A Novel Fast Modular Multiplier Architecture for 8,192-bit RSA Cryposystem

Wei Wang and Xinming Huang
Department of Electrial and Computer Engineering
Worcester Polytechnic Institute, Worcester, MA 01609, USA
{weiwang, xhuang}@wpi.edu

***Abstract*—Modular multiplication is the most crucial component in RSA cryptosystem. In this paper, we present a new modular multiplication architecture using the Strassen multiplication algorithm and Montgomery reduction. The architecture is different from the interleaved version of Montgomery multiplication traditionally used in RSA design. By selecting different bases of 16 or 24 bits, it could perform 8,192-bit or 12,288-bit modular multiplication. The design was synthesized on the Altera's Stratix-V FPGA using Quartus II. It performs one modular multiplication in 2,030 cycles. When operating at 209 MHz, the execution time for an 8K- or 12K-bit modular multiplication is about 9.7 μs.**

***Index Terms*—RSA, Strassen Multiplication, Montgomery Algorithm, FPGA**

I. Introduction

Information security plays a more crucial role in the computer and communication systems in the age of Internet and smartphones. Owing to the safety of key distribution, public key cryptosystem are more popular than traditional secret key cryptosystem such as the data encryption standard [1]. The Rivest–Shamir–Adelman (RSA) [2] system is one of the most widely used public key cryptography systems. The RSA operation is a modular exponentiation and its security level relies on that there are no effective procedures or algorithms that can factorize large integers within a short time period using current computer technology. Now the size of modulus is at least 1,024 bits to provide a good level of security. As the Moore law continues driving the computer technology, the key size of 1,024 bits can be broken. It becomes necessary to upgrade the key size to 2048, 4096 or even 8192 bits to provide a higher level security. It is hard to achieve a good throughput rate without the use of hardware acceleration because of computing complexity.

RSA cryptosystem recursively performs modular multiplications to finish one modular exponentiation. The large-number modular multiplication, which accounts for the most of computing complexity, is a crucial part in the RSA cryptosystem. As a result, the performance of a RSA system relies on the throughput rate of the modular multiplication. Traditionally, the interleaved Montgomery's multiplication algorithm [3] is used to speed up the modular multiplication calculation. For the small size RSA, the interleaved Montgomery modular multiplication algorithm is a good choice that can achieve high performance at a low cost of hardware [4] [5]. However, the interleaved version generates long carry chains, which impedes the throughout rate of the RSA cryptosystem. Various approaches to relax the problem of long carry propagation fall into two categories. In the first approach, the carry-save addition is used to keep the intermediate results in carry-save form to avoid carry propagation [6] [7] [8]. However, this approach needs extra work. For instance, the carry-save form of an operand needs to be converted back to the binary representation at the end of each modular multiplication [6] [7], resulting in longer computation time. Also the performance result depends on the operand length. Some work such as [8] uses 5-to-2 carry-save addition (CSA) and a three-level adder tree to reduce the operation of format conversion. But this method increases the critical path delay by applying extra control logic and multiplexers. The second approach employs the systolic array to solve the problems mentioned in the first approach [9] [10] [11]. But it increases the hardware complexity and latency. A 8192-bit RSA cryptoprocessor is designed based on systolic array [11]. But it still takes about 130 μs to calculate one modular multiplication when operating at 252.84 MHz on an FPGA.

In this paper, we take a completely different approach to speed up the process of modular multiplication by combining the Strassen algorithm [12] and Montgomery reduction [3]. First, a fast large-number multiplier is designed based on Strassen's algorithm. A memory-based in-place architecture is adopted for 1024-point finite-field fast Fourier transform (FFT) processor used in the multiplier. A radix-16 butterfly unit is repeatly used for four times to replace one radix-32 butterfly computation. The radix-16 unit is simplified to only additions and shift operations by employing a special Solinas prime. After the two large multiplicands are multiplied by the large-number multiplier, Montgomery reduction is applied for modular reduction.

The rest of the paper is organized as follows: Section II gives a brief introduction of Strassen multiplication algorithm; Section III presents the Montgomery modular multiplication; Section IV shows the VLSI architecture of the modular multiplication; Section V gives the experimental results of FPGA implementation followed by the conclusions in Section VI.

II. Strassen's Multiplication and Finite Field FFT

A. Strassen's Multiplication

Strassen described a multiplication algorithm based on FFT in [12]. It breaks each multiplicand into samples with each

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

sample the same number of bits. For example, we can break the large number into 16-bit samples and $\beta = 2^{16}$ is referred as the base. Briefly the Strassen's FFT algorithm can be summarized as follows [13]:

1) Decomposition and FFT: Break large numbers A and B into a series of samples $a(n)$ and $b(n)$ and compute FFT of the $a(n)$ and $b(n)$;
2) Component-wise Product: Compute the component-wise product of the FFT results, set $C = A \odot B$: $C[i] = FFT(A)[i] * FFT(B)[i]$.
3) IFFT: Compute the inverse FFT of $C[i]$, set $c(n) = IFFT(C)$.
4) Resolve the carries: when $c[i] \geq \beta$, set $c[i+1] = c[i+1] + (c[i] \operatorname{div} \beta)$, and $c[i] = c[i] \bmod \beta$.

Strassen algorithm could use complex-numbers arithmetic or modulo arithmetic to perform the calculation. But the algorithm over complex requires rounding operations, which makes it very difficult to get an accurate result. Also, floating-point operations consume significantly more hardware resources and is much slower than fixed-point operations. Instead, we choose to perform the computation in the finite field $\boldsymbol{Z}/p\boldsymbol{Z}$, with prime p. The computation in a finite field requires three operations: modular addition, modular subtraction and modular multiplication. By selecting a proper prime p (p =0xFFFFFFFF00000001) [14], the modular multiplication in the finite field can be computed rapidly. The prime p has special identities, such as ($2^{192} \bmod p = 1$), ($2^{96} \bmod p$ = -1) and ($2^{64} \bmod p = 2^{32} - 1$), which makes it support highly efficient modulo multiplication [14].

In our implementation, we choose the base b to be 16 or 24, so every sample has 16 or 24 bits. For a total of 512 samples, we can perform 8192-bit or 12,288-bit multiplication. As we know, the multiplication of two numbers is similar to the cyclic convolution result of two signals each with 512 samples. Typically, cyclic convolution involves "zero padding" and the result contains approximately twice many samples as that of the input signal. Thus, a high-speed 1024-point finite-field FFT processor is proposed in this design.

III. Montgomery Modular Multiplication

The most popular algorithms for modular reduction are the Montgomery reduction [3] and the Barrett reduction algorithms [15]. For the reason as stated in previous section, the interleaved Montgomery algorithm generates a long carry chain. If we use large residue without long carry chains, the Montgomery reduction has the similar complexity as the Barrett reduction. Since it is hard to design the control logic for Barrett algorithm, we choose the Montgomery method in the hardware design.

We employ the Strassen algorithm for the calculation of the three large-number multiplications in the Montgomery multiplication as shown in **Algorithm**1. Multiplying two numbers is equivalent to the component-wise product of the FFT results of two signals in the FFT domain. Thus, we can precompute the FFTs of M and M' to reduce the computational complexity.

Algorithm 1 Montgomery Multiplication Using Strassen FFT Algorithm

Procedure Montgomery(X,Y, M): $c = XYR^{-1} (\bmod M)$
Precomputation: $n = \lceil log_2^M \rceil$, $R = 2^n$,$M' = -M^{-1} (\bmod R)$
1. $T \leftarrow IFFT(\ FFT(X)\ \odot\ FFT(Y)\ )$;
2. $t\ \leftarrow T \bmod R$;
3. $U \leftarrow IFFT(\ FFT(t)\ \odot\ FFT(M')\ )$;
4. $u\ \leftarrow U \bmod R$;
5. $W \leftarrow IFFT(\ FFT(u)\ \odot\ FFT(M)\ )$;
6. $C \leftarrow T + W$;
7. $c\ \leftarrow C/R$;
8. If $c \geq M$ then $c \leftarrow c - M$, end if
end procedure.

IV. VLSI Design of the Modular Multiplication

As described in Section II-A, the finite-field FFT/IFFT is a key component for the FFT-based Strassen's multiplication algorithm. The memory-based in-place FFT architecture allows to store the intermediate results into the same memory where the input data are read from. As a result, it minimizes the memory usage while still produces high throughput [16]. In this work, we use the memory-based in-place FFT architecture and radix-32 butterfly computation. As a result, the 1024-point FFT is implemented using two stages of 32-point FFT. Using in-place memory-based design, these two stages are computed sequentially using the same hardware unit and memory space. The radix-16 butterfly unit can be recursively used four times to complete one radix-32 FFT computation. Therefore, we employ only one radix-16 butterfly unit instead of the much larger radix-32 unit to further reduce hardware cost.

A. Radix-16 FFT Unit

With the chosen prime p, 64 is a 32^{th} root, 4096 is a 16^{th} root, 4096^2 is a 8^{th} root and so on. This means that 32-point, 16-point and 8-point FFTs can be done with shift operations rather than costly multiplications. The 16-point FFT can be simplified as (1), since $4096^{16} \bmod p = 2^{192} \bmod p = 1$. For 192-bit operations, any carry-out bit can be simply routed back as a carry-in bit. This special property is useful for hardware design. The multiplications in 16-point FFT can be accomplished by circular shifting operations. Instead of performing modular operations after each addition, we add all 16 numbers first and perform the modular reduction only once to obtain the final result. Since $2^{192} \bmod p = 1$, only 192 bits needs to be kept during the additions.

$$X(k) = \sum_{n=0}^{15} x(n) 2^{12 \cdot nk \% 192} \bmod p \quad (1)$$

$$x(n) = \frac{1}{16} \sum_{k=0}^{15} X(k) 2^{(192 - 12nk) \% 192} \bmod p \quad (2)$$

For 192-bit addition, traditional carry-ripple adder would generate a long carry chain and slow down the clock speed

considerably. So we choose carry-save adders that support high-speed design. The diagram of a processing element (PE) in radix-16 unit is shown in Fig. 1. At every cycle, 16 samples are read into the PE, shifted by the shifter and accumulated by the carry-save adders. At the end, a reduction unit performs modulus p operation and converts the 192-bit result back to 64-bit. Again, the special identities mentioned above are employed to simplify the calculation as shown in (3), where a, b, c, d, e and f are 32-bit components of the 192-bit result. The radix-16 unit has 16 processing elements. At each clock cycle, the radix-16 unit takes 16 data inputs and outputs the 16-point FFT results after a few cycles of pipeline delay.

$$\begin{aligned} z &= 2^{160}a + 2^{128}b + 2^{96}c + 2^{64}d + 2^{32}e + f \qquad (3) \\ &= (2^{32}e + f) + (2^{32}d + a) - (2^{32}b + c) - (2^{32}a + d) \end{aligned}$$

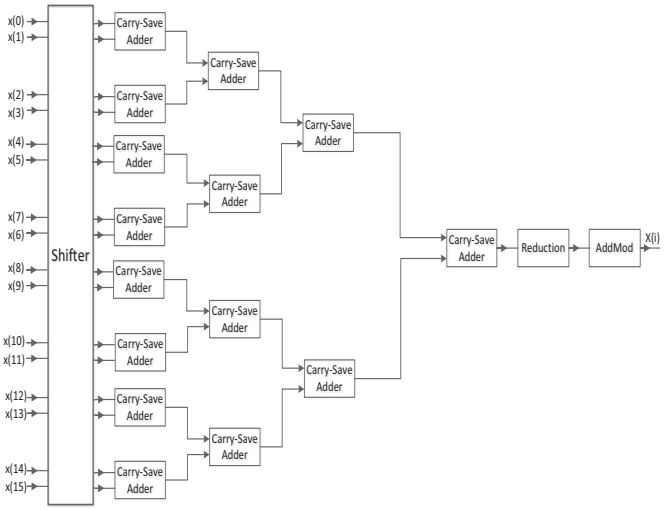

Figure 1. Diagram of One Processing Element.

B. Resolve the Carries

We take the 8192-bit Strassen's multiplier as an example to explain the process of resolving carries. Each 8192-bit multiplicand is first decomposed into 512 groups of 16-bit numbers. Then each 16-bit number is then extended to a 64-bit data sample. The multiplication results are expected to be 1024 groups of 16-bit numbers, or up to 16,384 bits. Following the Strassen's algorithm with 1024-point FFT, the IFFT output are 1024 samples of 64-bit data. The resolve carries unit is to obtain the actual 16,384 results from the IFFT output data.

Since each data is supposed to be 16-bit, each 64-bit data from IFFT output are actually overlapped 48-bit with the next one. For a structural design, we decompose the 64-bit number into four blocks of 16-bit words. The alignment among the words are illustrated as in Fig. 2.

Recall that the IFFT module outputs 32 data samples per clock cycle in operation. A total of 1024 data are output in 32 consecutive cycles. Therefore, we have to resolve the carriers in time to match the pipeline throughput. Apparently the traditional carry-ripple adder is too slow to add the numbers in a column. A hierarchical carry-look-ahead scheme for large-number addition as proposed in [17] is applied here to add the the numbers in parallel. For a high-speed design, we use a four-stage pipeline design for the resolving carries unit. Overall by using the carry-look-ahead scheme and four stage pipeline, the resolve carries unit can meet the throughput of the FFT/IFFT processor at high clock speed.

If we want to do a 12,288-bit multiplication, each multiplicand is first decomposed into 512 words each with 24 bits. Similarly, each 24-bit number is extended to a 64-bit data sample. After processed by FFT and IFFT, the 64-bits data samples are extended into 72-bit format as three blocks of 24-bit numbers. Then we use the similar parallel and hierarchical carry-look-ahead scheme to add the numbers in each column.

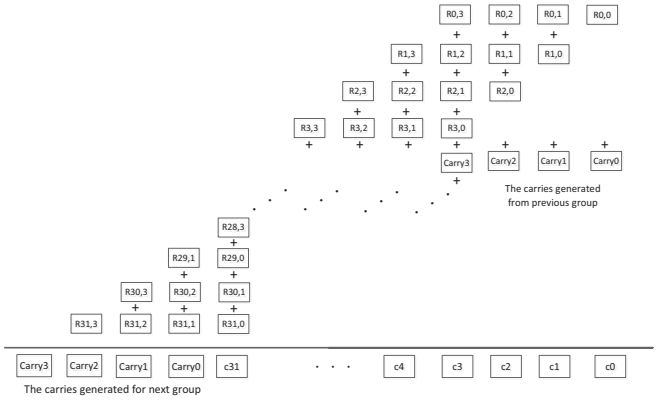

Figure 2. Two-stage pipeline carry resolving unit.

C. The Architecture for Modular Multiplication

Memory-based in-place scheme is used for the FFT design. The 1024-point FFT can be decomposed into 2 stages of 32-point FFT. At each stage, a total of 1024 samples are processed through a radix-32 butterfly unit. The radix-16 unit can be recursively used four times to complete one radix-32 butterfly computation. A group of 32 input samples are read from memory, permutated into a proper order by the Interchange Unit, fed to the radix-16 unit to process four times for one radix-32 butterfly computation, modular multiplied by the twiddle factors stored in ROMs, permutated again by the Interchange Unit, and written back to the memory. The memory needs to be partitioned into 32 banks with 32 words in each bank. An in-place memory addressing scheme can be derived to ensure there is no memory access conflict in reference to [16][18]. The data needs to be read from and written to the memory concurrently, so dual-port SRAMs shown in Fig. 3 are used to store two multiplicands X and Y.

One radix-16 unit are used both for FFT and IFFT to multiply, for instance, X by Y. In the first stage of FFT or IFFT, the 8 units of 64-bit ModMuls are used to multiply the

processed samples with twiddle factors. In the second stage of FFT, the same 8 units of 64-bits ModMuls can be reused to multiply $FFT(X)$ and $FFT(Y)$ for component-wise product to calculate the product of X and Y, or multiply $FFT(X)$ and $FFT(X)$ to obtain X^2. After the IFFT, a group of 32 data are fed into the resolve carries unit.

The FFT forms of M' and M are precomputed and stored in the single-port SRAMs to reduce the computation complexity. The large-number addition unit shown in Fig. 3 uses the same hierarchical carry-look-ahead scheme as in resolve carries unit. The large number addition performs the operation of $Step6$ in **Algorithm**1. The comparison in $Step\,8$ is actually a large-number subtraction. The 2's complement of M is precomputed and stored in the SRAMs so the large-number addition unit in Fig. 3 is reused for the subtraction.

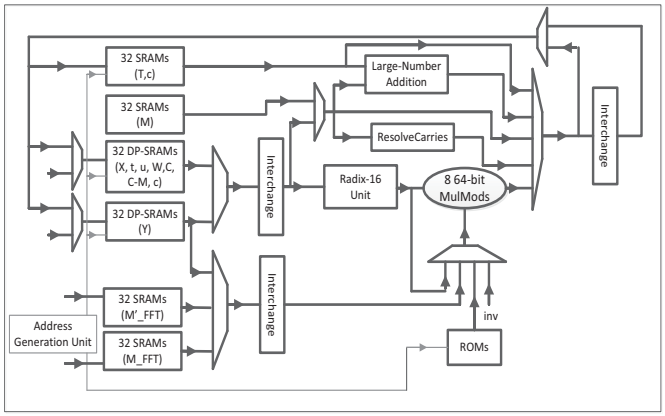

Figure 3. The Architecture for Modular Multiplication

V. Experimental Results

The hardware implementation is designed and coded in System Verilog. The design is then synthesized using Altera Quartus-II synthesize tool. After place and route, the design is implemented on Altera's Stratix-V 5SGSMD8N1F45I2 FPGA. The resource utilized by the modular multiplication are listed in Table 1.

Table I
TABLE I. Synthesis result and comparison

Logic Utilization	Our Design	8192-bit RSA [11]
Combinational ALU	214,321	32,262
Dedicated Logic Register	86,562	82,023
DSP Blocks	72	—-
Block Memory bits	474,010	—-
Frequency (MHz)	209.12	252.84
Cycles per Modular Mult	2030	32776

Table 1 presents the synthesis results of the 8,192-bit modular multiplier. The design can also support 12,288-bit modular multiplication if the base is set to 24 bits. The FPGA Operation Maximum Frequency (OMF) of the modular multiplier is 209.12 MHz. It takes 9.7 μs to complete one modular multiplication when the design operates at 209 MHz. The proposed modular multiplication is about 13.4 times faster than the RSA co-processor reported in [11].

VI. Conclusions

In this paper, a novel and fast modular multiplication architecture is presented for RSA with large key sizes. Instead of using the well-known interleaved version of Montgomery multiplication, we combined the Strassen multiplication and Montgomery reduction for the modular multiplier design. The design support both 8K- and 12K-bit modular multiplication. To the best of our knowledge, it is the first design that can support 12K-bit modular multiplication for RSA. The design can complete one 8K- and 12K-bit modular multiplication in 2,030 cycles, which is an order of magnitude faster than the existing design.

References

[1] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[3] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[4] G. D. Sutter, J.-P. Deschamps, and J. L. Imaña, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 7, pp. 3101–3109, 2011.

[5] M.-D. Shieh, J.-H. Chen, H.-H. Wu, and W.-C. Lin, "A new modular exponentiation architecture for efficient design of RSA cryptosystem," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 9, pp. 1151–1161, 2008.

[6] T.-W. Kwon, C.-S. You, W.-S. Heo, Y.-K. Kang, and J.-R. Choi, "Two implementation methods of a 1024-bit RSA cryptoprocessor based on modified Montgomery algorithm," in *The 2001 IEEE International Symposium on Circuits and Systems, 2001.*, vol. 4. IEEE, 2001, pp. 650–653.

[7] A. Cilardo, A. Mazzeo, L. Romano, and G. Saggese, "Carry-save Montgomery modular exponentiation on reconfigurable hardware," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 3*. IEEE Computer Society, 2004, p. 30206.

[8] C. McIvor, M. McLoone, and J. V. McCanny, "Modified Montgomery modular multiplication and RSA exponentiation techniques," in *Computers and Digital Techniques, IEE Proceedings-*, vol. 151, no. 6. IET, 2004, pp. 402–408.

[9] A. Fournaris and O. Koufopavlou, "A new RSA encryption architecture and hardware implementation based on optimized Montgomery multiplication," in *IEEE 2005 International Symposium on Circuits and Systems*. IEEE, 2005, pp. 4645–4648.

[10] J.-H. Hong and C.-W. Wu, "Cellular-array modular multiplier for fast RSA public-key cryptosystem based on modified Booth's algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 3, pp. 474–484, 2003.

[11] C. P. Rentería-Mejía, V. Trujillo-Olaya, and J. Velasco-Medina, "Design of an 8192-bit RSA cryptoprocessor based on systolic architecture," in *2012 VIII Southern Conference on Programmable Logic (SPL)*. IEEE, 2012, pp. 1–6.

[12] A. Schönhage and V. Strassen, "Schnelle multiplikation grosser zahlen," *Computing*, vol. 7, no. 3, pp. 281–292, 1971.

[13] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using gpu," in *2012 IEEE Conference on High Performance Extreme Computing (HPEC)*. IEEE, 2012, pp. 1–5.

[14] N. Emmart and C. Weems, "High precision integer multiplication with a gpu using Strassen's algorithm with multiple FFT sizes," *Parallel Processing Letters*, vol. 21, no. 3, p. 359, 2011.

[15] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology, CRYPTO-86*. Springer, 1987, pp. 311–323.

[16] L. Johnson, "Conflict free memory addressing for dedicated FFT hardware," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 5, pp. 312–316, 1992.

[17] N. Emmart and C. Weems, "High precision integer addition, subtraction and multiplication with a graphics processing unit," *Parallel Processing Letters*, vol. 20, no. 4, pp. 293–306, 2010.

[18] H. Lo, M. Shieh, and C. Wu, "Design of an efficient FFT processor for DAB system," in *The 2001 IEEE International Symposium on Circuits and Systems, 2001.*, vol. 4. IEEE, 2001, pp. 654–657.