Simulation Model for a Hardware Implementation of Modular Multiplication

NADIA NEDJAH AND LUIZA DE MACEDO MOURELLE
Department of Systems Engineering and Computation,
State University of Rio de Janeiro
São Francisco Xavier, 524, 5th. Andar, Rio de Janeiro,
BRAZIL

Abstract: Modular multiplication is fundamental to several public-key cryptography systems such as the RSA encryption system. It is also the most dominant part of the computation performed in such systems. The operation is time consuming for large operands. This paper examines the characteristics of yet another architecture to implement modular multiplication. An experimental modular multiplier prototype is described in VHDL and simulated. The simulation results are presented.

Key-Words: Modular multiplication, simulation, cryptosystems.

1 Introduction

The modular exponentiation is a common operation for scrambling and is used by several public-key cryptosystems, such as the RSA encryption scheme [1]. It consists of a repetition of modular multiplications: \( C = T^E \mod M \), where \( T \) is the plain text such that \( 0 \leq T < M \) and \( C \) is the cipher text or vice-versa, \( E \) is either the public or the private key depending on whether \( T \) is the plain or the cipher text, and \( M \) is called the modulus. The decryption and encryption operations are performed using the same procedure, i.e. using the modular exponentiation.

The performance of such cryptosystems is primarily determined by the implementation efficiency of the modular multiplication and exponentiation. As the operands (the plain text of a message or the cipher or possibly a partially ciphered) text are usually large (i.e. 1024 bits or more), and in order to improve time requirements of the encryption/decryption operations, it is essential to attempt to minimise the number of modular multiplications performed and to reduce the time requirement of a single modular multiplication.

An RSA cryptosystem consists of a set of three items: a modulus \( M \) of around 1024 bits and two integers \( d \) and \( e \) called private and public keys that satisfy the property \( T^e = T \mod M \). Plain text \( T \) obeying \( 0 \leq T < M \). Messages are encrypted using the public key as \( C = T^e \mod M \) and decrypted as \( T = C^d \mod M \). So the same operation is used to perform both processes: encryption and decryption. Hardware implementations of the RSA cryptosystem are widely studied as in [2, 3, 4].

In the rest of this paper, we start off by describing the algorithms used to implement the modular operation. Then we present the architecture of the hardware modular multiplier and explain in details how it executes a single multiplication. Then we comment the simulation results obtained for such architecture.

2. Multiplication algorithm

Algorithms that formalise the operation of multiplication generally consist of two steps: one generates a partial product and the other accumulates it with the previous partial products. The most basic algorithm for multiplication is based on the add-and-shift method: the shift operation generates the partial products while the add step sums them up [5].

The straightforward way to implement a multiplication is based on an iterative adder-accumulator for the generated partial products. However, this solution is quite slow as the final result is only available after \( n \) clock cycles, \( n \) is the size of the operands.

A faster version of the iterative multiplier should add several partial products at once. This could be achieved by unfolding the iterative multiplier and yielding a combinatorial circuit that consists of several partial product generators together with several adders that operate in parallel.

In this paper, we use such a parallel multiplier as described in Figure 1. Now, we detail the algorithms used to compute the partial products and to sum them up.
The possible values of \( \tilde{x}_i \) with the respective values of \( x_{j<s<i+1>} \) are \(-2\) (100), \(-1\) (101, 110), 0 (000, 111), 1 (001, 010) and 2 (011). This recoding will generate \([(n+2)/2]\) partial products.

Inspired by the above notation, the modified Booth algorithm [6, 7, 8] generates the partial products \( \tilde{x}_i \times Y \). These partial products can be computed very efficiently due to the digits of the new representation \( \tilde{x}_i \). The hardware implementation will be detailed in Section 3.

In the algorithm of Figure 3, the terms \( 4 \times 2^{m-1} \) and \( 3 \times 2^{m-1} \) are supplied to avoid working with negative numbers. The sum of these additional terms is congruent to zero modulo \( 2^{n+[(n+1)/2]} \). So, once the sum of the partial products is obtained, the rest of this sum in the division by \( 2^{n+[(n+1)/2]} \) is finally the result of the multiplication \( X \times Y \).

\[
\text{Algorithm ModMulti}(x_{2n+1}, x_{2n}, x_{2n-1}, Y) \{
  \text{int product } = 0;
  \text{int pp} \left(\lceil (n+1)/2 \rceil - 1\right); \text{pp}[0] = (\tilde{x}_0 \times Y + 4 \times 2^{n-1}) \times 2^{2x}; \\
  \text{for } i = 0 \text{ to } \left\lceil (n+1)/2 \right\rceil - 1 \{ \text{pp}[i] = (\tilde{x}_i \times Y + 3 \times 2^{n-1}) \times 2^{2x}; \text{product} = \text{product} + \text{pp}[i]; \} \\
  \text{return product mod } 2^n + \left\lceil (n+1)/2 \right\rceil - 1; \}
\]

Fig. 2: Multiplication algorithm.

3 Reduction algorithm

A modular reduction is simply the computation of the remainder of an integer division. It can be denoted by:

\[
X \mod M = X - \left\lfloor \frac{X}{M} \right\rfloor \times M
\]

However, a division is very expensive even compared with a multiplication. Using Barrett’s method [9, 10], we can estimate the remainder using two simple multiplications. The approximation of the quotient is calculated as follows:

\[
\left\lfloor \frac{X}{M} \right\rfloor = \left[ \frac{X}{2^{n-1}} \times \frac{2^{n+1}}{M} \right] \equiv \left[ \frac{X}{2^{n-1}} \times \frac{2^{2n}}{M} \right]
\]

The equation above can be calculated very efficiently as division by a power of two \( 2^i \) are simply a truncation of the operand’ x-least significant digits. The term \( \left\lfloor \frac{2^{2n}}{M} \right\rfloor \) depends only on the modulus \( M \) and is constant for a given modulus hence, can be pre-computed and saved in an extra register. Hence the approximation of the remainder using Barrett’s
method [9, 10] is a positive integer smaller than \(2 \times (M-1)\). So, one or two subtractions of \(M\) might be required to yield the exact remainder.

4. Modular multiplier architecture

In this section, we outline the architecture of the multiplier, which is depicted in Figure 3. Later on in this section and for each of the main parts of this architecture, we give the detailed circuitry, i.e. that of the partial product generator, adder and reducer.

The multiplier of Figure 4.1 performs the modular multiplication \(X \times Y \mod M\) in three main steps:

1. Computing the product \(P = X \times Y\);
2. Computing the estimate quotient \(Q = P / M \Rightarrow Q \equiv P / 2^{n-1} \times [2^{2^n} / M] - Q \times M\);
3. Computing the product \(Q \times M\);
4. Computing the final result \(P - Q \times M\).

During the first step, the modular multiplier first loads \(\text{register}_1\) and \(\text{register}_2\) with \(X\) and \(Y\) respectively; then waits for PPG to yield the partial products and finally waits for the ADDER to sum all of them. During the second step, the modular multiplier loads \(\text{register}_1\), \(\text{register}_2\), and \(\text{register}_3\), with the obtained product \(P\), the pre-computed constant \([2^{2^n} / M]\) and \(P\) respectively; then waits for PPG to yield the partial products and finally waits for the ADDER to sum all of them. During the third step, the modular multiplier first loads \(\text{register}_1\) and \(\text{register}_2\) with the obtained product \(Q\) and the modulus \(M\) respectively; then awaits for PPG to generate the partial products, then waits for the ADDER to provide the sum of these partial products and finally waits for the REDUCER to calculate the final result \(P - Q \times M\), which is subsequently loaded in the accumulator acc.

4.1 The multiplier

The multiplier interface is shown in Figure 4. It is composed of a partial product generator and an adder. The partial product generator is in turn composed of \(k\) Booth recoders [6, 7, 8] that communicate directly with \(k\) partial product selectors.

Fig. 3: The modular multiplier architecture.

![Fig. 3: The modular multiplier architecture.](image)

The interface of the Booth decoder is described in Figure 5.

Fig. 4: The partial product generator interface.

![Fig. 4: The partial product generator interface.](image)

The interface of the Booth decoder is described in Figure 5.

Fig. 5: The Booth recoder interface.

![Fig. 5: The Booth recoder interface.](image)
SelectY is set when the partial product to be generated is $Y = -Y$, the second one Select2Y is set when the partial product to be generated is $2 \times Y = -2 \times Y$, the last bit is the simply the last bit of the Booth digit given as input. It allows us to complement the bits of the partial products when a negative multiple is needed. The output signal are yielded from the input ones as in Figure 6:

```
SelectM <= lsb + midle
Select2M <= ~((midle + msb) + (lsb + midle))
Sign     <= msb
```

Fig. 6: Selection logic of the Booth decoder.

The required partial products, i.e., $\bar{x} \times Y$ are easy multiple. They can be obtained by a simple shift. The negative multiples in $2$’s complement form, can be obtained form the positive corresponding number using a bit by bit complement with a 1 added at the least significant bit of the partial product. The additional terms introduced in the previous section can be included into the partial product generated as three/two/one most significant bits computed as

- least significant bit of the partial product.
- the positive corresponding number
- negative multiples in 2’s complement form, can be obtained by a simple shift. The partial products, i.e., $\bar{x} \times Y$ are easy multiple. They can be obtained by a simple shift. The negative multiples in 2’s complement form, can be obtained form the positive corresponding number using a bit by bit complement with a 1 added at the least significant bit of the partial product. The additional terms introduced in the previous section can be included into the partial product generated as three/two/one most significant bits computed as follows, whereby, $\oplus$ is the bits concatenation operation, $A$ is the binary notation of integer $A$, $0^i$ is a run of $i$ zeros and $B < n:0^i$ is the selection of the $n$ less significant bits of the binary representation $B$.

\[
\begin{align*}
PP(0) &= (SelectM.M(0)) \oplus Sign \\
PP(1) &= (Select2M.M(0) + SelectM.M(1)) \oplus Sign \\
&\quad \ldots \\
PP(i) &= (Select2M.M(i-1) + SelectM.M(i)) \oplus Sign \\
&\quad \ldots \\
PP(n) &= (Select2M.M(n-1) + SelectM.M(n)) \oplus Sign \\
PP(n+1) &= (Select2M.M(n)) \oplus Sign
\end{align*}
\]

Fig. 7: The logic of the partial product selector.

The interface of the partial product selector is given in Figure 8 while the logic of the component is shown in Figure 8 and the corresponding logic in Figure 7.

4.2 The adder

In order to implement the adder of the generated partial products, we use a hybrid new kind of adder. It consists of an initial stage of carry save adders followed by a cascade of stages of delayed carry adders [11] and a final stage of full adder. The carry save adder (CSA) is simply a parallel ensemble of $f$ full adders without any horizontal connection. Its function is to add three $f$-bit integers $a$, $b$ and $c$ to yield two new integers carry and sum such that carry $+ \text{sum} = a + b + c$. The pair (carry, sum) will be called a delayed carry integer. The delayed carry adder (DCA) is a parallel ensemble of $f$ half adders. Its function is to add two delayed carry integers $(a_i, b_i)$ together with an integer $c$ to produce a delayed carry integer $(\text{sum}, \text{carry})$ such that \text{sum} $+$ carry $= a_i + b_i + c$. The general architecture of the proposed adder is depicted in Figure 9, where the partial products $PP_i$, $0 \leq i \leq 15$ are the input operands. Using the carry save adder, the $i$th bit of carry and sum are defined as $\text{sum}_i = a_i \oplus b_i \oplus c_i$ and $\text{carry}_i = a_i b_i + a_i c_i + b_i c_i$, respectively. The architecture of the delayed carry adder uses $5 \times n$ half adders as described in Figure 9.

4.3 The reducer

The main task of the reducer consists of subtracting $Q \times M$, i.e. the product obtained in the third step of the modular multiplier from $P$, i.e. the product yielded in the first step of the modular multiplier. A subtraction of an $p$-bits integer $K$ is equivalent to the addition of $2^p - x$. Hence the reducer simply performs the addition $P + (2^p - Q \times M)$. The latter value is simply the two’s complement of $Q \times M$.

The addition is performed using a carry look-ahead adder. It is based on computing the carry bits $C_i$ prior to the actual summation. The adder takes advantage of a relationship between the carry bits $C_i$ and the input bits $A_i$ and $B_i$.

\[
C_i = G_i + C_{i+1} + \cdots + C_{i-1} + \cdots + G_i + C_{i+1} P_i + \cdots P_i
\]

whereby $G = A \times B$ and $P = A_i + B_i$. The general structure of the used carry look-ahead adder is given in Figure 10.

5 Simulation results

The project of the modular multiplier described throughout this paper was specified in Very High Speed Integrated Circuit Description Language - VHDL [12], and simulated using the MyVHDL Station of MyCad Inc. [13].
In order to synchronise the work of the MULTIPLIER, ADDER and REDUCER, we designed a module called the CONTROLLER that consists of a simple state machine, that has 13 states defined as follows, where $next(S_i) = S_{i+1}$ and $next(S_{12}) = S_0$:

- $S_0$: initialisation of the state machine;
- $S_1$: loads multiplicator into register $r_1$; loads multiplicand into register $r_2$;
- $S_2$: waits for the MULTIPLIER;
- $S_3$: waits for the ADDER;
- $S_4$: waits for the SHIFTER;
- $S_5$: loads the product obtained $P$ into register $r_3$; loads the constant into register $r_4$; loads $P$ into register $r_5$;
- $S_6$: waits for the MULTIPLIER;
- $S_7$: waits for the ADDER;
- $S_8$: loads the product obtained $Q$ into register $r_3$; loads the modulus $M$ into register $r_4$;
- $S_9$: waits for the MULTIPLIER;
- $S_{10}$: waits for the ADDER;
- $S_{11}$: waits for the REDUCER;
- $S_{12}$: loads register $acc$ the final result.

In Figure 12 and 13, we show how the different registers of the modular multiplier are loaded, i.e. in which state, with the input data $X, Y, M$ and $Cte$ and/or with the results obtained from the three-steps multiplications, i.e. $P, Q, ModProd$.

### 6 Conclusion

In this paper, an alternative architecture for computing modular multiplication based on Booth algorithm and on Barrett’s relaxed residuum method is described.
The Booth algorithm is used to compute the product while Barrett’s method is used to calculate the remainder. The architecture was validated through behavioural simulation results using the 0.6µm CMOS-AMS standard cell library. The total execution time is 3570 nanoseconds for 1024-bit operands. One of the advantages of this modular multiplication implementation resides in the fact that it is easily scalable with respect to the multiplier and modulus lengths.

7 References


