Efficient Hardware Architectures for Modular Multiplication

by

David Narh Amanor

A Thesis

submitted to

The University of Applied Sciences Offenburg, Germany

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Communication and Media Engineering

February, 2005

Approved:

Prof. Dr. Angelika Erhardt
Thesis Supervisor

Prof. Dr. Christof Paar
Thesis Supervisor

Declaration of Authorship

David Narh Amanor

February, 2005

Preface

This thesis describes the research which I conducted while completing my graduate work at the University of Applied Sciences Offenburg, Germany.
The work produced scalable hardware implementations of existing and newly proposed algorithms for performing modular multiplication.

The work presented can be instrumental in generating interest in the hardware implementation of emerging algorithms for doing faster modular multiplication, and can also be used in future research projects at the University of Applied Sciences Offenburg, Germany, and elsewhere.

Of particular interest is the integration of the new architectures into existing public-key cryptosystems such as RSA, DSA, and ECC to speed up the arithmetic.

I wish to thank the following people for their unselfish support throughout the entire duration of this thesis.

I would like to thank my external advisor Prof. Christof Paar for providing me with all the tools and materials needed to conduct this research. I am particularly grateful to Dipl.-Ing. Jan Pelzl, who worked with me closely, and whose constant encouragement and advice gave me the energy to overcome several problems I encountered while working on this thesis.

I wish to express my deepest gratitude to my supervisor Prof. Angelika Erhardt for being in constant touch with me and for all the help and advice she gave throughout all stages of the thesis. If it was not for Prof. Erhardt, I would not have had the opportunity of doing this thesis work and therefore, I would have missed out on a very rewarding experience.

I am also grateful to Dipl.-Ing. Viktor Buminov and Prof. Manfred Schimmler, whose newly proposed algorithms and corresponding architectures form the basis of my thesis work and provide the necessary theoretical material for understanding the algorithms presented in this thesis.

Finally, I would like to thank my brother, Mr. Samuel Kwesi Amanor, my friend and Pastor, Josiah Kwofie, Mr. Samuel Siaw Nartey and Mr. Csaba Karasz for their diverse support which enabled me to undertake my thesis work in Bochum.

Abstract

Modular multiplication is a core operation in many public-key cryptosystems, e.g., RSA, Diffie-Hellman key agreement (DH), ElGamal, and ECC. The Montgomery multiplication algorithm [2] is considered to be the fastest algorithm to compute X**Y mod M* in computers when the values of *X*, *Y* and *M* are large.

Recently, two new algorithms for modular multiplication and their corresponding architectures were proposed in [1]. These algorithms are optimizations of the Montgomery multiplication algorithm [2] and interleaved modular multiplication algorithm [3].

In this thesis, software (Java) and hardware (VHDL) implementations of the existing and newly proposed algorithms and their corresponding architectures for performing modular multiplication have been done. In summary, three different multipliers for 32, 64, 128, 256, 512, and 1024 bits were implemented, simulated, and synthesized for a Xilinx FPGA. The implementations are scalable to any precision of the input variables *X*, *Y* and *M*.

This thesis also evaluated the performance of the multipliers in [1] by a thorough comparison of the architectures on the basis of the area-time product.

This thesis finally shows that the newly optimized algorithms and their corresponding architectures in [1] require minimum hardware resources and offer faster speed of computation compared to multipliers with the original Montgomery algorithm.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The rising growth of data communication and electronic transactions over the internet has made security to become the most important issue over the network.
To provide modern security features, public-key cryptosystems are used. The widely used algorithms for public-key cryptosystems are RSA, Diffie-Hellman key agreement (DH), the digital signature algorithm (DSA) and systems based on elliptic curve cryptography (ECC). All these algorithms have one thing in common: they operate on very huge numbers (e.g. 160 to 2048 bits). Long word lengths are necessary to provide a sufficient amount of security, but also account for the computational cost of these algorithms.

By far, the most popular public-key scheme in use today is RSA [9]. The core operation for data encryption processing in RSA is modular exponentiation, which is done by a series of modular multiplications (i.e., *X*Y mod M*). This accounts for most of the complexity in terms of time and resources needed. Unfortunately, the large word length (e.g. 1024 or 2048 bits) makes the RSA system slow and difficult to implement. This gives reason to search for dedicated hardware solutions which compute the modular multiplications efficiently with minimum resources.

The Montgomery multiplication algorithm [2] is considered to be the fastest algorithm to compute X**Y mod M* in computers when the values of *X*, *Y* and *M* are large. Another efficient algorithm for modular multiplication is the interleaved modular multiplication algorithm [4].

In this thesis, two new algorithms for modular multiplication and their corresponding architectures which were proposed in [1] are implemented. These algorithms are optimisations of Montgomery multiplication and interleaved modular multiplication. They are optimised with respect to area and time complexity. In both algorithms the product of two *n* bit integers *X* and *Y* modulo *M* are computed by *n* iterations of a simple loop. Each loop consists of one single carry save addition, a comparison of constants, and a table lookup.

These new algorithms have been proved in [1] to speed-up the modular multiplication operation by at least a factor of two in comparison with all methods previously known.

The main advantages offered by these new algorithms are;

- faster computation time, and
- area requirements and resources for the implementation of their architectures in hardware are relatively small compared to the Montgomery multiplication algorithm presented in [1, Algorithm 1a and 1b].

1.2 Thesis Outline

Chapter 2 provides an overview of the existing algorithms and their corresponding architectures for performing modular multiplication. The necessary background knowledge which is required for understanding the algorithms, architectures, and concepts presented in the subsequent chapters is also explained. This chapter also discusses the complexity model which was used to compare the existing architectures with the newly proposed ones.

In Chapter 3, a description of the new algorithms for modular multiplication and their corresponding architectures are presented. The modifications that were applied to the existing algorithms to produce the new optimized versions are also explained in this chapter.

Chapter 4 covers issues on the software implementation of the algorithms presented in Chapters 2 and 3. The special classes in Java which were used in the implementation of the algorithms are mentioned. The testing of the new optimized algorithms presented in Chapter 3 using random generated input variables is also discussed.

The hardware modeling technique which was used in the implementation of the multipliers is explained in Chapter 5. In this chapter, the design capture of the architectures in VHDL is presented and the simulations of the VHDL implementations are also discussed. This chapter also discusses the target technology device and synthesis results. The state machine of the implemented multipliers is also presented in this chapter.

In Chapter 6, analysis and comparison of the implemented multipliers is given. The vital design statistics which were generated after place and route were tabulated and graphically represented in this chapter. Of prime importance in this chapter is the area – time (AT) analysis of the multipliers which is the complexity metric used for the comparison.

Chapter 7 concludes the thesis by setting out the facts and figures of the performance of the implemented multipliers. This chapter also itemizes a list of recommendations for further research.

Chapter 2

Existing Architectures for Modular Multiplication

2.1 Carry Save Adders and Redundant Representation

The core operation of most algorithms for modular multiplication is addition. There are several different methods for addition in hardware: carry ripple addition, carry select addition, carry look ahead addition and others [8]. The disadvantage of these methods is the carry propagation, which is directly proportional to the length of the operands. This is not a big problem for operands of size 32 or 64 bits but the typical operand size in cryptographic applications range from 160 to 2048 bits. The resulting delay has a significant influence on the time complexity of these adders.

The carry save adder seems to be the most cost effective adder for our application. Carry save addition is a method for an addition without carry propagation. It is simply a parallel ensemble of *n* full-adders without any horizontal connection. Its function is to add three *n*-bit integers *X*, *Y*, and *Z* to produce two integers *C* and *S* as results such that

$$C + S = X + Y + Z,$$

where *C* represents the carry and *S* the sum.

The i^{th} bit s_i of the sum *S* and the $(i + 1)^{st}$ bit c_{i+1} of carry *C* are calculated using the boolean equations

$$s_i = x_i \oplus y_i \oplus z_i$$
$$c_{i+1} = x_i y_i \vee x_i z_i \vee y_i z_i$$
$$c_0 = 0,$$

When carry save adders are used in an algorithm one uses a notation of the form

$(S, C) = X + Y + Z$

to indicate that two results are produced by the addition.

The results are now represented in two binary words, an n-bit word S and an $(n+1)$ bit word C. Of course, this representation is redundant in the sense that we can represent one value in several different ways. This redundant representation has the advantage that the arithmetic operations are fast, because there is no carry propagation. On the other hand, it brings to the fore one basic disadvantage of the carry save adder:

- It does not solve our problem of adding two integers to produce a single result. Rather, it adds three integers and produces two such that the sum of these two is equal to that of the three inputs. This method may not be suitable for applications which only require the normal addition.

2.2 Complexity Model

For comparison of different algorithms we need a complexity model that allows for a realistic evaluation of time and area requirements of the considered methods. In [1], the delay of a full adder (1 time unit) is taken as a reference for the time requirement and quantifies the delay of an access to a lookup table with the same time delay of 1 time unit. The area estimation is based on empirical studies in full-custom and semi-custom layouts for adders and storage elements: The area for 1 bit in a lookup table corresponds to 1 area unit. A register cell requires 4 area units per bit and a full adder requires 8 area units. These values provide a powerful and realistic model for evaluation of area and time for most algorithms for modular multiplication.

In this thesis, the percentage of configurable logic block slices occupied and the absolute time for computation are used to evaluate the algorithms. Other hardware resources such as total number of gates and number of flip-flops or latches required were also documented to provide a more practical and realistic evaluation of the algorithms in [1].

2.3 Montgomery Multiplication Algorithm

The Montgomery algorithm [1, Algorithm 1a] computes $P = (X*Y*(2^n)^{-1}) \bmod M$. The idea of Montgomery [2] is to keep the lengths of the intermediate results smaller than *n*+1 bits. This is achieved by interleaving the computations and additions of new partial products with divisions by 2; each of them reduces the bit-length of the intermediate result by one.

For a detailed treatment of the Montgomery algorithm, the reader is referred to [2] and [1].

The key concepts of the Montgomery algorithm [1, Algorithm 1b] are the following:

- Adding a multiple of *M* to the intermediate result does not change the value of the final result; because the result is computed modulo *M*. *M* is an odd number.
- After each addition in the inner loop the least significant bit (LSB) of the intermediate result is inspected. If it is 1, i.e., the intermediate result is odd, we add *M* to make it even. This even number can be divided by 2 without remainder. This division by 2 reduces the intermediate result to *n*+1 bits again.
- After *n* steps these divisions add up to one division by 2^n.

The Montgomery algorithm is very easy to implement since it operates least significant bit first and does not require any comparisons. A modification of Algorithm 1a with carry save adders is given in [1, Algorithm 1b]:

Algorithm 1a: Montgomery multiplication [1]

```
Inputs: $X, Y, M$ with $0 \le X, Y < M$
Output: $P = (X*Y(2^n)^{-1}) \bmod M$
$n$: number of bits in $X$;
$x_i$: $i^{th}$ bit of $X$;
$p_0$: LSB of $P$;
(1) $P := 0$;
(2) for $(i = 0; i < n; i++)$ {
(3)     $P := P + x_i*Y$;
(4)     $P := P + p_0*M$;
(5)     $P := P \text{ div } 2$; }
(6) if $(P \ge M)$ then $P := P-M$;
```

Algorithm 1b: Fast Montgomery multiplication [1]

```
Inputs: X, Y, M with $0 \leq X, Y < M$
Output: $P = (X*Y(2^n)^{-1}) \bmod M$
n: number of bits in X;
$x_i$: $i^{th}$ bit of X;
$s_0$: LSB of S;
(1) $S := 0;\ C := 0;$
(2) for $(i = 0;\ i < n;\ i++)$ {
(3)     $S,C := S + C + x_i*Y;$
(4)     $S,C := S + C + s_0*M;$
(5)     $S := S \text{ div } 2;\ C := C \text{ div } 2;$}
(6) $P := S + C;$
(7) if $(P \geq M)$ then $P := P-M;$
```

In this algorithm the delay of one pass through the loop is reduced from *O*(*n*) to *O*(1). This remarkable improvement of the propagation delay inside the loop of Algorithm 1b is due to the use of carry save adders to implement step (3) and (4) in Algorithm 1a.

Step (3) and (4) in Algorithm 1b represent carry save adders. *S* and *C* denote the sum and carry of the three input operands respectively.
Of course, the additions in step (6) and (7) are conventional additions. But since they are performed only once while the additions in the loop are performed *n* times this is subdominant with respect to the time complexity.

Figure 1 shows the architecture for the implementation of the loop of Algorithm 1b. The layout comprises of two carry save adders (CSA) and registers for storing the intermediate results of the sum and carry. The carry save adders are the dominant occupiers of area in hardware especially for very large values of *n* (e.g. $n \geq 1024$).

In Chapter 3, we shall see the changes that were made in [1] to reduce the number of carry save adders in Figure1 from 2 to 1, thereby saving considerable hardware space. However, these changes also brought about other area consuming blocks such as lookup tables for storing precomputed values before the start of the loop.

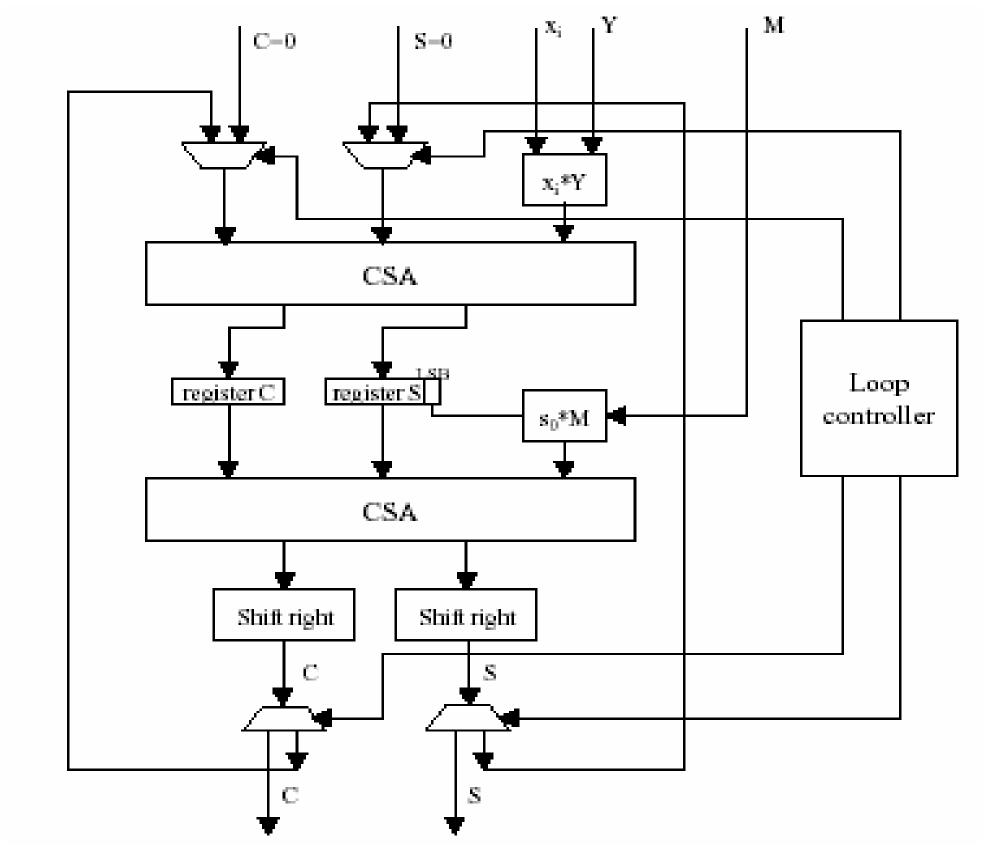


Fig. 1: Architecture of the loop of algorithm 1b [1].

There are various modifications to the Montgomery algorithm in [5], [6] and [7]. All these algorithms aimed at decreasing the operating time for faster system performance and reducing the chip area for practical hardware implementation.

2.4 Interleaved Modular Multiplication

Another well known algorithm for modular multiplication is the interleaved modular multiplication. The details of the method are sketched in [3, 4]. The idea is to interleave multiplication and reduction such that the intermediate results are kept as short as possible.

As shown in [1, Algorithm 2], the computation of *P* requires *n* steps and at each step we perform the following operations:

- A left shift: *2*P*
- A partial product computation: x_i* *Y*
- An addition: *2*P+* x_i* *Y*
- At most 2 subtractions:

  If (P ≥ M) Then P := P – M;
  If (P ≥ M) Then P := P – M;

The partial product computation and left shift operations are easily performed by using an array of AND gates and wiring respectively. The difficult task is the addition operation, which must be performed fast. This was done using carry save adders in [1, Algorithm 4], introducing only *O*(1) delay per step.

Algorithm 2: Standard interleaved modulo multiplication [1]

```
Inputs: X, Y, M with $0 \le X, Y < M$
Output: $P = X*Y \bmod M$
n: number of bits in X;
$x_i$: $i^{th}$ bit of X;
(1) $P := 0$;
(2) for $(i = n-1;\ i \ge 0;\ i--)$ {
(3)     $P := 2*P$;
(4)     $I := x_i*Y$;
(5)     $P := P + I$;
(6) if $(P \ge M)$ then $P := P-M$;
(7) if $(P \ge M)$ then $P := P-M$; }
```

The main advantages of Algorithm 2 compared to the separated multiplication and division are the following:

- Only one loop is required for the whole operation.
- The intermediate results are never any longer than *n*+2 bits (thus reducing the area for registers and full adders).

But there are some disadvantages as well:

- The algorithm requires three additions with carry propagation in steps (5), (6) and (7).

- In order to perform the comparisons in steps (4) and (5), the preceding additions have to be completed. This is important for the latency because the operands are large and, therefore, the carry propagation has a significant influence on the latency.

- The comparison in step (6) and (7) also requires the inspection of the full bit lengths of the operands in the worst case. In contrast to addition, the comparison is performed MSB first. Therefore, these two operations cannot be pipelined without delay.

Many researchers have tried to address these problems, but the only solution with a constant delay in the loop is the one of [8], which has an AT- complexity of $156n^2$.

In [1], a different approach is presented which reduces the AT-complexity for modular multiplication considerably. In Chapter 3, this new optimized algorithm is presented and discussed.

Chapter 3

New Architectures for Modular Multiplication

The detailed treatment of the new algorithms and their corresponding architectures presented in this chapter can be found in [1]. In this chapter, a summary of these algorithms and architectures is given. They have been designed to meet the core requirements of most modern devices: small chip area and low power consumption.

3.1 Faster Montgomery Algorithm

In Figure 1, the layout for the implementation of the loop of Algorithm 1b consists of two carry save adders. For large wordsizes (e.g. n = 1024 or higher), this would require considerable hardware resources to implement the architecture of Algorithm 1b. The motivation behind this optimized algorithm is that of reducing the chip area for practical hardware implementation of Algorithm 1b. This is possible if we can precompute the four possible values to be added to the intermediate result within the loop of Algorithm 1b, thereby reducing the number of carry save adders from 2 to 1. There are four possible scenarios:

- if the sum of the old values of S and C is an even number, and if the actual bit x_i of X is 0, then we add 0 before we perform the reduction of S and C by division by 2.

- if the sum of the old values of S and C is an odd number, and if the actual bit x_i of X is 0, then we must add M to make the intermediate result even. Afterwards, we divide S and C by 2.

- if the sum of the old values of S and C is an even number, and if the actual bit x_i of X is 1, but the increment x_i*Y is even, too, then we do not need to add M to make the intermediate result even. Thus, in the loop we add Y before we perform the reduction of S and C by division by 2. The same action is necessary if the sum of S and C is odd, and if the actual bit x_i of X is 1 and Y is odd as well. In this case, S+C+Y is an even number, too.

- if the sum of the old values of *S* and *C* is odd, the actual bit x_i of X is 1, but the increment $x_i * Y$ is even, then we must add *Y* and *M* to make the intermediate result even. Thus, in the loop we add *Y+M* before we perform the reduction of *S* and *C* by division by 2.

The same action is necessary if the sum of *S* and *C* is even, and the actual bit x_i of X is 1, and *Y* is odd. In this case, *S+C+Y+M* is an even number, too.

The computation of *Y+M* can be done prior to the loop. This saves one of the two additions which are replaced by the choice of the right operand to be added to the old values of *S* and *C*. Algorithm 3 is a modification of Montgomery's method which takes advantage of this idea.

The advantage of Algorithm 3 in comparison to Algorithm 1 can be seen in the implementation of the loop of Algorithm 3 in Figure 2. The possible values of *I* are stored in a lookup-table, which is addressed by the actual values of x_i, y_0, s_0 and c_0. The operations in the loop are now reduced to one table lookup and one carry save addition. Both these activities can be performed concurrently. Note that the shift right operations that implement the division by 2 can be done by routing.

Algorithm 3: Faster Montgomery multiplication [1]

```
Inputs: $X, Y, M$ with $0 \le X, Y < M$
Output: $P = (X*Y(2^n)^{-1}) \bmod M$
n: number of bits in $X$;
$x_i$: $i^{th}$ bit of $X$;
$s_0$: LSB of $S$,   $c_0$: LSB of $C$,   $y_0$: LSB of $Y$;
R: precomputed value of $Y + M$;
(1) $S := 0;\ C := 0;$
(2) for $(i = 0; i < n; i++)$ {
(3)     if $((s_0 = c_0)$ and not $x_i)$ then $I := 0;$
(4)     if $((s_0 \ne c_0)$ and not $x_i)$ then $I := M;$
(5)     if $(not(s_0 \oplus c_0 \oplus y_0)$ and $x_i)$ then $I := Y;$
(6)     if $((s_0 \oplus c_0 \oplus y_0)$ and $x_i)$ then $I := R;$
(7)     $S,C := S + C + I;$
(8)     $S := S$ div $2;\ C := C$ div $2;$}
(9) $P := S + C;$
(10) if $(P \ge M)$ then $P := P-M;$
```

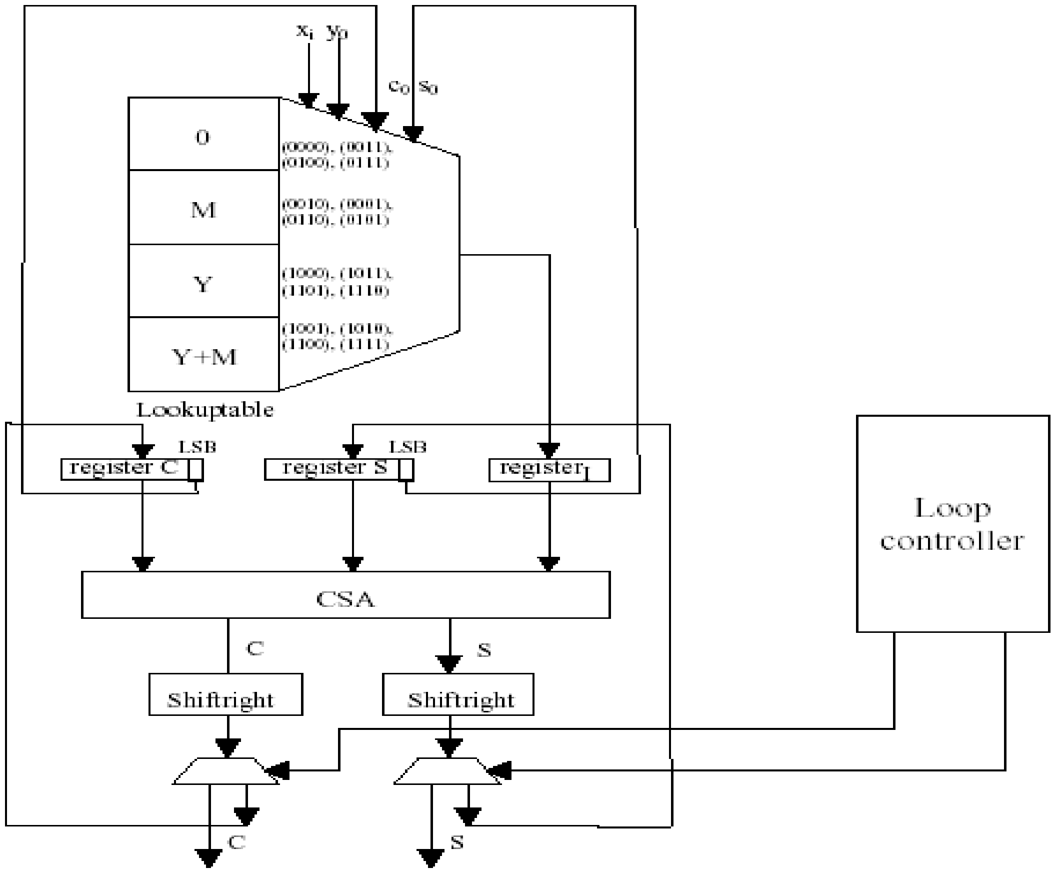


Fig. 2: Architecture of Algorithm 3 [1]

In [1], the proof of Algorithm 3 is presented and the assumptions which were made in arriving at an Area-Time (AT) complexity of $96n^2$ are shown.

3.2 Optimized Interleaved Algorithm

The new algorithm [1, Algorithm 4] is an optimisation of the interleaved modular multiplication [1, Algorithm 2]. In [1], four details of Algorithm 2 were modified in order to overcome the problems mentioned in Chapter 2:

- The intermediate results are no longer compared to *M* (as in steps (6) and (7) of Algorithm 2). Rather, a comparison to $k*2^n$ (k=0... 6) is performed which can be done in constant time. This comparison is done implicitly in the mod-operation in step (13) of Algorithm 4.

- Subtractions in steps (6), (7) of Algorithm 2 are replaced by one subtraction of $k*2^n$ which can be done in constant time by bit masking.
- Next, the value of $k*2^n$ mod M is added in order to generate the correct intermediate result (step (12) of Algorithm 4).
- Finally, carry save adders are used to perform the additions inside the loop, thereby reducing the latency to a constant. The intermediate results are in redundant form, coded in two words *S* and *C* instead of generated one word *P*.

These changes made by the authors in [1] led to Algorithm 4, which looks more complicated than Algorithm 2. Its main advantage is the fact that all the computations in the loop can be performed in constant time. Hence, the time complexity of the whole algorithm is reduced to O(n), provided the values of $k*2^n$ mod M are precomputed before execution of the loop.

Algorithm 4: Modular multiplication using carry save addition [1]

```
Inputs: X, Y, M with $0 \le X, Y < M$
Output: $P = X*Y \bmod M$
n: number of bits in X;
$x_i$: $i^{th}$ bit of X;
(1) $S := 0;\ C := 0;\ A := 0;$
(2) for ($i = n-1;\ i \ge 0;\ i--$) {
(3)     $S := S \bmod 2^n;$
(4)     $C := C \bmod 2^n;$
(5)     $S := 2*S;$
(6)     $C := 2*C;$
(7)     $A := 2*A;$
(8)     $I := x_i*Y;$
(9)     $(S,C) := CSA(S, C, I);$
(10)    $(S,C) := CSA(S, C, A);$
(11)    $A := (2*S_{n+1} + S_n + 2*C_{n+1} + C_n)*2^n \bmod M;$}
(12) $P := (S + C) \bmod M;$
```

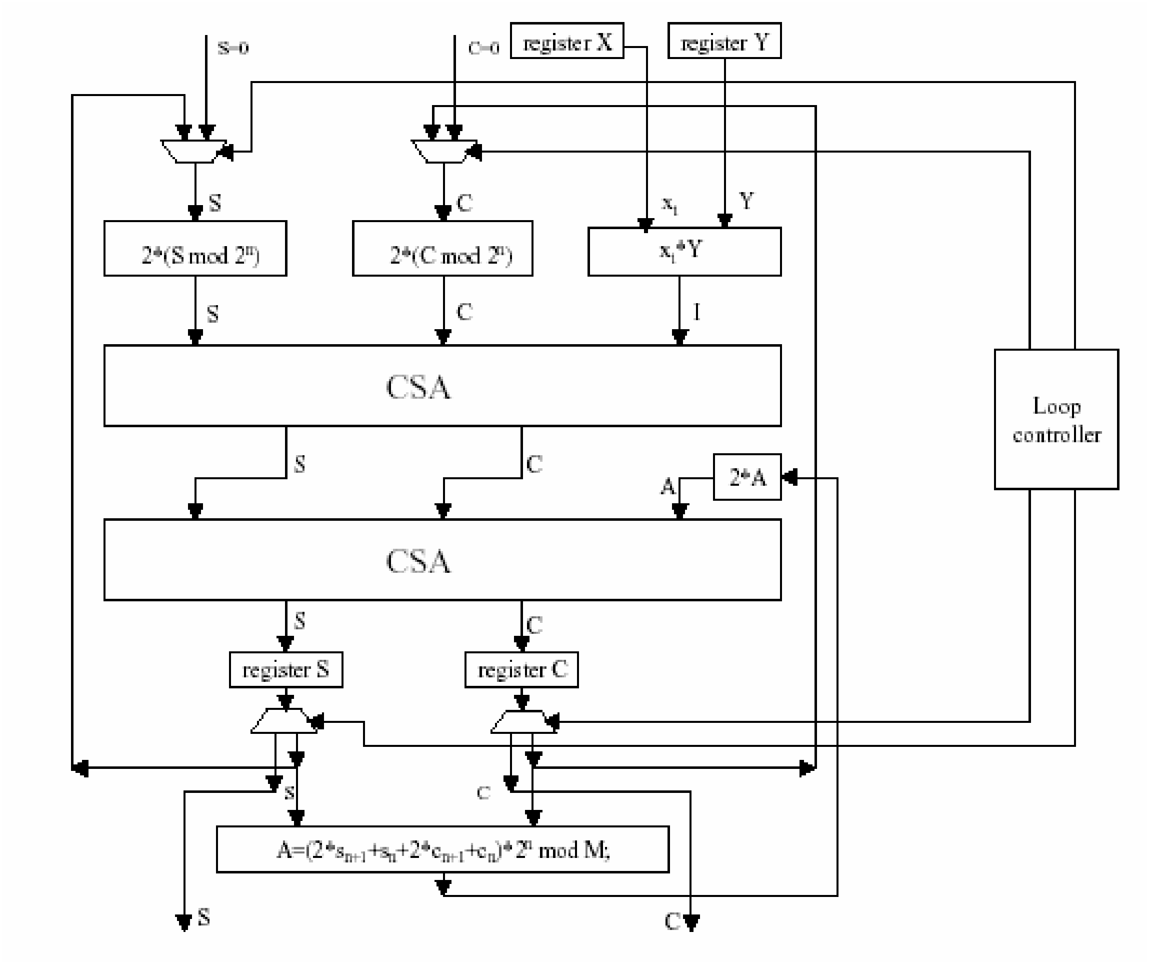


Fig. 3: Inner loop of modular multiplication using carry save addition [1]

In [1], the authors specified some modifications that can be applied to Algorithm 2 in order simplify and significantly speed up the operations inside the loop. The mathematical proof which confirms the correctness of the Algorithm 4 can be referred to in [1].

The architecture for the implementation of the loop of Algorithm 4 can be seen in the hardware layout in Figure 3.

In [1], the authors showed how to reduce both area and time by further exploiting precalculation of values in a lookup-table and thus saving one carry save adder. The basic idea is:

- the increment *I* within the loop can have only two possible values: i.e., 0 or *Y*. If x_i is 0 then *I* is 0 as well. If x_i is 1 then *I* is *Y*. Taking this into account we can reduce the number of operands to be added in the loop from four to three. In a precomputing phase each possible value for 2*A*+*I* is determined and stored in the lookup-table. During execution of the loop the lookup-table is addressed by the sum of the two leading digits of *S*+*C* and by x_i.

By these constructions, the number of carry save adders is reduced to 1 [1, Algorithm 5], thereby reducing the area complexity to 28*n*. The time complexity for one loop iteration is 2 (for the addition and lookup table access). The resulting area-time-complexity of Algorithm 5 is $56n^2$.

For a detailed explanation on how these AT complexity values were obtained, the reader is referred to [1].

Algorithm 5: Optimized version of the new algorithm [1]

```
Inputs: X, Y, M with $0 \le X, Y < M$
Output: $P = X*Y \bmod M$
n: number of bits in X;
$x_i$: $i^{th}$ bit of X;
Precomputing: LookUp(7) ----LookUp(0);
(1) $S := 0;$
(2) $C := 0;$
(3) $A := LookUp(x_{n-1});$
(4) for $(i = n-1;\ i \ge 0;\ i--)$ {
(5)     $S := S \bmod 2^n;$
(6)     $C := C \bmod 2^n;$
(7)     $S := 2*S;$
(8)     $C := 2*C;$
(9)     $(S,C) := CSA(S, C, A);$
(10)    $A := LookUp(2*(s_n + 2*c_{n+1} + c_n) + x_{i-1});$}
(11) $P := (S + C) \bmod M;$
```

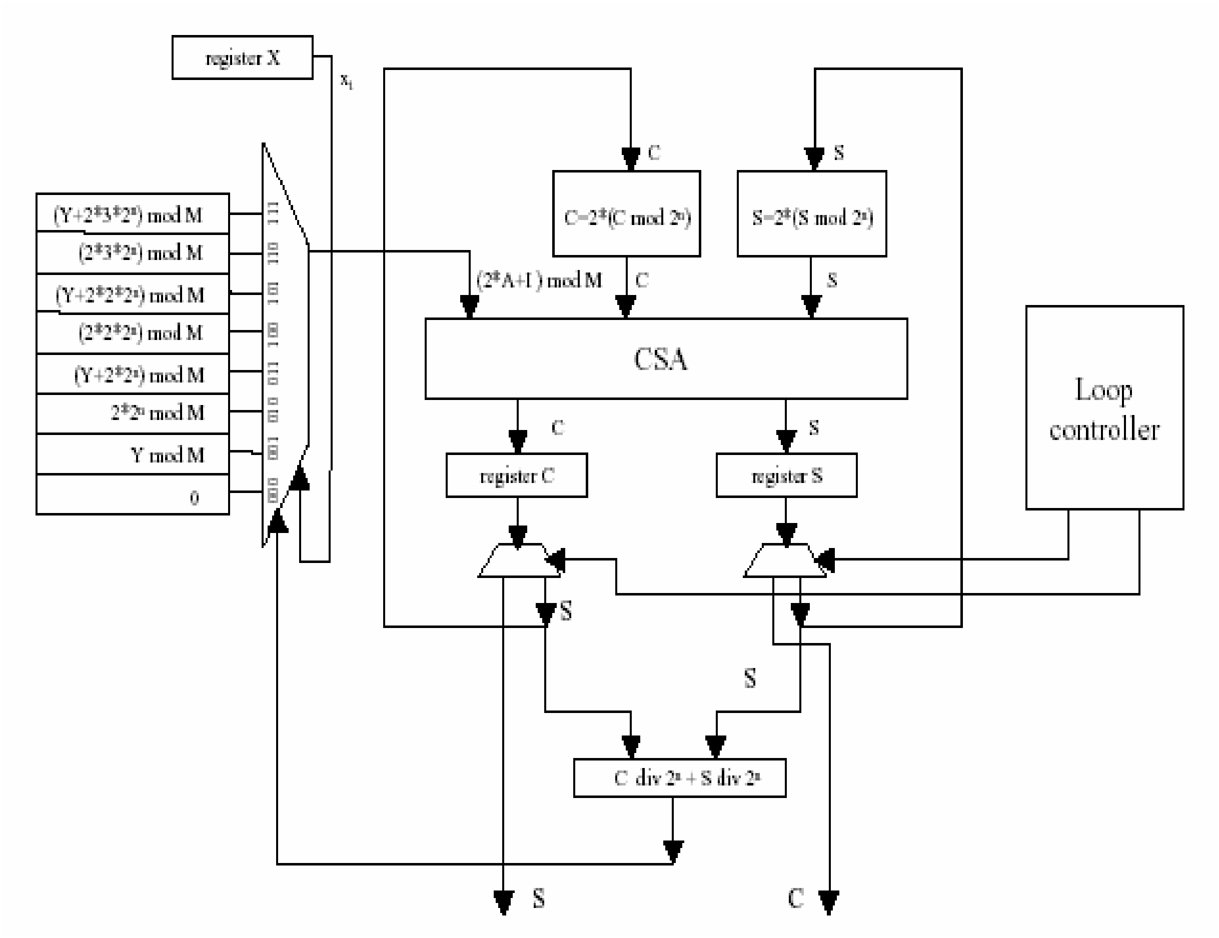


Fig. 4: Modular multiplication with one carry save adder [1]

Chapter 4

Software Implementation

4.1 Implementational Issues

The objective of this thesis comprises the hardware implementation and comparison of the architectures of the existing and new multipliers in [1] for performing modular multiplication. First of all, the corresponding algorithms of the architectures are to be examined by a high-level model written in an appropriate programming language. In this case JAVA was used because it has a collection of specialized functions and classes for doing long number arithmetic conveniently. Moreover, by using these unique classes and functions, the software implementations are not limited or restricted by the sizes of the normal data types in JAVA. As a result, the implementations are scalable and have been tested with actual practical implementation variables (e.g. 1024 bits input variables well suited for RSA).

4.2 Java Implementation of the Algorithms

The purpose of the software implementations is to verify the true functionality of the new algorithms and also to provide reliable test data for simulation of the VHDL implementations of the architectures. In that respect, the following algorithms were implemented:

- Montgomery Algorithm : [Algorithm 1a]
- Fast Montgomery Algorithm : [Algorithm 1b]
- Faster Montgomery Algorithm : [Algorithm 3]
- Interleaved Modular Multiplication Algorithm : [Algorithm 4]

- Optimized Interleaved Algorithm : [Algorithm 5]

The source codes for the JAVA implementation of these algorithms are available in electronic form.

In the following sections, the specialized classes in JAVA and functions used are discussed. The implementation details of the main steps within the algorithms are also explained.

4.2.1 Imported Libraries

Figure 5 shows the three main libraries that were imported for the implementation of the algorithms.

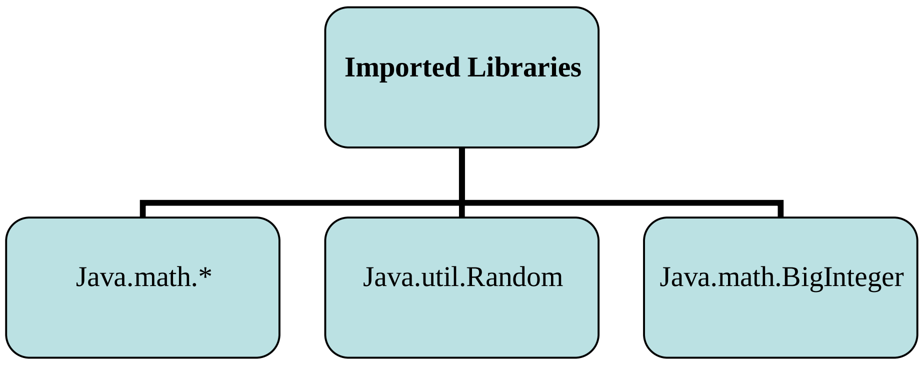


Fig. 5: Imported Libraries from Java

Java.math.BigInteger is a specialized class within *Java.math* library and has a collection of functions for the generation of random numbers for performing long number arithmetic conveniently. If you need to work with integers that are larger than fit into 64 bits, then you need to use the *Java.math.BigInteger* class. The main features of this class include the following:

- functions for performing bit operations
- random number generators.

With this class we are able to generate 1024 bit precision integers as shown in Figures 7, 8 and 9 to test the implementations. The function used for the generation of random numbers requires two arguments:

- The first argument represents the bit length of the random number to be generated and the second parameter invokes a random number generator object.

4.2.2 Implementation details of the Algorithms

In this section, a flowchart diagram is used to describe the sequence of steps, decisions and how certain processes are performed to implement an algorithm. For this purpose we shall use the Faster Montgomery algorithm [Algorithm 3]. The Algorithm is repeated here for quick reference.

Algorithm 3: Faster Montgomery multiplication

```
Inputs: $X, Y, M$ with $0 \le X, Y < M$
Output: $P = (X*Y(2^n)^{-1}) \bmod M$
n: number of bits in $X$;
$x_i$: $i^{th}$ bit of $X$;
$s_0$: LSB of $S$,   $c_0$: LSB of $C$,   $y_0$: LSB of $Y$;
R: precomputed value of $Y + M$;
(1) $S := 0;\ C := 0;$
(2) for $(i = 0; i < n; i{+}{+})$ {
(3)     if $((s_0 = c_0)$ and not $x_i)$ then $I := 0;$
(4)     if $((s_0 \ne c_0)$ and not $x_i)$ then $I := M;$
(5)     if $(\text{not}(s_0 \oplus c_0 \oplus y_0)$ and $x_i)$ then $I := Y;$
(6)     if $((s_0 \oplus c_0 \oplus y_0)$ and $x_i)$ then $I := R;$
(7)     $S,C := S + C + I;$
(8)     $S := S$ div $2;\ C := C$ div $2;$}
(9) $P := S + C;$
(10) if $(P \ge M)$ then $P := P - M;$
```

The flowchart in Figure 7 illustrates the path through the loop of Algorithm 3 where the first conditional if-statement (step 3) is assumed to be true. The statements within the carry save adder process block (CSA) reflect the true implementation of step 7 in JAVA. The *div* operator performs a 1-bit shift right operation. *LSB* stands for least significant bit.

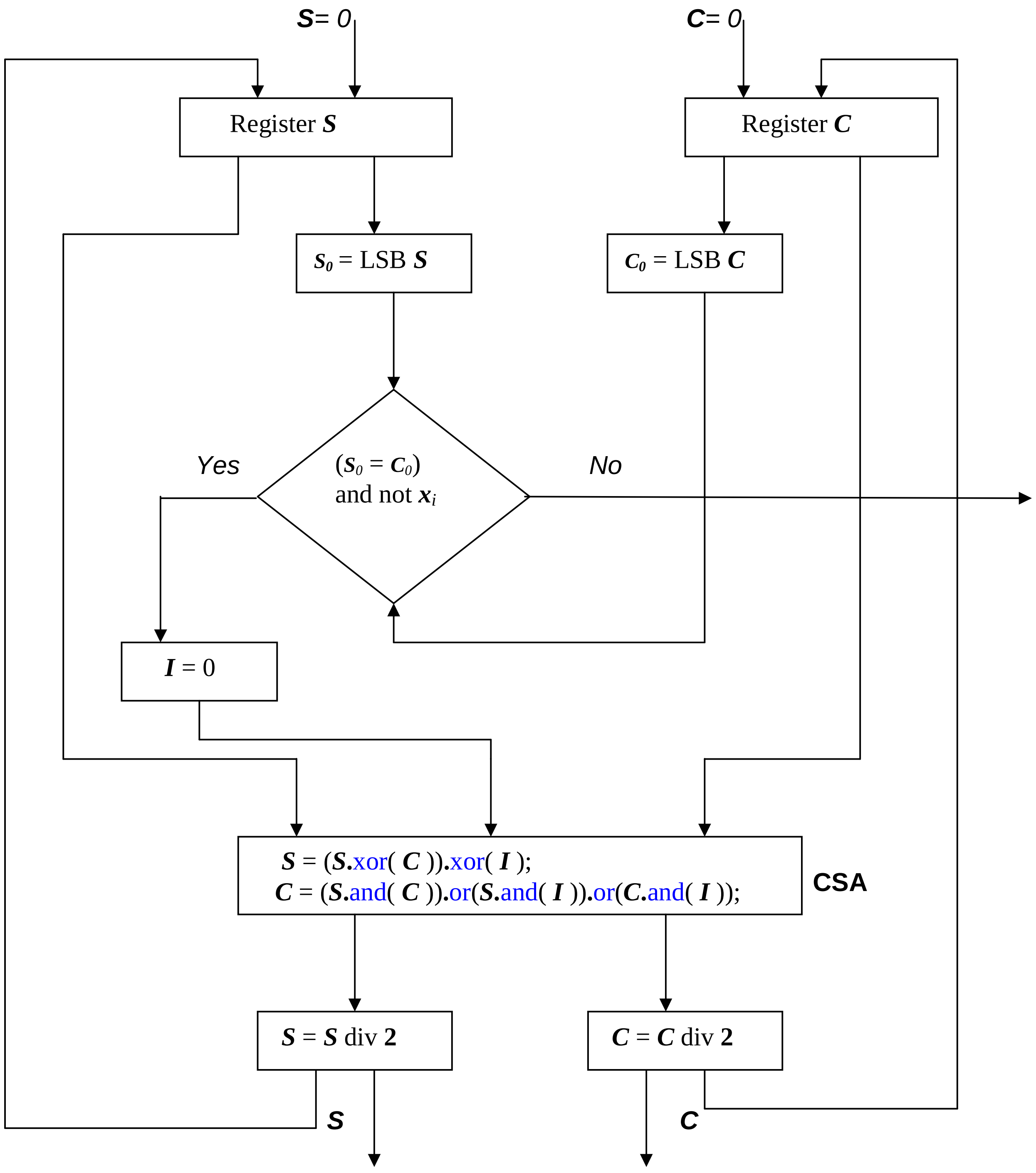


Fig. 6: Path through the loop of Algorithm 3, when first conditional if-statement holds true

4.2.3 1024 bits test of the implemented Algorithms

The three main algorithms under investigation namely, Fast Montgomery, Faster Montgomery, and optimized interleaved were tested with the following 1024 bit input variables of *X*, *Y* and *M* as shown in Figures 7, 8 and 9 respectively. The standard logic vector data type equivalents of these values are subsequently used to simulate the VHDL models of the implemented architectures.

Random Generated Input Variables of *X*, *Y* and *M* (*1024 bits word length*):

X = 13098850937954309318363672876874542737858161791561739401412035808036014938748
56607950021180786673123649416844204815241009423715163299921132647101057414199
03144173297218904663328612570525590711308709229768073510408936306809416529357
56130635864505657594150108752695816971831396604843358794293115978262582995684
7

Y = 168831365680513111978624418317634453213169183670184385942888306604872010884935
03598667977239163857923422467791416231857796837047115413435109916171264775387
97305563309664744378230706300562874159970474791271746426088883656787721660373
96111815963444672919333360118204338535009707172427640150846653211922666363797
3

M = 1376271285329512661642971345429058551738185151221229217966412465883677787945
2981055956324754119554183935686352725235860699928257944236774055672623604838
1287901369230875487236927579797247458441156695021712978582842954717289812458
1719656737696113924932157568437795806949459511960177310382684855752916418527
32489

The results of the sum *S* and carry *C* after the loop in [1, Algorithm 1b} are:

S = 1007167581825741918581576505446352226072365271120524333461285685492321430340358
9001984637640153217467060392633294166653866164319955338812997302949777955000706
1398207485904042849830278909757550264473484168682024426779894801040959876275251
603779948053187297116306582683193361237443487660705763218147879887398580

C = 44942371233251074460467671245692522292150515345253416484075364491855081020755844
9915250062096511995717256755370609990377776651804193075198598756031483629922741
1985687179891256055805007083025331223865040700030631795521776180834256375793569
47450071019652470615411906351261880470676024954108871407344442954228226

Fig. 7: A 1024 bit test of Algorithm 1b.

Random Generated Input Variables for *X*, *Y* and *M* (*1024 bits word length*):

X = 123608970757612173439914553432287822772544227541103372802441408276575144271 63
60227307524458400296213426516345314376196370265490709069 20919 914103729421710
085996911636221955119085138635456141085074098142752000458206680223851184986 13
971837077882674683387362239645835978607047489292248734439038220188097642325 80
21

Y = 162202340980223594225982548495395133203103645492911692960633899685858700248 77
723646027036151648651478658403659744879248438620855828986914960927731559968 13
760285660655333602605160865110722614537867629657904758779169934791482678770 51
4 87033640570046240102985928207552324332272336660200082187356566827068074151 7
57

M =1171406609175907621495633434753733570636944265813571952851922717375364925 5362
717648839891639569436751957785699280297057647277034862902065929477302728004 76
032037276178968218932751225616870357156123575539262123484458980577220018873 79
889457977784195686583692790780403389878812381843933561932625144275897724650 5
87

The results of the sum *S* and carry *C* after the loop in [1, Algorithm 3] are :

S = 164338102671004209641012075707604041394058475120280509057579976556762757
140402759833112002925687288589690217656371041537691617351772328748460307
981253518461172131114282367139131852051938433126452907914126883100597028
792067279820712015077511599041783949998140520331121232838369390738854377
93411982136542024922 7

C = 970514715792288368273714315637148422526859018286500710463843012796648540
190293455594672459398618248438757880650776198808650855352451092023738789
800770774917065729965553959330621029777210678735004497585736361417789006
789226517796688929228873964557959978670669942713027860531869552880526821
5037298744409069225

Fig. 8: A 1024 bit test of Algorithm 3

Random Generated Input Variables *X*, *Y* and *M* (*1024 bits word length*):

X =16741574119083621738081608241000477007940585133828670924158445528680413396596497
16358892214229025544049520015771150457665608175072159644364304170101619470561502
08721370781188072275421392421216855979196899088118731102272273813880244034235646
257672952770467781418922625133540053131948908056598710410824133299643

Y = 17906935790238361563010155590620460248158858714722425743147449205291265656683248
31415356976250047258309505921356733844066999902219805916722648390255538118669177
18297979451930552339329930574416892865810509598264062380157580972494478576544464
528538033641925637018123561589133279598567387900020749496094476712527

M = 14023547777108037889344154322683873117349871724307058347051188868065648662644526
01133673265430713494475417269916007589436909894235341555650190442449226195605535
03456847456032671535610050422831494231621894151272186114206557861995263982897279
197072311105905107894704402560392669938659934579999344234617958419021

The results of the sum S and carry C after the loop in [1, Algorithm 5] are :

S = 9287891079622132122984599988666409074390724118093603322713837944133720338348233
7144055828964256165856892674770596639850838873662807837698756201844434091050301
6384854344091134334150821567222122572986511101773489014917457944150512462594242
64335982269637576613176905943979634636707308399472636035881102365126962

C = 7074325870408808601984203381781263612050940720011766845713678882996634543543017
4509947879733356498994347823125124837439013651029767985688628827645504594129232
9001637466064488611401872569828546359458294570199082194214955927078405990646536
637755490671560733502182885367476438929293731516496463752734936329197264

Fig. 9: A 1024 bit test of Algorithm 5

The correctness of the results (i.e. sum and carry) of these tests were verified by means of functions available in *Java.math.BigInteger* library that enables us to compute $X*Y \bmod M$, and $(X*Y*(2^n)^{-1}) \bmod M$ *without* reference to the implemented algorithms. This was done only for testing purposes and was removed from the source codes once the test results of the algorithms were confirmed to be true.

Chapter 5

Hardware Implementation

5.1 Modeling Technique

The design capture of the architectures was done using Very High Speed Integrated Circuit Hardware Description Language (VHDL) and the complete source codes for 32 to 1024 bit implementations of Fast Montgomery, Faster Montgomery and Optimized Interleaved multipliers are available in electronic form.

For the implementation of the multipliers, a very structured approach was used which shows the hierarchical decomposition of the multipliers into submodules The basic units of the architectures which comprises carry save adders, shift registers and registers were modeled as components which are independently functional. These components are then wired together by means of signals to construct the structure of the multiplier as shown in Figure 10 for the Faster Montgomery architecture [1, Figure 2].

5.2 Structural Elements of Multipliers

Every VHDL design consists of at least an *Entity* and *Architecture* pair. *Entity* describes the interface of the system from the perspective point of its input and output, while *Architecture* describes the behavior or the functionality of the digital system itself. In the following sub-sections, the *Entity* and *Architecture* pair of the structural elements in the Faster Montgomery architecture presented in Figure 10 is described.

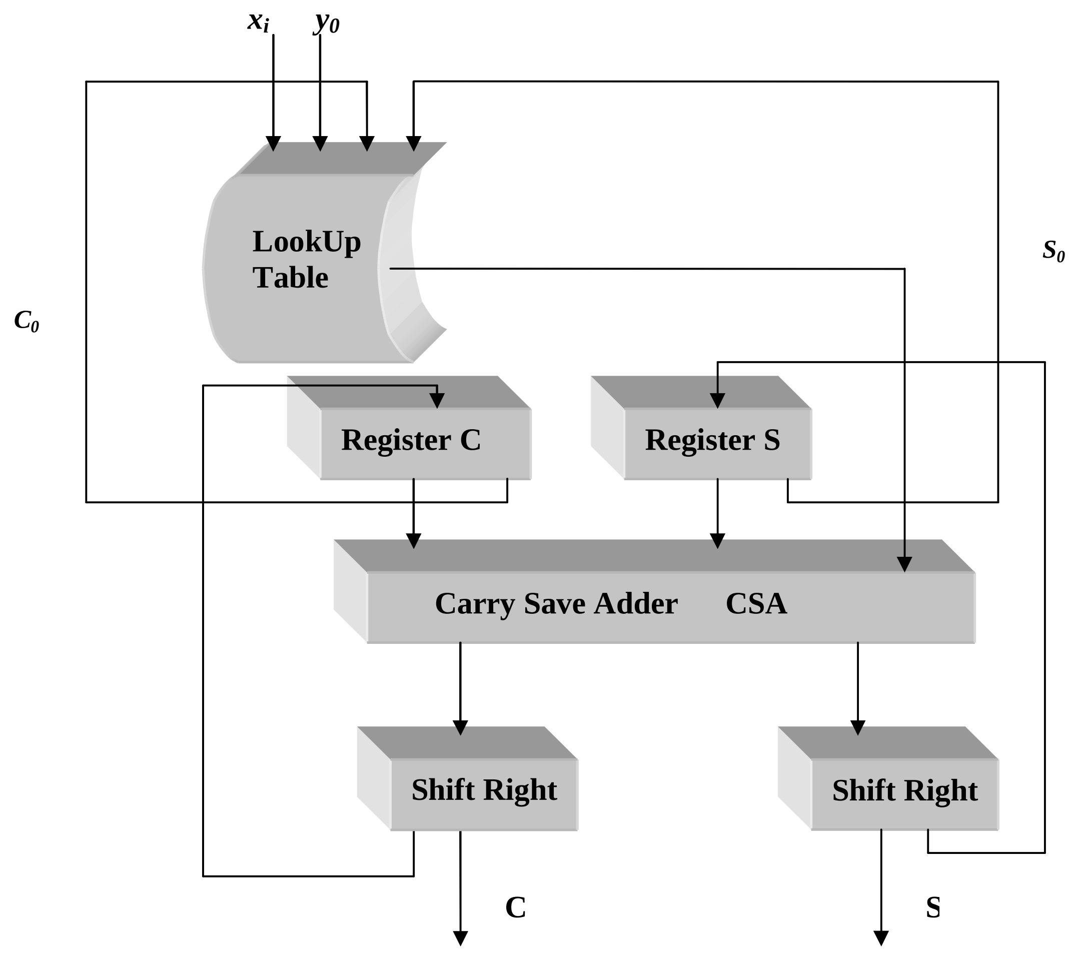


Fig. 10: Block diagram showing components that were implemented for the Faster Montgomery architecture

5.2.1 Carry save adder

The carry save adder is simply a parallel ensemble of *n* full-adders without any horizontal connection. Its main purpose is to add three *n*-bit integers *X*, *Y* and *Z* to produce two integers *C* and *S* such that

$$C + S = X + Y + Z$$

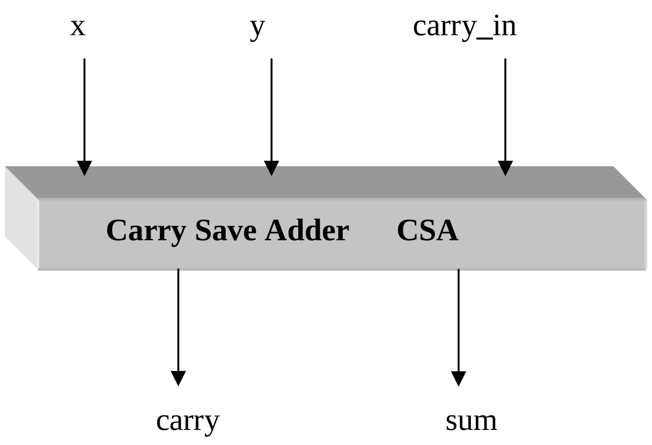
where *C* and *S* represent the carry and sum respectively.
The i^{th} bit s_i of the sum *S* and the $(i + 1)^{st}$ bit c_{i+1} of carry *C* are calculated using the boolean equations

$$s_i = x_i \oplus y_i \oplus z_i$$

$$c_{i+1} = x_i y_i \vee x_i z_i \vee y_i z_i$$

$$c_0 = 0,$$

The *Entity* and *Architecture* pair for the VHDL implementation of the carry save adder in Figure 10 is as shown in Figure 11.



```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

--    EXTERNAL PORTS
ENTITY c_s_adder1 IS
      GENERIC(n: integer := 32);
PORT( x:          IN     std_logic_vector(n DOWNTO 0);
      y:          IN     std_logic_vector(n DOWNTO 0);
      carry_in:   IN     std_logic_vector(n DOWNTO 0);
      sum:        OUT    std_logic_vector(n+1 DOWNTO 0);
      carry:      OUT    std_logic_vector(n+1 DOWNTO 0));
END c_s_adder1;

--    INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF c_s_adder1 IS

BEGIN

sum <= '0' & ((x XOR y) XOR carry_in);
carry <= (((x AND y) OR (x AND carry_in)) OR (y AND carry_in))& '0' ;

END behavioral;
```

Fig. 11: VHDL implementation of carry save adder

As shown in Figure 11, the interface between a module and its environment is described within the entity declaration which is preceded by the keyword *ENTITY*. In all the implementations, only standard logic data types are used. The most important reason to use standard logic data types is portability. It gives you a standard and a portable style simulation or changing simulation environment in interface with other components. To use the IEEE 1164 standard logic data types, at least two statements must be added in the source code. These statements are:

- LIBRARY IEEE
- USE IEEE.std_logic_1164.ALL

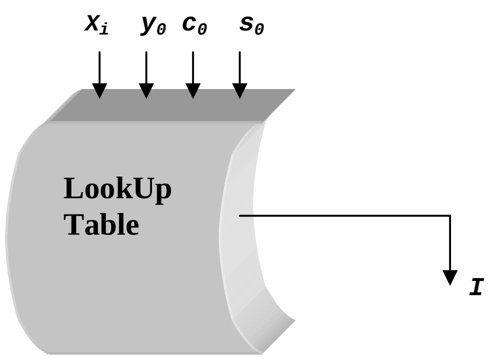
The VHDL source code in Figure 11 represents the actual implementation of the carry save adder in the architecture of Figure 10 (see also Figure 2). The bit length of the three inputs *x*, *y* and *carry_in* is *n+1,* because one of the three inputs (i.e. the value from the lookup table) in Figure 2 is *n+1* bit long as a result of the addition *Y+M.* The *sum* and *carry* outputs are represented in full length before they are sent back into the carry save adder for the next iteration of the loop to prevent sign detection problems.

5.2.2 Lookup Table

The lookup table is one of the most important units inside the new optimized architectures in [1]. It is used to store the values of precomputations that are performed prior to the execution of the loop. This eliminates time consuming operations that are performed inside the loop, thus improving the speed of computation.

Figure 12 shows the VHDL implementation of the lookup table in Figure 10 (see also Figure 2). The lookup table is addressable by 4 bits (i.e., x_i, y_0, c_0 and s_0) and its outputs are *n+1* bit long as a result of the sum *Y+M* (see Figure 2) which is used to generate some of its internal values. Note that both *Y* and *M* are *n* bit integers. In all, 16 probable values of *I* (see Algorithm 3) are precomputed and stored in the lookup table prior to the execution of the loop. The relationship between the probable values of *I* and the address bits is given by $16 = 2^k$, where *k* represents the address bits (i.e. 4).

The implementation is quite simple as depicted in Figure 12. By the use of CASE statement, VHDL enforces that all the cases must be covered, and this is guaranteed by the ‘OTHER’ clause. The main advantage in using the CASE statement for the implementation is that it eliminates the sequential evaluation of alternatives. Rather, the system jumps directly to the true condition, thus preventing any undue delay introduced by access to the lookup table.



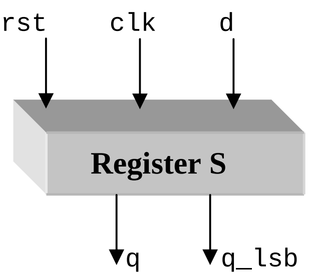
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

--EXTERNAL PORTS
ENTITY table_LU IS
      GENERIC ( n     : integer := 32);
      PORT (
            address : IN  std_logic_vector(3 DOWNTO 0); -- 4-bit
Address
            result  : OUT std_logic_vector(n DOWNTO 0)); -- output
END table_LU;

-- INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF table_LU IS

BEGIN
      pro_1: PROCESS(address)
            BEGIN
                  CASE address IS
                              WHEN "0000" =>
                  result <= "000000000000000000000000000000000";
                              WHEN "0001" =>
                  result <= "010110010111101011110100000001101";
                              WHEN "0010" =>
                  result <= "000000000000000000000000000000000";
                                    .
                                    .
                                    .
                              WHEN "1101" =>
                  result <= "010010000011010000001010010101011";
                              WHEN "1110" =>
                  result <= "010010000011010000001010010101011";
                              WHEN OTHERS =>
                  result <= "101000011010111011111100101110000";
                  END CASE;
      END PROCESS;
END behavioral;
```

Fig. 12: VHDL implementation of lookup table

5.2.3 Register

The purpose of the Registers (i.e. *Register C and Register S*) in Figure 2 and Figure 10 are to hold the intermediate values of the *carry* and *sum* respectively during the execution of the loop. Thus, the registers must have memory and be able to save their state over time. To obtain such a behavior, the following rules must be observed during the implementation:

- the sensitivity list of the process should not include all the signals that are being read with the process.
- 'IF-THEN-ELSE' statements should be incompletely used.



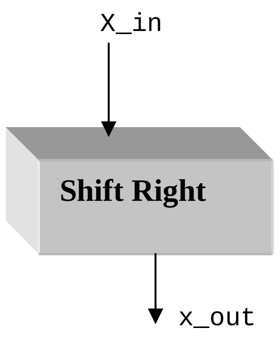
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

--    EXTERNAL PORTS
ENTITY h_register IS
      GENERIC( n: IN    integer := 32);
PORT(
      rst:  IN    std_logic;
      clk:  IN    std_logic;
      d:    IN    std_logic_vector(n DOWNTO 0);
      q:    OUT   std_logic_vector(n DOWNTO 0);
      q_lsb: OUT  std_logic);
END h_register;

--    INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF h_register IS

      SIGNAL q_temp: std_logic_vector(n DOWNTO 0);
BEGIN
      PROCESS(rst, clk)
      BEGIN
            IF rst = '1' THEN
                  q_temp <= (q_temp'RANGE => '0');
            ELSIF clk'EVENT AND clk = '1' THEN
                  q_temp <= d;
            END IF;
      END PROCESS;
      q <= q_temp;
      q_lsb <= q_temp(0);

END behavioral;
```

Fig. 13: VHDL implementation of register

Figure 13 illustrates the VHDL implementation of the registers inside Figure 2 and Figure 10. The reset signal *rst* is used to initialize the registers to ZERO at the start of the loop. The input data *d* is triggered to the output *q* on the positive rising edge of the clock signal. The output *q_lsb* represents the least significant bit of input *d*, which is one of the bits used to address the lookup table.

5.2.4 One-Bit Shifter

The one-bit shifters inside Figure 2 and Figure 10 are used to perform 1-bit, right-shift operations. The behavioral description of this unit is as shown in Figure 14. Here, the least significant bit of the input is discarded at the output, thereby reducing the bit length at the output by 1.



```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

--    EXTERNAL PORTS
ENTITY shift_right IS
      GENERIC( n: integer := 32);
PORT(
      x_in: IN    std_logic_vector(n+1 DOWNTO 0);
      x_out: OUT  std_logic_vector(n DOWNTO 0));
END shift_right;

--    INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF shift_right IS
      SIGNAL temp:      std_logic_vector(n DOWNTO 0);
BEGIN
      PROCESS(x_in)
      BEGIN
            FOR i IN n+1 DOWNTO 1 LOOP

            temp(i - 1) <= x_in(i);

            END LOOP;
      END PROCESS;

      x_out <= temp;

END behavioral;
```

Fig. 14: Implementation of ‘Shift Right’ unit

5.3 VHDL Implementational Issues

In this section, issues arising from the implementation of the multipliers are discussed.

The architectures presented in Chapter 2 and 3 were implemented without any modification, except Faster Montgomery multiplier. In Figure 2, the layout for the implementation of the loop of Algorithm 3 consists of two registers (register C and register S) for storing the intermediate values of the sum and carry. The layout also contains a third register (register I) designed to hold the bits received from the Look-Up table before they are transported into the carry save adder (CSA) on the positive rising edge of the clock. However, it turned out that the register (i.e., register I) can be omitted. The architecture was modified by removing register I so that the values from the Look-Up table are applied directly to the carry save adder (CSA) as shown in the revised architecture in Figure 10. Thus, the implemented VHDL model for Faster Montgomery was done after the removal of register I from the architecture in Figure 2.

The layout for the implementation of the optimized interleaved algorithm consists of units for performing the following operations:

- 2*(C mod 2^n)
- 2*(S mod 2^n)
- (C div 2^n + S div 2^n).

For the implementation of these units in VHDL, it is important to consider the following rule:

- Integer data types should not be used as it is far too insufficient and not scalable for long wordsizes. The *mod* and *div* functions operate on integer data types only.

A smart way of implementing these operations to avoid the above limitation is to deal with only 0's and 1's (i.e., using standard logic data types only!). For example, the mathematical statement 2*(C mod 2^n) means reduce C to *n* bits (i.e. select the *n*–1 downto 0 bits) and then shift the result to the left by 1-bit. This can be implemented by one simple concurrent statement in VHDL as shown below:

```
c_out <= (c_in(n-1 DOWNTO 0)) & '0';
```

where c_in represents C in the mathematical statement and c_out represents the result of 2*(C mod 2^n). The concatenation operator is used to perform the left shift.

Similarly, C div 2^n means select the higher order bits down to n, and this mathematical expression can be easily implemented in VHDL without making use of integer dependent *div* operator.

Hence, the elimination of all arithmetic operators in the actual implementations and dealing with only 0's and 1's has not only produced scalable multipliers but further improved the area-time product of the architectures.

It is important to note that [1] did not specify how the post computation (i.e. P = S + C) outside the loop can be implemented. This is necessary to arrive at the final result P of the modular multiplication operation and will involve the use of adders with significant carry propagation for n > 512 bits. In these implementations, this step is omitted.

Of particular interest are the input blocks of registers which are used to load data into the multipliers. By synthesis the functionality of the multipliers are to be mapped to a Xilinx Virtex 2000E target device. However, this device has only a limited number of input and output pins. As a result, the multipliers were modelled as two cascaded blocks; the first block representing 32 bit registers with memory designed to hold the input data during the loading stage, and the second block is the multiplier itself, which is triggered to start operation once the loading of data into the registers is complete. Note that we can also use 32 bit Block RAMs for this purpose.

Figure 15 below shows a simplified data flow diagram of a 32 bit design of the blocks of registers used to hold the input data bits. As depicted in the diagram, 3 blocks of parallel ensemble 32-bit registers are required for the three input variables *X*, *Y* and *M*. For this implementation, the registers are addressable by 2 bits since the total number of registers is 3.

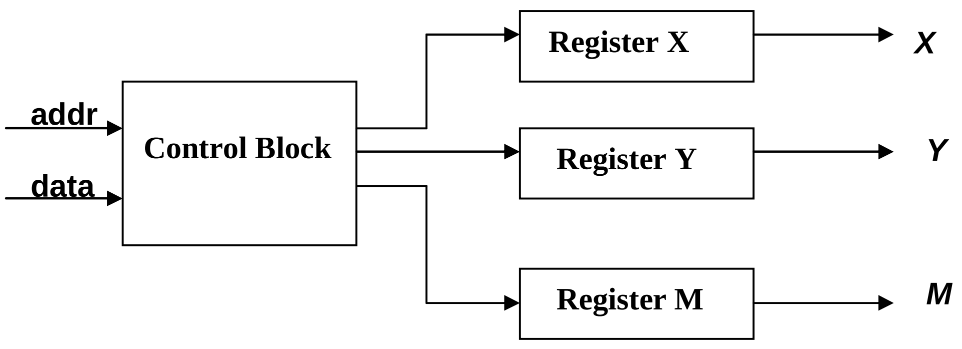


Fig. 15: 32 bit blocks of registers for storing input data bits

5.4 Simulation of Architectures

The functionality of the implemented architectures is to be verified by simulation using values generated by the software implementations in JAVA. All the architectures were simulated with variables of precision ranging from 32 to 1024 bit.

The waveform in Figure 18 illustrates one simulation of the Fast Montgomery multiplier where X = 47563, Y = 46337 and M = 57349. As shown on the left hand side of the waveform, *s_final* and *c_final* represents the values of the sum and carry after the loop when the state is FINISHED. Figure 16 illustrates the state-machine diagram of the implemented multipliers. Each multiplier has 3 main states: Starting, Running and Finished. A multiplier is triggered into operation by performing two sequential operations:

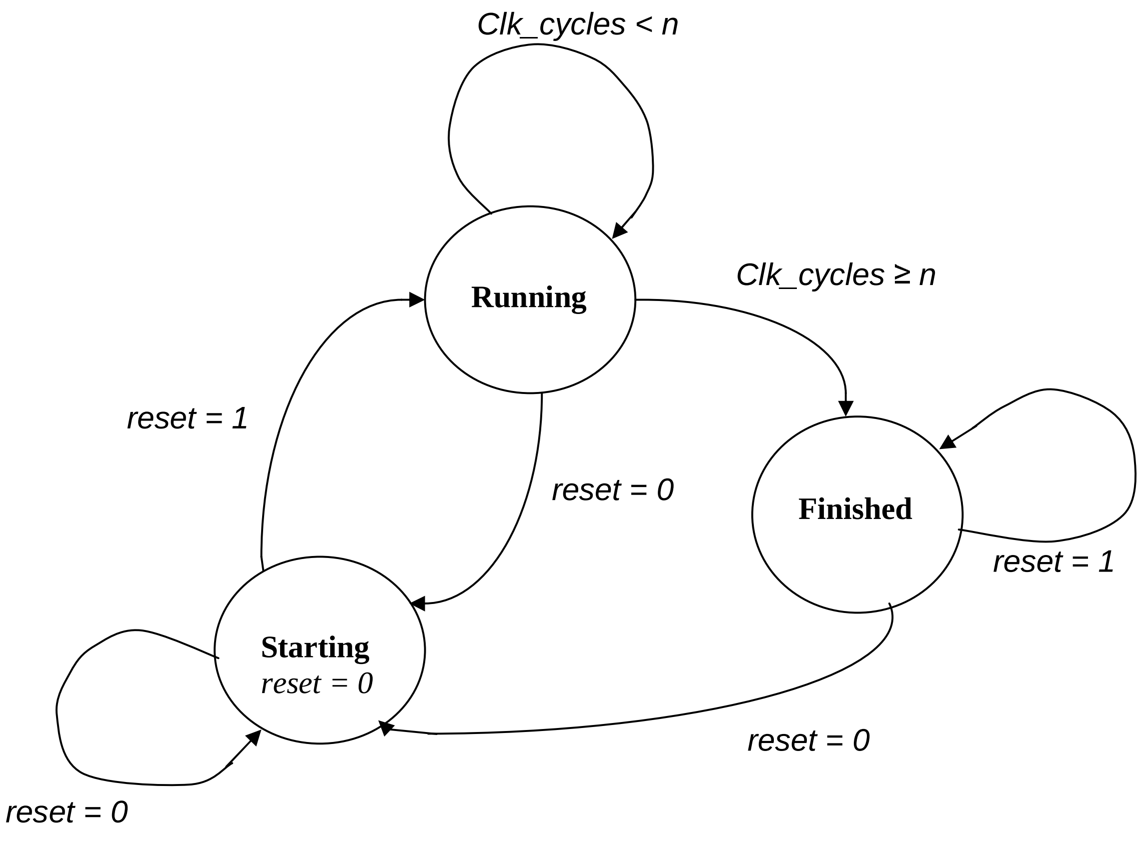


Fig. 16: State diagram of implemented multipliers

- Setting the reset signal to ‘0’. This is used to indicate that loading of data into the input registers *X*, *Y* and *M* is complete. It is also used to initialize the internal registers of the multipliers (i.e. Register S and Register C) to ZERO prior to the execution of the loop.

- Setting the reset signal to ‘1’. This operation must directly follow the first operation to trigger the multiplier into the *running* state.

In Figure 16, *clk_cycles* is use to represent the total number of clock cycles and *n* denotes the bit length of the input variables *X*, *Y* and *M,* which are being processed. In this thesis and [1], *X*, *Y* and *M* are all *n* bit long.

As stated in section 5.3, the multipliers were modelled as two cascaded blocks; the first block representing 32 bit registers with memory designed to hold the input data during the loading stage, and the second block is the multiplier itself. The state-machine diagram in Figure 16 represents the states of the multiplier only. Figure 17 describes the states of the block of registers for loading data into the multiplier. These states precede the states in Figure 16.

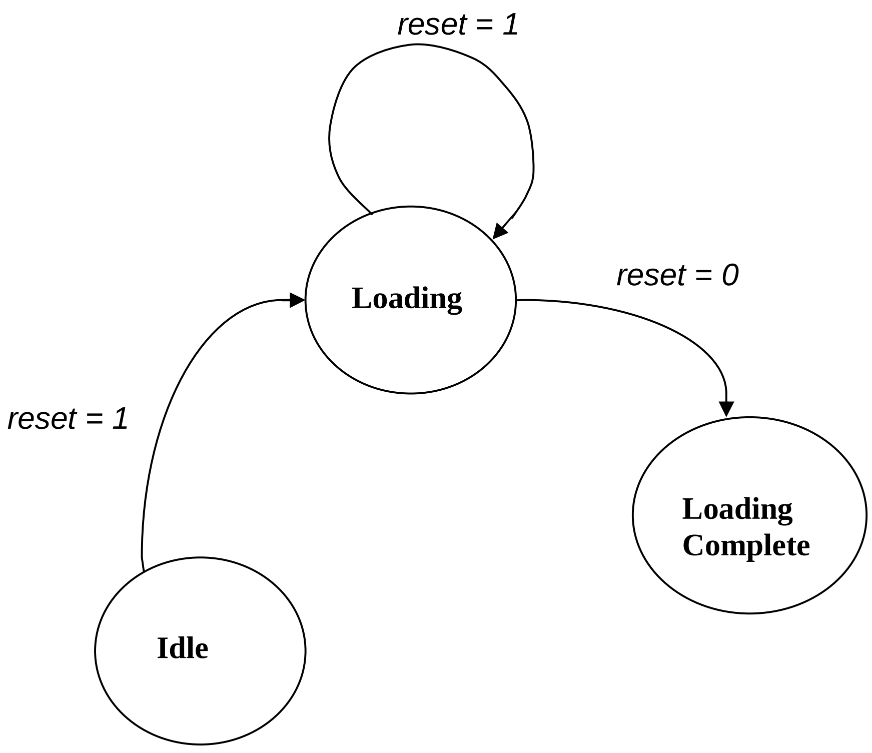
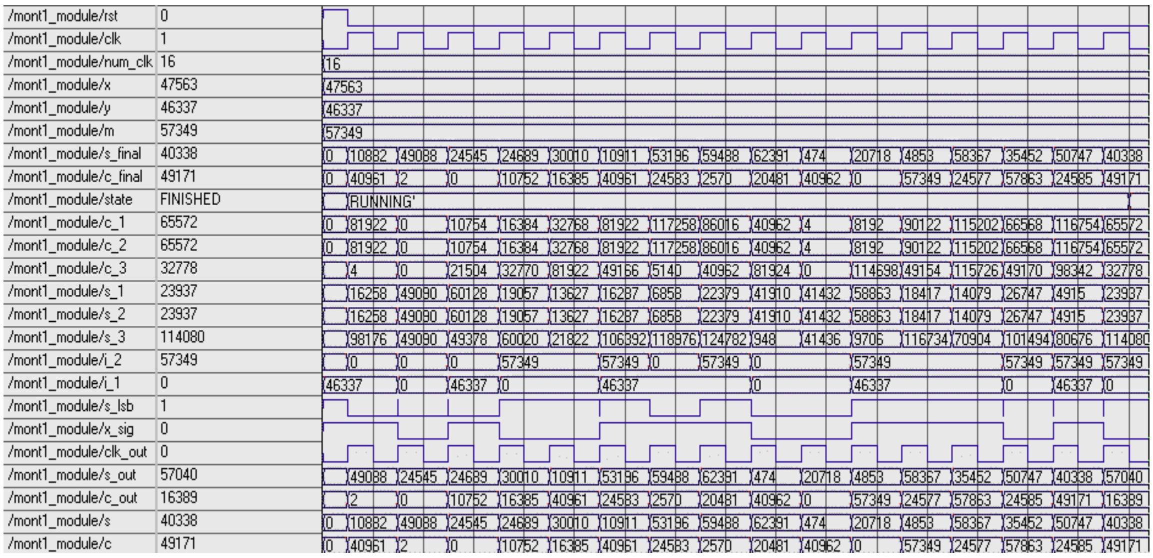


Fig. 17: State diagram for input block of registers

Thus, the multiplier together with the input block of registers for loading data consists of the following 5 states:

- Idle
- Loading
- Loading Complete/ Starting
- Running
- Finished.

The two states *Loading Complete* and *Starting* can be merged because the reset signal ‘0’ which is used to terminate *Loading* is the same signal used to trigger the multiplier into the *Starting* state.



/mont1_module/rst	0
/mont1_module/clk	1
/mont1_module/num_clk	16
/mont1_module/x	47563
/mont1_module/y	46337
/mont1_module/m	57349
/mont1_module/s_final	40338
/mont1_module/c_final	49171
/mont1_module/state	FINISHED
/mont1_module/c_1	65572
/mont1_module/c_2	65572
/mont1_module/c_3	32778
/mont1_module/s_1	23937
/mont1_module/s_2	23937
/mont1_module/s_3	114080
/mont1_module/i_2	57349
/mont1_module/i_1	0
/mont1_module/s_lsb	1
/mont1_module/x_sig	0
/mont1_module/clk_out	0
/mont1_module/s_out	57040
/mont1_module/c_out	16389
/mont1_module/s	40338
/mont1_module/c	49171

Fig. 18: Simulation of Fast Montgomery multiplier

5.5 Synthesis

Precision RTL Synthesis tools were used to map the implemented architectures to Xilinx Virtex 2000E device. The synthesis of the VHDL implementations of the architectures was done with minimum constraints imposition and this was followed by place and route. During place and route, vital design statistical reports were generated. Of prime importance to this thesis are the area and timing reports which are to be used to compare the implemented architectures.

Chapter 6

Results and Analysis of the Architectures

6.1 Design Statistics

During place and route, the synthesis tool generated the design reports for each of the implemented multipliers. In this chapter, the figures for the minimum clock period, total equivalent gates and flip-flops, percentage of configurable logic block slices used and the overall area requirements for each implementation are tabulated and graphically analysed.

The figures represent the overall hardware requirements for the complete multipliers including the block of registers for holding the input data bits.

Precision	Fast Montgomery	Faster Montgomery	Optimized Interleaved
32 bits	**1.18%**	**0.70%**	**0.68%**
64 bits	**2.35%**	**1.36%**	**1.36%**
128 bits	**4.69%**	**2.70%**	**2.70%**
256 bits	**9.37%**	**5.37%**	**5.36%**
512 bits	**16.05%**	**8.04%**	**8.04%**
1024 bits	**45%**	**21%**	**24%**

Table 1: Percentage of configurable logic block slices (out of 19200) occupied depending on bitlength

Precision	Fast Montgomery	Faster Montgomery	Optimized Interleaved
32 bits	262	140	150
64 bits	393	269	310
128 bits	784	528	570
256 bits	1568	1048	1118
512 bits	3644	2597	2698
1024 bits	6293	3163	4289

Table 2: Number of gates

Precision	Fast Montgomery	Faster Montgomery	Optimized Interleaved
32 bits	17.714ns(56.453MHz)	16.361ns(61.121MHz)	14.242ns(70.215MHz)
64 bits	20.335ns(49.176MHz)	15.990ns(62.539MHz)	17.323ns(57.727MHz)
128 bits	18.741ns(53.359MHz)	19.492ns(51.303MHz)	20.677ns(48.363MHz)
256 bits	21.271ns(47.012MHz)	21.760ns(45.956MHz)	21.366ns(46.803MHz)
512 bits	24.513ns(40.795MHz)	23.081ns(43.326MHz)	15.443ns(64.754MHz)
1024 bits	23.759ns(42.089MHz)	20.398ns(49.024MHz)	14.412ns(69.387MHz)

Table 3: Minimum period and maximum frequency

Precision	Fast Montgomery	Faster Montgomery	Optimized Interleaved
32 bits	454	267	260
64 bits	904	524	521
128 bits	1802	1037	1035
256 bits	3598	2062	2060
512 bits	6164	3087	3087
1024 bits	12319	5134	6160

Table 4: Number of Dffs or Latches

Precision	Fast Montgomery	Faster Montgomery	Optimized Interleaved
32 bits	235	146	155
64 bits	400	276	309
128 bits	793	536	573
256 bits	1581	1057	1117
512 bits	3663	2613	2613
1024 bits	6323	3180	4291

Table 5: Number of Function Generators

Precision	Fast Montgomery	Faster Montgomery	Optimized Interleaved
32 bits	5	5	5
64 bits	6	6	6
128 bits	7	7	7
256 bits	8	8	8
512 bits	9	9	9
1024 bits	10	10	10

Table 6: Number of MUX CARRYs

Precision	Fast Montgomery	Faster Montgomery	Optimized Interleaved
32 bits	4,490	2,844	2,848
64 bits	8,516	5,494	5,674
128 bits	16,912	10,780	10,992
256 bits	33,710	21,344	21,694
512 bits	62,128	37,350	37,872
1024 bits	118,118	54,062	68,942

Table 7: Total equivalent gate count for design

6.2 Area Analysis

Using the tables from the design reports in chapter 5, the approximate area requirements of the implemented multipliers are examined using the percentage of configurable logic block slices occupied by their architectures.

As shown in Figure 19, the Fast Montgomery architecture occupies the largest number of configurable logic block (CLB) slices for all implementations. The CLB requirements for the optimized architectures, namely, Faster Montgomery and Optimized Interleaved are approximately the same, but the Faster Montgomery uses less number of gates. It is interesting to note from Table 1 that an increase in the bitlength by a factor of 2 leads to a corresponding rise in the CLBs used. This behaviour holds true for the three different multipliers for bitlength in the range 32 to 256 bit. For bitlengths above 256 bit, the rise in CLBs is exponential. Figure 20 depicts this relationship between bitlength and CLBs for the Fast Montgomery multiplier. From the tables in Chapter 5, all the three different multipliers exhibit this behaviour of rising CLBs with bitlength. Hardware resources such as, number of gates, flip-flops or latches, and function generators bears the same relationship.

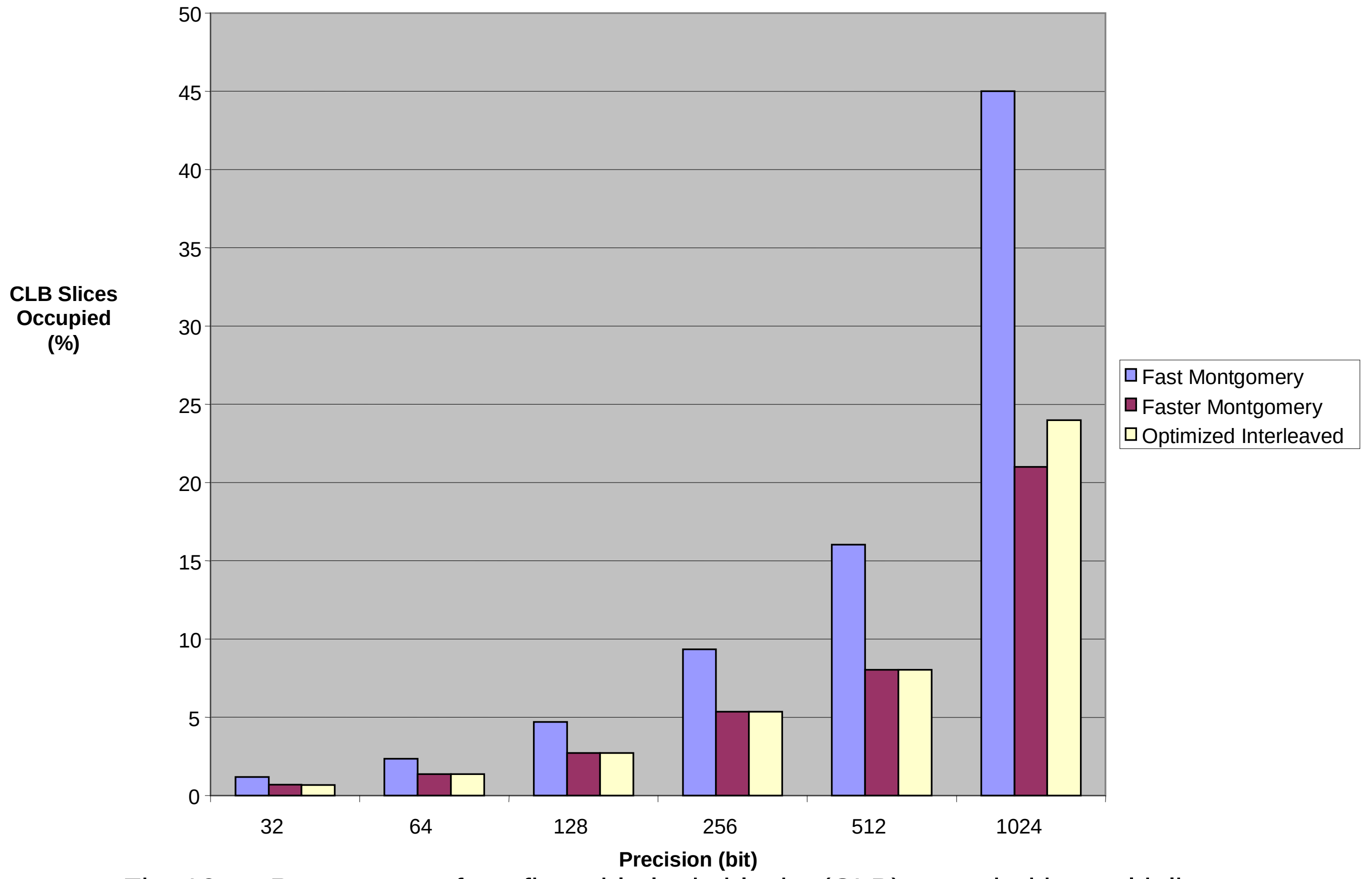


Fig. 19: Percentage of configurable logic blocks (CLB) occupied by multipliers

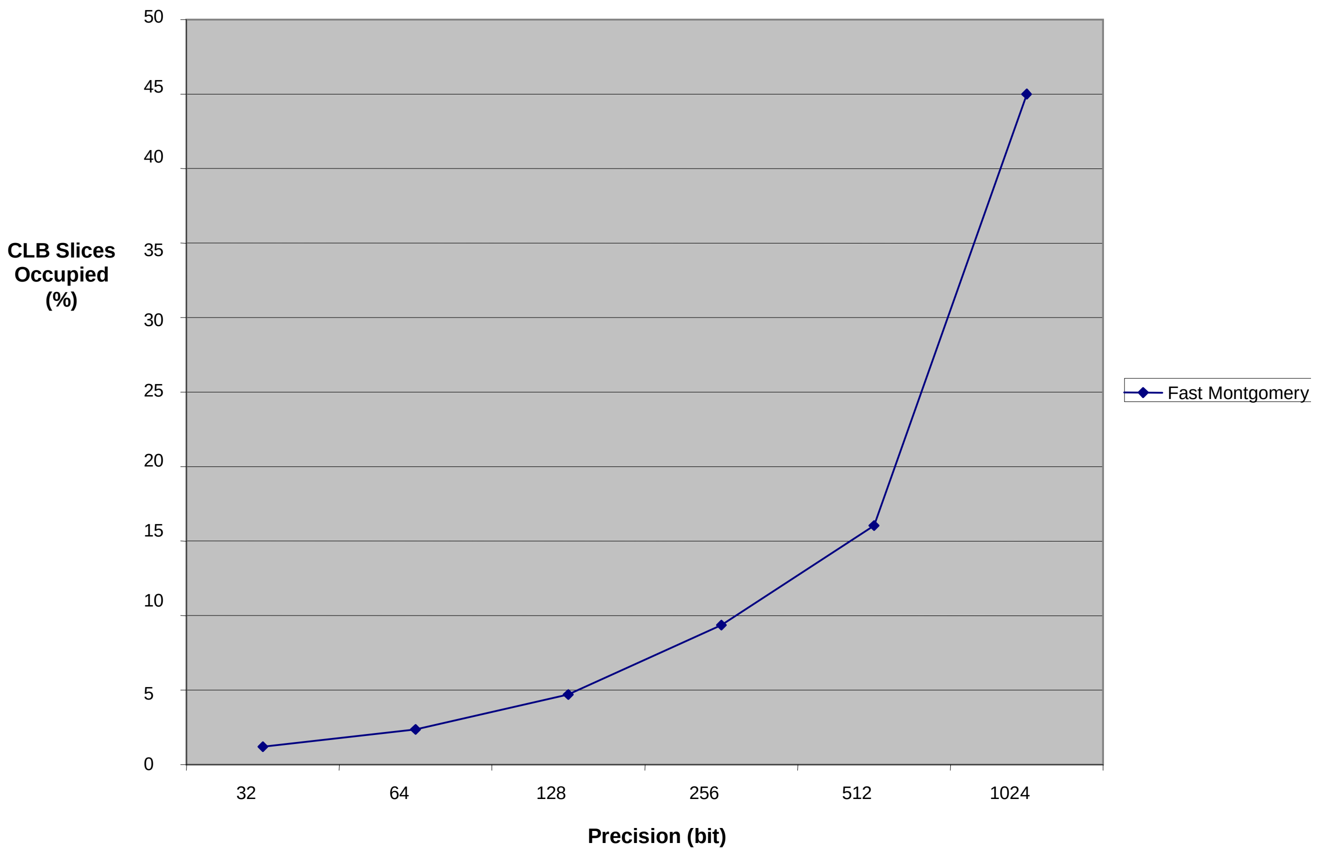


Fig. 20: CLB Slices versus bitlength for Fast Montgomery multiplier

6.3 Timing Analysis

The minimum clock period and absolute time for each different implementation are illustrated in Figures 21 and 22 respectively. The absolute time is derived from the minimum clock period by

Absolute time = (Minimum period)(No. of clock cycles).*

Where, *No. of clock cycles = bitlength (n) + 1*

The last clock cycle is used to trigger the values of *sum* and *carry* from the internal registers inside the loop to the outside for display or further post processing. In these implementations, post processing outside the loop is omitted.

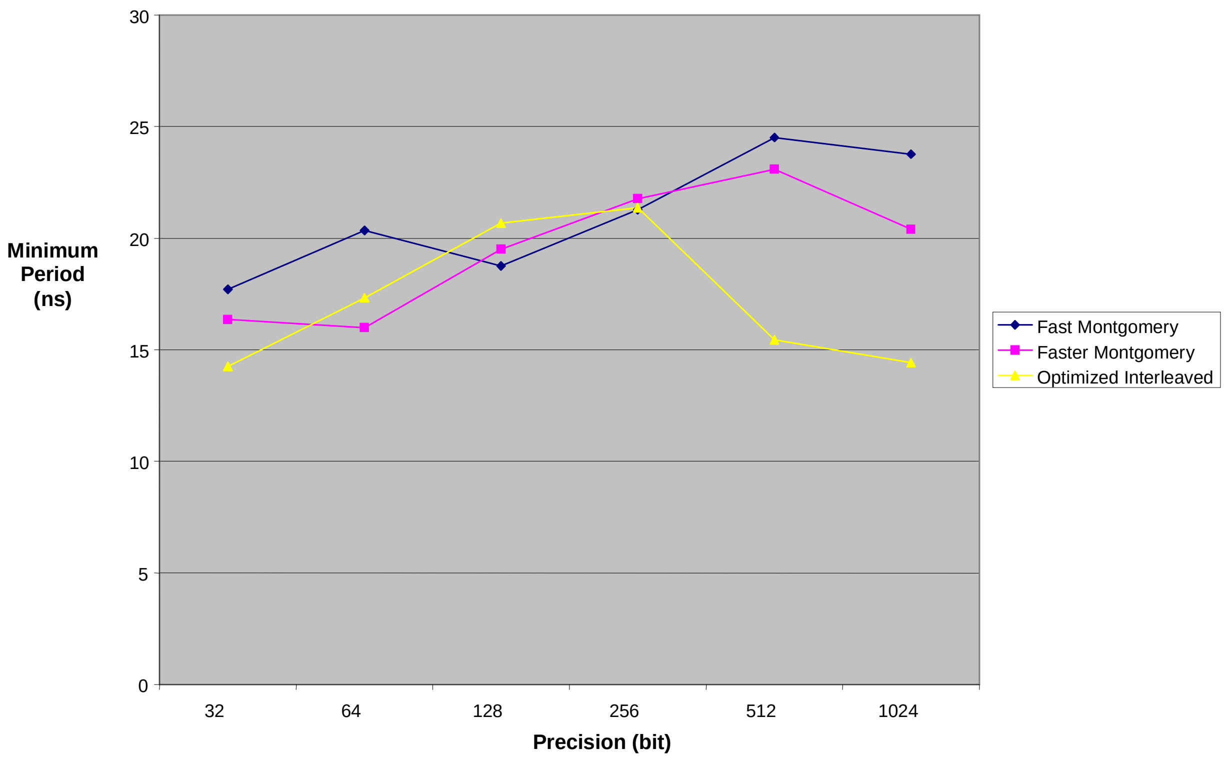


Fig. 21: Minimum clock periods for all implementations

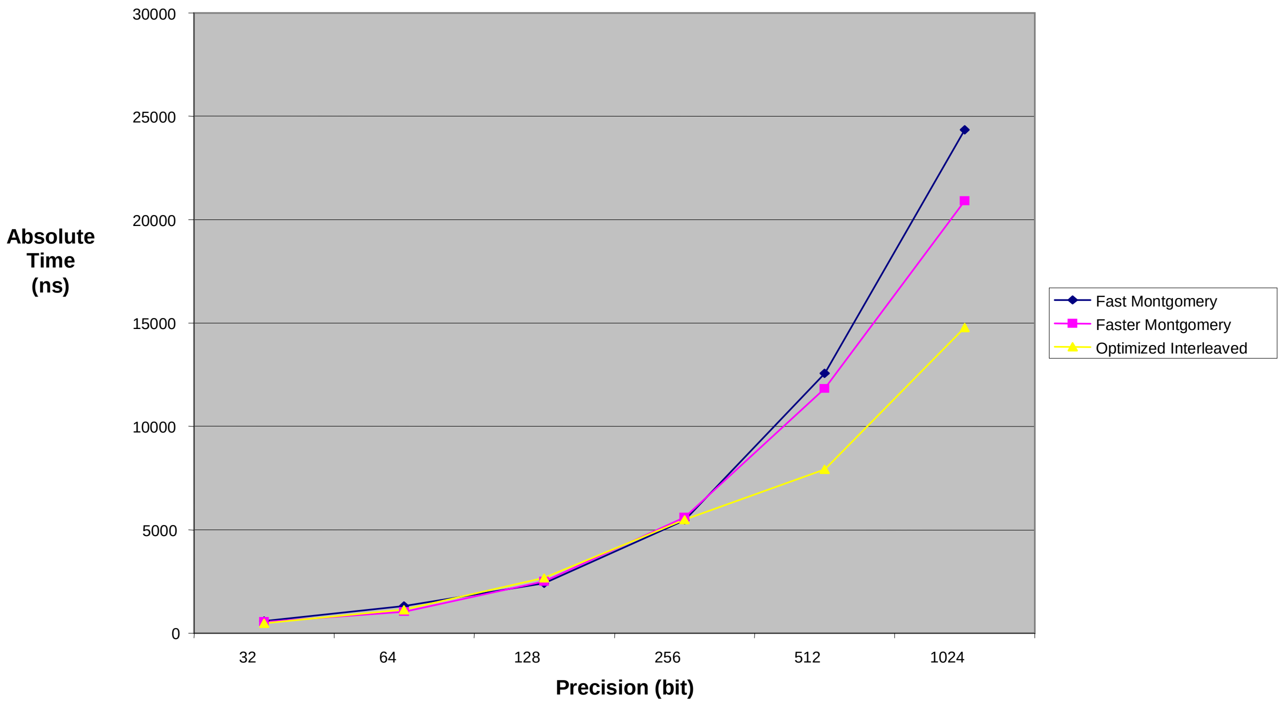


Fig. 22: Absolute times for all implementations

According to Table 8, the absolute time for each multiplier for all implementations holds a linear relationship with increasing bitlength. A rise in the bitlength by a factor of 2 leads to an approximate increase in the absolute time.

Precision	**Fast Montgomery**	**Faster Montgomery**	**Optimized Interleaved**
32 bits	584.562	539.913	469.986
64 bits	1321.775	1039.350	1125.995
128 bits	2417.589	2514.468	2667.333
256 bits	5466.647	5592.320	5491.062
512 bits	12575.169	11840.553	7922.259
1024 bits	24352.975	20907.950	14772.300

Table 8: Absolute time (ns) for all implementations

6.4 Area – Time (AT) Analysis

For a comparison of the implemented multipliers, we need a complexity model that allows for a realistic evaluation of time and area requirements of the considered architectures. As shown in Section 6.2 and 6.3, the architecture with the smallest area is Faster Montgomery. However, the optimized interleaved architecture is the most attractive in terms of speed of computation. Therefore, depending on one's requirements, a choice may be made between the two multipliers.

In Figure 23, the three implemented multipliers are examined in terms of the product of area and time. The percentage of CLBs occupied is used to represent the approximate area and the absolute times for computation are obtained using

Absolute time = (Minimum period)(No. of clock cycles).*

The product of area and time is computed from

Area-Time = (CLBs)(Absolute time).*

It is important to note that the use of a multiplier with Montgomery Algorithm requires post and pre-processing of the values (i.e., the conversion between Montgomery residues and the natural representation have to be taken into account). See Section 6.5 for a detailed explanation of RSA – example.

From the analysis of values in Table 9 and Figure 23, the following properties of the implemented multipliers were noted:

- The most inefficient multiplier in terms of the product of area and time for all bitlengths is the Fast Montgomery architecture.

- The most efficient multiplier in terms of the product of area and time for all implementations (i.e., bitlength ≥ 256 bit) is the optimized interleaved architecture.

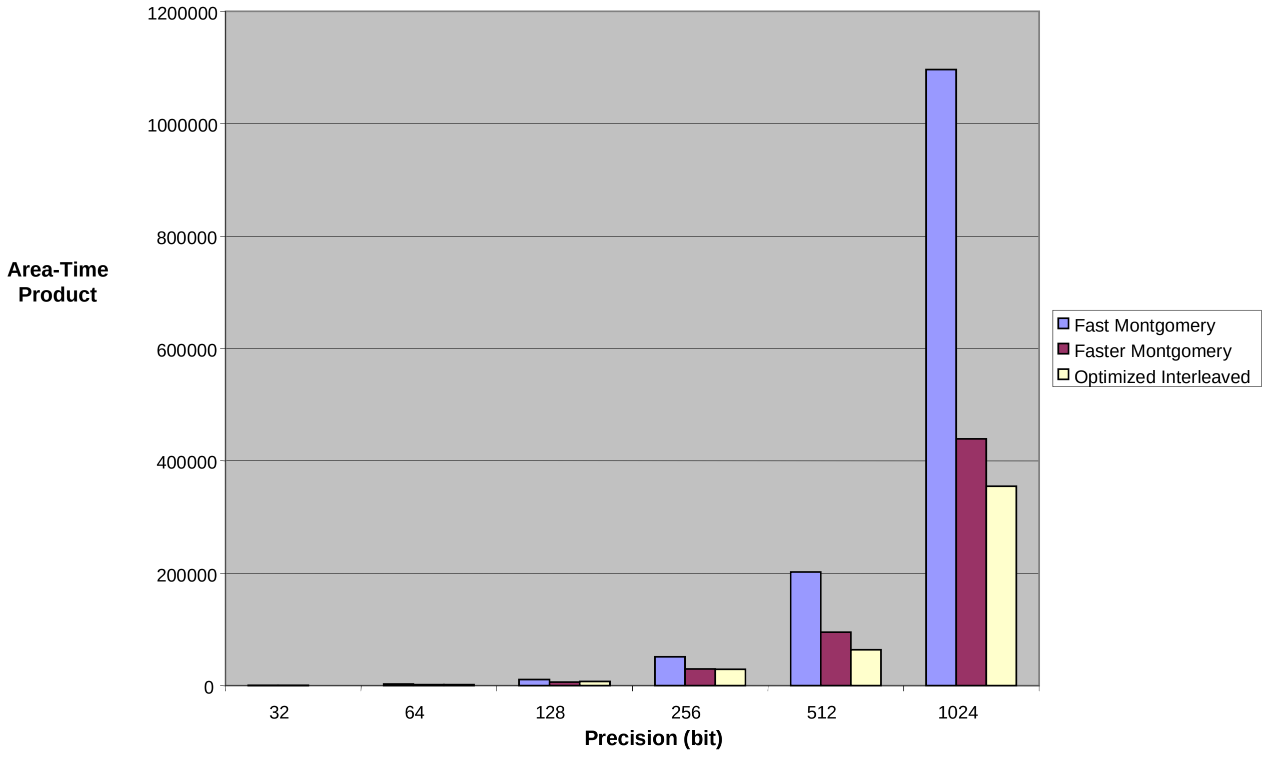


Fig. 23: Area – time product analysis

Precision	Fast Montgomery	Faster Montgomery	Optimized Interleaved
32 bits	689.783	377.939	319.590
64 bits	3106.171	1413.516	1531.353
128 bits	11338.492	6789.064	7201.799
256 bits	51222.482	30030.758	29432.092
512 bits	201831.463	95198.046	63694.962
1024 bits	1095883.875	439066.950	354535.200

Table 9: Area -time product values

6.5 RSA Encryption Time

For RSA encryption (e.g. 1024 bits) we need to compute

$$y = x^b \text{ mod } m$$

This can be done efficiently by the 'square and multiply algorithm' using

(1024)(1.5) = 1536 multiplications*

for a multiplier with the Optimized Interleaved Algorithm.

For a multiplier with Montgomery algorithm, we need to compute the Montgomery residue and convert back to the normal representation. With Montgomery residues, we can compute $y = x^b \text{ mod } m$ as follows:

- Compute Montgomery residue of x: $\boldsymbol{x'} = x\,(2^n)$ mod m.
- Compute whole exponentiation: $\boldsymbol{y'} = (\boldsymbol{x'})^b$ mod m.
- Convert from Montgomery residue back to normal representation:
  y = $\boldsymbol{y'}\,(2^{-n})$ mod m.

Thus, we need 1+1536+1 multiplications with Montgomery multipliers.

The time for a 1024 bit RSA encryption for the three different multipliers are shown in Table 10. The values were computed using

Time $_{\text{RSA encryption}}$ = (1536)*(T_{mul}), for multipliers with optimized interleaved algorithm;

and Time $_{\text{RSA encryption}}$ = (1538)*(T_{mul}), for Montgomery multipliers.

Where T_{mul} represents the absolute time for a 1024 bit multiplier.

Precision	**Fast Montgomery**	**Faster Montgomery**	**Optimized Interleaved**
1024 bit	37454875.55 ns	32156427.10 ns	22690252.8 ns

Table 10: Time (ns) for 1024 bit RSA encryption.

Chapter 7

Discussion

7.1 Summary and Conclusions

In this thesis, software (JAVA) and hardware (VHDL) implementations of existing and newly proposed algorithms and their corresponding architectures in [1] for performing modular multiplication have been done. In all, three different multipliers for 32, 64, 128, 256, 512 and 1024 bits were implemented, simulated and synthesized for a Xilinx FPGA. The implementations are scalable to any precision of the input variables X, Y and M.

This thesis also evaluated the performance of the multipliers in [1] by doing a thorough comparison of the architectures on the basis of the area-time product.

This thesis finally demonstrated with statistical figures that the newly optimized algorithms and their corresponding architectures in [1] for doing modular multiplication require minimum hardware resources and offer faster speed of computation compared to multipliers with the old Montgomery algorithm [1, Algorithm 1b].

The main advantages offered by the new algorithms are;

- Area requirements for the implementation of their architectures in hardware are significantly reduced.

- Latency for computation is tremendously improved. For example, a 1024 bit RSA data encryption can be performed in 22.69 ms with the optimized interleaved multiplier compared to 37.45 ms using a multiplier with the old Montgomery algorithm. These values hold a linear relationship with bitlength and will rise with increasing bitlength (e.g., bitlength > 1024).

- Depending on one’s requirements, the Faster Montgomery architecture [1, Algorithm 3] can be used for systems where area on chip is crucial. For applications where speed of computation is critical, the Optimized Interleaved algorithm is recommended.

- For applications where both area and time are limiting factors, the Optimized Interleaved architecture [1, Algorithm 5] offers a better performance compared to Fast Montgomery [1, Algorithm 1b], and Faster Montgomery [1, Algorithm 3].

- The new multipliers, namely, Faster Montgomery and Optimized Interleaved can be integrated into existing public-key cryptosystems such as RSA, DSA, and systems based on elliptic curve cryptography (ECC) to speed-up their arithmetic operations.

7.2 Further research

7.2.1 RAM of FPGA

In this thesis, data is loaded into the multipliers by means of registers with memory designed to hold the input data bits. An alternative approach is to
use the random access memory (RAM) blocks of the FPGA. This may reduce the percentage of CLBs required for their architectures.

7.2.2 Word Wise Multiplication

The algorithms presented in [1] require the full precision bit length of operands inside the multiplier. Research should also be aimed at investigating word wise multiplication. This would necessitate some modifications to the algorithms in [1].

REFERENCES:

[1] V. Bunimov, M. Schimmler, “Area-Time Optimal Modular Multiplication”, Embedded Cryptographic Hardware: Methodologie and Architectures, 2004, ISBN: 1 – 59454 – 012 - 8.

[2] P. L. Montgomery, “Modular multiplication without trial division”, Math. Computation, vol. 44, pp.519 - 521, 1985.

[3] G. R. Blakley, “A computer algorithm for the product AB modulo M”, IEEE Transactions on Computers, 32(5): pp 497 - 500, May 1983.

[4] K. R. Sloan, Jr. Comments on “A computer algorithm for the product AB modulo M”, IEEE Transactions on Computers, 34(3): pp 290 - 292, March 1985.

[5] T. Blum and C. Paar, “Montgomery modular exponentiation on reconfigurable hardware”, in Proc. 14th IEEE Symp. On Computer Arithmetic, 1999, pp. 70 - 77.

[6] S. E. Eldridge and C. D. Walter, “Hardware implementation of Montgomery’s modular multiplication algorithm”, IEEE Trans. Comput., vol. 42, pp. 693 - 699, June 1993.

[7] C. C. Yang, T. S. Chang, and C. W. Jen, “A new RSA cryptosystem hardware design based on Montgomery’s algorithm”, IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing, vol. 45, pp. 908 – 913, July 1998.

[8] C. K. Koc, “RSA Hardware Implementation”, RSA Laboratories, version 1.0 , August 1995.

[9] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, vol. 21, pp. 120126, February 1978.