DISTRIBUTED SECRETE SHARING APPROACH BASED ON RSA ALGORITHM

Ritesh Saraswat[1]
Ph.D Scholar, ECE Department.
JJTU, Jhunjhunu
Email: riteshsaraswat80@gmail.com

Dr. Om Prakash[2]
Professor, ECE Department.
JJTU, Jhunjhunu
Email: om4096@gmail.com

Abstract - This paper summarizes the study and analysis of cryptographic design techniques and their implementation on an FPGA board.The study began with the understanding of a popular HDL language, namely, Verilog. Based on the study an implementation of a modular cryptosystem based on the RSA and generic upto a 256 bit modulus was realized. Optimal techniques for developing a high speed RSA cryptosystem is presented in this paper. This work is implemented on Xilinx based ISE toolkit. However ***for validation purposes other simulators such as ModelSim was also used. However, the simulations presented in this work utilizes the Xilinx ISE 10.1 Simulator environment. The Xilinx XST 10.1 was used in the synthesis of the implementation.***

I.INTRODUCTION

FPGAs are gaining importance both in commercial as well as research settings. The former appreciate the short turn-around times, the lack of Non-reccuring Engineering costs for small volume production and the easy prototyping. Since the technology is far more affordable than custom-manufactured ASICs, even smaller companies can take advantage of the capabilities of large-scale integration.The choice of reconfigurable logic as a Target Platform for cryptographic algorithm implementations appears to be a practical solution for embedded systems and high- speed applications [1]. Reconfigurable hardware devices are usually distributed in a large ge- ographic area and operated over public networks, making on-site configuration inconvenient or infusible. Therefore, robust security mechanisms for remote control and configuration are highly needed.

The RSA algorithm is a secure public key algorithm if the modulus size is sufficiently large. It can be used in these applications as a method of exchanging secret information such as keys and producing digital signatures. However, the RSA algorithm is very computationally intensive, operating on very large integers. The RSA algorithm has been adopted by many commercial software products and is built into current operating systems by Microsoft, Apple, Sun, and Novell. Commercial Application Specific Standard Products (ASSPs) like the security processors offered by several vendors have a much higher RSA performance than software implementation [2].

In this work, the objective is to design a generic high frequency version of the RSA with the largest possible modulus on the Virtex II pro FPGA using the Verilog HDL. In this regard, a generic cryptosystem with the capabilities of handling a 256 bit modulus was synthesised with some of the best algorithms available for the intrinsic arithmetic operations involved, such as generic multiplication with a 512 bit product output, 256 bit modular exponentiation, 256 bit non-restoring division for the generation of modulus from a divident of size upto 512 bits and 256 bit modular inversion.This paper focuses primarily on cryptosystems which involve modular arithmetic. Although a key generation component was implemented, the 128 bit prime keys used in the generation of a 256 bit modulus was assumed to be known prior to the implementation. This is because the implementation of the multiplication hardware (recur- sive Karatsuba-Ofman [3]) was too large and occupied around 40 % of the available slices in the Virtex II pro FPGA. Hence the implementation of an RNG (Random Number Generator) hardware as well a primality tester was excluded for compactness.

II. THE RSA ALGORITHM

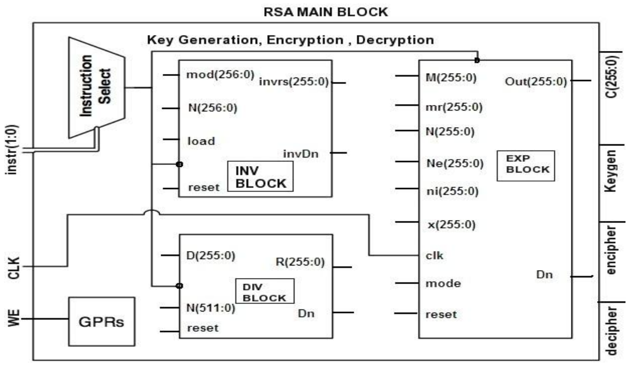


Fig. 1 Layout of the RSA implementation on FPGA

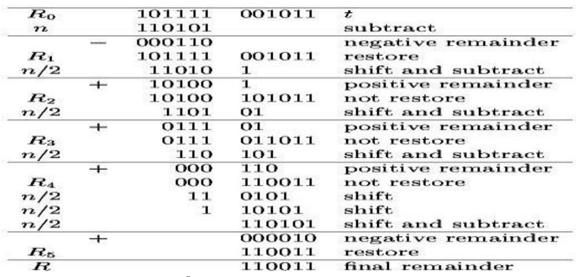
The plaintext, the value of encryption exponent *e*, the values of the 128 bit prime keys *p* & *q* for a 256 bit modulus *n* are expected to be known prior to the implementation in this this layout. The reason for this is that the Montgomery exponentiation design which utilizes the Karatsuba-Ofman multiplier with 512 bit product output, requires 69% of the available slices in the Virtex II pro FPGA which accounts for 9498 slices of the available 13696 slices with a 256 bit modulus. Hence, there leaves space only for implementing the top integration level module and the modules for inversion and modulus retrieval. However, with a view to make the entire cryptosystem efficient, with a high throughput and high speed, we have tried to optimize the modules for consuming the least possible resources available in the FPGA.

The complete system with the RSA integration module takes 90% of the slice count which is 12,374 out of 13,696 available slices in the Virtex II pro, hence producing a large design relative to this FPGA. As a result, an RNG(Random Number Generator), for both generation of prime keys and the value of encryption exponent *e*, and a *primality tester* which would create the prime keys was out of the scope for this implementation. In the above Fig, 1 , one can observe that the montgomery exponentiation module has a dual mode of functionality, i.e., it doubles up as a direct multiplier producing upto 512 bit products of which only a 256 bit product is produced for the value of modulus *n*. This is largely due to the modular structure of Verilog. A module that is instantiated at a top level cannot be shared with a submodule. Hence if the montgomery circuit requires the functionality of simple multiplication, it has to instantiated within the modular montgomery exponentiation circuitry. If resources are to be limited, then there occurs the need for a dual mode functionality.

The inversion module in the above layout utilizes a highly optimized version of the Extended Euclidean Algorithm which modifed by Laszlo hars [6]. This algorithm does not require a single division operation nor multiplication operation, thus making it one of the best resource efficient inversion circuits available. The division module which is used for modulus extraction uses a slight enhancement to the original non-restoring division method, whereby the subtrac- tion is performed at a bit level rather than at a word level so as to reduce the hardware resources required for calculation of partial remainders which would have otherwise involved shifting of the Dividend by an amount equal to the Divisor's bitlength. In this module, subtraction is performed at every bit, once the MSB(Most Significant Bit) of the Dividend is detected. The division module returns a 256 bit modulus output for a 512 bit Divident input.

III. NON-RESTORING DIVISION FOR MODULUS EXTRACTION

The Montgomery exponentiation and RSA require the calculation of the modulus in various steps. Although the by products of division include a quotient and a remainder, we are not interested in the quotient; we only need the remainder. Therefore, the steps of the division algorithm can somewhat be simplified in order to speed up the process. The reduction step can be achieved by making one of the well-known sequential division algorithms. In the following sections, we describe the restoring and the non-restoring division algorithms for computing the remainder of *t* when divided by *n*. Division is the most complex of the four basic arithmetic operations. Given a dividend *t* and a divisor *n*, a quotient *Q* and a remainder *R* have to be calculated in order to satisfy



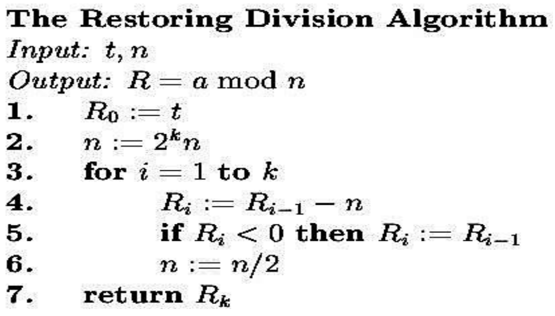
R_0		101111	001011	t
n		110101		subtract
	−	000110		negative remainder
R_1		101111	001011	restore
$n/2$		11010	1	shift and subtract
	+	10100	1	positive remainder
R_2		10100	101011	not restore
$n/2$		1101	01	shift and subtract
	+	0111	01	positive remainder
R_3		0111	011011	not restore
$n/2$		110	101	shift and subtract
	+	000	110	positive remainder
R_4		000	110011	not restore
$n/2$		11	0101	shift
$n/2$		1	10101	shift
$n/2$			110101	shift and subtract
	+		000010	negative remainder
R_5			110011	restore
R			110011	final remainder

$$t = Q \cdot n + R \text{ with } R < n \quad (1)$$

If *t* and *n* are positive, then the quotient *Q* and the remainder *R* will be positive. The sequen- tial division algorithm successively shifts and subtracts *n* from *t* until a remainder *R* with the property $0 \leq R < n$ is found. However, after a subtraction we may obtain a negative remainder. The restoring and non-restoring algorithms take different actions when a negative remainder is obtained [9]

3.1 Restoring Division Algorithm

Let R_i be the remainder obtained during the i^{th} step of the division algorithm. Since we are not interested in the quotient, we ignore the generation of the bits of the quotient in the following algorithm. The procedure given below (1) first left-aligns the operands *t* and *n*. Since *t* is 2k-bit number and *n* is a k-bit number, the left alignment implies that *n* is shifted *k* bits to the left, i.e., we start with $2^k n$. Furthermore, the initial value of R is taken to be t, i.e., $R_0 = t$. We then subtract the shifted *n* from *t* to obtain R_1; if R_1 is positive or zero, we continue to the next step. If it is negative the remainder is restored to its previous value. In Step 5 of the algorithm, we check the sign of the remainder; if it is negative, the previous remainder is taken to be the new remainder, i.e., a restore operation is performed. If the remainder R_i is positive, it remains as the new remainder, i.e., we do not restore.



```
The Restoring Division Algorithm
Input: $t, n$
Output: $R = a \bmod n$
1.  $R_0 := t$
2.  $n := 2^k n$
3.  for $i = 1$ to $k$
4.      $R_i := R_{i-1} - n$
5.      if $R_i < 0$ then $R_i := R_{i-1}$
6.      $n := n/2$
7.  return $R_k$
```

Fig. 2 Restoring Division Algorithm

The restoring division algorithm performs *k* subtractions in order to reduce the 2k-bit num- ber *t* modulo the k-bit

Figure 3: Restoring Division Example

number *n*. Thus, it takes much longer than the standard multiplication algorithm which requires *s* = *k/w* inner-product steps, where w is the word-size of the computer.

In the above example, we give an example of the restoring division algorithm for comput- ing 3019 mod 53, where 3019 = $(101111001011)_2$ and 53 = $(110101)_2$. The result is 51 = $(110011)_2$.

Also, before subtracting, we may check if the most significant bit of the remainder is 1. In this case, we perform a subtraction. If it is zero, there is no need to subtract since $n > R_i$. We shift *n* until it is aligned with a nonzero most significant bit of R_i. This way we are able to skip several subtract/restore cycles. In the average, k/2 subtractions are performed.

3.2 Synthesis Results

The synthesis results for the implementation of a modified non-restoring division with remain- der result of upto 256 bits is shown in table 1. As shown in table 1, the *div* module takes very little slice area. Further, the Maximum combinational path delay reported by Xilinx XST was 478 ηs, which makes it a very fast implementation, because of a high throughput rate of ≈ 510 Mb/s. Avoiding the use of for loops and substituting it with a combinational feedback switching technique, plays a

FPGA	Logic Slices Used	Combinational Delay(ηs)	Slices Available	Cycles (bits/s)	Throughput
Virtex II pro	455	478	13,696	1	510.75 M

role in the synthesis of large loops.

No comparison with previous implementations were available because of lack of detailed information in any of the recent work related with FPGAs and non-restoring division.

IV **MODULAR INVERSION HARDWARE**

4.1 Extended Euclidean Algorithm

Given two polynomials *A* and *B*, not both 0, we say that the greatest common divisor of *A* and *B*, is the highest polynomial *D* = *gcd*(*A*, *B*) that divides both *A* and *B*. Based on the property *gcd*(*A*, *B*) = *gcd*(*B* ±(*A*, *A*)), the revered Extended Euclidean Algorithhm (EEA) is able to find the unique polynomials *G* and *H* that satisfies Bezout's celebrated formula,

$$A \cdot G + B \cdot H = D, \qquad (2)$$

where *D* = *gcd*(*A*,*B*).

Several variations of the EEA have been proposed in the open literature [1]. EEA variants include: the almost inverse algorithm, first proposed in [22], the Binary Euclidean Algorithm (BEA), the Montgomery inverse algorithm, etc. All those algorithms show a computational complexity proportional to the maximum of A and B polynomial degrees. Algorithm 4.1 shows the binary algorithm as it was reported in . That algorithm takes as inputs the irreducible polynomial P of degree m and the field element A of degree at most $m - 1$.

Table 1: Synthesis Results for Non-restoring Division

It gives as output the field element A^{-1} such that

$$A \cdot A^{-1} = 1 \ (mod\ P) \qquad (3)$$

4.2 Binary Euclidean Algorithm

The binary GCD algorithm is an algorithm which computes the greatest common divisor of two nonnegative integers. It gains a measure of efficiency over the ancient Euclidean algorithm by replacing divisions and multiplications with shifts, which are cheaper when operating on the binary representation used by modern computers. This is particularly critical on embedded platforms that have no direct processor support for division. The following figure details the BEA, and a brief explanation regarding the functionality of the same is given.

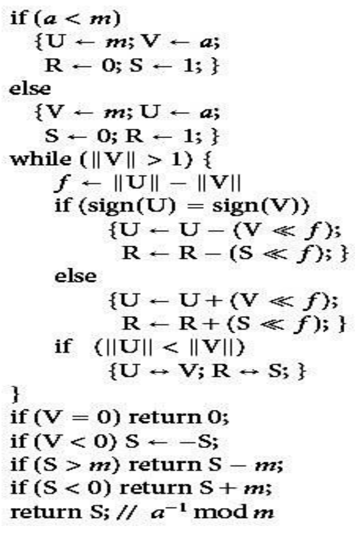
```
Require: An irreducible polynomial $P(X)$ of degree $m$, A polynomial $A \in GF(2^m)$.
Ensure: $A^{-1}$ mod $P(x)$.
 1: $U = A; V = P; G = 1; H = 0;$
 2: while ($u \neq 1$ AND $v \neq 1$) do
 3:     while $x$ divides $U$ do
 4:         $U = \frac{U}{x}$;
 5:         if $x$ divides $G$ then
 6:             $G = \frac{G}{x}$;
 7:         else
 8:             $G = \frac{G+P}{x}$;
 9:         end if
10:     end while
11:     while $x$ divides $V$ do
12:         $V = \frac{V}{x}$;
13:         if $x$ divides $G_2$ then
14:             $H = \frac{H}{x}$;
15:         else
16:             $H = \frac{H+P}{x}$;
17:         end if
18:     end while
19:     if (deg($U$)>deg($V$)) then
20:         $U = U + V; G = G + H;$
21:     else
22:         $V = V + U; H = H + G;$
23:     end if
24: end while
25: if U=1 then
26:     Return($G$);
27: else
28:     Return($H$);
29: end if
```

Fig 4: Binary Euclidean Algorithm

In steps 4 and 10, the operands *U* and *V* are divided by a as many times as possible, re- spectively. Furthermore, the variables *G* and *H* are also divided by *X* in steps 5-8 and 11-14, respectively. Notice that in case that either *G* or *H* are not divisible by a, then an addition with the irreducible polynomial *P* must be performed first. Eventually, after approximately m itera- tions, either *U* or *V* are equal to 1, which is the condition for exiting the main loop. Either *G* or *H* will contain the required multiplicative inverse. The number of iterations, N, required. by depends on several factors such as design's architecture, target platform and even the exact structure of the irreducible polynomial P(x). N can be estimated as $N \approx m$, where m is the size of the finite field.

4.3 Laszlo Hars Modification

This algorithm was proposed by Laszlo Hars in 2006 [6].The original Euclidean GCD algo- rithm replaces the larger of the two parameters by subtracting the largest number of times the smaller parameter keeping the result nonnegative: $x = x - [x/y] \cdot y$. For this we need to calculate the quotient [x/y] and multiply it with y. In this paper we do not deal with algorithms, which perform division or multiplication. However, the Euclidean algorithm works with smaller co-efficients $q \leq [x/y]$, too : $x = x - q \cdot y$. In particular, we can choose q to be the largest power of 2, such that $q = 2^k \leq [x/y]$. The reductions can be performed with only shifts and subtractions, and they still clear the most significant bit of x, so the resulting algorithm will terminate in a reasonable number of iterations. It is well known (see [12]) that for random input, in the course of the algorithm, most of the time [x/y] = 1 or 2, so the shifting Euclidean algorithm performs only slightly more iterations than the original, but avoids multiplications and divisions. See Algorithm 4.2.



```
if ($a < m$)
  {$U \leftarrow m$; $V \leftarrow a$;
   $R \leftarrow 0$; $S \leftarrow 1$; }
else
  {$V \leftarrow m$; $U \leftarrow a$;
   $S \leftarrow 0$; $R \leftarrow 1$; }
while ($\|V\| > 1$) {
    $f \leftarrow \|U\| - \|V\|$
    if (sign(U) = sign(V))
         {$U \leftarrow U - (V \ll f)$;
          $R \leftarrow R - (S \ll f)$; }
    else
         {$U \leftarrow U + (V \ll f)$;
          $R \leftarrow R + (S \ll f)$; }
    if  ($\|U\| < \|V\|$)
         {$U \leftrightarrow V$; $R \leftrightarrow S$; }
}
if ($V = 0$) return 0;
if ($V < 0$) $S \leftarrow -S$;
if ($S > m$) return $S - m$;
if ($S < 0$) return $S + m$;
return S; // $a^{-1}$ mod $m$
```

Fig 5: Laszlo Hars Shifting Euclidean Algorithm

Repeat the above reduction steps until $V = 0$ *or* ± 1, when $U = GCD(m, a)$. If $V = 0$, there is no inverse, so we return 0, which is not an inverse of anything. (The pathological cases like $m = a = 1$ need special handling, but these do not occur in cryptography.) In the course of the algorithm two auxiliary variables, R and S are kept updated. At termination S is the modular inverse, or the negative of it, within $\pm m$.

Justification for Laszlo Hars Euclidean Algorithm :

The algorithm starts with $U = m, V = a$, $R = 0$, $S = 1$. If $a > m$, swap (U, V) and (R, S). U always denotes the longer of the just updated U and V . During the course of the algorithm U is decreased, keeping $GCD(U,V) = GCD(m, a)$ true. The algorithm reduces U, swaps with V when $U < V$, until $V = \pm 1$ *or* 0 : U is replaced by $U - 2^k V$, with such a k, that reduces the length of U, leading eventually to 0 or ±1, when the iteration can stop. The *binarylength* $\|U\|$is reduced by at least one bit in each iteration, guaranteeing that the procedure terminates in at most $\|a\|+\|m\|$ iterations.

At termination of the algorithm either $V = 0$ (indicating that $U = 2^k V$ beforehand, and so there is no inverse) or $V = \pm 1$, otherwise a length reduction was still possible. In the later case $1 = GCD(\|U\|, \|V\|) = GCD(m, a)$. Furthermore, the calculations maintain the following two congruencies:

$$U \equiv Ra(\text{mod } m),\ V \equiv Sa(\text{mod } m) \qquad (4)$$

The weighted difference of the two congruencies in (4.1) gives $U - 2^k V \equiv (R - 2^k S) \cdot a \ (\text{mod } m)$, which ensures that at the reduction steps (4.1) remains true after updating the corre-sponding variables : $U = U - 2^k V$, $R = R - 2^k S$. As in the proof of correctness of the original extended Euclidean algorithm, we can see that $\|R\|$ and $\|S\|$ remain less than $2m$, so at the end we fix the sign of S to correspond to V, and add or subtract m to make $0 < S < m$. Now $1 \equiv Sa$ (mod m), and S is of the right magnitude, so $S = a^{-1}$ (mod m).

The synthesis results for the implementation of a modified

Extended Euclidean with remainder result of upto 256 bits is shown below in the following table:

The comparison shows the benefits of utilizing the algorithm 4.2. The space consumption (CLBs) is definitely lower than other implementations [23, 20] . The Timing details indicate a worst case scenario, where the clock cycles used included 256 bit values for both N and modulus m. Hence if we consider an average value of $N \approx 128 bits$ which is the exponent size in RSA, we will have a timing of $\approx 6\mu S$. However, this inversion

Work	*FPGA*	*Slices Used*	*Size*	*Cycles*	*Frequency (Mhz)*	*Timings*
Current work	Virtex II pro	688	256	512(Max)	63	8.192 μS
ITMIA [23]	Virtex II pro	9945	193	40	55	0.724 μS
BEA [23]	Virtex II pro	1195	193	191	76.1	2.509 μS
Parallel ITMIA [24]	Virtex 3200E	12021	193	20	21.2	0.943 μS
ITMIA [20]	Virtex 3200E	1195	193	20	21.2	1.32 μS

circuit is used for a dual purpose; it is also used to find the inversion of r in the Montgomery reduction step. However, Itoh-Tsuji inversion circuits have the disadvantage that it can only be used for modulii with powers of 2, hence limiting it's application only to ECCs . Hence, a good space utilization has been achieved in this implementation.

V .MODULAR MULTIPLICATION HARDWARE

FFT-based Multiplication Algorithm

There are many Fourier primes, i.e., primes p for which FFTs in modulo p arithmetic exist. Moreover, there exists a reasonably efficient algorithm for determining such primes along with their primitive elements [31]. From these primitive elements, the required primitive roots of unity can be efficiently computed. This method for multiplication of long integers using the fast Fourier transform over finite fields was discovered by Schönhage and Strassen [45]. It is described in detail by Knuth [19]. A careful analysis of the algorithm shows that the product of two k-bit numbers can be performed using O(k $logk$ $loglogk$) bit operations. However, the constant in front of the order function is high. The break-even point is much higher than that of Karatsuba-Ofman algorithm. It starts paying off for numbers with several thousand bits. Thus, they are not very suitable for performing RSA operations.

Laszlo Hars Inversion Implementation

Layout of Modular Inversion

The Layout for the inversion module is shown below

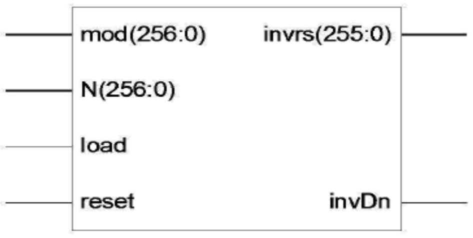


Fig 6 : Modular Inversion Layout

Table 2 : Synthesis Results for Modular Inversion

This module contains a for loop, which when rolled out in a circuit implementation, per- forms the operation of finding the bitlength. Since, finding the bitlength is pivotal to this mod- ule, a switching modulus style of looping was avoided. Ports are described as follows,

Mod takes in a 256 bit modulus ,N is given a 256 bit Number whose inverse is to be found,*load* functions as a 1 bit loading switch which initializes the algorithm 4.2,

reset is a switch which functions as an internal trigger that is used to simulate the while loop in the above algorithm.

invrs contains the result when all the operations are complete.

invDn serves as a notifier to a sequential algorithm above.

Hybrid Karatsuba-Ofman Implementation:

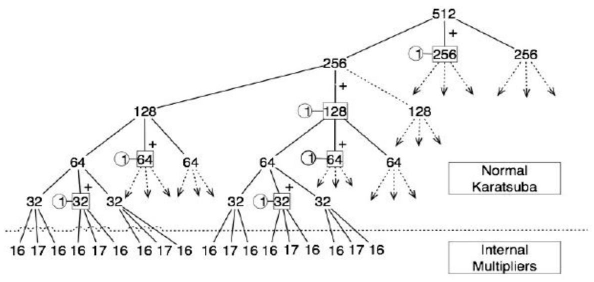


Fig 7 : Abstract 512 bit Hybrid Karatsuba Structure

The calls that are specially marked indicate an optimization step that generalizes the im- plementation. The optimization step removes the Most Significant Bit (MSB) from the sum of the parts in the recursive call. The step ensures an even division and a reduction in spe- cialized hardware that would have been necessary if the multiplication steps were sequentially performed. Hence this optimization is beneficial for the sequential version of the recursive karatsuba multiplier. However, only a slight benefit occurs for a parallel implementation. For the bit that was removed, a partial product is added by means of the following observation 5.4. Assume that the addition result is stored into SumX and SumY for the first and second addition operations respectively in the Karatsuba-Ofman Algorithm.

If MSB(SumX) = 1 and MSB(SumY) = 1 then, newProduct3 = SumX + SY,

where SX and SY are SumX and SumY with the MSB removed, respectively

If MSB(SumX) = 0 and MSB(SumY) = 1 then, newProduct3 = SumX,

If MSB(SumX) = 1 and MSB(SumY) = 0 then, newProduct3 = SY,

else newProduct3 = 0

Observation for MSB bits of the two Sum results in Karatsuba-Ofman:

This observation arises, from analysis of the carry bit in the partial products that are formed during any normal multiplication. Thus, the objective of this observation is only to remove the MSB bit from the sum which results initially causes an extra bit during multiplication. If the MSB bits were present in the sum result , then the even-ness of the Karatsuba division is lost. As a result, the extra bit requires a call to an instance of the recursive hardware with extra bits. This carries on to the sub-levels of the implementation. Hence, such a situation wasavoided.

VI Modular Exponentiation Hardware

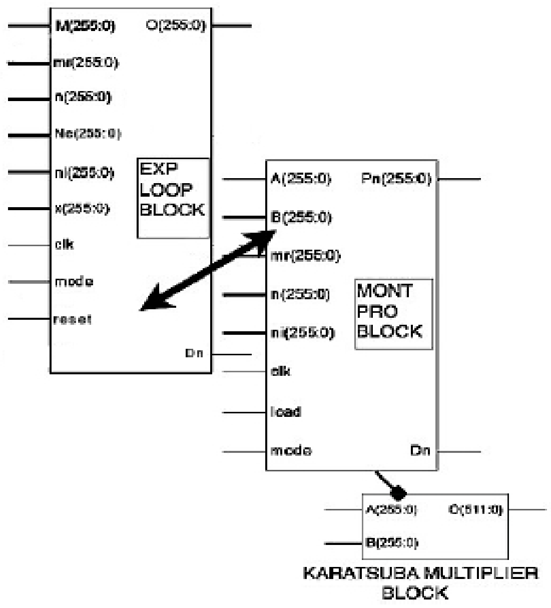
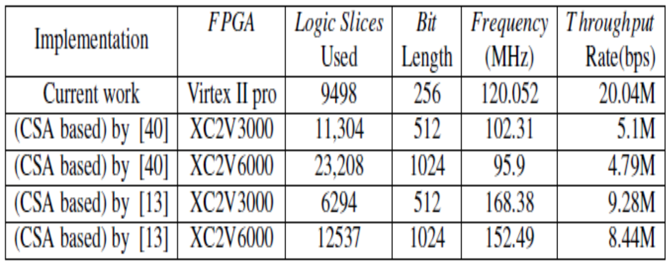


Fig 8 : Montgomery Exponentiation implementation

The synthesis results for the implementation of a modified Extended Euclidean with remainder result of upto 256 bits is shown below in the following table:



Implementation	*FPGA*	*Logic Slices* Used	*Bit* Length	*Frequency* (MHz)	*Throughput* Rate(bps)
Current work	Virtex II pro	9498	256	120.052	20.04M
(CSA based) by [40]	XC2V3000	11,304	512	102.31	5.1M
(CSA based) by [40]	XC2V6000	23,208	1024	95.9	4.79M
(CSA based) by [13]	XC2V3000	6294	512	168.38	9.28M
(CSA based) by [13]	XC2V6000	12537	1024	152.49	8.44M

Table 3 : Synthesis Results for Montgomery Modular Exponentiation

Since, this implementation uses the *Karatsuba − O f man* technique discussed earlier, the same uses only a single clock cycle to generate a 512 bit product. However, the Montgomery Product step involves 3 calls to the *Karatsuba Multiplier* Block. Hence 3 clock cycles are required for computing the Montgomery product. Further, the Montgomery Exponentiation uses the MSB-First Binary Exponentiation technique. With a worst case scenario of a 256 bit exponent e, we have by 6.3 , m = 256, $H(e)$ = 256 if all of the bits are 1s ; $P(e)$ = 510. Hence there are 510 calls to the Montgomery product. However we must consider the final call to the Montgomery product. This operation is trivial , however it consumes 3 extra cycles. Therefore, the total number of cycles for the exponentiation operation can be calculated as (510 x 3)+3 = 1533 cycles, if the addition operations are considered to be trivial.

In, the above table, the current results are compared with 512 bit and 1024 bit implementa- tions of RSA. It is evident from the above table that the present implementation gives a better throughput. However, the slice consumption for this implementation was comparatively higher and the bit lengths were very low in this work, which makes the comparison difficult to estimate the results. Further, the Maximum frequency reported by Xilinx ISE was 120 Mhz, which saturates the 100 Mhz bus frequency of the Virtex 2 pro board. Hence, the maximum frequency possible for this implementation was limited by the available *FPGA*.

VII. RSA IMPLEMENTATION

This paper presented some of most efficient techniques available for creating and developing a cryptosystem for modular arithmetic. Barring some of the initial conditions such as Random Number Generation and Primality testing, an RSA modular crypto engine which is generic upto a 256 bit modulus was synthesized. The results of this synthesis is presented in this chapter. The RSA may be implemented both as a low throughput system with less number of Logic Slices or as a high throughput system with larger number of slices. The latter method was preferred for this implementation because of a greater attention to speed. The objective of a generic RSA with upto a 256 bit modulus was thereby achieved.

However, it must be reminded that a Virtex II pro FPGA with speed grade -7 on a Digilent board was utilized for this implementation. This is because the Virtex II pro FPGA contains 13,696 available slices which accounts to ≈ 3424 CLBs (Configurable Logic Blocks) out of which 90 % was utilized(12,374 slices). Hence ≈ 3094 CLBs were utilized in this process.

A sample simulation of the RSA implementation is shown in figure

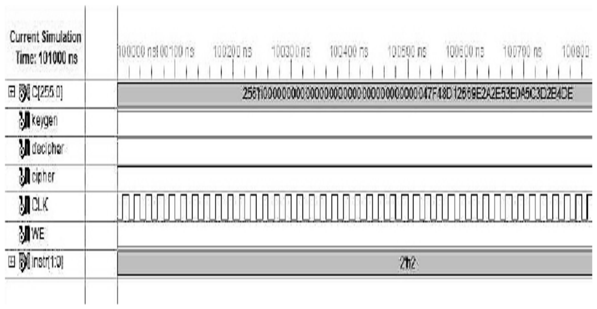


Fig 9 : RSA Implementation Simulation

The inputs to the RSA Engine were stored in a register block of size 3x256 bits.

The values stored are listed corresponding to the RSA algorithm equivalents.The primary keys *p* & *q* were 128 bit registers, which utilized a third of the register block. The value of *p* was 113680897410347 in decimal.

The value of *q* was 7999808077935876437321 in decimal.

The value of encryption component *e* is also initialized in this register block. It can be upto 256 bits. The value of *e* was 97 in decimal.

The port *instr* was used for a manual control of the Key Generation, Encryption and Decryption process. Additionally, if the *instr* is given the value zero for resetting the complete cryptosys- tem.

The bits for *keygen*,*cipher and decipher* indicate completion of key generation,encryption and decryption process.

The *WE* bit was used for re-initializing the register block with the default values. The value of plaintext *P* was a1124634758798086756746464764 in hexadecimal.

The resulting cipher *C* was obtained as 47f48d12669e2a2e53e0a5c3d2b4de in hexadecimal.

VIII. SYNTHESIS RESULTS

The synthesis results for the implementation of a modified Extended Euclidean with remainder result of upto 256 bits is shown below in the following table:

Logic Component	*Used*	*Available*
Number of Slices	12,374	13,696
Number of Slice Flip Flops	4,121	27,392
Number of 4 input LUTs	23,810	27,392
Number of bonded IOBs	263	556
Number of Multipliers 18x18	81	136

Table 4 : Synthesis Results for the RSA cryptosystem

The value of the plaintext was initialized prior to implementation. The input range can be upto 255 bits (adjusting for modulus). Alternatively an input port of 256 bit width can also be specified for this purpose.

The key-generation step initially performs two multiplication operations each. Each such operation takes 1 clock cycle in this implementation.The next operation performs a non-restoring division operation and an inversion operation in parallel, since both modules were implemented separately. The results were used for Montgomery exponentiation in the next stage of encryp- tion and subsequent decryption. Both modules consume 512 clock cycles in the worst case.

The encryption and decryption operations takes one initial remainder calculation and an exponentiation operation to complete.Thus, the encryption and decryption operations are the most relavant among all other operations (1533 cycles).

Throughout the implementation, the register block was used for primary key, decryption and plaintext storage. During synthesis , this block was converted to registers , which contributed to the overall slice count.

IX .CONCLUSION

Since the possiblity of this cryptosystem being used as a subsystem for implementation in an Embedded system is imminent, the portability to EDK was also considered. The EDK provided a complete set of ip cores which allows for adding or modifying the specifications of each device that can be used on the Virtex 2 pro board.The procedure starts of with a base system build that includes options for including a soft core or hard core processor.The preference was give to a hard core processor, so that the design would be a standard.The Xilinx EDK offers the *Microblaze* processor as their solution of a soft core.The Virtex 2 pro can be configured to have upto 2 hard core processors that is based on the Power PC *PPC405*. The base system offered a serial interface via RS232 and a 100 Mbps MAC PHY ethernet interface.The serial interface was chosen to setup a mini console to start or edit settings of the implementation.This left, only the ethernet interface, as an option to be used as input to the cryptographic algorithm. However, due to the sheer size of this implementation, only a webserver was set up as initial interface to this project in Xilinx based embedded C

REFERENCES

[1] F. Rodríguez-Henríquez, N. Saquib, A. D. Pérez, and Ç etin Kaya Koç, Cryptographic Algorithms on Reconfigurable Hardware, ser. Signals and Communication Technology. Springer, 2007, vol. XXVI.

[2] J. Fry and M. Langhammer, "Fpgas lower costs for rsa cryptography." [Online]. Available: http://www.design-reuse.com/articles/6358/fpgas-lower-costs-for-rsa-cryptography.html

[3] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," English Translation in Soviet Physics Doklady, vol. 7, pp. 595–596, 1963.

[4] "Tpm specification version 1.2 revision 103:part 1, design principles," 2007. [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpm specification version 12 revision 103 part 1 3

[5] I. C.E. and L. K., "Trusted hardware: Can it be trustworthy ?" Design Automation Con- ference. DAC '07. 44th ACM/IEEE, pp. 1–4, June 2007.

[6] L. Hars, "Modular inverse algorithms without multiplications for cryptographic applica- tions," EURASIP Journal on Embedded System, vol. 2006, January 2006.

[7] 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, February 1978.

[8] D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, 2nd ed. Addison-Wesley, 1981, vol. 2.

[9] Çetin Kaya Koç, "High-speed rsa implementation," RSA Laboratories, Redwood City, CA,, Tech. Rep. TR 201, 1994.

[10] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete log- arithms," IEEE Transactions on Information Theory, vol. 31, no. 4, pp. 469–472, July 1985.