Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic

Alan Daly[†] and William Marnane[*]
Department of Electrical and Electronic Engineering
University College Cork
IRELAND
[†]`aland@rennes.ucc.ie` [*]`marnane@ucc.ie`

ABSTRACT

This paper presents a review of some existing architectures for the implementation of Montgomery modular multiplication and exponentiation on FPGA (*Field Programmable Gate Array*). Some new architectures are presented, including a pipelined architecture exploiting the maximum carry chain length of the FPGA which is used to implement the modular exponentiation operation required for RSA encryption and decryption. Speed and area comparisons are performed on the optimised designs. The issues of targeting a design specifically for a reconfigurable device are considered, taking into account the underlying architecture imposed by the target technology.

I. INTRODUCTION

In a secure telecommunications network such as is increasingly required for electronic commerce and internet privacy, security requirements include confidentiality, authentication, data integrity and non-repudiation. These services are offered by public key cryptosystems, the most popular of which is the RSA encryption scheme [1]. The fundamental operation of the algorithm is modular exponentiation which is achieved by repeated modular multiplications. The Montgomery modular multiplication algorithm [2] is often used to perform these calculations. However, the high bit lengths required to provide adequate security (1024 bits is considered secure against attack in the near future), mean a high hardware throughput is difficult to achieve.

An efficient algorithm for the calculation of $(A \times B)$ mod M was developed by P. L. Montgomery [2], and forms the basis of the designs presented here. It should be noted that Montgomery's algorithm only works if the modulus is relatively prime to the radix, although this is always the case in RSA.

This paper is organised as follows: Section II. gives a brief introduction to the RSA cryptosystem and it's operation, Section III. introduces the Montgomery method of modular multiplication and presents 3 variations of the algorithm. Section IV. then explains how the Montgomery modular multiplier in Section III. may be used to implement modular exponentiation. Section V. gives some previous implementations presented by other authors, and their approaches and results. Section VI. describes how the underlying architecture of the target FPGA device may be utilised to produce an optimum design. Section VII. presents various hardware implementations of the basic Montgomery multiplier, and Section VIII. describes how the multiplier may be pipelined to exploit the maximum carry chain length available. This pipelined multiplier is then used in Section IX. to implement a full modular exponentiator capable of implementing RSA encryption and decryption. Speed and area results of the designs are presented in Section X., and finally some conclusions are drawn in Section XI.

II. THE RSA CRYPTOSYSTEM

The RSA public key cryptosystem [1] was proposed by R.L. Rivest, A. Shamir and L. Adleman in 1978. Encryption and decryption are identical modular exponentiation operations, the only difference being the inputs, making it relatively easy to implement. This modular exponentiation is performed by repeated modular multiplications. With the exponent being in the region of 1024 bits, this means that many multiplications must be performed per exponentiation. In order to achieve real-time encryption and decryption, it is therefore necessary to perform fast modular multiplications.

The public and private keys are found as follows:

Two large ($\approx$ 512bit) prime numbers, P and Q, are chosen, and their product M is computed.

$$M = P \times Q$$

The *Euler Totient Function*, $\phi(M)$ is defined as " *the number of positive integers smaller than M which are relatively prime to M*", or symbolically:

$$\phi(M) = (P-1) \times (Q-1)$$

An integer, D ($< M$), is chosen to be relatively prime to $\phi(M)$. ie:

$$\gcd(\phi(M), D) = 1$$

Then the integer, E is computed, which is the *multiplicative inverse* of D, modulo $\phi(M)$.

$$E \times D \equiv 1(\text{mod}\phi(M))$$

To encrypt a plaintext message P, the entire message must first be represented as a sequence of integers between 0 and $M-1$, and then raised to the E^{th} power modulo M. Similarly, to decrypt the ciphertext C, it must be raised to the D^{th} power modulo M. Therefore:

$$C = P^E (\text{mod}\, M)$$

$$P = C^D (\text{mod}\, M)$$

The public encryption key is then the pair of positive integers (E, M), and the private decryption key is the pair of positive integers (D, M).

The only known way of deriving D from E and M is to factor M into P and Q. Thus the security of the RSA cryptosystem ultimately relies on the inability of a potential attacker to effectively factor large integers. At present, with only currently available mathematical techniques and computing power, this is considered to be a valid assumption for integers of 1024 bits or greater.

III. Montgomery Multiplication

Modular multiplication is generally considered to be a complicated arithmetic operation because of the inherent multiplication and division operations. There are two main approaches to computing modular multiplication: (1) Perform the modulo operation *after* multiplication or (2) *during* multiplication. The modulo operation is accomplished by integer division in which only the remainder is needed for further computation. The first approach requires a $n \times n$ bit multiplier with a $2n$-bit register followed by a $2n \times n$ bit divider. In the second approach, the modulo operation occurs in each iteration step of integer multiplication. Therefore the first approach requires more hardware, while the second requires more addition/subtraction computations [3].

The ordinary modular multiplication algorithm for the computation of $(A \times B)$ mod M takes the normal multiplication method which accumulates digit products $A \times b_i$ and interleaves modular reductions to keep the result below M. These reductions are achieved by subtracting the correct multiple of the modulus from the intermediate result. This reduction is dependent on the most significant bits of the operand.

Montgomery's algorithm [2], on the other hand, reverses the order of treating the digits of the multiplicand, using the least significant bits of the intermediate result to perform an addition rather than a subtraction, and performs a shift down operation instead of a shift up operation on each iteration.

Three slightly modified Montgomery product (MonPro) algorithms are examined here, each lending to a slightly different hardware architecture.

The pseudo-code to perform the 3 MonPro operations is given in algorithms (1),(2) & (3) where:

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

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

and given that the modulus, M, is at most, a k-bit integer (ie: $0 < M < 2^k$), and $A, B < M$, then the bit length of the multiplier, n, is defined to be equal to $k+2$. It is necessary to make the bit length of the multiplier equal $k+2$ to ensure that the intermediate result, S, remains bounded (ie: $0 \leq S < 2^k$). This condition allows the intermediate result to be used as an input for the next iteration.

Algorithm 1 : MonPro1(A,B,M)

```
MonPro1 (A,B,M)
{
  $M' := \frac{M+1}{2}$ ;
  $S_{-1} := 0$ ;
  for $i = 0$ to $n$ do
      $q_i := (S_{i-1})$ Mod 2 ;                    (LSB of $S_{i-1}$)
      $S_i := S_{i-1}/2 + q_i M' + b_i A$;
  end for
  Return $S_n$ ;
}
```

Algorithm 2 : MonPro2(A,B,M)

```
MonPro2 (A,B,M)
{
  $S_{-1} := 0$ ;
  for $i = 0$ to $n-1$ do
      $q_i := (S_{i-1} + b_i A)$ Mod 2 ;            (LSB of Sum)
      $S_i := (S_{i-1} + q_i M + b_i A)/2$;
  end for
  Return $S_{n-1}$ ;
}
```

Algorithm 3 : MonPro3(A,B,M)

```
MonPro3 (A,B,M)
{
  $S_{-1} := 0$ ;
  $A := 2 \times A$ ;
  **for** $i = 0$ to $n$ **do**
      $q_i := (S_{i-1})$ Mod 2 ;                  (LSB of $S_{i-1}$)
      $S_i := (S_{i-1} + q_i M + b_i A)/2$;
  **end for**
  Return $S_n$ ;
}
```

Note that algorithm (1) requires one more iteration of the for loop than algorithm (2), and thus computes one Montgomery modular multiplication in $n+1$ clock cycles instead of n. Algorithm (1) is preferable for use in systolic structure arrays, where the multiplier is constructed from identical processing elements such as those proposed by Kornerup [4]. Algorithm (3) is a modification of algorithm (2) whereby the multiplicand A is shifted up by 1 bit to simplify the calculation of q_i. This implies that the result is a factor of 2 too large, and so the ***for*** loop must be executed an additional time to eliminate this factor. Thus, algorithm (3) also requires $n+1$ clock cycles.

Each of the MonPro functions then computes a Montgomery product of the form:

$$\text{MonPro }(A,\,B,\,M) \;= ABr^{-1} \bmod M \tag{1}$$

Unfortunately, the extra factor of r^{-1} is picked up in the calculation, and some pre- and post-calculations are required to produce the correct result. Here, r^{-1} is the inverse of r (mod M), ie: $r^{-1}r = 1(\text{mod } M)$, where r is given by:

$$r = 2^n$$

The m-residue A, of an integer $\mathcal{A} < M$ is defined as:

$$A = \mathcal{A}r \bmod M \tag{2}$$

The MonPro function can be used to convert an integer to it's M-residue as follows:

$$\begin{aligned} \text{MonPro }(\mathcal{A}, r^2, M) &= \mathcal{A}r^2r^{-1} \bmod M \\ &= \mathcal{A}r \bmod M \\ &= A \end{aligned}$$

So to convert a number, $\mathcal{A}$, to it's M-residue, A, it is necessary to compute MonPro $(\mathcal{A}, r^2, M)$.

However, the value $r^2 = 2^{2n}$ is outside the allowed range for inputs to the MonPro function $(0 < M < 2^{n-2})$, so we must compute:

$$\text{MonPro }(\mathcal{A}, 2^{2n} \bmod M, M)$$

This value of $(2^{2n} \bmod M)$ must be precomputed externally. However, this value is constant for a given bit-length and modulus, and could ideally be stored in a database along with the recipient's public key (M & E) in RSA applications.

It can be seen that the Motgomery product of 2 M-residues, A, B, is itself an M-residue, S :

$$\begin{aligned} S &= \text{MonPro }(A,B,M) \\ &= ABr^{-1} \bmod M \\ &= \mathcal{A}r\mathcal{B}rr^{-1} \bmod M \\ &= \mathcal{A}\mathcal{B}r \bmod M \\ &= \mathcal{S}r \bmod M \end{aligned}$$

so a final calculation is required to convert S back into the ordinary form of the integer, $\mathcal{S}$:

$$\begin{aligned} \mathcal{S} &= Sr^{-1} \bmod M \\ &= 1Sr^{-1} \bmod M \\ &= \text{MonPro }(1,S,M) \end{aligned}$$

A precondition for the algorithm to work is that the modulus, M, has to be relatively prime to the radix, r (ie: $\gcd(M,r) = 1$). This is always the case in the RSA cryptographic system [1] since $M = p \times q$, a product of 2 large primes, and therefore odd. And since r is a power of 2, it is always even.

Due to the fact that the operations of mod r and div r are intrinsically fast in binary systems since r is a power of 2, Montgomery's algorithm is faster and easier to compute than the 'ordinary' modular product $\mathcal{A}\mathcal{B}$ mod M. However, the additional operations of conversion to and from the M-residue format, and the precomputation of 2^{2n}mod M add extra steps to the calculation. Thus, it is not advantageous when only a single modular multiplication needs to be performed. It is more appropriate when several multiplications with respect to the same modulus are to be performed such as in the RSA modular exponentiation algorithm.

IV. Modular Exponentiation

Modular exponentiation is performed by repeated modular multiplications. There are two common algorithms which can be used: The *L-R Binary Method*, which is area optimised, and the *R-L Binary Method* which is speed optimised. These are given in algorithms (4)&(5), where $\mathcal{P}$ is the *plaintext*, $\mathcal{E}$ is the *exponent*, $\mathcal{M}$ is the *modulus*, $\mathcal{C}$ is the *constant* 2^{2n}Mod $\mathcal{M}$ (which must be precomputed), and $\mathcal{R}$ is the *result*.

Algorithm 4 : L-R Algorithm : MonExp1($\mathcal{P},\mathcal{E},\mathcal{M}$)

```
MonExp1 ( $\mathcal{P}$, $\mathcal{E}$, $\mathcal{M}$ )
{
  $\mathcal{C} := 2^{2n} \bmod \mathcal{M}$ ;
  $P$ :=MonPro($\mathcal{C},\mathcal{P},\mathcal{M}$) ;        (Mapping)
  $R$ :=MonPro($\mathcal{C},1,\mathcal{M}$) ;
  for $i = k-1$ downto 0 do
      $R$ :=MonPro($R,R,M$) ;        (Square)
      if ($\mathcal{E}_i = 1$) then
          $R$ :=MonPro($R,P,M$) ;        (Multiply)
      end if
  end for
  $\mathcal{R}$ :=MonPro($1,R,\mathcal{M}$) ;        (Re-Mapping)
  Return $\mathcal{R}$ ;
}
```

Algorithm 5 : R-L Algorithm : MonExp2($\mathcal{P},\mathcal{E},\mathcal{M}$)

```
MonExp2 ( $\mathcal{P}$, $\mathcal{E}$, $\mathcal{M}$ )
{
  $\mathcal{C} := 2^{2n} \bmod \mathcal{M}$ ;
  $P$ :=MonPro($\mathcal{C},\mathcal{P},\mathcal{M}$) ;        (Mapping)
  $R$ :=MonPro($\mathcal{C},1,\mathcal{M}$) ;
  for $i = 0$ to $k-1$ do
      if ($\mathcal{E}_i = 1$) then
          $R$ :=MonPro($R,P,M$) ;        (Multiply)
      end if
      $P$ :=MonPro($P,P,M$) ;        (Square)
  end for
  $\mathcal{R}$ :=MonPro($1,R,\mathcal{M}$) ;        (Re-Mapping)
  Return $\mathcal{R}$ ;
}
```

In algorithm (4), the square and multiply operations must be performed sequentially, and therefore the 2n multiplications must be performed in series. It does mean that both the square and multiply operations can be performed in the same single hardware multiplier, thus saving on area.

In algorithm (5), the square and multiply operations are independent, and may be performed in parallel. Thus, 50% less clock cycles are required to complete the exponentiation. However, two physical hardware multipliers are required to achieve this speed up. Therefore, the speed $\times$ area products of both algorithms are very similar.

For this paper, the R-L algorithm (algorithm (5)) will be used since the primary aim is to increase the data throughput or bit-rate of the exponentiator for real time encryption/decryption.

V. Previous Implementations

By considering Montgomery's algorithm at a bit-wise level, a Processing Element (PE) can be specified to perform the multiplication, and implement modular multipliers up to any desired length by simply combining multiple PE's. Figure 1 shows one of the basic PE's which will be used to construct the array multiplier illustrated in figure 2.

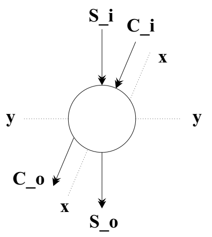

Fig. 1: Processing Element used in figure.2.

Each PE performs the following logical operations:

$$So = Si \oplus Ci \oplus x \cdot y$$

$$Co = (Si \cdot Ci) + (Si \cdot (x \cdot y)) + (Ci \cdot (x \cdot y))$$

where x and y represent A_i or M_i and b_i or q_i respectively.

All PE's are identical, and a total of $2n(n-1)$ are required to perform 1 multiplication as illustrated in the DDG of figure 2. The LSB of the Result, S_0 appears at the LSB of row n, and successive bits of the result ($S_1 - S_{n-1}$) appear on subsequent LSB's of rows $(n+1)-(2n-1)$. Thus the MSB of the result S_{n-1} appears at the LSB of row $2n-1$. The values of $q_n,...,q_{2n-2}$ and $b_n,...,b_{2n-2}$ must be set equal to 0. Walter presents a similar 2D array in [5], however the carry propagation is in the right-to-left horizontal direction, resulting in a maximum carry chain length of n-bits.

By examining the DDG of figure 2, it is observed that the PE's can be pipelined in a systolic fashion, according to the order indicated. By mapping calculations occuring at different times to a single PE, a bit serial

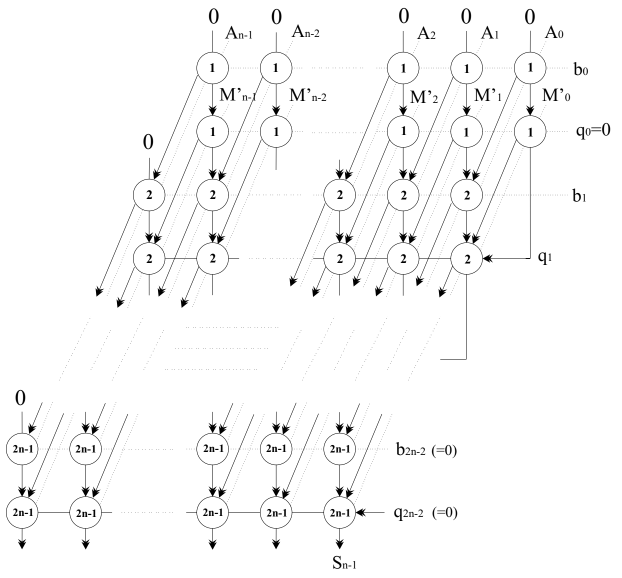

Fig. 2: Data Dependence Graph (DDG) for Montgomery multiplication using algorithm (1).

systolic array is derived. By introducing a clock into the system, only $2n$ PE's are required as illustrated in figure 3.

In this architecture, the multiplier b_i is loaded bit serially from the LSB of the array, and is fed in a systolic manner through the array.

After n clock cycles, the LSB of the result, S_0, is valid at the LSB and after a further $(n-1)$ clock cycles, the MSB of the result, S_{n-1}, is valid.

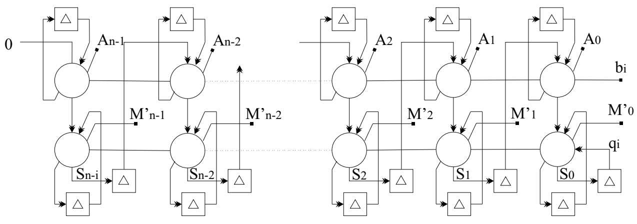

Fig. 3: Bit serial systolic array for Montgomery multiplication using algorithm (1).

This is the design developed for FPGA implementation by Marnane in [8]. By building the multiplier from individual processing elements, each operating on 1 bit of the calculation, a ratio of 2 bit slices to 5 CLB's (*Configurable Logic Block's*) was achieved, making it possible to implement a 360-bit multiplier on a Xilinx XC4025 FPGA. By exploiting the reconfigurability of the FPGA, the design could be optimised further to achieve an array length of 450-bits provided the value of M is pre-programmed, using 2 CLB's per bit slice. A clock speed of 50MHz was achieved, and a single multiplication required $2n-1$ clock cycles to execute. The issue of pre- and post-calculations to convert to and from M-residues was not addressed.

A similar systolic multiplier design is implemented by Kornerup in [4]. Both algorithms (1) & (2) as introduced in Section III. are presented, with algorithm (1) being favoured for the systolic implementation.

Instead of implementing the multipliers as an array of systolic processing elements, they may be implemented as two Carry Propagation Adders (CPA).

Recently, the authors of [9] proposed a design similar to the the ***MonArch2*** design presented in section A. of this paper. The authors chose to use the R-L exponentiation algorithm, thus increasing clock frequency by implementing two modular multipliers. The design was is targeted for VLSI implementation, based on 0.5μm SOG technology, and uses a Double Carry Save Adder (CSA) structure with an additional CPA to implement the multipliers. The design produces an M-residue result from 2 M-residue inputs. The issue of mapping to and from M-residues is dealt with as in section III. of this paper.

In their example, the two CSA's perform the basic operation in $n = 1026$ clock cycles, and another 32 clock cycles to perform the addition in the CPA. A bit rate of approximately 46.5kbit/s is achieved for the 1024 bit exponentiator.

In [9][10] the authors suggest that the size of r be 2 bits larger that that of the modulus to limit the range of the intermediate output and that the *for* loop of the MonPro algorithm be executed k+3 times to ensure that the result of one modular multiplication may be reused as an input for the next multiplication, and thus make it possible to pipeline the algorithm.

Walter and Eldridge present a good review of methods of improving multiplier efficiency in [6], and Walter provides further insight into the Montgomery algorithm in [5]&[7]. However, none of their discussions deal with implementation of the algorithm on FPGA devices, and so not all of their efficiency improvements can be implemented in the architectures presented here.

The Baud rates being achieved by other RSA designs are around 34kbit/s by Kim, Kang & Choi in 2000[11] for 1024 bits, 118kbit/s by Yang, Chang & Jen in 1998 [12], and 278kbit/s by Guo, Wang & Hu in 1999 [13] for 512 bit designs. These are VLSI implementations and obviously cannot be directly compared to the FPGA results provided in this paper, but this gives an idea of the data throughput rates expected.

VI. UNDERLYING FPGA ARCHITECTURE

Unlike ASIC design, there is an uderlying architecture imposed upon FPGA circuit design. Thus an optimum design for FPGA will exploit the features available.

Typically, FPL devices today are comprised of configurable units which consist of 4-input lookup tables, simple D-type Latches, and control logic. The lookup tables can be used to implement 4-input function generators or 16x1-bit RAM. This type of RAM is given the term *Distributed RAM*. Block RAM may also be provided on another area of the chip dedicated for this purpose.

Some devices contain high speed interconnect lines between vertically adjacent logic blocks which are designed to provide efficient carry propagation. This dedicated logic provides carry capability for high speed arithmetic functions. In order to exploit the fast carry chain, the logic blocks in which the carry signals are generated and propagated must be placed in a single column on the FPGA. This dedicated carry logic improves both the speed and area of arithmetic operations.

The architectures presented in this paper are optimal for implementation on any FPL device which has dedicated carry logic capability. Speed and area results may differ between technologies due to routing and different physical layout, however by exploiting the maximum carry chain length, the optimum design can be tailored to a given target device.

The carry save adder design which can be used to increase large digit addition speed on VLSI designs does not improve efficiency on FPGA designs, since it does not fully exploit the fast dedicated carry logic provided.

VII. Multiplier Architectures

A. Double Adder Architectures

If the direction of carry propagation in figure 2 is modified to flow from right to left, two carry propagation adders are obtained which can be mapped directly to two n-bit full-adders. This is presented as MonArch1 in figure 4. The fast carry chain of the Configurable Logic Blocks (CLB's) in the FPGA can be exploited to reduce the clock period. Also, by using the dedicated AND gate provided in each CLB, 2 adder and b_iA or q_iM functions can be performed in one CLB.

The first layout presented is based on algorithm (1) which requires an additional precomputation to convert M to M' before multiplication begins. This architecture requires $n+1$ clock cycles to complete a calculation.

By basing the architecture on algorithm (2), the number of clock cycles can be reduced to n as illustrated in figure 5. The value of q_i is now taken directly from the LSB of the sum of S_{i-1} and b_iA. The second adder must also be increased to $n+1$ bits to account for the division by 2 after addition rather than before (In fact, only the n most significant bits are taken from the $n+1$-bit result, the LSB being discarded).

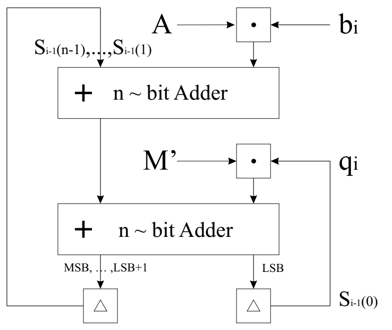

Fig. 4: *MonArch1*: Montgomery multiplier architecture based on algorithm (1).

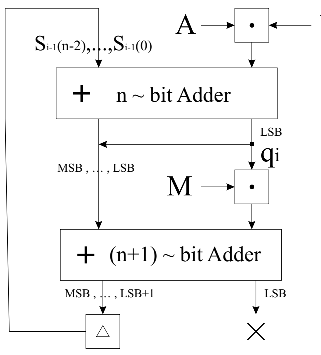

Fig. 5: *MonArch2*: Montgomery multiplier architecture based on algorithm (2).

To remove the dependency of q_i on the addition of b_iA and S_{i-1}, A can be shifted up 1 bit, thus forcing the LSB of S_{i-1} + b_iA to always be zero, as illustrated in figure 6. However, this extra factor of 2 must be removed by an extra iteration, meaning that the number of clock cycles is again equal to $n+1$.

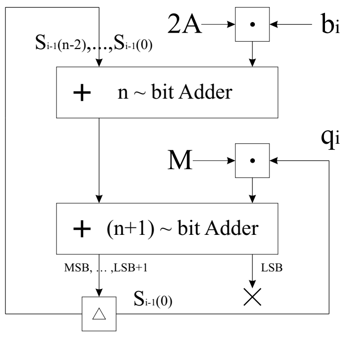

Fig. 6: *MonArch3*: Montgomery multiplier architecture based on algorithm (3).

If the order of the additions is rearranged, the architecture of figure 7 can be used. Again A must be shifted up by 1 bit and $n+1$ clock cycles are required. However, there is a significant increase in the operation speed of this architecture over the preceding ones due to the fact that the $b_i2A + q_iM$ calculation can be performed as soon as the 2 LSB's of the other addition are complete.

Thus, both adders can work almost in parallel. This design does however require slightly more area due to the need for 2 n-bit registers at the input to the second adder.

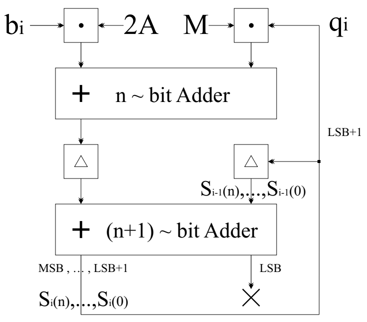

Fig. 7: *MonArch4*: Modified Montgomery architecture based on algorithm (3).

B. *Mux/Add Architecture*

By examining figure 7, it can be seen that, for any iteration of a given multiplication, there are only 4 possible outputs from the first adder. These are; 0, M, $2A$, or $M+2A$, depending on the values of b_i and q_i. Therefore the implementation of 2 adders is wasteful. Instead, a multiplexer could be used to select from the 4 possible inputs to the adder as shown in figure 8.

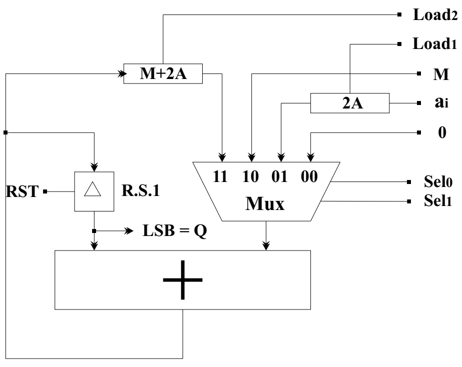

Fig. 8: *MuxMult1*: Montgomery modular multiplier architecture with single adder and multiplexer.

This design requires that the value of $2A+M$ be computed and stored in a register before the actual multiplication begins. This can be achieved with 2 extra clock cycles; the first to load M into the adder, and the second to add this value to $2A$. The result is then stored in the $2A+M$ register, and the inputs to the adder reset to 0. Two select lines control the output of the multiplexer. During the actual multiplication, one would equal q_i, and the other b_i, however additional control is required to ensure the precalculation is performed correctly.

C. *Distributed RAM Architecture*

An extension of the idea proposed in figure 8, is to replace the multiplexer with a RAM. The LUT in each CLB can be implemented as a 16×1 Distributed RAM block.

Therefore it is proposed to use a 4xn RAM to store these values, and by using b_i and q_i as address lines, any of the 4 possible outputs can be selected (figure 9). The value of A can be written into the RAM at the start of each calculation, and since M is relatively constant (for a given recipient in the case of RSA), it can be initialized in the RAM at configuration. Also, the value of $M+2A$ must be calculated at the beginning of each multiplication and then written to the RAM.

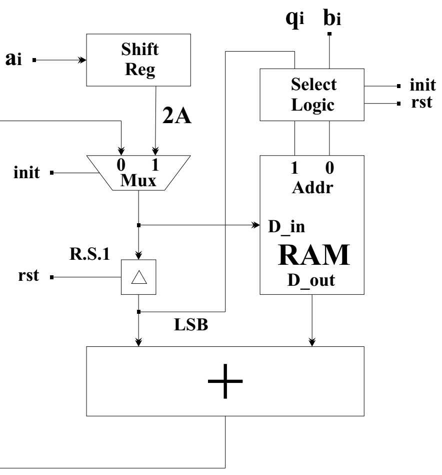

Fig. 9: *RAMMult* : Distributed RAM design with single adder.

Again relatively few extra initialization steps are required to load the RAM with the values of $2A$, M, and then $M+2A$. A shift register is still required to load A serially and load it into the RAM. The dedicated Block RAM can be used as a shift register to load, store, and input B to the multiplier.

Speed and area results for 64-bit implementations of the unpipelined multiplier of figures. 4-10 are given in Table 1. These were targeted to a Xilinx Virtex V600FG680-6 FPGA. Included in the designs were 64-bit shift registers to load in the values of A & M, and to output the result S in a bitwise manner. The data throughput figures are calculated for a single multiplication, since the result would need to be output, and the next set of inputs loaded into the registers before the next multiplication can commence. This could be overcome by including an extra n-bit register for A, B, & M, so that the values may be loaded while a multiplication is being performed.

It can be seen from Table 1 that architectures MonArch1-3 have very similar area and speed char-

Architecture	No. of Slices	Clock Freq.
MonArch1	162	104.46 MHz
MonArch2	161	102.40 MHz
MonArch3	161	105.74 MHz
MonArch4	194	119.47 MHz
RAMMult4	194	74.12 MHz
MuxMult1	161	114.53 MHz
MuxMult2	162	113.87 MHz

Table 1: Speed & area results for a 64-bit Montgomery modular multiplier.

acteristics. However, architecture MonArch4 is almost 13% faster. The RAM multiplier performs much slower than the other architectures, due to the access time of the RAM. The Mux/Add architecture doesn't perform quite as well as the MonPro4 architecture, but it is better suited to the requirements of the multiplier needed to perform modular exponentiation.

The number of CLB's per bit of the multiplier is approximately 1.25 for designs MonArch1-3 and MuxMult, and 1.5 for MonPro4 and the RAMMult designs. This is significantly lower than that achieved by previous implementations such as [8].

Overall, it was decided that the MuxMult architecture was best suited for use in the exponentiator design due to its high speed and lower area requirements. Therefore, the remainder of the paper will concentrate on this architecture.

VIII. Pipelined Mux/Add Architecture

When the bitlength of the multiplier exceeds the maximum carry chain length (or 1 column of the FPGA), the advantage of exploiting the carry logic is lost. By dividing the calculation into j p-bit words, and pipelining it, an improvement in clock speed is achieved (where p is the maximum length of the carry chain for the device in question, and $j = n/p$). Therefore p is is a function of the target device resulting from the underlying physical architecture of the device.

The inputs to the multiplier must be broken into p-bit words, and the carry out of the adder must be delayed by one clock cycle before being input to the next adder. However, due to the right-shift operation, the LSB of the sum must be fed *directly* into the MSB of the *previous* adder as illustrated in figure 10. This does not slow the operation of the circuit since the LSB of the higher adder will be computed long before the MSB of the lower adder needs to be computed.

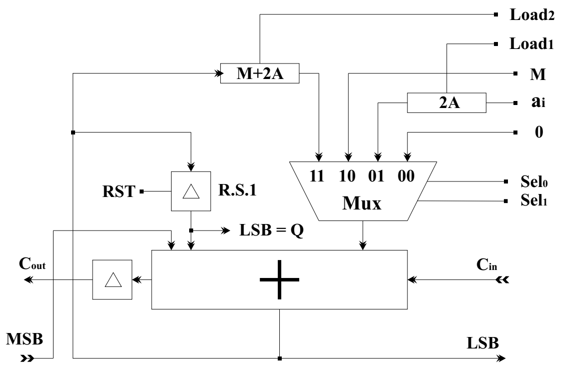

Fig. 10: *MuxMult2* : pipeline modular multiplier

Figure 11 illustrates the flow of data through a fully pipe-lined multiplier with $j = 4$. It can be seen that there is a delay of $n+3$ clock cycles between the loading of the first p-bit input word, A_0 and the first p-bit output word, S_0 being valid in the result register of the same pipelined unit. It takes $n+3$ clock cycles for each multiplication due to the 2 clock cycle initialisation (as discussed in section B.), and the $n+1$ clock cycle implementation of algorithm (3). For the next $j-1$ clock cycles, output words S_1-S_{j-1} are valid and are right shifted as they are bing loaded so as to output the result serially over k clock cycles.

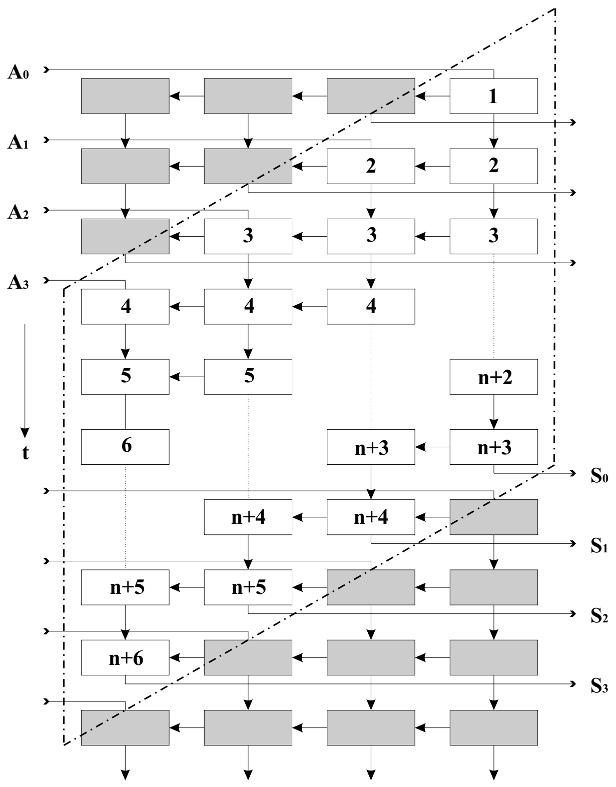

Fig. 11: *Pipelined Multiplication* : DDG using the pipelined multiplier units.

It is obvious that another multiplication can begin as soon as the first output word is valid, and has been latched into the result register. Refer to figure 11 where each of the blocks represents a single pipelined multiplier unit as in figure 10. Although it takes $(n+3)+(j-1)$ clock cycles from the input of A_0 to the latching of S_{j-1} into the result register, a new result is valid every $(n+3)$ clock cycles thereafter. Therefore for throughput calculation purposes, it takes (n+3) clock cycles to produce the consecutive multiplication results, with an extra latency of $(j-1)$ clock cycles to produce the first result. This latency will not affect the throughput of the exponentiator.

IX. Exponentiator Architecture

The basic architecture presented in figure 10 requires some modifications so as it may be used in the exponentiator where the multiplicands change from one multiplication to the next. The multiplexer now has 6 inputs instead of 4, and therefore requires 3 select or control lines. All control signals and the carry must be clocked through the unit from right to left. There are three possible parallel inputs to the multiplier depending on the stage of the multiplication, and so the multiplexer must be extended to allow for this. Additional registers are also necessary.

With reference to figure 10, the input A can now be any of the following: The constant, C, the initial start result, R_0, or the previous multiplication result, R. The M and C registers are used by both multipliers, so they are not within the pipelined unit. Figure 12 illustrates a p-bit multiplier unit for the implementation of an n-bit fully pipelined multiplier including all necessary registers. Minor additional control required for the initialisation stage is omitted for clarity in the figure.

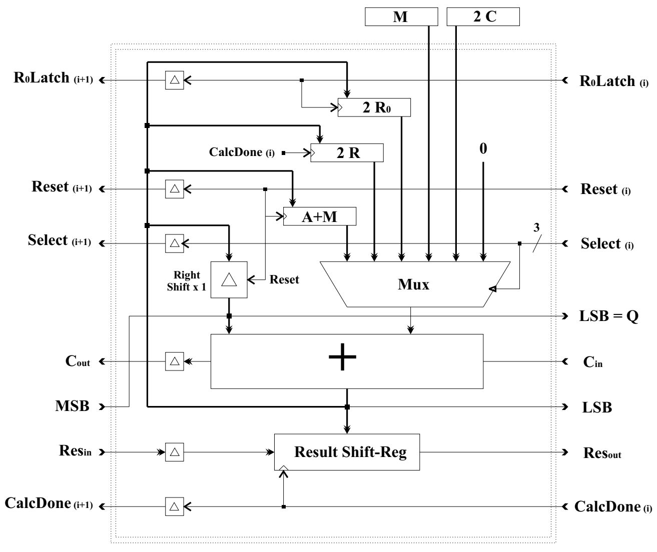

Fig. 12: *MuxMult3* : p-bit modular multiplier unit for exponentiator implementation.

A total of $2j$ of the pipelined multiplier units are needed to implement the 2 fully functional n-bit multipliers required for the exponentiator. Figure 13 illustrates how the 2 multipliers must be arranged to perform the exponentiation. In the figure the result is in parallel word form at the bottom of the multiplier block, and fed out serially from the internal shift register to the right. The input at the top is also in parallel form, and the input to the left is a serial input.

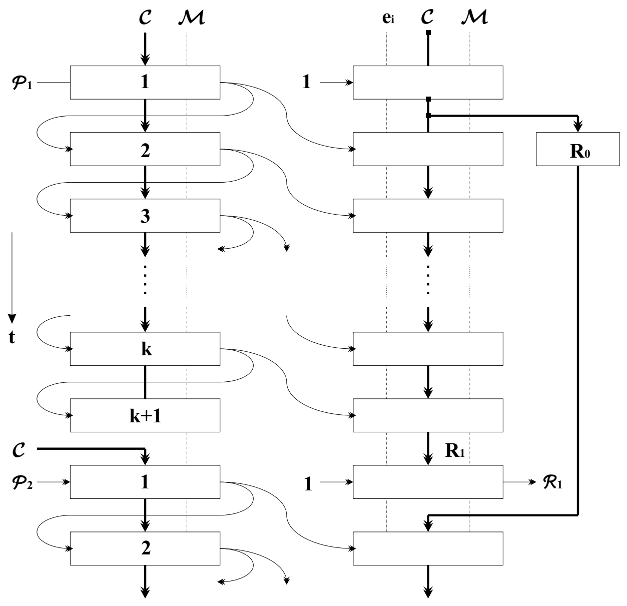

Fig. 13: *Exponentiation*: Multiplication schedule including mapping to and from M-residue representation.

The square operation is performed on every iteration, however the value of e_i determines whether or not the multiply operation is performed according to algorithm (5).

Firstly the mapping process must be performed to convert the plaintext $\mathcal{P}$ into its M-residue P, and to calculate R_0, the initial start result for all further multiplications . After that there are k parallel square & multiply operations, followed by a final re-mapping of the result back to an ordinary integer. $k+2$ square/multiplication operations are required before the first k-bit word of ciphertext is valid, but thereafter, another word is valid every $k+1$ square/multiplication operations.

The value of $1 \times C$ mod M need not be calculated each time since it is constant for a given modulus, and it can therefore be stored in a register to be used once every $k+1$ multiplications.

The number of clock cycles required to perform an exponentiation is therefore $(k+1)$ times the number of clock cycles required per multiplication, or $(k+1)(n+3)$ on average. So for a 1024-bit exponentiation, more than 1.05 million clock cycles are required.

Multiplier Size	No. of Slices (% of Chip)	Clk Freq (MHz)	Data Rate
120	603 (4%)	88.47	84.9 Mb/s
240	1211 (9%)	57.99	56.8 Mb/s
480	2426 (19%)	59.03	58.4 Mb/s
720	3641 (29%)	56.83	56.4 Mb/s
1080	5458 (44%)	54.61	54.4 Mb/s

Table 2: Speed & area results for the pipelined Mongtomery modular multiplier.

Multiplier Size	No. of Slices (% of Chip)	Clk Freq (MHz)	Data Rate
120	603 (4%)	88.47	84.9 Mb/s
240	1203 (9%)	41.41	40.6 Mb/s
480	2403 (19%)	18.52	18.3 Mb/s
720	3603 (29%)	12.64	12.6 Mb/s
1080	5403 (43%)	8.40	8.4 Mb/s

Table 3: Speed & area results for a non-pipelined Mongtomery modular multiplier.

The values of the constant, C, modulus, M, and exponents E and D can be stored in on-chip ROM, which can be altered via reconfiguration. Therefore, whenever a new public or private key is required for the encryptor/decryptor, the device can be reconfigured to store the values into memory.

X. Results

Speed and area comparisons for pipelined multipliers up to 1080 bits are given in Table 2. The target device was the Xilinx Virtex V1000FG680-6, which has 124 CLB's per column. Therefore p (the bit length of the pipelined multiplier unit) was chosen to be 120, allowing for space for control registers needed for each unit. So for the 1080-bit design, $p = 120$, $j = 9$, that is, 9 of the pipelined units were required for each multiplier. The design also included all the registers necessary for using the multiplier in the exponentiator design. Therefore, the designs occupied more area than would be required for a single purpose multiplier. The significant decrease in clock frequency when the bitlength is increased from 120-bits to 240 is due to the routing of the carry from the top of one column in the FPGA to the bottom of the next. However, this penalty is not as great as for the non-pipelined design as can be seen in Table 3, and the frequency does not continue to decrease dramatically.

Approximately 2.5 CLB's per bit of the multiplier are required for this design. The increase in area over the results in Table 1 is mainly due to the extra registers and larger multiplexer required for this architecture which is designed to be used in the expontiator.

Note that, with $p = 120$, a 1080-bit multiplier will operate at the same clock frequency as a 1026-bit multiplier (remembering that $n = k+2$), and so for relatively little extra area, the bit-length of the multiplier can be increased with no deterioration of the clock frequency.

Bit Length (n)	No. of Slices (% of Chip)	Clk Freq (MHz)	Data Rate
120	1146 (9%)	83.51	673.2 kb/s
240	2301 (18%)	58.15	238.3 kb/s
480	4610 (37%)	55.92	115.5 kb/s
720	6917 (56%)	50.66	70.0 kb/s
1080 (1024)	10369 (84%)	49.63	45.8 kb/s

Table 4: Speed & area results for the pipelined RSA encryptor/decryptor.

Table 3 provides an indication as to how effective the pipelining of the multiplier is. The design used to produce the results in Table 3 was identical to one multiplier unit, except the bit length and thus carry chain length were increased to 240, 480, 720 and 1080-bits.

A dramatic decrease in clock frequency is observed as the carry chain exceeds the column height of the FPGA. Thus it is obvious that the pipelined multiplier out-performs the non-pipelined design provided that a suitable pipeline bit-length is chosen to exploit the maximum length of the carry chain.

Finally, a full modular exponentiator is implemented and synthesised, and the results are presented in Table 4. This design can implement either RSA encryption or decryption.

A 1024-bit exponentiator can be implemented on the 1080-bit design at a data rate of 48.2 kb/s. The clock frequency figures were obtained from the Xilinx auto place and route tool. By floorplanning by hand, a smaller area and high clock rate can generally be achieved.

XI. Conclusions

By exploiting the fast carry logic provided in the Xilinx Virtex FPGA's it is possible to implement a high speed Montgomery multiplier for bit lengths shorter than the maximum length of the carry chain (ie: 1 column of the FPGA). In order to improve speed at higher bit-lengths it is necessary to break the multiplication up into stages, and pipeline the calculation. This was seen to improve performance significantly.

Reconfiguration can be used to load the values of M, C, and possibly E or D into the design before encryption or decryption begins. Generally this does not have to be performed very often, with many exponentiation calculations being executed with the same values of M,C, E, and D.

Also, by having both E and D stored in registers, digital signatures can be appended to messages by simply encrypting part (or all) of the plaintext with the sender's private key, and then reloading this ciphertext into the encryptor to further encrypt it with the recipient's public key. To decrypt this message the receiver must first decrypt it with his own private key and then with the senders public key to confirm its origin.

With some further optimisation, a 1024-bit RSA encryptor/decryptor with a clock rate exceeding 50MHz should be achievable. This would lead to data rates of 50kb/s which is faster than the throughput rates being achieved by some recent VLSI designs[9].

XII. Acknowledgement

This work was funded by a strategic research grant from Enterprise Ireland.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for obtaining digital signatures and public-key cryptosystems". *Comm. ACM*, 21:120–126, 1978.

[2] P. L. Montgomery. "Modular multiplication without trial division". *Math. Computation*, 44:519–521, 1985.

[3] Yong-Jin Jeong and Wayne P. Burleson. "VLSI Array Algorithms and Architectures for RSA Modular Multiplication". *IEEE Transactions on VLSI Systems*, 5(2):211–217, June 1997.

[4] Peter Kornerup. "A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms". *IEEE Transactions on Computers*, 43(8):892–898, Aug. 1994.

[5] C. D. Walter. "Systolic modular multiplication". *IEEE Transactions on Computers*, 42:376–378, Mar 1993.

[6] S. E. Eldridge and C. D. Walter. "Hardware implementation of Montgomery's modular multiplication algorithm". *IEEE Transactions on Computers*, 42:693–699, June 1993.

[7] C. D. Walter. "Still faster modular multiplication". *Electronics Letters*, 31:263–264, Feb 1995.

[8] W. P. Marnane. "Optimised bit serial modular multiplier for implementation on field programmable gate arrays". *IEE Electronics Letters*, 34(8):738–739, April 1998.

[9] Taek-Won Kwon, Chang-Seok You, Won-Seok Heo, Yong-Kyu Kang, and Jun-Rim Choi. "Two implementation methods of a 1024-bit RSA cryptoprocessor based on modified montgomery algorithm". *IEEE Int. Symp. on Circuits and Systems (ISCAS)*, pages 650–653, May 2001.

[10] T. Blum and C. Paar. "Montgomery Modular Exponentiation on Reconfigurable Hardware". *Proc. 14th IEEE Symp. on Computer Arithmetic*, pages 70–77, 1999.

[11] Young Sae Kim, Woo Seok Kang, and Jun Rim Choi. Implementation of 1024-Bit Modular Processor for RSA Cryptosystem. *The 2nd IEEE Asia Pacific Conference on ASICs*, Aug 2000.

[12] Ching-Chao Yang, Tian-Sheuan Chang, and Chein-Wei Jen. "A new RSA cryptosystem hardware design based on Montgomery's Algorithm". *IEEE Trans. Circuits and Systems – II: Analog and Digital Signal Processing*, 45(7):908–913, July 1998.

[13] Jyh-Huei Guo, Chin-Liang Wang, and Hung-Chih Hu. Design and Implementation of an RSA Public-Key Cryptosystem. *Proc. IEEE International Symposium on Circuits and Systems*, 1:504–507, 1999.

[14] Xilinx Inc. Website. http://www.xilinx.com.

Alan Daly is currently pursuing a Ph.D. in the Department of Electrical and Electronic Engineering, University College, Cork. Primary areas of interest include Reconfigurable Logic Design, and Cryptographic applications.