RSA & Public Key Cryptography in FPGAs

John Fry
Altera Corporation - Europe

Martin Langhammer
Altera Corporation

Abstract

In this paper an RSA calculation architecture is proposed for FPGAs that addresses the issues of scalability, flexible performance, and silicon efficiency for the hardware acceleration of Public Key crypto systems. Using techniques based around Montgomery math for exponentiation, the proposed RSA calculation architecture is compared to existing FPGA-based solutions for speed, FPGA utilisation, and scalability. The paper will cover the RSA encryption algorithm, Montgomery math, basic FPGA technology, and the implementation details of the proposed RSA calculation architecture. Conclusions will be drawn, beyond the singular improvements over existing architectures, which highlight the advantages of a fully flexible & parameterisable design.

1. Introduction

With the continual and rapid expansion of internet and wireless-based communications across open networks it is becoming increasingly necessary to protect transmitted data. RSA is a cryptographic technology that is widely used to provide necessary data protection services.

RSA relies heavily on complex large-number mathematics to provide its security services. Computationally intensive software, typically VPN applications, is used for computer-based RSA cryptography resulting in less than adequate communication performance. This can be overcome by using dedicated ASIC or ASSPs to accelerate the mathematics, but these are often expensive and inflexible as a solution. The combined cost and performance problem can be addressed by considering an FPGA-based implementation. For this, many research papers propose many different solutions, none of which to date are viable for practical implementations in FPGAs.

To achieve realistic hardware implementations for RSA, the complex math involved utilises a technique known as Montgomery Multiplication. Montgomery's techniques allow very efficient implementations of RSA-based cryptography systems. The calculations involved with Montgomery are based around the cyclic re-use of additions and the challenges faced with FPGA implementations centre around this.

FPGAs are based around SRAM technology where boolean logic functions are stored as their truth tables in numerous distributed small SRAM look-up-tables (LUTs). General-purpose routing and registering resources coupled with these LUTs means that an FPGA can be configured to implement any logic-based design. The challenge when implementing efficient designs in FPGAs is to understand the underlying fabric of the FPGA that is being targeted.

This paper will expand on the math behind RSA and Montgomery Multiplication in order to identify the critical areas when considering an FPGA implementation. The logic structure of the FPGA will also be examined in order to propose the most efficient hardware architecture. This will then be compared to existing solutions to gain a measure of the overall efficiency of the proposed FPGA-based RSA solution.

2. Fundamental Math of RSA

2.1. Public Key Cryptography

From the original RSA paper by R.L. Rivest, A. Shamir, & L. Adleman [1], an asymmetric cryptographic system was proposed that uses modular exponentiation for encryption and decryption. For ease of implementation it was proposed that both the encryption and decryption functions be identical; the only difference being the input data. The term 'asymmetric cryptographic' means that the keys used for encryption and decryption are different. In the case of RSA, the encryption key is made publicly available and the decryption key is kept private.

The encryption/decryption chain is described as follows where M is the original message data, C is the encrypted message or cipher text, e, m is the publicly available encryption key, and f is the decryption key:

$$C = M^e \bmod m$$
$$M = C^f \bmod m$$

Key generation for RSA starts with the selection of two prime numbers which are then multiplied together to produce the publicly visible modulus m:

$$m = p \cdot q$$

The strength of RSA is based on the difficulty of factoring m to discover the original primes p,q. Hence the larger the value of these primes, the harder the factorisation problem becomes. Again, typical values for these primes are 512 to 4096bits with the later exponentially stronger than the former.

Next an integer, e, that is relatively prime to $(p-1)(q-1)$, is chosen as the public key. Typically and for practical

reasons, *e* can be one of the first Fermat Numbers 3, 5, 17, 33 which will always satisfy the following:

$$mx = (p - 1) \cdot (q - 1)$$
$$\gcd(e, mx) = 1$$

gcd = greatest common divisor

To generate the private key, *f*, it is then necessary to find the multiplicative inverse of *e* mod *mx*.

$$(f \cdot e) \bmod mx = 1$$
$$f = Euc(e, mx)$$

Euc =Euclid's Algorithm

Finally, publish *e, m* as the public key and keep *f* secret. Ideally, the values of *p*, *q*, the generator primes should be destroyed.

The following is a practical example of RSA key generation and an RSA-based cryptographic exchange.

1. Generator primes create the modulus:

$$p = 47, q = 71$$
$$m = p \cdot q, m = 3337$$

2. Public key calculated:

$$(47 - 1) \cdot (71 - 1) = mx = 3220$$
$$\gcd(e, 3220) = 1$$
$$\therefore e = 79$$

3. Private key calculated:

$$(f \cdot 79) \bmod 3220 = 1$$
$$\therefore f = Euc(79, 3220) = 1019$$

4. Message, a data block to the value of 688, encrypted using the public key:

$$M = 688$$
$$C = 688^{79} \bmod 3337 = 1570$$

5. Cipher text decrypted with the private key to obtain the original data block:

$$C = 1570$$
$$M = 1570^{1019} \bmod 3337 = 688$$

2.2. Key Exchange

Beyond this asymmetric cryptography example, RSA Math has practical applications in symmetric cryptography (where the encryption and decryption keys are identical). In symmetric systems the problem of how to safely distribute or exchange private keys exists. A subset or optimised version of RSA math is used in what is known as the Diffie-Helman key exchange [2], where the public exchange of data leads to two parties securely establishing symmetric/private keys. Essentially this is an optimised way of using RSA public key technology to securely generate private keys.

The following table depicts the calculation flow for the Diffie-Helman key exchange.

Entity A	Entity B
Agree on and swap the systems variables *p & g* $p = prime$ $1 < g < p - 1$	
Generate private variable *x* $1 < x < p - 2$	Generate private variable *y* $1 < y < p - 2$
Calculate and swap the public variable *e* $e = g^x \bmod p$	Calculate and swap the public variable *f* $f = g^y \bmod p$
Calculate the shared key *k* $k = f^x \bmod p$	Calculate the shared key *k'* $k' = e^y \bmod p$
Both Entities share the same secret key $k = k'$	

The values *k & k'* can now be used as the basis for symmetric/private keys.

2.3. Modular Exponentiation

It is clear from the RSA and Diffie-Helman operations that, beyond the generation and testing of variables and primes, the fundamental operation of both of these is the modular exponentiation:

$$Z = X^E \bmod M$$

Performance of RSA-based systems is therefore tightly coupled to the calculation speed of the modular exponentiation implementation.

3. RSA Math for FPGA

3.1. Modular Exponentiation

A common basic-form method for calculating modular exponentiation is the multiply and square algorithm:

$$Z = X^E \bmod M \text{ where } E = \sum_{i=0}^{n-1} e_i 2^i, e_i \in \{0,1\}$$

1. $Z_0 = 1, P_0 = X$
2. FOR *i* = 0 to *n-1* loop
3. $P_{i+1} = P_i^2 \bmod M$
4. IF (e_i = 1) THEN $Z_{i+1} = Z_i \cdot P_i \bmod M$
   ELSE $Z_{i+1} = Z_i$
5. END FOR

Essentially, the algorithm is a running accumulation of squaring and multiplication steps. At each stage the mod *M* function is performed to keep any intermediate variables within the integer range of *M*; a second option is to allow the intermediate variables to grow and perform the mod *M* as a single final operation. Considering FPGA hardware implementations, the first option is more desirable as it will keep the multiplication functions in hardware down to a practical bit width. However, both methods use the modulus operation. The implementation of any modulus function usually involves a divide operation (to discover the remainder). A divide operation in computing hardware terms is considered complex and, wherever possible, is avoided.

The efficiency of the modular multiplier used in the multiply and square algorithm is key to the performance of RSA-based crypto systems.

3.2. Montgomery Math

Performing modular multiplication using Montgomery's techniques [3][4] removes the requirement for the division operation, leading to more a efficient FPGA-based implementation. To implement a modular exponentiation using Montgomery's techniques, a Montgomery Multiplier is used to implement the squares and multiplies used in the standard square and multiply algorithm.

To calculate the product of AB mod M, where:

$$A = \sum_{i=0}^{n-1} a_i 2^i \quad B = \sum_{i=0}^{n-1} b_i 2^i \quad M = \sum_{i=0}^{n-1} m_i 2^i$$

$$a_i, b_i, m_i \in \{0,1\}$$

Using Montgomery Multiplication the following iterative algorithm is used:

MontProd(A,B,M)

1. $S_{-1} = 0$
2. FOR $i = 0$ to $n-1$ loop
3. $q_i = (S_{i-1} + b_i A) \bmod 2$
4. $S_i = \dfrac{(S_{i-1} + q_i M + b_i A)}{2}$
5. END FOR
6. RETURN S_{n-1}

In this algorithm, any divide operations needed to keep the intermediate variables (and hence the final result) within the integer range of *M* are done using powers of 2, which is a division that is very cheap in hardware! With Montgomery Multipliers, accumulation of the product is the same as in a normal modular multiplier, i.e. an addition when a bit is set. The difference is that instead of subtracting the modulus when the intermediate variables outgrow *M* by examining the MSB end of the variables, it is done by examining the LSB end and adding the modulus. To accommodate this change the running accumulation is right shifted each iteration, and not left shifted. A benefit of using the LSB end of the variables is that no long carry chains are needed to decide whether or not to readjust for the integer range of *M*.

As a result of the optimisations in the Montgomery Multiplier, specifically the right shifts, there is an inherent factor present in the result; the actual calculation performed is:

$$S = A \cdot B \cdot 2^{-n} \bmod M$$

To remove this factor it is necessary to convert the input operands A & B into *m-residue* format. To convert to *m-residue* format use the following:

$$Ar = A \cdot 2^n \bmod M$$

As both operands need to be converted to *m-residue* format for compatibility the result from the Montgomery Multiplier will now be:

$$S = Ar \cdot Br \cdot 2^n \bmod M$$

To remove this extra factor of 2^n it is simply a matter of performing one more Montgomery Multiply by 1 (as the integer, NOT as its *m-residue* representation). This is because multiplication by a non *m-residue* value will re-introduce the 2^{-n} factor:

$$S = 1 \cdot S \cdot 2^n \bmod M$$

Implementing modular exponentiation using Montgomery Multiplication is a simple process beginning with converting *X* to *m-residue* format:

$$Xr = X \cdot 2^n \bmod M$$

which can of course be implemented using Montgomery Multiplication:

$$Nr = 2^{2n} \bmod M \qquad Xr = Montprod(X, Nr, M)$$

Then, use the standard multiply and square algorithm to bind together the Montgomery Multipliers, including the conversion to *m-residue* and the final multiplication by 1:

MontExp(X,E,M)

1. $Nr = 2^{2n} \bmod M$
2. $Z_0 = Montprod(1, Nr, M)$
3. $P_0 = Montprod(X, Nr, M)$
4. FOR $i = 0$ to $n-1$ loop
5. $P_{i+1} = Montprod(P_i, P_i, M)$
6. IF ($e_i = 1$) THEN $Z_{i+1} = Montprod(Z_i, P_i, M)$
   ELSE $Z_{i+1} = Z_i$
7. END FOR
8. $Z_n = Montprod(1, Z_n, M)$
9. RETURN Z_n

Further to the examples and explanations given, the Montgomery Exponentiator can be ordered to calculate the square first then multiplication or vice versa; both of which have various hardware implementation issues which aren't discussed in this section; but should be noted. More importantly the adder at the heart of the Montgomery Multiplier can be broken down into iterative bit blocks or words. The algorithms described thus far show the additions happening at full bit width which has a significant impact on hardware implementations. For example a 1024-bit RSA crypto system would require a 1024-bit adder, which in FPGA architectures would require a huge carry chain that will require significant pipelining resources.

3.3. Extended Montgomery Math

To reduce the problems associated with the large adders required to implement the core of a Montgomery Multiplier, it is possible to break the adder down into smaller words and use an iterative approach. This technique will use a smaller adder with carry in and carry out capability to work through the larger addition, word by word, from the LSW up. Using smaller carry chains at the core of the multiplier minimises the effort required to successfully place and route any designs when targeting an FPGA.

Beyond the advantages of limiting arithmetic operations to keep the carry chains to a manageable length, it is also desirable to match the core arithmetic to standard memory and interface sizes. 16, 32, or 64 bits are common interface sizes to and from FPGA memories, processors, and bus systems. Therefore, any of these bits sizes is optimal for the adder at the core of the Montgomery Multiplier.

As a result of the iterative use of the core addition to achieve modular exponentiation, natural word growth will occur before any reduction can be applied to correct for the modulus. To make head room for the extra 1.5bits (2 in hardware terms) that can be gained from each iteration of the Montgomery Multiplier, Blum recommends appending the extra bits to the top of any input exponent data and allowing two extra iterations of the exponentiation algorithm to keep in range of the modulus [4]. Just by automatically increasing the number of bits to the next convenient word boundary will result in simpler and smaller FPGA hardware.

The following section will discuss further the specifics of a proposed architecture for a Montgomery Exponentiation core.

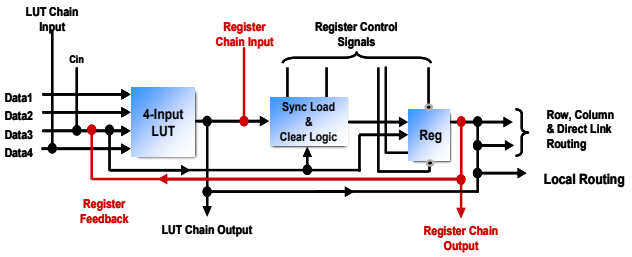
4. Proposed FPGA Architectures

4.1. FPGA Basics

For the proposed architecture, referred to as ARSA, the target FPGA technology will be the Cyclone or Stratix family from Altera. Both of theses FPGA families feature fast dedicated arithmetic logic in their base programmable fabric and dedicated memory resources. Details of the arithmetic modes will be described before any specific details of ARSA are given.

The smallest element of an Altera Stratix or Cyclone FPGA is the logic element (LE). Essentially, the LE is a four-input SRAM-based lookup table and register stage. Any combinational logic function of four inputs can be stored and registered in an LE. The LE can operate in one of two different modes. The first is normal mode, the second is dynamic arithmetic mode. The illustration below depicts the LE in normal mode:

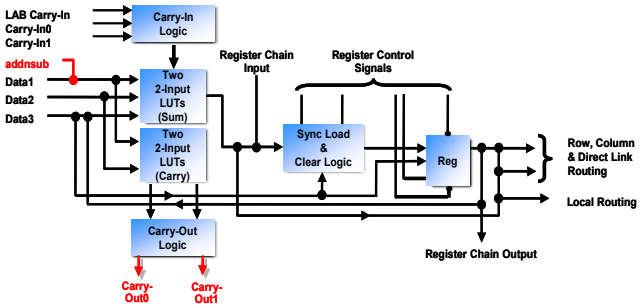
Fig 1. An Altera LE configured in normal mode.



In this mode the LE is suitable for general logic applications and combinatorial functions. The four data inputs come from the general-purpose programmable routing on a local level. Each LE in this mode also has LUT chain connections where the LUT outputs drives the fourth data input of the next LUT. Local dedicated routing provides the register controls signals and the register can be asynchronously loaded from the data 3 input.

In dynamic arithmetic mode, the LE is optimised for implementing adders, counter, and accumulators. As illustrated below:

Fig 2. An Altera LE configured in arithmetic mode.

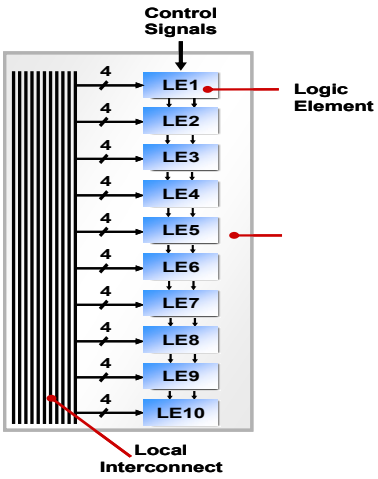


The LE in this mode is partitioned into four two-input LUTs. The first two LUTs compute two summations based on a possible carry input of 1 or 0, and the other two LUTs generate carry outputs that drive the dedicated carry-out localised routing. The carry-in signals select the relevant output from the summation LUTs and hence reduce the combinatorial path in silicon when rippling a carry chain through an adder. Likewise, the other two LUTs calculate the possible carry outs, again one for a carry in of 1, another for a carry in of 0. This technique is known as carry-select arithmetic. In this mode there are

also controls for counter enable, dynamic adder / subtractor select, and register reset and preset.

Local routing is used to connect LEs together into columns of 10 to form logic array blocks (LABs). A LAB contains localised general-purpose routing for the data inputs and dedicated routing for the carry chains and the LE control signals. Specifically, the LAB also routes a LAB carry-in signal that allows each LE to select either carry 1 or carry 2 hence selecting an entire pre-computed carry chain of ten bits. This removes a lot of LEs from the critical path making any adder performance dependant only on the propagation delay through the LAB carry-in routing. This feature gives Altera's FPGAs the ability to implement very high-speed adder- and subtractor-based logic. The diagram below shows the grouping of 10 LEs into a LAB:

Fig 3. The Altera Logic Array Block Structure.



Adder performance in FPGAs is a critical when considering a core architecture for a Montgomery Multiplier-based exponentiator.

4.2. RSA Core Architecture

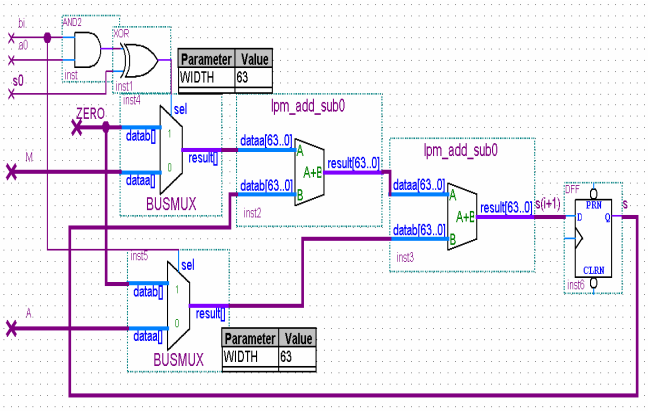
From the math behind Montgomery Exponentiation, it is clear that the iterative use of the following expressions:

$$q_i = (S_{i-1} + b_i A) \bmod 2$$

$$S_i = \frac{(S_{i-1} + q_i M + b_i A)}{2}$$

need to be bound in a controlling architecture that routes the word-wise portions of the data through them. Calculation of q_i can be achieved through simple logic functions and uses a select line to allow the modulus to route into a two-stage adder tree. The following diagram depicts the arithmetic core for the proposed ARSA.

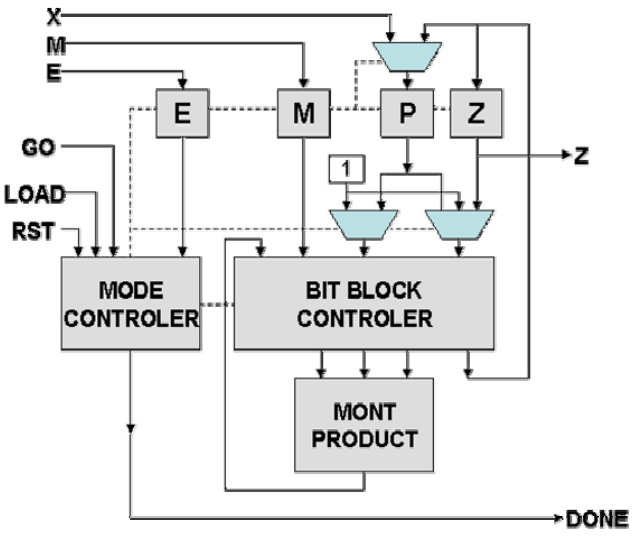
Fig 4. ARSA Montgomery Multiplier Core Logic.



Here, the adders in cascade are registered at the output before the value of S is fed back for the next iteration. In this depiction the fed back path is a conceptual one to illustrate the need for single cycle operation, whereas in the proposed ARSA core the S word-wise data will come from and go to intermediate memory stores until the full bit-length S is calculated. The adders and the intermediate stores are configurable to 16, 32, or 64 bits in width; the choice of which width obviously affects the word-wise cycle time to calculate the intermediate values of S in the Montgomery Multiplier.

To implement the complete exponentiator, the intermediate stores for the value of S can be partially merged into on-chip memories that are used to store the values of P & Z as referenced by the Montgomery Exponentiator function. From a high-level perspective the complete core for the proposed ARSA will look as follows:

Fig 5. ARSA Top Level Architecture.



It is clear that the values of Z, P, & M need to be fetched from memory to feed the Montgomery Multiplier, and that the resulting values of either Z or P need to be

written back to their respective memories. To keep the performance of the ARSA at a maximum, it is necessary to maintain single-cycle performance through this read-modify-write process. The latency associated with fetching the relevant data and writing it back to the global *P & Z* memories can be hidden by a registered pipeline. However the core addition block must maintain single-cycle performance.

Because of the single-cycle requirement of the cores the carry chain path will be a critical factor in the overall performance of the ARSA. For reference, in the highest speed grade device in Altera's Stratix family, a 16-bit adder will run at 370 MHz, and a 64-bit adder at 290 Mhz. In the Montgomery Multiplier block there are two stages of addition that must be performed, essentially making the carry chain twice as long for each of the configurable 16, 32, or 64-bit modes. Based on the Stratix performance figures it is anticipated that the even in 64-bit mode the cascaded adders will achieve 210+MHz performance.

In order to make the core generically and linearly scalable in performance, it is necessary to ensure that the most critical path in the proposed ARSA core is not one that can be modified by the generic bit word width. To ensure this, the ARSA core has been architected to make sure that the generic arithmetic and memory read and write paths are as tightly designed to use localised routing and resources as possible. This means that the most critical path in the proposed ARSA architecture exists in the control-path state machines that have to guarantee the single-cycle operation of the arithmetic logic. Currently, state machine execution has a longest combinatorial path of 5 nS, allowing 200 MHz performance. The result is that the generic settings for word width can be varied without affecting the 200 Mhz ARSA clock rate resulting in a linear 1x, 2x, and 4x performance increase.

Interfacing to the core is designed to provide the simplest connection to processors and bus systems. A memory style interface, where the values of *X, E, & M* are written in and *Z* is read out by means of an address bus and read/write line, allowing the ARSA core to interoperate with any common DMA and bus standard available. The only ancillary signal required is the GO flag to indicate that input data has been loaded and to begin execution of the Montgomery Exponentiation algorithm.

On completion, DONE is flagged high and can be used as an interrupt to read the results from the ARSA core. Due to the simple nature of the interface, it is clear that during the write and read period of the ARSA the core is idle. For applications that require continuous data flow at a rate, where the write, read, and core execution times may interrupt this flow, the ARSA can extend its internal memories to work as double, triple, quad, extending to any size buffered configurations. By accessing these memories round-robin fashion, the write, read, and execution times are converted into latency, not degradation in throughput.

Current work into the proposed ARSA architecture is focused towards using the advanced place and route technologies in Altera's FPGA design tool, Quartus II, to build an optimised 128-bit arithmetic core to achieve an 8x linear performance gain. This can be achieved by moving the position of the single-cycle register and the registers that exist at the output of the memories to break the carry chains between the larger cascaded 128-bit adders. This process of register balancing yields very successful results adding to the overall scalability of the ARSA core.

4.3. Resources and Performance Summary

The tables and expressions in this section show a summary of the resource utilisation required for each generic implementation of the proposed ARSA core and the achievable performance in the Altera Stratix and Cyclone FPGA Family.

Table 1. ARSA Core Resource & Performance.

Core	LE's	FMax MHz
ARSA 16	300	200
ARSA 32	500	200
ARSA 64	700	200
ARSA 128	900	200

The generics that control the implementation of the ARSA core are *step_size* and *rsa_size*. It is possible to change the *rsa_size* parameter on-the-fly so any references to ARSA core use the *step_size* parameter as a prefix, e.g. ARSA16 for the 16bit ARSA core. It is also assumed that the values of *E, X, & M* are all the same length in bits as is in the case of RSA decryption.

To calculate the memory requirements in bits for each ARSA core use the following expression:

$$bits = 6 \cdot (rsa\_size + step\_size)$$

Table 2. ARSA Core Memory Requirements in Bits.

	Modular Exponentiation Size in Bits				
	256	**512**	**1024**	**2048**	**4096**
ARSA 16	1632	3168	6240	12384	24672
ARSA 32	1728	3264	6336	12480	24768
ARSA 64	1920	3456	6528	12672	24960
ARSA 128	2304	3840	6912	13056	25344

To calculate the number of clock cycles required to calculate a Modular Exponentiation and hence the total time for calculation use the following expressions:

$$iters = 1 + 1.5(rsa\_size)$$

$$cycs\_iters = \frac{(rsa\_size + step\_size + 2)^2}{step\_size}$$

$$cycles = iters \cdot cycs\_iters$$

$$time\_mS = cycles \cdot \frac{1000}{FMax}$$

Table 3. ARSA Core Execution Time in mS.

	Modular Exponentiation Size in Bits				
	256	**512**	**1024**	**2048**	**4096**
ARSA 16	9.032	67.50	521.5	4099	32500
ARSA 32	5.059	35.82	268.8	2081	16380
ARSA 64	3.118	20.07	142.6	1073	8316
ARSA 128	2.241	12.38	79.95	569.4	4287

5. Discussion inc. RSA Implementations

5.1. Other Hardware Implementations

To fully appreciate the benefits of the features and performance of the ARSA core, it is useful to compare it with other existing solutions. Two recent notable implementations have been those done by Blum [4] and Daly & Marnane [5]. In both of the cases examined here, they are very large (see Table 4) and therefore expensive solutions. In almost all designs, the size of intellectual property (IP) cores, like Modular Exponentiators, seriously impacts the architecture of a system's design in two ways: (a) as the IP block becomes larger, it becomes more difficult to route the rest of the design at speed, and (b) usually IP is a smaller, albeit essential, building block of a system and therefore size and cost should be kept to a minimum. For this reason the ARSA core was originally targeted at a size-efficient implementation that is linearly generic in performance to serve as an application-specific processor.

The ARSA, Blum, and Daly & Marnane Modular Exponentiators all use different architectures. The Blum design uses a systolic array approach and the Daly & Marnane design uses a large pipelined adder approximately the size of the modulus. The major disadvantage of these two architectures is their lack of scalability. Both have been optimised for RSA sizes up to 1024 bits. For larger sizes like 2048, 4096, & 8192, as specified by IPSEC for the key exchange protocols, both architectures will grow very large; much larger than can be sensibly implemented cost effectively in an FPGA at the time of this writing. Also, in the case of the Daly & Marnane design, due to the large adder, the performance drops off with increasing bit lengths making the design untenable for large RSA applications.

The ARSA design runs at a much higher clock rate than the other two designs but only occupies a fraction of the FPGA LUT resources giving it a better overall size/speed ratio. The following section will analyse the overall size/speed ratios of all 3 designs.

5.2. Performance Comparisons & Summary

All comparisons are based on 1024-bit RSA decryption operations. The Blum and Day & Marnane designs were implemented in Xilinx FPGAs and have had their slice resource counts converted to Altera LEs. The following table details device utilisation and performance for the different designs, Blum proposes two.

Table 4. Modular Exponentiation Design Summary.

Design	LE's	MHz	Exp mS	LE x FMax
Daly	20700	50	20	1656000
Blum1	13200	72	12	950400
Blum2	9600	66	40	633600
ARSA 16	300	200	521.5	60000
ARSA 32	500	200	268.8	100000
ARSA 64	700	200	142.6	140000
ARSA 128	900	200	79.95	180000

Clearly, although the ARSA core is slower it is also significantly smaller. The ARSA architecture is based around very tight localised routing in a very small space making it possible to parallel up many ARSA cores to boost overall performance. The next table shows the reduction factors for area and speed when comparing the ARSA to the other designs.

Table 5. RSA Speed Vs LE's.

	ARSA Vs Daly		ARSA Vs Blum1		ARSA Vs Blum2	
ARSA	**Smaller**	**Slower**	**Smaller**	**Slower**	**Smaller**	**Slower**
16	69.00	26.08	44.00	43.46	32.00	13.04
32	41.40	13.44	26.40	22.40	19.20	6.72
64	29.57	7.13	18.86	11.89	13.71	3.57
128	23.00	4.00	14.67	6.66	10.67	2.00

On average there is a 2.7x performance boost when using the ARSA design. It is appreciated that the Blum and Daly & Marnane designs were implemented in slightly older technology and that this has to be taken into account. To do this, the LE x FMax product is used to remove the effects of the higher performing Altera silicon. This will put the different designs on a level playing field for comparison. The Daly & Marnane design is only a single technology step behind and it is very unlikely that it will attain the same FMax as the ARSA core. A leap in FMax performance to match the ARSA by the Blum design is also unlikely when targeting the latest and fastest FPGA silicon; but by normalising out FMax the fairest 'architecture-only' comparison can be made. The following table again shows the reduction factors for area and speed but this time independent from recent advances in FPGA technology.

Table 6. RSA Speed Vs LE x FMax Products.

	ARSA Vs Daly		ARSA Vs Blum1		ARSA Vs Blum2	
ARSA	**Smaller**	**Slower**	**Smaller**	**Slower**	**Smaller**	**Slower**
16	*17.25*	*26.08*	*15.84*	*43.46*	*10.56*	*13.04*
32	*10.35*	*13.44*	*9.50*	*22.40*	*6.34*	*6.72*
64	7.39	7.13	*6.79*	*11.89*	4.53	3.57
128	5.45	4.00	*5.28*	*6.66*	3.52	2.00

Because the ARSA core is generically scalable in performance there are a number of different speed/size ratios that can be chosen. For the 64-bit and 128-bit

ARSA core, the average performance increase for the FMax independent comparisons is 1.15x.

All of the comparisons shown are based on 1024-bit RSA decryption. The lack of scalability and the non-linear performance increase associated with the other designs means that the 2.7x and 1.15x performance gains of the ARSA core will continue to grow as the RSA word size grows due to its limitless linear scalability.

6. Conclusions

Although raw performance is desirable it is not necessary for all uses of RSA-based technology. A combination of scalability and linearly increasing performance is more useful when considering the range of RSA-based encryption applications. These can range from low-bit rate single-channel voice communication, simple key exchanges, credit card transactions, to high-speed IPSEC key exchanges and bulk-traffic encryption on the internet. The ARSA core achieves a mix of performance and scalability to cover all of these applications suitably. By exploiting Altera's high speed arithmetic capabilities in their Stratix & Cyclone family it has been shown how to architect the best RSA solution whilst making the most out of the resources available. Silicon efficiency has a significant impact on cost beyond the obvious advantages of a scalable architecture in performance terms. Thus, the ARSA core provides a very broad range of cost/performance points. Interfaces to ARSA core have been designed in such away that connection to any standard processor or bus is as simple as possible. All of these in combination make the ARSA core the most cost-effective broad-performing RSA solution for accelerating RSA-based cryptography systems.

References

[1] "A Method For Obtaining Digital Signatures And Public-Key Crypto Systems" by R.L. Rivest, A. Shamir, and L. Adleman

[2] "Diffie Helman" by W.Diffie and M.Helman

[3] "Montgomery Exponentiation With No Final Subtractions: Improved Results" by Gachel Hachez and Jean-Jaques Quisquater

[4] "Modular Exponentiation on Reconfigurable Hardware" by Thomas Blum

[5] "Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic" by Alan Daly and William Marnane

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
literature@altera.com