CRYPTOPROCESSOR PLD001

Master Thesis

by

Jüri Põldre

Department of Computer science
Tallinn Technical University

June 1998

1 Acknowledgments

Author is grateful to prof. Raimund Ubar for the advises and patience, European Union TEMPUS office and EUROPRACTICE organization for giving us the opportunity and dr. Ülo Jaaksoo and Institute of Cybernetics for financial support.

Jüri Põldre,

Spring 1998

2 Annotatsioon

Käesolev töö kirjeldab autori poolt projekteeritud krüpteerimisprotsessori PLD001 arhitektuuri ning temas kasutatavaid RSA (avaliku võtme) ning IDEA (salajase võtme) algoritme. Valitud algoritme analüüsitakse lähtudes realiseerimise riistvaralisest keerukusest ning töökiirusest. Töö lisasse kuulub PLD001 tehnospetsifikaat ning katsepartii testimise aruanne eesti keeles.

Töö toimus Tallinna Tehnikaülikoolis aastatel 1994-1997. Protsessori prototüüp toodeti Euroopas 1997 a. veebruaris Küberneetika instituudi finantseerimisel. Prototüüp läbis katsetused 25 MHz töösagedusel vastavalt simuleerimistulemustele.

3 Summary

The thesis paper describes cryptoprocessor PLD001 developed and designed by author. The RSA (key exchange) and IDEA (secret key) algorithms are analysed based on hardware implementation resources and encryption speed. The Technical Specifications and Prototype Testing Report (both in Estonian) are a part of this work.

Work was carried out at Tallinn Technical University during 1994-1997. The prototype was produced in Europe in February 1997 with financing from Institute of Cybernetics. The circuit passed tests at 25 MHz as expected from simulation results.

Contents

4 Introduction

Data encryption based on asymmetric key exchange algorithms and symmetric block encryption has been used in data communications for many years. These systems usually consist of general-purpose processor and some additional logic to speed up modular calculations. As the key generation for RSA [APPC] is complicated and time-consuming most hardware implementations use externally generated keys. This presents serious security problem connected with the possibility to access sensitive data. If the key generation, inversion and the key exchange field handling can be done in a single tamper-proof device, there is no need to enable user access to these procedures. As these procedures can be realised using a reasonable amount of memory, it becomes feasible to realise such a tamper-proof device in a single integrated circuit. Such a realisation would not only improve security but is also cost effective.

The anticipated explosion in Internet commerce will first require proven cryptographic solutions that can move, manage, and store information securely. Neither individuals, businesses, nor governments will entrust their data to an open network unless they are assured their information is secured with cryptographic implementations strong enough for commercial Internet requirements.

While many claims have been asserted about the ability of software-based cryptographic products to assure transactions and other vital information on the Internet, this brief will address the limitations of these kinds of products, and the strength of hardware-based cryptographic implementations.

Information in a network has varying economic value, which may change over time. Some information needs to be protected for a short period of time while other information may require protection for years or even decades. Some examples and their protection requirements are:

An electronic funds transfer of millions of dollars may need to be protected for the duration of the transaction. This could be a short term of seconds, minutes, or hours. A Company’s strategic business plan must remain confidential for years. A government’s nuclear research data must remain secret for years or decades. A credit card payment transaction over the Internet. The individual’s credit card number must remain confidential on the Internet. Specifically, it must not be revealed to the merchant. Cryptography is the enabling technology for securing all of this data. In planning a security strategy, companies have to take into consideration how long the cryptographic keys must be protected.

Cryptography allows two participants to exchange information while preventing others from reading it. This is accomplished by scrambling data using mathematical processes that make it very difficult and time consuming for anyone other than the authorised recipients to recover the original data.

Either computer software or hardware can perform cryptography. Cryptographic algorithms may be placed in a ROM or PROM for execution by a microprocessor or stored on a disk and executed by a computer-processing unit in a host system. This approach is generally referred to as a software implementation of encryption. An alternative cryptographic implementation is to isolate storage and execution in a hardware peripheral device such as a smart card, PCMCIA card, or a special tamper resistant "black box." These implementations are hardware solutions. In particular, the tamper resistant "black box" is called a hardware cryptographic module (HCM).

Companies must keep in mind that there is often more than one way for an opponent to attack a cryptographic system. If the size of the key space becomes too large then the opponent will look elsewhere for weaknesses. For instance, if the cryptographic system is viewed as a combination lock, then the attacker will get a bolt cutter if the number of digits becomes too large. The combination lock will simply be cut off the door and the adversary will never go through the exhaustive process of trying a large number of combinations. This can be prevented if the lock materials are stronger than the bolt cutter.

In properly implemented transaction security architecture, three things must be accomplished.

First, the only element of the system that must be kept secret is the cryptographic keying materials. With a good cryptographic scheme it is perfectly acceptable to have everyone„including the adversaries„know the algorithm because knowledge of the algorithm without the key does not help untangle the scrambled information. Second, the integrity of the algorithms must be maintained. This means that an adversary must not be able to modify the algorithm and cause it to perform in a way that would allow encrypted information to be easily read. Third, the cryptographic processing performance must be consistent with transaction processing requirements.

Beyond these three conditions, the security architect is free to make public which algorithms; key sizes, protocols and data or message formats are being used without compromising security.

Absolute security is assumed to be impossible to obtain. With enough money, effort, and skill virtually any security scheme can be defeated. Furthermore, as technology continues to evolve, new techniques may be developed to attack a security scheme previously impervious to attack. Ultimately, we must be cognisant of the cost to the adversary of mounting a successful attack and the financial benefits that the adversary could realise from such an attack.

Secure key exchange with RSA in a reasonable time requires a lot of hardware resources. Most of it is used for modular arithmetic. Also this device should include hardware for a secure block cipher. The calculations used by most block ciphers are bit operations and table lookups, which are hard to share with integer arithmetic and have to be realised by separate hardware. One solution would be to use IDEA cipher for block encryption. IDEA has 128-bit key length. The encryption process consists of 8 rounds. The operations in one round contain 16-bit modular additions and multiplication, which can be easily shared with integer calculations used in RSA. Also the key inversion algorithms for both ciphers are similar. When using the same ALU for both asymmetric key exchange algorithms and block encryption, it is possible to save silicon area.

The combination of RSA and IDEA has also been used with success in freeware e-mail encryption system PGP.

In the following pages we give an overview of RSA and IDEA and present a VLSI implementation of the device discussed above consisting of

- data path description,
- control part,
- self-test
- future directions

5 The RSA Algorithm

RSA is a public-key cryptosystem for both encryption and authentication; it was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman [RSA78]. First two large primes, p and q are found and then multiplied to find their product $n = pq;$ n is called the modulus. Then we choose a number, e, less than n and relatively prime to (p-1)(q-1), which means that e and
$(p\text{-}1)(q\text{-}1)$ have no common factors except 1. Last we find another number d such that (ed - 1) is divisible by (p-1)(q-1). The value e and d are called the public and private exponents, respectively. The public key is the pair (n, e); the private key is (n, d). The factor p and q maybe kept with the private key, or destroyed.
It is generally thought to be NP difficult to obtain the private key d from the public key (n, ^{e}). If one could factor n into p and q, however, then one could obtain the private key d. Thus the security of RSA is related to the assumption that factoring is difficult. An easy factoring method or some other feasible attack would "break" RSA.
The RSA algorithm can be used for privacy and authentication in the following way:

5.1 RSA privacy

RSA privacy (encryption): Suppose Alice wants to send a message m to Bob. Alice creates the ciphertext c by exponentiation: $c = m^e$ mod n, where e and n are Bob's public key. She sends c to Bob. To decrypt, Bob also caclulates: $m = \mathrm{c}^d$ mod n; the relationship between e and d ensures that Bob correctly recovers m. Since only Bob knows d, only Bob can decrypt.

5.2 RSA authentication

RSA authentication: Suppose Alice wants to send a message m to Bob in such a way that Bob is assured that the message is authentic and is from Alice. Alice creates a digital signature s by exponentiation: $s = m^d$ mod n, where d and n are Alice's private key. She sends m and s to Bob. To verify the signature, Bob caclulates and checks that the message m is recovered: $m = s^e$ mod n, where e and n are Alice's public key.

Thus encryption and authentication take place without any sharing of private keys: each person uses only other people's public keys and his or her own private key. Anyone can send an encrypted message or verify a signed message, using only public keys, but only someone in possession of the correct private key can decrypt or sign a message.

5.3 RSA operation

An "RSA operation," whether for encrypting or decrypting, signing or verifying, is essentially a modular exponentiation, which can be performed by a series of modular multiplication.

In practical applications, it is common to choose a small public exponent for the public key; in fact, entire groups of users can use the same public exponent, each with a different modulus. (There are some restrictions on the prime factors of the modulus when the public exponent is fixed.) This makes encryption faster than decryption and verification faster than signing. With typical modular exponentiation algorithms, public-key operations take $O(k2)$ steps, private-key operations take $O(k^3)$ steps, and key generation takes $O(k^4)$ steps, where k is the number of bits in the modulus [EXPALG]. "Fast multiplication" techniques, such as FFT-based methods, require asymptotically fewer steps, though in practice they are not as common due to their great software complexity and the fact that they may actually be slower for typical key sizes.

There are many commercially available software and hardware implementations of RSA, and there are frequent announcements of newer and faster chips. On a 90 MHz Pentium, RSA Data Security's cryptographic toolkit BSAFE 3.0. has a throughput for private-key operations of 21.6 Kbits per second with a 512-bit modulus and 7.4 Kbits per second with a 1024-bit modulus. The fastest RSA hardware [VICHILD] has a throughput greater than 300 Kbits per second with a 512-bit modulus, implying that it performs over 500 RSA private-key operations per second. (There is room in that hardware to execute two RSA 512-bit RSA operations in parallel, hence the 600 Kbits/s speed reported in. For 970-bit keys, the throughput is 185 Kbits/s.) It is expected that RSA speeds will reach 1 Mbits/second within a year or so. For more RSA realisations see [RSAGER], [PAM93], [VICTOR].

By comparison, DES is much faster than RSA. In software, DES is generally at least 100 times as fast as RSA. In hardware, DES is between 1,000 and 10,000 times as fast, depending on the implementation. Implementations of RSA will probably narrow the gap a bit in coming years, as there are growing commercial markets, but DES will get faster as well.

5.4 Modular exponent operation

The modular exponentiation operation is simply an exponentiation operation where multiplication and squaring operations are modular operations. The exponentiation heuristics developed for computing M^e are applicable for computing $M^e (mod\ n)$. The binary method for computing $M^e\ (mod\ n)$ given the integers M, e, and n has two variations depending on the direction by which the bits of e are scanned: Left-to-Right (LR) and Right-to-Left (RL). The LR binary method is more widely known:

```
LR Binary Method
Input: M; e; n
Output: C := $M^e$ mod n
1. if $e_{h-1}$ = 1 then C := M else C := 1
2. for i = h - 2 downto 0
2a.     C := C •C (mod n)
2b.     if ei = 1 then C := C • M (mod n)
3. return C
```

The bits of e are scanned from the most significant to the least significant, and a modular squaring is performed for each bit. A modular multiplication operation is performed only if the bit is 1. An example of LR binary method is illustrated below for $h = 6$ and $e = 55 = (110111)$. Since $e5 = 1$, the LR algorithm starts with $C := M$, and proceeds as

i	ei	Step 2a (C)	Step 2b (C)
4	1	$(M)^2 = M^2$	$M^2 \bullet M = M^3$
3	0	$(M^3)^2 = M^6$	M^6
2	1	$(M^6)^2 = M^{12}$	$M^{12} \bullet M = M^{13}$
1	1	$(M^{13})^2 = M^{26}$	$M^{26} \bullet M = M^{27}$
0	1	$(M^{27})^2 = M^{54}$	$M^{54} \bullet M = M^{55}$

The RL binary algorithm, on the other hand, scans the bits of e from the least significant to the most significant, and uses an auxiliary variable P to keep the powers M.

RL Binary Method

```
Input: M; e; n
Output: C := $M^e$ mod n
1. C := 1 ; P := M
2. for i = 0 to h - 2
    2a. if $e_i$ = 1 then C := C • P (mod n)
    2b. P := P • P (mod n)
3. if $e_{h-1}$ = 1 then C := C • P (mod n)
4. return C
```

The RL algorithm starts with *C := 1* and *P := M*, proceeds to compute M^{55} as follows:

i	e_i	Step 2a (C)	Step 2b (P)
0	1	$1 \bullet M = M$	$(M)^2 = M^2$
1	1	$M \bullet M^2 = M^3$	$(M^2)^2 = M^4$
2	1	$M^3 \bullet M^4 = M^7$	$(M^4)^2 = M^8$
3	0	M^7	$(M^8)^2 = M^{16}$
4	1	$M^7 \bullet M^{16} = M^{23}$	$(M^{16})^2 = M^{32}$

Step 3: $e_5 = 1$, thus $C := M^{23} \bullet M^{32} = M^{55}$

We compare the LR and RL algorithm in terms of time and space requirements below:

- Both methods require *h - 1* squaring and an average of ½• (h -1) multiplication's.
- The LR binary method requires two registers: M and C.
- The RL binary method requires three registers: M, C, and P. However, we note that P can be used in place of M, if the value of M is not needed thereafter.
- The multiplication (Step 2a) and squaring (Step 2b) operations in the RL binary method are independent of one another, and thus these steps can be paralleled. Provided that we have two multipliers (one multiplier and one squarer) available, the running time of the RL binary method is bounded by the total time required for computing *h - 1* squaring operations on k-bit integers.

The advanced exponentiation algorithms are often based on word-level scanning of the digits of the exponent e. Technical report [EXPALG] contains several advanced algorithms for computing the modular exponentiation, which are slightly faster than the binary method. The word-level algorithms, i.e., the m-ary methods, require some space to keep precomputed powers of M in order to reduce the running time. These algorithms may not be very suitable for hardware implementation since the space on-chip is already limited due to the large size of operands involved (e.g., 1024 bits). Thus, we will not study these techniques here

The remainder of this report reviews algorithms for computing the basic modular arithmetic operations, namely, the addition, subtraction, and multiplication. We will assume that the underlying exponentiation heuristic is either the binary method, or any of the advanced m-ary algorithm with the necessary register space already made available. This assumption allows us to concentrate on developing time and area efficient algorithms for the basic modular arithmetic operations, which is the current challenge because of the operand size.

The literature is replete with residue arithmetic techniques applied to signal processing. However, in such applications, the size of operands is very small, usually around 5-10 bits, allowing table lookup approaches. Besides the modulo are fixed and known in advance, which is definitely not the case for our application. Thus, entirely new set of approaches are needed to design time and area efficient hardware structures for performing modular arithmetic operations to be used in cryptographic applications.

5.5 Modular multiplication operation

The modular multiplication problem is defined as the computation of $P = AB \ (mod\ n)$ given the integers A, B, and n. It is usually assumed that A and B are positive integers with $0 \leq A;\ B < n$, i.e., they are the least positive residues. There are basically four approaches for computing the product P:

- Multiply and then divide.
- The steps of the multiplication and reduction are interleaved.
- Brickell’s method.
- Montgomery’s method.

5.5.1 Multiply and divide method

The multiply-and-divide method first multiplies A and B to obtain the 2k-bit number

$$P' := AB .$$

Then, the result P’ is divided (reduced) by n to obtain the k-bit number

$$P := P' \% n.$$

We will not study the multiply-and-divide method in detail since the interleaving method is more suitable and also more efficient for our problem. The multiply-and-divide method is useful only when one needs the product P’.

5.5.2 Interleaving Multiplication and Reduction

The interleaving algorithm has been known. Let A_i and B_i be the bits of the k-bit positive integers A and B, respectively. The product P’ can be written as:

$$P' = A \bullet B = A \bullet \Sigma_i B_i \bullet 2^i = 2(\ldots 2(2(0 + A \bullet B_{k-1}) + A \bullet B_{k-2}) + \ldots) + A \bullet B0$$

This formulation yields the shift-add multiplication algorithm. We also reduce the partial product modulo n at each step:

```
1. P := 0
2. for i = 0 to k - 1
2a.     $P := 2P + A \bullet B_{k-1-i}$
2b.     $P := P \bmod n$
3. return P
```

Assuming that A; B; $P < n$, we have

$$P := 2P + A \bullet B_j \leq 2(n - 1) + (n - 1) = 3n - 3 .$$

Thus, the new P will be in the range $0 \leq P \leq 3n - 3$ and at most 2 subtractions are needed to reduce P to the range $0 \leq P < n$. We can use the following algorithm to bring P back to this range:

```
P’ := P - n ; If P ≥ 0 then P = P’
P’ := P - n ; If P ≥ 0 then P = P’
```

The computation of P requires k steps, at each step we perform the following operations:

- A left shift: 2P
- A partial product generation: $A \bullet B_j$
- An addition: P := 2P + A _ Bj
- At most 2 subtractions:
  P' := P - n ; If $P \geq 0$ then P = P'
  P' := P - n ; If $P \geq 0$ then P = P'

Wiring easily performs the left shift operation. The partial products, on the other hand, are generated using an array of AND gates. The most crucial operations are the addition and subtraction operations: they need to be performed fast. We have the following solutions to explore:

- We can use the carry propagate adder, introducing O(k) delay per step. However, Omura's method can be used to avoid unnecessary subtractions:
  1. P := 2P
  2. If carry-out then P := P + m
  3. P := P + A • Bj
  4. If carry-out then P := P + m

- We can use the carry save adder, introducing only O(1) delay per step. However, recall that the sign information is not immediately available in the CSA. We need to perform fast sign detection in order to determine whether the partial product needs to be reduced modulo n.

5.5.3 Brickell's method

This method is based on the use of a carry-delayed integer. Let A is a carry-delayed integer, then, it can be written as:

$$A = \Sigma_{i=0,..k-1} (Ti + Di) \bullet 2^i$$

The product P = AB can be computed by summing the terms:

$$(T_0 \bullet B + D_0 \bullet B) \bullet 2^0 +$$

$(T_1 \bullet B + D_1 \bullet B) \bullet 2^1 +$

$(T_2 \bullet B + D_2 \bullet B) \bullet 2^2 +$

. . .

$(T_{k-1} \bullet B + D_{k-1} \bullet B) \bullet 2^{k-1}$

Since D0 = 0, we rearrange to obtain

$2^0 \bullet T_0 \bullet B \quad + \quad 2^1 \bullet D_1 \bullet B +$

$2^1 \bullet T_1 \bullet B \quad + \quad 2^2 \bullet D_2 \bullet B +$

$2^2 \bullet T_2 \bullet B \quad + \quad 2^3 \bullet D_3 \bullet B +$

. . .

$2^{k-2} \bullet T_{k-2} \bullet B \qquad +$

$2^{k-1} \bullet D_{k-1} \bullet B \qquad +$

$2^{k-1} \bullet T_{k-1} \bullet B$

Also recall that either T_i or D_{i+1} is zero due to the property of the carry delayed adder. Thus, each step requires a shift of B and addition of at most 2 carry delayed integers:

- Either: $(P_d; P_t) := (P_d; P_t) + 2^I \bullet T_i \bullet B$
- Or: $(P_d; P_t) := (P_d; P_t) + 2_{i+1} \bullet D_{i+1} \bullet B$

After k steps P = (Pd; Pt) is obtained. In order to compute P (mod n), we perform reduction:

If $P \geq 2^{k-1} \bullet n$ then $P := P - 2^{k-1} \bullet n$

If $P \geq 2^{k-2} \bullet n$ then $P := P - 2^{k-2} \bullet n$

If $P \geq 2^{k-3} \bullet n$ then $P := P - 2^{k-3} \bullet n$

. . .

If $P \geq n$ then $P := P - n$

We can also reverse these steps to obtain:

$P := T_{k-1} \bullet B \bullet 2^{k-1}$

$P := P + T_{k-2} \bullet B \bullet 2^{k-2} + D_{k-1} \bullet B \bullet 2^{k-1}$

$P := P + T_{k-3} \bullet B \bullet 2^{k-3} + D_{k-2} \bullet B \bullet 2^{k-2}$

. . .

$P := P + T_1 \bullet B \bullet 2^1 + D_2 \bullet B \bullet 2^2$

$P := P + T_0 \bullet B \bullet 2^0 + D_1 \bullet B \bullet 2^1$

Also, the multiplication steps can be interleaved with reduction steps. To perform the reduction, the sign of $P - 2^I \bullet n$ needs to be determined (estimated). Brickell's solution is essentially a combination of the sign estimation technique and Omura's method of correction.

We allow enough bits for P, and whenever P exceeds 2^k add $m = 2^k - n$ to correct the result. 11 steps after the multiplication procedure started, the algorithm starts subtracting multiples of n. In the following, P is a carry delayed integer of $k + 11$ bits, m is a binary integer of k bits, and t_1 and t_2 control bits, whose initial values are $t_1 = t_2 = 0$.

1. Add the most significant 4 bits of P and $m \bullet 2^{11}$.
2. If overflow is detected, then $t_2 = 1$ else $t_2 = 0$.
3. Add the most significant 4 bits of P and the most significant 3 bits of $m \bullet 2^{10}$.
4. If overflow is detected and $t_2 = 0$, then $t_1 = 1$ else $t_1 = 0$.

The multiplication and reduction steps of Brickell's algorithm are as follows:

$$B' := T_i \bullet B + 2 \bullet D_{i+1} \bullet B$$

$$m' := t_2 \bullet m \bullet 2^{11} + t_1 \bullet m \bullet 2^{10}$$

$$P := 2 (P + B' + m')$$

$$A = 2A.$$

5.5.4 Montgomery's Method

The Montgomery algorithm computes

$MonPro(A; B) = A \bullet B \bullet r^{-1} \bmod n$ given A; $B < n$ and r such that $gcd(n; r) = 1$. Even though the algorithm works for any r which is relatively prime to n, it is more useful when r is taken to be a power of 2, which is an intrinsically fast operation on general-purpose computers, e.g., signal processors and microprocessors. To find out why the above computation is useful for computing the modular exponentiation, we refer the reader [HOPHD] what contains good survey of the algorithm.

In this section we introduce an efficient binary add-shift algorithm for computing $MonPro(A; B)$, and then generalise it to the m-ary method. We take $r = 2^k$, and assume that the number of bits in A or B is less than k. Let $A = (A_{k-1} A_{k-2} \dots A_0)$ be the binary representation of A. The above product can be written as:

$$2^{-k} \bullet (A_{k-1} A_{k-2} \ldots A_0) \bullet B = 2^{-k} \bullet \Sigma_{i=0..k-1} A_i \bullet 2^i \bullet B \text{ (mod n) .}$$

The product $t = (A_0 + A_1 2 + \ldots A_{k-1} \bullet 2^{k-1}) \bullet B$ can be computed by starting from the most significant bit, and then proceeding to the least significant, as follows:

```
1. t := 0
2. for i = k - 1 to 0
2a.     t := t + $A_i \bullet$ B
        2b.     t := 2 • t
```

The shift factor 2^{-k} in $2^{-k} \bullet A \bullet B$ reverses the direction of summation. Since

$$2^{-k} \bullet (A_0 + 2A_1 + \ldots + A_{k-1} \bullet 2^{k-1}) = (A_{k-1} . 2^{-1} + A_{k-2} . 2^{-2} + \ldots + A_0 . 2^{-k}),$$

we start processing the bits of A from the least significant, and obtain the following binary add-shift algorithm to compute $t = A \bullet B \bullet 2^{-k}$

```
1. t := 0
2. for i = 0 to k - 1
2a.     t := t + $A_i \bullet$ B
2b.     t := t/2
```

The above summation computes the product $t = 2^{-k} \bullet A \bullet B$, however, we are interested in computing $u = 2^{-k} \bullet A \bullet B \text{ (mod n)}$. This can be achieved by subtracting n during every add-shift step, but there is a simpler way: We add n to u if u is odd, making new u an even number since n is always odd. If u is even after the addition step, it is left untouched. Thus, u will always be even before the shift step, and we can compute

$$u := u \bullet 2^{-1} \text{ (mod n)}$$

by shifting the even number u to the right since u = 2v implies

$$u := 2v \bullet 2^{-1} = v \pmod{n} .$$

The binary add-shift algorithm computes the product $u = A \bullet B \bullet 2^{-k} \pmod{n}$ as follows:

```
1. u := 0
2. for i = 0 to k - 1
2a. u := u + $A_i \bullet B$
2b. If u is odd then u := u + n
2c. u := u/2
```

We reserve a *(k + 1)-bit* register for u because if u has k bits at beginning of an add-shift step, the addition of $A_i \bullet B$ and n (both of which are k-bit numbers) increases its length to $k + 1$ bits. The right shift operation then brings it back to k bits. After k add-shift steps, we subtract n from u if it is larger than n.

Also note that Steps 2a and 2b of the above algorithm can be combined: We can compute the least significant bit u0 of u before actually computing the sum in Step 2a. It is given as

$$u_0 := u_0 \otimes (A_i B_0) .$$

Thus, we decide whether u is odd prior to performing the full addition operation u := $u+A_iB$. This is the most important property of Montgomery’s method. In contrast, the classical modular multiplication algorithms (e.g., the interleaving method) compute the entire sum in order to decide whether a reduction needs to be performed.

5.5.5 High-Radix Interleaving Method

Since the speed for radix 2 multipliers is approaching limits, the use of higher radices is investigated. High-radix operations require fewer clock cycles, however, the cycle time and the required area increases. Let 2^b is the radix. The key operation in computing $P = AB \pmod{n}$ is the computation of an inner-product steps coupled with modular reduction, i.e., the computation of

$$P := 2^b \bullet P + A \bullet B_i \bullet Q_n,$$

where P is the partial product and B_i is the ith digit of B in radix 2^b. The value of Q determines the number of times the modulus n is subtracted from the partial product P in order to reduce it modulo n. We compute Q by dividing the current value of the partial product P by n, which is then multiplied by n and subtracted from the partial product during the next cycle. This implementation is illustrated in the following figure:

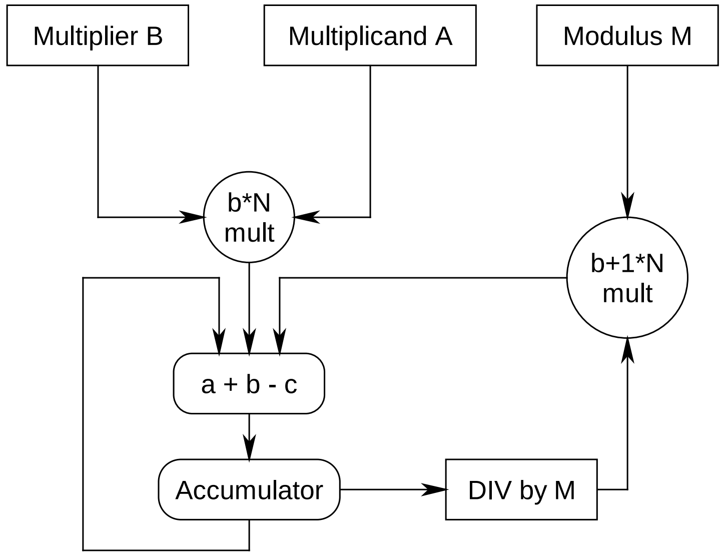


Figure 1. High-radix interleaving method

For the radix 2, the partial product generation is performed using an array of AND gates. The partial product generation is much more complex for higher radices; e.g., Wallace trees and generalised counters need to be used. However, the generation of the high-radix partial products does not greatly increase cycle time since this computation can be easily pipelined.

The most complicated step is the reduction step, which necessitates more complex routing, increasing the chip area.

5.5.6 High-Radix Montgomery’s Method

The binary add-shift algorithm is generalised to higher radix (m-ary) algorithm by proceeding word by word, where the wordsize is w bits, and k = sw. The addition step is performed by multiplying one word of A by B and the right shift is performed by shifting w bits to the right. In order to perform an exact division of u by 2, we add an integer multiple of n to u, so that the least significant word of the new u will be zero.

Thus, if $u \neq 0 \ (mod\ 2)$ we find an integer m such that $u + m \bullet n = 0 \ (mod\ 2^b)$. Let u_0 and n_0 be the least significant words of u and n, respectively. We calculate m as:

$$m = -u_0 \bullet n_0^{-1} \ (\text{mod } 2^b)$$

The word-level (m-ary) add-shift Montgomery product algorithm is thus:

```
1. u := 0
2. for i = 0 to s - 1
2a. u := u + $A_i \bullet B$
2b. m :=  $-u_0 \bullet n_0^{-1}$ ( mod $2^b$)
2c. u := u + m • n
2d. u := $u/2^b$
```

This algorithm specializes to the binary case by taking $b= 1$. In this case, when u is odd, the least significant bit u_0 is nonzero, and thus,

$$m = -u_0 \bullet n_0^{-1} \ (\text{mod } 2^b) = 1 \ (\text{mod } 2).$$

5.6 Conclusion

Advantages of RSA over other public-key cryptosystems include the fact that it can be used for both encryption and authentication, and that it has been around for many years and has successfully withstood much scrutiny. RSA has received far more attention, study, and actual use than any other public-key cryptosystem, and thus RSA has more empirical evidence of its security than more recent and less scrutinised systems. In fact, a large number of public-key cryptosystems, which at first appeared secure, were later broken.

There have been several interesting approaches to realise modular exponent calculations. Some of them are in [SYSTOL], [FASTMR],

Very good survey of approaches and realisations is in Holger Orups PhD thesis [HOPHD].

6 The block cipher IDEA

The block cipher IDEA (for International Data Encryption Algorithm) was first presented in [LAI91]; its previous version was PES (for Proposed Encryption Standard). In both ciphers, the plaintext and the ciphertext are 64 bit blocks, while the secret key is 128 bits long. Both ciphers were based on the new design concept of \mixing operations from different algebraic groups". The required “confusion" was achieved by successively using three “incompatible" group operations on pairs of 16-bit subblocks and the cipher structure was chosen to provide the necessary “diffusion". The cipher structure was further chosen to facilitate both hardware and software implementations. The IDEA cipher is an improved version of PES and was developed to increase the security against differential cryptanalysis.

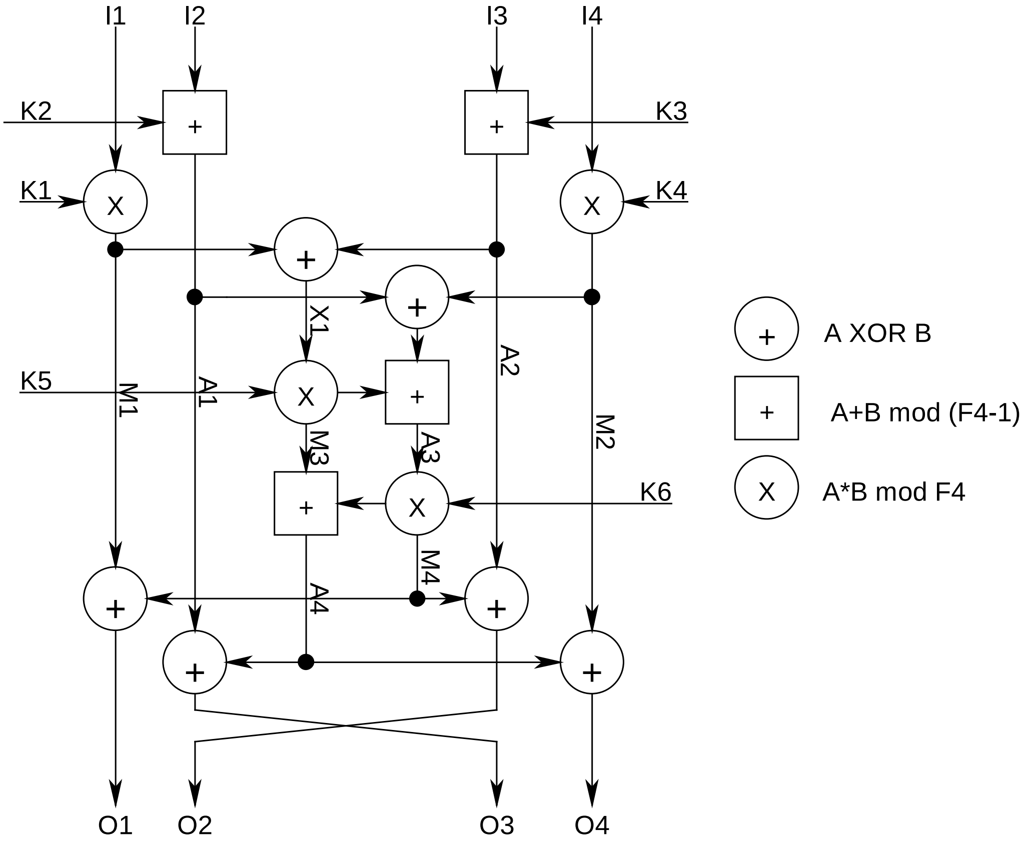
6.1 Description of IDEA

The cipher IDEA is an iterated cipher consisting of 8 rounds followed by an out put transformation. The complete first round and the output transformation are depicted in the computational graph shown in Fig. 2.

6.2 The encryption process

In the encryption process shown in Fig.3.1, three different group operations on pairs of 16-bit subblocks are used, namely,

- bit-by-bit exclusive-OR of two 16-bit subblocks, denoted as ⊗
- addition of integers modulo 2^{16} where the 16-bit subblock is treated as the usual radix-two representation of an integer; the resulting operation is denoted as +;
- multiplication of integers modulo $2^{16}+1$ (F4, Fermat’s 4th digit) where the 16-bit subblock is treated as the usual radix-two representation of an integer except that the all-zero subblock is treated as representing 2^{16}



Ii: 16-bit plaintext subblock

Oi: 16-bit ciphertext subblock

Ki: 16-bit key subblock

⊕ : Bit-by-bit exclusive-OR of 16-bit subblocks

⊗ : multiplication modulo $2^{16}+1$ of 16-bit integers with the zero subblock corresponding to 2^{16}

Figure 2. Computational graph for the encryption process of the IDEA cipher.

As an example of these group operations, note that

$$(0; :::; 0) \otimes (1; 0; :::; 0) = (1; 0; :::; 0; 1)$$

$$\text{because } 2^{16}2^{15} \bmod (2^{16}+1) = 2^{15}+1:$$

The 64-bit plaintext block X is partitioned into four 16-bit subblocks X_1; X_2; X_3; X_4; i.e., $X = (X_1; X_2; X_3; X_4)$. The four plaintext subblocks are then transformed into four 16-bit ciphertext subblocks Y_1; Y_2; Y_3; Y_4 [i.e., the ciphertext block is Y = (Y1; Y2;

Y3; Y4)] under the control of 52 key subblocks of 16 bits that are formed from the 128-bit secret key in a manner to be described below. For r = 1; 2; : : : ; 8, the six key subblocks used in the r-th round will be denoted as $Z_1(r)$. Four 16-bit key subblocks are used in the output transformation; these subblocks will be denoted as $Z_1(9)$, $Z_2(9)$, $Z_3(9)$, $Z_4(9)$,

6.3 The decryption process

The computational graph of the decryption process is essentially the same as that of the encryption process, the only change being that the decryption key subblocks $K_i(r)$ are computed from the encryption key subblocks $Z_i(r)$ i as follows:

$K_{1,4}(r)= Z_{1,4}(10-r)^{-1}$; $K_2(r)= -Z_3(10-r)$; $K_3(r)= -Z_2(10-r)$ for r = 2,3,..,8

$K_{1,4}(r)= Z_{1,4}(10-r)^{-1}$; $K_{2,3}(r)= -Z_{2,3}(10-r)$ for r = 2,3,..,8

$K_{5,6}(r) = K_{5,6}(r)$ for r = 1,2,..,8

Where Z^{-1} denotes the *multiplicative inverse (modulo* $2^{16}+1$*) of* Z, i.e., $Z \otimes Z^{-1} = 1$ and -Z denotes the *additive inverse (modulo* 2^{16}*) of* Z, i.e., $-Z + Z = 0$.

6.4 The key schedule

The 52 key subblocks of 16 bits used in the encryption process are generated from The 128-bit user-selected key as follows: The 128-bit user-selected key is partitioned Into 8 subblocks that are directly used as the first eight key subblocks, where the ordering of the key subblocks is defined as follows: $Z_{1,..,6}(1),..,Z_{1,..,6}(8)$, $Z_{1,2,3,4}(9)$ 1 6 1 2 3 4 The 128-bit user-selected key is then cyclic shifted to the left by 25 positions, after which the resulting 128-bit block is again partitioned into eight subblocks that are taken as the next eight key subblocks. The obtained 128-bit block is again cyclic shifted to the left by 25 positions to produce the next eight key subblocks, and this procedure is repeated until all 52 key subblocks have been generated.

7 PROCESSOR ARCHITECTURE

RSA was chosen for implementation for the reasons pointed out in chapter 5.6. Also RSA is behind most of commercial software encryption packages and protocols - PEM, PGP, and SSL to name a few [APPC]. From the description of RSA in chapter 5 it became clear that the operation behind RSA is A•B mod C. To complement RSA used mainly in key exchange protocols due to its slow speed we need a strong cipher for block encryption of data after key exchange. For that we had three choices: DES, RC4 and IDEA. There exists many other block encryption algorithms, but in cryptoworld you can trust only public algorithms that have withstood the scrutiny of time. The limiting factor was also key length what had to be at least 128 bits.

For IDEA the main reason for not implementing it in hardware is the process of key reversal- for decryption you must find multiplicative inverses of 16-bit integers as described in chapter 6.3. It can be done by standard processor and is fast (modified GCD algorithm, logarithmic time), but demands integer arithmetic.

DES and RC4 are free of that deficiency but they need separate block for hardware realisation. Looking at IDEA cipher (fig. 2) we can see that it uses the same modular arithmetic as RSA. So it is possible to join ALUs and thus save silicon area. As we already have integer arithmetic for RSA it also solves the key reversal problem.

What we need is two 16•16 mod 2^{16}+1 multipliers to perform operation:

$$(A \bullet B \bmod F4 + C) \bmod 2^{16}$$

The multiplication can be carried out by low-high algorithm. Finally we feed the result into CSA tree. We must have four (8•24)-bit multipliers if we want to run two 16•16-bit multiplication at the same time.

Finally by connecting four (8•24)-bit multipliers we get one 8•96-bit multiplier what can be used for RSA calculations.

What we achieve is and hybrid hardware accelerator. It is optimised for IDEA (Block cipher) while delivering reasonable RSA key exchange speed (ca. 0,5 sec.). For most applications the 10 Mbit/sec secure channel is enough and 0.5 sec to initiate is also tolerable. Because of shared ALU the chip area is low and power consumption is in tolerable region for mobile equipment.

If we add random number generator and programmable microcode we get secure “package” what can be programmed with a few lines of code to generate and exchange keys transparently to user – This feature frees the user from possibility to sell the keys to third party as (s)he does not know them and thus lets the user sleep in peace. The IDEA session key is generated new for every session.

Programmability also solves the import restriction and copyright problem- without “few lines of code” it is basically an integer calculator. Also it is possible to program several other key exchange algorithms into circuit.

Now let us look how the author’s design fulfils these requirements.

7.1 Data Path

ALU is divided into 4 different units, each 24-bit wide. For IDEA calculations these units can be configured as two 16-bit multipliers. The calculations are carried out using low-high algorithm [LAI91]. Each 24 bit block can calculate 8•16-bit multiplication. Using 2 such multipliers we can multiply 16•16 bits in one cycle.

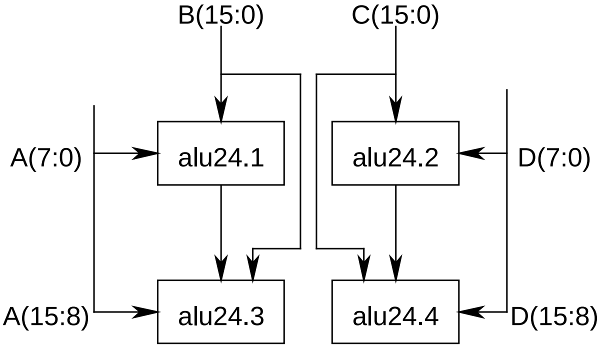


Fig. 3 ALU in IDEA multiply mode

The second cycle will be used for modular reduction step of low-high algorithm. During these steps lower parts of calculations from alu24.1 and alu24.2 are fed into CSA tree of alu blocks alu24.3 and alu24.4, avoiding carry propagation delay [HEN90]. The rest of IDEA calculations do not take much silicon area and consist of some XORs and two 16-bit adders. For more detailed description see IDEA control below.

In long modular calculations mode the ALU is configured as 8•96 bit multiplier or 96-bit adder/negator with additional carry logic.

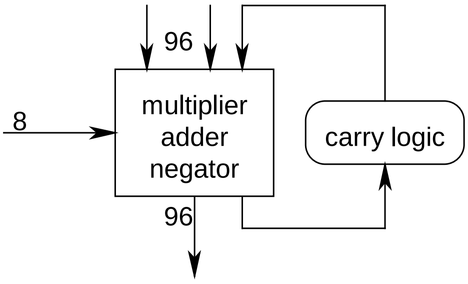


Fig. 4 ALU in modular calculation mode

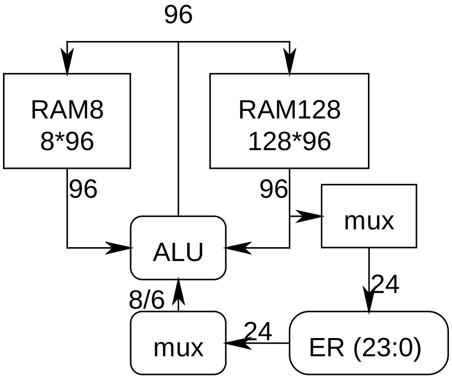


Fig. 5 The structure of ALU datapath

This mode enables us to handle long numbers 8 bits at a time. The datapath consists of ALU; two RAM modules and one 24-bit register ER (Fig.5). This small register is used for multiply and modular reduction as a source of multiplicand. RAM modules can be used as two data register groups of 1 and 16 registers. All data registers are 768 bits wide.

7.2 Alu Control Structures

ALU control structures are shown on fig. 7. The processor has two levels of code. Microcode is fixed and contains two types of commands.

Index calculating commands. For that purposes the INDEX_CALC state machine has 4 internal 8-bit registers enabling to address 6-bit entities inside the 768-bit data registers as shown on Fig. 6.

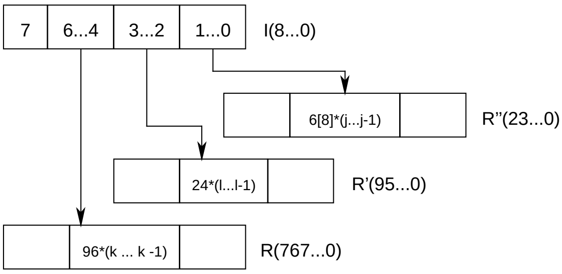


Fig. 6. Index register layout

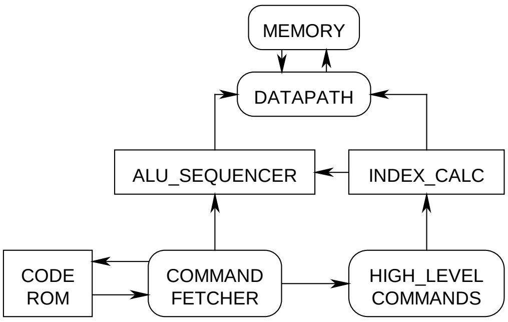


Fig. 7. ALU control structure

The index registers are used for loading parts of 768-bit registers into 24-bit register ER. Also it is possible to do logic operations between immediate values, index registers and ER.

Arithmetic commands operate on long data registers. Typical commands of that type are adding, subtracting, multiplying and transfers of data registers. These are executed in ALU_SEQUENCER in parallel with index calculations.

To ease the programming job a level of hierarchy is built upon microcode.

HIGH_LEVEL COMMANDS state machine decodes these commands and executes the necessary microcode program. The microcode program memory contains jump table at the start of code area. Each time a high level command gets executed the state machine checks the entry in this table to find the appropriate microcode.

At the highest level it is possible to choose between 32 external commands. These are selected with setting logic levels on circuits pads. The high level code has similar table as microcode to decode all external commands of the circuit.

7.3 Modular Multiply Algorithm

Let us now look more closely at the modular multiply step. Let A, B and C is data registers and S is an index register. First the operands are reduced so that for $A \bullet B \bmod C$ the first step would be

$$A := A \bmod C \quad \text{and} \quad B := B \bmod C.$$

Then the larger of the arguments A and B is loaded into RAM8 (Fig.5), after what the higher 24 bits of second argument are loaded into 24-bit register ER. Then we multiply the higher 6 bits of the first argument with the second argument, and do the modular reduction step. For that we use binary search algorithm to find out the higher bits of constant m what we must use in reduction step. Assuming $A<B<C$, modular multiply consists of the following steps:

```
1. RAM8 := B
2. C := -C
3. S := length of A in 24 bit fields
4. ER := A[S]
5. for I=1 to 4
6.       CALL MODMUL
7. S:= S -1
8. if S ≠ 0 GOTO 4
```

MODMUL:

```
1. RAM8 := ER(higher 6 bits) • RAM8
2. Find the largest 7-bit m using binary search so
   that  $C \cdot m + RAM8 \geq 0$
3. Reduce $RAM8 := RAM8 + C \cdot m$
4. Shift ER 6 bits
5. RETURN
```

The binary search algorithm starts with the highest bit and then moves through all bits. To speed up the calculations we use only the higher 96-bit part of RAM8 for compare. Only when $C \cdot m$ and RAM8 are "almost equal" the full compare is used. Fortunately, this happens very rarely (with the probability 2^{-89}) and can be safely ignored in speed calculations. The number N of cycles for multiplication can be calculated as follows:

$$N = (R + R + 1 + m) \cdot R \cdot L / m,$$

where:

N -- number of cycles for multiply,

m -- ALU multiplier length

R -- length of register in RAM fields

L -- length of one RAM field in bits.

In our case:

$$N := (8 + 8 + 7) \bullet 8 \bullet 96 / 6 := 23 \bullet 8 \bullet 16 := 2944$$

using 25 MHz clock we get 8491 modular multiplications per second.

The small register can also be used for right shifting arguments in divide operations and for bit operations.

7.4 I/o and IDEA

The I/O control structure takes care of all I/O and starts IDEA cipher process when the circuit is in block encryption mode. The circuit has two I/O modes. One is active when command fetcher has been stopped by the WAIT micro-command. During this it is possible to read out all internal registers. The other is active when circuit is in I/O mode. I/O mode lets user read out only last 2 data registers. This is necessary to avoid the leak of secret information from the circuit.

In IDEA mode the I/O starts with reading and filling temporary input registers. After the last word of 64 bits has been read the IDEA transform starts and the temporary registers are ready to accept new input data. After IDEA has finished, it gives out the IDIO_RDY signal and fills the output registers, what can then be read out from the circuit. All this is necessary, because IDEA transform is rather fast. During 50 cycles of operating time it must be possible to read in next 4 words and read out previous ones. As one I/O operation takes 4 cycles, the total time for I/O is $4 \bullet 4 \bullet 2 = 32$ cycles. The encryption speed will be 32 Mbit/s using 25 MHz clock.

The I/O control has at its final stage the possibility to select between byte-parallel, word-parallel and serial transfer mode.

The IDEA control system takes us through the IDEA transform in 50 cycles. To see how it is achieved let us first look at IDEA cycle on Fig. 6. Here and after F4 stands for Fermat’s 4th digit, namely $2^{16}+1$. The IDEA transform consists of 8 similar rounds and output transform. One IDEA round contains 6 multiply and 4 add operations. It transforms 64 bit of input data to 64 bit of output data using 96 bits of key information. Output transform contains only the upper 2 additions and multiplications.

The time-consuming operation here is modular multiply. Two16 bit additions at beginning are cheap and can be done in parallel with multiplications. XOR is also no problem.

As it was pointed out before we use low-high algorithm for modular multiplication. This algorithm for $C := A \bullet B$ mod F4 consists of the following steps:

```
1. D := A • B,
D.lo := D(15..0),  D.hi :=D(31..16)
2. if(D.lo ≥ D.hi)            C := D.lo - D.hi
   else                   C := D.lo - D.hi +  F4
```

When describing IDEA datapath we noted that ALU could be used for two 16-bit multiplications in one cycle. So the first step of the algorithm is quite easy. For second step we use ALU units alu24.1 and alu24.3 for the M1 and alu24.2 and alu24.4 for the M2 multiplication. They both work alike so I will take a look only at the first multiplication. Alu24.1 calculates D.lo-D.hi, alu24.3 calculates D.lo-D.hi+F4. It is possible to use them like this, because ALU units are multipliers and consist of 8 argument CSA (carry-save) adder with CLA (carry-look-ahead) at the end [HEN90]. For M3 and M4 multiplication we use all units and calculate the multiplication in the alu24.1 and alu24.3. Alu24.2 and alu24.4 will have at reduction stage the addition argument added. Then we check the result of alu24.1 and alu24.3 and select the result from alu24.2 and alu24.4. That enables us to run all multiplications in 2 clock cycles. So for each IDEA cycle we have 2•3:=6 clock cycles. As the output transform contains only the beginning of the cycle it adds 2 clock cycles to the total time leaving us with:

$$N = 2 \bullet 3 \bullet 8 + 2 = 50$$

clock cycles per IDEA transform.

Finally, the following table presents the ALU datapath schedule in IDEA mode. Here .lo and .hi represent the lower and higher part of calculation. The M1 of next cycle is selected from M1 and M1’ depending on the value of M1 according to low-high algorithm. M2 is selected similarly. M3 is selected from M3’’ and M3’’’ depending on the value of M3. M4 is selected as M3.

nr	alu24.1	alu24.3	alu24.2	alu24.4
1	T1 := I1•K1	T2:=I1•K1	T3:=I4•K4	T4:=I4•K4
2	M1:=T1.lo-T1.hi	M1’:=T2.lo-T2.hi+F4	M2:=T3.lo-T3.hi	M2’:=T4.lo-T4.hi+F4
3	T1:=X1 • K5	T2:=X1 • K5	T3:=X1 • K5	T4:=X1 • K5
4	M3:=T1.lo-T1.hi	M3’:=T2.lo-T2.hi+F4	M3’’:=T3.lo-T3.hi+X2	M3’’’:=T4.lo-T4.hi+X2+F4
5	T1:=A3 • K5	T2:=A3 • K5	T3:=A3 • K5	T4:=A3 • K5
6	M4:=T1.lo-T1.hi	M4’:=T2.lo-T2.hi+F4	M4’’:=T3.lo-T3.hi+M3	M4’’’:=T4.lo-T4.hi+M3+F4

Table 1. IDEA datapath schedule.

The keys are selected from RAM128 with the possibility to change between the starting location externally before each transform, thereby selecting between encryption and decryption.

7.5 Self Test

The circuit contains self-test system for evaluating error condition and verifying hardware. To accomplish it we use state hashing of all FSMs. The bits are fed into 8 bit analysers running in parallel. After 255 states the registers are shifted into 32 bit analyser and the readout of that analyser is possible in test mode. Datapath can be checked by running arithmetic test program and IDEA transform. The state hashing help us to verify that the desired result were achieved in the correct way. By making the source code available it is possible by checking the state hashing to verify that the hardware has not been tampered with to include backdoor.

7.6 Future Directions

This circuit has several deficiencies:

1. No internal code ROM. This makes it easy for the attacker to rewrite control programs.

2. No internal cryptographically secure random number generator. It has a built-in PRNG for testing purposes only. External random generator must be used to run it in real applications. But the attacker can manipulate external generators.
3. The module length is too short- only 768 bits. For RSA to be secure it should be at least 1024 bits. In the next version we will double the module length by adding second ALU unit. This will also increase IDEA speed two times. Also by running two ALUs in parallel and using Chinese Remainder theorem we can increase RSA decryption speed by a factor of 4.
4. Datapath design is not the best. Registers should be added to increase the speed, also the layout information should be extracted from VHDL files to produce compact layout. This was one important by-product of the project. By using VHDL generate statements and some control pragmas the program extracts layout control information from source file. This information is then converted into Cadence tile generator form to produce layout for datapath.
5. The possibility to control clock frequency with built-in PLL synthesiser should be included to enable us run the circuit at lower speeds what is crucial for mobile equipment where high encryption speeds must give a way to low power consumption.
6. Some Message Authentication Code calculation should be included to internally test the integrity of bypassing data.
7. The support for signed public key database should be included.
8. The switch to faster technology should increase clock speed and decrease area.

7.7 Project Development

The project started on year 1993 with the financial and educational aid from Europe. TEMPUS JEP4772 project with Prof. Manfred Glesner from Germany, Darmstadt, Prof. Bernard Courtois from France, Grenoble and prof. Raimund Ubar from Tallinn Technical University gave us the tools for synthesis and mapping and the know-how.

In the beginning we chose the architecture and then wrote the circuit simulator on PC. At that time Ahto Buldas held theoretical seminars about cryptology in Institute of Cybernetics. Using the PC simulator we refined the code for calculations. At the same time we started to write the behavioural description of the circuit using SYNOPSYS development software. Whole development was carried out using VHDL language. Finally we added the self-test feature. The behavioural description is in 22 files with total size 509 KB. At the end of 1996 the circuit was ready for prototype run. The placement and routing was done with CADENCE development system. The prototype silicon run was financed by Institute of Cybernetics and manufactured via Europractice in ES2 1.0 µm technology [ES210]. We received the prototypes at the beginning of March 1997. Since then we have developed the interface to PC using 2 XILINX FPGAs and rewrote the simulator to include the possibility to download and test the code on real device [SILUR], [SVMDOK]. As the result we have found these devices comply with the expectations, with 3 out of 20 prototypes not functional due to production faults. The calculated worst-case speed was 20 MHz. As experiments showed the real maximal operating speed was 25 MHz.

Now we are developing the add-in card for PC to carry out disk and network encryption and are rewriting the simulator to allow other people to experiment with the circuit.

The circuit was presented at NORCHIP'97 conference. [NOR97]

Bibliography

[APPC] Schneier, Bruce. Applied Cryptography Second Edition: Protocols, algorithms, and source code in C. 1996, John Wiley & Sons, Inc.

[HEN90] Hennessy, J.L., Patterson, D.A., “Computer architecture: a quantitative approach”, Morgan Kaufman Publishers, Inc., 1990.

[ES210] Europractice, ATMEL 1.0 µm ECPD10 CMOS technology library databook.

[SILUR] Ahto Buldas, Jüri Põldre. Krüptokiibi silur. Programmi dokumentatsioon. DO-TD-X-09-1094.

[SVMDOK] Ahto Buldas, Jüri Põldre. Krüptokiibi emulaator. Projekti dokumentatsioon. DO-TD-X-13-1097.

[NOR97] Ahto Buldas, Jüri Põldre. A VLSI implementation of RSA and IDEA encryption engine. Proceedings of NORCHIP’97 conference. Estonia, Tallinn, autumn 1997.

[LAI91] Lai, X., “on the design and security of block ciphers”, ETH Series in information Processing, J.L.Massey (editor), vol. 1, Hartung-Gorre Verlag Konstanz, Technische Hochule (Zurich), 1992.

[RSAGER] Sedlak, H. “Theory and Single-Chip implementation of a Cryptography Processor”, Institut für Teoretische Informatik, Technische Universität Braunschweig, Germany. PhD Thesis, 1985.

[FASTMR] Ç.K. Koç, C. Y. Hung. “A Fast Algorithm for Modular Reduction”. To Appear in Computers and Digital Techniques, IEE Proceedings.

[SYSTOL] Peter Kornerup. “A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms”, Dept. of Mathematics and Computer Science, Odense University, DK-5230, Denmark.

[EXPALG] B.J.Phillips N.Burgess. “Algorithms for Exponentation of Long Integers. A survey of published algorithms.”, Digital Arithmetic Group, The University of Adelaide, Australia.1996

[PAM93] Patrice Bertin, Didier Roncin and Jean Vullemin. “Programmable Active Memories: a performance assessment”. Research report 24, Digital Equipment, Paris Research Laboratory, 1993.

[VICTOR] Holger Orup, “Victor- an efficient RSA hardware implementation” Computer Science Department, Aarhus University, Denmark, 1990

[VICHILD] Holger Orup, “Modular exponentation processor”, Computer Science Department, Aarhus University, Denmark, 1993

[HOPHD] Holger Orup, “Exponentation, Modular Multiplication and VLSI Implementation of High-Speed RSA cryptography”, PhD thesis Computer Science Department, Aarhus University, Denmark, 1995