# Modular Exponentiation using Parallel Multipliers

S.H. Tang, K.S. Tsui and P.H.W. Leong
{shtang,kstsui}@alumni.cse.cuhk.edu.hk, phwl@cse.cuhk.edu.hk
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, NT, Hong Kong

## Abstract

*A field programmable gate array (FPGA) semi-systolic implementation of a modular exponentiation unit, suitable for use in implementing the RSA public key cryptosystem is presented. The design is carefully matched with features of the FPGA architecture, utilizing embedded* $18 \times 18$*-bit multipliers on the FPGA and employing a carry save addition scheme. Using this architecture, a 1024-bit modular exponentiation can operate at 90 MHz on a Xilinx XC2V3000-6 device and perform a 1024-bit RSA decryption in 0.66 ms with the Chinese Remainder Theorem.*

## 1 Introduction

The RSA algorithm is the most widely used public key cryptosystem and modular exponentiation of long integers is the primary operation required in its computation. Since RSA is often the bottleneck for e-Commerce servers, there has been great interest in finding efficient techiques to perform long modular exponentiation.

Field-Programmable Gate Arrays (FPGAs) are hardware devices which are reconfigurable, i.e. programming an FPGA can change its functionality. Implementations of cryptographic hardware using FPGAs offer higher performance than software implementations since higher degrees of parallelism can be achieved. Compared with traditional implementations using application specific integrated circuits (ASICs), FPGAs offer several advantages:

- With FPGAs, it is possible to reconfigure the chip for different encryption standards on demand. This means that unused encryption schemes need not reside on the FPGA, saving resources. In contrast, all supported encryption schemes must reside on an ASIC.
- It is possible to offer field upgrades for FPGA based systems to support bug fixes and new standards.
- FPGAs offer lower costs for small volumes, shorter development times and faster time to market over ASIC technology.
- The performance of an FPGA implementation can be improved by replacing an existing device with a faster one and does not involve any further engineering.

FPGA-based RSA processors have been previously reported. In 1993, Shand and Vuillemin built a RSA implementation which used 16 Xilinx XCV3090 devices and achieved a decryption speed of 165 kb/s for a 1024-bit key [13]. It used a number of optimization techniques including the Chinese Remainder Theorem, precomputation of small powers, Hensel's odd division, Karatsuba multiplication, squaring optimization, a carry completion adder and quotient pipelining. The design was the fastest RSA implementations for many years. In 2001 [1], Blum and Paar proposed a systolic architecture using a single Xilinx XC40250XV-09 device which performed 1024-bit decryption at 330 kb/s.

In this work, a modular exponentiation architecture is presented which takes advantage of recent features incorporated in FPGA devices. Specifically, the $18 \times 18$-bit multipliers embedded in the Xilinx Virtex II series devices were used to speed up computation of partial products in a Montgomery multiplier, and a semi-systolic pipeline scheme were employed to maximize the clock rate and parallelism achieved. The radix employed is much higher than previous designs, resulting in fewer clock cycles and hence higher performance. The design was able to achieve a thoughput of 1.51 Mb/s on a Xilinx XC2V4000-6 device, approximately four times faster than the previously best reported figure [1].

The rest of this paper is organized as follows. In Section 2, the RSA algorithm [12] is introduced. Section 3 is an overview of the design philosophy employed. Section 4 describes the algorithms used in the implementation including modular exponentiation and multiplication. Section 5 looks into the architectural issues and gives a hardware structural design of the RSA processor. Results are presented in Sec-

| | |
|---|---|
| $\langle a \rangle_b$ | $a \bmod b$ |
| $h$ | Exponent ($E$) size in bits. |
| $k$ | Modulus ($M$) size in bits. |
| $r$ | Radix size in bits. |
| $\beta$ | radix $= 2^r$ |
| $s_0$ | Least significant digit of S $= \langle S \rangle_\beta$ |
| $n$ | Modulus size in digits. |

**Table 1. Notation used in this paper.**

tion 6 and conclusions drawn in Section 7. Table 1 is a summary of the notation used throughout this paper.

## 2 The RSA Algorithm

Suppose Alice wishes to send a secret message to Bob. In a secret key system, Alice and Bob need to know the same secret key and so it must be communicated via some secure channel. In a public key system, Bob broadcasts his public key and Alice can use it to encode a message. The design of the cryptosystem is such that only Bob can decode the encrypted message, but no exchange of secret keys is necessary.

In the RSA cryptosystem, Alice must first find Bob's public key $M, E$ which are the modulus and exponent respectively. She calculates the cipher-text ($\mathcal{C}$) from the plain-text ($\mathcal{P}$) by:

$$\mathcal{C} = \mathcal{P}^E \pmod{M} \tag{1}$$

To decode the message, Bob uses his private key ($D$) to recover the plain text by:

$$\mathcal{P} = \mathcal{C}^D \pmod{M} \tag{2}$$

To generate the key, two large prime numbers $P$ and $Q$ are first generated, and two equations are used to calculate $D$, $E$ and $M$:

$$\begin{array}{rcl} M & = & PQ \\ DE & \equiv & 1 \pmod{(P-1)(Q-1)} \end{array} \tag{3}$$

where $E$ is relatively prime to $(P-1)(Q-1)$. In practice, $E$ is often chosen to be a small number such as $2^{16}+1$ which reduces the amount of computation required to perform encryption. The strength of RSA depends on the key size $k$, the number of bits in $M$.

Breaking RSA is believed to be as hard as factorizing $M$ to $P, Q$, which is intractable for $k \geq 1024$ with current technology.

Generally, it takes a considerable time for RSA decryption due to the very large exponent. Chinese Remainder Theorem (CRT) can be applied to speed up decryption 4 times with minimal hardware addition. CRT makes use of the secret factors $P, Q$ to break a 1024-bit (for example) modulus $M$ and exponent $E$ in modular exponentiation into two halves, each roughly 512-bits long, (depending on size of $P, Q$). The result of RSA decryption $\mathcal{P}$ is defined by the following steps [5].

$$\begin{array}{ll} T_1 = \mathcal{C}^{D_1} \pmod{P} & , D_1 = \langle D \rangle_{P-1} \\ T_2 = \mathcal{C}^{D_2} \pmod{Q} & , D_2 = \langle D \rangle_{Q-1} \\ \mathcal{P} = T_1 + \langle (T_2 - T_1) \times (P^{-1} \bmod Q) \rangle_Q \times P & \end{array} \tag{4}$$

Halving the modulus size makes the modular multiplier $2\times$ faster and 50% smaller. Thus compared with a non-CRT implementation, it allows for the use of two smaller units in parallel (to calculate $T_1, T_2$). Halving the exponent size makes another $2\times$ speedup. In total, a $4\times$ speedup can be achieved with the same chip area.

## 3 Design Considerations

The encryption and decryption process in RSA cryptography requires the computation of a modular exponentiation. This in turn requires an efficient implementation of modular multiplication.

The number $A$ in radix $\beta$ can be represented as a sequence of digits $a_i$ as follows:

$$A = [a_{n-1} \dots a_0]_\beta = \sum_{i=0}^{n-1} a_i \beta^i$$

In such a case, a standard serial n-digit multiplication requires O($n^2$) [4] single digit multiplications. In a serial-parallel hardware design processing $n$ digits in parallel, the number of cycles increases linearly with $n$. Given a fixed operand size, increasing the radix size decreases the number of digits. Hence, doubling the radix size halves the number of cycles.

While it is common to use radixes greater than 2 to reduce the number of clock cycles, previous hardware implementations (FPGA and ASIC) used a relatively low radix because the area of the resulting implementation grows rapidly with radix size. The number of logic levels and hence the maximum clock frequency is also reduced as the radix is increased. Shand [13] and Orup [11] used radices of $2^2$ and $2^5$ respectively, while Blum [1] used a radix of $2^4$, implementing the multiplier using a look-up table (LUT). In contrast, software implementations often use radices $2^{32}$ or $2^{64}$ to take advantage of 32 or 64-bit multipliers respectively.

The introduction of dedicated multipliers in FPGAs such as the Xilinx Virtex II and Altera Stratix devices provides an opportunity to use a high radix without the area and performance issues described above. A Virtex-II FPGA chip can have up to 168 18-bit signed multipliers evenly distributed in the LUT matrix [15], and the speed and area of these multipliers are much better than that which could be expected from an implementation using the FPGA's logic resources. In the design described in this paper, a radix of $\beta = 2^{17}$ was chosen so that unsigned arithmetic could be used thoughout the design, and smaller radices could be used for testing.

## 4 Algorithm

In this section, the algorithms used in the design are presented in a top-down fashion.

### 4.1 Modular Exponentiation

**Algorithm 1** L-R Binary method.

```
Input: $E = \sum_{i=0}^{h-1} e_i \times 2^i$
    (Binary representation of $E = [e_{h-1} \dots e_0]_2$)
Output: $P = C^E$
  $P <= 1$
  for $i = h-1 \dots 0$ do
    $P <= P \times P$
    if $e_i = `1'$ then
      $P <= P \times C$
    end if
  end for
```

The L-R binary method [4] was used for exponentation. For every bit in the exponent, $P$ is squared (corresponding to doubling the index) and if the current bit of exponent is '1', a multiplication with $C$ is performed (corresponding to adding 1 to the index). Squaring is performed by ordinary multiplication. The total number of multiplications is $\frac{3}{2}h$ (where $h$ is defined in Table 1), assuming the exponent has an equal ratio of '1' and '0' bits.

The multiplier output $P$ is always passed to one of the inputs of next step, while the other input is either one of $P$ or $C$. This leads to to a very simple datapath (figure 7).

In modular exponentation, every intermediate product is immediately reduced before being passed to the next step. The size of every operation stays within the range $[0, M)$.

Shand [13, 4] used a Star Chain to perform exponentiation with a $1.3\times$ speed up, over the L-R binary method. However, a star chain has the disadvantage that one must calculate the Star Chain for each different exponent, which is an additional overhead.

### 4.2 Modular Multiplication

Montgomery's method [9] provides an efficient way to perform modular multiplication without reduction by the modulus $M$, if $M > 2$ and $M$ is odd. It transforms the original $M$-residue to an $R$-residue, where $R$ is deliberately chosen as some power of 2, allowing divisions to be replaced by shift operations. An additional input $M'$ is pre-computed as the negative modular inverse of $M$, i.e. $(-MM') \bmod R = 1$.

An implementation of Montgomery's method for RSA typically interleaves multi-precision multiplication steps with Montgomery's reduction steps, thereby reducing the number of multiplication operations. The interleaved version of the algorithm listed below is taken from [6]:

```
Output: $S = ABR^{-1} \pmod{M}$
   $S <= 0$
1: for $i = 0 \dots n$ do
2:   $q <= \langle \langle S \rangle_\beta \times \langle M' \rangle_\beta \rangle_\beta$
3:   $S <= (S + qM)/\beta + a_i B$
4: end for
```

There are two row-times-digit operations: $a_i \times B$ and $q \times M$. In line 2, the modulo operation extracts the least significant digit. In line 3, division by $\beta$ is implemented with a right shift by one digit.

Montgomery Multiplication produces a result with an unwanted $R^{-1}$ factor. Therefore, pre-processing and a post-processing steps are needed. In the following example, $C^{10}$ is computed in the left column. The right column operates with the $R^{-1}$ factor. As can be seen, preprocessing step involves a Montgomery multiplication with $R^2$ (externally precomputed) and the postprocessing is a multiplication with 1.

$$\begin{array}{r|r}
 & CR <= (C)(R^2)R^{-1} \\
C^2 <= (C)(C) & C^2R <= (CR)(CR)R^{-1} \\
C^4 <= (C^2)(C^2) & C^4R <= (C^2R)(C^2R)R^{-1} \\
C^5 <= (C^4)(C) & C^5R <= (C^4R)(CR)R^{-1} \\
C^{10} <= (C^5)(C^5) & C^{10}R <= (C^5R)(C^5R)R^{-1} \\
 & C^{10} <= (C^{10}R)(1)R^{-1}
\end{array}$$

Orup [10] points out the possibility of simplifying the operations in line 2 by pre-computing the product $M \times (M' \bmod \beta) = \widetilde{M}$. The algorithm is rewritten as

```
1: for $i = 0 \dots n$ do
2:   $q <= \langle S \rangle_\beta$
3:   $S <= (S + q\widetilde{M})/\beta + a_i B$
4: end for
```

An immediate consequence is the increase in range of $\widetilde{M}$ by one digit, causing the range of output $S$ to also increase by one digit. The following proof shows that the output is in the range $[0, 2\widetilde{M}]$ [2]:

**Proof:** Given input $A, B \in [0, 2\widetilde{M}], R > 4\widetilde{M}$
output $S \in [0, 2\widetilde{M}]$

$$\begin{aligned}
U &= \langle AB \rangle_R \Rightarrow U \in [0, R] \\
S &= (U\widetilde{M} + AB)/R \\
&< (U\widetilde{M} + (2\widetilde{M})^2)/R \\
&= (U + 4\widetilde{M})\widetilde{M}/R \\
&< (R + R)\widetilde{M}/R = 2\widetilde{M}
\end{aligned}$$

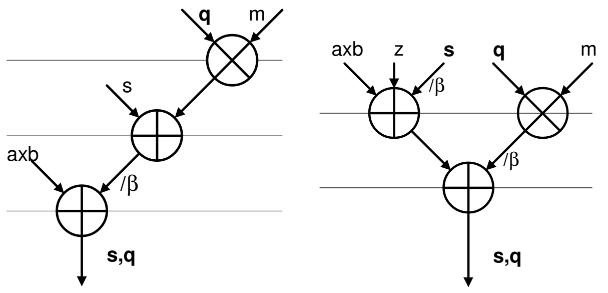


**Figure 1. Diagram showing number of logic levels for two different addition methods.**

Line 3 of the algorithm requires $S$ to be added to $q\widetilde{M}$ before the addition with $a_iB$. As illustrated in the left hand part of Figure 1, this is an undesirable constraint as it adds another level to the logic in its implementation. Line 3 can be rewritten as:

$$S <= S/\beta + q\widetilde{M}/\beta + z + a_iB \qquad (5)$$

Where $z$ is the carry out of $\langle S\rangle_\beta + \langle q\widetilde{M}\rangle_\beta$. Montgomery's reduction ensures that $S + q\widetilde{M}$ is always divisible by $\beta$. Thus $\langle S\rangle_\beta$ and $\langle q\widetilde{M}\rangle_\beta$ has a carry $z = 1$ iff $q\widetilde{M}$ is non-zero. As $\widetilde{M}$ is always non-zero, it is possible to determine $z$ by simply inspecting $q$. This allows the addition to be done in the order $(S/\beta + a_iB) + q\widetilde{M}/\beta$ and Figure 1 illustrates that fewer levels of logic are required. Kornerup [6] and Orup [10] proposed a different method which achieves the same result without conditional branching.

Algorithm 2 describes the final Montgomery Multiplication algorithm that was adopted.

**Algorithm 2** Montgomery Multiplication.

```
Input: $\widetilde{M} = M \times (M' \bmod \beta)$,    $R = 4\beta^{n+1}$
Output: $S = ABR^{-1} \pmod M$
1: $S <= 0$
2: for $i = 0 \ldots n$ do
3:     $q <= \langle S\rangle_\beta$
4:     if $q = 0$ then
5:         $S <= S/\beta + q\widetilde{M}/\beta + a_iB$
6:     else
7:         $S <= S/\beta + q\widetilde{M}/\beta + a_iB + 1$
8:     end if
9: end for
```

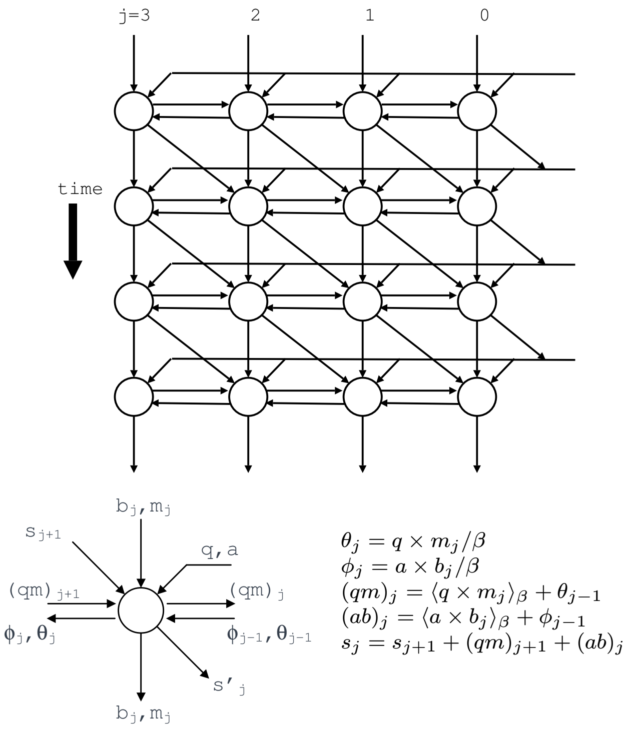

j=3
2
1
0
time
b_j,m_j
s_j+1
q,a
(qm)_j+1
(qm)_j
φ_j,θ_j
φ_j-1,θ_j-1
s'_j
b_j,m_j
$\theta_j = q \times m_j/\beta$
$\phi_j = a \times b_j/\beta$
$(qm)_j = \langle q \times m_j\rangle_\beta + \theta_{j-1}$
$(ab)_j = \langle a \times b_j\rangle_\beta + \phi_{j-1}$
$s_j = s_{j+1} + (qm)_{j+1} + (ab)_j$

**Figure 2. DG graph for n=4.**

## 5 Architecture

### 5.1 Processing Elements

In order to utilize many multipliers in parallel, a serial-parallel multiplier was employed. That is, for the computation of $A \times B$, $a_iB$ was computed per clock cycle.

A direct implementation of Algorithm 2 is unlikely to be able to achieve a high operating frequency for 1024-bit numbers. A systolic array [8] architecture was thus employed. Rewriting Algorithm 2 into digit-wise operations of recursive equations, a dependency graph (DG) was derived.

Figure 2 shows a single cell. As in Algorithm 2 it accepts $a, b, q, m$ as inputs. $b, m$ are fixed for every cell so all digits are input to all cells in parallel, and passed vertically. $q, a$ are the loop variants so they are input in a digit serial fashion. Multiplication of two $r$-bit numbers produces a $2r$-bit partial product. $\phi$ and $\theta$ are the most significant digits of the $qm_j$ and $ab_j$ multiplications respectively and are passed to the left cell. Line two of Algorithm 2 requires the computation of $s/\beta$ and $qm/\beta$. The division of $\beta$ is done by shifting $s$ and $qm$ one digit to the right. The output of the right-most cell, $s_0$, is taken as the quotient digit in of line 1 in algorithm. The output $s$ appears at the bottom of the DG. The DG was projected vertically to form the chain of PEs as shown Figure 3.

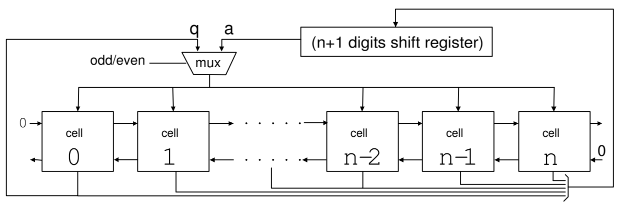


**Figure 3. A row of PEs showing data communications.**

The existence of broadcasting in our design makes it semi-systolic. Compared to fully systolic implementations by Kornerup [7] or Blum [1], our clock frequency may be hindered by the global signals. Since we use a large radix, the number of digits is fewer and therefore the effect is diluted, and the performance is acceptable. A 1024-bit design has a 10% lower clock frequency than a 512-bit design. On the other hand, the semi-systolic approach does not need extra registers for the signals and enjoys a shorter latency.

## 5.2 Carry Save Addition

As can be seen from the DG, two additions are required to form $S$ and two for the partial products $q\widetilde{M}$ and $a_iB$. Thus each processing element must perform 4 additions. Although we partition the 1024-bit row into $n$ PEs (one for each digit), the carry propagation path is not shown in the DG. If the carry is propagated horizontally, it will result in a 1024-bit ripple carry path.

A 1024-bit ripple-carry path, accelerated by Xilinx's carry chain [15], will result in the carry being broken into four vertical columns. Measurements show that such an adder has a delay of 70 ns[1] and is a clear bottleneck in the design.

A solution is to propagate the one-bit carry *vertically* to the next iteration using a carry save adder. Since a division by the radix is performed, the carry is propagated back to the same PE. Hence a redundant representation, namely a (one-bit-carry : r-bit-sum) tuple was used. A ripple-carry adder is used for adding each digit, and the carry-out of each digit is saved for the next cycle. This can be validated by noticing that $s$ is an accumulator, and is hence commutative. At each iteration only the digit $s_0$ is used for the computation of the next iteration.

The redundant representation is restored to a unique positional representation at completion of modular multiplication by using a carry completion circuit [13], an OR-gate connected to the carry of all digits. The circuit keeps adding the carries until there is no further carry-out from any digit.

[1]Timing and structure verified using Xilinx ISE 5.1 and XC2V3000-6 target device.

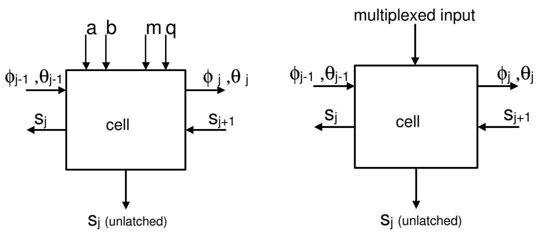


**Figure 4. Time Multiplexed I/O of Cells.**

| | time = 2t | time = 2t + 1 | time = 2t + 2 |
|---|---|---|---|
| Multiplier | $a_t*b_0 \rightarrow (\phi_0^t, i_0^t)$<br>...<br>$a_t*b_k \rightarrow (\phi_k^t, i_k^t)$<br>...<br>$a_t*b_n \rightarrow (\phi_n^t, i_n^t)$ | $q^t*m_1 \rightarrow (\theta_1^t, j_1^t)$<br>...<br>$q^t*m_k \rightarrow (\theta_k^t, j_k^t)$<br>...<br>$q^t*m_n \rightarrow (\theta_n^t, j_n^t)$<br>$z^t=(q^t=0?\ 0:1)$ | (next iteration)<br>...<br>$a_{t+1}*b_k \rightarrow (\phi_k^{t+1}, i_k^{t+1})$ |
| Adder 1 | (previous iteration)<br>...<br>$j_k^{t-1} + \theta_{k-1}^{t-1} \rightarrow v_k^{t-1}$ | $i_0^t + z^t \rightarrow u_0^t$<br>$i_1^t + \phi_0^t \rightarrow u_1^t$<br>...<br>$i_k^t + \phi_{k-1}^t \rightarrow u_k^t$<br>...<br>$\phi_n^t \rightarrow u_{n+1}^t$ | $j_1^t + \theta_0^t \rightarrow v_1^t$<br>...<br>$j_k^t + \theta_{k-1}^t \rightarrow v_k^t$<br>...<br>$\theta_n^t \rightarrow v_{n+1}^t$ |
| Adder 2 | (previous iteration)<br>...<br>$w_k^{t-1} + v_{k+1}^{t-1} \rightarrow s_k^t$<br>$s_0^t \rightarrow q^t$ | $u_0^t + s_1^t \rightarrow w_0^t$<br>...<br>$u_k^t + s_{k+1}^t \rightarrow w_k^t$<br>...<br>$u_{n+1}^t \rightarrow w_{n+1}^t$ | $w_0^t + v_1^t \rightarrow s_0^{t+1}$<br>...<br>$w_k^t + v_{k+1}^t \rightarrow s_k^{t+1}$<br>...<br>$w_{n+1}^t \rightarrow s_{n+1}^{t+1}$<br>$s_0^{t+1} \rightarrow q^{t+1}$ |

Superscript represents time and subscript represents space.

**Figure 5. Timing diagram of 2-cycle pipelined design.**

In practice, only the very rare case in which there are more than r-bits of consecutive set bits require more than one cycle to complete the conversion.

## 5.3 Pipelining and Timing

Figure 1 shows that two cycles are needed for resolving the data dependency in the computation of $S$. The schedule of steps for the multipliers and adders are listed in figure 5. In the description below, odd/even steps refer to the two different types of cycles in this schedule.

By observing the similarity between the operations in the first and second cycles, data-path components were reused, in particular, sharing of arithmetic circuits (adders, multipliers), registers and global routing paths was achieved. This is served to minimize global routing resources, e.g. a multiplexed I/O was used to combine $q, a$ into one port. The same port can be used to feed $m$ into the cells during initialization by using an extra cycle. The datapath design is shown in figure 6.

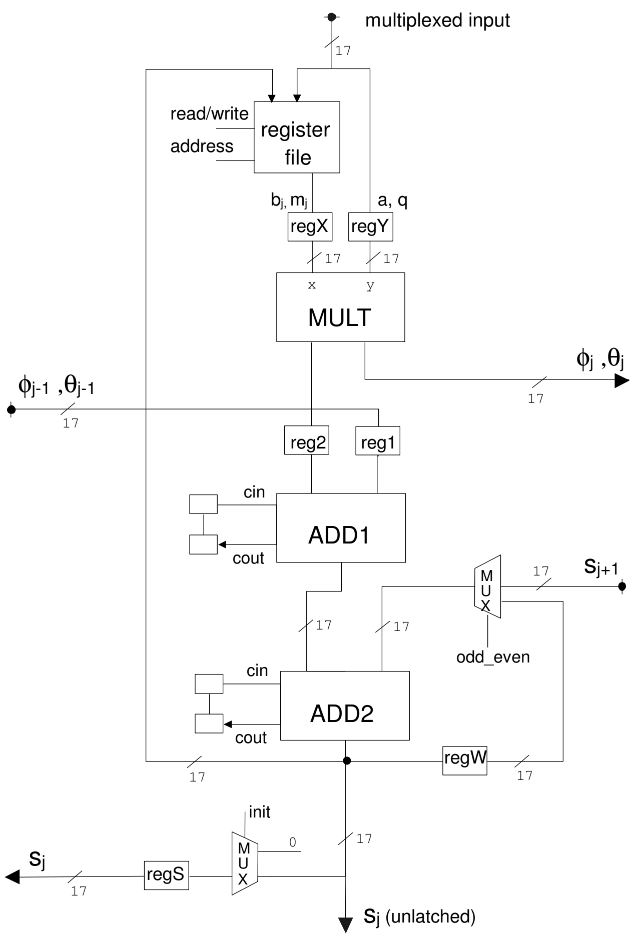
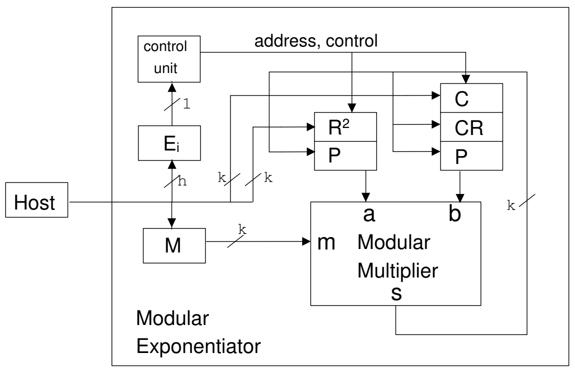


**Figure 6. Datapath of Multiplier.**



The register files at input $a$, $b$ correspond to the content of shift register in figure 3 and register file inside PE in figure 6 respectively. The operands $C$, $P$ and $R$ can be referred to algorithm 1 and the example of $C^{10}$ at section 4.2.

**Figure 7. Datapath of exponentiation unit.**

| **Operation** | **Input** $b\times$**Input** $a$ | **Output** $s$ |
|---|---|---|
| pre-1[1] | $C \times R^2$ | $CR$ |
| pre-2 | $1 \times R^2$ | $P$ |
| square | $P \times P$ | $P$ |
| multi | $CR \times P$ | $P$ |
| post | $1 \times P$ | $P$ |

**Table 2. Pre and post processing for modular exponentiation.**

## 5.4 Datapath

MULT takes one parallel input 'x' and one digit-serial input 'y'. 'x' accepts inputs $b$ and $m$, where $b$ will be substituted for some other values in exponentiation. These values are initially stored in a register file in every cell. 'y' accepts broadcast, time-multiplexed input of $q$ and $a$. As in figure 3, $a$ is supplied by a shift register, and $q$ is passed from the output of cell 0. MULT gives out a 34-bit product, split into upper and lower digits. The upper digit ($\phi$, $\theta$) is passed to the cell on the right. The upper and lower digits are registered in reg1 and reg2.

ADD1 adds the partial products to produce $u$ and $v$. The carry-out is registered and passed as a carry-in for the next iteration (corresponding to passing the carry vertically in the DG graph). Because two cycles are used for one iteration, the carry is latched twice so that carry of $u^t$ is added to $u^{t+1}$, and carry of $v^t$ is added to $v^{t+1}$. Cell 0 does an additional check on $q$ and asserts the carry-in if $q \neq 0$ (not shown in diagram).

ADD2 adds all the numbers to form the next $s$. In the first cycle (2t+1 in table 5), the multiplexer is set to pass $s$ stored in regS to the next cell. ADD1 gives out $u$. The sum $w$ is stored in regW. In the second cycle (2t+2), regW is used as an input and ADD2 gives out $v$. The sum $s$ is stored in regS. For cell 0, $s$ is also broadcasted to all cells as input $q$.

## 5.5 Exponentiation

The Montgomery multiplication cells are put to use in exponentiation steps (algorithm 1) and Montgomery pre and post processing (described in section 4.2). The input and output for *futa* in these steps are defined in table 2. Input $b$ is selected from the register file inside the PE. Input $a$ is selected from the shift register in figure 3. After completion of a modular multiplication, the result $s$ is placed in a suitable address in the register file and the shift register.

As the multiply and exponentiation units operate with parallel data input, the key and data are input to the chip and registered in the register file of the PEs and shift register of figure 3 during initialization. The processor uses 64-bit ports for key and data input.

Table 4 shows cycle usage equations deduced from sim-

| Key size | Clock Freq. | No. of slices | No. of multipliers |
|---|---|---|---|
| 512 | 99.26MHz | 8235 | 32 |
| 1024 | 90.11MHz | 14334 | 62 |

The XC2V3000 device used has a total of 14336 slices and 96 multipliers

**Table 3. Speed and area of 512 and 1024-bit design reported by Xilinx ISE 5.1.**

ulation measurement. For modular multiplication, $n+1$ iterations are required, and each iteration consumes 2 cycles in our pipelined design. There are a few extra cycles for register transfer and control, which might be further optimized. In total, $2(n+5)$ cycles are used. For exponentiation, the average number of modular multiplications is $\frac{3h}{2}+b-h$ where the $(b-h)$ term is from the number of zero 'padding' bits.

## 6 Results

The RSA processor was implemented using VHDL with automatic Place-and-route (PAR). Simulation was done using Modelsim 5.5f and the result was verified to be correct by comparison with the open source OpenSSL library. Synthesis, PAR, bitstream implementation and timing measurement were done using the Xilinx ISE 5.1 package. The target device was XC2V3000 with speed grade -6. The speed and area utilization as reported by the Xilinx tools are summarized in Table 3.

Table 4 shows the number of cycles required for different operations measured from simulation waveforms. Table 5 shows the average RSA processing speed measured from ModelSim for different sized keys. The measured time included key/data I/O time but excluded external precomputation of key and external normalization of result. The Chinese Remainder Theorem (CRT) result was determined by assuming two half-size data are processed in parallel using one RSA processor. Finally, Table 6 compares the speed of our implementation with those reported in the literature. For a 1024-bit RSA using the CRT, our design can achieve a decryption time of 0.66 ms (corresponding to 1.51 Mb/s) which is 4 times faster than the previously fastest reported result [1].

## 7 Conclusion

An RSA processor was presented which employs multipliers embedded in an FPGA to achieve high performance. The multipliers enable a high radix ($2^{17}$) to be used, reducing both latency and the number of cycles for a modular

| Operation | Generalized[1] | 512b | 1024b |
|---|---|---|---|
| **Modular Multiplication** | | | |
| MP $\times$ MP | $2(n+5)$ | 74 | 134 |
| **Modular Exponentiation** | | | |
| Montgomery pre/post-processing | $2(n+5)\times 3$ | 222 | 402 |
| Exponentiation | $2(n+5)\times(b-h+\frac{3h}{2})$ | 59200 | 209844 |
| **Input / Output** | | | |
| Input of $R^2, M, A, E$ | $4\times(b/64)+1$ | 37 | 69 |
| Output of $A^E \bmod M$ | $(b/64)+1$ | 10 | 18 |
| **Measured total clock cycles** | | 59468 | 210333 |

[1] $b$ – number of gross bits, 544 and 1054 for r=17 and h=512, 1024 respectively

**Table 4. A breakdown of measured cycle usage for different operations.**

| Key Size | Clock | RSA Time | RSA Rate |
|---|---|---|---|
| 512 w/o CRT | 100MHz | 0.59ms | 1.68 Mb/s |
| 1024 w/o CRT | 90MHz | 2.33ms | 429 kb/s |
| 1024 with CRT[1] | 90MHz | 0.66ms | 1.51 Mb/s |

[1] This design is too large to fit on XC2V3000 device so an XC2V4000 was used.

**Table 5. Processing speed for one block of data in the RSA processor.**

| | Year | Device / Technology | RSA Decryption (1024 CRT) |
|---|---|---|---|
| [11] | 1991-94[1] | ASIC 0.6u | 5.5 ms (512b) |
| [13] | 1993 | 16 XC3090 | 6.06 ms |
| [3] | 1999 | DSP TMS320C6201 | 11.7 ms[2] |
| [1] | 2001 | XC40250XV-09 | 3.10 ms |
| [14] | 2001 | ASIC 0.6u | 2.2 ms (512b)[3] |
| **Ours** | **2003** | **XC2V4000-6** | **0.66 ms** |
| OpenSSL[4] | | Pentium4 1.7G | 6.9 ms |

[1]The design is made in 1991 and the chip is fabricated in 1994.
[2]RSA private key signing, equivalent to decryption speed.
[3]It only gives the best and worst case decryption rate. We take the average here.
[4]Linux kernel-2.4.18-14 i686, OpenSSL version 0.9.6b-29

**Table 6. Comparison with other implementations.**

exponentiation. Through the use of a semi-systolic architecture, a high operating frequency of 90 MHz was achieved. This simple yet efficient architecture achieved a throughput of 1.51 Mb/s for 1024-bit RSA decryption on a Xilinx Virtex XC2V4000-6 device while maintaining a simple yet efficient design.

## References

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

[2] S. R. Dusse and B. S. K. Jr. A cryptographic library for the motorola dsp56000. *Advances in Cryptology, EUROCRYPT 90. Lecture Notes in Computer Science*, 473:230–244, 1990.

[3] K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara. Fast implemenation of public-key cryptography on a DSP TMS320C6201. In *In Proceedings of the First Workshop on Cryptographic Hardware and Embedded Systems (CHES), Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[4] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. AddisonWesley, Massachusetts, 1981.

[5] K. Koc. High-speed RSA implementation. Paper, RSA Laboratories, 1994.

[6] P. Kornerup. High-radix modular multiplication for cryptosystems. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 277–283, Windsor, Canada, 1993. IEEE Computer Society Press, Los Alamitos, CA.

[7] P. Kornerup. A systolic, linear-array multiplier for a class of right-shift algorithms. *IEEE Transactions on Computers*, 43(8):892–898, 1994.

[8] S. Y. Kung. *VLSI Array Processors*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1988.

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

[10] H. Orup. Simplifying quotient determination in high-radix modular multiplication. *Proc. of the 12th Symposium on Computer Arithmetic*, 1995.

[11] H. Orup and P. Kornerup. A high-radix hardware algorithm for calculating the exponential $m^e$ modulo n. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 26–28, Windsor, Canada, 1991. IEEE Computer Society Press, Los Alamitos, CA.

[12] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[13] M. Shand and J. E. Vuillemin. Fast implementations of RSA cryptography. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, Windsor, Canada, 1993. IEEE Computer Society Press, Los Alamitos, CA.

[14] C.-H. Wu, J.-H. Hong, and C.-W. Wu. RSA cryptosystem design based on the chinese remainder theorem. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 391–395. IEEE, 2001.

[15] Xilinx, Inc. *Xilinx Vertex-II Platform FPGA Handbook*, 2001.