Efficient Modular Squaring Algorithms for Hardware Implementation in GF(p)

Lo'ai Tawalbeh, Saed Swedan, Adnan Gutub*

Computer Engineering Department
Jordan University of Science and Technology, Irbid, Jordan
* King Fahd University of Petroleum & Minerals, Dhahran 31261, Saudi Arabia
Email{tawalbeh@just.edu.jo, saed_swedan2000@yahoo.com, gutub@kfupm.edu.sa}

Abstract. Some of the most popular public key encryption algorithms use exponentiation as their core operation which can be mostly broken into several modular squaring operations. In this paper, we present GF(p) modular squaring algorithms and efficiently implement them on hardware. We present different algorithms, two for squaring and one for reduction combined with the squaring to provide a general modular squaring algorithm. The algorithms are implemented through datapaths that uses redundant Carry-Save Adders making the computation time independent form the operands precision. The proposed algorithms are compared with each other as well as with the existing modular squaring algorithms. The experimental results are obtained by synthesizing the hardware designs for FPGA *Virtex5* chip (*xc5vlx50 – ff1153* technology) which showed interesting results making our ideas very attractive.

1 Introduction

Currently, sharing and securing data has become a basic and essential need, which gives higher demand for faster and more secure encryption algorithms. There are two main branches in cryptography; the first is Symmetric Key Cryptography, where the same key, which must remain secret, is used for encryption and decryption. The second is Public Key Cryptography (PKC), where two keys are used. One is called Public Key, which is publicly shared, and is used to encrypt data, the other is called Private Key, which is kept private, and is used to decrypt data. PKC is the most widely used type of cryptography for authentication and digital signatures [3, 4], and depends heavily on modular arithmetic including addition, multiplication, and exponentiation [1, 2].

Author Posting. (c) 'Copyright Holder', 2009.
This is the author's version of the work. It is posted here by permission of 'Copyright Holder' for personal use, not for redistribution.
The definitive version was published in Information Security Journal: A Global Perspective, Volume 18 Issue 3, 2009.
doi:10.1080/19393550902926053 (http://dx.doi.org/10.1080/19393550902926053)

Among modular arithmetic operations, modular exponentiation is the main operation used in many PKC algorithms, such as El-Gamal cryptosystem [1], Diffie-Hellman key exchange algorithm [2], RSA [3], and the Digital Signature Standard [4]. Since modular exponentiation can be broken-down into successive modular squaring and multiplication operations, the performance of PKC algorithms can be enhanced by using fast and efficient modular squaring algorithms [7, 10, 11, 12].

A. Karatsuba and Y. Ofman [5], were one of the first to show that multiplication can be done in time less than $O(n^2)$, where n is the number of digits in the operands. They introduced a recursive algorithm that divides the integer into two parts and continues to be divided recursively, up to a certain limit.

P. G. Comba [6], presented a multiplication algorithm that has improvement over the classical method of multiplication by employing smart programming optimizations. Although the algorithm is mathematically identical to the classical method, it has an advantage of computing partial products directly to reduce the number of required memory writes.

G. Joseph [7], compared the classical multiplication method with the Karatsuba algorithm, the Comba algorithm, and a hybrid between the two algorithms. He found that the Karatsuba algorithm outperforms both the classical method and the Comba algorithm, but the best result came from combining both Karatsuba's and Comba's algorithms up to a certain breakpoint.

C. K. Koc [8], presented the standard squaring algorithm. C. Wu, D. Lou, and T. Chang [10], have shown that Yang-Hseih-Laih's squaring algorithm [9] has an error-indexing bug, and presented an algorithm to fix it.

In this work, we will present the hardware architecture simulation results for several algorithms and their implementation. We will compare their performance with each other and with already known other algorithms in this field. The proposed algorithms are compared against four algorithms: the Standard Squaring Algorithm, the Wu-Lou-Chang Squaring Algorithm [10], the Barrett modular reduction algorithm [11], and finally, the Montgomery modular multiplication algorithm [12]. The experimental results were obtained by synthesizing the hardware designs using FPGA *Virtex5* chip (*xc5vlx50 – ff1153* technology), for different operand sizes (in bits): 8, 16, 32, 64, 128, 256, 512, and 1024. Then, the obtained results are compared with other algorithms results using the same operands sizes.

2 Proposed Algorithms

We present three algorithms; two for modular squaring and one for modular reduction. The squaring algorithms will be denoted as PA1 and PA2 (Proposed Algorithms 1 and 2). The Proposed Modular Reduction algorithm will be denoted as PMR. PA1 and PA2 use an idea that was inspired while researching Carry-Save Adders (CSA).

As in CSAs, we divided the squaring result into two results, i.e., sum and carry. Both sum and carry can be calculated with a delay of one full adder. The sum can be calculated in one step (wiring in case of radix-2, or LUT in case for radix-4 and radix-8),

and the rest of the algorithm's function is to calculate the carry. Instead of starting with an empty vector for the result, we start with the value of the sum. To get the final result, the sum and carry are added. Additionally, all intermediate additions are calculated using CSAs.

3 The PA1 Squaring Algorithm

PA1 is a modification of the Wu-Lou-Chang squaring algorithm [10]. Wu-Lou-Chang algorithm follows the traditional pencil-and-paper method of squaring. It is to be noted that Wu-Lou-Chang algorithm replaced the term $2*x_i*x_j$ with x_i*x_j to ensure that the intermediate calculations do not exceed two digits. Also, to improve the performance, it retrieved the value of x_i*x_j from a Look-Up Table (LUT). The reader is referred to [10] for more details of the Wu-Lou-Chang algorithm.
We modified this algorithm by making the starting index of the inner loop start at *i+1* instead of *0*. We also are exchanging "$(uv)_b = x_i * x_j$" with "$(tuv)_b = 2 * x_i * x_j$", which reduce the number of iterations from "n^2" to "$(n^2-n) / 2$". The modified algorithm (PA1) is shown in Figure 1.

Algorithm 1 (Squaring Algorithm 1)
Input: $X = (x_{n-1}, x_{n-2}, ..., x_1, x_0)_b$
Output: $S = X^2 = (s_{2n-1}, s_{2n-2}, ..., s_1, s_0)_b$
, Where b is the radix

begin
$Q = (x_{n-1}^2, 0, x_{n-2}^2, 0, ..., 0, x_1^2, 0, x_0^2)_b$
for i=0 to n-1
for j=i+1 to n-1
$(tuv)_b = 2 * x_i * x_j$
$q_{i+j} = q_{i+j} + v$
$q_{i+j+1} = q_{i+j+1} + u$
$q_{i+j+2} = q_{i+j+2} + t$
if $(q_{i+j} \geq b)$ then
$q_{i+j} = q_{i+j} - b$
$q_{i+j+1} = q_{i+j+1} + 1$

if $(q_{i+j+1} \geq b)$ then
$q_{i+j+1} = q_{i+j+1} - b$
$q_{i+j+2} = q_{i+j+2} + 1$

return $S = Q$
end.

Figure 1: The PA1 Squaring Algorithm

PA1 also uses the sum-carry technique discussed earlier. Although this is a simple modification, we found it reducing the number of iterations by half assuming values for

"x_i * x_j" are pre-computed; the proposed algorithm has been implemented in radix-2, radix-4, and radix-8.

- For radix-2: $x_i * x_j = x_i$ *and* x_j
- For radix-4 and radix-8, we need to pre-compute 16 and 64 values, respectively.

4 The PA2 Squaring Algorithm

The PA2 algorithm is based on an equation in [13]. They present an idea and derive a recursive equation. Figure 2 shows the derivation of the equation.

$$\begin{aligned} R &= C^2 \ mod\ N \\ &= (c_{k-1}b^{k-1} + c_{k-2}b^{k-2} + ...+ c_1b^1 + c_0)^2 \ mod\ N \\ &= ((...(c_{k-1}b + c_{k-2})b + ...+ c_1)b + c_0)^2 \ mod\ N \\ &= (C^{(1)}b + c_0)^2 \ mod\ N \\ &= ((C^{(1)})^2b^2 + 2c_0C^{(1)}b + c_0^2) \ mod\ N \\ &\quad \text{where } C^{(1)} = (...(c_{k-1}b + c_{k-2})b + ...+ c_2)b + c_1 \end{aligned}$$

Figure 2: Equation derivation for our squaring algorithm 2

The relationship between consecutive $C^{(i)}$ can be expressed by the following equation:

$$C^{(i)} = \frac{C^{(i-1)}}{b}$$

When we expand the recursive equation (for a 4-bit operand in radix-2, for example), we will get the following:

$$4[4[4 * c_3^2 + 2 * c_2 * \frac{C}{8} * 2 + c_2^2] + 2 * c_1 * \frac{C}{4} * 2 + c_1^2] + 2 * c_0 * \frac{C}{2} * 2 + c_0^2$$

If we divide this equation into two parts, we will get the sum and carry parts discussed earlier.

$$sum = 64 * c_3^2 + 16 * c_2^2 + 4 * c_1^2 + c_0^2$$

$$carry = 16 * 2 * 2 * \frac{C}{8} * c_2 + 4 * 2 * 2 * \frac{C}{4} * c_1 + 2 * 2 * \frac{C}{2} * c_0$$

The final form for sum and carry can be calculated as follows:

$$sum = (c_3^2, 0, c_2^2, 0, c_1^2, 0, c_0^2)_2$$

$$carry = 2 * 2 * \sum_{i=1}^{3} 2^{2(i-1)} * \frac{C}{2^i} * c_{i-1}$$

The general formulas for the sum and carry are shown in Figure 3. Our PA2 Squaring Algorithm is shown in Figure 4.

$$sum = (c_{n-1}^2, 0, c_{n-2}^2, 0, ..., 0, c_1^2, 0, c_0^2)_b$$

$$carry = 2 * b * \sum_{i=1}^{k-1} b^{2(i-1)} * \frac{C}{b^i} * c_{i-1}$$

Figure 3: General sum and carry equations.

In the carry equation in Figure 3, $b^{2(i-1)}*C/b^i$ can't be reduced because C/b^i is an integer division operation not a floating division operation, which means that, to get the correct result, we must divide first, then multiply. The next example shows the difference between dividing then multiplying and multiplying then dividing.

1. If we have C = $(001111)_2 = (15)_{10}$, and we wanted to calculate 8 * C / 4. The correct way to calculate it for our algorithm is:
   - C / 4 = 000011
   - 8 *(C / 4) = $(011000)_2$ = 24
2. If we replace it with 2 * C, or if we multiply first then divide we will get a wrong result.
   - 2 * C = $(011110)_2 = (30)_{10}$
   - Or 8*C = 1111000, (8*C) / 4 = $(011110)_2 = (30)_{10}$

To calculate $\{b * b^{2(i-1)} * C / b^i\}$ efficiently, the following operation is used:

- Suppose R = C
- To calculate R that will be used in current iteration:
- We set Digit $r_{2i} = 0$. this represents $\{b^{2(i-1)} * C / b^i\}$
- $R = R << k$. this represents $R * b$, where $k = log_2 b$.

When we calculate x_i*2*R, the value of R is shifted to the left by one bit, then CSAs are used instead of a multiplier to perform the multiplication operation. For example, to calculate $3*2*R$:

- First we shift R to the left by one bit: $R = R << 1$. This represents $2*R$.
- To calculate $3*R$, we add $R + 2R$, meaning that we add R with a shifted copy R (i.e. $3*R = R + 2R = R + (R << 1)$).
- All the additions are done using CSAs.

Algorithm 2 (Squaring Algorithm 2)
Input: $X = (x_{n-1}, x_{n-2}, ..., x_1, x_0)_b$
Output: $S = X^2 = (s_{2n-1}, s_{2n-2}, ..., s_1, s_0)_b$
, Where b is the radix
$k = log_2 b$

begin
    $Q = (x_{n-1}^2, 0, x_{n-2}^2, 0, ..., 0, x_1^2, 0, x_0^2)_b$
    $R = X$
    for i=0 to n-2
        $r_{2i} = 0$
        $Q = Q + x_i * 2 * R$ (CSA used here)
        $R = R << k$ (i.e. shift left k bits)
    return $S = Q$
end.

Figure 4: The PA2 Squaring Algorithm.

Let's give an example of this algorithm by using it to calculate 231^2 in radix-4. So we have $X = (3213)_4$, $n=4$ (i.e. X has 4 digits), $b=4$.

Note that $(3^2)_4 = (21)_4$, $(2^2)_4 = (10)_4$, $(1^2)_4 = (01)_4$

1. $Q = (21100121)_4$
2. $R = X = (00003213)_4$
3. $i=0$:
    1. $r_0 = (0)_4 \Rightarrow R = (00003210)_4$
    2. $Q = Q + 3*2*R = (21100121)_4 + (00111120)_4 = (21211301)_4$
    3. $R = R << 2 \Rightarrow R = (00032100)_4$
4. $i=1$:
    1. $r_2 = (0)_4 \Rightarrow R = (00032000)_4$
    2. $Q = Q + 1*2*R = (21211301)_4 + (00130000)_4 = (22001301)_4$
    3. $R = R << 2 \Rightarrow R = (00320000)_4$
5. $i=2$:
    1. $r_4 = (0)_4 \Rightarrow R = (00300000)_4$
    2. $Q = Q + 2*2*R = (22001301)_4 + (03000000)_4 = (31001301)_4$
    3. $R = R << 2 \Rightarrow R = (00000000)_4$

    $(31001301)_4 = (3213^2)_4 = (231^2)_{10} = (53361)_{10}$

5 The PMR Modular Reduction Algorithm

To calculate the modulus of X on P, with P and X in the form:

$$P = (p_{k-1}, p_{k-2}, ..., p_1, p_0)_2$$
$$X = (x_{2n-1}, x_{2n-2}, ..., x_1, x_0)_2$$

, where $n \le k$.

The result of $R=X \bmod P$, can be calculated as the following:

$$R = \sum_{i=0}^{n-1} x_i * 2^i \; mod \; P$$

But because $x_i * 2^i < P$, for $0 \le i < k$, we have:

$$x_i * 2^i \; mod \; P = x_i * 2^i \text{ ,for } 0 \le i < k$$

So the final form for the reduction will be:

$$R = X(0 \rightarrow k-1) + \sum_{i=k}^{n-1} x_i * 2^i \; mod \; P$$

We require a pre-calculation of k values, which is denoted to as *PreMod*, as the following:

$$\text{for } i = k \text{ to } 2k-1$$
$$PreMod(i) = 2^i \bmod P$$

When we want to reduce an integer X modulo P, we start by taking the value of the first k bits of X (i.e. $X(0$ to $k-1)$) and we add to it the pre-calculated values with indices i if $x_i = 1$, where $k \le i < 2k$. When we add n k-bit values, we will have a number of additional carry bits equal to:

$$r = \lceil log_2 n \rceil$$

We end up with an extra r bits in the result, so, the algorithm is applied again to the result. This operation is repeated until $r \le 2$. If we end up with $r = 2$, then there will

be three values to add; *X(0* to *k-1)*, x_k*PreMod(k)*, and x_{k+1}*PreMod(k+1)*. In other words:

$$S = X(0 \rightarrow k-1) + x_k * PreMod(k) + x_{k+1} * PreMod(k+1)$$

If we add any 3 *k*-bit numbers, bits *k* and *k+1* will never be 1 at the same time (i.e. bits *k* and *k+1* will be $(10)_2$,$(01)_2$, or $(00)_2$). To show why this happens, let's take the worst case of adding 3 8-bit numbers where all the bits have a value of 1:

S = $(11111111)_2 + (11111111)_2 + (11111111)_2$
= $(1011111101)_2$

From the example we can see that bits 8 and 9 are $(10)_2$. So, the result of the addition of any 3 *k*-bit numbers will have bits *k* and *k+1* equal to $(10)_2$, $(01)_2$, or $(00)_2$.

This means that when we get *r=2,* at the worst case, we will only have to add two *k*-bit numbers; the first one will be $0 \leq X(0 \text{ to } k\text{-}1) < 2P$, and the second will be either x_k*PreMod(k)* or x_{k+1}*PreMod(k+1)* which are less than *P*. So the result of this addition will be $0 \leq sum < 3P$, which means that we may need to subtract *P* from the sum a maximum of 2 times.

If $r = 1$, then we have the same case as with $r = 2$. And if $r = 0$, then we just check if the *sum* is larger than *P*, if so, we subtract *P* to get the final result.

To reduce an integer with a length *n*, with a prime modulo of length *k*, where $k < n < 2k$, we need to apply the algorithm twice if $2 \leq k \leq 7$, three times if $8 \leq k \leq 127$, and four times if $128 \leq k \leq 2^{127}-1$. Figure 5 shows the PMR modular reduction algorithm.

Algorithm 3 (Modular Reduction Algorithm)
Input: $X = (x_{2n-1}, x_{2n-2}, ..., x_1, x_0)_2$
Input: $M = (m_{k-1}, m_{k-2}, ..., m_1, m_0)_2$
Output: $Y = XmodM = (y_{n-1}, y_{n-2}, ..., y_1, y_0)_2$
, Where $k \leq 2n \leq 2k$

begin
    R = X
    for iteration=1 to 4 (this value depends on the length of k)
        $Sum = R(0 \rightarrow (k-1))$
        for i=k to 2n-1
            if ($r_i == 1$)
                $Sum = Sum + PreMod(i)$
        end for
        $R = Sum$
    end for
    if($Sum \geq M$)
        $Sum = Sum - M$
    if($Sum \geq M$)
        $Sum = Sum - M$
    return Sum
end.

Figure 5: The PMR modular reduction algorithm

To give an example of this algorithm at work, we will calculate *17156 mod 131*, this is shown in Figure 6.

$$X = (100001100000100)_2 = (17156)_{10}$$

iteration 1:
$bit(0 \rightarrow 7) => 00000100$
$bit(8) = 1 => 01111101$
$bit(9) = 1 => 01110111$
$bit(14) = 1 => 00001001$
$=> Sum = (100000001)_2 = (257)_{10}$

iteration 2:
$bit(0 \rightarrow 7) => 00000001$
$bit(8) = 1 => 01111101$
$=> FinalResult = Sum = (1111110)_2 = (126)_{10}$

Figure 6: Modular reduction algorithm with *M = 131*, *X = 17156*

6 Results

The hardware design of the proposed and original algorithms was coded using *VHDL*. The tool used for simulation and functional correctness is *ModelSim Xilinx* Edition III v6.2g. The synthesis tool used to get the experimental results was *Xilinx ISE 9.2i.* The target technology was set to *Virtex5* (*xc5vlx50 – ff1153* technology). All algorithms were synthesized on a PC machine with Core 2 Duo E4300 processor 2.7 GHz and 2 GB of RAM.

The proposed squaring algorithms, the Standard Squaring Algorithm, and the Wu-Lou-Chang are implemented in radix-2, radix-4, and radix-8 bases. The Montgomery multiplication algorithm, Barrett's reduction algorithm, and our proposed modular reduction algorithm have been implemented in radix-2 only. All the algorithms have been implemented with input lengths 8, 16, 32, 64, 128, 256, 512, and 1024. The Standard Squaring Algorithm will be denoted to as SA1, and Wu-Lou-Chang's Squaring Algorithm will be denoted to as SA2.

Figure 7 shows the time result obtained from comparing the total time of PMR with Barrett's reduction algorithm.

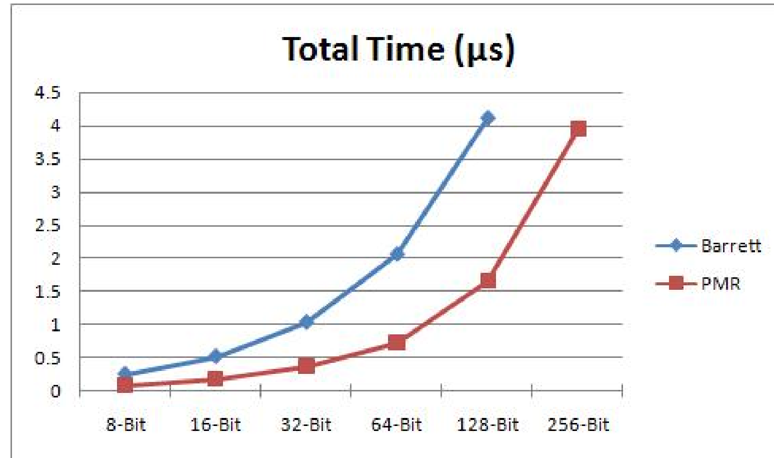

Figure 7: Time results for PMR vs. Barrett's reduction algorithm.

We can see from the previous figure that the PMR modular reduction algorithm has a better time curve than Barrett's Reduction. This makes it a more suitable option for hardware implementation.

Figures 8, 9, and 10 shows the time results obtained from the different possible combinations between the squaring algorithms and the reduction algorithms in radix-2, radix-4, and radix-8, respectively. The combinations that we will cover are: PA1 + PMR, PA1 + Barrett, PA2 + PMR, PA2 + Barrett, SA1 + PMR, SA1 + Barrett, SA2 + PMR, and SA2 + Barrett. The radix-2 results of those combinations will be compared with Montgomery's Multiplication algorithm results.

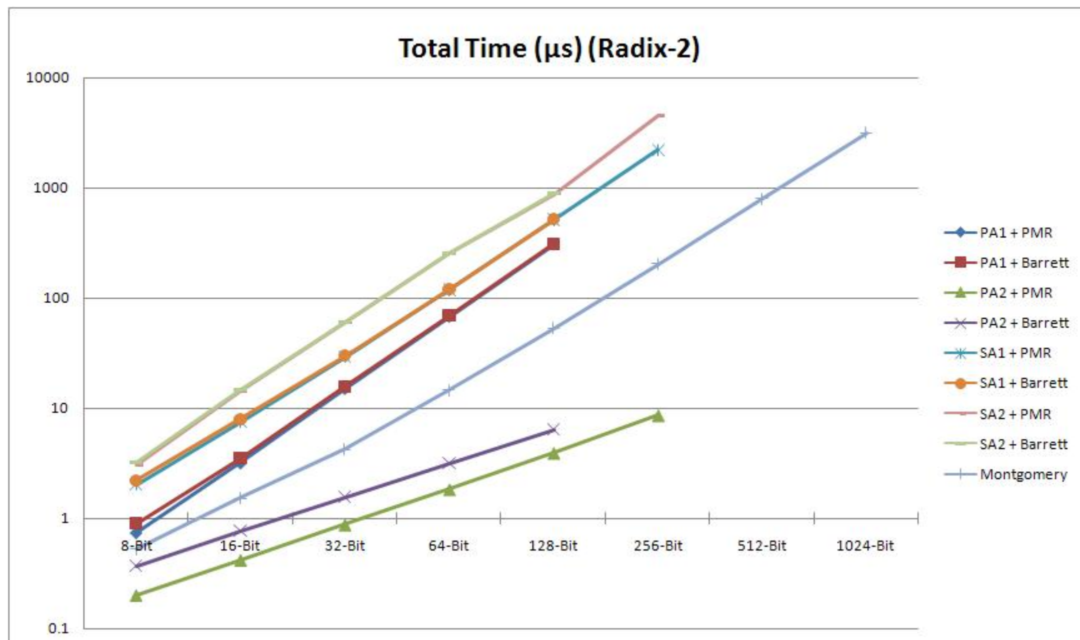

Figure 8: Time Results for the Modular Squaring Algorithms in Radix-2 using a logarithmic scale

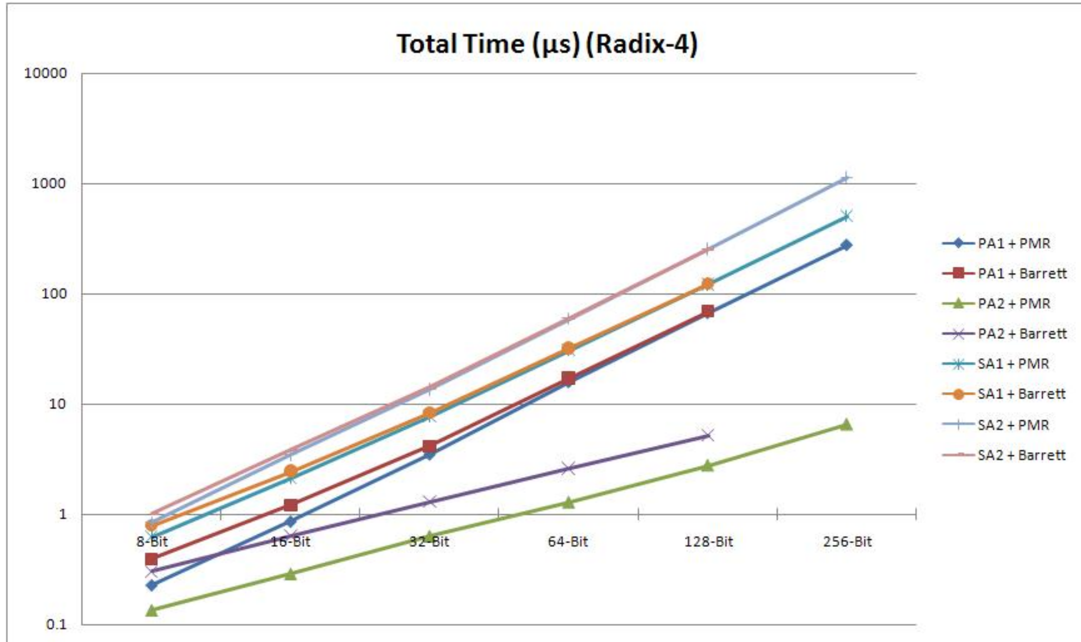

Figure 9: Time Results for the Modular Squaring Algorithms in Radix-4 using a logarithmic scale

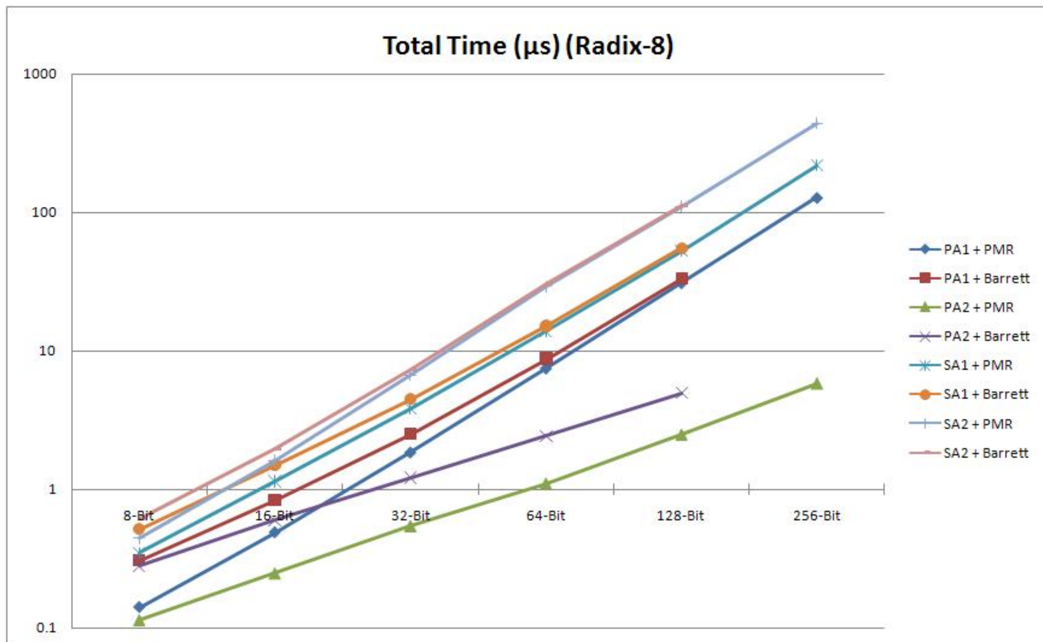

Figure 10: Time Results for the Modular Squaring Algorithms in Radix-8 using a logarithmic scale

7. Conclusion

This work presented two new squaring algorithms, PA1 and PA2, and one modular reduction algorithm, PMR, for GF(p). Modular squaring algorithms play an

important role in Public Key Cryptography (PKC). Some of the most popular public key encryption algorithms such as El-Gamal cryptosystem, Diffie-Hellman key exchange algorithm, RSA, and the Digital Signature Standard. Since modular exponentiation can be broken-down into modular squaring and multiplication operations, the performance of PKC algorithms can be enhanced by using a fast and efficient modular squaring algorithm.

In this paper we implemented the proposed algorithms in hardware, and compared their performance with each other and other already known algorithms. First, we presented the algorithms that we compared our proposed algorithms against. Second, we've presented the proposed algorithms and have shown their derivation. The hardware architecture for the proposed algorithms was also presented.

The algorithms that we compared against consisted of two squaring algorithms, one modular reduction algorithm, and one modular squaring algorithm. The squaring algorithms were the Standard Squaring Algorithm and the Wu-Lou-Chang Squaring Algorithm. The modular reduction algorithm was the Barrett Reduction Algorithm, and the modular squaring algorithm was the Montgomery Multiplication Algorithm.

After that, we've shown the synthesis results for our algorithms for different bit lengths; 8, 16, 32, 64, 128, 256, 512, and 1024 bits. Then the results were compared to the synthesis results of the other algorithms using the same bit lengths. The experimental results were obtained by synthesizing the hardware design for FPGA *Virtex5* chip (*xc5vlx50 – ff1153* technology).

Montgomery's Multiplication algorithm has been discussed and compared with different combinations of squaring algorithms and modular reduction algorithms for radix-2 implementations. Radix-2, 4, and 8 designs were implemented and compared in terms of area and total execution time. The PA1 algorithm which is a modification on the SA2 algorithm has shown good improvement, although it introduces a small increase in area, it gives a good improvement in total execution time. We have shown that the PA2 algorithm, which is and implementation of an equation found in [5], in combination with PMR has the best area and total execution time results, and gives a better performance than Montgomery's Multiplication algorithm in radix-2.

Our proposed modular reduction algorithm (PMR) was shown to have very low area requirements compared to Barrett's reduction algorithm. This is due to the fact that Barrett's algorithm requires two multipliers, where PMR only uses adders and CSAs. Also, PMR has shown a better time performance than Barrett's reduction algorithm. This makes PMR a much better choice to be combined with Squaring algorithms to form efficient modular squaring algorithms.

Acknowledgment

Authors would like to thank both Computer Engineering Departments in Jordan University of Science and Technology, Irbid, Jordan, and King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia, for supporting this research and the fruitful cooperation and collaboration between the universities in the region.

References

1. ElGamal T. A public key cryptosystem and signature scheme based on discrete logarithms. IEEE Trans. - Information Theory 1998; vol. IT-3(4):469-472.

2. Hellman ME, Diffie W. New directions on cryptography. IEEE transactions on Information Theory 1976; 22: 644-654.

3. Adleman L, Rivest RL, Shamir A. A method for obtaining digital signature and public-key cryptosystems. Comm. of the ACM 1978; 21(2): 120-126.

4. Digital signature standard. National Institute of Standards and Technology, Washington; 2000.

5. Karatsuba A, Ofman Y. Multiplication of multi-digit numbers on automata. Soviet Physics Doklady 1963; 7: 595-596.

6. Comba PG. Exponentiation cryptosystems on the IBM PC. J-IBM-SYS-J 1990; 29(4): 526-538.

7. George J. Design and Implementation of High-Speed Algorithms for Public Key Cryptosystems MS. Thesis. South Africa: University of Pretoria; 2005.

8. KOÇ ÇK. High-Speed RSA Implementation. Technical Report TR 201. RSA Laboratories; 1994 November.

9. Hseih PY, Laih CS. An Exception Handling Model and Its Application to the Multiple-Precision Integer Library MS. Thesis; 2003.

10. Wu CL, Lou DC, Chang TJ. Fast modular squaring method for public key cryptosystems. Annual Conference on TAiwan INTERNET (2006TANET). Hualien, Taiwan. 2006 November; F51: 1-10.

11. Barrett P. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. CRYPTO 1986: 311-323.

12. Montgomery PL. Modular multiplication without trial division. Mathematics of Computation 1985; 44(170): 519-521.

13. Hong SM, Oh SY, Yoon H. New modular multiplication algorithms for fast modular exponentiation. Lecture Notes in Computer Science 1996; 1070: 166-177.