm 5 Summarized Montgomery Reduction [15]

```
x = Multiplicand
Y = Multiplier
n = bit length of modulus
m = modulus
m' = $-m[0]^{-}1 \; mod \; r$
n is the size of the multiple of x and y
$Z = 0$
for $i = 0$ to $n - 1$ do
    $Z = Z + XY[i]$
    $q_M = (Z \; mod \; r)M' \; mod \; r$
    $Z = \frac{(Z+q_M M)}{R}$
end for
if $Z >= M$ then
    $Z = Z - M$
end if
```

calculated value based on the result of the shift. As one can see from Algorithm 6, the algorithm is very simple and has hardware implementable mathematical quirks that would seem to produce an efficient hardware implementation. On the other hand, there is no hardware implementation given in the original paper [7] by the authors.

A lot of the magic of this algorithm is accomplished by taking advantage of hardware's ability to drop a high order bit in order to modulo by the 2^{index} where the index is the index at which we drop the bit. We then repeatedly add in the difference between this value and our modulus, these mathematical phenomena lend themselves well to an application-specific implementation for encryption as one can use many different types of approaches in hardware to accomplish these calculations, therefore, the user can make a judgment call as to if the prefer area, critical path, or cycle-wise performance.

The original paper suggests that the algorithm appears to have real world benefits over previous algorithms when it comes to software implementation. In the original paper [7] the authors compared their new algorithm with Montgomery reduction,Barrett reduction, fast modular reduction, and a real world test where the proposed algorithm was compared with a GNU multiple precision algorithm that is common to many Linux systems and thus widely used. To summarize the findings of the original work [7], comparison with the Barrett reduction found that the proposed algorithm had a significant performance increase not in the number of operations required but in the operational complexity as Barrett reduction requires a large sized multiplication while the `mod without mod` algorithm does not. When compared with Montgomery reduction the original paper states that based on half of the bits in the input vector being high , a normal case if you're normalizing your RNG output, Montgomery modular multiplication would require more addition operations than the proposed mod without mod algorithm and also sometimes requires a few extra cycles for correction. Lastly, the paper in which this algorithm is proposed [7] states that

Algorithm 6 Mod without mod reduction algorithm [7]

```
m = modulus value
modLength = length in bits of modulus value
z = input value
n = size of z in bits
k = size of reduced output
modVal = $2^{modLength}\ mod\ m$
$shiftcnt = 0$
$result = z[n : \frac{n}{2}]$ # (upper half in bits)
# Shift an equal amount to the number of bits in m
while $shiftcnt < modLength$ do
    # If result[n+1] is not a 1
    if $mod.bit\_length() >= result.bit\_length$ then
        $result = result << 1$
        $shiftcnt = shiftcnt + 1$
    else
        $result = result[\frac{n}{2} - 1 : 0]$ # in bits
        $result = result + modVal$
    end if
end while
# Add back the bottom half of z
$result = result + z[\frac{n}{2} - 1 : 0]$
while $result >= mod$ do
    $result = result - mod$
end while
# Check for overflow one last time
if $mod.bit\_length() > result.bit\_length$ then
    $result = result[\frac{n}{2} : 0]$ # in bits
    $result = result + modVal$
end if
```

the fast modular reduction method is similar as it only requires processing the upper half of the bits and, therefore, only requires $radix/2$ operations, but the fast modular reduction method, as stated in [7] requires more clock cycles on average despite it being unsubstantiated.

Through examination of the algorithm we can see that a lot of the iterations are being used in the shift and add section. It would appear at face value that it would be possible to take further advantage of the shift and add quirk that is provided by the algorithm. Through further examination it is simple to extrapolate that it is possible to shift out multiple bits at a time. And then if we recalculate a modulus value based on the two highest order bits shifted out past the index at length of modulus. For example, if we have a modulus length of 384 bits then we would be able to calculate modVal = 2^{384} mod MOD and modVal2 = 2^{385} mod MOD. Here we can see that we can simply calculate these multiple modVals based on any number of extended bits that we would like to calculate in parallel. In algorithm 7 you can see that we can take advantage of this quirk in order to decrease the approximate number of iterations by $\frac{1}{\#ofparallelbits}$

Algorithm 7 Mod without mod with 2 bit parallelization [7]

```
m = modulus value
modLength = length in bits of modulus value
z = input value
n = size of z in bits
k = size of reduced output
modVal = $2^{modLength}$ $mod\ m$
$shiftcnt = 0$
$result = z[n : \frac{n}{2}]$ # (upper half in bits)
# Shift an equal amount to the number of bits in m
while  $shiftcnt < modLength - 1$ do
    # If result[n+1] is not a 1
    if $mod.bit\_length() >= result.bit\_length$ then
        if shiftcnt + 2 ¡= modLength then
            result = result << 2
            shiftcnt = shiftcnt + 2
        else
            result = result << 1
            shiftcnt = shiftcnt + 1
        end if
        $result = result << 1$
        $shiftcnt = shiftcnt + 1$
    else
        $result = result[\frac{n}{2} - 1 : 0]$ # in bits
        $result = result + modVal$
    end if

    if $(result[modlength] == 1) and (result[modlength + 1] == 1)$ then
        result = result & 0xFF
        result = result + $2^{modlength}$
        result = result + $2^{modlength+1}$
    else if $result[modlength + 1] == 1]$ then
        result = result & 0xFF
        result = result + $2^{modlength+1}$
    else if $result[modlength] == 1]$ then
        result = result & 0xFF
        result = result + $2^{modlength}$
    end if
end while

# Add back the bottom half of z
$result = result + z[\frac{n}{2} - 1 : 0]$

while $result >= mod$ do
    $result = result - mod$
end while

# Check for overflow one last time
if $mod.bit\_length() > result.bit\_length$ then
    $result = result[\frac{n}{2} : 0]$ # in bits
    $result = result + modVal$
end if
```

CHAPTER III

IMPLEMENTATION

The implementation proposed by this thesis is comprised of a control system and a datapath, where the control is namely the finite state machine (FSM) that manages all of the system's control signals.. In the Finite State Machine, a counter is implemented in order to count the number of shifts that have happened through the use of a flag which has been output from the datapath. This allows the design to keep track of how many shifts are performed such that it can ensure that the design will complete size/2 shifts. In the following section there are diagrams for these key components as well as an explanation of their function and how they connect. One of the primary operations in this design is that anytime there is a 1 in the most significant bit position it is dropped in order to essentially take mod 2^n. After this bit dropping, the module will perform an addition of `modVal` which is precalculated using the equation $modVal = 2^{mod\ length}\ \%\ mod$. Due to the nature of this value it will need to be recalculated every time the modulus changes, which is also required for a value in the barrett implementation. This course of action essentially accomplishes a modulo by 2^n followed by an adding of the different between 2^n and mod.

Figure 2.2 the datapath of the reducer. This datapath is comprised of two distinct sections. These sections are the initial shifting and adding section as well as the final correction section.

In the shifting and adding section the design utilizes a multiplexer that is selects between loading of a new input value or the output of the shift and add section. After the initial value is loaded into the register this multiplexer then selects the output of

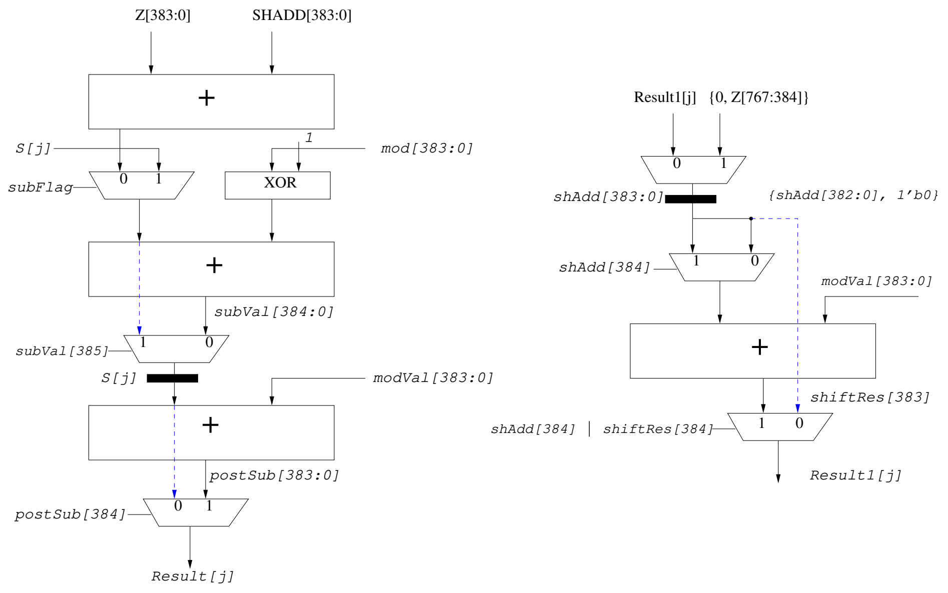

Figure 3.1: The primary datapath of the proposed reducer

the shift and add section to be looped back into the flop with enable This flop has it's enable signal to `shift_done` where it will only save values flag is 0 meaning that the requisite number of shifts has not been reached. The `shift_done` flag becomes a 1 when the counter signals to the datapath that the module has completed a number of shifts that is equal to the size of the reduced output in bits. When disabled, after the requisite number of shifts has been reached, the register will hold the final value calculated by the shift and add section. When enabled, and not accepting new values, the shift and add loop is fed into the next set of multiplexers that will decide what portion of the shift and add section needs to be used to compute the next result. Although the datapath within Figure 2.2 is designed for a 384-bit implementation, it can be easily adapted to any size.

There are two distinct cases that the shift and add section is designed to deal with. First, we have the case that there is no overflow on `shadd_reg` in which case a shift operation is required that selects the output of the shifter to be the input of the adder, which splits itself into two other subcases. One subcase is that a 1 is shifted out in the shift process, in this case the multiplexer selects the result of the adder and that value is saved to the register. In the second subcase, where the shifter shifts out a 0, the multiplexer pair selects the `shift_result` to be saved to the register. The second case is that the register saves a value that has an overflow itself, usually as the result of the add. In this subcase the datpath chooses the output of the register that allows the design to perform an add without a shift to handle the overflow. This set of operations takes place in State 2 until the desired number of shifts has been reached. After this, the register stops accepting new values and the second stage begins operating.

Beginning from the 3rd state, the primary calculation takes place in the bottom half of the design. To begin, there is an adder that reads the bottom half of the input back in. Then, a multiplexer utilizes the value of `subFlag` (from the FSM) to decide between the output of the second register or the adder. Initially, `subFlag` selects the output of the aforementioned adder. It then subtracts the value of mod and checks the output of the subtraction for a negative sign [16]. This subtraction and register stage is utilized to make sure that the value of the result is less than the value of mod. When the output of the subtraction is negative this symbolizes that the current result is less than mod. In the case that the output is not negative, the second multiplexer selects the output of the subtractor and saves it into the register. This value is then fed back into the first multiplexer and then goes through the subtraction stage again until the value is negative. Once the value is negative, the final mux and adder will add modVal one last time in the case of an overflow. The result of the final multiplexer select is then taken as the final reduced result $z \ mod \ m$

Figure 3.2 shows the state transitions of the FSM. This specific finite state machine has six states that have their own function. State 0 waits for the start signal to become 0 in order for the machine to start a new operation. State 1 is the setup that waits for the start signal to be asserted again. State 2 is the state that waits for the counter to complete the amount of shifts required by the system. State 3 sets up the subtraction stage for its operation and performs the initial add of the lower half and the initial subtract. State 4 allows the subtraction stage to subtract until the result of the subtract becomes negative. Finally, state 5 asserts the `done` signal and returns to the ready state to await new inputs. In Figure 3.3, the state transition table is given to

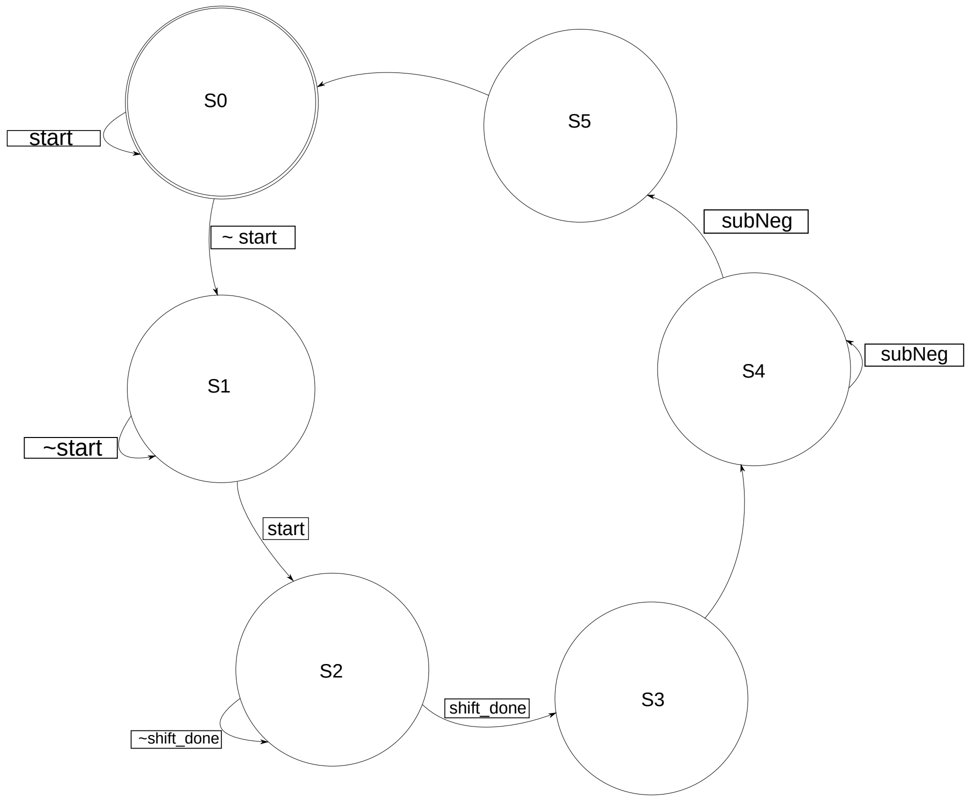

Figure 3.2: Finite State transitions of the FSM

State	reset_count	load_z	done	subFlag
S0	0	0	0	0
S1	start	1	0	0
S2	0	0	0	0
S3	0	0	0	0
S4	0	0	0	1
S5	0	0	1	0

Figure 3.3: Signal values based on FSM state

reveal the the value of each signal.

Algorithm 8 N bit counter's function on clock

```
n = length of reduced output

if reset = 1 then
    shift_done = 0
    count = n
end if

if reset_count = 1 then
    count = n
end if

if count_decrement & !shift_done then
    count = count - 1
end if

if count = 0 then
    shift_done = 1
end if
```

Algorithm 8 displays the function of the counter in the FSM. This counter is used to count down the amount of shifts that have taken place (including accounting for when we shift and do not add) as well as being able to reset from within the state machine. This algorithm describes the equivalent function of the counter that will take place on every clock cycle.

3.0.1 Architecture Improvement

The base design for this implementation is quite simple but a diligent designer can choose to use particular improvements such as replacing some of the simple components with more complex ones that compromise in a particular area. For example, compounding operations that take multiple cycles into operations that take only one or replacing the carry propagate adders with faster prefix adders. One example that has been implemented for this thesis is the use of a leading zero detector (LZD) [17]. A leading zero detector can be implemented to shift out all of the leading zeroes of the result at once. This addition allows the reducer to reduce computation time by one cycle for every recurring leading zero. For example, if the result value saved in the register has 7 leading zeros then the implementation will save 7 cycles by shifting out all 7 of these zeros in a single cycle using the output of the LZD to signal a variable shifter. The LZD itself resides inside the counter so that it can easily signal how many shifts are going to happen so the counter can subtract that value from it's remaining count. The number of shifts needed to shift out all of the zeroes is then passed over to the datapath. This allows for considerable speedup on large operations as displayed in the results section.
Another improvement is the possibility of processing multiple bits in parallel. Through the addition of a couple of multiplexers to the base datapath we can accomplish the multiple bits in parallel operation as outlined in the background. The 2 bit parallel version of the implementation is shown in figure 3.4.

Examination of figure 3.4 shows that through adding a couple of multiplexers we can significantly reduce the number of cycles required by the shift and add section. As outlined in the results section, this provides a significant reduction to the number of total cycles required for the implementation to accomplish a reduction operation.

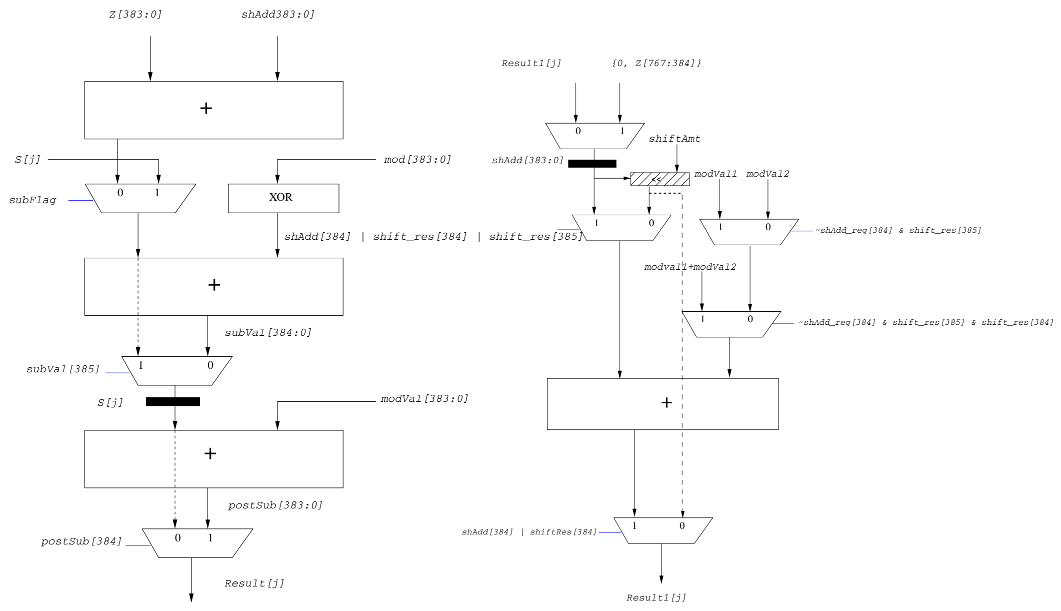

Figure 3.4: The primary datapath of the 2 bit parallel reducer

The additional interesting quirk of this parallel implementation is that it requires very little hardware for a significant reduction in cycles. In theory, we would parallelize an infinite number of bits by creating a lookup table that holds the modVal entries at each possible value of the shifted out tag. Using the leading zero detector we can shift out the leftmost zero to the Size + (# of bits) space. Which then allows us to up the counter by (# of parallel bits) + (# of zeroes). This lookup table scales exponentially with the number of bits in parallel that it processes. For example, a 3 bit parallel table will require 2^3 entries and a 4 bit parallel table will require 2^4 entries. This shows that there should be an ideal number of bits of parallelization that can be obtained without adding a large amount of area or delay. This is further explored in the results section.

Overall, the proposed implementation is generally more simple and straight forward than other implementations while still maintaining good speed and performance. The proposed design also has the ability to be further expanded and upgraded and has high modularity which you can use to tailor the new design to your needs such as adding new logic in order to further improve on the performance of the device or minimize the area for example. This versatility and upgradability are a good reason why this design is more versatile than its peers while maintaining good performance.

CHAPTER IV

RESULTS

In order to compare this module with others we must first discuss some alternative designs that are close in function to the proposed design. To adequately measure the performance compared to its peers for a reduction, the design will be compared with the DAR multiplier as well as the Barrett reducer in order to test it's abilities in both the modular arithmetic and pure reduction use cases.

The flow used to obtain these results requires a couple of different industry standard tools and libraries using typical PVT conditions.

First, the design was implemented into HDL in order to test functionality. We then passed the HDL over to Cadence Design System (CDS) Genus$^{\text{TM}}$ synthesis engine which is utilized in order to synthesize the design and get basic timing and area results. We then used the standard cells to get a rough estimate of the design's real world performance this place and route was accomplished by Cadence Design System's Innovus which used the 14, 32, and 45nm ARM standard-cell libraries that were provided to it. Innovus$^{\text{TM}}$ then probes the placed and routed circuit in order to find the critical path of the design and estimates the power usage and based on the parameters defined by the standard cell kit. The result of this design flow with each of the aformentioned designs is presented and analyzed in the following section.Cycle-wise performance was obtained using Mentor Graphics Corporation Modelsim$^{\text{TM}}$ in order to cycle-wise simulate the HDL version of the design we used over $10,000$ of the same vectors on each of the designs for the comparsion. algorithmically as seen by Tables 4.2, 4.4, and 4.6

4.0.1 Proposed Reducer compared to Barrett Reducer

Initially , the proposed reducer is compared against the Barrett reducer which is one of the most common implementations in the field and is currently considered the "Gold Standard" of modular reduction hardware. The Barrett reducer is a very well designed piece of hardware which requires few clock cycles to calculate the modular reduction operation. Unfortunately for the Barrett reducer this is a difficult tradeoff as the use of a large sized multiplier (384 bit $*$ 384 bit in this case) causes high area, complexity, and critical path delay. Tables 4.1 and 4.2 demonstrate the comparison between the new reducer design and the Barrett reducer in both area and critical path delay. Both the Barrett and the proposed reducer are implemented in a 768 to 384 bit reduction form and are verified using the NIST P-384 curve modulus values [18]. The Barrett reducer, as anticipated, outperforms the newly proposed reducer in cycle-wise performance as shown in Table 4.4, however, as seen in the other tables this is at the expense of a significant area and energy cost.

Comparison of the proposed design with the Barrett reducer in terms of critical path delay shows that the proposed design wins outright with or without the added leading zero detector. This large difference in other areas is due mostly to the critical path introduced by the 384 $\times$ 384-bit multiplier in the Barrett reducer. Comparison of the proposed design with the proposed design with the LZD shows an increase in delay and area due to the use of both a variable shift shifter as well as the leading zero detector itself. Overall the proposed design without the LZD is able to run at over

Technology	Proposed [ps]	Proposed w/ LZD [ps]	Barrett Reducer [ps]
14nm	437.4	655.2	1,103.4
32nm	359.6	515.0	1,018.1
45nm	438.0	617.0	1,040.0

Table 4.1: Critical path comparison of the Barrett and proposed reducers designs with P-384 modulus

Technology	Proposed Reducer [μm^2]	Proposed Reducer w/LZD [μm^2]	Barrett Reducer [μm^2]
14nm	10.426	16.308	186.326
32nm	16.633	38.605	452.643
45nm	42.396	92.065	1,226.104

Table 4.2: Area comparison of the Barrett and proposed reducer with P-384 modulus

Technology	Proposed Reducer	Proposed Reducer w/LZD	Barrett Reducer
14nm	14,667	25,894	254,228
32nm	11,363	25,308	153,151
45nm	11,878	25,197	268,819

Table 4.3: Comparison of cell counts in Proposed Reducer vs Barrett

double the speed of the Barrett reducer on average while the design with LZD is able to remain ahead of the Barrett reducer with a critical path savings of approximately $25 percent$.

While area is a good metric to compare different reducer designs in the same technology, another important metric is to look at the number of cells used for this design as that gives an idea of the size of the reducer regardless of the specific technology used. It also demonstrates a value proportional to the overall static and dynamic power utilization for each design. Table 4.3 again demonstrates the large difference in number of cells used in the proposed reducer compared to the Barrett reducer.

By examining the area results it is once again obvious that the proposed design wins outright, this is likely, again, due to the 384×384-bit multiplier that takes a large amount of routing and space to do the multiplication logic. When comparing the values between the proposed design without the LZD and the barrett reducer the large multiplier size causes the barrett reducer to unreasonably scale at larger radix sizes. That is, with a large size the area of the proposed design scales relatively linearly whereas, the Barrett reducer has trouble with the large area required to accomodate the multiplier required for Barrett reduction. When comparing the design with LZD

Vectors	Proposed Reducer	Proposed reducer w/LZD	Barrett Reducer
ECP384	390 cycles	104 cycles	6 cycles

Table 4.4: Average Cycle comparison of the Proposed reducer and the Barrett Reducer

Technology	Proposed Reducer [mW]	Proposed Reducer w/LZD [mW]	Barrett Reducer [mW]
14nm	0.3167	0.6664	8.9051
32nm	4.7443	13.7276	104.5652
45nm	2.9356	8.6443	157.8278

Table 4.5: Power comparison of the Barrett and proposed reducer with P-384 modulus

to the Barrett reducer it is again obvious to see that the design scales much better than the Barrett reducer. Unfortunately, as a result of the logic required for a shifter, the proposed design with LZD is quite a bit larger than the original design but this once again shows that the proposed design is able to increase its performance at the expense of area.

Cycle-wise comparison of each reducer, shows evidently that Barrett reducer still has a clear advantage over the proposed design. This is the trade off that the Barrett reducer takes in exchange for a high critical path and area impact. path. When comparing the LZD and non-LZD version of the proposed reducer it is shown that, as expected, the LZD significantly decreases the number of cycles required on average for the reducer to complete an operation.This set of results shows that the improvement expected from the leading zero detector can be readily applied to a real world use case as a more than viable upgrade to the base design.

Another important design consideration when comparing two implementations is the amount of power used as this shows the efficiency of the design when compared to its peers. The amount of synthesis estimated power is shown in Table 4.5.

By examining the power figures given by the synthesis engine we can draw the conclusion that the dsign proposed in this thesis also consumes significantly less power than the barrett reducer. This is largely due to the poor scalability of the Barrett reducer at large sizes. Table 4.2 indicates a 94.40% and 91.25% area reduction for the Barrett reducer compared to the proposed algorithm and the modified proposed algorithm with LZD, respectively.

When considering only cycle-wise performance the Barrett reducer is an obvious winner.But on the other hand, for implementations that require low area and energy requirements, the proposed architecture can be a significantly better choice than the barrett reducer.

4.0.2 Proposed Reducer + Multiplier compared to DAR multiplier

Next, the proposed design and the DAR modular multiplier are compared using random data. All designs are coded using Register-Transfer Language Verilog to take advantage of synthesis and any intellectual property (e.g., ChipWare) that is inserted. All designs are completely verified using hundreds of thousands of test vectors generated by python scripts. In both timing and cycle-wise performance the proposed design beats the DAR implementation. Specifically, clock-cycle wise, a 384-bit (i.e., a 384 bit output as for example 384×384 bit mod 384 bit) the DAR implementation uses on average 772 clock cycles while the multiplier + 768 bit to 384 bit proposed reducer comes in at an average of only 580 clock cycles. This is tested through HDL simulation across $10,000$ fully random (non-P384) vectors.

It is important to note that the DAR implementation includes modular multiplication as opposed to the reducer alone. Therefore, this subsection deals with adding modular multiplication along with the proposed algorithm in this thesis which will be larger in area/energy. The proposed algorithm is compared versus modular multiplication using several random vectors (not P-384) against the DAR architecture.

Table 4.8 demonstrates the difference in power consumption between the proposed reducer and the DAR modular multiplier. This data shows that while the proposed reducer offers significant improvements in areas other than power there is a tradeoff required for that increased performance.

Table 4.8 shows again, that the new design uses only slightly more cells than the DAR modular multiplier and thus is a great design alternative with only very small drawbacks.

The results presented in Tables 4.6, 4.7, 4.8, and 4.9 show that, while being combined with a standard cell multiplier, the proposed implementation beats the DAR modular multiplier in both critical path delay and clock cycles. This data shows that the proposed implementation can outperform other implementations in a real world use case.

Next, it is important to compare the new reducer directly with other reducers. As some of the reducers do not function as well at a higher radix then to better compare the proposed reducer with some of the other reducers it was concluded that the best option is to use a lower radix to give an advantage to the other reducers. Since this is the case, a 16 bit to 8 bit comparison was chosen for this comparison. In figure 4.1 the comparison in clock cycles and critical path delay using the same technologies as the comparison with the DAR is shown.

The data shown in Figure 4.1 shows that the proposed reducer performs an opeara-tion in a similar amount of cycles as some of it's contemporaries. While this operation takes a similar average amount of cycles the delay per cycle is obviously improved. The performance of the proposed reducer is approximately 10% better than the other compared reducers, this performance increase is likely due to the proposed reducer not requiring the use of a complicated operation, namely multiplication. Critical path

Proposed Design + Multiplier	DAR modular multiplier
580 cycles	772 cycles

Table 4.6: Average cycle comparison for 384-bit design based on $10,000$ random input vectors

Technology	Proposed Design [ps]	DAR modular multiplier [ps]
14nm	661.1	939.9
32nm	667.3	964.4
45nm	826.0	1,013.0

Table 4.7: Critical path comparison for proposed design vs DAR

Technology	Proposed [mW]	DAR mult. [mW]
14nm	0.2598	0.1364
32nm	3.3118	3.1438
45nm	1.7163	2.4945

Table 4.8: Power Comparison for proposed design vs DAR

Technology	Proposed	DAR mult.
14nm	13,167	8,805
32nm	7,914	8,617
45nm	8,130	9,027

Table 4.9: Comparison of cell counts for proposed design vs DAR multiplier

Reducer	Avg Cycles	14nm	32nm	45nm
Proposed	12	186 ps	132 ps	385 ps
Barrett	12	212 ps	183 ps	390 ps
$2^k - a$	13	210 ps	200 ps	404 ps

Figure 4.1: Cycle and critical path comparison of the proposed reducer, Barrett reducer, and $2^k - a$ reducer

delay is very important as it will allow a system using this design to have a higher overall clock speed without being bottlenecked by the reducer.

4.0.3 Parallel Reducer Results

This section displays results for different amounts of parallel calculation for the proposed reducer. These results show the performance of an implementation with 2, 3, 4, 8, and 16 bits in parallel. These different amounts of parallelization adequately show the scaling factor based on more or less bits of parallelization and their effects on cycle, critical path, and area performance. These results are shown in figure 4.2.

# of parallel bits	Avg Cycles	Critical Path	Area
2	71	481	6981
3	53	480	7060
4	45	476	7700
8	28	473	6795
16	18	700	10033

Figure 4.2: Parallel Results of a 768 to 384 bit reducer

Examination of these results suggests that up to approximately 8 bits processed in parallel there is very little effect on critical path and area. This is due to only requiring the addition of a small LUT to calculate the additional addition parameters required for the calculation. This results in a significant reduction in required cycles without much of a tradeoff on any of the other fronts. An 8 bit table significantly reduces the number of cycles required for calculation down to the point where it is nearly comparable to that of the gold standard Barrett reducer. These results show very promisingly that the reducer with this addition is both capable of modularity as well as very competitive with the performance of it's contemporaries.

In conclusion, the new reducer design is very comparable with it's contemporaries. The new design shows significant improvement in the areas expected through analysis of the original algorithm. More precisely, in comparison with the gold standard Barrett reducer the design shows significantly better performance with respect to both critical path and area. Specifically, area is in the order of 10x better and critical path performance is 2x better than the Barrett reducer but requires a larger amount of clock cycles. When compared behaviorally to it's other contemporaries in the modular arithmetic space the proposed reducer also shows significant benefits, boasting a significantly lower critical path and cycle requirement with a slightly larger size than the DAR modular multiplier.

CHAPTER V

CONCLUSION AND FUTURE RESEARCH

This work demonstrates a new implementation of a modular reduction algorithm as presented in [7] in which an algorithm is presented that is able to calculate the remainder of an operation without the use of multiplication or division operations. An important element of this paper is that it demonstrates an architecture for implementation of this algorithm, whereas, the original paper [7] did not have an architecture or discuss its potential implementation. A newly designed implementation of this algorithm is proposed as well as results suggesting that it is a viable improvement for other alternative reducer designs. Synthesis and simulation results are shown which represent the idea that the new implementation of this algorithm shows improvements in both critical path delay and amount of clock cycles required for an operation. The new implementation uses a shifter and adder but does not require a multiplier or divider for the reduction which allows for a low critical path delay. This implementation has uses in fields such as encryption for modular arithmetic which will get more complicated as the encryption standards get more strict. The current implementation manages to keep a relatively low completixy while still performing the remainder calculation with better performance than it's contemporaries in cycle count, critical path delay, or area.

REFERENCES

[1] M. Rosing, *Implementing Elliptic Curve Cryptography.* USA: Manning Publications Co., 1999.

[2] D. Jao and L. De Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Post-Quantum Cryptography* (B.-Y. Yang, ed.), (Berlin, Heidelberg), pp. 19–34, Springer Berlin Heidelberg, 2011.

[3] M. Muller-olm and H. Seidl, "Analysis of modular arithmetic," vol. 29, pp. 139–139, 03 2005.

[4] P. Kornerup, "High-radix modular multiplication for cryptosystems," in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pp. 277–283, 1993.

[5] A. Carter, P. Ning, W. Koven, D. M. Harris, M. Braly, N. Jones, J. Massas, T. Murakami, A. Simoni, and S. Mathew, "Comparison of parallelized radix-2 and radix-4 scalable montgomery multipliers," in *2013 Asilomar Conference on Signals, Systems and Computers*, pp. 1144–1148, 2013.

[6] F. Rodriguez-Henriquez, N. A. Saqib, A. D. Prez, and C. K. Koc, *Cryptographic Algorithms on Reconfigurable Hardware.* Springer Publishing Company, Incorporated, 1st ed., 2010.

[7] M. A. Will and R. K. L. Ko, "Computing mod without mod." Cryptology ePrint Archive, Report 2014/755, 2014. `https://eprint.iacr.org/2014/755`.

[8] R. Swann and J. E. Stine, "An improved hardware architecture for modulo without multiplication," in *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 635–638, 2020.

[9] "What is modular arithmetic?."

[10] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st ed., 2009.

[11] J.-P. Deschamps, *Hardware Implementation of Finite-Field Arithmetic*. USA: McGraw-Hill, Inc., 1 ed., 2009.

[12] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86* (A. M. Odlyzko, ed.), (Berlin, Heidelberg), pp. 311–323, Springer Berlin Heidelberg, 1987.

[13] S. C. Coutinho, *The Mathematics of Ciphers: Number Theory and RSA Cryptography*. USA: A. K. Peters, Ltd., 1999.

[14] J.-P. Deschamps, G. D. Sutter, and E. Cant, *Guide to FPGA Implementation of Arithmetic Functions*. Springer Publishing Company, Incorporated, 2014.

[15] M. Knezevic, F. Vercauteren, and I. Verbauwhede, "Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods," *IEEE Transactions on Computers*, vol. 59, pp. 1715–1721, Dec 2010.

[16] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2003.

[17] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 124–128, 1994.

[18] National Institute of Standards and Technology, "Digital signature standard (DSS)," tech. rep., 2013. `https://csrc.nist.gov/publications/detail/fips/186/4/final`.

[19] S. Ghosh, M. Alam, D. R. Chowdhury, and I. S. Gupta, "Parallel crypto-devices for gf(p) elliptic curve multiplication resistant against side channel attacks," *Computers and Electrical Engineering*, vol. 35, no. 2, pp. 329–338, 2009.

[20] M. S. Hossain, Y. Kong, E. Saeedi, and N. C. Vayalil, "High-performance elliptic curve cryptography processor over nist prime fields," *IET Computers Digital Techniques*, vol. 11, no. 1, pp. 33–42, 2017.

[21] H. Ahmadi, A. Afzali-Kusha, and M. Pedram, "A power-optimized low-energy elliptic-curve crypto-processor," *IEICE Electronics Express*, vol. 7, no. 23, pp. 1752–1759, 2010.

[22] C. Chen and Z. Qin, "Improved elliptic curve cryptographic processor for general curves over gf(p)," in *IEEE 10th INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING PROCEEDINGS*, pp. 1849–1852, Oct 2010.

[23] G. Chen, G. Bai, and H. Chen, "Hello," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, pp. 412–416, May.

[24] S. Chung, J. Lee, H. Chang, and C. Lee, "A high-performance elliptic curve cryptographic processor over gf(p) with spa resistance," in *2012 IEEE International Symposium on Circuits and Systems*, pp. 1456–1459, May 2012.

[25] E. Öztürk, B. Sunar, and E. Savaş, "Low-power elliptic curve cryptography using scaled modular arithmetic," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3156, pp. 92–106, 2004.

[26] D. Zhang and G. Bai, “Ultra high-performance asic implementation of sm2 with spa resistance,” vol. 9543, pp. 212–219, Springer Verlag, 2016.

[27] H. R. Ahmadi and A. Afzali-Kusha, “Very low-power flexible gf(p) elliptic-curve crypto-processor for non-time-critical applications,” in *2009 IEEE International Symposium on Circuits and Systems*, pp. 904–907, May 2009.

[28] Z. Zhao and G. Bai, “Ultra high-speed sm2 asic implementation,” in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 182–188, Sep. 2014.

[29] J.-W. Lee, S.-C. Chung, H.-C. Chang, and C.-Y. Lee, “An efficient countermeasure against correlation power-analysis attacks with randomized montgomery operations for df-ecc processor,” vol. 7428, pp. 548–564, 2012.

[30] Sangook Moon, Jaemin Park, and Yongsurk Lee, “Fast vlsi arithmetic algorithms for high-security elliptic curve cryptographic applications,” *IEEE Transactions on Consumer Electronics*, vol. 47, pp. 700–708, Aug 2001.

[31] K. C. C. Loi and S. Ko, “Scalable elliptic curve cryptosystem fpga processor for nist prime curves,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, pp. 2753–2756, Nov 2015.

[32] A. Salman, A. Ferozpuri, E. Homsirikamol, P. Yalla, J. Kaps, and K. Gaj, “A scalable ecc processor implementation for high-speed and lightweight with side-channel countermeasures,” in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, Dec 2017.

[33] P. L. Montgomery, “Modular multiplication without trial division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[34] C. Paar, *Understanding cryptography : a textbook for students and practitioners.* Berlin ; London: Springer, 2009.

[35] E. Savas and C. K. Koc, "The montgomery modular inverse-revisited," *IEEE Transactions on Computers*, vol. 49, pp. 763–766, July 2000.

[36] M. Kaihara and N. Takagi, "Bipartite modular multiplication method," *IEEE Transactions on Computers*, vol. 57, pp. 157–164, Feb 2008.

[37] C. Juvekar, *Hardware and protocols for authentication and secure computation.* PhD thesis, 01 2018.

[38] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations.* Norwell, MA, USA: Kluwer Academic Publishers, 1994.

[39] M. D. Ercegovac and T. Lang, *Digital Arithmetic.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2003.

[40] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, pp. 21–29, Mar 2018.

APPENDICES

APPENDIX A

8 BIT EXAMPLE OF MOD WITHOUT MOD ALGORITHM

```
#################### To run
   ######################################
#  To run this program you can change the x and y values
# to your hearts content and then run the progam by using
#        `  python3 fastmod.py  `
#  This should output the amount of adds and if the modulo
    succeeded
# then it should also print the modulo from python's built
    in function
# and also the modulo calculated with the implemented
   function.
#
# Sidenote: The number should work regardless of x and y
   as it is 0 extended
#       during the program but this means that if you
   change the number that you
#       are moduloing by or the "mod" variable then it
   will not run as
#       the zero extension will no longer be the correct
   length
#
   ####################################################################




#Calculate the multiplied value
z = 0x71f3

#Calculate the P-384 modulo value
mod = 0x8a
#mod = 0xe0
modlength = mod.bit_length()
#print(bin(mod))

#Force strZ to correct size and get its proper length
strZ = bin(z).strip('0b').zfill(16)
lenZ = len(strZ)

#Split into top and bottom
```

```
topZ, botZ = int(strZ[:int(len(strZ)/2)],2), int(strZ[int(
   len(strZ)/2):],2)
print(hex(topZ))
print(hex(botZ))

#Set the end variable to the top half in bits of the
   multiplied number
result = topZ
#This is the value that you add to result when an overflow
    occurs
modVal = (2**mod.bit_length() % mod)
print("Modval: " + str(hex(modVal)))

#Initialize shift counter we basically want to shift an
   equal amount of times
#To the radix of the number we are moduloing by. As this
   modulo number will be
#static we can make a very ASIC style modulo module that
   has the values for
#modulo "hard coded" as modulo value / radix isn't dynamic
   . We right shift until
# there is a 1 in the "overflow" slot and then we get rid
   of the 1 in that position
# and add modVal for each overflow
#The rest of this is based on https://eprint.iacr.org
   /2014/755.pdf
shiftcnt = 0
addcnt = 0
print("Start Shifting")
while(shiftcnt < modlength):
  if(result.bit_length() <= mod.bit_length()):
    result = result << 1
    shiftcnt = shiftcnt + 1
  else:
    result = int(bin(result)[3:],2)
    result = result + modVal
    addcnt = addcnt+1
    print(str(hex(result)))
print("The shifting resulted in  about " + str(shiftcnt) +
    " cycles")
print("Result after shift and add : " + hex(result))
#Add the least significant half to the result
print(hex(result))
result = result + botZ
print(hex(result))

#Subtract the P384 value out of the result until the
   result is less than the P384mod
while(result >= mod):
  result = result - mod

#Check for overflow one last time
if(result.bit_length() > mod.bit_length()):
  print("Add\n______")
  print(hex(result))
  result = int(bin(result)[3:],2)
  result = result + modVal

#Check if the result matches up with python's inbuilt
   modulo operation.
if(result == (z%mod)):
  print("Modulo operation matches with python's modulo
     !!!!!!")
  print("Python's Modulo    : " + hex(z%mod))
  print("Calculated Modulo : " + hex(result))
else:
  print("Modulo operation did not match with python's
     modulo :(")
}
```

APPENDIX B

384 Bit Python Example of Mod Without Mod Algorithm

```
#################### To run
   #####################################
#  To run this program you can change the x and y values
# to your hearts content and then run the progam by using
#                ‘  python3 fastmod.py  ‘
#  This should output the amount of adds and if the modulo
    succeeded
# then it should also print the modulo from python's built
    in function
# and also the modulo calculated with the implemented
   function.
#
#   Sidenote: The number should work regardless of x and y
    as it is 0 extended
#             during the program but this means that if
   you change the number that you
#             are moduloing by or the "mod" variable then
   it will not run as
#             the zero extension will no longer be the
   correct length
#
   ##################################################################



# initialize z as a 768 bit value like we would see out of
    our multiplier
x = int("
   aa87ca22be8b05378eb1c7ffff20ad746e1d3b628ba79b9859f741e082542a38550
   ",16)
y = int("3617
   de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da311ffff0b8c00a60b1c
   ", 16)

#Calculate the multiplied value
z = x*y
print(hex(z))
#Calculate the P-384 modulo value
mod = ( 2**384 - 2**128 - 2**96 + 2**32 - 1 )
print(hex(mod))
modlength = mod.bit_length()
#print(bin(mod))
```

```
#Force strZ to correct size and get its proper length
strZ = bin(z).strip('0b').zfill(768)
lenZ = len(strZ)

#Split into top and bottom
topZ, botZ = int(strZ[:int(len(strZ)/2)],2), int(strZ[int(
   len(strZ)/2):],2)


#Set the end variable to the top half in bits of the
   multiplied number
result = topZ
#This is the value that you add to result when an overflow
    occurs
modVal = (2**mod.bit_length() % mod)
print(hex(modVal))

#Initialize shift counter we basically want to shift an
   equal amount of times
#To the radix of the number we are moduloing by. As this
   modulo number will be
#static we can make a very ASIC style modulo module that
   has the values for
#modulo "hard coded" as modulo value / radix isn't dynamic
   . We right shift until
# there is a 1 in the "overflow" slot and then we get rid
   of the 1 in that position
# and add modVal for each overflow
#The rest of this is based on https://eprint.iacr.org
   /2014/755.pdf
shiftcnt = 0
addcnt = 0
while(shiftcnt < modlength):
    if(result.bit_length() <= mod.bit_length()):
        result = result << 1
        shiftcnt = shiftcnt + 1
    else:
        result = int(bin(result)[3:],2)
        result = result + modVal
        addcnt = addcnt+1
print("The shifting resulted in " + str(addcnt) + " adds")


#Add the least significant half to the result
result = result + botZ

#Subtract the P384 value out of the result until the
   result is less than the P384mod
while(result >= mod):
    result = result - mod

#Check for overflow one last time
if(result.bit_length() > mod.bit_length()):
    result = int(bin(result)[3:],2)
    result = result + modVal

#Check if the result matches up with python's inbuilt
   modulo operation.
if(result == (z%mod)):
    print("Modulo operation matches with python's modulo
       !!!!!!")
    print("Python's Modulo    : " + hex(z%mod))
    print("Calculated Modulo : " + hex(result))
else:
    print("Modulo operation did not match with python's
       modulo :(")
}
```

APPENDIX C

HDL FOR 384-bit REDUCER

```
module top;
  reg clk,reset,start,shift_done;
  reg [767:0] z;
  reg [383:0] result, mod;
  reg [383:0] modVal;
  //load_z is the flag from the fsm that tells the mnd
     module to load a new value
  reg load_z;

  //The done signal is asserted when the fsm has fully
     cycled
  reg done;

  //Count_decrement is a signal from mnd to signal the
     counter in the FSM to countdown
  reg  count_decrement;

  //subNeg is 1 when the subtraction portion subtracts
     resulting in a negative number
  reg subNeg;

  /* subFlag selects between the registered input and
     previous stage
   * input into the subtraction portion
   */
  reg subFlag;

  //Test vector signals
  reg [1919:0] kmem [100000:0];
  reg [100:0] tvnum;
  reg [8:0] errors;
  reg [383:0] correctResult;
    logic [9:0] countAmt;
    logic [384:0] shAdd_reg;
  integer i;

  /* The mnd module is the primary arithmetic module to
     accomplish
   * modulo through the use of primarily shifting and
      adding
   */
  mnd384 mnd(.clk(clk),
        .reset(reset),
        .load_z(load_z),
        .shift_done(shift_done),
        .z(z),
        .result(result),
        .mod(mod),
        .modVal(modVal),
        .count_decrement(count_decrement),
        .subNeg(subNeg),
        .subFlag(subFlag),
              .countAmt(countAmt),
              .shAdd_reg(shAdd_reg));
  /* The fsm module is used to control the mnd module and
   * sets the flags and keeps track of the cycles in the
   * mnd module
   */
  fsm384 fsm(.start(start),
       .reset(reset),
       .clk(clk),
       .shift_done(shift_done),
       .count_decrement(count_decrement),
       .load_z(load_z),
       .done(done),
       .subNeg(subNeg),
       .subFlag(subFlag),
             .countAmt(countAmt),
             .shAdd_reg(shAdd_reg));

  //Run the clk signal
    initial
      begin
      i = 0;
        clk = 1'b1;
        forever #5 clk = ~clk;
      end


/* Read in test vectors to kmem, you can uncomment
 * the appropriate readmemh line to change between
 * tv.txt         : 10,000 test vectors with random modulo
   values
 * tv_ecp384.txt : 10,000 ecp384 test vectors
 * tv_ecp256.txt : 10,000 ecp256 test vectors
 */
initial
  begin
  //Uncomment one of these lines to change test vectors
  //$readmemh("tv/tv.txt", kmem);
  $readmemh("tv/tv_ecp384.txt",kmem);
  //$readmemh("tv/tv_ecp256.txt",kmem);

  //Initiate device values
  tvnum =1'b0;
  #20 reset = 0;
  #10 reset = 1;
  #20 reset = 0;
  #10 start = 0;
  #20 start = 1;
  modVal = {kmem[tvnum][1919:1536]};
  mod = {kmem[tvnum][1535:1152]};
  z = {kmem[tvnum][1151:384]};
  correctResult = {kmem[tvnum][383:0]};
  errors=0;
  end



always @(posedge clk)
  begin
    if(start)
  begin
    i = i +1;
  end
  if(done)
    begin
    //If incorrect result then notify through console
    if(correctResult == result) begin end
    else
      begin
      $display("Error in vector %d",tvnum+1);
      errors = errors+1;
      end

    //Increment test vector value and set new input values
    tvnum = tvnum + 1;
    modVal = {kmem[tvnum][1919:1536]};
    mod = {kmem[tvnum][1535:1152]};
    z = {kmem[tvnum][1151:384]};

    //Grab the correct result from test vectors
    correctResult = {kmem[tvnum][383:0]};

    //Reset device with new input parameters
    start = 0;
    #100
    #20 reset = 1;
    #20 reset = 0;
    #20 start = 1;
    end
  end

//Check if the test vector set is done and exit
always @(done)
  begin
  if(tvnum == 10000)
    begin
    $display("Completed all test vectors with %d errors",
        errors);
    $display("The average amount of cycles is %d", i/tvnum
        );
    $stop;
    $finish;
    end//if(tvnum == 100001)
  end
//Print the progress through the test vectors
always @(posedge done)
  begin
  if( tvnum % 100 == 0)
    begin
    $display("Completed %d test vectors with %d errors",
        tvnum,errors);
    end //if(tvnum % 100 == 0)
  end

endmodule

/*
This file contains the new mod module that does not
use division but rather a shift and add methodology
*/
module mnd384
    ( input logic clk, reset, load_z, shift_done, subFlag,
      input logic  [767:0] z,
      input logic  [383:0] mod,modVal,
      output logic [383:0]  result,
          output logic [384:0] shAdd_reg,
          input logic [9:0] countAmt,
      output logic count_decrement,subNeg);

    wire [384:0] shMuxed;
    wire [384:0] shadd_result;
    wire [385:0] subVal;
    wire [385:0] postSub_reg;
    wire [385:0] aIn;
    wire [385:0] a;
    wire [385:0] postSub;
    wire [384:0] subAdd;
    wire [384:0] shift_res;
    wire [384:0] add_res;
    wire [383:0] addend;

        //LZD logic
        logic [384:0] LZDdec;
        logic [9:0] LZDenc;

    //Input mux to select between the top half of z and
       the result of the shift or add
    mux2 #(385) shMux(.d0(shadd_result),
               .d1({1'b0,z[767:384]}),
               .s(load_z),
               .y(shMuxed));
    //Register to keep the shifted result
    flopenr #(385) shaddReg(.d(shMuxed),
                .q(shAdd_reg),
                .clk(clk),
                .reset(reset),
                .en(~shift_done));

    //Shifter for the shift/adding operations
    assign shift_res = shAdd_reg << countAmt;

    //If current shadd_shAdd_reg has carry out then add to
        it otherwise do a possible shift and add
    //We then need to change shAdd_mux to select based on
       shift_res[8] | shAdd_reg[8]
    //Adder for adding modval when necessary
    mux2 #(384) addMux(.d0(shift_res[383:0]),
          .d1(shAdd_reg[383:0]),
          .s(shAdd_reg[384]),
          .y(addend));
    assign add_res = addend + modVal;

    //Mux to select between the output of the shifter and
       the output of the adder
    mux2 #(385) shaddMux(.d0(shift_res),
               .d1(add_res),
               .s(shift_res[384] | shAdd_reg[384]),
               .y(shadd_result));
    //Any time we select the shift result we want to
       decrement the FSM counter
    assign count_decrement = ~(shAdd_reg[384]);


    //Adder that is used once the shifting is done to add
       the bottom half back in
    assign aIn = shAdd_reg + z[383:0];
    mux2 #(386) aMux(.d0(aIn),
            .d1(postSub_reg),
            .s(subFlag),
            .y(a));
    //Subtractor to check if result is greater than mod
       and subtract once if it is
    assign subVal = a - mod;
    flopenr #(386) subReg(.clk(clk),
              .reset(reset),
              .d(postSub),
              .q(postSub_reg),
              .en(~subVal[385] |  ~subFlag ));
    //mux to select between the subtracted and non
       subtracted values
    //we need to pass out subVal[7] in order for the FSM
```

```
        to signal to do the subtract state
      //and disable subReg whenever the value goes negative.
      assign subNeg = subVal[385];
      mux2 #(386) subMux(.d0(subVal),
            .d1(a),
            .s(subVal[385]),
            .y(postSub));
      //Adder to add modval one last time if necesary
      assign subAdd = postSub + modVal;
      //multiplexer to choose between the sub output and the
          last modval add
      mux2 #(384) resultMux(.d0(postSub_reg[383:0]),
                .d1(subAdd[383:0]),
                .s(postSub[384]),
                .y(result));

endmodule

module fsm384
    (input logic start, reset, clk,count_decrement, subNeg
      ,
        output logic [9:0] countAmt,
        input logic [384:0] shAdd_reg,
     output logic shift_done ,load_z, done, subFlag);

  //We need a signal to reset the counter to the correct
     amount when we hit state 2
  reg reset_count;

  logic [3:0]   CURRENT_STATE;
  logic [3:0]   NEXT_STATE;

  parameter [2:0]
   S0=4'h0, S1=4'h1, S2=4'h2,
   S3=4'h3, S4=4'h4, S5=4'h5;

  always @(posedge clk)
      begin
    if(reset == 1'b1)
      CURRENT_STATE <= S0;
    else
      CURRENT_STATE <= NEXT_STATE;
    end

  always_comb
    begin
      case(CURRENT_STATE)

      //Begin State 0
      S0:
        begin //S0
          reset_count = 1'b1;
          load_z = 1'b0;
        done = 1'b0;
  subFlag = 1'b0;
  if(start == 1'b0)
    begin
    NEXT_STATE = S1;
    end
  else
    begin
    NEXT_STATE = S0;
    end
  end //S0

//Begin State 1 load z and get ready to shift
S1:
  begin //S1
    reset_count = 1'b1;
    load_z = 1'b1;
    done = 1'b0;
  subFlag = 1'b0;
  if(start == 1'b1)
    begin
      reset_count = 1'b1;
    NEXT_STATE = S2;
    end
  end //S1

//Begin State 2 or the state that does all the
   shifts
S2:
  begin //S2
    reset_count = 1'b0;
  load_z = 1'b0;
  done = 1'b0;
  subFlag = 1'b0;
  if(shift_done == 1'b1)
    begin
    NEXT_STATE = S3;
    end
  else
    begin
    NEXT_STATE = S2;
    end
  end //S2

//Begin State 3 which is where the first subtract
   and lower half add
S3:
  begin //S3
    reset_count = 1'b0;
      load_z = 1'b0;
  done = 1'b0;
  subFlag = 1'b0;
  NEXT_STATE = S4;
  end //S3
```

```
      //Begin State 4 which is where the remaining mod
         subtractions take place
      S4:
        begin //S4
          reset_count = 1'b0;
          load_z = 1'b0;
        done = 1'b0;
        subFlag = 1'b1;
        if(subNeg == 1'b1)
          begin
          NEXT_STATE = S5;
          end
        else
          begin
          NEXT_STATE = S4;
          end
        end //S4

      //Begin State 5 where we hold the value and assert
         done
      S5:
        begin //S5
          reset_count = 1'b0;
          load_z = 1'b0;
        done = 1'b1;
        subFlag = 1'b0;
        NEXT_STATE = S0;
        end //S5
      endcase; //case(CURRENT_STATE)

    end//always @(CURRENT_STATE or start)

  counter384 countdown (.clk(clk),
            .count_decrement(count_decrement),
                        .countAmt(countAmt),
            .shift_done(shift_done),
            .reset_count(reset_count),
            .reset(reset),
                        .shAdd_reg(shAdd_reg),
                        .load_z(load_z));

endmodule
```

APPENDIX D

HDL FOR 384-bit Parallel REDUCER

```
module fsm384
    (input logic start, reset, clk,count_decrement, subNeg
      ,
     input logic [384:0] shAdd_reg,
     output logic [7:0] shiftAmt,
     output logic shift_done ,load_z, done, subFlag);

  //We need a signal to reset the counter to the correct
     amount when we hit state 2
  reg reset_count;

  logic [3:0]   CURRENT_STATE;
  logic [3:0]   NEXT_STATE;

  parameter [2:0]
   S0=4'h0, S1=4'h1, S2=4'h2,
   S3=4'h3, S4=4'h4, S5=4'h5;

  always @(posedge clk)
      begin
    if(reset == 1'b1)
      CURRENT_STATE <= S0;
    else
      CURRENT_STATE <= NEXT_STATE;
    end

  always_comb
    begin
      case(CURRENT_STATE)

      //Begin State 0
      S0:
        begin //S0
          reset_count = 1'b0;
          load_z = 1'b0;
        done = 1'b0;
        subFlag = 1'b0;
        if(start == 1'b0)
          begin
          NEXT_STATE = S1;
          end
        else
          begin
          NEXT_STATE = S0;
          end
        end //S0

      //Begin State 1 load z and get ready to shift
      S1:
        begin //S1
          reset_count = 1'b0;
          load_z = 1'b1;
          done = 1'b0;
        subFlag = 1'b0;
        if(start == 1'b1)
          begin
            reset_count = 1'b1;
          NEXT_STATE = S2;
          end
        end //S1

      //Begin State 2 or the state that does all the
         shifts
      S2:
        begin //S2
          reset_count = 1'b0;
        load_z = 1'b0;
        done = 1'b0;
        subFlag = 1'b0;
        if(shift_done == 1'b1)
          begin
          NEXT_STATE = S3;
          end
        else
          begin
          NEXT_STATE = S2;
          end
        end //S2

      //Begin State 3 which is where the first subtract
         and lower half add
      S3:
        begin //S3
          reset_count = 1'b0;
            load_z = 1'b0;
        done = 1'b0;
        subFlag = 1'b0;
        NEXT_STATE = S4;
        end //S3


      //Begin State 4 which is where the remaining mod
         subtractions take place
      S4:
        begin //S4
          reset_count = 1'b0;
```

```
          load_z = 1'b0;
        done = 1'b0;
        subFlag = 1'b1;
        if(subNeg == 1'b1)
          begin
          NEXT_STATE = S5;
          end
        else
          begin
          NEXT_STATE = S4;
          end
        end //S4

      //Begin State 5 where we hold the value and assert
         done
      S5:
        begin //S5
          reset_count = 1'b0;
          load_z = 1'b0;
        done = 1'b1;
        subFlag = 1'b0;
        NEXT_STATE = S0;
        end //S5
      endcase; //case(CURRENT_STATE)

    end//always @(CURRENT_STATE or start)

  counter384 countdown (.clk(clk),
            .count_decrement(count_decrement),
            .shift_done(shift_done),
            .reset_count(reset_count),
            .reset(reset),
            .shiftAmt(shiftAmt),
            .shAdd_reg(shAdd_reg));

endmodule



module counter384 (clk, count_decrement, reset_count,
   shift_done, reset, shiftAmt, shAdd_reg);
input clk, reset_count, count_decrement, reset;
input [384:0] shAdd_reg;
output reg [7:0] shiftAmt;
output reg shift_done;

  reg [8:0] count;
  wire LZDdec;
  wire [7:0] LZDenc;

  lz256 lzd(.B({1'b0,shAdd_reg[384:130]}),
        .ZP(LZDenc),
        .ZV(LZDdec));
  always @(*)
    begin
    if(LZDenc > count)
      begin
      shiftAmt = count;
      end
    else
      begin
      shiftAmt = LZDenc;
      end

    end


  always @(posedge clk)
    begin
      if(reset == 1'b1)
      begin
        shiftAmt = 0;
      shift_done = 1'b0;
      count = 9'h180;
      end

    if(reset_count == 1'b1)
      begin
        //Set this to radix
      count = 9'h180;
      end //if(load == 1'b1)
    else if((count_decrement & ~shift_done) == 1'b1)
      begin
      count = count - shiftAmt;
      end//else if update == 1'b1)
    else
      begin
      count = count;
      end

    if(count == 9'h000)
      begin
      shift_done = 1'b1;
      end
    else
      begin
      shift_done = 1'b0;
      end

    end
endmodule


/*
This file contains the new mod module that does not
use division but rather a shift and add methodology
```

```
*/
module mnd384
    ( input logic clk, reset, load_z, shift_done, subFlag,
      input logic  [767:0] z,
      input logic  [383:0] mod,modVal,modVal2,
      input logic  [384:0] modValC,
      input logic  [7:0] shiftAmt,
      output logic [384:0] shAdd_reg,
      output logic [383:0]  result,
      output logic count_decrement,subNeg);

    wire [384:0] shMuxed;
    wire [384:0] shadd_result;
    wire [385:0] subVal;
    wire [385:0] postSub_reg;
    wire [385:0] aIn;
    wire [385:0] a;
    wire [385:0] postSub;
    wire [384:0] subAdd;
    wire [383:0] addend;


    //Parallelization wires
    wire [384:0] add_res;
    wire [385:0] shift_res;

    wire [385:0] addP1;
    wire [385:0] addP2;
    wire [385:0] p1p2Res;

    wire [385:0] addP3;

    //Input mux to select between the top half of z and
       the result of the shift or add
    mux2 #(385) shMux(.d0(shadd_result),
               .d1({1'b0,z[767:384]}),
               .s(load_z),
               .y(shMuxed));
    //Register to keep the shifted result
    flopenr #(385) shaddReg(.d(shMuxed),
                .q(shAdd_reg),
                .clk(clk),
                .reset(reset),
                .en(~shift_done));
    //Shifter for the shift/adding operations
    assign shift_res = shAdd_reg << shiftAmt;



    //If current shadd_shAdd_reg has carry out then add to
        it otherwise do a possible shift and add
    //We then need to change shAdd_mux to select based on
```

```
   shift_res[8] | shAdd_reg[8]
//Adder for adding modval when necessary
mux2 #(384) addMux(.d0(shift_res[383:0]),
      .d1(shAdd_reg[383:0]),
      .s(shAdd_reg[384]),
      .y(addend));

//Logic for the parallelization
assign addP1 = addend + modVal;
assign addP2 = addend + modVal2;
assign addP3 = addend + modValC;

assign p1p2Sel   = (~shAdd_reg[384]) & (shift_res
   [385]);
assign p1p2p3Sel = (~shAdd_reg[384]) & (shift_res[385]
    & shift_res[384]);

mux2 #(386) p1p2mux(.d0(addP1),
          .d1(addP2),
          .s(p1p2Sel),
          .y(p1p2Res));
mux2 #(385) p1p2p3mux(.d0(p1p2Res[384:0]),
            .d1(addP3[384:0]),
            .s(p1p2p3Sel),
            .y(add_res));

//Mux to select between the output of the shifter and
   the output of the adder
mux2 #(385) shaddMux(.d0(shift_res[384:0]),
           .d1(add_res),
           .s(shift_res[385]  |shift_res[384] |
              shAdd_reg[384]),
           .y(shadd_result));
//Any time we select the shift result we want to
   decrement the FSM counter
assign count_decrement = ~(shAdd_reg[384]);


//Adder that is used once the shifting is done to add
   the bottom half back in
assign aIn = shAdd_reg + z[383:0];
mux2 #(386) aMux(.d0(aIn),
        .d1(postSub_reg),
        .s(subFlag),
        .y(a));
//Subtractor to check if result is greater than mod
   and subtract once if it is
assign subVal = a - mod;
flopenr #(386) subReg(.clk(clk),
          .reset(reset),
          .d(postSub),
          .q(postSub_reg),
              .en(~subVal[385] |  ~subFlag ));
    //mux to select between the subtracted and non
       subtracted values
    //we need to pass out subVal[7] in order for the FSM
       to signal to do the subtract state
    //and disable subReg whenever the value goes negative.
    assign subNeg = subVal[385];
    mux2 #(386) subMux(.d0(subVal),
          .d1(a),
          .s(subVal[385]),
          .y(postSub));
    //Adder to add modval one last time if necesary
    assign subAdd = postSub + modVal;
    //multiplexer to choose between the sub output and the
        last modval add
    mux2 #(384) resultMux(.d0(postSub_reg[383:0]),
              .d1(subAdd[383:0]),
              .s(postSub[384]),
              .y(result));

endmodule
```

VITA

Ryan Swann

Candidate for the Degree of

Master of Science

Thesis: AN EXTENSION AND IMPLEMENTATION OF THE MOD WITHOUT MOD ALGORITHM TO EFFICIENTLY COMPUTE THE MODULUS OF A NUMBER IN HARDWARE

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2020.

Completed the requirements for the Bachelor of Science in Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in 2019.

Experience:

Graduate Research Assistant - VLSI Computer Architecture Research Group OSU
May 2019 - December 2020
Undergraduate Research Assistant - Visual Computing and Image Processing Lab OSU
August 2017 - May 2019