Journal of Information Assurance and Security.
ISSN 1554-1010 Volume 17 (2022) pp. 113-121

Verilog HDL Implementation for an RSA Cryptography using Shift-Sub Modular Multiplication Algorithm

Yamin Li[1] **and Wanming Chu**[2]

[1]Department of Computer Science, Hosei University,
Tokyo 184-8584 Japan
yamin@hosei.ac.jp

[2]Division of Information Systems, University of Aizu,
Aizu-Wakamatsu 965-8580 Japan
w-chu@u-aizu.ac.jp

Abstract: **RSA public-key cryptography requires modular exponentiation and modular multiplication on large numbers. Montgomery Modular Multiplication is a fast method for performing modular multiplication. The modular exponentiation can be calculated by repeatedly calling Montgomery Modular Multiplication. Transformations to the Montgomery Domain are required before the calculations, and a transformation back to the normal domain is also required to get the final result. The domain transformations require a value that is calculated by costly modular arithmetic. Many hardware RSA implementations use precomputed values for such domain transformations. As a result, the flexibility to use different public keys is lost. This paper introduces a Shift-Sub Modular Multiplication (SSMM) algorithm for calculating such values in fields. The algorithm does not require modular arithmetic and precomputed values. Instead, it uses shift and addition/subtraction calculations. The SSMM algorithm can also be used directly for RSA public-key cryptography. We give the source codes of the hardware implementation of RSA public-key cryptography using SSMM in Verilog HDL and compare the cost and performance to that of RSA public-key cryptography implementation using Montgomery Modular Multiplication. The results show that the performance of the two implementations is about the same, but the implementation using SSMM uses less hardware resource (55% to 59% adaptive logic modules and 69% to 85% flip-flops) because it does not require modular arithmetic or domain transformations.**

Keywords: RSA public-key cryptography, Montgomery modular multiplication, hardware security circuit, clock frequency, hardware resource.

I. Introduction

RSA public-key cryptography was published by Rivest, Shamir, and Adleman [11] in 1978 and is widely used for secure data transmission. It is based on the use of the product of two very large prime numbers (greater than 10^{100}), relying on the fact that the determination of the prime factors of such large numbers is so computationally difficult as to be effectively impossible to compute [3].

RSA public-key cryptography works as follows. To find an encryption key e and a decryption key d, choose two large prime numbers p and q, and form $m = pq$ and $z = (p - 1)(q - 1)$. Choose an encryption key e such that e and z are relatively prime. To find a decryption key d, solve the equation $de = 1 \bmod z$. That is, de is the smallest element in the series $kz + 1$ divisible by e for $k \in \mathbb{N}$. The function for encrypting a single block of plaintext b with $b < m$ is

$$r = b^e \bmod m$$

where the result r is the ciphertext. The function for decrypting r is

$$s = r^d \bmod m$$

where the result s will be equal to b, the original plaintext. The encryption function, encryption key e, and m are publicly opened but the decryption key d is kept privately.

The correctness of the RSA algorithm is shown below. $r^d \bmod m = b^{ed} \bmod m = b^{k(p-1)(q-1)+1} \bmod (pq) = b \times b^{k(p-1)(q-1)} \bmod (pq) = b \times 1^k$ by applying the Fermat's Little Theorem and Chinese Remainder Theorem. That is,

$$(b^e \bmod m)^d \bmod m = b$$

We give an example as below where p and q have 64 bits (m has 128 bits).

$$\begin{aligned}
p &= 16856020000513437973;\\
q &= 17274135032339836727;\\
m &= 291173165596690131543379395216261834371;\\
z &= 291173165596690131509249240183408559672;\\
e &= 78624383815806095082831236375207684303;\\
d &= 232543530691965449749356023879307323711;\\
b &= 179441695220040973036856247560209845703;\\
r &= 212957456342734650649396939600336433714;\\
s &= 179441695220040973036856247560209845703 = b.
\end{aligned}$$

The numbers of e, d, m, b, and r in this example will be used in the testbench for the simulations of the proposed algorithms and their Verilog HDL implementation codes which will be given later in this paper.

RSA encryption $r = b^e \bmod m$ and decryption $s = r^d \bmod m$ can be performed with *left-to-right binary exponentiation* or *right-to-left binary exponentiation* which requires repeated *modular multiplications*.

Montgomery Modular Multiplication [9] is a fast method for performing modular multiplication. The algorithm does not require the trial division during the calculation but it requires transforming the original variables to *Montgomery Domain* like $\widetilde{x} = xR \bmod m$, where x is the original variable, $R = 2^n$, m is odd and has n bits, and $\widetilde{x}$ is a new representation in Montgomery Domain. We can use *Montgomery Modular Reduction* $\text{MMRed}(z) = zR^{-1} \bmod m$ to perform such transformations to Montgomery Domain if we have a $q = R^2 \bmod m$: $\widetilde{x} = \text{MMRed}(xq) = xR^2R^{-1} \bmod m = xR \bmod m$. That is, if we had the q, the domain transformations will not require the modular calculations.

Many RSA hardware implementations use precomputed value $q = R^2 \bmod m$ for fixed R and m [4, 6, 12]. A lookup table can be used to store multiple precomputed values [8, 2]. Using precomputations can speed up the calculations but reduces the flexibility of using different moduli m. The main contributions of this paper are 1) to introduce a Shift-Sub Modular Multiplication (SSMM) algorithm to calculate $R^2 \bmod m$ in fields without using divisions; 2) to show how to use the SSMM algorithm directly for RSA public-key cryptography; 3) to implement the SSMM algorithm and RSA cryptography using SSMM in Verilog HDL; and 4) to compare the hardware cost and performance of RSA cryptography using SSMM to that of RSA cryptography using Montgomery Modular Multiplication. We expect that RSA cryptography using SSMM can be implemented with less hardware resource and can be performed as well as RSA cryptography using Montgomery Modular Multiplication/Reduction.

This paper is an extension of the paper published at IAS2021 [7]. We added the hardware implementation details for the proposed algorithms. The rest of the paper is organized as follows. Section II reviews Montgomery Modular Multiplication/Reduction algorithms. Section III introduces the SSMM algorithm and RSA cryptography using SSMM and shows the hardware implementation details. Section IV gives hardware cost/performance comparisons for RSA cryptography implementations. And Section V concludes the paper.

II. Montgomery Modular Algorithms

This section reviews Montgomery Modular Reduction and Montgomery Modular Multiplication algorithms.

A. Montgomery Modular Reduction Algorithm

Montgomery modular algorithms [9, 10] use a special representation for variables:

$$\widetilde{x} = xR \bmod m$$

where x is the original variable, R is an auxiliary value, m is odd and coprime to R, and $\widetilde{x}$ is a new representation in *Montgomery Domain*.

For a single multiplication, $w = xy \bmod m$, we want to have a new representation for w in Montgomery Domain:

$$\widetilde{w} = xyR \bmod m$$

If we perform a normal multiplication on $\widetilde{x}$ and $\widetilde{y}$, we get

$$z = \widetilde{x}\widetilde{y} = xyR^2 \bmod m^2$$

which is not a representation in Montgomery Domain. *Montgomery Modular Reduction* $\text{MMRed}(z)$ translates z to a representation in Montgomery Domain:

$$\text{MMRed}(z) = zR^{-1} \bmod m$$

such that $\text{MMRed}(z) = xyR^2R^{-1} \bmod m = xyR \bmod m$, which is a representation in Montgomery Domain.

By selecting a suitable R, we can perform Montgomery Modular Reduction without using divisions. If m is an n-bit number (then z has $2n$ bits), we can use $R = 2^n$, such that the division can be done with shift. Montgomery Reduction with precomputed $m' = -m^{-1} \bmod R$ is formally given in **Algo 0**.

Algo 0. MMRedP(z, m) Montgomery Reduction with precomputed m'

---

inputs: $z = \sum_{i=0}^{2n-1} z_i 2^i$, $R = 2^n$, $m < R$ with m odd,
    $0 \le z < mR$, and precomputed m' such that
    $m' = -m^{-1} \bmod R$
output: $zR^{-1} \bmod m$
begin
1 $U \leftarrow zm' \bmod R$
2 $t \leftarrow (z + Um)/R$
3 **if** $t \ge m$
4      $t \leftarrow t - m$
5 **return** t
end

Algo 0 MMRedP(z, m) generates $zR^{-1} \bmod m$. The reason is as follows. $U = zm' \bmod R$ is an integer and $U < R$. Thus $(z + Um) \bmod m = z \bmod m$. Then $(z + Um)/R \bmod m = zR^{-1} \bmod m$ if $z + Um$ is divisible by R. Because $m' = -m^{-1} \bmod R$, $m'm = -1 + jR$ for some integers j. $U = zm' + kR$ for some integers k. $z + Um = z + zm'm + kmR = z + z(-1 + jR) + kmR = zjR + kmR = (zj + km)R$. Because z, j, k, and m are integers, $z + Um = (zj + km)R$ is divisible by R. Because $z < mR$ and $U < R$, $t = (z+Um)/R < (mR+Rm)/R = 2m$, the lines 3 and 4 in Algo 0 are needed.

The parameter m' needs to be precomputed once for fixed R and m [5]. The calculation can be performed using the extended Euclidean algorithm. For example, for $n = 128$, $m = 291173165596690131543379395216261834371$, we have $m' = -133419654858893623771608150101834274859$. Because $U = zm' \bmod R = zm' \ \& \ ((1 \ll n) - 1)$, for $z = 145791500050062650629481369540128853000$,

$$t = (z + Um) \gg n$$

we get $t = 179441695220040973036856247560209845703$ which is $zR^{-1} \bmod m$. If we select a special m such that $m^2 = 1 \bmod R$, then $m = m^{-1} \bmod R$. Thus $m' = -m^{-1} \bmod R = -m \bmod R$. That is, for such special moduli m, the requirement of the precomputation can be eliminated [1].

Note that in RSA encryption and decryption, a huge amount of modular multiplications can be performed in Montgomery Domain. After getting $xyR \bmod m$, we can transform it from Montgomery Domain back to the normal domain by applying Montgomery Modular Reduction once again:

$$\text{MMRed}(xyR \bmod m) = xyRR^{-1} \bmod m = xy \bmod m$$

We can use Montgomery Modular Reduction also for transforming the original variables from the normal domain to Montgomery Domain:

$$q = R^2 \bmod m$$

$$\widetilde{x} = \text{MMRed}(xq) = xR^2R^{-1} \bmod m = xR \bmod m$$

$$\widetilde{y} = \text{MMRed}(yq) = yR^2R^{-1} \bmod m = yR \bmod m$$

Algo 0 is not an efficient way to realize Montgomery arithmetic [10]. In practice, a bit-oriented version is often used. Now, consider how to implement Montgomery Modular Reduction $\text{MMRed}(z) = zR^{-1} \bmod m$ in *bit level*. Because we can add km to z for $k \in \mathbb{Z}$ and

$$R^{-1} = \frac{1}{R} = \frac{1}{2^n} = \prod_{i=0}^{n-1} \frac{1}{2}$$

a bit-level Montgomery Modular Reduction can be implemented with an iteration loop for n and dividing z by 2 in each iteration. In order to make z divisible by 2, we can add an m to z if z is odd (m is odd, then the sum will be even, divisible by 2). The Montgomery Modular Reduction algorithm in bit level is formally given in **Algo 1** where z_0 is the least significant bit of z. We can see that the computational complexity of the algorithm is $O(n)$, where n is the bit length of m. Note that Algo 1 itself does not require any precomputation.

Algo 1. MMRed(z, m) Montgomery Modular Reduction

inputs: $z = \sum_{i=0}^{2n-1} z_i 2^i$, $R = 2^n$, $m < R$ with m odd, and $0 \le z < mR$
output: $zR^{-1} \bmod m$
begin
1 $p \leftarrow z$ /* product */
2 **for** $i = 0$ **to** $n - 1$
3 $\quad p \leftarrow p + p_0 m$ /* make p even */
4 $\quad p \leftarrow p \gg 1$ /* $p/2$: reduction */
5 **if** $p \ge m$
6 $\quad p \leftarrow p - m$
7 **return** p
end

After finishing the "for" loop, we get $z < 2m$. The reason is as follows. Consider the case of maximum z. That is, in every iteration for n, we add m to z. Because $z < mR = m2^n$, $z/2^n < m$, after finishing the "for" loop, we have

$$\frac{z}{2^n} + \frac{m}{2^n} + \frac{m}{2^{n-1}} + \cdots + \frac{m}{2^2} + \frac{m}{2^1} = \frac{z}{2^n} + \frac{(2^n-1)m}{2^n} < 2m$$

Therefore, the lines 4 and 5 in Algo 1 are needed. RSA encryption calculates

$$r = b^e \bmod m$$

where b is a plaintext variable and encrypted with a public-key $\{e, m\}$ and r is the ciphertext result. Similarly, RSA decryption calculates

$$s = r^d \bmod m$$

where r is a ciphertext variable and decrypted with a private key $\{d, m\}$ and $s = b$ is the plaintext result.

Consider the calculation of RSA encryption $r = b^e \bmod m$. Suppose e has n bits, that is

$$e = e_{n-1} \ldots e_1 e_0 = \sum_{i=0}^{n-1} e_i 2^i$$

The exponentiation calculation can be performed with an iteration loop for n, dividing e by 2 and multiplying b by b (*squaring*) in each iteration. If e_0 is a 1, r will be multiplied by b (*multiply*). Ignoring the modulation, the following example calculates 2^5. Here, we have $b = 2$ and $e = 5 = 101_2 = e_2e_1e_0$ for $n = 3$. Let $r = 1$. For $i = 0$, $e_0 = 1$, then $r \leftarrow rb = 1 \times 2 = 2$. After that, $e \leftarrow e/2 = 10_2 = e_1e_0$ and $b \leftarrow b^2 = 4$. For $i = 1$, $e_0 = 0$, $e \leftarrow e/2 = 1_2 = e_0$ and $b \leftarrow b^2 = 16$. For $i = 2$, $e_0 = 1$, then $r \leftarrow rb = 2 \times 16 = 32$. After that, $e \leftarrow e/2 = 0$ and $b \leftarrow b^2 = 256$. The final result is $r = 32$ and e becomes 0.

Both the squaring and multiply need modulation. We can use Montgomery Modular Reduction for these calculations. As mentioned before, the initial $r = 1$ and b must be transformed to Montgomery Domain once. After doing a huge number of multiplies and Montgomery Modular Reductions, r must be transformed back to the normal domain. The algorithm of the modular exponentiation using Montgomery Modular Reduction is formally given in **Algo 2**. We used right-to-left binary exponentiation algorithm here.

We have implemented Algo 1 and Algo 2 in Verilog HDL. Figure 1 shows the simulation waveform for RSA encryption $r = b^e \bmod m$ using Montgomery Modular Reduction, where $n = 128$ and r is the ciphertext of the plaintext b:

$$\begin{aligned} b &= 179441695220040973036856247560209845703; \\ e &= 78624383815806095082831236375207684303; \\ m &= 291173165596690131543379395216261834371; \\ r &= 212957456342734650649396939600336433714. \end{aligned}$$

Figure 2 shows the simulation waveform for RSA decryption $r = b^e \bmod m$ using the Montgomery Modular Reduction, for $n = 128$. Note that b is the same as r of Figure 1 and a private decryption key d is used as e in the simulation:

$$\begin{aligned} b &= 212957456342734650649396939600336433714; \\ e &= 232543530691965449749356023879307323711; \\ m &= 291173165596690131543379395216261834371; \\ r &= 179441695220040973036856247560209845703. \end{aligned}$$

Algo 2. MExpRed(b, e, m) Modular Exponentiation using MMRed()

inputs: $b = \sum_{i=0}^{n-1} b_i 2^i$, $e = \sum_{i=0}^{n-1} e_i 2^i$, $R = 2^n$, $m < R$ with m odd
output: $b^e \bmod m$
begin
$q \leftarrow R^2 \bmod m$
$r \leftarrow \mathrm{MMRed}(q, m)$ /* 1 to Montgomery Domain */
$s \leftarrow \mathrm{MMRed}(bq, m)$ /* b to Montgomery Domain */
$t \leftarrow e$
while $t > 0$
  if $t_0 = 1$
    $r \leftarrow \mathrm{MMRed}(rs, m)$ /* multiply */
  $t \leftarrow t \gg 1$
  $s \leftarrow \mathrm{MMRed}(s^2, m)$ /* squaring */
$r \leftarrow \mathrm{MMRed}(r, m)$ /* r to normal domain */
return r
end

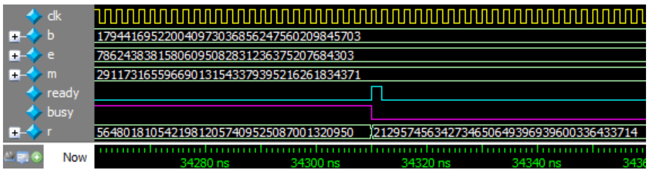

Figure. 1: Montgomery Modular Reduction RSA encryption

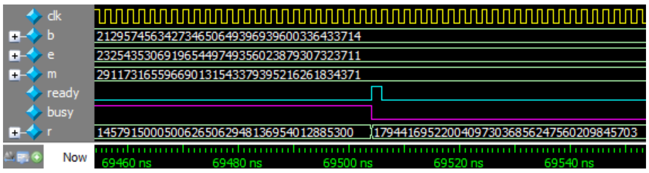

Figure. 2: Montgomery Modular Reduction RSA decryption

We can see that r is the same as the original plaintext, the input b in Figure 1.

Transforming variables to Montgomery Domain requires to calculate $q = R^2 \bmod m$ which has a modular calculation. Note that it just is a one-time calculation for a modular exponentiation. In software implementations, there is no any problem for the calculation. But in hardware implementations, we have to design a circuit to perform such a calculation, or use a precomputed q for the fixed R and m [10]. In this paper, we use a Shift-Sub Modular Multiplication (SSMM) algorithm to calculate $q = R^2 \bmod m$ which we will describe later.

B. Montgomery Modular Multiplication Algorithm

The Montgomery Modular Reduction algorithm does a reduction on a product of multiplicand and multiplier. The cost of using it is high because we have to use big multipliers to calculate rb (multiply) and b^2 (squaring). The Montgomery Modular Multiplication algorithm calculates the partial product by shift and addition during the reductions. **Algo 3** formally gives a bit-level Montgomery Modular Multiplication algorithm. Note that there is no shifting the multiplicand to the left by one bit, because we shift the partial product to the right by one bit at each iteration. We can see that the computational complexity of the algorithm is $O(n)$, the same as that of Algo 1, where n is the bit length of m.

Algo 3. MMMul(a, b, m) Montgomery Modular Multiplication

inputs: $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, $R = 2^n$, $a, b < m < R$, m: odd
output: $abR^{-1} \bmod m$
begin
$p \leftarrow 0$ /* product */
for $i = 0$ **to** $n - 1$
  $p \leftarrow p + b_i a$ /* add multiplicand a to p if $b_i = 1$ */
  $p \leftarrow p + p_0 m$ /* make p even */
  $p \leftarrow p \gg 1$ /* $p/2$: reduction */
if $p \geq m$
  $p \leftarrow p - m$
return p
end

The algorithm of the modular exponentiation using bit-level Montgomery Modular Multiplication is formally given in **Algo 4**. We also used right-to-left binary exponentiation algorithm. Instead of passing the product to MMRed() in Algo 2, the multiplicand and multiplier are passed to MMMul() here.

Algo 4. MExpMul(b, e, m) Modular Exponentiation using MMMul()

inputs: $b = \sum_{i=0}^{n-1} b_i 2^i$, $e = \sum_{i=0}^{n-1} e_i 2^i$, $R = 2^n$, $m < R$ with m odd
output: $b^e \bmod m$
begin
$q \leftarrow R^2 \bmod m$
$r \leftarrow \mathrm{MMMul}(1, q, m)$ /* 1 to Montgomery Domain */
$s \leftarrow \mathrm{MMMul}(b, q, m)$ /* b to Montgomery Domain */
$t \leftarrow e$
while $t > 0$
  if $t_0 = 1$
    $r \leftarrow \mathrm{MMMul}(r, s, m)$ /* multiply */
  $t \leftarrow t \gg 1$
  $s \leftarrow \mathrm{MMMul}(s, s, m)$ /* squaring */
$r \leftarrow \mathrm{MMMul}(1, r, m)$ /* r to normal domain */
return r
end

We have also implemented Algo 3 and Algo 4 in Verilog HDL. The simulation waveforms of the Montgomery Modular Multiplication RSA encryption and decryption are the same as that of the Montgomery Modular Reduction RSA encryption and decryption, shown as in Figure 1 and Figure 2, respectively.

Montgomery Modular Multiplication can be implemented using the carry-save adder (CSA) [13, 14, 15]. At each iteration, the carry and sum are stored in separate registers, eliminating carry propagation. However, extra clock cycles are required to convert the final carry-save modular product into binary form for the modular exponentiation.

III. Shift-Sub Modular Multiplication Algorithm

The Montgomery Modular Reduction MMRed() and Multiplication MMMul() perform calculations in Montgomery Domain. As mentioned before, we must get the value of $q = R^2 \bmod m$ for the domain transformations. Some hardware implementations use a precomputed q for fixed R and m. Such implementations reduce flexibility for changing R and m. This section introduces a Shift-Sub Modular Multiplication (SSMM) algorithm SSMMul(a, b, m) to calculate $z = ab \bmod m$ that uses only addition, subtraction, and shift calculations for $a, b < m$.

For $i, j \in \mathbb{Z}$ and $x, y, m \in \mathbb{N}$, because $(x + im)(y + jm) \bmod m = (xy + xjm + imy + imjm) \bmod m = xy \bmod m$, we have

$$q = R^2 \bmod m = (R - m)(R - m) \bmod m$$

$$= \text{SSMMul}(R - m, R - m, m)$$

where m is an n-bit odd number and $R = 2^n$.

It is not difficult to prove that $R - m < m$: n-bit m means $m = m_{n-1} \times 2^{n-1} + \ldots + m_1 \times 2^1 + m_0 \times 2^0$ and $m_{n-1} = 1$ (n bits). Because m is an odd number, we have $m_0 = 1$ (odd). Then $2m > 2^n = R$, $2m - m > R - m$, $m > R - m$, that is $R - m < m$, satisfying $a, b < m$ for SSMMul(a, b, m).

The SSMM algorithm SSMMul(a, b, m) calculates $z = ab \bmod m$, where $a, b < m < R$ with $R = 2^n$ and m is an n-bit odd number. At the beginning, let product $z = 0$. We check the least significant bit of multiplier b. If it is a 1, we add multiplicand a to product z. And then we shift a to the left and b to the right by one bit, respectively.

Because $a < m$, $a \leftarrow 2a < 2m$. Similarly, $z = 0$ at the beginning or $z = a$ after the first adding a to z, then we have $z \leftarrow z + a \leq 2a < 2m$. After the addition and shift, we perform the following operations: If $z > m$, $z \leftarrow z - m$; if $a > m$, $a \leftarrow a - m$. Such operations ensure $z < m$ and $a < m$.

The correctness of the SSMM algorithm is based on the following facts: Because we are calculating $z = ab \bmod m$, we can add/subtract multiples of m to/from z. And because $(a + im)b \bmod m = (ab + imb) \bmod m = ab \bmod m$ for $i \in \mathbb{Z}$, we can also add/subtract multiples of m to/from a.

The SSMM algorithm is given formally in **Algo 5**. We can see that the computational complexity of the algorithm is $O(n)$, the same as that of Algo 1 and Algo 3, where n is the bit length of m.

Below shows the 128-bit Verilog HDL code "modu_mult_128.v" that implements the Shift-Sub Modular Multiplication algorithm (Algo 5).

```
module modu_mult_128(x,y,m,p,clk,strobe,rst_n,
                     ready,busy);
parameter NLEN = 128;
input clk, strobe, rst_n;
input [NLEN-1:0] x;
input [NLEN-1:0] y;
input [NLEN-1:0] m;
output reg ready, busy;
output [NLEN-1:0] p; // p = (x * y) % m
reg  [NLEN+1:0] x_reg;
reg  [NLEN-1:0] y_reg;
reg  [NLEN+1:0] m_reg;
reg  [NLEN+1:0] p_reg;
wire [NLEN+1:0] x1, x2;
wire [NLEN+1:0] p1, p2, p3;
assign p  = p3[NLEN-1:0];
assign p1 = y_reg[0] ? (p_reg + x_reg) : p_reg;
assign p2 = p1 - m_reg;
assign p3 = p2[NLEN+1] ? p1 : p2;
assign x1 = {x_reg[NLEN:0],1'b0} - m_reg;
assign x2 = x1[NLEN+1] ? {x_reg[NLEN:0],1'b0} : x1;
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    x_reg <= 0;
    y_reg <= 0;
    m_reg <= 0;
    p_reg <= 0;
    busy  <= 0;
    ready <= 0;
  end else begin
    ready <= 0;
    if (strobe) begin
      x_reg <= {2'b00,x};
      y_reg <= y;
      m_reg <= {2'b00,m};
      p_reg <= 0;
      busy  <= 1;
    end else begin
      if (busy) begin
        if (y_reg == 0) begin
          ready <= 1;
          busy  <= 0;
        end else begin
          x_reg <= x2;
          y_reg <= {1'b0,y_reg[NLEN-1:1]};
          p_reg <= p3;
        end
      end
    end
  end
end
endmodule
```

Algo 5. SSMMul($\boldsymbol{a, b, m}$) Shift-Sub Modular Multiplication

inputs: $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, $R = 2^n$,
$a, b < m < R$, m: odd
output: $ab \bmod m$
begin
$p \leftarrow 0$ /* product */
$c \leftarrow a$ /* multiplicand */
for $i = 0$ **to** $n - 1$
  $p \leftarrow p + b_i c$ /* add multiplicand c to p if $b_i = 1$ */
  if $p \geq m$
    $p \leftarrow p - m$ /* subtract m from p */
  $c \leftarrow c \ll 1$
  if $c \geq m$
    $c \leftarrow c - m$ /* subtract m from c */
return p
end

We developed the SSMM algorithm SSMMul() to calculate $q = R^2 \bmod m$ for Montgomery Modular Reduction and Multiplication domain transformations. After that, we found that it can be used directly in the exponentiation modulation calculation for RSA cryptography.

The exponentiation modulation using SSMMul() for RSA cryptography is illustrated in Figure 3(b). We use SSMMul() to perform multiply and squaring inside the "while" loop.

(a)	(b)
$q \leftarrow R^2 \bmod m$ $r \leftarrow \mathrm{MMMul}(1, q, m)$ $s \leftarrow \mathrm{MMMul}(b, q, m)$	$r \leftarrow 1$ $s \leftarrow b$

While loop on e
if ()
$r \leftarrow$ MMMul() // multiply
$s \leftarrow$ MMMul() // squaring
	While loop on e
	if ()
	$r \leftarrow$ SSMMul() // multiply
	$s \leftarrow$ SSMMul() // squaring

$r \leftarrow \mathrm{MMMul}(1, r, m)$ Return r	Return r
(a) Exponentiation using MMMul()	(b) Exponentiation using SSMMul()

Figure. 3: Algorithm comparison

Note that there is no any domain transformations. Figure 3(a) shows RSA cryptography using MMMul() (Algo 4) for comparison.

Algo 6 formally gives the modular exponentiation using SSMMul(). As discussed before, the computational complexity of the SSMMul() is the same as that of MMMul(). We expect that RSA cryptography using SSMMul() can be performed as well as RSA cryptography using MMMul(). However, RSA cryptography using SSMMul() does not require domain transformations and hence there is no need to calculate $q = R^2 \bmod m$, so we expect that it saves hardware resource.

Algo 6. SSExpMul(b, e, m) Modular Exponentiation using SSMMul()

inputs: $b = \sum_{i=0}^{n-1} b_i 2^i$, $e = \sum_{i=0}^{n-1} e_i 2^i$, $R = 2^n$, $m < R$ with m odd
output: $b^e \bmod m$
begin
$r \leftarrow 1$
$s \leftarrow b$
$t \leftarrow e$
while $t > 0$
  if $t_0 = 1$
    $r \leftarrow \mathrm{SSMMul}(r, s, m)$ /* multiply */
  $t \leftarrow t \gg 1$
  $s \leftarrow \mathrm{SSMMul}(s, s, m)$ /* squaring */
return r
end

In Algo 2, Algo 4, and Algo 6, the complexity of the "while" loop itself is $O(n)$, and inside the loop, it calls a reduction or multiplication whose complexity is also $O(n)$. Therefore, the complexity of Algo 2, Algo 4, and Algo 6 is $O(n^2)$ where n is the number of bits of m.

Below shows the 128-bit Verilog HDL code "modu_expo_128.v" that implements the modular exponentiation algorithm (Algo 6). This module invokes two "modu_mult_128" modules for parallel computations of multiply and squaring.

```
module modu_expo_128(b,e,m,r,clk,strobe,rst_n,
                     ready,busy);
parameter NLEN = 128;
input clk, strobe, rst_n;
input [NLEN-1:0] b;
input [NLEN-1:0] e;
input [NLEN-1:0] m;
output reg ready, busy;
output [NLEN-1:0] r; // r = bˆe % m
wire            ready_r;
wire            ready_b;
wire            busy_r;
wire            busy_b;
wire [NLEN-1:0] p_r;
wire [NLEN-1:0] p_b;
reg             strobe_r;
reg             strobe_b;
reg             state;
reg  [NLEN-1:0] b_reg;
reg  [NLEN-1:0] e_reg;
reg  [NLEN-1:0] m_reg;
reg  [NLEN-1:0] r_reg;
assign r = r_reg;
modu_mult_128 res (r_reg,b_reg,m_reg,p_r,clk,strobe_r,
                   rst_n,ready_r,busy_r); // multiply
modu_mult_128 bas (b_reg,b_reg,m_reg,p_b,clk,strobe_b,
                   rst_n,ready_b,busy_b); // squaring
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    b_reg <= 0;
    e_reg <= 0;
    m_reg <= 0;
    r_reg <= 0;
    busy  <= 0;
    ready <= 0;
    state <= 0;
  end else begin
    ready    <= 0;
    strobe_r <= 0;
    strobe_b <= 0;
    if (strobe) begin
      b_reg <= b;
      e_reg <= e;
      m_reg <= m;
      r_reg <= 1;
      busy  <= 1;
      ready <= 0;
      state <= 0;
    end else begin
      if (busy) begin
        if ((e_reg == 1) && ready_r) begin
          ready <= 1;
          busy  <= 0;
          r_reg <= p_r;
        end else begin
          case (state)
          0: // multiply
          begin
            if ((~busy_r) && (~ready_r)) begin
              if (e_reg[0]) begin
                if (~strobe_r)
                   strobe_r <= 1;
                else strobe_r <= 0;
              end
            end
            state <= 1;
          end
          1: // squaring
          begin
            if ((~busy_b) && (~ready_b)) begin
              if (~strobe_b)
                 strobe_b <= 1;
              else strobe_b <= 0;
            end
            if (ready_b) begin
              b_reg <= p_b;
              e_reg <={1'b0,e_reg[NLEN-1:1]};
              state <= 0;
            end
```

```
            if (ready_r) begin
              r_reg <= p_r;
            end
          end
          endcase
        end
      end
    end
  end
end
endmodule
```

For verifying the correctness of the proposed algorithms and their Verilog HDL codes, we prepared the testbench code "modu_expo_128_tb.v" as below.

```
`timescale 1ns/1ns
module modu_expo_128_tb;
parameter NLEN  = 128;
reg  clk, strobe, rst_n;
reg  [NLEN-1:0] b;
reg  [NLEN-1:0] e;
reg  [NLEN-1:0] m;
wire            ready,busy;
wire [NLEN-1:0] r;
modu_expo_128 inst (b,e,m,r,clk,strobe,rst_n,
                    ready,busy);
initial begin
  #0 rst_n = 0; clk = 1;
  #0 strobe = 0;
  #1 rst_n = 1;                        // encryption
  b = 128'd179441695220040973036856247560209845703;
  e = 128'd78624383815806095082831236375207684303;
  m = 128'd291173165596690131543379395216261834371;
  #2 strobe = 1;
  #2 strobe = 0;
  wait(ready);
  #353                                 // decryption
  b = 128'd212957456342734650649396939600336433714;
  e = 128'd232543530691965449749356023879307323711;
  #2 strobe = 1;
  #2 strobe = 0;
  wait(ready);
  #800 $stop;
end
always #1 clk = !clk;
endmodule
```

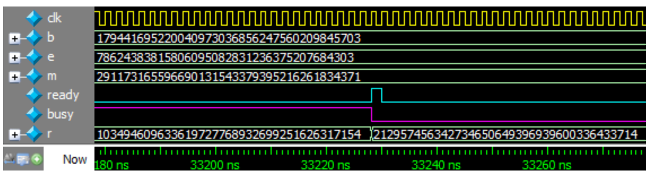

Figure. 4: Shift-Sub Modular Multiplication RSA encryption

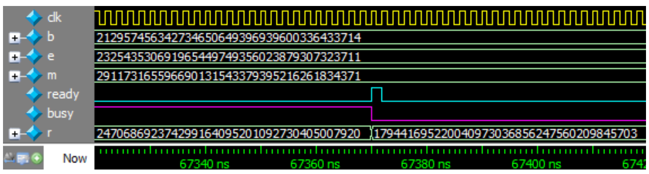

Figure. 5: Shift-Sub Modular Multiplication RSA decryption

Figure 4 and Figure 5 show the simulation waveforms of Algo 6, generated with ModelSim, for SSMM RSA encryption and decryption, respectively. We can see that the input b of the decryption is the output r of the encryption, and the output r of the decryption is exactly the same as the input b of the encryption ($b = (b^e \bmod m)^d \bmod m = 179441695220040973036856247560209845703$):

$$
\begin{aligned}
b &= 179441695220040973036856247560209845703;\\
e &= 78624383815806095082831236375207684303;\\
d &= 232543530691965449749356023879307323711;\\
m &= 291173165596690131543379395216261834371.
\end{aligned}
$$

Note that the decryption key d is shown as e in the decryption waveform Figure 5. From the four waveform figures, we found that RSA cryptography using SSMMul() uses fewer clock cycles than that using MMMul().

IV. Cost/Performance Comparisons

We have developed the Verilog HDL codes for Algo 1 to Algo 6 in 64, 128, 256, 512, 1024, and 2048-bit and tried to implement them on an Intel/Altera Cyclone V FPGA (Field-programmable gate array) chip. Algo 6 SSExpMul() which calls Algo 5 SSMMul() is the simplest because it does not require domain transformations. In Algo 6 SSExpMul(), the two calculations, multiply and squaring, can be performed in parallel. Therefore, we arrange two SSMMul() modules.

We also arrange two MMRed() modules for the simultaneous calculations of multiply and squaring for Algo 2 MExpRed(). The algorithm requires domain transformations before and after doing the main iterations of the calculations. Such transformations can be done with MMRed() modules. If we arrange dedicated MMRed() modules for those domain transformations, the hardware cost will be high. Because the domain transformations are not performed in parallel with main iterations, for saving hardware resource, we use only two MMRed() modules for both domain transformations and main iterations.

For the domain transformations, we need to calculate $q = R^2 \bmod m$. In order to increase flexibility for using different R and m, in our implementations, we used SSMMul() module to get $q = R^2 \bmod m = \mathrm{SSMMul}(R - m, R - m, m)$. Thus our Montgomery arithmetic implementations do not require any precomputed values.

Algo 2 MExpRed() requires multipliers to get the product on which MMRed() performs the Montgomery Modular Reduction. The cost of this algorithm is the highest among all the algorithms due to using big multipliers. As the bit width increases, it may not be possible to implement it on the FPGA chip.

Algo 4 MExpMul() also needs the domain transformations but does not require multipliers. It invokes MMMul() which performs additions during the Montgomery Modular Reduction. We also use only two MMMul() modules for both the domain transformations and main iterations. The calculation of $q = R^2 \bmod m$ is performed also by SSMMul() module.

Table 1 lists the cost-performance of all the configurations. Some configurations cannot be implemented due to the lack of hardware resource, mainly the configurations that use big multipliers. The column of Cycles shows the average number of clock cycles of RSA encryption and decryption. The column of F (MHz) shows the frequency in MHz at which the circuit can work. The column of T (ms) shows the time

Table 1: Cost-performance comparison

Bits	Impl.	Cycles	F (MHz)	T (ms)	ALMs	Regs	DSPs
	Algo 2	4,413	42.32	0.104	1,391	1,043	27
64	Algo 4	4,413	103.52	0.043	1,018	1,112	0
	Algo 6	4,076	103.98	0.039	564	874	0
	Algo 2	17,286	27.66	0.625	4,982	1,354	75
128	Algo 4	17,286	81.31	0.213	1,952	1,646	0
	Algo 6	16,752	81.82	0.205	1,084	1,334	0
	Algo 2	67,334	N/A	N/A	71,884	2,619	156
256	Algo 4	67,334	58.35	1.154	3,812	3,242	0
	Algo 6	66,226	59.30	1.117	2,130	2,667	0
	Algo 2	264,442	N/A	N/A	N/A	N/A	N/A
512	Algo 4	264,442	37.55	7.042	7,071	6,420	0
	Algo 6	261,996	35.82	7.314	4,195	5,118	0
	Algo 2	1,053,178	N/A	N/A	N/A	N/A	N/A
1,024	Algo 4	1,053,178	23.40	45.008	16,237	12,772	0
	Algo 6	1,048,412	23.38	44.842	8,863	10,809	0
	Algo 2	4,204,630	N/A	N/A	N/A	N/A	N/A
2,048	Algo 4	4,204,630	N/A	N/A	27,312	24,630	0
	Algo 6	4,206,448	N/A	N/A	16,397	18,450	0

in millisecond ms. It is calculated by dividing the clock cycles by the clock frequency. The column of ALMs shows the required number of Adaptive Logic Modules. The column of Regs shows the required number of flip-flops. The column of DSPs shows the required number of DSP (Digital signal processing) blocks (for implementing multipliers). The N/A (not available) means that the configuration cannot be implemented on FPGA chips due to the lack of hardware resource.

From the table, we can see that when the bit width n is doubled, the number of clock cycles required is approximately quadrupled. This is because that the computational complexity of Algo 2, Algo 4, and Algo 6 is $O(n^2)$, as discussed before. The execution times of Algo 4 and Algo 6 are almost the same. However, Algo 6 uses less hardware resource (55%∼59% ALMs and 69%∼85% Regs).

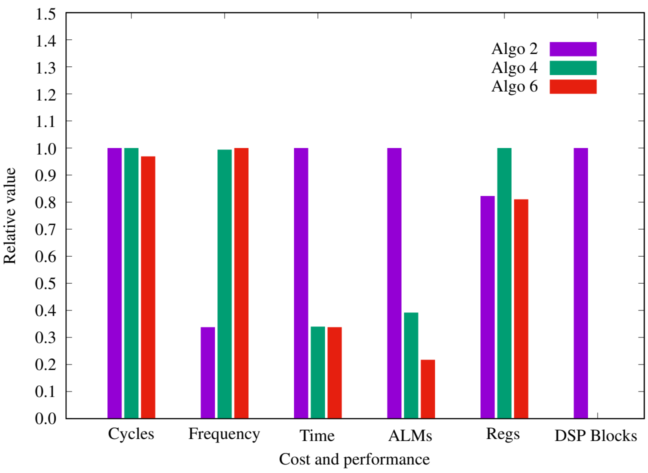

Figure. 6: Cost and performance of exponentiation (128 bits)

Figure 6 and Figure 7 plot the cost and performance of RSA cryptography implementations for $n = 128$ bits and $n = 1024$ bits, respectively. From the figures and Table 1, we conclude that the proposed Algo 6 is a better implementation by considering the cost/performance issue because it achieves almost the same performance but requires less hard-

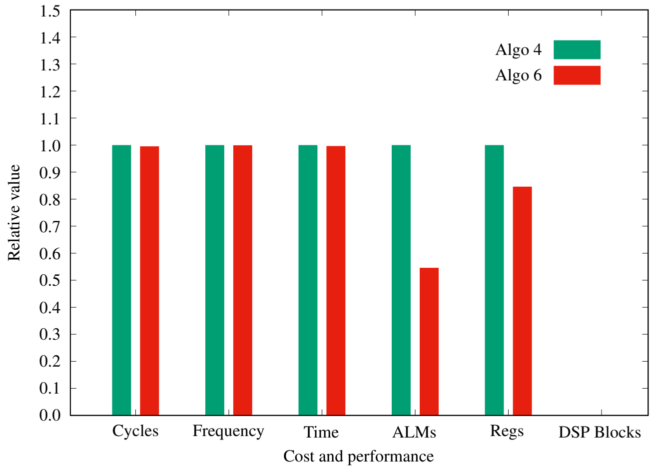

Figure. 7: Cost and performance of exponentiation (1024 bits)

ware resource compared to Algo 4 which uses the Montgomery Modular Multiplication algorithm. Also, Algo 2, which uses the Montgomery Modular Reduction algorithm, is not recommended as it requires big multipliers.

V. Concluding Remarks

RSA cryptography can be performed using the Montgomery Modular Multiplication/Reduction algorithm, and transforming to Montgomery Domain before the calculation and back to the normal domain after the calculation is required. Domain transformations require hardware resources and a special value of $q = R^2 \bmod m$, where $R = 2^n$ and m is an n-bit odd number. In many hardware implementations, q is precomputed for fixed R and m, which reduces flexibility for changing R and m.

The Shift-Sub Modular Multiplication (SSMM) algorithm can be used to calculate $q = R^2 \bmod m$ in fields for RSA cryptography using the Montgomery Modular Multiplication/Reduction algorithm. The SSMM algorithm does not require modular arithmetic, eliminating the need for precomputation and hence increasing the flexibility of hardware implementation.

Furthermore, the SSMM algorithm can be used directly for RSA cryptography. RSA cryptography using the SSMM algorithm can be performed as well as RSA cryptography using Montgomery Modular Multiplication/Reduction. However, RSA cryptography using SSMM does not require domain transformations and therefore reduces the hardware implementation cost.

Our implementations also show that RSA cryptography using Montgomery Modular Multiplication on a multiplicand and a multiplier uses less hardware resource than that using Montgomery Modular Reduction on a product, because the latter requires big hardware multipliers.

References

[1] Tolga Acar and Dan Shumow. Modular reduction without pre-computation for special moduli. *Technical report, Microsoft Research*, January 2010.

[2] Zhengjun Cao, Ruizhong Wei, and Xiaodong Lin. A fast modular reduction method. *IACR Cryptology ePrint Archive*, 2014:1–12, 2014.

[3] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *DISTRIBUTED SYSTEMS Concepts and Design Fifth Edition*. Addison-Wesley, 2012.

[4] Stephen E. Eldridge and Colin D. Walter. Hardware implementation of montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, 1993.

[5] Serdar Süer Erdem, Tuğrul Yanık, and Anıl Celebi. A general digit-serial architecture for montgomery modular multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(5):1658–1668, May 2017.

[6] Miaoqing Huang, Kris Gaj, Soonhak Kwon, and Tarek El-ghazawi. An optimized hardware architecture of montgomery multiplication algorithm. *IACR Cryptology ePrint Archive*, 2007:1–14, 01 2007.

[7] Yamin Li and Wanming Chu. Shift-sub modular multiplication algorithm and hardware implementation for rsa cryptography. In *17th International Comference on Information Assurance and Security*, pages 1–12, On the World Wide Web, December 2021. HIS 2021, LNNS 420.

[8] Chae Hoon Lim, Hyo Sun Hwang, and Pil Joong Lee. Fast modular reduction with precomputation. In *Korea-Japan Joint Workshop on Information Security and Cryptology (JWISC97)*, pages 65–79, 1997.

[9] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[10] Christof Paar. *Implementation of Cryptographic Schemes 1*. Ruhr University Bochum, 2015.

[11] Ronald Linn Rivest, Adi Shamir, and Leonard Max Adleman. A method of obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[12] Colin D. Walter and Royal Holloway. *3 - Hardware Aspects of Montgomery Modular Multiplication*. Cambridge University Press, October 2017.

[13] Ciaran McIvor, Máire McLoone, and John V McCanny. Fast Montgomery modular multiplication and RSA cryptographic processor architectures. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers*, pages 379-384, 2003.

[14] Yuan-Yang Zhang, Zheng Li, Lei Yang, and Shao-Wu Zhang. An efficient CSA architecture for montgomery modular multiplication. *Microprocessors and Microsystems*, 31(7):456–459, November 2007.

[15] Aashish Parihar and Sangeeta Nakhate. Low latency high throughput Montgomery modular multiplier for RSA cryptosystem. *Engineering Science and Technology, an International Journal*, August 2021.

Author Biographies

Yamin Li received his Ph.D in computer science from Tsinghua University in 1989. He currently is a professor in the Department of Computer Science at Hosei University. His research interests include computer arithmetic algorithms, computer architecture, CPU design, parallel and distributed computing, interconnection networks, and fault tolerant computing. Dr. Li is a senior member of the IEEE and a member of the IEEE Computer Society.

Wanming Chu is a faculty member of the Division of Information Systems at the University of Aizu. Her research interests include computer arithmetic algorithm and hardware implementation, multithreaded computer architecture, interconnection networks, fault tolerant computing, performance evaluation, web query, web interface for GIS, and general web-based database management.