AN ABSTRACT OF THE THESIS OF

Lo'ai Tawalbeh for the degree of Master of Science in Electrical & Computer Engineering presented on October 23, 2002. Title: Radix-4 ASIC Design of a Scalable Montgomery Modular Multiplier Using Encoding Techniques.

Redacted for privacy

Abstract approved: _____

Alexandre Ferreira Tenca

Modular arithmetic operations (i.e., inversion, multiplication and exponentiation) are used in several cryptography applications, such as decipherment operation of RSA algorithm, Diffie-Hellman key exchange algorithm, elliptic curve cryptography, and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm. The most important of these arithmetic operations is the modular multiplication operation since it is the core operation in many cryptographic functions.

Given the increasing demands on secure communications, cryptographic algorithms will be embedded in almost every application involving exchange of information. Some of theses applications such as smart cards and hand-helds require hardware restricted in area and power resources.

Cryptographic applications use a large number of bits in order to be considered secure. While some of these applications use 256-bit precision operands, others use precision values up to 2048 or 4096 such as in some exponentiation-based cryptographic applications. Based on this characteristics, a scalable multiplier that operates on any bit-size of the input values (variable precision) was recently proposed. It is replicated in order to generate long-precision results independently of the data path precision for which it was originally designed.

The multiplier presented in this work is based on the Montgomery multiplication algorithm. This thesis work contributes by presenting a modified radix-4 Montgomery

multiplication algorithm with new encoding technique for the multiples of the modulus. This work also describes the scalable hardware design and analyzes the synthesis results for a 0.5 μm CMOS technology. The results are compared with two other proposed scalable Montgomery multiplier designs, namely, the radix-2 design, and the radix-8 design. The comparison is done in terms of area, total computational time and complexity.

Since modular exponentiation can be generated by successive multiplication, we include in this thesis an analysis of the boundaries for inputs and outputs. Conditions are identified to allow the use of one multiplication output as the input of another one without adjustments (or reduction).

High-radix multipliers exhibit higher complexity of the design. This thesis shows that radix-4 hardware architectures does not add significant complexity to radix-2 design and has a significant performance gain.

Radix-4 ASIC Design of a Scalable Montgomery Modular Multiplier Using Encoding Techniques

by

Lo'ai Tawalbeh

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented October 23, 2002
Commencement June 2003

Master of Science thesis of Lo'ai Tawalbeh presented on October 23, 2002

APPROVED:

Redacted for privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for privacy

Chair of the Department of Electrical & Computer Engineering

Redacted for privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Lo'ai Tawalbeh, Author

ACKNOWLEDGMENT

I would like to thank Dr. Tenca and Dr. Koç for giving me the opportunity to work on this interesting subject. Dr. Tenca's discussions, directions, reviews and valuable comments helped me too much in accomplishing this work.

I would also like to thank Georgi Todorov who gave me a chance to use his design for the purpose of comparison.

After all, I want to give my biggest thanks to my family for their support and patience during my study overseas.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

RADIX-4 ASIC DESIGN OF A SCALABLE MONTGOMERY MODULAR MULTIPLIER USING ENCODING TECHNIQUES

1. INTRODUCTION.

Modular arithmetic operations (i.e., addition, multiplication and inversion) are used in several cryptography applications, such as decipherment operation of RSA algorithm [1], Diffie-Hellman key exchange algorithm [2], elliptic curve cryptography [3], and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm [4]. The most important of these three arithmetic operations is the modular multiplication operation since it is the core operation in many cryptographic functions.

Given the increasing demands on secure communications, cryptographic algorithms will be embedded in almost every application involving exchange of information. Some of theses applications, such as smart cards [5] and hand-helds, require hardware restricted in area and power resources [6].

An efficient algorithm to implement modular multiplication is the Montgomery Multiplication algorithm [7], and it has many advantages over ordinary modular multiplication algorithms. The main advantage is that the division step in taking the modulus is replaced by shift operations which are easy to implement in hardware [6].

Cryptographic applications use large number of bits in order to be considered secure. Some of these applications use 256-bit precision operands, others use larger precision, up to 2048 or 4096, as in some exponentiation-based cryptographic applications [8].

Many of the proposed designs are fixed-precision [9] which uses operands of fixed size. Other designs are scalable [10, 11], and can take operands with an arbitrary precision.

An important factor that should be taken into consideration is the area/time tradeoff [12]. In general the fastest design is better, but most of the fast designs use large area and more complicated logic.

This thesis presents a modification of a radix-4 Montgomery multiplication algorithm (as obtained from [11]) which involves an encoding step for the multiples of the modulus. This work also describes the scalable (variable-precision) hardware design and analyzes the synthesis results for a .5 μm. The results are compared with two other proposed scalable Montgomery multiplier designs, namely, the radix-2 design presented in [10], and the radix-8 design presented in [6]. The comparison is done in terms of area, total computational time and complexity.

Next section of this chapter presents the definition of Montgomery multiplication. Section 2 explains radix-4 Booth encoding with an example. Section 3 reviews the high-radix Montgomery multiplication algorithm originally presented in [11]. Section 4 talks about the motivation for radix-4 design. A brief justification for the use of the extra level of encoding proposed in this work is given in Section 5. The organization of this thesis is presented in Section 6.

1.1. Montgomery Multiplication

The Montgomery multiplication algorithm generates the products of two n-bit integers X (multiplier) and Y (multiplicand) in modulo M according to the following expression:

$$MM(X,Y) = XYr^{-1} \ mod \ M$$

where $r = 2^n$. M is chosen such that the greatest common divisor of r and M is one ($\gcd(r,M) = 1$). In other words, r and M should be relatively prime. This condition is easily achieved by choosing M as an odd integer, since $r = 2^n$ is an even number. We usually have $2^{n-1} < M < 2^n$. The Montgomery image of an integer can be obtained by multiplying it by the constant r and taking it modulo M: $\bar{a} = ar \ mod \ M$.

The Montgomery multiplication over the images $\bar{a}$ and $\bar{b}$ results in:

$$\bar{c} = cr \ mod \ M = MM(\bar{a}, \bar{b}) = abr \ mod \ M$$

which corresponds to the image of $c = ab \ mod \ M$, the modular product of a and b.

Figure 1.1 shows the transformation between the integers and their images performed using MM. This process can be explained as follows:

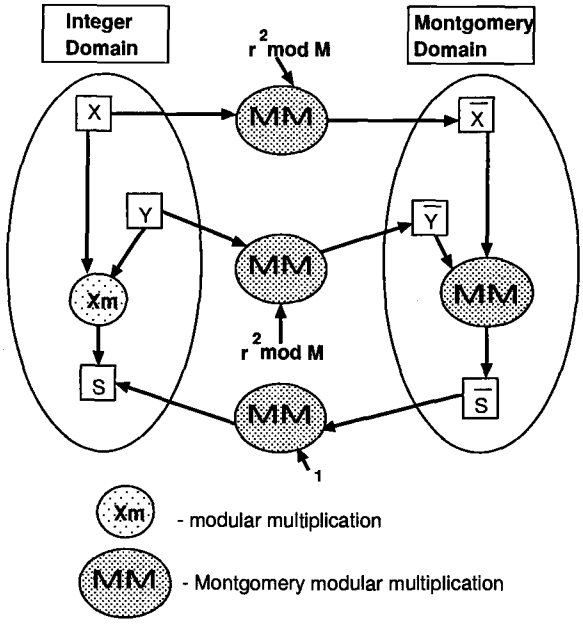

FIGURE 1.1: Modular multiplication using MM.

- to transform an integer a to its image $\bar{a}$, we do: $\bar{a} = MM(a, r^2) = ar^2r^{-1} \ modM = ar \ mod \ M$.

- to transform from an image $\bar{a}$ to the integer a, we compute: $a = MM(\bar{a}, 1) = arr^{-1} \ mod \ M = a \ mod \ M$.

Observe that the constant $r^2 \ mod \ M$ is pre-computed and used in the process as shown in Figure 1.1.

Step

1: $S := 0$

2: FOR $i := 0$ TO n-1

3: $(S := S + x_i Y)$

4: IF $s_0 = 1$ THEN

5: $S := S + M$

END IF

6: $S := S/2$

END FOR

7: IF $S \geq M$ THEN $S := S - M$

END IF

FIGURE 1.2: Radix-2 Montgomery Multiplication (R2MM) algorithm

The Radix-2 Montgomery Multiplication algorithm is shown in Figure 1.2. Each iteration takes one bit x_i of $X = (x_{n-1}, ..., x_1, x_0)$ and multiply it by Y. S accumulates the partial product. If the least significant bit of S is 1 (s_0), the modulus M is added to the result (step 5) to make the least significant bit of S equal to zero. With this condition a shift operation (step 6) may be executed to keep S inside a small interval. The algorithm consists of simple operations that can be implemented easily in hardware. When the algorithm reaches step 7, S is in the range [0,2M-1].

1.2. Booth Encoding of Multiples

The number of iterations is reduced when higher radix is used in the representation of the multiplier. Each computation step uses k-bits of the multiplier in radix 2^k. This group of bits forms a digit of the multiplier and has 2^k values. For radix 4, with conventional representation of the multiplier digit, the digit values are 0,1,2, and 3. The

generation of the multiples Y and $2Y$ is simple in hardware, but the multiple $3Y$ requires an addition of Y and $2Y$. In order to avoid this extra complexity, it is possible to recode the multiplier into the digit set $\{-2, -1, 0, 1, 2\}$ using Booth encoding [13]. Booth encoding is applied to a bit vector to reduce the complexity of multiple generation in the hardware [11]. Considering that Z_i represents an encoded digit i of radix-4, Booth function for radix-4 digit $X_i = (x_{2i+1}, x_{2i})$ is given as:

$$Z_i = Booth(X_i, x_{2i-1}) = Booth(x_{2i+1..2i-1}) = -2x_{2i+1} + x_{2i} + x_{2i-1}$$

where i = 0,1,2,..., $\frac{n}{2}$-1, and x_{2i-1} is the most significant bit of the previous digit, and the $x_{2i+1..2i-1}$ represents all the bits from x_{2i+1} to x_{2i-1}.

As an example, lets consider the number $X = 125$. In binary, it is represented using eight bits as

$$X = 01111101 = \sum_{j=0}^{7} (2)^j * x_j,$$

where x_i is a single bit of X at position i. The rightmost bit corresponds to position 0.

The radix-4 Booth encoded digits of X are:

$$\begin{aligned} Z_0 &= Booth(x_{1..-1}) = 1 \\ Z_1 &= Booth(x_{3..1}) = -1 \\ Z_2 &= Booth(x_{5..3}) = 0 \\ Z_3 &= Booth(x_{7..5}) = 2 \end{aligned}$$

Notice that the bits of X used to determine Z overlap in one position. So, coefficients Z_1 and Z_2 share the bit x_3. The first digit (Z_0) needs a non-existing bit $x_{-1} = 0$ in order to be generated. Then the representation for X using Booth encoding is:

$$X = Z = (Z_3, Z_2, Z_1, Z_0) = \sum_{j=0}^{3} (2^2)^j * Z_j.$$

It is true that $X = 1 - 1 * (2^2) + 0 * (2^2)^2 + 2 * (2^2)^3 = 125$.

Step

1: $S := 0$

$x_{-1} := 0$

2: FOR $j := 0$ TO N - 1 STEP k

3: $q_{Y_j} = Booth(x_{j+k-1..j-1})$

4: $S := S + (q_{Y_j} * Y)$

5: $q_{M_j} := S_{k-1..0} * (2^k - M^{-1}_{k-1..0}) \ mod \ 2^k$

6: $S := signext(S + q_{M_j} * M)/2^k$

END FOR;

7: IF $S \geq M$ THEN $S := S - M$

END IF;

FIGURE 1.3: High-Radix ($Radix - 2^k$) Montgomery Multiplication (R2^{k}MM) Algorithm

1.3. Word-based High-Radix ($Radix-2^k$) Montgomery Multiplication (R2^{k}MM) Algorithm

The generic algorithm used as the base for the radix-4 algorithm presented in this work is shown in Figure 1.3. It is presented and proven to be correct in [11]. The multiplicand Y and the modulus M are divided into words. The parameter k changes depending on how many bits of the multiplier X will be scanned during each computational loop [6]. Also, k determines the radix of computation. For radix 2, one bit of X is scanned, $k = 1$; for radix 8, three bits of X are scanned, k =3; and so on.

Booth encoding is used to recode the multiplier X. The next step is to add multiples of Y ($q_Y Y$) to the partial product S, which is being shifted right by k-bits and sign extended (step 6 of R2^{k}MM algorithm). To avoid data loss, the least significant

k-bits of S are made zero before shifting. This is done by adding multiples of M ($q_M M$) to S.

1.4. Motivation for Radix 4

When applying Booth encoding to a k-bit digit, the resulting encoded digit value is in the range [-2^{k-1},2^{k-1}]. For radix 8, k=3 and the encoded multiplier digit is in the range [-4,4]. The implementation of some values like -3 and 3 increases the complexity of the design. The radix-8 design proposed by [6] uses two four-input muxes to generate the multiples of X. This forces the use of 4-2 Carry-Save Adders to perform addition, which increases the area and the critical path delay when compared to using 3-2 Carry-Save Adders. More details are presented in [6]. On the other hand, k=2 in radix 4, and so, the encoded digit of the multiplier is in the range [-2,2], which can be implemented in easier and less costly way. In addition, Carry-Save adders are used for addition making radix-4 addition process as fast as the addition in the radix-2 design. This makes the two designs close in area. On the other hand, radix-4 design is twice faster than radix-2 (since radix-4 algorithm scans 2 bits of the multiplier at a time, which reduces the total number of computation cycles to half of what is needed for radix 2). The complete design is described in Chapter 3.

At a certain step of the Montgomery multiplication algorithm, multiples of the modulus M should be added to the partial product. For radix 8, multiples of M are in the range [0,7]. Generating the values 5 and 7 adds extra logic to the system. While in radix 4 the multiples are in the range [0,3]. the generation of $3Y$ is a problem that we solve using another level of encoding for multiples of M.

The total computational time of any of the three designs depends on the number of clock cycles needed to complete the computation, and the clock period. The critical path delay and the area affects the clock period (long wires add parasitic resistance). Radix-4 design has less critical path delay, less area and complexity than radix 8.

1.5. Extra Level of Encoding

The high-radix Montgomery multiplication algorithm presented in [11] uses Booth encoding to recode the multiplier digits.

The radix-4 Montgomery multiplication algorithm presented in the next chapter of this thesis, has an extra level of encoding (Encoding2). The Encoding2 function is applied to the algorithm to simplify the generation of the modulus multiples.

As mentioned in the above section, the multiples of the modulus M is in the range [0,3]. The multiples M and $2M$ are easily generated in hardware. While the multiple $3M$ needs addition of M and $2M$. This addition step is in the critical path. Encoding2 is applied to avoid the generation of this multiple. This way, the generation of multiples of M will not affect the critical path anymore. More details are presented in the next Chapter.

1.6. Thesis Organization

The remainder of this thesis work is organized in 5 chapters. Chapter 2 presents Multiple-word Radix-4 Montgomery Multiplication (R4MM) algorithm with encoding. Chapter 3 presents a detailed description of the architecture and implementation of the radix-4 Montgomery multiplier. Chapter 4 describes briefly two previous designs, radix-2 and radix-8, and compare them with radix-4 design. The experimental results of the three designs are presented and compared in Chapter 5. Chapter 6 concludes this work.

2. MULTIPLE-WORD RADIX-4 MONTGOMERY MULTIPLICATION (R4MM) ALGORITHM.

This Chapter presents a radix-4 Montgomery multiplication algorithm with a second level of encoding. Definition of scalable architecture and multi-precision addition is presented in Section 2. The Chapter ends with a brief review of related work.

2.1. Multiple-word Radix-4 Montgomery Multiplication (R4MM) Algorithm Using Extra Encoding

The notation used in the algorithm presented in this section is shown in Table 2.1.

Figure 2.1 shows a multiple-word Radix-4 Montgomery Multiplication algorithm (R4MM), an extension of the Multiple-Word High-Radix ($R2^k$) Montgomery Multiplication algorithm ($MWR2^kMM$) presented and proved to be correct in [11].The (R4MM) uses an extra encoding step for the multiples of the modulus M, which wasn't used before in $MWR2^kMM$.

There are two types of encoding applied in the R4MM. The first one is Booth encoding [13] applied to the multiplier X, as explained in the previous Chapter. The second level of encoding (Encoding2) is applied to multiples of the modulus M, and will be explained in the next subsection.

The combination of a radix-4 digit at position i (X_i) and the most significant bit of a radix-4 digit at position i-1 is called $X_{EXT_i} = (X_i, x_{2i-1})$ which will be used by Booth encoding to determine the encoded radix-4 digit, as shown in Figure 2.1. The two carry bits C_a and C_b are propagated from the computation of one word to the computation of the next word. In order to make the least-significant 2 bits of S all zeros, a multiple of M, namely $q'_{M_j}M$, is added to the partial product [11]. This step is required to make sure that there are no significant bits lost in the shift operation performed in step 10.

- X_j- a single radix-4 digit of X at position j;
- q_{M_j} - quotient digit that determines a multiple of the modulus M which is added to the partial product S in the j^{th} iteration of the computational loop;
- q'_{M_j}- encoded digit of q_{M_j};
- w - number of bits in a word of either Y, M or S;
- $e = \lceil \frac{n+1}{w} \rceil$ - number of words in either Y, M or S;
- NS - number of stages;
- C_a, C_b - carry bits;
- $(Y^{(e-1)}, ..., Y^{(1)}, Y^{(0)})$ - operand Y represented as multiple words;
- $S^{(i)}_{k-1..0}$ - bits k - 1 to 0 of the i^{th} word of S.

TABLE 2.1: Notation

To compute the digit q'_{M_j} we need to examine the least 2-bits of the partial product S generated in step 4 of the (R4MM) algorithm.

In the next subsection we will talk more about the possible values of q'_{M_j} and how we encode it to another digit set in order to simplify the design.

The most significant (MS) word of S is generated in step 11, and since negative values of S are now possible, the sign extension operation is performed in step 12. The partial product S might have negative intermediate values as a result of using Booth encoding for the multiplier and the second (extra) encoding for multiples of the modulus. More about the boundaries of the partial product S is discussed in subsection 2.1.2. The final reduction step (step 7 in the radix-2^k Montgomery multiplication algorithm shown in Figure 1.3) was intentionally not included.

It is shown in [11] that q_M, as computed in step 5, satisfies the relation:

$$q_M * M = -S \ mod \ 4$$

which can be rewritten as:

$$S_{1..0} + q_M * M_{1..0} = 0 \ mod\ 4$$

and represents the requirement that the last 2 bits of S must be zeros.

It is easy to show from Booth encoding properties that the multiplier X is represented by digits of Z_i (Section 1.2).

However, it is still necessary to show that applying Encoding2 (step 5a of R4MM) and using the encoded digit q'_{M_j} still generates an equivalent result. In order to prove that this algorithm is correct, we need to have $q'_{M_j} \equiv q_{M_j}\ mod\ 4$.

2.1.1. Encoding of q_{M_j}

The values for the quotient digit q_{M_j} are in the set $\{0, 1, 2, 3\}$. Applying an encoding function (Encoding2) we transform the quotient digit q_{M_j} to the digit set $\{-1, 0, 1, 2\}$. It consists in replacing $q_{M_j} = 3$ by the encoded value q'_{M_j} = -1. It makes the generation of multiples of M less complex.

The proof that the algorithm is still correct after Encoding2 comes from the fact that

$$3 \equiv -1\ mod\ 4$$

and thus,

$$q'_{M_j} \equiv q_{M_j}\ mod\ 4 \rightarrow q'_{M_j} * M \equiv q_{M_j} * M\ mod\ 4.$$

In fact steps 5 and 5a of the R4MM algorithm are done at the same time.

2.1.2. Boundaries for The Partial Product S

The radix-2^k Montgomery multiplication algorithm takes two operands X and Y and compute $MM(X, Y) = XY2^{-m}\ mod\ M$, where m is positive integer greater than n (operands precision), and M is the modulus. The value of the partial product S at a

given iteration i, may be expressed by:

$$S = (S + z_i Y + q_i' M)/4$$

where $0 \leq i \leq p\text{-}1$, and $p = \lceil \frac{m+1}{2} \rceil$, which is the number of radix-4 digits being considered. Observe that $m+1$ bits are considered to account for the sign.

Using the recoding scheme proposed in this work, there is an invariant for each iteration of the for loop given as $|S| < \frac{2}{3}M + \frac{2}{3}Y$.

Proof: after the first loop iteration, the value of S is given as $S = z_i Y + q_i' M$. The values of z_i are in the range [-2,2] , and q_i' is in the range [-1,2] after applying recoding. The maximum positive value for z_i is 2, and so, the maximum positive value that $z_i Y$ can get after the first iteration is $(2)Y/4$. The second iteration adds up to $2Y/4^2 + 2Y/4$, and after p iterations we get:

$$\sum_{i=0}^{p-1} 4^i (2) \frac{Y}{4^p} = \frac{2Y}{4^p} \sum_{i=0}^{p-1} 4^i$$

Knowing that $\sum_{i=0}^{p-1} 2^{ik}(2^k - 1) = (2^{pk} - 1)$, then $\sum_{i=0}^{p-1} 2^{ik} = \frac{(2^{pk}-1)}{2^k-1}$, and so, the above summation results in:

$$\frac{2Y}{4^p}\left(\frac{4^p - 1}{4 - 1}\right) = \frac{2}{3}\left(Y - \frac{Y}{4^p}\right)$$

but since $2p = m+1$ (the maximum number of bits in the operands is less or equal to this value), the result of $Y/4^p$ tends to zero. Thus, the addition of $z_i Y$ results in at most $\frac{2}{3}Y$. Since the maximum positive value for q_i' is also 2, the same reasoning is used for $q_i' M$, and at most it will sum to $\frac{2}{3}M$. Similar calculations can be done to the negative range of values, however q_i' = -1 is the most negative value and $-\sum_{i=0}^{p-1} 4^i \frac{M}{4^p} = -\frac{M}{4^p}\left(\frac{4^p-1}{4-1}\right) = -\frac{1}{3}M$, and so, $-\frac{1}{3}M - \frac{2}{3}Y < S$. Therefore, $|S| < \frac{2}{3}M + \frac{2}{3}Y$ after each loop.

Multiplication can be used in exponentiation. The result of one multiplication can be applied as input to another multiplication.

Using the conditions

$$|X| < M < 2^{m-3}$$

$$|Y| < M$$

and $R = 2^m$, we are able to show that $|S| < M$.

Proof: In this case, the MS digit of the recoded X, Z_{p-1}, is zero, since $|X| < 2^{m-3} = 2^{2p-4} = 2^{2(p-2)} = 4^{(p-2)}$, (notice that $m = 2p-1$), and Z can have at most the digit $Z_{p-2} \neq 0$ (forced by sign and recoding scheme). With this condition, the sum of the maximum positive values for $z_i Y$ results in:

$$\sum_{i=0}^{p-2} 4^i(2)\frac{Y}{4^p}$$

$$\frac{2Y}{4^p}\sum_{i=0}^{p-2} 4^i$$

$$\frac{2Y}{4^p}\left(\frac{4^{(p-1)}-1}{3}\right) = 2/3\frac{Y}{4} - 2/3\frac{Y}{4^p} = 2/3\frac{Y}{4} = \frac{Y}{6}$$

Thus, the condition after the last iteration of the for loop is: $S < \frac{2}{3}M + \frac{1}{6}Y$ and since $Y < M$ then $S < \frac{2}{3}M + \frac{1}{6}M = \frac{5}{6}M$, and consequently $S < M$.

From the symmetry of the values of z_i and by using the same procedure we can prove that $S > -M$. So, $|S| < M$ when $|X| < M$ and $|Y| < M$.

As a result, no reduction is needed when the radix-4 algorithm is used and two extra digits of X is considered.

2.2. Scalable Multiplier Architecture.

A scalable arithmetic unit can be reused in order to generate long-precision results independently of the data path precision for which the unit was originally designed [8]. To speed up the multiplication operation, various dedicated multiplier modules were developed [10, 14], which use fixed-precision operands. They are fixed-precision designs because a multiplier designed for n bits cannot be immediately used in a system which requires $k > n$ bits, forcing a complete redesign [10]. The multiplier presented in [8] use processing elements that can be adjusted in size and number in order to fit into a given area and also explore the parallelism of the operations in the Montgomery multiplication algorithm.

Multiplying two n-bit operands at one computation cycle will be time consuming, requires a significant amount of hardware, and is complex to design, especially for large values of n. To solve this problem, multi-precision addition is used in the scalable Montgomery multiplication algorithm and architecture. In multi-precision addition, the n-bit operands are divided into words of a certain size (w). Then the multiplication is applied to theses words instead of the whole n-bit vector, and the partial results are added. Using carry propagate adders to add the partial products would result in a long critical path delay. To avoid carry propagation, the partial products are represented using Carry-Save representation. The result will be converted to non-redundant representation only at the end of computation.

2.3. Literature Review for Montgomery Multiplication.

A high-radix Montgomery multiplication algorithm is described and mathematically proven to be correct in [15]. A radix-8 implementation of modular multiplication was proposed in [11]. The proposed design has less total computational time compared to radix-2. On the other hand, there was a significant increase in area and complexity.

Any implementation of Montgomery multiplication should consider the tradeoff between chip area and computational speed [11, 12]. The multiplier is scanned faster by increasing the radix, however, the determination of the q_M quotient digit becomes more complex. Thus, the overall effect on the computational time has to be investigated in detail [11].

Simplifying the determination of q_M in high-radix modular multipliers is discussed in [16]. The simplification includes transforming the modulus M. The intermediate steps of addition and modular reduction are simplified for the cost of additional pre-processing and a wider range of the final result [11].

A flexible multiplier can be integrated into a system as an autonomous co-processor attached to the system bus [8, 17]. Also, the multiplier can be integrated as a functional

unit to the main CPU. With the idea of implementing more cryptographic operations in hardware, this approach is becoming increasingly attractive.

A single chip, 1024-bit RSA implementation is shown in [18]. The multiplication part is implemented as an array multiplier. This approach for multiplication requires multiple clock cycles to complete. Another approach to perform modular multiplication is to use a core with a small bit size and reuse it with bit portions of the operands [11]. It is shown in [10] that limiting the size of the computing unit has certain advantages. Thus, the second approach is attractive because of reusing fixed core many times, and so, this approach is used in this thesis work.

Implementing the multiplier using reconfigurable hardware provides the means of solving problems for both high-precision and variable-precision computation [11]. The main candidates for flexible hardware are FPGAs [19, 17]. It is pointed out in [17] that a flexible design would have flexibility and adaptability comparable to conventional software and good performance because of the hardware speed. A 12×12 bits modular multiplier implementation based on Montgomery multiplication algorithm is presented in [20].

An approach for modular multiplication based on residue arithmetic is presented in [21]. The multiplication algorithm is distributed among a ring of processors. Each processor operates on a set of data, then forwards this data to the next processor.

A unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$ is presented in [8]. It is shown that a Montgomery multiplication module can operate in both fields without significant increases in the design area compared to a multiplier that works on $GF(p)$.

Step

1: $S := 0$

$x_{-1} := 0$

2: FOR $j := 0$ TO N - 1 STEP 2

3: $Z_j = Booth(x_{j+2-1..j-1}) = Booth(X_{EXT_j})$

4: $(Ca, S^{(0)}) := S^{(0)} + (Z_j * Y)^{(0)}$

5: $q_{M_j} := S^{(0)}_{1..0} * (4 - M^{(0)^{-1}}_{1..0}) \ mod \ 4$

5a: $q'_{M_j} := Encoding2[q_{M_j}]$

6: $(Cb, S^{(0)}) := S^{(0)} + (q'_{M_j} * M)^{(0)}$

7: FOR $i := 1$ TO e - 1

8: $(Ca, S^{(i)}) := Ca + S^{(i)} + (Z_j * Y)^{(i)}$

9: $(Cb, S^{(i)}) := Cb + S^{(i)} + (q'_{M_j} * M)^{(i)}$

10: $S^{(i-1)} := (S^{(i)}_{1..0}, S^{(i-1)}_{BPW-1..2})$

END FOR;

11: $Ca := C_a \ or \ C_b$

12: $S^{(e-1)} := signext(Ca, S^{(e-1)}_{BPW-1..2})$

END FOR;

FIGURE 2.1: Multiple-word Radix-4 Montgomery Multiplication (R4MM) Algorithm.

3. DESIGN OF A RADIX-4 MONTGOMERY MULTIPLIER.

This Chapter presents the top level description of the radix-4 scalable Montgomery multiplier and its main functional blocks. The architecture and logic design are described in detail.

3.1. Overall Organization

The top level design of a Montgomery multiplier implementing the R4MM is shown in Figure 3.1. The main functional blocks are *Kernel Datapath*, *IO & Memory* and the *Control* block. The computation takes place in the kernel datapath according to the R4MM algorithm. The control signals for the kernel datapath and the registers between the kernel and the *IO & Memory* block are provided by the control block.

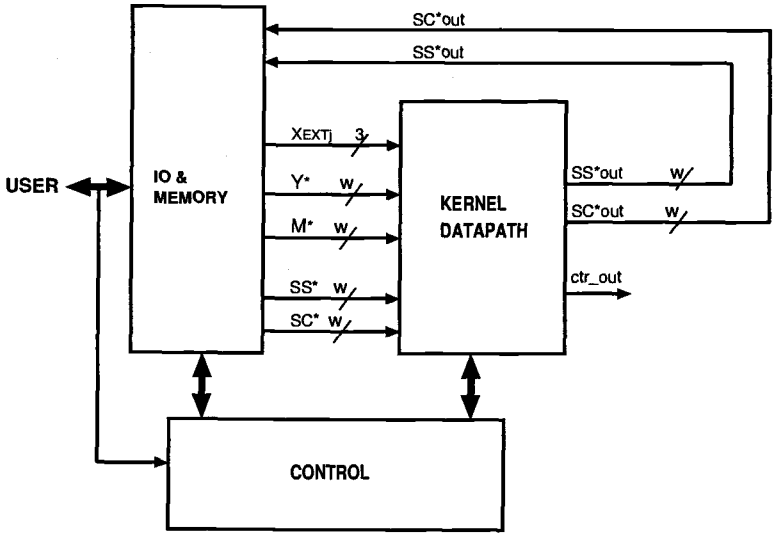

FIGURE 3.1: System Level Diagram of Modular Multiplier.

To avoid carry propagation during word addition, Carry-Save Adders (CSA) are used, and S is kept in Carry-Save (CS) form. The final result is converted to non-redundant form only at the end (using a CS converter inside the IO & $Memory$ block).

The bits of X needed to compute Z_j (step 3 in the R4MM) are provided by the signal X_{EXT_j}.

Other inputs to the kernel datapath are w-bit words of the multiplicand Y, modulus M, and the partial product S (which is represented in Carry-Save form as two vectors SS and SC). All these signals are provided by the IO & $Memory$ block. To identify one word of a bit-vector, the superscript star (*) was used. For example, $M^{(*)}$ represents one word of vector M. A new word of Y, M, SS, and SC is applied to the kernel in every clock cycle. The kernel was designed in such a way that it has two configuration papameters, the number of stages (NS) and the word size (w). The operands must pass through the datapath several times depending on the values of these two parameters [10, 11] and the precision of the operands.

Using multiplexers (MUXs) and shifters, we are able to generate words of the multiples required in the computation, $(Z_j * Y)^{(*)}$ and $(q'_{M_j} * M)^{(*)}$.

The IO & $Memory$ block provides the interface between the user and the memory elements for the operands, modulus, and partial result. The only requirement for this block is to meet the timing specifications for the Kernel. Therefore, there are many different flexible solutions to implement this block, depending on the system's architecture in which the multiplier will be integrated. So, the architecture of this functional unit is out of the scope of this work.

The kernel datapath is organized as a pipeline of Montgomery Multiplication cells, also called Processing Elements (PE), separated by registers (Figure 3.2). Each PE implements one iteration of the FOR loop (steps 3 to 12) in the R4MM algorithm. A stage consists of a PE and a register. At each clock cycle, one word of Y, M, SS, and SC are applied as inputs to a stage. Additionally, $(NS * 2)$ bits of X are transferred to the kernel over 2*NS clock periods, where NS corresponds to the number of stages. Each stage needs these bits at different times, thus, this signal is made common for all stages

with internal control loading the signal in the right stage at the right time. The pipeline outputs are $SS_{OUT}^{(*)}$ and $SC_{OUT}^{(*)}$.

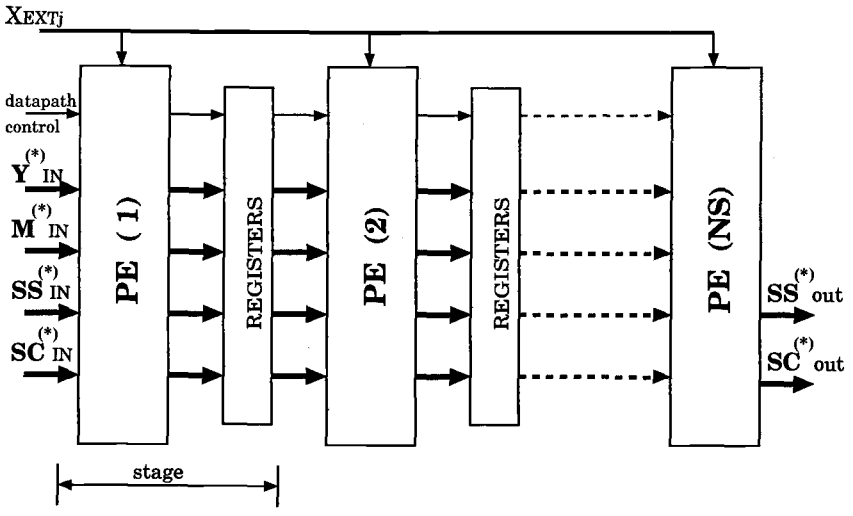

FIGURE 3.2: Top Level Diagram of Kernel datapath.

The newly computed words of SS and SC, in addition to the words of Y and M, are propagated by each PE to the next PE, which performs another computational loop of the Montgomery multiplication algorithm and on its turn propagates the same type of data to the following PE after a latency of 2 cycles. In order to complete the multiplication, the data must flow through the pipeline several times.

3.2. Radix-4 PE Design

The kernel processing element is organized as shown in Figure 3.3. The Figure shows the main blocks in the design: booth encoding, multiple generation, adders, generation of q'_{M_j}, and registers (shaded boxes). Shifting and alignment is done by proper combination of signals. The design uses a re-timing technique explained in [11].

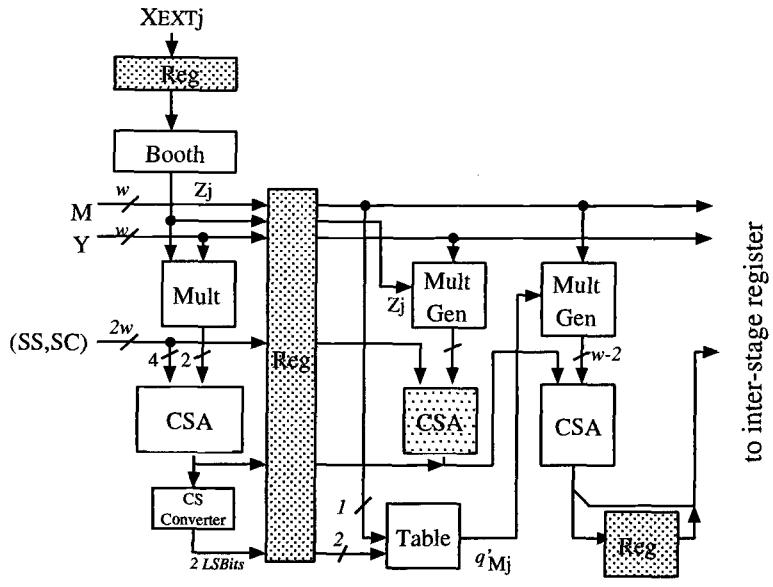

FIGURE 3.3: PE Organization.

The Processing Element (PE) is divided into two sections. The first section computes only the first 2 least-significant (LS) bits of each word of $S + xY$. One can observe that q'_{M_j} depends on 2 LSbits of the partial product from the previous computational cycle, $S^{(0)}_{1..0}$, the 2 LSbits of $Y^{(0)}$, and the encoded multiplier digit Z_j. The word size for S needs to be at least 4 bits, since the 2 LSbits of S for the next pipeline stage will be available well before the whole word $S^{(0)}$ is available, and so they can be used in determining q'_{M_j} for the next computation.

The second section completes the computation of the word bits of $S + Z_jY$ and the addition of full words of $S + q'_{M_j}M$.

The computation done on the LSbits by the first section is also done for all the other remaining operand words. So, while the leftmost adder works on the LS bits of a word of Z_jY, the topmost adder (after the input register) works on the other bits of the same word, therefore, there is one clock cycle difference between the two circuits.

Figure 3.4 shows the diagram for the Radix-4 PE (Montgomery Multiplication cell). As can be seen from the R4MM algorithm the multiplier is scanned two bits at a

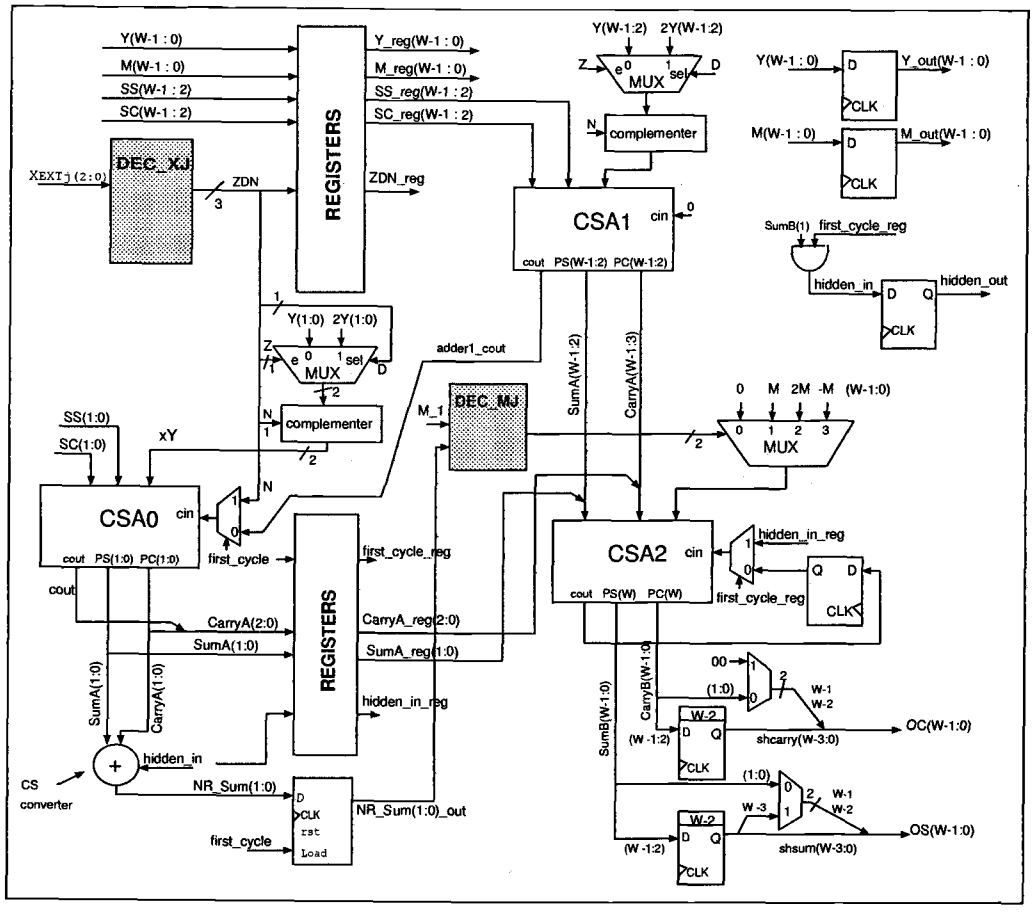

FIGURE 3.4: Radix-4 PE.

time. Booth encoding on these two bits and one bit from the previous scan is used to find the digit Z_j in module DEC_XJ according to Table 3.1. The negative multiples of Y are implemented by inverting the positive bit-vector Y and introducing a carry-in with a value of '1'. Since the encoded radix-4 digit (Z_j) have 5 possible values, we use a 2-input mux and complementer to generate the multiples of Y. The control signals for this module are the outputs from the DEC_XJ block and called ZDN (Zero, Double, and Negative respectively). Z is mapped to the mux enable (when it is one the output will be zero). D is mapped to the mux select. N is mapped to the complementer to generate one's complement of either Y or $2Y$. To generate the 2's complement of Y or $2Y$, a carry-in of '1' should be added during the first cycle of computation. This is done

$X_{EXT_j}(2{:}0)$	Z_j	cin	ZDN
000	0	0	100
001	1	0	000
010	1	0	000
011	2	0	010
100	$\bar{2}$	1	011
101	$\bar{1}$	1	001
110	$\bar{1}$	1	001
111	0	0	100

TABLE 3.1: Booth encoding for Z_j, the $\bar{}$ over the number means bit−complement

by inserting N as the carry input (cin) for CSA0 during the first cycle (least-significant word).

The addition step in the R4MM ($S + Z_jY$) is implemented by two Carry-Save Adders (CSA0 and CSA1). CSA0 is operating on the LSbits of words j of S and Z_jY, (so the logic determining q'_{M_j} can be done on them), while the first Carry-Save Adder (CSA1) is operating on the MSbits of word j-1. This arrangement requires that the carry-out propagation among words of the partial Sum A ($CarryA$ and $SumA$) be considered carefully. The carry-out of (CSA1), adder1_cout, is introduced immediately as carry-in for CSA0. The carry-out of the CSA0 is concatenated as MS bit of $CarryA(1:0)$. The output of CSA0 ($SumA(1:0)$, $CarryA(2:0)$) is stored in a register for one clock cycle before it will be concatenated with the output of CSA1 to generate one word of $SumA$ and $CarryA$.

In step 10 of the R4MM algorithm the partial product is right-shifted by two bits, these two bits must be made zeros before the shifting operation happens to avoid data loss. This is done by adding q'_{M_j} times the modulus M (steps 6 and 9). q'_{M_j} depends on the least significant two bits of the partial-product S (which is represented by two

vectors SS and SC), and the least significant two bits of the modulus M. There is one additional bit used in determination of q'_{M_j} (the hidden-bit).

Because Carry-Save representation (CS) is used for S, the Least Significant (LS) words of the two bit-vectors $SumB^{(0)}$, $CarryB^{(0)}$ (which are zeroed in step 6 in the algorithm) can be, for example: $SumB^{(0)} = \times..\times 11$ and $CarryB^{(0)} = \times..\times 01$, where $\times$ represents any value of the bit in this position. The LS two bits of S are equivalent to zeros when converted to a non-redundant form. However, data will be lost if these bits are shifted out in the CS form without taking into account the carry propagation ($11 + 01 = \underline{1}00$). The carry bit generated in this case is the "hidden-bit".

Knowing that the LS bit of M is always 1 (M is odd), q'_{M_j} will depend only on six bits: $SumA(1:0)$, $CarryA(1:0)$, *hidden-bit* and $M(1)$.

To detect the hidden-bit it is enough to test if either the second bit of $SumB$ or $CarryB$ has value '1', this can better explained as follows: $SumB(1:0) + CarryB(1:0) = b00$, b$\in \{0,1\}$, where b is the hidden-bit. The possible combinations are 00 + 00 or 01 + 11 or 11 + 01 or 10 + 10 $\Longrightarrow$ b = $SumB(1)$. So, the circuit for the hidden-bit is reduced to $SumB(1)$. This bit is stored into a flip-flop as *hidden_out*. The hidden-bit is used to compute q'_{M_j}(CS converter in Figure 3.4) and is also used as a carry input for CSA2 during the first word (LS word) computation. After the first cycle, CSA2 receives the carry-out of the previous addition as carry-in (controlled by a mux) in order to perform word serial addition.

The number of entries in the table DEC_MJ can be reduced by assimilating the carries for $SumA(1:0)$, $CarryA(1:0)$, and hidden-bit by a two-bit carry-propagate adder (Cs converter). The resulting two-bit vector (NR_Sum) is represented as:

$$NR\_Sum(1:0) = (SumA(1:0) + CarryA(1:0) + hiddenbit) \ mod\ 4$$

and reduces the Table for q'_{M_j} to 8 entries only as shown in Table 3.2. The output of DEC_MJ is the control signal for the 4-input multiplexer used to select the multiples of M.

	q'_{M_j}	
$NR\_Sum_{1..0}$	$M_1^{(0)}$	
	0	1
00	0	0
01	-1	1
10	2	2
11	1	-1

TABLE 3.2: Encoding for q_{M_j}

We know that to generate -M we need to obtain the bit complement of the bit-vector M and then add '1' as carry-in (2's complement sign change). But the carry-in for the second Carry-Save Adder (CSA2) cannot be used for this purpose. The mux attached to the cin input of CSA2 has the *hidden_bit_reg* and the delayed carry-out from the same adder as inputs, and it is controlled by the $first\_cycle\_reg$ signal. So, the problem is where to insert the carry-in of '1' to get two's complement of M. The solution of this problem comes from the fact that M is odd, this means the least significant bit (bit 0) is always one. This will cause the least significant bit of -M to be also one. By using this fact we can get negative M by performing bit complement on all the bits of M (except bit 0), and attaching a '1' in position 0 as shown below:

$$-M = \overline{M_{(W-1..1)}} \ \& \ M_0 = \overline{M_{(W-1..1)}} \ \& \ '1'$$

where $\bar{x}$ means one's complement of x, and & means concatenation.

The multiples of Y and M, like $2Y$, $2M$, require that these operands be left-shifted. Caused by the word-serial scanning of this algorithm, this shifting requires the MSbit from the previous words of Y and M to be used with the new coming words. If it is the first word ($first\_cycle = 1$), then a zero is shifted in to produce the needed multiple. Otherwise, the MSbit of the previous word is shifted in as the LSbit of the current word. This process is shown in Figure 3.5.

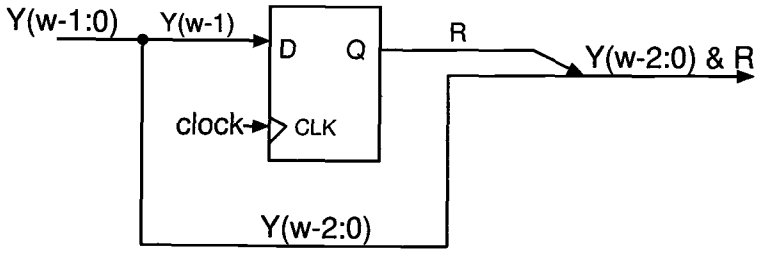

FIGURE 3.5: Word-serial bit shifter.

4. COMPARISON WITH DESIGNS FOR RADICES 2 AND 8.

This Chapter reviews kernel implementations of radix-2 and radix-8 scalable Montgomery multipliers done in the past [10, 6]. The algorithm and cell description are presented for each design. Comparison between these designs and radix-4 design are done when possible.

4.1. Radix-2 Implementation.

The simplest design of the Montgomery multiplier is the radix-2 design. Following the approach presented in [10], the main part of the multiplier is a kernel (pipeline of Processing Elements (PE's)). The algorithm and the PE implementation for radix 2 has been proposed in [10]. This proposed design is described briefly and compared to radix-4 design in the next two subsections.

4.1.1. Multiple-Word Radix-2 Montgomery Multiplication (M WR2MM) Algorithm.

Figure 4.1 shows the MWR2MM algorithm. For radix 2, the multiplier X is scanned one bit at a time. Therefore, we don't need encoding for the multiplier. Determining the quotient digit q_{M_j} is done by examining the LS bit of the partial product ($S_0^{(0)}$). The encoded multiplier digit q_{Y_j} and the quotient digit q_{M_j} are determined by a single bit, their values are either one or zero.

4.1.2. Radix-2 PE Description.

Figure 4.2 shows the block diagram for a radix-2 kernel Processing Element. The two Carry-Save Adders (CSA) are the main functional blocks. They perform steps (4, 6,

Step

1: $S := 0$

2: FOR $j := 0$ TO $N - 1$ STEP 1

3: $q_{Y_j} = x_j$

4: $(C_a, S^{(0)}) := S^{(0)} + (q_{Y_j} * Y)^{(0)}$

5: $q_{M_j} := S_0^{(0)}$

6: $(C_b, S^{(0)}) := S^{(0)} + (q_{M_j} * M)^{(0)}$

7: FOR $i := 1$ TO $NW - 1$

8: $(C_a, S^{(i)}) := C_a + S^{(i)} + (q_{Y_j} * Y)^{(i)}$

9: $(C_b, S^{(i)}) := C_b + S^{(i)} + (q_{M_j} * M)^{(i)}$

10: $S^{(i-1)} := (S_0^{(i)}, S_{BPW-1..1}^{(i-1)})$

END FOR;

11: $C_a := C_a \; or \; C_b$

12: $S^{(NW-1)} := (C_a, S_{BPW-1..1}^{(NW-1)})$

END FOR;

13: IF $S \geq M$ THEN $S := S - M$

END IF;

FIGURE 4.1: Multiple-Word Radix-2 Montgomery Multiplication (MWR2MM) Algorithm [10].

8, and 9) in the MWR2MM algorithm presented in the last subsection. The PE takes a single bit of the multiplier X (x_j) as input, and uses as a select line for a two-input mux which has as inputs one word of the multiplicand Y and Zero. The least significant bit of the sum output of the first Carry-Save Adder (CSA1) is stored in a registers for the next clock cycle to be used in determining the quotient q_M. The registers also stored the carry-out bit of CSA1. This bit is concatenated to the carry-vector to form one of the inputs to the CSA2. The *AddM* signal (one or zero) selects the output of the second mux (one word of the modulus M or Zero). The shifting unit is used to perform step 12

in the algorithm. The registers between two Processing Elements propagate words of the

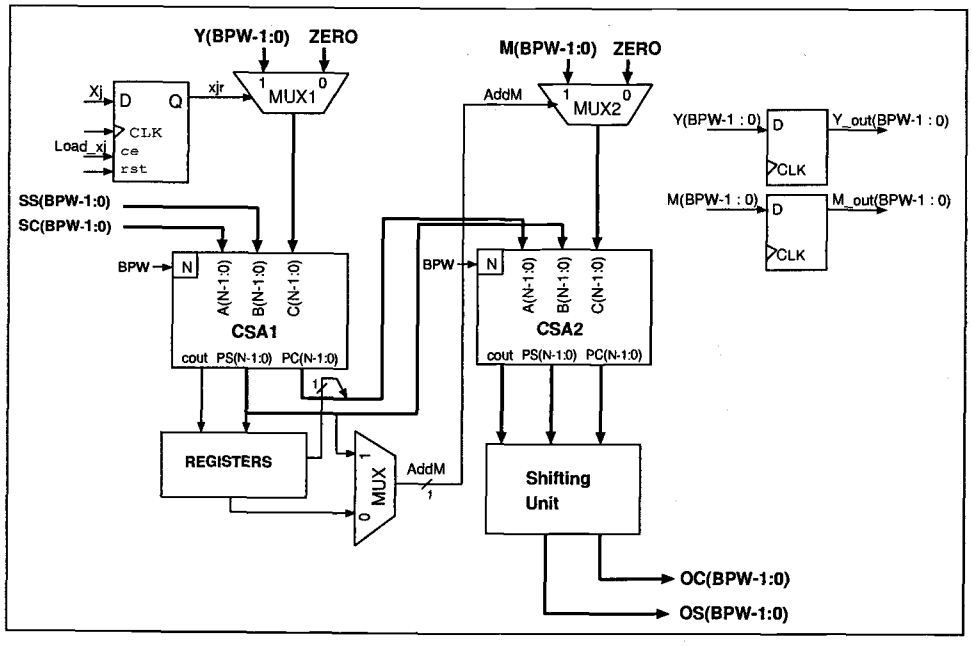

FIGURE 4.2: Radix-2 PE

partial product S, the multiplicand Y, and the modulus M from PE to PE. The control signals for the PE are delayed two clock cycles and then propagated to the next PE, as shown below in Figure 4.3. This operation is fully synchronous. Each cell propagates the control signals with strict timing and no decision is made in a cell to either speed up or delay the propagation of the control signals [6]. When comparing radix-4 and radix-2 designs, we notice that both designs are simple. The designs are similar in having the Carry-Save Adders and two-input muxes. In radix 4, the multiplier is scanned two bits at a time, and as a result the number of computation cycles is reduced to basically half of that needed for radix-2 computations. Since radix-4 algorithm uses two bits to determine the coefficient q_{Y_j} and the quotient q_{M_j}, little extra hardware is added to the design. The adder section is the same as the one in the radix-2 design, but the multiple generation and determination of the quotient digit is slightly more complex.

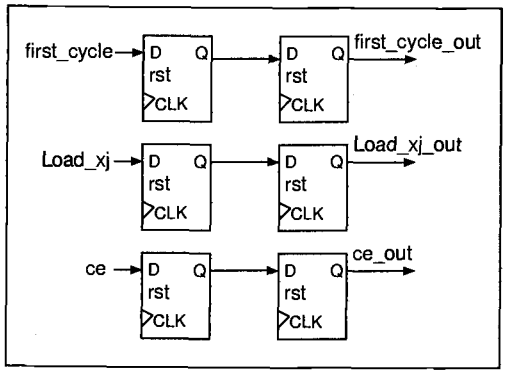

FIGURE 4.3: Control signals' propagation between two pipeline stages.

4.2. Radix-8 Implementation.

This Section presents the radix-8 Montgomery multiplier design. [6, 11], the main functional part of the multiplier is a kernel (pipeline of Processing Elements (PE's)). The next two subsections reviews the radix-8 algorithm and design which was proposed in [6].

4.2.1. Multiple-Word Radix-8 Montgomery Multiplication (MWR8MM) Algorithm.

Figure 4.4 shows the MWR8MM algorithm. For radix 8 the multiplier X is scanned three bits at a time. Booth encoding is used to recode the multiplier digits. The recoded multiplier digit is called qy_j. The possible values for qy_j are in the interval [-4,4]. The quotient digit q_{M_j} has values in the range [0,7]. The digit qy_j is divided into two parts $q1y_j$ and $q2y_j$ as shown in Table 4.1, and the same thing is applied to q_{M_j}. This is done to remove the generation of Y and M multiples from the critical path. As we know, the generation of some multiples like $3Y$ needs addition of other two multiples (in this case Y and $2Y$). This addition step increases the critical path delay. To solve this problem, the author in [6] generates the multiples of Y ($q1y_j$ and $q2y_j$) and M ($q1M_j$ and

$q2M_j$) obtained by shift operations only, and uses 4-2 Carry-Save Adders to perform the addition step. This explains why the Y and M multiple generators has two outputs.

Step

1: $S := 0$

$x_{-1} := 0$

$q_{Y_0} = q1_{Y_0} + q2_{Y_0} = Booth(x_{3..-1})$

$q_{M_0} := (q1_{Y_0} * Y^{(0)}_{2..0} + q2_{Y_0} * Y^{(0)}_{2..0}) * (2^3 - M^{(0)^{-1}}_{2..0})\ mod\ 8$

2: FOR $j := 0$ TO $N - 1$ STEP 3

3: $q_{Y_{j+3}} = q1_{Y_{j+3}} + q2_{Y_{j+3}} = Booth(x_{j+3+3..j+3-1})$

4: $(C_a, S^{(0)}) := S^{(0)} + q1_{Y_j} * Y^{(0)} + q2_{Y_j} * Y^{(0)}$

6: $(C_b, S^{(0)}) := S^{(0)} + q1_{M_j} * M^{(0)} + q2_{M_j} * M^{(0)}$

$q_{M_{j+3}} := q1_{M_{j+3}} + q2_{M_{j+3}} := S^{(0)}_{5..3} * (2^3 - M^{(0)^{-1}}_{2..0})\ mod\ 8$

7: FOR $i := 1$ TO $NW - 1$

8: $(C_a, S^{(i)}) := C_a + S^{(i)} + q1_{Y_j} * Y^{(i)} + q2_{Y_j} * Y^{(i)}$

9: $(C_b, S^{(i)}) := C_b + S^{(i)} + q1_{M_j} * M^{(i)} + q2_{M_j} * M^{(i)}$

10: $S^{(i-1)} := (S^{(i)}_{2..0}, S^{(i-1)}_{BPW-1..3})$

END FOR;

11: $C_a := C_a or C_b$

12: $S^{(NW-1)} :=$ sign ext $(C_a, S^{(NW-1)}_{BPW-1..3})$

END FOR;

13: IF $S \geq M$ THEN $S := S - M$

END IF;

FIGURE 4.4: Multiple-Word Radix-8 Montgomery Multiplication (MWR8MM) Algorithm [11].

This algorithm scans 3 bits of the multiplier X at a time, while radix-4 MM algorithm takes 2 bits of X at a time, and so the determination of the recoded multiplier digit qy_j is simpler in radix-4. The extra encoding applied to the radix-4 MM algorithm makes the determination of the quotient q_{M_j} also simpler in radix-4.

4.2.2. Radix-8 PE Description.

Figure 4.5 shows the block diagram for a radix-8 processing Element (Multiplication cell). The general architecture of the radix-8 multiplier is the same as the radix-4 design represented in Chapter 3. Four-to-two Carry-Save Adders (4-2 CSA) are used

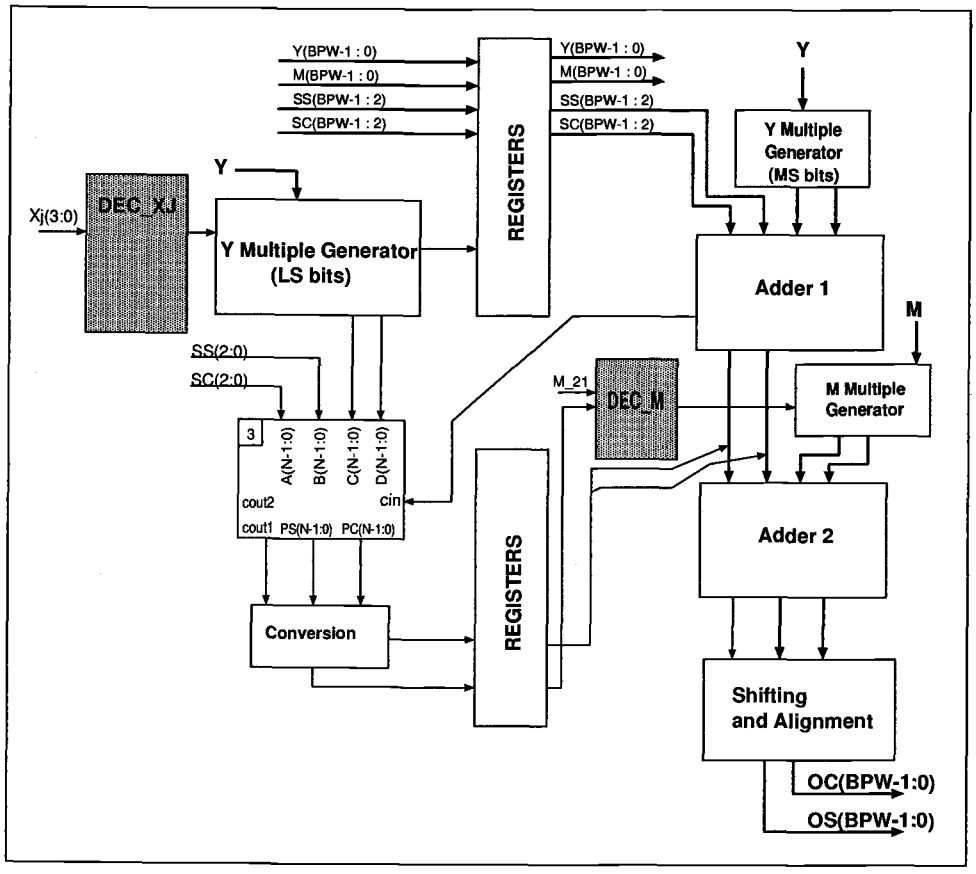

FIGURE 4.5: Radix-8 PE.

qy_j	$q1y_j$	$q2y_j$
-4	-4	0
-3	-4	1
-2	-4	2
-2	-2	0
-1	-1	0
0	0	0
1	0	1
2	0	2
2	0	2
3	-1	4
4	0	4

TABLE 4.1: Possible combinations for $qy_j = q1y_j + q2y_j$

instead of simple CSA. Portion of adder1 is moved to operate on the least significant three bits of the partial product S. This re-timing technique speeds up the computation necessary to compute the quotient digit q_{M_j}. The output of this adder goes to the conversion block which generates a three-bit vector called $AddM$.

The DEC_XJ block takes as input four bits of the multiplier X (one bit comes from the previous computation cycle). A radix-8 version of the Booth encoding uses these bits to compute the recoded multiplier digit qy_j (which is divided into two parts as mentioned before). The DEC_MJ block takes $AddM(2:0)$ and two bits of the modulus M to generate the quotient digit q_{M_j} as two components. Table 4.2 shows the possible combinations of q_{M_j}. The component values of q_{M_j} and qy_j are powers of two, and can be easily generated in hardware.

The registers propagate words of the partial product S, the multiplicand Y and the modulus M from PE to PE. The shifting and alignment block used to generate the

q_{M_j}	$q1_{M_j}$	$q2_{M_j}$
0	0	0
1	1	0
2	2	0
3	-1	4
3	1	2
4	0	4
5	1	4
6	2	4
7	-1	8

TABLE 4.2: Possible combinations for $q_{M_j} = q1_{M_j} + q2_{M_j}$

output by suitable wiring as in the previous two designs, the control signals for the PE are delayed two clock cycles and then propagated to the next PE. More details about radix-8 design are presented in [6, 11].

When comparing radix-4 and radix-8 designs, radix-8 design uses one extra bit which basically doubles the complexity of the logic used in the DEC_XJ and DEC_MJ blocks. On the other hand, and by applying encoding to the multiplier and the modulus in radix-4 design, the DEC_XJ and DEC_MJ blocks became simpler, and so, the critical path delay and design area improved significantly. Also, radix 8 uses four-to-two Carry-Save Adders (4-2 CSA) which has large area and delay than simple Carry-Save Adders (CSA) used in the radix-4 design.

5. EXPERIMENTAL RESULTS AND ANALYSIS.

5.1. Synthesis and Simulation Environment.

The experimental data presented in this chapter was generated by Mentor Graphics package. The target technology was set to $AMI05\_fast\ auto$ (0.5 μm $CMOS$ with hierarchy preserved) provided in the ASIC Design Kit (ADK) from the same company. A data-book for this technology is available at [22]. The experimental data for radix-2 and radix-8 kernel implementations were taken from [6], where the $AMI05\_slow$ flattened (no-hierarchy) technology was used. The flattened designs were laid-out using ICStation. The radix-4 design presented in this thesis was described in VHDL. and then simulated in ModelSim for functional correctness. It was synthesized using Leonardo synthesis tool for the mentioned technology. It has to be noted that the ADK has been developed for educational purposes and therefore cannot be fully compared to technologies used for commercial ASICs, however, it provides a consistent environment for comparison between the designs, and a reasonable approximation of the system performance when using commercial ASIC technology.

5.2. Radix-4 Kernel.

5.2.1. Area Estimation for Radix-4 Kernel.

The area of the kernel depends on the two design parameters: number of stages in the pipeline (NS), and the word size (w) of the operands (Y, M) and the result (S). The Processing Element (PE) for the radix-4 algorithm has an inter-stage register incorporated in it.

An inter-stage register holds one word of M, one word of Y, two words of S, and one extra single bit (the hidden bit). Flip-flops (DFFs) can be used for M and Y while S and the hidden bit requires flip-flops with asynchronous reset (DFFRs). Thus, the

area for an inter-stage register is:

$$A_{STAGE\_REG} = 2 * W * A_{DFF} + 2 * W * A_{DFFR} + A_{DFFR}.$$

After considering several experimental results we found that the area of the multiplication cell (or PE) for radix-4 is expressed by:

$$\begin{aligned} A_{cell_{R4}} = A_{STAGE\_REG} + 2 * A_{CSA(W)} + A_{YMUX_{xY}}(W) + A_{MUX_M}(W) + \\ +2 * W * A_{DFF} + 2 * (W) * A_{DFFR} + 8 * A_{DFFR} + 7 * A_{REGN} \\ +5 * A_{DFF} + A_{DEC-x_j} + A_{DEC-M} + A_{2-bitadd} + A_{OA} \end{aligned}$$

The following area estimates are given by technology specifications (obtained by a Leonardo synthesis tool), shown as a number of 2-input NOR gates:

- $A_{FA} = 6$;
- $A_{MUX} = 1.4$;
- $A_{DFF} = 4.79$;
- $A_{DFFR} = 5.92$;
- $A_{REG} = 7.97$;
- $A_{OA} = 1.24$;
- $A_{DEC-x_j} = 8$;
- $A_{DEC-M} = 7$;
- $A_{CSA_{(W)}} = W * A_{FA}$;
- $A_{YMUX_{xY}} = A_{MUX\&Complementer} = 4.87$;
- $A_{2-bitadd} = 12$.

Where $YMUX_{xY}$ is the mux and complementer used to generate multiples of Y.

The area of the two-level four-input multiplexer that is used to select multiples of M can be represented as:

$$A_{MUX_M}(W) = W * 3 * A_{MUX}.$$

So, as a final result, the area of radix-4 multiplication cell (PE) is:

$$A_{cell_{R4}} = 62.86 * W + 146,$$

and then, the total area of the kernel is:

$$A_{kernel_{R4}} = 62.86 * NS * W + 146 * NS - 4.875 * W - 13. \quad (5.1)$$

Table 5.1 is constructed using Eq. 5.1. The numbers in this Table and the numbers obtained by synthesizing the design are very close.

5.2.2. Time Estimation for Radix-4 Kernel.

The total computational time for the kernel is a product of the number of clock cycles it takes and the clock period. Table 5.2 shows the critical path delay as a function of the number of stages in the pipeline (NS), as well as the word size (w) of the operands. The points in the Table are tested configurations. As can be seen from the Table, the critical path delay in some cases remains constant even if the number of stages is increased. In radix-2 and radix-8 designs the critical path delay is increased by increasing NS.

The total computational time is also affected by the number of clock cycles it takes. Two cases should considered when analyzing the results, (i) when $e \leq 2 * NS$, and (ii) when $e > 2 * NS$, where $e = \lceil \frac{N}{w} \rceil$ is the number of words in the N-bit operands with chosen word size of w bits. Analysis and optimal design points are presented in the next subsection.

A word of Y, M, and S propagates through the pipeline for $(2 * NS + 1)$ clock cycles. The speed of scanning the bits of X for radix-4 is two bits per stage, or $\lceil \frac{N}{2*NS} \rceil$.

NS	Word Size (w)				
	8	16	32	64	128
1	598	1060	1989	3844	7555
2	1246	2212	4146	8013	15747
3	1895	3364	6304	12182	23939
4	2544	4516	8461	16351	32131
5	3193	5667	10617	20520	40323
6	3842	6819	12776	24689	48516
7	4491	7971	14934	28858	56708
8	5139	9123	17091	33027	64900
9	5788	10274	19248	37196	73092
10	6437	11426	21406	41365	81284
11	7086	12578	23563	45534	
12	7735	13730	25721	49704	
13	8384	14881	27879	53873	
14	9033	16033	30036	58042	
15	9681	17185	32194	62211	
16	10331	18337	34351	66380	
20	12926	22944	42981	83056	
25	16170	28703	53769		
30	19415	34461	64557		
35	22659	40220	75344		

TABLE 5.1: Area in number of NOR gates for radix-4 kernel.

Based on these observations, Equation 5.2 represents the total number of clock cycles needed for radix-4 Montgomery multiplication.

$$T_{CLKs} = \begin{cases} \lceil \frac{N}{2*NS} \rceil * (2*NS+1) + \lceil \frac{N}{W} \rceil + 1 & ,if \lceil \frac{N}{W} \rceil \leq 2*NS \\ \lceil \frac{N}{2*NS} \rceil * (\lceil \frac{N}{W} \rceil + 1) + 2*NS & ,if \lceil \frac{N}{W} \rceil > 2*NS \end{cases} \tag{5.2}$$

	Bits Per Word				
NS	8	16	32	64	128
1	5.52	5.81	6.25	7.3	6.79
2	5.62	6.13	6.28	7.34	6.84
3	5.62	6.13	6.28	7.34	6.84
4	5.62	6.13	6.28	7.34	6.84
5	5.62	6.13	6.28	7.34	6.84
6	5.62	6.34	6.28	7.34	6.84
7	5.62	6.34	6.28	7.34	6.84
8	5.62	6.34	6.28	7.34	6.84
9	5.62	6.34	6.28	7.34	6.84
10	5.62	6.34	6.28	7.34	6.84
11	5.62	6.34	6.28		
12	5.62	6.34	6.28		
13	6.21	6.34	6.28		
14	6.21	6.34	6.28		
15	6.21	6.34	6.28		
20	6.21	6.34	6.28		
25	6.21	6.34	6.28		
30	6.21	6.34			
35	6.21	6.34			

TABLE 5.2: Critical path delay for radix-4 kernel.

The total computational time is obtained by multiplying T_{CLKs} by the corresponding critical path delay (clock period) shown in Table 5.2, which was obtained from synthesis tools.

Figures 5.1 and 5.2 show the total time graphs for radix-4 kernel with 256-bit and 1024-bit operands respectively, for several possible configurations (w, NS).

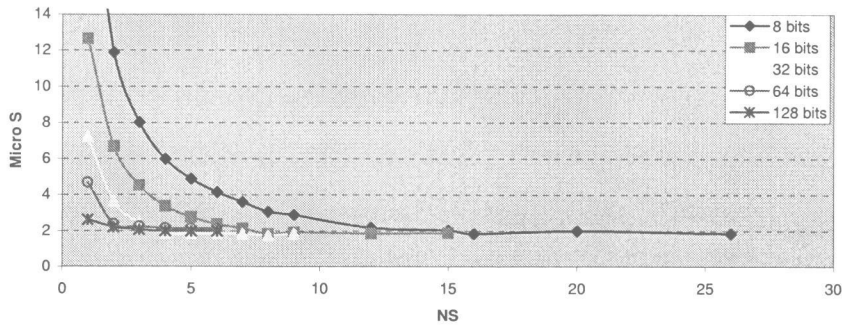

FIGURE 5.1: Total computational time for radix-4 kernel, 256 operands.

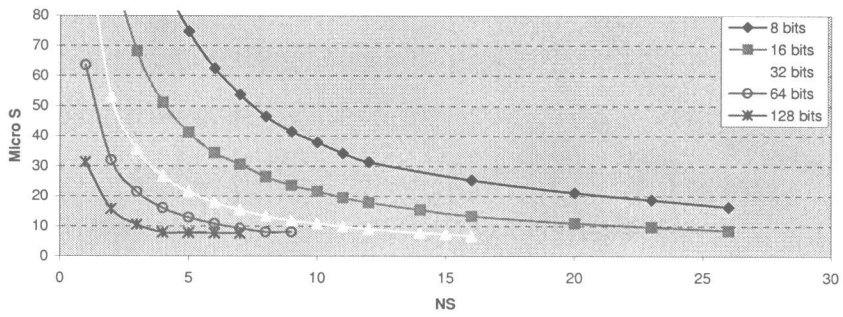

FIGURE 5.2: Total computational time for radix-4 kernel, 1024 operands.

5.2.3. Optimal Design Points for Radix-4 Kernel.

As mentioned before, the two cases that should be considered in analysis for radix-4 design are when $e \leq 2 * NS$ and $e > 2 * NS$.

By doing some calculation we observe that by crossing the boundaries $e = 2 * NS$ and $e > 2 * NS$, the first minimum computational time happens. Also, Figure 5.1 shows that the computational time goes through a series of minimal and maximal values by increasing the number of pipeline stages. Operands with lower precision (256 bits) will require a smaller number of stages in the pipeline than operands with higher precision (1024 bits) in order to execute the operation in minimal time. The optimal design point should have computational time for 256-bit precision close to its absolute minimal value and at the same time to have small computational time for 1024-bit precision.

From the experimental data, the fastest design is achieved when the word size (w) is 8 bits. For 256-bit operand with $w = 8$, the first optimal design point is when $NS = 16$, with area of 10,330 NOR gates. Each additional stage adds 649 gate to the area compared with 1,005 gates in radix-8 [11]. Table 5.3 compares several design points for the radix-4 kernel with word size of 8 bits. The Table presents the design area and the ratio of the computational time related to the point $NS = 16$. It can be seen that the

NS	15	16	18	22	24	26
Area, gates	10330	11627	12925	14223	14872	16819
$\frac{t_{NS=16}}{t}$ for 256-bit	1	.9	0.93	0.98	0.94	0.99
$\frac{t_{NS=16}}{t}$ for 1024-bit	1	1.1	1.23	1.33	1.38	1.58

TABLE 5.3: Optimal design points for radix-4 kernel, 8-bit word size, 256-bit and 1024-bit operand precision.

design point with $NS = 26$ is very suitable since the computational time for 256-bit precision is almost the same as its minimal value. At the same time the computational time for 1024-bit precision is improved by 56% as compared to the point with $NS = 16$.

5.3. Comparison With Radix-2 and Radix-8 Kernel Experimental Data.

This section compares the experimental data obtained for radix-4 with the experimental data for radices 2 and 8, which is taken from [6]. In the next section, radix 4 is compared with experimental data resulting from re-synthesizing radix-2 and radix-8 designs.

The area of radix-2 kernel is given by:

$$A_{kernel_{R2}} = 59.65 * NS * W + 51.4 * NS - 31 * W - 35.5.$$

And for radix-8 the kernel area is:

$$A_{kernel_{R8}} = 92 * W * NS + 269 * NS - 9.42 * W - 35.5.$$

The area of the kernel cell of the three designs depends on the word size (w), and the number of stages in the pipeline (NS). At $w = 8$, the radix-2 kernel cell area is 529 gates, and radix-4 cell is 649 gates. The radix-8 cell has area of about 1,005 gates at the same word size. The area of radix-2 and 4 are close to each other. We notice that, for a given configuration (w, NS), the area increases with the radix, i.e., $A_2 < A_4 < A_8$.

The critical path delays for radix 2 and 8 are provided in [6]. Radix-4 design has big reduction in the critical path delay compared to radix-2 and 8 designs.

Table 5.4 shows the total computational time in μsec for the three radices at the same area (7,800 gates). The improvement of the radix-4 design over the radices 2 and 8 designs is also shown in the Table. Other points on the figures have more gain and others have less. We conclude the radix-4 design has a significant gain in reducing the total computational time over the radices 2 and 8 designs presented in [6].

	r = 2	r=4	r=8	$\lvert\frac{t_{r=4}-t_{r=2}}{t_{r=2}}\rvert$	$\lvert\frac{t_{r=4}-t_{r=8}}{t_{r=8}}\rvert$
Total Time	5.1	2.2	4.7	56%	45%

TABLE 5.4: Comparison between the total computation time (μsec) for the three designs taken at area of 7,800 gates with 256-bit operand precision

5.4. Re-synthesizing Radix-2 and Radix-8 Designs

The results presented in [6] for the radix-2 and 8 designs are obtained by flattening the designs in ICStation. This might add some wire delays to the critical path, and makes the gain obtained from radix-4 design over the two designs very high.

To make the comparison more fair, we re-synthesized the radix-2 and radix-8 designs presented in [6], using the same $AMI05\_fast\ auto$ technology used to synthesize radix-4 design. The total computational time for the radix-2 and radix-8 designs using the above technology are presented in Figures 5.3, 5.4 Also, Table 5.5 shows the total

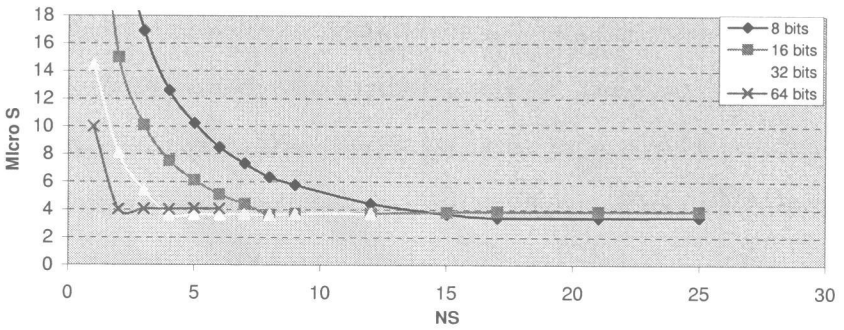

FIGURE 5.3: Total computational time for radix-2 kernel, 256 operands.

computational time in μsec for the three radices at the same area (7,800 gates). The improvement of the radix-4 design over the re-synthesized radix-2 and 8 designs as can be seen from the Table, is 48% and 34% respectively. The Figures and the Table show

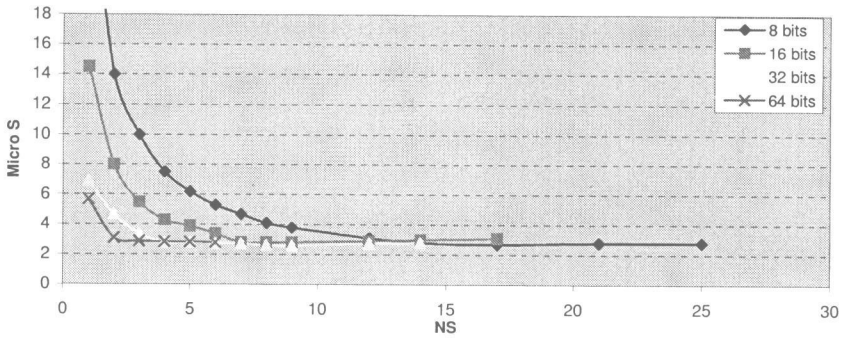

FIGURE 5.4: Total computational time for radix-8 kernel, 256 operands.

	r = 2	r=4	r=8	$\lvert\frac{t_{r=4}-t_{r=2}}{t_{r=2}}\rvert$	$\lvert\frac{t_{r=4}-t_{r=8}}{t_{r=8}}\rvert$
Total Time	4.2	2.2	3.31	48%	34%

TABLE 5.5: Comparison between the total computation time (μsec) for the three designs (synthesized using the same technology) taken at area of 7,800 gates with 256-bit operand precision

that radix-4 design still have less total computational time than the other two designs. This proves that radix 4 has the best performance among the three radices.

6. CONCLUSION AND FUTURE WORK.

6.1. Area-Time Comparison of The Three Kernel Implementations

In this Section the three kernel implementations are compared in terms of the total computational time as a function of the design area (area $\times$ time).

Figure 6.1 shows what overall time can be achieved for Montgomery multiplication of 256-bit operands as a function of the area. It can be observed that all three curves reach a wide region where the computational time stays close to its minimal value.

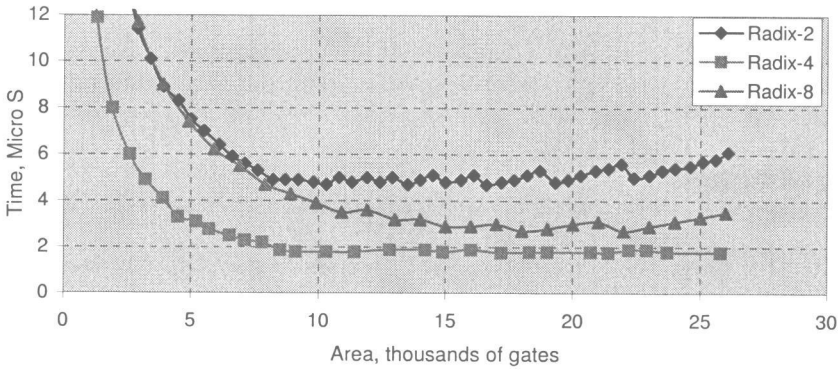

FIGURE 6.1: Total computational time compared to area usage, for 256-bit operands.

Three regions can be defined on the Figure: the first region is for design area below 8,000 gates; the second region is for area between 8,000 and 23,000 gates; and the third region is for area above 23,000 gates. It can be observed that for area below 8,000 the radix-2 design and the radix-8 design provide approximately the same computational time, and radix-4 design provides less time than the two designs. For area above 8,000 gates, the radix-8 design performs better than radix-2 design, but as in the first region, radix-4 has the lowest time. For a larger area, above 23,000 gates, the computational

time slightly begins to increase for both radix-2 and radix-8 designs, while the time stays around its minimal value in radix-4 design.

For different operand precision the computational time will reach its minimal value for different design area, as it can be seen from the data in Chapter 5. For 256-bit operands it is not worth to use more than 8,000 gates. However, when the precision increases, a large area would provide significantly more performance. So, the small precision case is the worst case scenario for the comparison among designs.

One can conclude that the proposed design and implementation of the modified radix-4 Montgomery multiplication algorithm proposed in this work has advantage over the radix-2 and radix-8 designs. The radix-4 design has less total computational time than the other two designs with reasonable area, which makes it the best solution for hardware implementation.

6.2. Why Radix-4 Was not Used Before?

Only radix-2 and radix-8 designs were discussed and designed before this work. In [6] several reasons for not using radix-4 were presented. One reason is related to the generation of the multiples of Y. The range of values for Y multiples (q_Y) is [-2,2]. It is indicated in [6]that one choice to implement the step $S + (q_Y * Y)^{(*)}$ is to use a two-level multiplexer tree and a Four-to-Two adder. A multiplexer has a gate delay approximately equivalent to a XOR-gate delay, and the 4-2 adder has approximately 3*XOR-gate delay. So, $S + (q_Y * Y)^{(*)}$ is implemented with a 5*XOR-gate delay, and this is equivalent to what was used in radix-8 design.

To solve this problem, we used a 2-input mux and a bit complementer. This module implements the multiples of Y with delay of approximately 2*XOR-gate delay. For the addition step, we used Carry-Save Adder (CSA) instead of using 4-2 CSA adder.

Another reason mentioned in [6] that makes radix-4 design less attractive is related to multiples of M. The range for the quotient digit (q_M) used to obtain the multiples of M in radix-4 is [0..3]. The value 3 for q_M would need to be implemented as two parts

(as done in radix-8 design). Therefore, the second adder in a radix-4 PE would need be a four-to-two adder. But in our design we used an encoding strategy to replace the value of $q_M = 3$ by $q_M = -1$. This encoding makes the generation of multiples easier by avoiding addition and using only a 4-input mux. As a consequence, a CSA adder is used instead of a 4-2 adder.

From the above discussion, we can say that the radix-4 critical path delay is less than radix-8 delay by approximately 3*XOR-gate delay. This is due to using CSA's instead of 4-2 adders and using new module in generating the multiples of Y. As a result, the radix-4 design is more attractive design than both radix-2 and 8 designs.

6.3. Future work

The complexity of Montgomery multiplier makes the testing process a big challenge. A methodology for developing testing modules is introduced in [23]. Including a self-testing block in the multiplier's system will be beneficial and will reduce the time and effort for testing. A self-testing block will perform Montgomery multiplication of hardwired numbers and compare the result with predefined values. A flag bit can be used to indicate an error.

Power dissipation study of the design is also needed in the context of power differential attack. This type of attack on a cryptographic system tries to deduce parameters of the system by observing system's power dissipation. This study would be applicable to show the adequacy of this design approach to hw-power devices, such as portable computers.

More study need to be done to see the effect of applying re-timing technique to radix-2 design, and how the re-timing will affect the performance of the design.

Some investigations need to be done to show how the radix-4 design presented in this text can be extended to cover the unified architecture as presented in [8].

The integration of multiplication and exponentiation can be included as part of a hardware co-processor.

BIBLIOGRAPHY

1. L. Adleman, R. L. Rivest, and A. Shamir, "A method for obtaining digital signature and public-key cryptosystems," *Comm. of the ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

2. M. E. Hellman and W. Diffie, "New directions on cryptography," *IEEE transactions on Information Theory*, vol. 22, pp. 644–654, November 1976.

3. N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, January 1987.

4. National Institute for Standards and Technology, "Digital signature standard (dss)," Tech. Rep. 168-2, FIPS PUB, January 2000.

5. D. M'Raihi and D. Naccache, "Cryptographic smart cards," *IEEE Micro*, vol. 16, no. 3, pp. 14–23, June 1996.

6. G. Todorov, "ASIC design, implementation and analysis of a scalable high-radix Montgomery multiplier," Master thesis, Oregon State University, USA, December 2000.

7. P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, April 1985.

8. E. Savas, A. F. Tenca, and Ç. K. Koç, "A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$," in *Cryptographic Hardware and Embedded Systems — CHES 2000*, Ç. K. Koç and C. Paar, Eds. 2000, Lecture Notes in Computer Science, No. 1717, pp. 281–296, Springer, Berlin, Germany.

9. E. F. Brickel, "A fast modular multiplication algorithm with application to two key cryptography," in *Advances in Cryptography - CRYPTO '82*. 1983, pp. 51–60, Plenum, New York.

10. A. F. Tenca and C. K. Koc, "A word-based algorithm and scalable architecture for montgomery multiplication," in *Cryptographic Hardware and Embedded Systems — CHES 1999*, Ç. K. Koç and C. Paar, Eds. 1999, Lecture Notes in Computer Science, No. 1717, pp. 94–108, Springer, Berlin, Germany.

11. A. F. Tenca, G. Todorov, and Ç. K. Koç, "High-radix design of a scalable modular multiplier," in *Cryptographic Hardware and Embedded Systems — CHES 2001*, Ç. K. Koç and C. Paar, Eds. 2001, Lecture Notes in Computer Science, No. 1717, pp. 189–206, Springer, Berlin, Germany.

12. C. D. Walter, "Space/time trade-offs for higher radix modular multiplication using repeated addition," *IEEE Transactions on computing*, vol. 46, no. 2, pp. 139–141, February 1997.

13. A. D. Booth, "A signed binary multiplication technique," *Q. J. Mech. Appl. Math.*, vol. 4, no. 2, pp. 236–240, 1951.

14. A. Royo, J. Moran, and J. C. Lopez, "Design and implementation of a coprocessor for cryptography applications," in *European Design and Test Conference*, Paris, France, March 17-20 1997, pp. 213–217.

15. P. Kornerup, "High-radix modular multiplication for cryptosystems," in *IEEE 11th Symposium on Computer Arithmetic.* 1993, pp. 277–283, IEEE Computer Society Press, Los Alamitos, CA.

16. H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Proceedings, 12th Symposium on Computer Arithmetic*, S. Knowles and W. H. McAllister, Eds., Bath, England, July 19–21 1995, pp. 193–199, IEEE Computer Society Press, Los Alamitos, CA.

17. R. R. Taylor and S. C. Goldstein, "A high-performance flexible architecture for cryptography," in *Cryptographic Hardware and Embedded Systems*, C. Paar Ç K. Koç, Ed. 1999, number 1717 in Lecture Notes in Computer Science, pp. 231–245, Springer, Berlin, Germany.

18. A. Vandemeulebroecke and et al, "A new carry-free decision algorithm and its application to a single-chip 1024-b rsa processor," *IEEE Journal of Solid-state Circuits*, vol. 25, no. 3, pp. 748–755, June 1990.

19. T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *Proceedings, 14th Symposium on Computer Arithmetic*, I. Koren and P. Kornerup, Eds., Bath, England, April 14–16 1999, pp. 70–77, IEEE Computer Society Press, Los Alamitos, CA.

20. A. Bernal and A. Guyot, "Design of a modular multiplier based on Montgomery's algorithm," in *13th Conference on Design of Circuits and Integrated Systems*, Madrid, Spain, November 17–20 1998, pp. 680–685.

21. J. C. Bajard, L. S. Didier, and P. Kornerup, "An RNS Montgomery modular multiplication algorithm," *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 766–776, July 1998.

22. ASIC Design Kit. Mentor Graphics Co, "http://www.mentor.com /partners/hep/AsicDesignKit/dsheet/ami05databook.html," .

23. C.D. Walter, "Moduli for testing implementations of the rsa cryptosystem," in *in IEEE 14th Symposium on Computer Arithmetic.* 1999, pp. 78–85, IEEE Computer Society Press, Los Alamitos, CA.