RSA Encryption and Decryption using the Redundant Number System on the FPGA

Koji Nakano, Kensuke Kawakami, and Koji Shigemoto
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, JAPAN

Abstract

The main contribution of this paper is to present efficient hardware algorithms for the modulo exponentiation $P^E \bmod M$ *used in RSA encryption and decryption, and implement them on the FPGA. The key ideas to accelerate the modulo exponentiation are to use the Montgomery modulo multiplication on the redundant radix-64K number system in the FPGA, and to use embedded* 18×18*-bit multipliers and embedded 18k-bit block RAMs in effective way. Our hardware algorithms for the modulo exponentiation for* R*-bit numbers* P*,* E*, and* M *can run in less than* $(2R + 4)(R/16 + 1)$ *clock cycles and in expected* $(1.5R + 4)(R/16 + 1)$ *clock cycles. We have implemented our modulo exponentiation hardware algorithms on Xilinx VirtexII Pro family FPGA XC2VP30-6. The implementation results shows that our hardware algorithm for 1024-bit modulo exponentiation can be implemented to run in less than 2.521ms and in expected 1.892ms.*

1 Introduction

It is well known that the addition of two n-bit numbers can be done using a ripple carry adder with the cascade of n full adders [4]. The ripple carry adder has a carry chain through all the n full adders. Thus, the delay time to complete the addition is proportional to n. The carry look-ahead adder [4, 9] which computes the carry bits using the prefix computation can reduce the depth of the circuit. Although the delay time is $O(\log n)$, its constant factor is large and the circuit is much more complicated than the ripple carry adder. Hence, it is not often to use the carry look-ahead adder for actual implementations. On the other hand, redundant number systems can be used to accelerate addition. Using redundant number systems, we can remove long carry chains in the addition. The readers should refer to [9] (Chapter 3) for comprehensive survey of redundant number systems. In our previous paper [6], we have presented redundant radix-64K number system that accelerates arithmetic operations.

The Montgomery modulo multiplication is used to speed the modulo multiplication $X \cdot Y \cdot 2^{-R} \bmod M$ for R-bit numbers X, Y, and M. The idea of Montgomery modulo multiplication is not to use direct modulo computation, which is very costly in terms of the computing time and hardware resources. By iterative computation of Montgomery modulo multiplication, the modulo exponentiation $P^E \bmod M$ can be computed, which is a key operation for RSA encryption and decryption [2, 10]. In our previous paper [6], we have presented an efficient implementation of the Montgomery modulo multiplication in an FPGA. The key ideas are to use 18-bit embedded multiplier in the FPGA and to perform the computation based on the redundant radix-64K number system. The experimental results in [6] show that 1024-bit Montgomery multiplication can be performed in $1.54\mu s$ using 6824 slices and 129 multipliers on a Xilinx Virtex II Family FPGA. However, this implementation needs a lot of multipliers and has long critical path through multipliers. We have improved this result using 18k-bit block RAMs as look up tables [11]. The experimental results in [11] show that 1024-bit Montgomery multiplication can be performed in $1.23\mu s$ using 7883 slices, 64 multipliers, and 29 block RAMs.

The main contribution of this paper is to present hardware algorithms for the modulo exponentiation used in RSA encryption and decryption [10] based on our previous work [6, 11]. Both hardware algorithms using redundant-64K number system run in $R/16 + 1$ clock cycles to complete the Montgomery modulo multiplication $X \cdot Y \cdot 2^{-R} \bmod M$ for R-bit numbers X, Y, and M. We have used these hardware algorithms to complete the modulo exponentiation $P^E \bmod M$ for R-bit numbers P, E, and M. Our implementations for both hardware algorithms run in less than $(2R+4)(R/16+1)$ clock cycles and in expected $(1.5R+4)(R/16+1)$ clock cycles. Thus, the 1024-bit modulo exponentiation can be done in less than 133380 clock cycles and in expected 100100 clock cycles. We have implemented our modulo exponentiation hardware algorithms on Xilinx VirtexII Pro family FPGA XC2VP30-6. The implementation results show that our hardware algorithm for 1024-bit modulo exponentiation can be implemented to run in 2.521ms and in expected 1.892ms.

There are a lot of works that have been presented for the modulo exponentiation (i.e. RSA encryption and decryption) using the Montgomery modulo multiplication. Blum et al. [3] showed that 1024-bit modulo exponentiation can be done in 11.95ms using 6633 CLBs on XC40150XV-8. Amanor et al. [1] presented a hardware algorithm for the Montgomery modulo multiplication and estimated the running time if it is used for 1024-bit modulo exponentiation. From their estimation, it runs in expected 22.7ms on XCV2000E-6. Thus, our implementation runs more than 10 times faster. Mazzeo et al. [7] have presented that RSA encryption $P^E \bmod M$ for 1024-bit numbers P and M, and $E = 2^{16} + 1$ can be done in 2.99ms using 2,902 slices on XCV2000E. Garg and Vig [5] have shown RSA encryption for the same instance in 0.167ms using 28,891 slices on a Xilinx Virtex 4 family FPGA. In our implementation, the RSA encryption can be done in less than $(18+4)(1024/16+1) = 1430$ clock cycles and in 0.027ms for $E = 2^{16} + 1$. Consequently, our implementation runs much faster than known implementations.

2 Redundant Radix-2^r Numbers and Arithmetic Operations

In this paper, we use the following notation to represent the consecutive bits in a number. For a number X, let $X[i,j]$ $(i \geq j)$ be consecutive bits from i-th to j-th bits, where the least significant bits is 0-th bit. For example, $X[6,2] = 11100$ for $X = 11110000$.

A d-digit redundant radix-2^r number is a sequence X of d $(r+2)$-bit numbers $(X_{d-1}, X_{d-2}, \ldots, X_0)$. The value of X is $\sum_{i=0}^{d-1} X_i \cdot 2^{ir}$. We call, for each X_i with $r+2$ bits, $X_i[r-1,0]$ and $X_i[r+1,r]$, *principal bits* and *redundant bits*, respectively. For example, $X = (\underline{00}0101, \underline{01}0011, \underline{11}1111, \underline{10}1111)$ is a 4-digit redundant radix-2^4 number, where underlined binary numbers are redundant bits. If all the redundant bits of this redundant radix-2^4 number are zero, it can be converted to the non-redundant radix-2^4 number by just removing the redundant bits. Also, the non-redundant numbers can be converted to the equivalent redundant numbers by attaching redundant bits $\underline{00}$ to each digit.

From the definition, the value of a d-digit redundant radix-2^r number X is up to $\sum_{i=0}^{d-1}(2^{r+2}-1) \cdot 2^{ir} = \frac{(2^{dr}-1)(2^{r+2}-1)}{2^r-1} > 2^{dr}$. However, we assume that *the valid value* of X is up to $2^{dr}-1$. If the value of X is greater than $2^{dr}-1$, it is regarded as *overflow*. For example, 4-digit redundant radix-2^4 numbers $(\underline{01}0000, \underline{00}0000, \underline{00}0000, \underline{00}0000)$ and $(\underline{00}1101, \underline{11}0000, \underline{00}0000, \underline{00}0000)$ are overflows, because their values are greater than $2^{16}-1$. We assume that, if the resulting value of an operation is a d-digit redundant radix-2^r number and it is greater than $2^{dr}-1$, it is not necessary for a circuit or a program performing the operation to guarantee the correct result due to *the overflow error*. Clearly, the redundant bits $X_{d-1}[r+1,r]$ of the most significant digit X_{d-1} of a d-digit redundant radix-2^r number X are not zero, then the value of X is overflow. Note that X can be overflow even if $X_{d-1}[r+1,r]$ is zero.

In our previous paper [6], we have presented hardware algorithms for various arithmetic operations for redundant radix-2^r numbers. For the reader's benefit, we will review the hardware algorithms for arithmetic operations. The reader should refer to [6] for the details.

2.1 Addition for Redundant Numbers

Let us see the computation of the sum of two redundant numbers. For two 4-digit redundant radix-2^4 numbers $X = (\underline{00}0101, \underline{11}0011, \underline{11}1111, \underline{10}1111)$ and $Y = (\underline{00}0011, \underline{10}1111, \underline{01}1111, \underline{01}0001)$, their sum $Z = X + Y$ can be computed by the position sum as follows:

$$\begin{array}{rrrrr} & \underline{11} & \underline{11} & \underline{10} & \\ & 0101 & 0011 & 1111 & 1111 \\ & \underline{10} & \underline{01} & \underline{01} & \\ + & 0011 & 1111 & 1111 & 0001 \\ \hline & \underline{00}1101 & \underline{01}0110 & \underline{10}0001 & \underline{01}0000 \end{array}$$

Clearly, the addition has no block carry. Let us see the addition of two d-digit redundant radix-2^r numbers X and Y. The sum $Z = X + Y$ can be computed as follows:

$$\begin{aligned} Z_0 &= X_0[r-1,0] + Y_0[r-1,0] \\ Z_i &= X_{i-1}[r+1,r] + X_i[r-1,0] + Y_{i-1}[r+1,r] \\ &\quad + Y_i[r-1,0] \quad (1 \leq i < d) \end{aligned}$$

Hence, $Z_0 < 2^r + 2^r = 2^{r+1}$ and $Z_i < 4 + 2^r + 4 + 2^r < 2^{r+2}$ holds if $r \geq 2$. Thus, Z is a correct redundant radix-2^r number.

Let us design a combinational circuit to compute the sum $Z = X + Y$. Let $\mathrm{ADD}(2,2,r,r)$ denote an adder circuit that computes the sum of two 2-bit and two r-bit integers. Also, let $\mathrm{ADD}(A,B,C,D)$ denote the resulting value of the sum of 2-bit numbers A and B, and r-bit numbers C and D. Clearly, $Z_0 = \mathrm{ADD}(0,0,X_0[r-1,0],Y_0[r-1,0])$ and $Z_i = \mathrm{ADD}(X_{i-1}[r+1,r], Y_{i-1}[r+1,r], X_i[r-1,0], Y_i[r-1,0])$. Thus we have,

Lemma 1 *The addition of two d-digit redundant radix-2^r numbers can be computed using d adders* $\mathrm{ADD}(2,2,r,r)$ *without block carries, whenever* $r \geq 2$.

2.2 Multiplication of Redundant Numbers

We show that the multiplication of 3-digit and 1-digit redundant radix-2^4 numbers can be computed without block carry. Let $X = (\underline{01}0011, \underline{10}0011, \underline{10}1111)$ and $Y = (\underline{10}0101)$. The product $X \cdot Y$ can be computed using 6-bit×6-bit=12-bit multiplications as follows.

$$
\begin{array}{rrrrrr}
 & & & \underline{01}0011 & \underline{10}1001 & \underline{01}0001 \\
\times & & & & & \underline{10}0101 \\
\hline
 & & & 0010 & 0111 & 0101 \\
 & & 0101 & 1110 & 1101 & \\
+ & 0010 & 1011 & 1111 & & \\
\hline
 & \underline{00}0010 & \underline{01}0000 & \underline{01}1111 & \underline{01}0100 & \underline{00}0101
\end{array}
$$

Clearly, we do not have the block carries. Let us formally confirm that the multiplication of d-digit and 1-digit redundant radix-2^r numbers can be computed without block carries. Let X and Y be d-digit and 1-digit redundant radix-2^r numbers. Also, let $P_i = X_i \cdot Y$ $(0 \leq i \leq d-1)$ be the partial multiplication. Since both X_i and Y has $r+2$ bits, P_i has $2r+4$ bits. We can compute the product $S = X \cdot Y$ as follows.

$$
\begin{aligned}
S_0 &= P_0[r-1,0] \\
S_1 &= P_0[2r-1,r] + P_1[r-1,0] \\
S_i &= P_{i-2}[2r+3,2r] + P_{i-1}[2r-1,r] \\
&\quad +P_i[r-1,0] \quad (2 \leq i \leq d-1) \\
S_d &= P_{d-2}[2r+3,2r] + P_{d-1}[2r-1,r] \\
S_{d+1} &= P_{d-1}[2r+3,2r]
\end{aligned}
$$

Hence, $S_0 < 2^r$, $S_1 < 2^r + 2^r = 2^{r+1}$, $S_d < 2^r$, and $S_{d+1} < 2^4$ hold. Also, if $r \geq 3$ then $S_i < 2^4 + 2^r + 2^r \leq 2^{r+2}$ holds. Thus, $S = (S_{d+1}, S_d, \ldots, S_0)$ is a redundant radix-2^r number.

Let $\mathrm{MUL}(r+2, r+2)$ and $\mathrm{ADD}(4,r,r)$ denote combinational circuits to compute the $(2r+4)$-bit product of two $(r+2)$-bit numbers and the $(r+2)$-bit sum of one 4-bit and two r-bit numbers. Each of the partial products $P_{d-1} = X_{d-1} \cdot Y$, $P_{d-2} = X_{d-2} \cdot Y, \ldots, P_0 = X_0 \cdot Y$ can be computed using $\mathrm{MUL}(r+2, r+2)$. After that, each S_i can be computed using $\mathrm{ADD}(4,r,r)$. Thus, we have

Lemma 2 *The product of d-digit and 1-digit redundant radix-2^r numbers can be computed using d $\mathrm{MUL}(r+2, r+2)$s, and d $\mathrm{ADD}(4,r,r)$s, whenever $r \geq 3$.*

Next, to show a circuit to compute two d-digit redundant numbers, we will show how to add a $(d+1)$-digit radix-2^r number C to the product $X \cdot Y$. More specifically, we will show how to compute $S = X \cdot Y + C$. Later, C is used to store interim results of the product sum. We can compute each digit of the sum T can be computed as follows.

$$
\begin{aligned}
T_0 &= P_0[r-1,0] + C_0[r-1,0] \\
T_1 &= P_0[2r-1,r] + P_1[r-1,0] \\
&\quad +C_0[r+1,r] + C_1[r-1,0] \\
T_i &= P_{i-2}[2r+3,2r] + P_{i-1}[2r-1,r] \\
&\quad +P_i[r-1,0] + C_{i-1}[r+1,r] \\
&\quad +C_i[r-1,0] \quad (2 \leq i \leq d-1) \\
T_d &= P_{d-2}[2r+3,2r] + P_{d-1}[2r-1,r] \\
&\quad +C_{d-1}[r+1,r] + C_d[r-1,0] \\
T_{d+1} &= P_{d-1}[2r+3,2r] + C_d[r+1,r]
\end{aligned}
$$

Clearly, each T_i can be computed using $\mathrm{ADD}(2,4,r,r,r)$, and the resulting value has no more than $r+2$ bits if $r \geq 5$. Thus, T is a $(d+2)$-digit redundant radix-2^r number and we have,

Lemma 3 *For a d-digit redundant radix-2^r number X, a 1-digit redundant radix-2^r number Y, and a $(d+1)$-digit redundant radix-2^r number C, the product sum $X \cdot Y + C$ can be computed using d $\mathrm{MUL}(r+2, r+2)$s, $d+2$ $\mathrm{ADD}(2,4,r,r,r)$s, and a $(d+1)(r+2)$-bit registers, whenever $r \geq 5$.*

Let $T = \mathrm{PS}(X,Y,C)$ denote the circuit (or function) for Lemma 3. Using $\mathrm{PS}(X,Y,C)$ we can compute the sum C of two d-digit redundant radix radix-2^r numbers X and Y. Let $X = (X_{d-1}, X_{d-2}, \ldots, X_0)$ and $Y = (Y_{d-1}, Y_{d-2}, \ldots, Y_0)$ be two d-digit redundant radix radix-2^r numbers. We will show how to compute the product $P = (P_{2d-1}, P_{2d-2}, \ldots, P_0) = X \cdot Y$ using $\mathrm{PS}(X, Y_i, C)$. We compute partial products $X \cdot Y_0$, $X \cdot Y_1$, …, $X \cdot Y_{d-1}$ in turn. We use $C = (C_d, C_{d-1}, \ldots, C_0)$ to denote registers storing a interim $(d+1)$-digit redundant radix radix-2^r number. We first compute $\mathrm{PS}(X, Y_0, 0)$. Then, P_0 is the least significant digit $\mathrm{PS}(X, Y_0, 0)[r+1, 0]$. We store the remaining $d+1$ digits $\mathrm{PS}(X, Y_0, 0)[(d+1)(r+2)-1, r+2]$ in C. After that, we compute $\mathrm{PS}(X, Y_1, C)$. Clearly, P_1 is the least significant digit $\mathrm{PS}(X, Y_1, C)[r+1, 0]$ holds, and then we store the remaining $d+1$ digits $\mathrm{PS}(X, Y_1, C)[(d+1)(r+2)-1, r+2]$ in C. Continuing similarly, we can obtain the product $M = X \cdot Y$. Thus we have,

Lemma 4 *For two d-digit redundant radix-2^r numbers X and Y, the product $X \cdot Y$ in the redundant radix-2^r representation can be computed in d clock cycles using d $\mathrm{MUL}(r+2, r+2)$s, $d+2$ $\mathrm{ADD}(2,4,r,r,r)$s, and a $(d+1)(r+2)$-bit register, whenever $r \geq 5$.*

3 Montgomery Modulo Multiplication

In the RSA encryption/decryption, the modulo exponentiation $C = P^E \bmod M$ or $P = C^D \bmod M$ are computed, where P and C are plain and cypher text, and (E, M) and (D, M) are encryption and decryption keys. Usually, the number of bits in P, E, D, and M is 1024 or larger. Also, the modulo exponentiation is repeatedly computed for fixed E, D, and M, and various P and C. Since modulo operation is very costly in terms of the computing time and hardware resources, we use *Montgomery modulo multiplication* [8], which does not use direct modulo operations. In the Montgomery modulo multiplication, three R-bit numbers X, Y, and M are given, and $(X \cdot Y + k \cdot M) \cdot 2^{-R} \bmod M$ is computed, where an integer k is selected such that the least significant R bits of $X \cdot Y + k \cdot M$ are zero. The value of k can be computed as follows. Let $(-M^{-1})$ denote the minimum non-negative number such that $(-M^{-1}) \cdot M \equiv -1($ or $2^R - 1)$ $(\bmod\ 2^R)$. If M is odd, then $(-M^{-1}) < 2^R$ always holds. We can select k such that $k = ((X \cdot Y) \cdot (-M^{-1}))[r-1, 0]$. For such k, $(X \cdot Y + k \cdot M)[r-1, 0]$ are zero.

Since $0 \leq X, Y < M < 2^R$ and $0 \leq k < 2^R$, we can guarantee that $(X \cdot Y + k \cdot M) \cdot 2^{-R} < 2M$. Thus, by subtracting M from $(X \cdot Y + k \cdot M) \cdot 2^{-R}$, we can obtain $(X \cdot Y + k \cdot M) \cdot 2^{-R} \bmod M$ if it is not less than M. Since $X \cdot Y + k \cdot M \equiv X \cdot Y \pmod{M}$, we write $(X \cdot Y + k \cdot M) \cdot 2^{-R} \bmod M = X \cdot Y \cdot 2^{-R} \bmod M$.

4 Modulo Exponentiation using Montgomery Modulo Multiplication

Let us see how Montgomery modulo multiplication is used to compute $C = P^E \bmod M$. Since R and M are fixed, we can assume that $2^R \bmod M$ and $2^{2R} \bmod M$ are computed beforehand. Let the binary representation of E be $e_{R-1}e_{R-2}\cdots e_1$. The modulo exponentiation $C = P^E \bmod M$ can be computed using the following algorithm based on the right-to-left method:

```
1. $C \leftarrow 2^R \bmod M$;
2. $\mathcal{P} \leftarrow \underline{P \cdot (2^{2R} \bmod M) \cdot 2^{-R} \bmod M}$;
3. for $i = R-1$ downto 0 do
4.     begin
5.         $C \leftarrow \underline{C \cdot C \cdot 2^{-R} \bmod M}$;
6.         if $e_i = 1$ then $C \leftarrow \underline{C \cdot \mathcal{P} \cdot 2^{-R} \bmod M}$;
7.     end
8. $C \leftarrow \underline{C \cdot 1 \cdot 2^{-R} \bmod M}$;
```

The underlined formulas are computed by the Montgomery modulo multiplication. Let us confirm that $C = P^E \bmod M = P^{e_{R-1}e_{R-2}\cdots e_0} \bmod M$ holds. Clearly, $\mathcal{P} = P2^R \bmod M$ holds. Let us show that, at the end of the **for**-loop for $i = k$, $C = P^{e_{R-1}e_{R-2}\cdots e_k}2^R \bmod M$ holds by induction on k. Suppose that $C = P^{e_{R-1}e_{R-2}\cdots e_k}2^R \bmod M$ holds at the end of the **for**-loop for $i = k$. After executing line 5 of the following **for**-loop, we have $C = (P^{e_{R-1}e_{R-2}\cdots e_k}2^R \bmod M) \cdot (P^{e_{R-1}e_{R-2}\cdots e_k}2^R \bmod M) \cdot 2^{-R} \bmod M = P^{e_{R-1}e_{R-2}\cdots e_k 0}2^R \bmod M$. If $e^{k-1} = 0$ then the Montgomery modulo multiplication in line 6 is not executed. Hence, $C = (P^{e_{R-1}e_{R-2}\cdots e_{k-1}}2^R \bmod M)$ holds. If $e^{k-1} = 1$ then it is executed, and thus, we have $C = (P^{e_{R-1}e_{R-2}\cdots e_k 0}2^R \bmod M) \cdot \mathcal{P} \cdot 2^{-R} \bmod M = P^{e_{R-1}e_{R-2}\cdots e_k 1}2^R \bmod M = P^{e_{R-1}e_{R-2}\cdots e_{k-1}}2^R \bmod M$. This completes the proof of the induction. Thus, after terminating the **for**-loop, we have $C = P^E 2^R \bmod M$. Finally, by the Montgomery modulo multiplication in line 8, we have $C = (P^E 2^R \bmod M) \cdot 1 \cdot 2^{-R} \bmod M = P^E \bmod M$.

In the worst case, $e_i = 1$ for all i. If this is the case, the Montgomery modulo multiplication is executed no more than $2R + 2$ times. Also, since $e_i = 1$ with probability $1/2$, it is executed expected $1.5R + 2$ times. Thus we have,

Lemma 5 *The modulo exponentiation $C = P^E \bmod M$ for R-bit numbers P, E, and M can be computed by executing the Montgomery modulo multiplication $2R+2$ times and expected $1.5R + 2$ times. if $2^R \bmod M$ and $2^{2R} \bmod M$ are given.*

5 Hardware Algorithms for Montgomery Modulo Multiplication

The main purpose of this section is to review hardware algorithms [6, 11] for Montgomery modulo multiplication.

5.1 Block-Carry-Free Implementation of Montgomery Modulo Multiplication

Recall that in the Montgomery modulo multiplication, R-bit numbers X, Y, and M are given. In this subsection, we assume X and Y are a d-digit redundant radix-2^r number and a 1-digit redundant radix-2^r number, respectively. We will show a circuit to compute the Montgomery modulo multiplication $(X \cdot Y + k \cdot M) \cdot 2^{-r}$ for such X, Y, and M. We assume that the value of X and Y are given to the circuit as inputs, M is fixed and $(-M^{-1})$ is computed beforehand. This assumption makes sense if Montgomery modulo multiplication is used to compute the modulo exponentiation for RSA encryption and decryption.

Recall that, using the circuit for Lemma 2, $X \cdot Y$ can be computed using d $\mathrm{MUL}(r+2, r+2)$s and d $\mathrm{ADD}(4, r, r)$s. After computing $X \cdot Y$, we need to compute k such that the least significant r bits of $(X \cdot Y + k \cdot M)$ are zero. We can compute $k = ((X \cdot Y)[r-1, 0] \cdot (-M^{-1}))[r-1, 0]$ using a $\mathrm{MUL}(r, r)$. Once k is obtained, the product $k \cdot M$ is computed using the circuit for Lemma 2. Finally, the sum $(X \cdot Y + k \cdot M)$ is computed by the circuit for Lemma 1. Note that both $X \cdot Y$ and $k \cdot M$ are $(d+1)$-digit redundant radix-2^r numbers. However, since the least significant digit of $X \cdot Y$ and $k \cdot M$ are zero, we can omit the addition of the least significant digit. The readers should refer to Figure 1 for illustrating the circuit for Lemma 6.

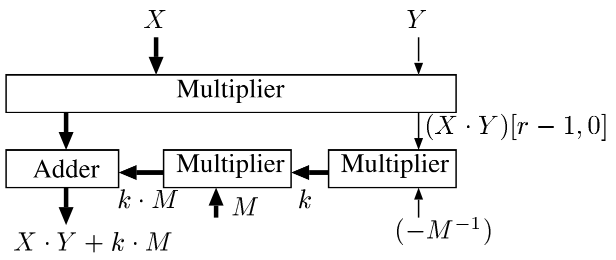

Figure 1. Circuit to compute $(X \cdot Y + k \cdot M)$ **using multipliers**

To compute the multiplication $X \cdot Y$, we can use a circuit for Lemma 2 which uses d $\mathrm{MUL}(r+2, r+2)$s and d $\mathrm{ADD}(4,r,r)$. To compute k, we use a $\mathrm{MUL}(r,r)$. After that to compute the multiplication $k \cdot M$, we also use a circuit for Lemma 2 and the addition $X \cdot Y + k \cdot M$ can be computed using d $\mathrm{ADD}(2,2,r,r)$ by Lemma 1.

Therefore, we have,

Lemma 6 *[6] Montgomery modulo multiplication* $(X \cdot Y + k \cdot M) \cdot 2^{-r}$ *for d-digit X and 1-digit Y of redundant radix-*2^r *representation can be computed using* $2d+1$ $\mathrm{MUL}(r+2, r+2)$*s,* $2d$ $\mathrm{ADD}(4,r,r)$*s, and* d $\mathrm{ADD}(2,2,r,r)$*, without block carries, whenever r* ≥ 4.

5.2 Montgomery Modulo Multiplication Using a Memory

The circuit for Lemma 6 has a cascade of three multipliers, which can be a long critical path. Also, it needs too many multipliers. We remove multipliers for computing k to improve the circuit for Lemma 6. The key idea is to use a memory to look up the value of $k \cdot M$.

Let f be a function such that $f(Z) = (Z[r-1,0] \cdot (-M^{-1}))[r-1,0] \cdot M$. The function f can be computed using a 2^r word $(d+1)r$-bit memory as follows. The value of $f(i)$ $(0 \leq i \leq 2^r - 1)$ is stored in address i of the memory in advance. Then, by reading address $Z[r-1,0]$ of the memory, we can obtain the value of $f(Z)$ in one clock cycle. Using this memory, $f(X \cdot Y)$ can be computed in one clock cycle. After that, the addition $X \cdot Y + f(X \cdot Y)$ can be computed using $d+1$ $\mathrm{ADD}(2,2,r,r)$s from Lemma 1. Figure 2 illustrates the circuit to compute $X \cdot Y + f(X \cdot Y)$.

Note that the least significant digit of $X \cdot Y + f(X \cdot Y)$ is always zero. Hence, we can omit the computation of the least significant digit of f and the following addition. Thus, we use a 2^r word dr-bit memory for computing $f(X \cdot Y)$ and d $\mathrm{ADD}(2,2,r,r)$s to compute the sum $X \cdot Y + f(X \cdot Y)$. Therefore, we have,

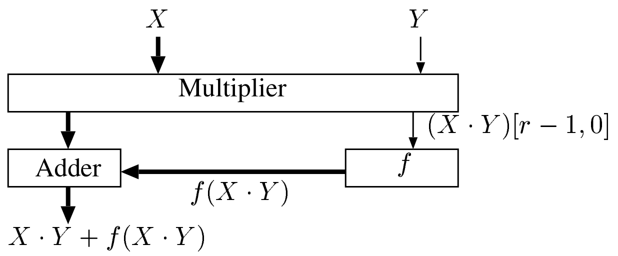

Figure 2. Circuit to compute $X \cdot Y \cdot + f(X \cdot Y)$ **using a memory**

Lemma 7 *[11] Montgomery modulo multiplication* $(X \cdot Y + k \cdot M) \cdot 2^{-r}$ *for d-digit X and M, and 1-digit Y of redundant radix-*2^r *representation can be computed using* d $\mathrm{MUL}(r+2, r+2)$*s,* $d+2$ $\mathrm{ADD}(2,4,r,r,r)$*s,* d $\mathrm{ADD}(2,2,r,r)$*, and a* 2^r*-word dr-bit memory, without block carries, whenever r* ≥ 5.

If $r = 16$ and $dr = 1024$, then the circuit for Lemma 7 needs $64K$-word 1024-bit memory of size $64M$ bits. Since the size of block memory of current FPGAs is up to few mega bits, this circuit cannot be implemented in FPGAs.

5.3 Montgomery Modulo Multiplication Using a Fewer Memory

We will reduce the size of memory to compute the function f. Recall that, k is a r-bit number such that the least significant r bits of $X \cdot Y + k \cdot M$ are zero. Let r-bit k number partition into two $r/2$ bits such that $\overline{k} = k[r-1, r/2]$ and $\underline{k} = k[r/2-1, 0]$. We can compute the values of $\overline{k}$ and $\underline{k}$ separately as follows. Let $(-M^{-1})$ be the minimum non-negative integer such that $(-M^{-1}) \cdot M \equiv -1 \pmod{2^{r/2}}$. Also, let $\overline{XY} = (X \cdot Y)[r-1, r/2]$ and $\underline{XY} = (X \cdot Y)[r/2-1, 0]$. We set $\underline{k} = \underline{XY} \cdot (-M^{-1})$. Then, the least significant $r/2$ bits of $X \cdot Y + \underline{k} \cdot M$ are zero. Let g be a function such that $g(Z) = ((Z[r/2-1,0] \cdot M)[r-1, r/2] \cdot (-M^{-1}) + c$ and $c = 0$ if $(Z \cdot M)[r/2-1, 0] = 0$ and $c = 1$ otherwise. Function g can be computed using a combinational circuit with $r/2$ input bits and $r/2$ out put bits. We set $\overline{k} = (\overline{XY} + g(\underline{XY}))[r/2-1, 0]$. Then, the least significant digit of $X \cdot Y + \underline{k} \cdot M + \overline{k} \cdot M \cdot 2^{r/2}$ is zero.

We will implement this idea in the same way as Lemma 7. Instead of computing $\underline{k}$ and $\overline{k}$, we compute $\underline{k} \cdot M$ and $\overline{k} \cdot M$ using a memory. Let h be a function such that $h(Z) = (Z[r/2-1, 0] \cdot (-M^{-1}))[r/2-1, 0] \cdot M$. Similarly to f, function g can be computed using $2^{r/2}$-word $(d(r+2) + r/2)$-bit memory. Then, $\underline{k} \cdot M = h(\underline{X \cdot Y})$ and $\overline{k} \cdot M = h(\overline{XY} + g(\underline{XY}))$ holds. Thus, $k \cdot M = \underline{k} + \overline{k} \cdot 2^{r/2} = h(\underline{X \cdot Y}) + h(\overline{XY} + g(\underline{XY})) \cdot 2^{r/2}$. The readers should refer to Figure 3 for illustrating the circuit to compute $X \cdot Y + k \cdot M = X \cdot Y + h(\underline{XY}) + h(\overline{XY} + g(\underline{XY})) \cdot 2^{r/2}$. Since a FPGAs has dual port memories, two modules to compute h in Figure 3 can be computed by a single $2^{r/2}$-

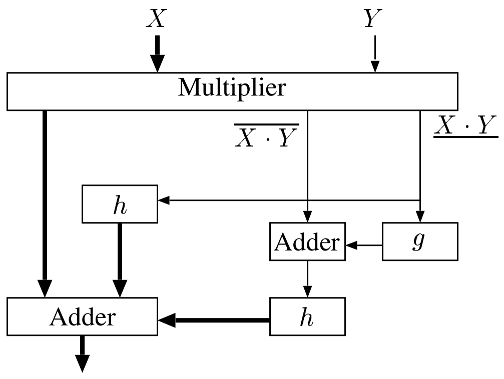

Figure 3. Circuit to compute $X \cdot Y + h(\underline{X \cdot Y}) + h(\overline{X \cdot Y} + g(\underline{X \cdot Y})) \cdot 2^{r/2}$

word $(dr + r/2)$-bit dual port memory in the same time. The readers may think that a combinational circuit to compute g is not necessary. However, block RAMs in most FPGAs to implement a memory support only synchronous read. Thus, one clock cycle is necessary to read a memory. It follows that, if we use a memory to implement the computation of g, two clock cycles are necessary to compute $h(\overline{XY} + g(\underline{XY}))$.

Let us evaluate the hard aware resources necessary to compute $X \cdot Y + h(\underline{X \cdot Y}) + h(\overline{X \cdot Y} + g(\underline{X \cdot Y})) \cdot 2^{r/2}$. The multiplication $X \cdot Y$ can be computed using d $\mathrm{MUL}(r+2, r+2)$s and d $\mathrm{ADD}(4, r, r)$s from Lemma 2. Function $g(\underline{X \cdot Y})$ can be computed using a combinational circuits with 8 input bits and 8 output bits and addition $\overline{X \cdot Y} + g(\underline{X \cdot Y})$ can be computed $\mathrm{ADD}(r/2, r/2)$. After that the value of function h for two arguments can be computed using a $2^{r/2}$-word dr-bit dual-port memory. Finally, the sum $(X \cdot Y) + h(\underline{X \cdot Y}) + h(\overline{X \cdot Y} + g(\underline{X \cdot Y})) \cdot 2^{r/2}$ can be computed using d $\mathrm{ADD}(2, 2, 2, r, r, r)$ by straightforward generalization of Lemma 1. Consequently, we have,

Lemma 8 *[11] Montgomery modulo multiplication* $(X \cdot Y + k \cdot M) \cdot 2^{-r}$ *for d-digit* X *and* M*, and* Y *and 1-digit* Y *of redundant radix-*2^r *representation can be computed using* d $\mathrm{MUL}(r+2, r+2)$*s,* d $\mathrm{ADD}(4, r, r)$*s, one* $\mathrm{ADD}(r/2, r/2)$*,* d $\mathrm{ADD}(2, 2, 2, r, r, r)$*, a* $2^{r/2}$*-word dr-bit dual-port memory, and a combinational circuit with* $r/2$*-bit input and* $r/2$*-bit output, without block carries, whenever* $r \geq 4$.

5.4 Montgomery Modulo Multiplication for Two d-digit Numbers

Recall that we have shown a hardware algorithm for the product of d-digit and 1-digit radix-2^r numbers is shown for Lemma 2. Using this hardware algorithm iteratively, the product of two d-digit radix-2^r numbers can be computed as shown in Lemma 4. In the previous subsection, we have shown hardware algorithms for Montgomery modulo multiplication for $(X \cdot Y + k \cdot M) \cdot 2^{-r}$ for d-digit X and M, and 1-digit Y are shown for Lemmas 6 and 8. Using the same technique, we can obtain hardware algorithms for d-digit numbers. More specifically, from Lemma 6, we have,

Theorem 9 *Montgomery modulo multiplication* $(X \cdot Y + k \cdot M) \cdot 2^{-dr}$ *for three d-digit redundant radix-*2^r *numbers* X*,* Y *and* M *can be computed in* d *clock cycles using* $2d+1$ $\mathrm{MUL}(r+2, r+2)$*s,* $2d$ $\mathrm{ADD}(4, r, r)$*s,* d $\mathrm{ADD}(2, 2, 2, r, r, r)$*, and a* $(d+1)(r+2)$*-bit register, without block carries, whenever* $r \geq 5$.

Further, from Lemma 8, we have,

Theorem 10 *Montgomery modulo multiplication* $(X \cdot Y + k \cdot M) \cdot 2^{-dr}$ *for three d-digit redundant radix-*2^r *numbers* X*,* Y *and* M *can be computed in* d *clock cycles using* d $\mathrm{MUL}(r+2, r+2)$*s,* $d+2$ $\mathrm{ADD}(2, 4, r, r, r)$*s,* d $\mathrm{ADD}(2, 2, 2, r, r, r)$*, a* $2^{r/2}$*-word dr-bit dual port memory, and a combinational circuit with* $r/2$*-bit input and* $r/2$*-bit output, and a* $(d+1)(r+2)$*-bit register, without block carries, whenever* $r \geq 5$.

The readers should refer to [6, 11] for the details.

6 Modulo Exponentiation on the FPGA

The hardware algorithms for the Montgomery modulo multiplication shown in Section 5 compute $(X \cdot Y + k \cdot M) \cdot 2^{-dr}$. Recall that $(X \cdot Y + k \cdot M) \cdot 2^{-dr}$ can be larger than M. Thus, we need to subtract M if it is no less than M to obtain $X \cdot Y \bmod M$. In other words, we need to check if $(X \cdot Y + k \cdot M) \cdot 2^{-dr}$ is less than M. Unfortunately, the comparison of two redundant radix-2^r numbers is not obvious. To perform the comparison, we need to convert them into the non-redundant numbers and the conversion is very costly. Therefore, we do not check if $(X \cdot Y + k \cdot M) \cdot 2^{-dr}$ is less than M. Alternatively, we check if the redundant bits of the most significant digit are not zero. If this is the case, we add $-M$ to $(X \cdot Y + k \cdot M) \cdot 2^{-dr}$. Since the redundant bits of the most significant digit are either 00 or 01, we can guarantee that, after the addition, they are 00. Note that, $-M$ may not be added if $(X \cdot Y + k \cdot M) \cdot 2^{-dr}$ is no less than than M. However, since we can guarantee that the redundant bits of the most significant digit are 00, we can avoid the overflow. Thus, the Montgomery modulo multiplication hardware algorithms for Theorems 10 and 9 can be modified to obtain the resulting value with the redundant bits of the most significant digit being 00 in $d+1$ clock cycles.

Suppose that these modified circuits for Montgomery modulo multiplication are used to compute the modulo exponentiation algorithm based on the algorithm in Section 4.

At the end of the algorithm the value C is stored as a d-digit redundant radix-2^r number. We first convert it to the d-digit non-redundant radix-2^r number. For this purpose, we add zero d times. After that, all that redundant bits. Note that the resulting value can be no less than M. Hence we check if it is no less than M. If this is the case, we add $-M$ and perform the k iterations of addition zero again to convert it to non-redundant number. If $M \geq 2^{dr-1}$, we can guarantee that the value thus obtained is less than M. In this way, we can obtain $C = P^E \bmod M$ in non-redundant number system.

Let us evaluate the clock cycles to obtain $C = P^E \bmod M$ using Theorems 9 or 10. The Montgomery modulo multiplication takes $d + 1$ clock cycles, and it is executed at most $2dr + 2$ times from Lemma 5. After that, it is converted to non-redundant number in d clock cycles. If the resulting value is no less than M, $-M$ is added and then zero is added d times to it in totally $d + 1$ clock cycles. Thus, the modulo exponentiation can be computed in $(2dr + 2)(d + 1) + 2d + 1 < (2R + 4)(R/r + 1)$ clock cycles and in expected $(1.5R + 4)(R/r + 1)$ clock cycles.

Theorem 11 *The modulo exponentiation $C = P^E \bmod M$ for R-bit numbers can be computed using hardware algorithms for Theorems 9 or 10 in less than $(2R+4)(R/r+1)$ clock cycles.*

If we use non-redundant numbers, the conversion from the redundant numbers is not necessary. If this is the case, the modulo exponentiation can be computed in $(2R+2)(R/r+1)$ clock cycles and in expected $(1.5R+2)(R/r+1)$ clock cycles.

7 Experimental Results

We have implemented our hardware algorithms for the modulo exponentiation on Virtex II Pro Family FPGA XC2VP30-6, which has 13,696 slices, 136 18×18-bit multipliers, and 136 18k-bit dual-port block RAMs. We have used XST in ISE Foundation 10.1i for logic synthesis and analysis. Since this FPGA has 18-bit multipliers as building blocks, it makes sense to let $r = 16$. Thus, we use redundant radix-64K (i.e. radix-2^{16}) number system.

Table 1 shows the performance of the experimental results of the modulo exponentiation shown in Theorem 9 and Theorem 11, which use 18×18-bit multipliers. Table 2 shows that for Theorem 10 and Theorem 11, which use 18×18 multipliers and 18k-bit block RAMs. In both table, the performance are evaluated for d-digit redundant radix-64K numbers and non-redundant numbers. Clearly, the clock frequency for redundant numbers are fixed, while it decreases as the number of bits increases for non-redundant numbers. For example, in Table 2, the 1024-bit modulo exponentiation runs in 52.9MHz for redundant numbers and in 18.46 MHz for non-redundant numbers. Hence, the total computing time for redundant numbers is 2.521 ms, while that for non-redundant numbers is 7.218 ms. Thus, we have achieved the speedup factor of 2.86 using redundant number systems.

8 Conclusion

We have presented hardware algorithms for the modulo exponentiation used in RSA encryption and decryption. The best algorithm runs in 2.67ms and in expected 1.99ms to compute $P^E \bmod M$ for 1024-bit numbers P, E, and M on Xilinx Virtex II Pro Family FPGA XCVP30-6. It also runs in 0.027ms if $E = 2^{16} + 1$. Our hardware algorithms run faster than previously presented algorithms.

References

[1] D. N. Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler. Efficient hardware architectures for modular multiplication on FPGAs. In *Proc. of International Conference on Field Programmable Logic and Applications*, pages 539–542, 2005.

[2] T. Blum and C. Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE Trans. on Computers*, 50(7):759–764, 2001.

[3] T. Blum and C. Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE Transactions on Computers*, 50(7):759–764, 2001.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[5] R. Garg and R. Vig. An efficient montgomery multiplication algorithm and RSA cryptographic processor. In *Proc. of International Conference on Computational Intelligence and Multimedia Applications*, pages 188–195, 2007.

[6] K. Kawakami, K. Shigemoto, and K. Nakano. Redundant radix-2^r number system for accelerating arithmetic operations on the FPGAs. In *Proc. of International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 370–377, 2008.

[7] A. Mazzeo, L. Romano, G. P. Saggese, and N. Mazzocca. FPGA-based implementation of a serial RSA processor. In *Proc. of Design, Automation and Test in Europe Conference and Exhibition*, 2003.

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

[9] B. Parhami. *Computer Arithmetic - Algorithm and Hardware Designs*. Oxford University Press, 2000.

[10] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120 – 126, 1978.

[11] K. Shigemoto, K. Kawakami, and K. Nakano. Accelerating montgomery modulo multiplication for redundant radix-64k number system on the FPGA using dual-port block RAMs. In *Proc. of International Conference On Embedded and Ubiquitous Computing(EUC)*, pages 44–51, 2008.

Table 1. Modulo exponentiation using 18-bit multipliers (Theorems 9 and 11)

	bits	64	128	256	512	1024
redundant	clock(MHz)	40.80	41.50	40.38	40.30	40.21
	clock cycles (worst)	660	2340	8772	33924	133380
	time (ms)	0.016	0.056	0.217	0.841	3.317
	clock cycles (expected)	500	1764	6596	25476	100100
	time (ms)	0.012	0.042	0.161	0.632	2.489
	slices	865	1708	2905	5811	13467
	multipliers	9	17	33	65	129
non-redundant	clock(MHz)	47.43	41.75	33.61	25.12	16.38
	clock cycles (worst)	650	2322	8738	33858	133250
	time (ms)	0.013	0.055	0.259	1.347	8.134
	clock cycles (expected)	480	1746	6562	25344	99970
	time (ms)	0.010	0.041	0.195	1.008	6.103
	slices	530	974	1971	3909	7549
	multipliers	9	17	31	63	123

Table 2. Modulo exponentiation using 18-bit multipliers and 18k-bit block RAMs(Theorems 10 and 11)

	bits	64	128	256	512	1024
redundant	clock(MHz)	53.20	52.51	52.77	52.57	52.90
	clock cycles (worst)	660	2340	8772	33924	133380
	time (ms)	0.012	0.044	0.166	0.645	2.521
	clock cycles (expected)	500	1764	6596	25476	100100
	time (ms)	0.009	0.033	0.124	0.484	1.892
	slices	896	1652	3204	5868	11589
	multipliers	4	8	16	32	64
	block RAMs	2	4	8	15	29
non-redundant	clock(MHz)	68.06	58.83	44.47	30.40	18.46
	clock cycles (worst)	650	2322	8738	33858	133250
	time (ms)	0.09	0.039	0.196	1.113	7.218
	clock cycles (expected)	480	1746	6562	25344	99970
	time (ms)	0.07	0.029	0.147	0.833	5.415
	slices	613	1086	2034	3911	7708
	multipliers	4	8	15	31	61
	block RAMs	2	4	8	15	29