An Asynchronous Modulo Multiplier for Cryptosystems

M. Mahmoud and Alaaeldin Amin***

King Fahd University of Petroleum and Minerals/Computer Engineering Department, Dhahran, Saudi Arabia

* muhammad@ieee.org

** amin@ccse.kfupm.edu.sa

ABSTRACT

A new algorithm for asynchronous modulo multiplication has been devised. A locally synchronous globally asynchronous hardware implementation of the algorithm has been modeled in VHDL. Results show that the developed hardware has a superior AT cost for use with GF(P) elliptic curve cryptosystems.

1. INTRODUCTION

With the increased use of business and commercial transactions through public communication channels and high speed networks, data encryption has become a major requirement to ensure secrecy of such transactions. Encryption speed is a major performance measure in communication cryptosystems. The speed of the encryption/decryption process is a direct function of the complexity of the encryption algorithm, the speed of the underlying hardware arithmetic unit and the implementation technology. Public-key cryptosystems, e.g. RSA **[1]** and Elgamal **[2]**, are based on modulo exponentiation of large numbers. Being the basic operation in these systems, efficient modulo multiplication algorithms and circuitry have been the subject of many research works, e.g. **[3]** and **[6]**.

This work investigates the use of asynchronous techniques for the design of an efficient modulo multiplier. With the large size operands commonly used in cryptosystems, using array or parallel multipliers would require prohibitively large areas. Instead, sequential multipliers are employed in this work. Since sequential multipliers use repeated add and shift operations, an asynchronous implementation can significantly improve the speed at a modest increase in area. The speed of an asynchronous adder is O(Log n) on the average **[7]** compared to the O(n) speed of carry-propagate adders. We have used asynchronous event logic based on transition signaling **[8]** where signal transitions are used as control events.

The multiplication process consists of a number of add and shift operations with addition requiring much more time than the shift operation. In addition to using an asynchronous adder with O(Log n) average speed **[9]**, a number of other measures were adopted to further improve the overall speed of the system. For one, the developed algorithm uses radix-4 which retires two bits per iteration instead of one. For another, multiplier recoding as a signed-digit number **[10]** is used to allow skipping over chains of zeros as well as chains of ones which results in a considerable reduction in the number of add operations, and hence a significant speed improvement.

The rest of the paper is organized as follows. Section 2 shows a fast efficient modular multiplier algorithm and its data flow. Section 3 presents some implementation and performance issues. Finally, the paper is concluded in section 4.

2. THE BASIC ALGORITHM

It is required to compute $\boldsymbol{P = X \times Y \bmod N}$, where the modulus N, the multiplicand X and the multiplier Y are k-bit unsigned numbers. Typically, N is a very large odd number, i.e. generally $N_{k-1} = 1$ and $N_0 = 1$. The developed algorithm uses radix 4, but may be extended to higher radixes as well. In addition, a Booth-like recoding of the multiplier (Y) into an equivalent signed digit representation is performed. This generally results in increasing the number of 0's and reducing the number of 1's and -1's leading to a reduction in the number of add/subtract operations thus improving the overall speed.

Starting from the most significant digit, the algorithm scans one multiplier digit (2bits) plus one look-ahead bit ach iteration. Even though the algorithm requires a 2-bit left shift per iteration, proper scaling of the result restricts all *addition* operations to be only *k*-bit additions. Multiplier recoding is based on Table 1.

Table 1: Left-to-Right Multiplier Recoding.

Scanned Multiplier Digit $y_i\ y_{i-1}$	Look Ahead Bit y_{i-2}	Action
00	0	Shift 2-bits
00	1	+1 X; Shift 2-bits
01	0	+1 X; Shift 2-bits
01	1	+2 X; Shift 2-bits
10	0	-2 X; Shift 2-bits
10	1	-1 X; Shift 2-bits
11	0	-1 X; Shift 2-bits
11	1	Shift 2-bits

a. ***Initialization***:
P ← 0 {P is left padded with 3 bits}
where k, the number of bits in N, is assumed to be even
Left pad Y by two bits, i.e. $Y_{k+1}=Y_k=0$.
Compute (N-X), 3N and 5N. i=k+1

b. ***Shift, Recode and Add***:
WHILE *i >0 DO*
P ←4P;
CASE *P_{k+2} Y_i Y_{i-1} Y_{i-2}* ***IS***
X000, X111 : skip
0001, 0010 : P ← P-(N-X)
0011 : P ← P-2(N-X)
0100 : P ← P-2X
0110, 0101 : P ← P-X

1110, 1101 : P ← P+(N-X)
1100 : P ← P+2(N-X)
1011 : P ← P+2X
1001, 1010 : P ← P+X
END CASE

c. ***Scaling***:
CASE *P_{k+2} P_{k+1} P_k P_{k-1} N_{k-2}* ***IS***
000XX, 111XX : skip
001XX: P ← P-2N
010X1 : P ← P-3N
010X0, 011X1 : P ← P-4N
01100 : P ← P-5N
01110 : P ← P-6N

110XX: P ← P+2N
101X1 : P ← P+3N
101X0, 100X1 : P ← P+4N
10010 : P ← P+5N
10000 : P ← P+6N
END CASE
i=i-2
END WHILE

d. ***Correction***:
IF *P>N* ***THEN***
P ← P-N
ELSEIF *P< 0* ***Then***
P ← P+N
ENDIF

Figure 1. The Modulo Multiplication Algorithm

To compute ***P = X×Y mod N***, the product register P is left padded with three bits to accommodate the left shift operation by one digit (2bits) plus the sign bit. The multiplier is also left padded with 0's for proper recoding. The algorithm consists of four major phases (Figure 1):

- The *initialization phase*, where the values (N-X) and 3N are pre-computed
- The *shift, recode and add phase*. In this phase, the product register is shifted left by one digit and the proper multiple of X is added or subtracted from P based on the current recoded multiplier digit and the sign of P, i.e. P_{k+2}. It should be noted that instead of subtracting (adding) X, (N-X) may be equivalently added (subtracted). The performed operation, i.e. adding X or subtracting (N-X), is chosen to oppose the current sign of P so as to reduce the chance of overflow. For example, according to Table 1, if y_i y_{i-1} y_{i-2}=001 then X should be added to P. In this case, if P is negative, we add X to P, but if P is positive we subtract (N-X) from P. This requires pre-computation and storage of the value (N-X).
- The *scaling phase*. Here, the proper multiple of N is added / subtracted to guarantee that register P will not overflow in the subsequent left shift operation (P = 4P). Thus, the objective of this phase is to make the 3-leftmost bits of register P have the same value, i.e. 000 or 111.
- The *correction phase*. After all k/2 iterations, the resulting value of P may need correction which is guaranteed to require no more than one add/subtract operation.

Example: Compute *(9×11) Mod 13*

Initialization:
X=1001, Y=001011, N=1101,
i =5, (N-X) =0100,
3N=100111, 5N=10000001,
and P= 000_0000.
Shift, Recode, and Add
P=P-(N-X)
P=111_1100
Scaling
Skip
i =3
Shift, Recode, and Add
P=111_0000 (P=4P)
P=P+(N-X)= 111_0100

Scaling
Skip
i =1
Shift, Recode, and Add
P=101_0000 (P=4P)
P=P+(N-X)
P=101_0100
Scaling
P=P+3N= 111_1011
i = -1
Correction
P=P+N= 000_1000 = **8**

The data flow of the algorithm is illustrated in Figure 2 and the multiplier data path is shown in Figure 3.

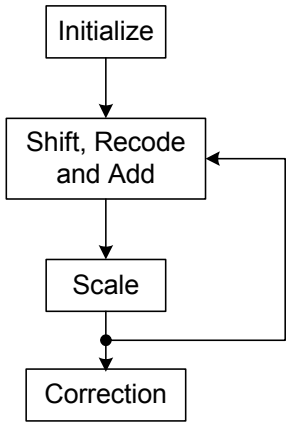

Figure 2. Data flow of the Modular Multiplication Algorithm

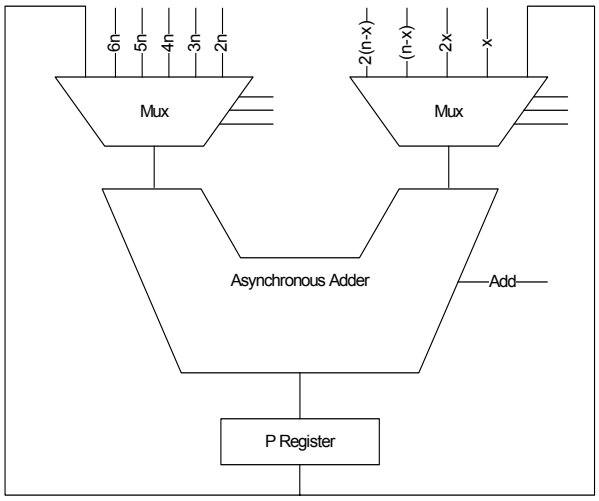

Figure 3: Asynchronous Multiplier Data Path.

3. IMPLEMENTATION ISSUES AND PERFORMANCE

The adopted asynchronous system implementation of the above algorithm is based on event control logic [8]. The implementations were modeled using VHDL. The select module [8] was used to implement decisions (IF statements). Loops were implemented using a merge element with the loop condition checked through a select module [8] as shown in Figure 4.

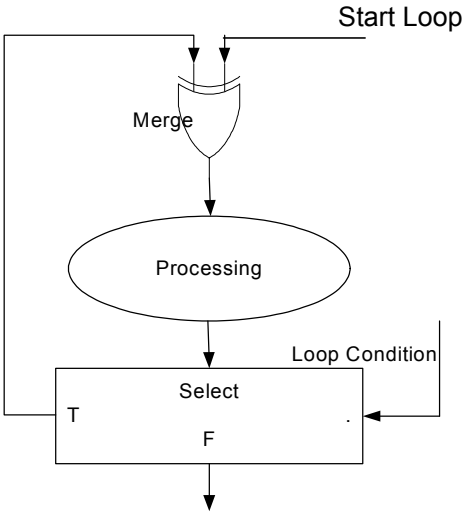

Figure 4: Loop Implementation.

3.1. Globally Asynchronous Locally Synchronous.

For area efficiency, the implementation followed a *Globally Asynchronous Locally Synchronous* (GALS) strategy with counters and registers implemented as clocked synchronous elements. The local clock input of a register or counter receives a single clock pulse whenever a signal event is received at the input request line. This is achieved through the use of edge detection and one shot circuitry as shown in Figure 5.

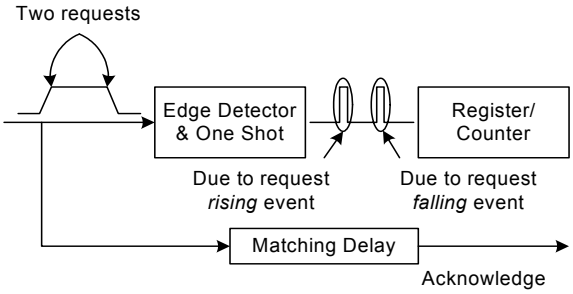

Figure 5: Clock Pulse Generation for GALS Designs.

3.2. Algorithm Cost.

Table 2 shows the area cost taking one register bit as a reference unit. The total area cost is 10 k or O(K).

Table 2: Hardware Area Cost .

Count	Module	Area Estimate
6	Register	6k
1	Adder	1.5k
1	Mux	2k

The algorithm FOR LOOP is executed k/2 times. On the average addition/subtraction is performed only 75% of the time in the *shift, recode and add* phase. Likewise,

in the scaling phase, addition/subtraction is performed only 75% of the time on the average.Therefore, since the average addition/subtraction time for a self-timed asynchronous adder is $O(Log\ k)$ [9], the algorithm overall area-delay (AT) cost is $O(k^2\ log\ k)$. Study of the cost *constant factor* showed that the cost of this multiplier is superior to other multipliers for k values less than 200 bits. This makes such implementation superior for elliptic curve cryptosystems [11], as well as Residue Number Systems (RNS).

4. ACKNOWLEDGMENT

The authors would like to acknowledge the support of the Computer Engineering Department of King Fahd University of Petroleum and Minerals (KFUPM) and King Abdul-Aziz City of Science and Technology (KACST) for support.

5. REFERENCES

[1] R. L. Rivest , A. Shamir , L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems” Communications of the ACM, Vol.:21, No. : 2, pp 120-126, Feb. 1978.

[2] Elgamal, T. “A public key cryptosystem and a signature scheme based on discrete logarithms” IEEE Transactions on Information Theory, Vol.: 31 No.: 4, pp 469-472, Jul 1985.

[3] H.Orup and P. Kornerup, “A high-radix hardware algorithm for calculating the exponential M/sup E/ modulo N," in Proc. IEEE 10th symp. Comput. Arithmetic, June 1991, pp51-56.

[4] N. Takagi, “A radix-4 modular multiplication hardware algorithm efficient for iterative modular multiplications” 10th IEEE Symposium on Computer Arithmetic. Proceedings., pp 35 -42, 26-28 Jun 1991.

[5] N. Takagi, "A radix-4 modular multiplication hardware algorithm for modular exponentiation" IEEE Transactions on Computers, Vol. 41 No. 8, pp. 949 – 956, Aug. 1992

[6] Takagi, N. and Yajima, S.” Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem” IEEE Transactions on Computers, Volume: 41 Issue: 7 , pp 887 -891, Jul 1992.

[7] G. W. Reitwiesner, “The Determination of Carry Propagation Length of Binary Addition,” IRE Transactions on Electronic Computers, pp 35 – 38, 1960.

[8] I. E. Sutherland, "Micropipelines", Communications of the ACM, Vol. 32 No. 6, pp. 720 – 738, June. 1989.

[9] Alaaeldin Amin and Feras Maadi, “Double-rail encoded self-timed adder with matched delays” to appear in the proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems, Dec. 2003, (ICECS-2003).

[10] Behrooz Parhami, "COMPUTER ARITHMETIC Algorithms and Hardware Design" Oxford, Oxford University Press, 2000.

[11] Standards for Efficient Cryptography Group/Certicom Research, SEC 2: Recommended Elliptic Curve Cryptography Domain Parameters, Version 1.0, 2000.