An Efficient and Scalable Radix-4 Modular Multiplier Design Using Recoding Techniques

Alexandre F. Tenca, Lo’ai A. Tawalbeh
Department of Electrical & Computer Engineering
Oregon State University, Corvallis, Oregon 97331, USA
tenca,tawalbeh@ece.orst.edu

Abstract—This paper presents the algorithm and architecture of a scalable radix-4 Montgomery Multiplier. The straightforward implementation of a radix-4 design based on the techniques already published results in a poor solution. In this paper we present an algorithm and architecture for the scalable radix-4 multiplier that makes use of two types of digit recoding in order to generate an efficient solution. The word-by-word algorithm used in the multiplier gives to the designer the freedom to select the level of parallelism according to the available area. Experimental results are shown to demonstrate that the proposed radix-4 Montgomery Multiplier design has better area/performance tradeoff than previous radix-2 and 8 scalable designs.

I. Introduction

Modular addition, multiplication, and inversion are extremely important in cryptography. The most important arithmetic operation is modular multiplication and it is used in many cryptographic algorithms such as RSA [1] and Diffie-Helman key exchange [2]. The Montgomery Modular Multiplication algorithm (MM) has enabled considerable advantage in speeding up cryptographic algorithms. When implemented in hardware, many aspects need to be considered, such as area and time tradeoffs, use of higher radix, and use of redundant number representation [3].

Previously published scalable modular multiplier designs use radix-2 Montgomery multiplication algorithm [4], [5] or radix 8 [6]. Basically, the proposed designs are able to work on words of the operands. A computation kernel may be used as many times as needed to perform a variable-precision computation. Besides, the kernel complexity is adjustable to the available area. If more area is used, more performance can be extracted from the kernel.

The radix-4 design generated by the use of the same techniques applied to radix 8 turned out to be very poor in terms of area utilization and performance. However, radix 4 is usually the best design point for several arithmetic operations since it provides the best gain in performance with very small increase in area. In this paper we show that a more elaborate design using two types of digit recoding makes the radix-4 design the best solution for the implementation of this scalable multiplier.

The following Section presents some basic concepts of Montgomery multiplication and most of the mathematical notation used in this paper. The proposed Radix-4 Montgomery Multiplication algorithm is presented in Section III. The overall organization and design of the multiplier is presented and explained in detail in Section IV. Section V shows the building boxes for the computational kernel. Comparison and analysis of the experimental results for radix 2, 4, and 8 are presented in Section VI. Section VII concludes this paper.

II. Montgomery Multiplication

The Montgomery multiplication algorithm generates the product of two n-bit integers $X$ (multiplier) and $Y$ (multiplicand) in modulo $M$ as:

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

where $r = 2^n$. $M$ and $r$ must satisfy the condition $\gcd(r,M) = 1$, which is easily accomplished when $M$ is odd. It is usually the case that $2^{n-1} < M < 2^n$. The Montgomery image of an integer $a$ can be obtained by multiplying it by the pre-computed 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 $b$.

III. Proposed Radix-4 MM Algorithm

The proposed multiple-word Radix-4 Montgomery Multiplication (R4MM) algorithm is shown in Fig. 1. It is an extension of the Multiple-Word High-Radix (R2k) Montgomery Multiplication algorithm (MWR2kMM) presented and proved to be correct in [6]. The R4MM uses an extra encoding step for the multiples of the modulus $M$, which wasn’t used in MWR2kMM.

In the figure, a single radix-4 digit at position $j$ is represented as $X_j$, $q_M$, is the quotient digit that determines a multiple of $M$ to be added to the partial product $S$. The vectors $Y$, $M$, and $S$ are divided in $w$-bit words. A total of $e = \lceil \frac{w+1}{2} \rceil$ words are obtained, and represented as $(Y^{(e-1)}, ..., Y^{(1)}, Y^{(0)})$ for the $Y$ operand. Bits $k = 1$ to $0$ of the $i^{th}$ word of $S$ are denoted as $S^{(i)}_{k = 1...0}$. $C_a$ and $C_b$ are carry bits.
Step 1: \( S := 0 \) 
\( x_{-1} := 0 \)

2: FOR \( j := 0 \) TO \( n - 1 \) STEP 2
3: \( Z_i = \text{Encoding1}(x_{j+1,j-1}) \)
4: \( (Ca, S^{(0)} := S^{(0)} + (Z_i \ast Y)^{(0)} \)
5: \( qM := S^{(0)} \ast (4 - M_{1,0}^{(0)}) \) mod 4 
5a: \( q'_{M} := \text{Encoding2}(qM) \)
6: \( (Cb, S^{(0)} := S^{(0)} + (q'_{M} \ast M)^{(0)} \)
7: FOR \( i := 1 \) TO \( e - 1 \)
8: \( (Ca, S^{(i)} := Ca + (Z_j \ast Y)^{(i)} \)
9: \( (Cb, S^{(i)} := Cb + (S^{(i)} + (q'_{M} \ast M)^{(1)} \)
10: \( S^{(i-1)} := (S^{(i)}_{1,0}, S^{(i-1)}_{w-1,2}) \)
END FOR;
11: \( Ca := C_a \) or \( C_b \)
12: \( S^{(e-1)} := \text{signext}(Ca, S^{(e-1)}_{w-1,2}) \)
END FOR;

Fig. 1. Multiple-word Radix-4 Montgomery Multiplication (R4MM) Algorithm.

There are two types of encoding applied in the R4MM. The first one (Encoding1) is Booth encoding [7] applied to the multiplier \( X \). This recoding scheme translates conventional digits in radix 4 \( \{0,1,2,3\} \) into the digit set \( \{-2, -1, 0, 1, 2\} \) [6]. The recoded digit \( Z_i \) is obtained from the radix-4 digit \( X_i = (x_{2i+1, x_{2i-1}}) \) of the multiplier as:

\[ Z_i = \text{Encoding1}(X_i, x_{2i-1}) = -2x_{2i+1} + x_{2i} + x_{2i-1} \]

where \( i = 0, 1, 2, \ldots, n - 2, n - 1 \).

The combination of the radix-4 digit at position \( i \) \((X_i)\) and the most significant bit of radix-4 digit at position \( i-1 \) is represented as \( X_i^{\text{EXT}} = (X_i, x_{2i-1-1}) \). 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 two least-significant bits of \( S \) all zeros, a multiple of \( M \), namely \( qM, M \), is added to the partial product. Digit \( qM \) is computed from the two least-significant bits of the partial product \( S \) generated in step 4. In the next subsection we discuss more about the possible values of \( qM \) and how recoding was used to simplify the design.

The most significant (MS) word of \( S \) is generated in step 11 with a corresponding sign extension operation \((\text{signext})\) performed in step 12, since the partial-product \( S \) can have negative values. The final modular reduction was intentionally not included.

It is shown in [6] that \( qM \), as computed in step 5, satisfies the relation \( qM \ast M = -S \) mod 4, which can be rewritten as:

\[ S_{1,0} + qM \ast 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 \( Z_i \). However, it is still necessary to show that Encoding2 (step 5a) generates an equivalent result. The algorithm is correct if \( q'_{M} \equiv q_M \) mod 4.

A. Encoding of \( qM \)

The values for the quotient digit \( qM \) are in the set \( \{0, 1, 2, 3\} \). Applying an encoding function \((\text{Encoding2})\) we transform the quotient digit \( qM \) to the digit set \( \{-1, 0, 1, 2\} \). The recoding scheme consists in replacing \( qM = 3 \) by the recoded value \( q'_{M} = -1 \). It makes the generation of multiples of \( M \) less complex. Based on the fact that \(-1 \equiv 3 \mod 4 \) it is possible to conclude that

\[ q'_{M} \equiv q_M \) mod 4 \rightarrow q'_{M} \ast M \equiv q_M \ast M \) mod 4.

and therefore the application of \(\text{Encoding2} \) generates a result that is congruent to the correct result, modulo \( M \). In fact steps 5 and 5a can be executed in a single step. Two steps were shown for clarity only.

B. Boundaries for the Partial Product \( S \)

The performance of modular exponentiation (used in cryptographic algorithms) is improved if the result of one multiplication is used as the input to another one without modular reduction [8]. In this section we will verify that the range of the output values allows them to be used as inputs in another multiplication.

The radix-4 Montgomery multiplication algorithm takes two operands \( X \) and \( Y \) and computes \( MM(X, Y) = XY4^{-n} \mod M \), where \( 2m > n, n \) represents the operands’ precision, and \( M \) is the modulus. The value of the partial product \( S \) at a given iteration \( i \), may be expressed by \( S_i = (S_{i-1} + z_iY + q_iM) \), where \( 0 \leq i \leq m - 1 \). Observe that \( S \) may have positive or negative values.

Using the recoding scheme proposed in this work, there is an invariant for each iteration of the algorithm, given as \( |S| < \frac{3}{4}M + \frac{3}{4}Y \).

Proof: after the first loop iteration, the value of \( S \) is given as \( S = z_iY + q_iM \). 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 \( z_iY \) can contribute for the value of \( S \) after the first iteration is \((2Y)/4\). The contribution of these multiples after the second iteration adds up to \(2Y/4 + 2Y/4\), 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 \( \sum_{i=0}^{p-1} 4^i = (2^k - 1) = (2^k - 1), \) then \( \sum_{i=0}^{p-1} 4^i = \frac{(2^k - 1)}{2^k - 1} \), and the above summation results in:

\[ \frac{2Y}{4^p} \left( \frac{4^p - 1}{4 - 1} \right) = \frac{2}{3} (Y - \frac{Y}{4^p}) < \frac{2}{3} Y \]
The addition of shifted \( z_i Y \) multiples results in a contribution of at most \( \frac{3}{2} Y \) to the value of \( S \). When considering the \( M \) multiples it is observed that the maximum positive value for \( q_i^* \) is also 2, the same reasoning is used for \( q_i^* M \), and largest contribution of these multiples sums up to \( \frac{3}{2} M \). Similar calculations can be done to the negative range of values, however \( q_i^* = -1 \) is the most negative value and \(-\sum_{i=0}^{m-1} 4^{i} \frac{M}{2^i} = -\frac{M}{4^{m-1}} = -\frac{1}{4} M \). So, \(-\frac{1}{4} M - \frac{3}{2} Y < S < \frac{1}{2} Y + \frac{3}{2} M \). Therefore, \(|S| < \frac{3}{2} M + \frac{3}{2} Y \) after each loop.

Using the conditions \(|X| < M < 4^{m-2}\) and \(|Y| < M\), with \( R = 4^m\), we are able to show that \(|S| < M\).

Proof: The contribution of the multiples of \( M \) is still the same as in the previous case, however, the contribution of \( Z_i Y \) is different. Using the given condition, the MS digit of the recoded multiplier \( X \), \( Z_{m-1} \), is zero, since \(|X| < 4^{m-2}\), and \( Z \) can have at most the digit \( Z_{m-2} \neq 0 \) (forced by sign information and recoding scheme). With this condition, the sum of the maximum positive values for \( z_i Y \) results in:

\[
\sum_{i=0}^{m-2} 4^i (2) \left( \frac{3}{4^m} \right) = 2 \frac{Y}{4^m} (\frac{4^{m-1}}{3} - 1) = \frac{2}{3} Y - \frac{Y}{4^m} < \frac{Y}{6}
\]

Thus, the condition after the last iteration of the for loop is: \( S < \frac{3}{2} M + \frac{3}{2} Y \) and since \( Y < M \) then \( S < \frac{3}{2} M + \frac{1}{2} M = \frac{5}{2} M \), and consequently \( S < M \). Based on the symmetry of the values of \( z_i \), using the same procedure it is possible to prove that \( S > -M \). Thus, \(|S| < M\) 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 \( X \) are considered.

IV. OVERALL ORGANIZATION

The top level organization of the radix-4 Montgomery multiplier is shown in Fig. 2. The main functional blocks are Kernel Datapath, I/O & Memory, and the Control block. The computation takes place in the kernel datapath.

Carry-Save (CS) Adders are used in the kernel. Since the kernel output may be used again to complete the variable-precision computation, the internal variable \( S \) is kept in CS form (vectors \( SS \) and \( SC \)). The final result is converted to non-redundant form only at the end (CS converter inside the I/O & Memory block).

The bits of \( X \) needed to compute the recoded digits \( Z_j \) are provided by the signal \( X_{EXT_j} \). Other inputs to the Kernel datapath are words of \( Y \), \( M \), and \( S \), which are provided by the I/O & Memory. To identify one word of a bit-vector, the superscript star (*) was used. A new word is applied to the kernel in every clock cycle. The kernel has two configuration parameters: the number of stages \((NS)\) and \( w \). The operands must pass through the kernel datapath several times, depending on the values of these two variables and the precision of the operands (pipeline cycles).

The I/O & Memory block provides the interface between the user and the memory elements for the operands, modulus, and partial result. It is also responsible to deliver words of the required data to the Kernel at the proper time. More information about the system organization may be found in [4], [6].

The kernel datapath is organized as a pipeline of Processing Elements (PE) as shown in Fig. 3. Each PE implements one iteration of the R4MM algorithm (steps 3 to 12). A stage consists of a PE and a register. \((NS + 2)\) bits of \( X \) are transferred to the kernel over \( 2*NS \) clock cycles in each pipeline cycle. Each stage needs these bits at different times, thus, this signal is made common to all stages with internal control loading the signal in the right stage at the right time.

The newly computed words of \( SS \) and \( SC \), together with words of \( Y \) and \( M \), are propagated by each stage to the next, which performs another algorithm iteration and on its turn propagates the same type of data to the following stage after a latency of 2 cycles.
TABLE I
Booth recoding for $X_{EXT_j}$

<table>
<thead>
<tr>
<th>$X_{EXT_j}$ (2 : 0)</th>
<th>$Z_j$</th>
<th>$ZDN$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>000</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>000</td>
</tr>
<tr>
<td>011</td>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>011</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

V. Radix-4 Processing Element

Fig. 4 shows a more detailed diagram of the radix-4 PE. Booth encoding is performed by the module DEC.X.J according to Table I, using a 3-bit code ZDN: Z-zero, D-double, and N-negative. A 2-input mux and a complementer are used to generate the multiples of $Y$. The way ZDN is connected to the mux and complementer is shown in the Figure. The carry-in of 1 used to obtain $-Y$ and $-2Y$ must be inserted only when the LS word is applied as input and after that the carries from previous word additions should be inserted. A mux is connected to the carry input of CSA0 in order to perform this task.

![Fig. 4. Radix-4 Processing Element.](image)

The addition operation $S + Z_jY$ is performed cooperatively by the carry-save adders CSA0 and CSA1. CSA0 operates on the LS bits of words $S^{(j)}$ and $(Z_jY)^{(j)}$ while CSA1 operates on the MS bits of words $S^{(j-1)}$ and $(Z_jY)^{(j-1)}$ (previous words). This arrangement requires that the carry-out between these two adders be carefully considered. The carry-out of CSA1, $adder1\_cout$, is introduced immediately as carry-in for CSA0. The carry and sum bits of the CSA0 $(SumA(1 : 0), CarryA(2 : 0))$, on the other hand, are registered and applied as inputs to CSA2 one clock cycle later.

The signal first_cycle is used when the LS words are coming as inputs to the first section, and it is delayed one clock cycle first_cycle_reg before it is used in the second section.

The addition performed by CSA2 is done such that the LS two bits of $SumB^{(0)}$ and $CarryB^{(0)}$ (values at step 0) correspond to zeros. Because CS representation is used, these zero bits may be represented by, for example: $SumB^{(0)} = x\ldots x11$ and $CarryB^{(0)} = x\ldots x01$, where $x$ represents any value of the bit in this position. Therefore, data will be lost if the two LS bits of $SumB^{(0)}$ and $CarryB^{(0)}$ are simply discarded without taking into account the carry propagation $(11 + 01 = 100)$. The carry bit generated in this case is called "hidden-bit". This bit is not immediately assimilated in the addition process, but it is transferred to the next PE, where it is inserted as carry input of CSA2 at step 0.

To detect the hidden-bit it is enough to test the condition $SumB(1) = 1$. This signal is stored into a flip-flop and passed to the next PE as hidden_out. The hidden-bit is used to compute $q'_M_j$ (CS converter in Fig. 4) and it is also used as a carry input for CSA2 during the first word computation (step 0). 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 determination of the recoded quotient digit ($q'_M_j$) is done by the module DEC.M.J. Knowing that the LS bit of $M$ is always 1 ($M$ is odd), $q'_M_j$ will depend only on six bits: $SumA(1 : 0)$ (let us call it $SA$), $CarryA(1 : 0)$ (let us call it $CA$), hidden-bit and $M(1)$. DEC.MJ is basically a table. The number of table entries 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 $(SA+CA+hidden\_bit)\mod 4$ and reduces the Table for $q'_M_j$ to 8 entries only as shown in Table II. The output of DEC.MJ is the control signal for the 4-input multiplexer used to select the multiples of $M$.

The carry input of CSA2 cannot be used to obtain the value $-M$ required when $q'_M_j = -1$. The mux attached to the carry input of CSA2 has the hidden-bit-reg and the delayed carry-out from the same adder as inputs. So, there is no place to introduce the extra 1 needed to obtain $-M$. proportional to

![Table II](image)

<table>
<thead>
<tr>
<th>$M_1^{(0)}$</th>
<th>$NR_Sum(1..0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

The addition performed by CSA2 is done such that the LS two bits of $SumB^{(0)}$ and $CarryB^{(0)}$ (values at step 0) correspond to zeros. Because CS representation is used, these zero bits may be represented by, for example: $SumB^{(0)} = x\ldots x11$ and $CarryB^{(0)} = x\ldots x01$, where $x$ represents any value of the bit in this position. Therefore, data will be lost if the two LS bits of $SumB^{(0)}$ and $CarryB^{(0)}$ are simply discarded without taking into account the carry propagation $(11 + 01 = 100)$. The carry bit generated in this case is called "hidden-bit". This bit is not immediately assimilated in the addition process, but it is transferred to the next PE, where it is inserted as carry input of CSA2 at step 0.

To detect the hidden-bit it is enough to test the condition $SumB(1) = 1$. This signal is stored into a flip-flop and passed to the next PE as hidden_out. The hidden-bit is used to compute $q'_M_j$ (CS converter in Fig. 4) and it is also used as a carry input for CSA2 during the first word computation (step 0). 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 determination of the recoded quotient digit ($q'_M_j$) is done by the module DEC.M.J. Knowing that the LS bit of $M$ is always 1 ($M$ is odd), $q'_M_j$ will depend only on six bits: $SumA(1 : 0)$ (let us call it $SA$), $CarryA(1 : 0)$ (let us call it $CA$), hidden-bit and $M(1)$. DEC.MJ is basically a table. The number of table entries 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 $(SA+CA+hidden\_bit)\mod 4$ and reduces the Table for $q'_M_j$ to 8 entries only as shown in Table II. The output of DEC.MJ is the control signal for the 4-input multiplexer used to select the multiples of $M$.

The carry input of CSA2 cannot be used to obtain the value $-M$ required when $q'_M_j = -1$. The mux attached to the carry input of CSA2 has the hidden-bit-reg and the delayed carry-out from the same adder as inputs. So, there is no place to introduce the extra 1 needed to obtain $-M$. proportional to

![Table II](image)

<table>
<thead>
<tr>
<th>$M_1^{(0)}$</th>
<th>$NR_Sum(1..0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>
The solution of this problem comes from the fact that \( M \) is odd, which forces the least significant bit of \( -M \) to 1. By using this fact we can get \( -M \) by performing bit complementation on all the bits of \( M \) (except bit 0), and forcing a value 1 at bit position 0:

\[
-M = \overline{M}_{(w-1)} \& M_0 = \overline{M}_{(w-1)} \& '1'
\]

where \( \overline{x} \) represents the bit complementation of \( x \), and \( \& \) indicates concatenation. The LS bit must than be controlled by the signal first_cycle.

The multiples \( 2Y \) and \( 2M \), require a left-shift. Caused by the word-serial nature of this algorithm, the shifting requires the MS bit from the previous words of \( Y \) and \( M \) to be used with the coming words. If it is the first word (first_cycle_reg = 1), then a zero is shifted in to produce the needed multiple. Otherwise, the MS bit of the previous word is attached as the LS bit of the current word.

VI. EXPERIMENTAL RESULTS AND ANALYSIS

The experimental data presented in this Section were generated using Mentor Graphics CAD tools. The target technology was set to AMI05 fast aut0 (0.5 μm CMOS with hierarchy preserved) provided in the ASIC Design Kit (ADK) from the same company [9]. The experimental data for radix-2 and radix-8 kernel implementations were taken from [10]. The radix-4 design presented in this paper was described in VHDL and simulated in ModelSim for functional correctness. It was synthesized using Leonardo synthesis tool for the mentioned technology [11]. ADK provides a consistent environment for comparison between the designs, and a reasonable approximation of the system performance when using commercial ASIC technology.

The area for the radix-4 kernel was extracted from the experimental data as:

\[ A_{kernel_{4}} = 62.86 \times NS \times w + 146 \times NS - 4.875 \times w - 13. \]

The total computational time for the kernel is a product of the number of clock cycles (\( T_{CLKs} \)) it takes and the clock period (\( t_{p} \)) derived form the synthesis results. The clock period is affected by the number of stages and word size. The number of clock cycles is obtained from the algorithm and the kernel organization. Table III shows the average critical path delay for radix-4 design. The complete experimental results are presented in [11]. The results for radix-2 and radix-8 designs show that \( NS \) has more significant effect on the critical path, caused mainly by the wire delay introduced in the layout phase. This phase was not done for the radix-4 design.

The same two cases considered in the analysis of designs for radix 2 and 8, should be considered in radix-4 analysis. The two cases are when \( e \leq 2 \times NS \) and \( e > 2 \times NS \). Where as before the variable \( e = \lceil \frac{N_{s}+1}{w} \rceil \) is the number of words in the \( N \)-bit operands, with word size \( (w) \) [4]. As

<table>
<thead>
<tr>
<th>Bits Per Word</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical path delay for radix-4 kernel (nsec).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>6.0</td>
<td>6.3</td>
<td>6.5</td>
<td>6.8</td>
<td></td>
</tr>
</tbody>
</table>

in radix-8 design, a word propagates through the pipeline for \((2NS+1)\) clock cycles [10], [6]. The speed of scanning the bits of \( X \) for radix-4 is two bits per stage, or \( \lceil \frac{N}{2NS} \rceil \). With this observation made, Equation VI represents the total number of clock cycles needed for radix-4 design.

\[ T_{CLKs} = \begin{cases} \left\lfloor \frac{N}{2NS} \right\rfloor (\left\lfloor \frac{N}{w} \right\rfloor + 1) + 2NS & \text{if } \left\lfloor \frac{N}{w} \right\rfloor > 2NS \\ \left\lfloor \frac{N}{2NS} \right\rfloor (2NS + 1) + \left\lfloor \frac{N}{w} \right\rfloor + 1 & \text{otherwise} \end{cases} \]

Fig. 5 shows the total computational time \( (T_{CLKs} \times t_{p}) \) for 1024-bit operands as a function of the number of stages in the kernel pipeline. For different operand precision the computational time will reach its minimal value for different number of stages. For 256-bit operands, there is no benefit in using more than 8,000 gates. However, when the precision increases, a larger number of stages (more area) would provide significantly more performance. So, the small precision case is the worst case scenario for the comparison among designs.

![Graph showing total computation time for radix-4 kernel, 1024 operands.](image)

The goal of choosing best design point is to have computational time for 256-bit precision close to its absolute minimal value and at the same time to have as small computational time for 1024-bit precision as possible, provided that it has reasonable design area. For 256-bit operands with \( w = 8 \), the best design point obtained for 16 PEs, with an area of 10,330 gates. Each additional stage adds 649 gates to the area compared with 1,005 gates in radix-8 [6]. Table IV compares several design points for the radix-4 kernel with \( w = 8 \) bits. The Table presents the design area and the ratio of the computational time related to the point \( NS = 16 \). From previous work [12], [10] it was
also concluded that the word size of 8 bits is the one that provides the best design points.

<table>
<thead>
<tr>
<th>TABLE IV</th>
<th>Optimal design points for radix-4 kernel, 8-bit word size, 256-bit and 1024-bit operand precision.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>16</td>
</tr>
<tr>
<td>Area, gates</td>
<td>10330</td>
</tr>
<tr>
<td>( \frac{\text{time}}{\text{NS}} ) (256-bit)</td>
<td>1</td>
</tr>
<tr>
<td>( \frac{\text{time}}{\text{NS}} ) (1024-bit)</td>
<td>1</td>
</tr>
</tbody>
</table>

It can be seen that the design point with \( NS = 26 \) is very suitable since the computational time for 256-bit precision is almost the same as its minimal value while the computational time for 1024-bit precision is improved by 56% as compared to the point with \( NS = 16 \).

Fig. 6 shows a comparison of performance between the radix-4 design discussed in this paper, the radix-2 design [4], and the radix-8 design [6]. The data shows the time to compute the modular multiplication for 256-bit operands as a function of the design area. For small areas, the radix-2 design performs as well as the radix-8 design. For a design area of 9,000 gates or more, the radix-8 design is better than the radix-2 design. The radix-4 design performs better than the other two designs for all area values. Since the data in [10] was obtained using a slightly more detailed synthesis process, the data for radices 2 and 8 were synthesized again using the same steps and technology applied to radix 4. These data are shown in the lines labeled **radix-2 fast** and **radix-8 fast** in the Figure. The data are slightly better than the one presented in [10], but the radix-4 solution is still significantly better than the other two cases. The complete data analysis is presented in [11].

VII. Conclusion

The use of radix 4 has been proven to be efficient in the implementation of several arithmetic operations. It was not different for our scalable Montgomery multiplier. The proposed design made use of recoding schemes to reduce the complexity of multiple generation and multi-operand addition. The result was a design that is as slim as a radix-2 design, and can actually work at twice the performance. Comparing the radix-4 design with previously presented radix-2 and radix-8 designs for the same area, it was possible to confirm that the solution is significantly better. Previous work done for radix 2 and radix 8 [6], [7], [4] already demonstrated that this hardware solution is better than using a microprocessor at the same clock speed. The gain using radix 4 is even bigger.

Acknowledgments This work was supported by NSF CAREER grant CCR-0093434- "Computer Algorithms and Scalable Hardware Designs for Cryptographic Applications" and by Jordan University of Science and Technology (JUST).

References


