Fast Montgomery Modular Multiplication
by Pipelined CSA Architecture

Kooroush Manochehri
Department of Computer Engineering and IT,
Amirkabir University of Technology, Tehran,
IRAN
kmanochehri@ce.aut.ac.ir

Saadat Pourmozafari
Department of Computer Engineering and IT,
Amirkabir University of Technology, Tehran,
IRAN
saadat@ce.aut.ac.ir

Abstract

Montgomery modular multiplication algorithm is commonly used in implementations of the RSA cryptosystem or other cryptosystems based on modular arithmetic. There are several architectures for speed up its calculations. In this paper we use carry save adder (CSA) architecture and pipeline it to increase its performance. We show that this architecture has greater performance for FPGA design than other architectures. So it is appropriate for RSA processors based on FPGAs.

1. Introduction

RSA[1] is the most widely used public-key cryptosystem. An RSA operation is a modular exponentiation like this

$$C = A^r \mod N$$

which requires repeated modular multiplications. For security reasons RSA operand size need to be 512-bits or greater in length meaning that high data throughput rates are difficult to achieve.

The Montgomery multiplication algorithm [2] is an efficient method for modular multiplication with an arbitrary modulus, particularly suitable for implementation on general-purpose computers. The method is based on an ingenious representation of the residue class modulo N, and replaces division by N operation with division by power of 2. This operation is easily accomplished on a computer or hardware design since the numbers are represented in binary form. This algorithm is the basic building block for the modular exponentiation operation which is required in the Diffie-Hellman[3] and RSA public-key cryptosystems. Various architectures attempt to improve its performance[4,5,6].

Main operation of Montgomery multiplier is modular addition that consumes time for propagating carry through long length. For this reason we can use carry save adder for avoiding long carry propagation. There are two main multiplier architectures, one using a five-to-two CSA and other a four-to-two CSA with two additional registers[7]. Each multiplier can perform a Montgomery multiplication in only k+1 and k+2 clock cycles respectively, where k is the operand bit length. In [7] shown that four-to-two CSA has greater performance than other in 1024-bits or higher.

In this paper we introduce new CSA architecture that use 3k+3 clock cycles and because it has pipelined architecture can work with higher frequency than four-to-two architecture and if define performance as throughput*(1/area), it has greater performance than other CSA architectures. By its result can find that this architecture has best result for FPGA implementations but for ASIC design it has less performance than four-to-two architecture.

The main idea for this architecture is to break one CSA module to 1/3 length and remove additional registers and instead of them add registers for pipelined three stages of CSA modules in five-to-two architecture.

This paper is organized as follows: In section 2 we described briefly the Montgomery algorithm and its modified algorithm for hardware implementation. In section 3 two main architectures for CSA presented. In section 4 we discussed new pipelined architecture for CSA. Finally we concluded this paper in section 5.

2. Montgomery multiplication

Let the modulus $n$ be a k-bit integer, i.e., $2^{k-1} \leq n < 2^k$ and let $r$ be $2^k$. The Montgomery multiplication algorithm requires that $r$ and $n$ be relatively prime, i.e., $\gcd(r,n) = \gcd(2^k,n) = 1$. This requirement is satisfied if $n$ is odd. In order to describe the Montgomery multiplication algorithm, we first define the n-residue of an integer $a$ as $a' = a \cdot r \mod n$. Given two n-residues $a'$ and $b'$, the Montgomery product is defined as the n-residue $c' = a' \cdot b' \cdot r^{-1} \mod n$.
where \( r' \) is the inverse of \( r \) modulo \( n \), i.e., it is the number with the property \( r'r \equiv 1 \) (mod \( n \)). The resulting number \( c' \) is the \( n \)-residue of the product \( c = a \cdot b \) (mod \( n \)), since
\[
c' = a' \cdot b' \cdot r' \equiv a \cdot r \cdot b \cdot r' \equiv c \cdot r \pmod{n}
\]
In order to describe the Montgomery reduction algorithm, we need an additional quantity, \( n' \), which is the integer with the property \( r \cdot r' - n \cdot n' = 1 \). The integers \( r' \) and \( n' \) can both be computed by the extended Euclidean algorithm.[8] The computation of \( \text{Mongpro}(a', b') \) is achieved as follows:

**function Mongpro(a', b')**

1. \( t := a' \cdot b' \)
2. \( u := t + (t \cdot n \mod r) \cdot n / r \)
3. if \( u = n' \) then return \( u \) else return \( u \)

The following exponentiation algorithm is one way to compute \( x := a^e \) mod \( n \) by using Montgomery multiplication algorithm. Note that \( \text{Mongpro}(x', 1) = x' \cdot r' = x \cdot r \cdot r' = x \) mod \( n \).

**function ModExp(a, e, n)**

1. \( a' := a \cdot r \mod n \)
2. \( x' := 1 \mod n \)
3. for \( i = j - 1 \) downto 0
   - \( x' := \text{Mongpro}(x', x') \)
   - if \( q_i = 1 \) then \( x' := \text{Mongpro}(x', a') \)
4. return \( x := \text{Mongpro}(x', 1) \)

The radix-2 version of Montgomery multiplication algorithm[9] which calculates the Montgomery product of \( A \) and \( B \) is summarized in the pseudo code below.

**Montgomery Multiplication \((A, B, n)\)**

- \( S[0] = 0 \);
- for \( i = 0 \) to \( k - 1 \) loop
  - \( q_i := (S[i] + A \cdot B_0) \mod 2 \);
  - \( S[i+1] := (S[i] + A \cdot B + q_i \cdot n) \div 2 \);
- end loop;
- return \( S[k] \);

This algorithm is suitable for hardware implementations because it uses modulus and division by 2 that can be implemented easily. The critical delay of this algorithm occurs during the calculation of the \( S \) values given by the three input addition

\[
S[i+1] = (S[i] + A \cdot B + q_i \cdot n)
\]

The main contributing factor to this delay is the carry propagation resulting from the very large operand additions.

### 3. CSA architectures

#### 3.1. Five-to-two CSA Architecture

The main calculations of CSA are

- \( \text{SUM} = X_1 \oplus X_2 \oplus X_3 \)
- \( \text{CARRY} = (X_1 \text{ and } X_2) \text{ or } (X_1 \text{ and } X_3) \text{ or } (X_2 \text{ and } X_3) \)

The sum of bit vectors \( \text{SUM} \) and \( \text{CARRY} \) is equal to the sum of the three input bit vectors \( X_1, X_2, \) and \( X_3 \).

An outline diagram of five-to-two CSA[7] operation is shown in Figure 1. Montgomery algorithm can be changed for this architecture as follow.

**Five-to-two CSA Montgomery Multiplication**

\((A_1, A_2, B_1, B_2, n)\)

1. \( S[0] = 0; \)
2. \( S[2] = 0; \)
3. for \( i \) in \( 0 \) to \( k - 1 \) loop
   - \( q_i := (S[i] + S[2][i] + (A_i \cdot (B_1 + B_2)) \mod 2; \)
   - \( S[i+1], S[2][i+1] = \text{CSR}(S[i] + S[2][i] + A_i \cdot (B_1 + B_2) + q_i \cdot n) \div 2; \)
- end loop;
- return \( S[1][k], S[2][k]; \)

![Figure 1. Block diagram of five-to-two CSA](image)

Figure 1. Block diagram of five-to-two CSA

Note that the input operand \( A \) and \( B \) and the output product \( S \) are now represented in carry save format as \( A_1 \) and \( A_2 \), \( B_1 \) and \( B_2 \), and \( S_1 \) and \( S_2 \) respectively. CSR stands for carry save representation.

For determination of \( A_i \) we can use BRFA as Figure 2. So we don't need to compute full addition of the input operands \( A_1 \) and \( A_2 \).

![Figure 2. Barrel register full adder diagram](image)

Figure 2. Barrel register full adder diagram

For this architecture an extra one clock cycle is required to reset the signal \( S[1][0] \) and \( S[2][0] \) to zero at the beginning of each multiplication. Thus this algorithm can be executed in only \( k + 1 \) clock cycles.

#### 3.2. Four-to-two CSA architecture

This architecture approach is based on the use of a four-to-two rather than a five-to-two CSA resulting in a saving of a full level of carry save logic as shown in Figure 3[7].
Algorithm for this approach is as below.

Four-to-two CSA Montgomery Multiplication

\[ D1, D2 = \text{CSR}(B1+B2+n+0) \]

\[ S1[0] = 0 \]

\[ S2[0] = 0 \]

for \( i \) in \( 0 \) to \( k-1 \) loop

\[ q_i = (S1[i]+S2[i]+(A_i+B1+B2)) \mod 2 \]

if \( A_i = 0 \) and \( q_i = 0 \) then

\[ S1[i+1], S2[i+1] = \text{CSR}(S1[i]+S2[i]+0) \div 2 \]

elsif \( A_i = 1 \) and \( q_i = 0 \) then

\[ S1[i+1], S2[i+1] = \text{CSR}(S1[i]+S2[i]+B1+B2) \div 2 \]

elsif \( A_i = 0 \) and \( q_i = 1 \) then

\[ S1[i+1], S2[i+1] = \text{CSR}(S1[i]+S2[i]+n) \div 2 \]

else

\[ S1[i+1], S2[i+1] = \text{CSR}(S1[i]+S2[i]+D1) \div 2 \]

end if:

end loop;

return \( S1[k], S2[k] \).

The main computation to be performed here is the four-to-two carry save addition

\[ S1[i+1], S2[i+1] = \text{CSR}(S1[i]+S2[i]+y+z) \]

The value of \( y \) and \( z \) depends on the value of both the \( A_i \) and \( q_i \). Indeed, the determination of the state of both these signals simultaneously, represents extra control logic in the form of a more complex multiplexer than is required for five-to-two CSA case. This needs to be taken into account in determining the critical delay. However as shown in [7] this delay is still less than that implied by five-to-two architecture. This decrease in the critical path delay comes at the expense of two extra registers, \( D1, D2 \). The calculation of \( D1 \) and \( D2 \) at beginning of each multiplication means than an extra clock cycle is required to complete a full Montgomery multiplication.

In [7] shown that this architecture has greater performance than five-to-two for 1024-bits length and higher.

4. Pipelined CSA architecture

In two previous designs it calculate full length of data in one clock cycle so we must use 1024 bits CSA for 1024 bits operands or 2048 bits CSA for 2048 bits operands. If use some registers to hold intermediate result, can use smaller CSA blocks and also can pipeline it to increase its speed. For this reasons in our architecture we break CSA to 1/3 of its original length and instead of using two extra registers in four-to-two registers, two extra registers with 1/3 length are used for pipelining three CSA stages in five-to-two architecture. Block diagram of this architecture is shown in Figure 4.

Note that for holding some value such as \( A_i \) or \( q_i \) for stages of pipelines some extra registers are used that hold proper value for that cycle.

For this architecture the algorithm for internal loop must assume that it is done in pipeline, for simplicity didn’t show each stage.

Pipelined CSA Montgomery Multiplication

\[ (A1, A2, B1, B2, n) \]

\[ S1[0] = 0 \]

\[ S2[0] = 0 \]

\[ A_i = 0 \]

\[ q_i = 0 \]

for \( i \) in \( 0 \) to \( k-1 \) loop

Compute \( A_i \) and hold for three cycles.

for \( p \) in \( 0 \) to \( 2 \) loop ‘3 pipelined stage

\[ S1[i+1], S2[i+1] = \text{CSR}(S1[i]+S2[i]+A_i+B1+B2) \div 2 \]

end loop;

end loop;

end loop.

In this architecture, like other architecture, one extra clock cycle is needed for reset but for each complete addition three cycles are used so for full multiplication 3k+3 cycles are needed. So one may think that its performance is less than four-to-two architecture but whereas it has more frequency and has less area we show with experience that this architecture has greater
performance than other architectures. For this reason we implemented four-to-two and our architectures by VHDL and synthesized it by Leonardo Spectrum 2002 tool for Xilinx FPGAs. Table 1 and 2 provides performance results for these implementations.

<table>
<thead>
<tr>
<th>Table 1. Results for four-to-two architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Device</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>XC2V1000</td>
</tr>
<tr>
<td>XC2V3000</td>
</tr>
<tr>
<td>XC2V6000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. Results for pipelined CSA architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Device</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>XC2V1000</td>
</tr>
<tr>
<td>XC2V3000</td>
</tr>
<tr>
<td>XC2V6000</td>
</tr>
</tbody>
</table>

By results can see there are decrease in throughput rate for pipelined CSA implementations but if we don't ignore decrease in area and define performance as throughput*1/area, one can find there are increase in performance, for instance for 512 bits implementations there is 2398.25 increase in performance. So in designs that are restricted in area and want to have proper throughput, can use pipelined CSA architecture instead of four-to-two.

For comparing with ASIC implementation, also synthesized this implementations for 0.6 CMOS by Leonardo Spectrum 2002 and these results have shown in table 3 and 4.

<table>
<thead>
<tr>
<th>Table 3. Results for four-to-two (ASIC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Length</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>512</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>1536</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4. Results for pipelined CSA (ASIC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Length</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>512</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>1536</td>
</tr>
</tbody>
</table>

The results show there is no increase in performance for ASIC implementations of pipelined CSA architectures.

5. Conclusion

In this paper new architecture by CSA adder is introduced that has greater performance than other architectures. For this approach pipelined stages are used and for this reason could reach to higher frequency and because of breaking each CSA module to 1/3 of its original length it used less area. By results one can find that this architecture is useful for FPGA implementations and for ASIC design, four-to-two architecture is better.

We can categorized benefits of pipelined CSA as :

1) Used less area with proper throughput.
2) Can be scaled for smaller area by breaking CSA module to smaller module, for example 1/3 of its original length.
3) High frequency that is useful when use multiplier module in a big design because it has less waste of time.

6. References