Radix – 4 Implementation of a Montgomery Multiplier for a RSA Cryptosystem

Steven Hubbard

Abstract – As cryptosystems become more important to computing, so does the need to make the encryption and decryption of larger and stronger keys faster. The RSA algorithm presents one of these problems specifically in performing modular exponentiation. A popular method to implement the necessary modular multiplication and squaring required for exponentiation is the Montgomery Multiplication (MM) algorithm. This project is an extension of a thesis done by Allen Michalski that explores the use of Montgomery Multiplication algorithms and looks to include a radix-4 implementation of the multiplications done in the Montgomery domain using FPGAs, optimizing for speed and area. It then shows the additions necessary for a radix-4 Montgomery Multiplier implantation, as well as the potential benefits of the architecture.

I. Introduction and Background

RSA, as proposed by Rivest, Shamir, and Adleman in 1978, is one of the simplest, most effective public-key cryptosystems used today. The algorithm is based on a public-key and a private-key consisting of the following variables. The private key contains two large prime numbers, \( p \) and \( q \), as well as a secret exponent \( d \). The public key then consists of the modulus \( n = p \cdot q \), and the exponent \( e \). The public exponent \( e \) is a number such that:

\[
\gcd(e, \phi) = 1 \text{ where } 1 < e < \phi \\
\phi = (p - 1)(q - 1)
\]

The private exponent \( d \) is then obtained by inverting \( e \) modulo \( n \):

\[
d = e^{-1} \mod \phi
\]

Therefore the encryption operation is performed by computing

\[
C = M^e \mod n
\]

And decryption is the inverse

\[
M = C^d \mod n
\]

Where \( M \) is the plaintext message and \( C \) is the cipher text. Because of this algorithm's reliance on modular exponentiation, which is often implemented by a series of multiplications and additions, it is advantageous to find a faster method for these calculations. As stated above, a common algorithm used to speed-up the multiplication is the use of Montgomery Multiplication proposed by Peter L. Montgomery. The Montgomery Multiplication algorithm is used to calculate the modular multiplications and squaring required during the exponentiation process by converting the numbers to the Montgomery domain to perform the calculations needed.

II. Modular Montgomery Multiplication and Exponentiation

Before discussing exponentiation in the Montgomery domain, it is advantageous to first look at the algorithms used for normal binary exponentiation. The algorithms commonly used for these calculations were derived around 200 B.C. in India called “Chandah-Sûtra”. There are two different methods of implementing this algorithm, right to left, and left to right, and are shown below:

Right-to-left binary exponentiation

\[
E = (e_{L-1}, e_{L-2}, \ldots, e_1, e_0)_2 \\
Y = 1; \\
S = X; \\
\text{for } i = 0 \text{ to } L-1
\]
\{ 
  \text{if}(e_i == 1) 
  \quad Y = Y \cdot S \mod N; 
  \quad S = S^2 \mod N;
\}

**Left-to-right binary exponentiation**

\[
Y = 1; \quad \text{for} \ i=L-1 \ \text{downto} \ 0 \\
\{ 
  \quad Y = Y^2 \mod N; \\
  \quad \text{if}(e_i == 1) 
  \quad \quad Y = Y \cdot X \mod N;
\}
\]

*Figure – 1 [3]*

Although at first these algorithms appear to be executable in the same amount of time, because the “if” statement in the left-to-right method assigns a value to the same variable, Y, as the previous line, these operations cannot be done in parallel. The right-to-left method however assigns values to two different variables, S and Y, allowing implementations in hardware to take advantage of a parallel structure.

The principles of Montgomery Multiplication can then further increase the speed to calculate this algorithm. The Montgomery Multiplication algorithm provides the advantage that the division step in taking the modulus is replaced by bit-shifting, and easy to implement in hardware operation. For Montgomery Multiplication, given an integer \(a < M\), where \(M\) is the \(k\)-bit modulus, \(A\) is said to be its \(M\)-residue with respect to \(r\) if,

\[
A = a \cdot 2^k \pmod{M}
\]

Likewise, given an integer \(b < n\), \(B\) is said to be its \(n\)-residue with respect to \(r\) if,

\[
B = b \cdot 2^k \pmod{M}
\]

The Montgomery product of \(A\) and \(B\) can then be defined as,

\[
C = A \cdot B \cdot r^{-1} \pmod{M}
\]

where \(r^{-1}\) is the inverse of \(r\), modulo \(M\). [11]

When these transformations are applied to the *Chandah-Sûtra* algorithm the result is the following:

**Right-to-left binary exponentiation**

\[
\begin{align*}
E &= (e_{L-1}, e_{L-2}, \ldots, e_1, e_0)_2 \\
Y &= 1; \\
S &= X; \\
Y' &= MP(Y', 2^k \mod N, N) \\
S' &= MP(S', 2^k \mod N, N)
\end{align*}
\]

\text{for} \ i=0 \ \text{to} \ L-1 \\
\{ 
  \quad \text{if}(e_i == 1) 
  \quad \quad Y' = MP(Y', S', N); 
  \quad S' = MP(S', S', N);
\}

\quad S = MP(S', 1, N);

*Figure – 2 [3]*

In the case of Michalski’s thesis this algorithm was expanded to consider \(w\)-bit words for better scalability. Therefore operands with \(m\) bits of precision, \(e = [m/w]\) words are required. In the proposed algorithm, operand \(Y\) (multiplicand) is scanned word-by-word, and the operand \(X\) (multiplier) is scanned bit-by-bit. The algorithm is called Multiple Word Radix-2 Montgomery Multiplication algorithm (MWR2MM) [3]. The process for this new algorithm is:

\[
S = 0 \quad -- \text{initialize all words of \(S\)} \\
\text{for \(i = 0 \ \text{to} \ m-1\)}
\]

\}

\[
C = c \cdot 2^k \pmod{M}
\]
\[ S^{(0)} := x_i Y^{(0)} + S^{(0)} \]

if \( S^{(0)} = 1 \) then
\[
(C, S^{(0)}) := S^{(0)} + M^{(0)}
\]

for \( j = 1 \) to \( e-1 \)
\[
\{
(C, S^{(j)}) := C + x_i Y^{(j)} + S^{(j)}
S^{(j-1)} := (S^{(j)}, S_{w-1 \ldots 1}^{(j-1)})
\}
\]

\( S^{(e-1)} := (C, S_{w-1 \ldots 1}^{(e-1)}) \)

else
\[
C := 0
\]

for \( j = 1 \) to \( e-1 \)
\[
\{
(C, S^{(j)}) := C + x_i Y^{(j)} + S^{(j)}
S^{(j-1)} := (S^{(j)}, S_{w-1 \ldots 1}^{(j-1)})
\}
\]

\( S^{(e-1)} := (C, S_{w-1 \ldots 1}^{(e-1)}) \)

This produces a partial sum \( S \) for each bit of \( X \), while scanning the words of \( Y \) and \( M \). It is then repeated for the next bit of \( X \). The operations are therefore performed in \( w \) bits precision, independent of the precision of the operands. The total number of cycles required by the algorithm = \( m.e = m^2/w \) [3]. The results from this implementation will be discussed later, however, as stated in its title this was done only in radix-2.

For the implementation of this algorithm done by Michalski, the data is broken into multiple Processing Elements shown below:

\[ \text{Figure – 3 [3]} \]

This segment shows the definition and simple implementations of a radix-4 multiplier. A standard approach to perform multiplication is to "shift and add". That is, for each column in the multiplier, shift the multiplicand the appropriate number of columns and multiply it by the value of the digit in that column of the multiplier, to obtain a partial product. The partial products are then added to obtain the final result. With this system, the number of partial products is exactly the number of columns in the multiplier.

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1
\end{array}
\]

It is possible to reduce the number of partial products by half, by using the technique of radix-4 Booth recoding. Instead of shifting and adding for every column of the multiplier term and multiplying by 1 or 0, take every second column, and multiply by \( \{0,3\} \), or use Booth’s Recoding to change to \( \{-2, 2\} \), to obtain the same results. For example, to multiply by 7, multiply the partial product aligned against the least significant bit by -1, and multiply the partial product aligned with the third column by 2:

Partial Product 0 = Multiplicand * -1, shifted left 0 bits (x -1)
Partial Product 1 = Multiplicand * 2, shifted left 2 bits (x 8)

This is the same result as the equivalent shift and add method:

Partial Product 0 = Multiplicand * 1, shifted left 0 bits (x 1)
Partial Product 1 = Multiplicand * 1, shifted left 1 bits (x 2)
Partial Product 2 = Multiplicand * 1,
shifted left 2 bits (x 4)
Partial Product 3 = Multiplicand * 0,
shifted left 3 bits (x 0)

The advantage of this method is the halving of the number of partial products which in circuit design relates to the propagation delay in the running of the circuit, and the complexity and power consumption of its implementation. It is also important to note that there is comparatively little complexity penalty in multiplying by 0, 1 or 2. All that is needed is a multiplexer which has a delay time that is independent of the size of the inputs. [6]

The following shows the basic structure of the radix-4 multiplier

![Figure - 5: Radix-4 Implementation [0,3] [6]](image)

Because the operand 3a is slow to calculate, it is faster to subtract a and force a carry of 4a as shown below:

![Figure - 6: Radix-4 implementation [-1,2] [6]](image)

A final architecture using Carry Save Adders (CSA) shows the circuit with the partial product adder.

![Figure - 7: Radix-4 Architecture using CSAs [6]](image)

As stated above, radix-4 multiplication is easier to calculate over the range [-2, 2] rather then [0, 3]. The process for converting the bitstream accordingly is called Booth’s Recoding, and is described in the table below.

### Radix-4 Booth’s recoding yielding \((x_{i+1} \cdots x_0)_{10}\)

<table>
<thead>
<tr>
<th>(n)</th>
<th>(x_i)</th>
<th>(x_{i+1})</th>
<th>(y_{i+1})</th>
<th>(y_i)</th>
<th>(z_{i+1})</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No string of 1s in sight</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>End of a string of 1s in r</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>Isolated 1 in r</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>End of a string of 1s in r</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>Beginning of a string of 1s in r</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Beginning of a string of 1s</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Continuation of string of 1s in r</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Continuation of string of 1s</td>
</tr>
</tbody>
</table>

![Figure - 8: Booth Recoding [6]](image)

Combining this recoding with the use of CSA, the final architecture that will be used to implement radix-4 multiplication is:
IV. Radix – 4 Montgomery Multiplier Implementation

The primary design of a radix-4 Montgomery Multiplier that will be looked at in this paper is one that was presented by Lo'ai A. Tawalbeh, Alexandre F. Tenca, and C.K. Koc in their paper “A Radix-4 Design of a Scalable Modular Multiplier With Recoding Techniques” [1]. They propose an extension of the Multiple Word Radix-2 Montgomery Multiplication algorithm (MWR2MM) to include High-Radix (R2k) multiplication. Again modifying the basic algorithm for binary exponentiation a new procedure is described as follows:

Step
1: \[ S = 0 \]
   \[ x_1 = 0 \]
2: \[ \text{FOR } j = 0 \text{ TO } N - 1 \text{ STEP 2} \]
3: \[ Z_j = \text{Recoding}_1(x_{j+1..j+3}) \]
4: \[ (C_a, S^{(0)}) = S^{(0)} + (Z_j * Y)^{(0)} \]
5: \[ q_{M_j} = S^{(0)}_{1..0} * (4 - M^{(0)-1}_{1..0}) \text{ mod 4} \]
5a: \[ q'_{M_j} = \text{Recoding}_2(q_{M_j}) \]
6: \[ (C_b, S^{(0)}) = S^{(0)} + (q'_{M_j} * M)^{(0)} \]
7: \[ \text{FOR } i = 1 \text{ TO } e - 1 \]
8: \[ (C_a, S^{(0)}) = C_a + S^{(0)} + (Z_j * Y)^{(0)} \]
9: \[ (C_b, S^{(0)}) = C_b + S^{(0)} + (q'_{M_j} * M)^{(0)} \]
10: \[ S^{(i-1)} = (S^{(i-1)}_{1..0} + S^{(i-1)}_{BPW-1..2}) \]
   \[ \text{END FOR;} \]
11: \[ C_a = C_a \text{ or } C_b \]
12: \[ S^{(e-1)} = \text{signext}(C_a, S^{(e-1)}_{BPW-1..2}) \]
   \[ \text{END FOR;} \]

- \( X_j \) - a single radix-4 digit of \( X \) at position \( j \);
- \( q_{M_j} \) - quotient digit that determines a multiple of the modulus \( M \) to be added to the partial product \( S \);
- \( e \) - number of bits in a word of either \( Y, M \) or \( S \);
- \( e = \left\lceil \frac{w+1}{w+1} \right\rceil \) - number of words in either \( Y, M \) or \( S \);
- \( N \) - number of stages;
- \( C_a, C_b \) - carry bits;
- \( (Y^{(e-1)}, \ldots, Y^{(1)}, Y^{(0)}) \) - operand \( Y \) represented as multiple words;
- \( S^{(0)}_{k+1..0} \) - bits \( k - 1 \) to 0 of the \( j \)th word of \( S \).

The recoding used in this algorithm is the same as discussed before. Recoding1 is Booth recoding to change \([0, 3]\) to \([-2, 2]\). Similarly, recoding2 will convert \([0, 3]\) to \([-1, 2]\) as shown in Figure – 5 of a radix-4 multiplier.

The overall structure of the multiplier can be divided into three main sections, Datapath (or Kernel), I/O & Memory, and the Control Block. The I/O & Memory and the Control Block have been created by Michalski, and are not significantly changed by the addition of the radix-4 architecture. Tawalbeh, Tenca, and Koc propose a pipelined organization of Precessing Elements (PE) separated by registers to perform one iteration of the MWR4MM algorithm, similar to the PE in the radix-2 implementation shown above. The processing element consists of booth recoding, multiple generation, multi-precision Carry-Save Adders, \( q'_{M_j} \) table and registers. The first segment of the PE computes the two least-significant bits of each word. These are then used to compute \( q'_{M_j} \) for the rest of the calculation. Therefore for each clock cycle one word of \( Y, M, SS, \) and \( SC \) is calculated.

Several of these small PEs then work concurrently to perform multiple iterations of the R4MM algorithm at once.

Comparing this to the radix-2 implementation shows a much more complex data path. In particular with the addition of the additional register needed to calculate the significant bits. The signals used in both the radix-2 and radix-4 organizations are the same, therefore the entity in VHDL can be reused. However, the width of the signals as well as the control from the rest of the circuit will have to be...
changed to allow for the more complex structure. However, if successful, the output of the processing element can be significantly improved as shown below.

V. Results

There are varying results of these implementations depending on which methods and algorithms are used for the multiplication. With a radix-2 implementation of the MWR2MM algorithm Michalski’s circuit did not perform as well as a comparable software implementation. He was, however, able to show the relationship between time in cycles and the number of stages for different sizes of m.

Other researchers using the Radix-4 Montgomery Multiplier also found similar results. Hong and Wu from the National Tsing Hua University implemented a pipelined radix-4 modular multiplier which also performed at twice the speed of the original Montgomery algorithm [9]. Higher radix multipliers have also been looked at to find the best solution. Blum and Paar from the Worcester Polytechnic Institute implemented a radix-16 multiplier using an extended version of this technique. There results were better then a simple radix-2, but it was not shown that a significant increase in speed over a radix-4 [8]. Therefore from the research done for this paper and by Tawalbeh, Tenca, and Koc, the radix-4 appears to be the best way to implement Montgomery Multiplication.

VI. Further Optimization and Study

The experiment left by Michalski failed to optimize the use of Montgomery Multiplication using an FPGA. It was not able to significantly outperform software implementations of the RSA calculations. It is hoped that by introducing a radix-4 processing element, a significant increase in performance can be achieved. This can be done using the techniques described above for the MWR2-MM along with the architecture shown in Figure – 11. Once the radix-4 processing element is operational, the final optimization will come from analyzing the number and structure of these processing elements, as well as the other circuitry in design. With the increase in speed of the most process intensive operations in RSA it should then allow for larger key values, and
therefore more security without a sacrifice to performance.

VII. References


