ISCAS 2000 - IEEE International Symposium on Circuits and Systems, May 28-31, 2000, Geneva, Switzerland

A PROGRAMMABLE PROCESSOR FOR CRYPTOGRAPHY

Sukumar S. Raghuram and Chaitali Chakrabarti

Department of Electrical Engineering, Arizona State University
Tempe, AZ 85287-5706, USA
sukumar@asu.edu and chaitali@asu.edu

ABSTRACT

Cryptography has numerous applications in today's world, the most prevalent one being transferring messages safely over the network. Cryptographic algorithms are either implemented in software on a general-purpose processor or in hardware on an application-specific processor. While the software implementations tend to be time consuming, the hardware implementations are too specific and cannot even support small modifications. In this paper, a programmable architecture that can handle a large number of algorithms including DES, RSA, Blowfish, SAFER, et cetera has been developed. The architecture consists of addition, subtraction, modular multiplication, exponentiation and XOR units and thus can support a majority of the cryptographic algorithms. A high data rate is achieved by applying loop unrolling to the Montgomery algorithm that is used for modular multiplication and exponentiation. The differences in the number of bits, key length, and sequence of operations is handled by the microprogrammed control unit. A VHDL model has been developed and synthesized using AutoLogic II from Mentor Graphics. The results show a frequency of operation of 77 Megahertz and an area of 23,000 "Optimization COST" units.

1. INTRODUCTION

Cryptography is the science of keeping secrets. While traditionally, encryption was used to transfer confidential matter amongst the highest government agencies, today, it is an important aspect of our everyday life. This is because of the importance of transferring messages such as credit card numbers, bank transactions, etc. safely over the network. Cryptography consists of two basic functions, encryption and decryption [1]. A known text called the plaintext, is hidden by means of encryption. Decryption is conversion of the encrypted text back to the plaintext. Encryption and decryption are carried out using a secret code, known as key. The sender of the message uses the key to encrypt a message and the receiver uses the key to decrypt it. The key length differs from one algorithm to another and directly depends on the application for which it is being used. The longer the key, the more secure the system is.

Cryptographic algorithms can be broadly classified into symmetric and public-key based algorithms. Symmetric algorithms are the algorithms where the encryption key can be calculated from the decryption key and vice versa. They are further divided into stream ciphers, which operate on the plaintext a single bit at a time, and block ciphers, which operate on the plaintext in groups of bits called blocks. Public-key algorithms are designed so that the key used for encryption is different from the key used for decryption. Usually the encryption is carried out by the public key and the decryption by the private key. DES, IDEA and SAFER are examples of symmetric algorithm whereas RSA is an example of public-key algorithm.

Most of the cryptographic algorithms have a common structure--an initial permutation followed by a main body consisting of modular multiplications and exponentiations, additions XOR's etc., followed by a final permutation. The differences lie in the number of bits, generation of the key, sequence of operations in each round, number of rounds and input-output permutation. A comparison of a few important cryptographic algorithms have been included in Table 1.

Algorithm	Plaintext	# of rounds	Key Length	I/O Transfers
DES	64 bits	16	56 bits	YES
IDEA	64 bits	8	128 bits	YES
Blowfish	64 bits	16	32 to 448 bits	NO
SAFER K-128	64 bits	varying	128 bits	YES
GOST	64 bits	32	256 bits	NO
RC5	varying	varying	varying	NO
FEAL	64 bits	8	64 bits	YES
LOKI	64 bits	16	64 bits	NO

Table 1 . Comparison of famous cryptographic algorithms

While all the cryptographic algorithms have been implemented in software, some of them have been implemented in hardware. For instance, there are several hardware implementations of DES and RSA—the two most popular cryptographic algorithms. The main drawback of the hardware implementations are that they are inflexible and cannot even support a small modification of the algorithm. This is quite restrictive, especially since cryptographic algorithms are evolving at a fast rate to combat cryptanalysis (the art of breaking the ciphertext).

In this paper, a high speed programmable architecture that can handle a large number of cryptographic algorithms has been designed. We chose a hardware implementation over software, for several reasons. First, hardware implementations are faster. Second, hardware implementations are safer. Hardware encryption devices can be securely encapsulated, and hence there is a physical protection. We chose a programmable architecture over an application-specific architecture because of the increased flexibility of a programmable architecture. The proposed architecture has a data-path that consists of permutation units, modular multipliers, XOR arrays, adder/subtractors to compute the main mathematical functions (modular multiplication, exponentiation etc.) and a control unit that handles the differences in the bit lengths and in the sequence of operations. Since modular multiplication is the most computationally intensive operation, loop unrolling is used to speed up the algorithm. As a result, our

architecture achieves significantly higher data rates compared to existing architectures.

2. PROGRAMMABLE ARCHITECTURE

In this section, we describe the details of the proposed programmable architecture. Since a large class of cryptographic algorithms require addition, subtraction, modular multiplication, exponentiation, XOR, substitution and permutation, we have designed an architecture that supports all of these operations. Figure 1 gives the block diagram of this architecture. While the adder/subtractor, multiplier, XOR array etc. are hardwired, substitution and permutation are implemented by an EPROM. The programmability of this architecture is implemented by the control unit. The control unit is also hardwired and stores the control signals required at each time step during the operation of the different algorithms. Finally, the data path width is chosen to be 32 bits. Operations on bit widths larger than 32 can easily be incorporated into our architecture.

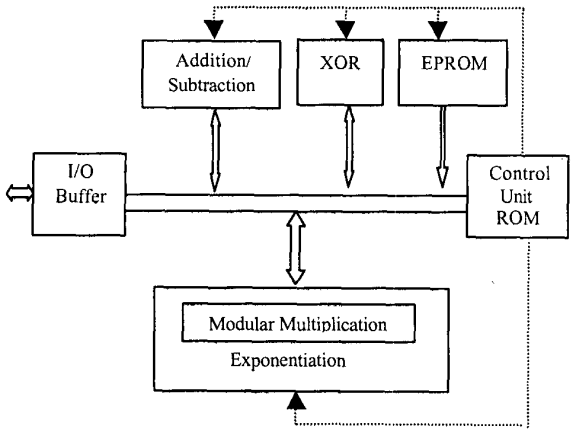

Figure 1 . High level Block Diagram.

2.1 Adder/Subtractor

This is implemented using Brent and Kung's tree structure [2]. We chose this adder since it helps in reducing the computation delay. A 32-bit adder is built using four 8-bit adders. Additions on words with bit widths greater than 32 can easily be handled by this structure.

2.2 Exponentiation Unit

Modular exponentiation of integers is certainly the most computationally intensive operation. The most popular way of implementing this is by the following square and multiplication algorithm [3]:

Calculation of $P = M^E \bmod N$.
$P_0 = 1, M_0 = M$
for i = 0 to n-1
$M_{i+1} = M_i^2 \pmod N$
if $e_i = 1$
$P_{i+1} = P_i M_i \pmod N$
else
$P_{i+1} = P_i$
where $E = (e_{n-1}e_{n-2}.....e_1e_0)_2$ and P_i is the partial product.

This algorithm takes *2n* operations in the worst case and *1.5n* on an average. Thus modular exponentiation is reduced to a series of modular multiplications and squarings. The most sought after algorithm for modular multiplication, referred to as the Montgomery algorithm, was proposed by P.L.Montgomery in 1985 [4]. The Montgomery algorithm for computing AB mod N in radix-2 is described below [5]:

$R_0 = 0$
for i = 0 to n-1
$q_i = R_i + a_iB \pmod 2$
$R_{i+1} = (R_i + a_iB + q_iN) / 2$

Here a_i is the i^{th} bit of A, R_i is the running total of the partial results and n is the precision. Note that while calculating R_{i+1}, the term q_iN is added only when $R_i + a_iB$ is odd. Thus, q_i is used to make R_{i+1} an integer. A precondition for the algorithm to work is that the modulus N has to be relatively prime to the radix. The algorithm above actually calculates $R_n \equiv 2^{-n}$ AB (mod N). To get the correct result, we need an extra Montgomery modular multiplication by 2^{2n} mod N. Thus it is helpful to pre-multiply all the inputs by a factor of 2^{2n} mod N. Hence every intermediate result carries a factor of 2^n. A final Montgomery multiplication of the result by 1 eliminates this factor.

Several of the architectures for implementing Montgomery multiplication is based on systolic arrays [5]. While some use a 2-D array with each processing unit computing a single bit, others process *u* bits at a time. Thus these architectures differ in the area – time requirements. There are still others that do all the computations using the redundant number system and convert the results into a binary system only at the very end. The architecture proposed in this paper is a 1-D systolic array implementation of the modified Montgomery algorithm. The modification is due to application of loop unrolling on the original Montgomery algorithm. As a result of this modification, the proposed architecture achieves higher speed up compared to the existing implementations. The derivation of the modified algorithm is given below.

$2R_{i+1} = R_i + a_iB + q_iN$
$2R_{i+2} = R_{i+1} + a_{i+1}B + q_{i+1}N$

The above two equations can be combined as follows:

$4R_{i+2} = R_i + (2a_{i+1} + a_i)B + (2q_{i+1} + q_i)N$

The modified Montgomery algorithm is described below:

$R_{-1} = R_0 = 0, a_{-1} = 0, q_{-1} = 0$
for i = -1 to n-2 in steps of 2
$q_{i+1} = R_{i+1} + a_{i+1}B \pmod 2$
$R_{i+2} = [R_i + (2a_{i+1} + a_i)B + (2q_{i+1} + q_i)N] / 4$

Note that only half of the original R values are now calculated. This results in significant reduction in the computation time. All the q values have to be calculated however. We propose a one-dimensional systolic array to implement the modified Montgomery algorithm. Each processing unit computes 8 bits of modular multiplication. For a 32-bit modular multiplication, we need five such processing units. The computation takes place in a pipelined

fashion. For instance, if in clock cycle i unit 0 computes 0 to 7 bits of R_i, then in cycle $i+1$, unit 1 computes bits 8 to 15 of R_i, in cycle $i+2$, unit 2 computes bits 16 to 23 of R_i, in cycle $i+3$, unit 3 computes bits 24 to 31 of R_i. Unit 4 is added to take care of the carry bits. Thus, every processing unit is active only 50% of the time. This can be avoided by using a different set of inputs in alternate cycles.

If the number of bits is greater than 32, then the number is divided into groups of 32 bits, and each 32 bit group is operated upon individually. The operation starts with the lower 32 bits, and the carry from this is circulated to the next round of multiplication involving the next 32 bits.

Figure 2 describes one such processing unit. Recall that each processing unit computes 8 bits of R_{i+2} where $R_{i+2} = [R_i + (2a_{i+1} + a_i)B + (2q_{i+1} + q_i)N] / 4$. The term $(2a_{i+1}+a_i)B$ is calculated in the following way. Since, a_{i+1}, $a_i \in \{0,1\}$, $2a_{i+1}+a_i \in \{0,1,2,3\}$. Thus $(2a_{i+1}+a_i)B$ is either 0, B, 2B or 3B. Instead of calculating $(2a_{i+1}+a_i)B$ during processing, the values of 0, B, 2B, 3B are precalculated and stored in registers. A multiplexer MUX B, is then used to select one of these values (0, B, 2B, 3B) based on a_{i+1}, a_i. A similar procedure is used to calculate $(2q_{i+1}+q_i)N$. The 8-bit outputs of these two multiplexers are then added by ADDER 1and the result is stored in REG 1. ADDER 2 adds the two bits of CARRY_IN generated by the right neighbor with the partial product of the previous iteration and stores the result in REG2. Then, ADDER 3 adds the contents of REG 1 and REG 2.

Since at the end of the computation each processing unit stores only 8 bits, some additional manipulations have to be done. The 2 most significant bits, bit 9 and bit 8, of the 10-bit output form CARRY_OUT which in turn becomes the CARRY_IN of the left neighbor. Bits 0 through 7 are stored as the partial product at the end of each round in RESULT_j. Of these bits, bit 1 and bit 0 form LSB_OUT, which is the LSB_IN of the right neighbor. These 2-bits help implementing the division by 4 in the following way. The result obtained at the end of the i^{th} iteration in RESULT_j is $4R_i$. In order to implement the right shift by 2, the LSB_IN bits (of the left neighbor) are concatenated with the six higher order bits of RESULT_j to form R_i. After $[n/2]$ rounds, the final result is stored in RESULT_j.

As stated earlier, two different streams of numbers are multiplied in alternate cycles in order to fully utilize the system. RES_REG 1 contains the partial product of the first set of numbers while RES_REG 2 contains that of the second set. The multiplexer RES_MUX selects between RES_REG1 and RES_REG2 depending on which cycle it is.

By loop unrolling the Montgomery algorithm, the number of iterations to calculate R has been reduced to half ($n/2$ instead of n). But, all n q values have to be generated. To facilitate this, a separate q-generation unit has been designed as shown in Figure 3. We denote the two least significant bits of B as B[1]B[0], of N as N[1]N[0] and of R as R[1]R[0]. Since $a_i \in \{0,1\}$, the two least significant bits of a_iB is either B[1]B[0] or 00. Similarly, the two least significant bits of q_iN is either N[1]N[0] or 00. The two least significant bits of a_iB and R_i are combined to form q_i, as given by the equation in the Montgomery algorithm. Then, the LSB 2-bits of q_iN, a_iB and R_i are combined to calculate the least significant bit of R_{i+1}. Then, the least significant bits of R_{i+1} and $a_{i+1}B$ are added to form q_{i+1}. These values are fed to unit 0 of the systolic array multiplier.

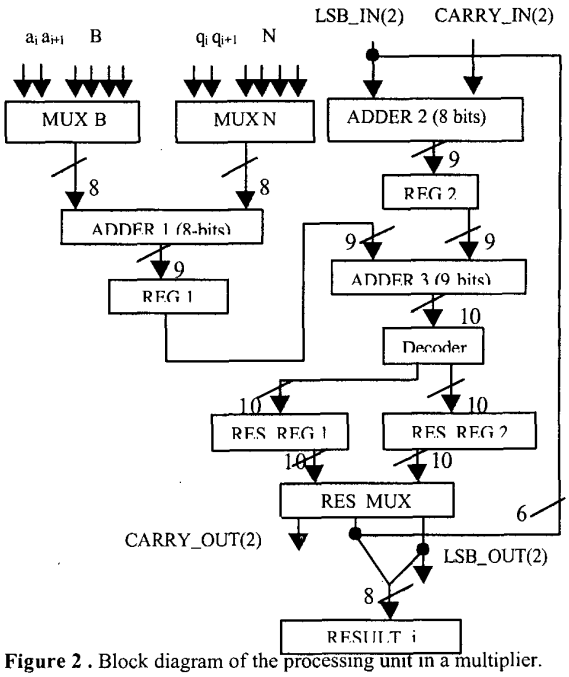

Figure 2 . Block diagram of the processing unit in a multiplier.

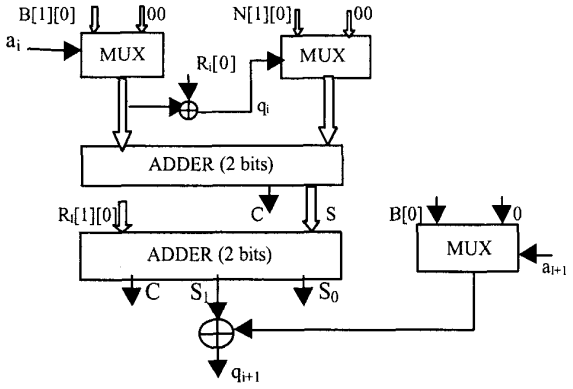

Figure 3 . Q-generation Unit

The modular multiplier unit described is used to carry out exponentiation. $P = M^R \bmod N$ is carried out using the square and multiply algorithm as shown in Figure 4. Both the inputs are first pre-multiplied by 2^{2n}. The value of M is stored in the B-REG, which acts as the input B for the modular multiplication unit. The other input a_i and a_{i+1} alternates between M and P. If the i^{th} bit of the exponent $e_i = 1$, then the modular multiplication $P_{i+1} = P_iM_i \bmod N$ is carried out. If e_i is not 1, then no multiplication is done and $P_{i+1} = P_i$. In every alternate iteration, $M_{i+1} = M_i^2 \bmod N$ is calculated. At the end of n iterations, the result is Montgomery multiplied by 1 to get the final result.

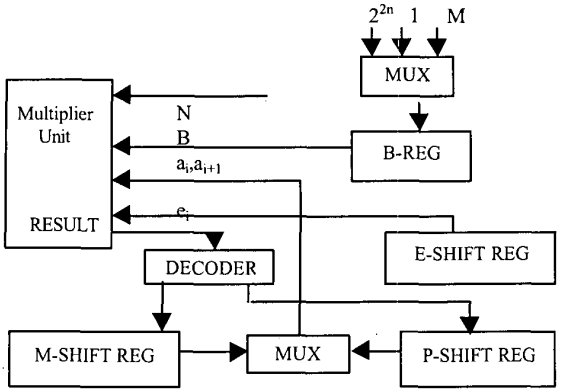

Figure 4. Exponentiation Unit

2.3 Substitution and permutation Units

Substitution and permutation are used to create confusion and diffusion during encryption. Substitution is a mapping of m input bits to n output bits and takes place in a separate unit called the S-Box. The larger the S-Box, the more difficult it is to break the algorithm. Permutation can be of several types--direct, expansion where the number of bits are increased, and compression where the number of bits are reduced. Since the number of input bits, output bits and the overall size of these boxes vary from algorithm to algorithm, it is not possible to have a hardwired unit that supports all algorithms. In our architecture, an EPROM is used to implement the S-Box and P-Box for the different algorithms.

2.4 Control Unit

A master processor sends the instructions, the bit length and the number of rounds as the input to this control unit. After receiving the required signals from the master, the control unit takes charge. ROM stores the number of cycles required to process a particular instruction with a particular bit length. The control unit interacts between the different arithmetic and logic units and gives out the valid output data after the required number of rounds are completed.

3. RESULTS

The circuit designed in this paper was synthesized using AutoLogic II, a software application from Mentor Graphics. The entire circuit runs at a frequency of 77 MHz and has an area of 23,000 "Optimization COST" units as shown in Table 2.

4. CONCLUSIONS

We conclude this paper by comparing the performance of our architecture with that of other DES architectures in Table 3 and other RSA architectures in Table 4 [6]. Note that the data rate of our architecture is significantly higher than other DES and RSA architectures even though the technology used was 1µ. If our architecture was synthesized using submicron technology, the data rate would have been even higher. The high data rate is primarily due to the applications of loop unrolling on the Montgomery algorithm. Finally, the architecture proposed here is that of a programmable processor and can be used for a large number of cryptographic algorithms.

Unit	Area (Optimization COST)	Latency (ns)	Max. Prop. Delay (ns)
XOR	676.9	8.4	2.7
8-BIT ADDER	1645.8	9.2	3.4
8-BIT MULTIPLIER	1787.2	12.8	4.3
32-BIT ADDER	6645.7	9.5	3.4
32-BIT MULTIPLIER	9483.6	51.6	6.5
Q-GENERATION	85.7	21.6	0.7
RESET COUNTER	327.5	8.1	2.0
INSTRUCTION COUNTER	99.6	7.4	1.1
READ COUNTER	280.2	7.8	2.0
REGISTER BANK	2077.9	21.0	3.1
CONTROL	3003.0	8.9	3.1

Table 2. Synthesis results of the proposed architecture.

ITEM	YEAR	CLOCK (MHz)	DATA RATE (Mbyte/s)
Broscius [7]	1991	N/A	11.6
VM009 [1]	1993	33	14.0
Supercrypt CE99C003A [1]	1994	30	20.0
Our design	1999	77	44.0

Table 3. Comparison with some DES architectures.

Item	Year	# of bits	Technology (µm)	Clock (MHz)	Date rate (Kbit/s)
Victor	1994	512	1	25	100
NTT	1994	1024	0.5 gate array	40	20
Chen	1995	512	0.8	50	24.3
Yang	1996	512	0.6	125	118
Guo	1998	512	0.6	143	278
Our design	1999	Variable	2	77	300

Table 4. Comparison with some RSA architectures.

5. REFERENCES

1. Bruce Schneier, 'Applied Cryptography,' second edition, John Wiley & Sons, Inc.
2. R. Brent and H. Kung, 'A regular layout for parallel adders,' *IEEE transactions on Computers,* March 1982, volume C-31, no. 3, pp. 260-264.
3. P. Wang, 'New VLSI architectures of RSA public key cryptosystems,' *Proceedings of 1997 IEEE International Symposium on Circuits and Systems,* vloume 3, 1997, pp. 2040-2043.
4. P. Montgomery, 'Modular multiplication without trial division,' *Mathematics of Computation,* 44(170), April 1985.
5. C. Walter, 'Systolic Modular Multiplication,' *IEEE Transactions on Computers,* 42(3), March 1993, pp. 376-378.
6. J. Guo, C. Wang and H. Hu, 'Design and implementation of an RSA public-key cryptosystem,' *Proceedings of 1999 IEEE International Symposium on Circuits and Systems.*
7. A.G. Broscius and J.M. Smith, 'Exploiting Parallelism in Hardware Implementation of the DES,' *Advances in Cryptology: CRYPTO-91 Proceedings,* Springer-Verlag, 1992, pp.367-376.