Efficient Hardware Implementation of RSA Cryptography

Mostafizur Rahman, Iqbalur Rahman Rokon and Miftahur Rahman
Department of Electrical Engineering and Computer Science
North South University
Dhaka, Bangladesh
mz.rahman@inbox.com, {irahman,mrahman}@northsouth.edu

Abstract— **This paper presents the design and implementation of a RSA crypto accelerator. The purpose is to present an efficient hardware implementation technique of RSA cryptosystem using standard algorithms and HDL based hardware design methodology. The paper will cover the RSA encryption algorithm, Interleaved Multiplication, Miller Rabin algorithm for primality test, extended Euclidean math, non restoring division and Verilog HDL based hardware implementation in FPGA device of the proposed RSA calculation architecture. The results of fast implementations of RSA architecture using Xilinx's Virtex FPGA device are presented and analyzed. Finally, conclusion is drawn, which highlights the advantages of a fully flexible & parameterized design.**

Keywords-crypto accelerator, FPGA, RSA, verilog.

I. INTRODUCTION

The RSA (Rivest, Shamir, Adleman) algorithm is a secure, high quality, public key algorithm. However, the RSA algorithm is very computationally intensive, operating on very large (typically thousands of bits long) integers. One way to address this problem is to apply cryptographic hardware. A RSA accelerator which works like co-processor can provide means of performing the computationally expensive workload that usually accompanies various algorithms and protocols. RSA Cryptographic accelerator can provide usefulness on two fronts. First and most noticeable is increased speed. The second benefit is a spin-off of the first one: By reducing the workload on the system's CPU, accelerators allow the system to be used more efficiently for other tasks.

This paper describes an efficient design and implementation technique of an RSA accelerator. It is organized as follows: In Section II, we introduce basics on RSA algorithm. Then, in Section III, fundamental algorithms that were used in the designed are presented. Section IV deal with design architecture. A top level view of the design and the hierarchy is shown. Then in section V, we discussed implementation strategies. Detail of design architecture and hardware blocks are shown. We explain how parallel computation can generate faster results, efficient methods of implementing those hardware blocks and the data path. Further on, in Section VI, implementation results are presented and conclusion is drawn.

II. RSA BASICS

The basics of RSA algorithm is as follows-

KEY GENERATION

Select p,q p , q both prime, $p \neq q$
Calculate $n=pxq$
Calculate $\phi(n)=(p-1)x(q-1)$
Select integer e $gcd(e,\phi(n))=1, 1<e<\phi(n)$
Calculate d

Public key: $KU = \{e,n\}$
Private key: $KR = \{d,n\}$

ENCRYPTION

Plaintext: **M** $\mathbf{M} < n$
Ciphertext: **C** $\mathbf{C = M^e \pmod n}$

DECRYPTION

Plaintext: $\mathbf{M = C^d \pmod n}$

III. FUNDAMENTAL ALGORITHMS

In this section, algorithms that were used for exponential calculation, modular multiplication, division, prime number testing and GCD calculation are presented.

A. Exponential Calculation[1]

Algorithm 1 is optimized version of RL square algorithm that is used for exponential calculation.

Algorithm 1. Square and multiply (optimized version)

Input: a, e, m,firstOne, Output: $p = a^e \bmod m$, Require: $e > 1$
1: p = a
2: for j = firstOne downto 0 do
3: p = p * p mod m
4: if(e[j] == 1) then
5: p = p * a mod m
6: return p

B. Interleaved Modular Multiplication[2]

The basic idea of this algorithm is to interleave multiplication and reduction such that the intermediate results are kept as short as possible. The computation of *P* requires *n* steps and at each step the following operations are performed:

- A left shift: *2*P*
- A partial product computation: *xi* Y*
- An addition: *2*P+ xi* Y*
- At most 2 subtractions:
  If (P>M) then P=P-M;
  If (P>M) then P=P-M;

The partial product computation and left shift operations are easily performed by using an array of AND gates and wiring respectively.

C. Division

The most common implementation of digit recurrence division in modern microprocessors is *SRT* division and non SRT division. Non-restoring division [3] is similar to restoring division except that the value of 2*P[i] is saved, so D does not need to be added back in for the case of TP[i] ≤ 0. Non-restoring division uses the digit set {−1,1} for the quotient digits instead of {0,1}.

Algorithm 2: Binary (radix 2) non- restoring division

Input: N, D, Output: P, q
1: P[0] = N, i = 0
2: while i < n do
3: if P[i] ≥ 0 then
4: q[n-(i+1)] := 1
5: P[i+1] := 2*P[i] – D
6: else
7: q[n-(i+1)] := -1
8: P[i+1] := 2*P[i] + D
9: i := i + 1
10: end while

D. Primality tester[4][5]

Prime number test is the critical part of RSA key generation. We implemented the algorithm developed by Miller and Rabin. It exploits Fermat's theorem that states (a^{n-1} mod n) = 1, if n is a prime. Miller and Rabin's primality test algorithm is as follows:

Algorithm 3: Miller and Rabin's primality test

Input: n, Output: isPrime
1: Find integer k, q with k>0, q odd, so that (n - 1 = 2^kq);
2: Select a random integer a, 1 < a < n - 1;
3: if a^q mod n = 1 or n – 1
4: Return isPrime=1;
5: for j = 1 to k – 1
6: if a^{q2^j} mod n = n - 1
7: Return isPrime=1;

For efficient hardware implementation, the random number n is counted as a prime only if TEST returns 10 successive "inconclusive". According to the distribution of primes, on average every 142 trials (0.4 ln(2512)) will find a prime.

E. GCD[6]

The extended Euclidean algorithm is an extension to the Euclidean algorithm for finding the greatest common divisor (GCD) of integers a and b: it also finds the integers x and y in Bézout's identity: ax+by=gcd (a,b). The extended Euclidean algorithm is particularly useful when a and b are co prime, since x is the modular multiplicative inverse of a modulo b. The algorithm is simplified by removing unnecessary variables and computations and made it more suitable for hardware implementation.

Algorithm 4: Simplified Extended Euclidean

Input m, b;
1: start: (A2, A3) <= (0, m); (B2, B3) <= (1, b);
2: loop: if (B3 = 0)
3: b <= fetch_new_b;
4: goto start;
5: if (B3 = 1) return (B3, B2);
6: Q <= ⌊A3/B3⌋; T3 <= A3 mod B3;
7: T2 <= A2 - Q*B2;
8: (A2, A3) <= (B2, B3); (B2, B3) <= (T2, T3);
9: goto loop;

The algorithm keeps trying new b until it finds gcd(m,b)=1, then it returns b and its multiplicative inverse modulo m.

IV. DESIGN ARCHITECTURE

The system architecture of the design is shown in Figure 1.

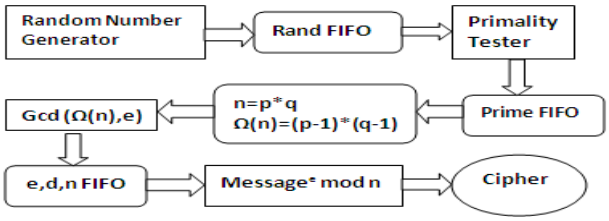

Figure 1: System Architecture of RSA

A random number generator generates pseudo random numbers and stores them in the rand FIFO. Once the FIFO is full, the random number generator stops working until a number is pulled out by the primality tester. The primality tester takes a random number as input and tests if it is a prime. Confirmed primes are put in the prime FIFO. Similar to the random number generator, primality tester starts new test only when prime FIFO is not full. A lot of power is saved by using the two FIFOs because computation is performed only when needed. When new key pair is required, the down stream component pulls out two primes from the prime FIFO, and calculates n and φ(n). φ(n) is sent to the GCD block, where public exponent 'e' is selected such that gcd(φ(n),e) = 1, and private exponent 'd' is obtained by inverting 'e' modulo φ(n). E, d and n is then stored in a bigger FIFO which holds 48 bit. Once 'n','d', and 'e' are generated, RSA encryption/decryption is simply a modular exponentiation operation.

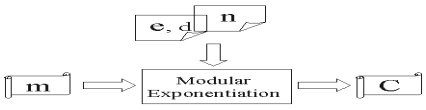

Figure 2: RSA encryption in hardware implementation

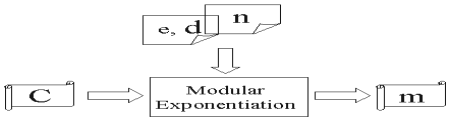

Figure 3: RSA decryption in hardware implementation.

The core of the RSA implementation is how efficient the modular arithmetic operations are, which include modular

addition, modular subtraction, modular multiplication and modular exponentiation. The RSA also involves some regular arithmetic operations, such as regular addition, subtraction and multiplication used to calculate n and φ(n), and regular division used in GCD operation. This thesis shows the detail work of those units. The design hierarchy is show below-

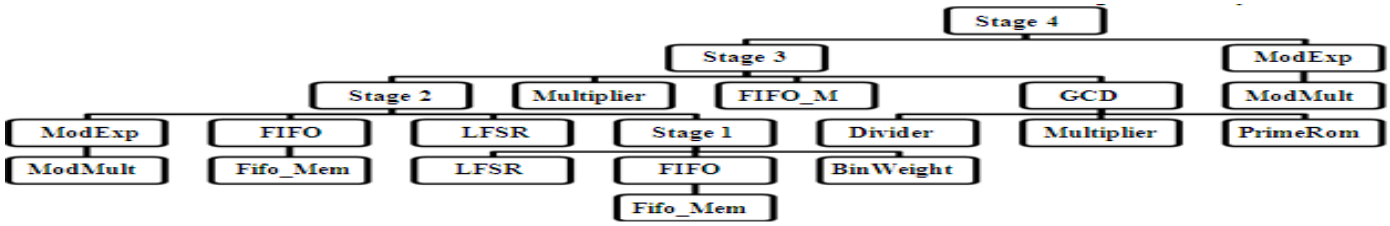

Figure 4: Design Hierarchy for RSA crypto-accelerator

V. Implementation Details of Basic Blocks

All the blocks in this design are dependent on new different operand for functionality, that is- they start working only when any of the input operands are different. This has a great impact on overall design throughput and allows maximum utilization in minimum time. Basic blocks for this design are – FIFO, LFSR, Binary weight calculator, Modular Multiplier, Multiplier, Divider, Modular Exponent calculator, Rom.

A. FIFO

A specialized FIFO is used for this particular design. The FIFO is 14 bits and avoids consecutive duplicate numbers and numbers less then 3 as input. It's another Feature is, it operates as circular queue, so- it can be used for long period without initialization. The FIFO is full when there is only one input left, Thus, helping to avoid erroneous outputs

B. LFSR

Linear Feedback Shift Register (LFSR) is used to generate pseudo random numbers. In particular, Fibonacci LFSR was implemented because it is more suitable for hardware implementation than Galois LFSR. In theory, an n-bit linear feedback shift register can generate a $(2^n - 1)$ bit long pseudo random sequence before repeating. However, an LFSR with a maximal period must satisfy the following property: the polynomial formed from a tap sequence plus the constant 1 must be a primitive polynomial modulo 2. We implemented simple 8 bit LFSR with 4 tap-in, it can be easily modified to work with 1024 bit numbers with specific tap placement as described in [7].

C. Binary Weight Calculator

Binary Weight calculator calculates number of 1's in a binary number; also it gives the position of first one as output, which is very helpful in modular multiplication and modular exponentiation unit to avoid unnecessary calculation.

D. Modular Multiplication

Using ripple carry adders, modular multiplication using shift-add multiplication algorithm is constructed. Interleaved Modular Multiplication described in algorithm 2 is used for modular multiplication, which is particularly suitable for hardware implementation, the advantages are—

- No additional Modular addition or Subtraction.
- Direct results are obtained.
- Relatively simple operation only involves shift and add.
- 2 subtraction operations can be reduced to one only.

For an 8-bit modular multiplier, inputs opA and opB are both 8 bits. However, opB might be a small number with a lot of leading 0s. In the hardware implementation, before getting into the shift-add iterations, we search for the position of the first leading 1 in opB, and set (k - 1) to be this position, a separate block called BinaryWeight is implemented to perform the operation. By doing this, unnecessary shift and modular operations are avoided, making the multiplication faster when opB is small.

E. Modular Exponent Calculator

We implemented the 8-bit and 16-bit modular exponentiation components using RL binary method, where RL stands for the right-to-left scanning direction of the exponent. Algorithm 1 showed detail of RL optimized algorithm. Similar to modular multiplication, we search for the position of the first leading 1 in exponent B and set (k - 1) to be the position. This avoids unnecessary modular squaring operations. For small exponent such as the public exponent 'e', the modular exponentiation is much faster than big exponent such as the private exponent'd'.

F. Divider

The Non Restoring Division is particularly suitable for hardware design because of its simplicity and less computation. It requires only n steps and gives quotient and remainder in relatively small time. The algorithm is described in algorithm 3. The divider accepts big dividend and small divisor, and returns a big quotient and a small remainder.

G. Rom

A Rom is implemented which holds only 32 words, prime numbers from 11- 255 are stored in ROM to assist in GCD calculation, for 1024 bit implementation, a ROM with 2048 words will be required.

VI. Implementation Details of Building Blocks

Figure 5 shows, a simplified datapath of the whole design, the clock signal is synchronous and reset is asynchronous. Initially, the reset signal has to be high for few pulses to initialize all system signals. Additionally a start signal is used as synchronous input in building blocks to control execution of the block and therefore helps in reducing power consumption.

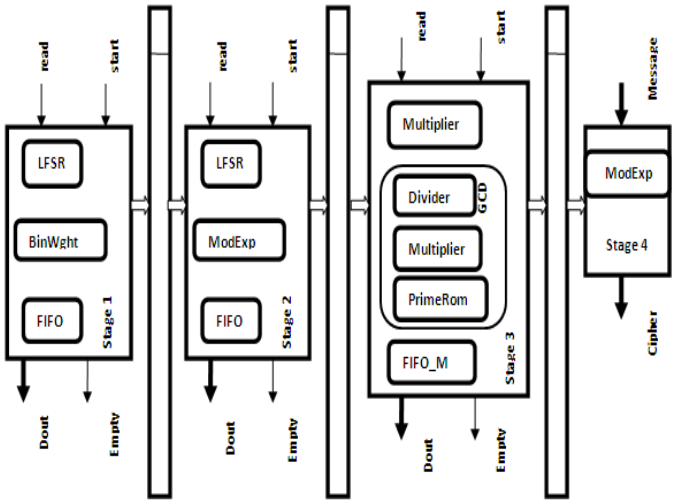

Figure 5: RSA Datapath

A. *Stage 1*

In stage 1, only odd numbers that are greater then 2 are elected to store in FIFO, this check removes unnecessary computation for even prime numbers. The numbers are then placed into BinWght for Binary weight and position of first one calculation. The 8 bit random number with its binary weight and position of first one then placed into FIFO which holds 8 words 14 bit each. The block continues to execute until the FIFO is full, when FIFO is full, it enters IDLE state.

B. *Stage 2*

Stage 2 executes miller, rabin's prime tester for prime check. Separate LFSR block is used for random number generation with different initial value, which allows the sequence to be altered and complete random checking for primes. Special care is taken to avoid random numbers less then or equal to 3, and to avoid duplicate random numbers for prime test. ModExp and InterleavedMult unit does the modular multiplication and modular exponent computations that are required. The 14 bit FIFO stores 8 bit prime number along with its binary weight and the position of first one which are 3 bits each for faster multiplication and exponentiation. The block continues to execute until the FIFO is full, which stores maximum 8 words.

C. *GCD*

GCD calculator calculates gcd of input and a prime number from ROM and finds suitable multiplicative inverse. Sub blocks of this module are Divider, primeRom, Multiplier. For 8 bit implementation only 32 prime numbers are sufficient, but for 1024 bit implementation, To get a valid b faster, pre-computation of all primes less than 2000 excluding 2 is required. Because the product of all these primes is greater than 2^{1024}, which means that any m less than 2^{1024} is prime to at least one of these primes. These primes are hard-coded in the on-chip block memory called primeRom.

Extended Euclidean requires a regular multiplier to compute Q*B2. Normal add-shift multiplication algorithm was used for that.

D. *Stage 3*

Stage 3 is the key generation stage. In this stage both private keys (d, n) and public keys (e, n) are generated. Sub blocks that do the computations are Multiplier, GCD, and FIFO. At first two different prime number p, q are fetched from Stage2 FIFO which holds 8 prime number always, result of (p-1)x(q-1) is then put into GCD module for e, d calculation. GCD calculator returns a suitable prime number e with the condition gcd(ϕ(n),e)=1 and the multiplicative modulo inverse d of e using ϕ(n). The multiplier and GCD calculator works in parallel, as the multiplier calculates pxq while gcd calculator works with ϕ(n). The result e, d, n are 16 bit each, and a larger FIFO that holds 48 bit word is required for the result. Thus the FIFO holds both private and public keys at the end of execution. The block enters is idle state when the FIFO is full.

E. *Stage 4*

Stage 4 is the higher block in the hierarchy, it encapsulates all the lower level blocks. This block takes 8 bit message (M) as input and gives 16 bit cipher text as output. A 16 bit modular exponential unit does the M^e (mod n) computation

F. *Conclusion*

In this paper, a detail implementation technique for 8-bit RSA circuit is shown. It is a full-featured parameterized design of RSA circuit including key generation and data encryption, which can be extended for 1024/2048 bit implementation of RSA cryptography. Both RSA key generation and data encryption component for this 8 bit RSA is suitable to fit into a single Xilinx Virtex II pro FPGA. The test was conducted in real hardware. Each sub-component was simulated in XST and implemented in FPGA and is functionally correct. According to the synthesis statistics, critical parts like modular multiplication and modular exponentiation takes 4.2 µs, 10µs and 45µs and 94µs for 8 and 16 bits respectively. Turnout time for 8 bit RSA implementation is 1.2 ms for 100 Mhz clock.

REFERENCE

[1] Lu, Jing, and Qian Wan, . Implementing a 1024 Bit RSA on FPGA. 03 May 2003. Dept. of CSE., Washington U. 21 Jan. 2008 <www.arl.wustl.edu/~jl1/education/cs502/doc/report.doc>.

[2] Amanor, David Narh, comp. Efficient Hardware Architectures For. 19 Feb. 2005. The University of Applied Sciences Offenburg. 07 Dec. 2007 <www.crypto.rub.de/imperia/md/content/texte/theses/dnamanorthesis.pdf>.

[3] "Division (Digital)." Wikipedia. 17 Apr. 2007. Wikipedia. 12 Dec. 2007 <http://en.wikipedia.org/wiki/Restoring_division#Restoring_division>.

[4] G. Miller, Riemann's Hypothesis and Tests for Primality. Proceeding sof the 7th Annual ACM Symposium on the Theory of Computing, May 1975.

[5] M. Rabin, Probabilistic Algorithms for Primality Testing. Journal of Number Theory, Dec. 1980.

[6] Wu, C.-L., Lou, D.-C., Chang, T.-J.," Fast Binary Multiplication Method forModular Exponentiation." Tanet, 2005. 22 Nov. 2007. <tanet2005.nchu.edu.tw/session/TANet2005Sessiondetail.pdf>.

[7] "DS257",Linear Feedback Shift Register v3.0. V-1, 28 MAR 2003. Xilinx. 04 Jan. 2008.

[8] Wockinger, thomas. High-Speed RSA Implementation.07 Jan 2005, Institute of applied information., Graz U. 27 Nov 2007 <www.iaik.tugraz.at/teaching/11_diplomarbeiten/archive/woeckinger.pdf>.