Modified Montgomery modular multiplication and RSA exponentiation techniques

C. McIvor, M. McLoone and J.V. McCanny

Abstract: Modified Montgomery multiplication and associated RSA modular exponentiation algorithms and circuit architectures are presented. These modified multipliers use carry save adders (CSAs) to perform large word length additions. These have the attraction that, when repeatedly used to perform RSA modular exponentiation, the (carry save) format of the output words is compatible with that required by the multiplier inputs. This avoids the repeated interim output/input format conversion, needed when previously reported Montgomery multipliers are used for RSA modular exponentiation. Thus, the lengthy and costly conventional additions required at each stage are avoided. As a consequence, the critical path delay and, hence, the data throughput rate of the resulting Montgomery multiplier architectures are also word length independent. The approach presented is based on a reformulation of the solution to modular multiplication within the context of RSA exponentiation. Two algorithmic variants are presented, one based on a five-to-two CSA and the other on a four-to-two CSA plus multiplexer. The practical application of the approach has been demonstrated by using this to design special purpose RSA processing units with 512-bit and 1024-bit key sizes. The resulting RSA units exhibit the highest data rates reported in the literature to date, reflecting the very low and word length independent critical path delay achieved.

1 Introduction

Public-key cryptosystems are important as they provide confidentiality, authentication, data integrity and non-repudiation [1, 2]. RSA [3] is the most widely used public-key cryptosystem. An RSA operation is a modular exponentiation, which requires repeated modular multiplications. For security reasons RSA operand sizes need to be 1024 bits or greater in length meaning that high data throughput rates are difficult to achieve.

The Montgomery multiplication algorithm [4] is the most efficient modular multiplication algorithm available. It replaces trial division by the modulus with a series of additions and divisions by a power of two. Thus, it is well suited to hardware implementation and forms the basis of many of the currently reported RSA hardware architectures [5–7].

To date, several techniques have been proposed in order to avoid carry propagation during the addition stages of the computation, as this is a key factor in determining performance. One approach proposed by Elbirt and Paar [6] is to break these additions into x-bit stages, where x is an optimal bit length chosen to take advantage of the fast carry chains available on modern FPGAs. However, a drawback of this approach is that the circuits developed can be very heavily technology and implementation dependent. For example, it is unlikely that a design created in this manner for a specific FPGA family will show the same speed advantages if migrated to a modern ASIC technology or, indeed, an alternative type of FPGA or PLD. An alternative approach presented by Blum and Paar [7] is based on the use of FPGA systolic array multiplier architectures with varying processing element sizes (i.e. 4, 8 and 16 bits). However, these systems are again tailored specifically for the Xilinx XC4000 FPGA series.

Research has also been undertaken to develop Montgomery multiplication architectures, which have a small critical path delay independent of the target silicon technology. One way to achieve this is to use carry save adders (CSAs) to perform the addition stages of Montgomery's algorithm. For example, Kim *et al.* [8] used two levels of carry save logic (CSL) and a 32-bit carry propagate adder along with a 32×32-bit shift register in order to perform the 1024-bit additions required. Bunimov *et al.* [9] improved this by replacing one level of CSL with a look-up table. This reduces the addition stages to one level of CSL with a k-bit conventional adder to perform a full Montgomery multiplication, where k is the operand bit length (usually $>$ 1024 bits). An important limitation of both these approaches is that when used for RSA cryptography there is a need for a significant level of repeated modular multiplication in order to perform modular exponentiation. Thus, there is a need at each modular multiplication to perform an output-to-input format conversion i.e. to convert the results from a carry save format to a conventional format. This adds to the computation time required and means that throughput rates are word length dependent.

We now introduce two, modified Montgomery multiplication algorithms suitable for RSA exponentiation, which avoid this problem. These are based on using a five-to-two CSA (three levels of CSL) and a four-to-two CSA (two levels of CSL), respectively. Each can perform a Montgomery multiplication in only $k + 1$ and $k + 2$ clock cycles, respectively, where k is the operand bit length. A key aspect

IEE Proceedings online no. 20040791

doi: 10.1049/ip-cdt:20040791

Paper first received 24th February and in revised form 20th May 2004

The authors are with The Institute of Electronics, Communication and Information Technology (ECIT), Queen's University Belfast, Northern Ireland Science Park, Queen's Road, Queen's Island, Belfast, BT3 9DT, UK

of these systems is that all inputs and outputs to the Montgomery multipliers are maintained in a redundant carry save format. This avoids the need for the lengthy conventional addition normally needed to complete such a multiplication in RSA exponentiation applications. As will be discussed, we believe this more than compensates for the fact that two and one extra levels of CSL respectively are required when compared with that described in [9]. The approach presented is based on reformulating and solving the problem of modular multiplication within the context of RSA exponentiation. As a consequence, the costly and previously necessary conversion addition at the end of each multiplication stage is avoided and the critical path delay and, hence, the data throughput rate of the architectures presented are word length independent.

As will be discussed in Section 3, the practical application of our approach has been demonstrated by using this in the design of special purpose RSA processing units, in this case with 512-bit and 1024-bit key sizes. These are based on the five-to-two modular multiplier and a modified modular exponentiation algorithm. The resulting RSA units have a very low critical path delay, again independent of the word length. These perform encryption in $(k+2)(k_e+3)$ clock cycles, where k is the modulus bit length and k_e is the public exponent bit length. Here the Fermat prime, $F_4 = 2^{16}+1$, is used as the public exponent and RSA decryption is performed in $(k/2+2)(k_d/2+3)$ clock cycles, with k_d being the private exponent bit length. The well known Chinese remainder theorem (CRT) technique [10] is also used to speed-up the RSA decryption function.

2 Montgomery multiplication algorithms

2.1 Montgomery multiplication background

Given an integer $a < n$, where n is the k-bit modulus, A is said to be its n-residue with respect to r if:

$$A = a \times r \,(\mathrm{mod}\, n) \text{ where } r = 2^k \tag{1}$$

Likewise, given an integer $b < n$, B is said to be its n-residue with respect to r if:

$$B = b \times r \,(\mathrm{mod}\, n) \tag{2}$$

The Montgomery product of A and B can then be defined as:

$$Z = A \times B \times r^{-1} (\mathrm{mod}\, n) \tag{3}$$

where r^{-1} is the inverse of r, modulo n.

The modulus n must also be an odd number, a condition always satisfied in an RSA cryptosystem. Because a conversion to and from n-residue format is required when using Montgomery multiplication, its use only really becomes attractive in applications requiring many repeated modular multiplications, such as RSA exponentiation. The radix-2 version of Montgomery's multiplication algorithm [11], which calculates the Montgomery product of A and B, is summarised in the pseudo-code below.

Algorithm 1: Montgomery multiplication (A, B, n)

$S[0] = 0;$

for i in 0 **to** $k-1$ **loop**
  $q_i = (S[i]_0 + A_i^* B_0)$ **mod** 2;
  $S[i+1] = (S[i] + A_i * B + q_i * n)$ **div** 2;

end loop;

return S[k];

The critical delay of algorithm 1 occurs during the calculation of the S values given by the three input addition:

$$S[i+1] = (S[i] + A_i \times B + q_i \times n) \tag{4}$$

The main contributing factor to this delay is the carry propagation resulting from the very large operand additions. However, this can be avoided by using CSAs to calculate (4) [8, 9]. This observation led us to rewrite algorithm 1 in alternative formats, defined as algorithm 2 and algorithm 3 in Section 2.2, which we believe provide the basis of highly efficient modular multiplier implementations suitable for performing RSA modular exponentiation. These have been explicitly derived so that they can employ different CSA variants when implemented architecturally. The overall motivation is to produce silicon circuits with small critical path delays but also with competitive gate counts allowing for single-chip implementation on media such as microprocessor smart cards, for example.

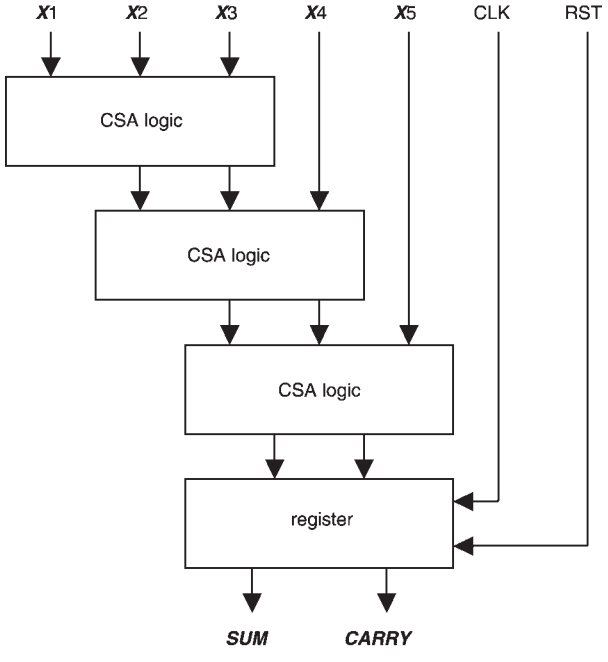

Fig. 1 *Block diagram of five-to-two CSA*

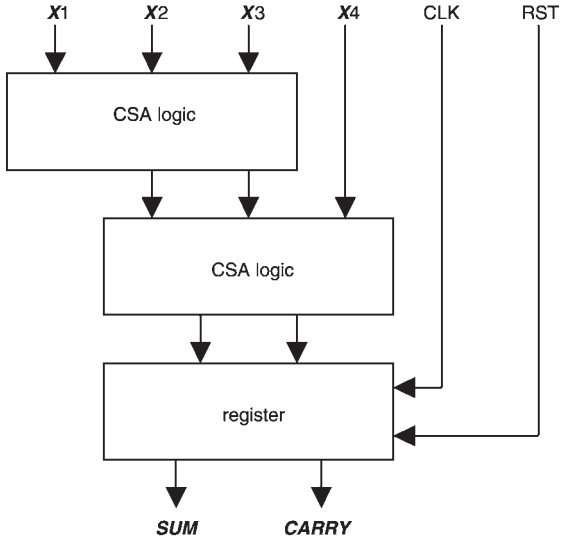

Fig. 2 *Block diagram of four-to-two CSA*

2.2 Modified Montgomery multiplication algorithms for RSA exponentiation

The first of these variants is based on a five-to-two CSA, which is used to calculate (4). This type of adder comprises three levels of CSL, where the sum of the bit vectors ***SUM*** and ***CARRY*** is equal to the sum of the five input bit vectors ***X***1, ***X***2, ***X***3, ***X***4 and ***X***5, as shown in Fig. 1. The carry save representation of the five input operands is output from the register after only one clock cycle using this approach.

Algorithm 1 can then be rewritten as algorithm 2, given below. Note that the input operands A and B and the output product S are now in a carry save representation (CSR) denoted by $A1$ and $A2$, $B1$ and $B2$, and $S1$ and $S2$ respectively.

Algorithm 2: Five-to-two CSA Montgomery multiplication ($A1$, $A2$, $B1$, $B2$, n)

$S1[0] = 0;$

$S2[0] = 0;$

for i in 0 **to** k − 1 **loop**
  $q_i = (S1[i]_0 + S2[i]_0) + (A_i * (B1_0 + B2_0))$ **mod** 2;
  $S1[i+1], S2[i+1] = CSR(S1[i] + S2[i] + A_i * (B1 + B2) + q_i * n)$ **div** 2;

end loop;

return S1[k], S2[k];

The critical delay of algorithm 2 occurs during the calculation of the five-to-two carry save addition:

$$S1[i+1], S2[i+1] = CSR(S1[i] + S2[i] + A_i \times (B1 + B2) + q_i \times n) \quad (5)$$

The second approach is based on the use of a four-to-two rather than a five-to-two CSA resulting in a saving of a full level of CSL, as shown in Fig. 2.

Again, the CSR of the four input operands is output from the register after only one clock cycle. Therefore, by using the four-to-two CSA to calculate (4), algorithm 1 can be reformulated as algorithm 3.

Algorithm 3: Four-to-two CSA Montgomery multiplication ($A1$, $A2$, $B1$, $B2$, n)

$D1, D2 = CSR(B1 + B2 + n + 0);$

$S1[0] = 0;$

$S2[0] = 0;$

for i in 0 **to** k − 1 **loop**
  $q_i = (S1[i]_0 + S2[i]_0) + (A_i * (B1_0 + B2_0))$ **mod** 2;
  if $A_i = 0$ and $q_i = 0$ **then**
    $S1[i+1], S2[i+1] = CSR(S1[i] + S2[i] + 0 + 0)$ **div** 2;
  elsif $A_i = 1$ and $q_i = 0$ **then**
    $S1[i+1], S2[i+1] = CSR(S1[i] + S2[i] + B1 + B2)$ **div** 2;
  elsif $A_i = 0$ and $q_i = 1$ **then**
    $S1[i+1], S2[i+1] = CSR(S1[i] + S2[i] + n + 0)$ **div** 2;
  else
    $S1[i+1], S2[i+1] = CSR(S1[i] + S2[i] + D1 + D2)$ **div** 2;
  end if;

end loop;

return S1[k], S2[k];

The main computation to be performed here is the four-to-two carry save addition:

$$S1[i+1], S2[i+1] = CSR(S1[i] + S2[i] + y + z) \quad (6)$$

where the pair y and z can represent zero and zero, $B1$ and $B2$, n and zero or $D1$ and $D2$ respectively, depending on the values of A_i and q_i. Indeed, the determination of the state of both these signals simultaneously, represents extra control logic in the form of an additional multiplexer, which needs to be taken into account in determining the critical delay. This multiplexer can be implemented as two 4:1 multiplexers working in parallel, addressed by the two signals A_i and q_i. Two extra registers, $D1$ and $D2$, are also required, as shown in algorithm 3. These are used to store the CSR of the input operands $B1$, $B2$ and n, which are added at the beginning of each new multiplication.

These modified algorithms show a significant improvement in the computation times required to perform each of the many repeated modular multiplications needed in an RSA exponentiation. Table 1 provides a comparison of this with previous approaches based on CSAs [8, 9]. The 2 XOR gates and 1 AND gate in the loop delay of algorithms 2 and 3 occur as a consequence of the q_i computations.

In this, it has been assumed that the algorithms in [8] and [9] use Walter's technique [11] i.e. no final subtractions are required when used for modular exponentiation. The major advantage that our approach offers over those previously described [8, 9] is that when used for RSA exponentiation, no carry save to conventional conversion delay is encountered at each modular multiplication when used in RSA exponentiation. Algorithms 2 and 3 achieve this by maintaining inputs and outputs in a consistent, redundant carry save format throughout the exponentiation process. Of course, one final conversion addition from carry save to a conventional format needs to be performed, but this is only calculated at the end of a full RSA exponentiation i.e. after a large number of repeated multiplications. Moreover, the overall critical delay of the system is not increased because of this, as will be discussed in Section 3.2. The cost of this is the additional two/one levels of CSL respectively, plus the multiplexer in the case of algorithm 3.

Because they are based solely on CSAs, algorithms 2 and 3 allow Montgomery multiplier architectures to be developed for RSA exponentiation that are highly parallel and

Table 1: Timing delays required for a single modular multiplication when used for RSA exponentiation

Algorithm	Loop delay	Conversion delay	Clock cycles	Critical delay
Kim *et al.* [8]	2 CSAs	$k/32$ iterations of 32-bit CPA	$k + 34$ (for $k = 1024$)	32 full adders
Bunimov *et al.* [9]	1 CSA + 4 : 1 multiplexer	k-bit conventional adder	not documented	carry propagation of k conventional adders
Algorithm 2	3 CSAs + 2 XORs + 1 AND	none	$k + 1$	3 full adders + 2 XORs + 1 AND
Algorithm 3	2 CSAs + 4 : 1 multiplexer + 2 XORs + 1 AND	none	$k + 2$	2 full adders + 4 : 1 multiplexer + 2 XORs + 1 AND

have a very low critical path delay. Moreover, this critical path delay and, hence, the data throughput rate of such multipliers is word length independent. This contrasts with [8] and [9] where RSA performance is determined by a lengthy conventional addition (Table 1). Kim *et al.* [8] tried to counteract this by using a 32-bit CPA in their multiplier. However, not only does this result in circuits with a longer critical delay than those presented here, but they also require additional clock cycles to complete the computation by virtue of the fact that they only use a 32-bit CPA.

In our approach because the inputs and outputs to the modified algorithms remain in redundant carry save format, a full addition of the $A1$ and $A2$ input operands needs to be performed in order to determine the correct A_i values in algorithms 2 and 3. This is achieved by using the barrel register full adder (BRFA) described in Section 2.3.

2.3 The BRFA component

The determination of the correct A_i values is calculated 'on-the-fly' using the BRFA shown in Fig. 3.

The correct A_i value is ready when needed by adding the least significant bit of the $A1$ and $A2$ input operands using a single full adder. Once the current A_i value has been used, the $A1$ and $A2$ operands are barrel shifted one place to the right so that the next A_i value can be determined. The BRFA does not add to the critical delay of the modified algorithms, as only a 1-bit full addition is required per clock cycle and this is computed in parallel with the CSAs. Thus, no extra cycles are required to complete the addition of $A1$ and $A2$. This means that the operations in (5) and (6) can both be computed in a single clock cycle.

2.4 Modified Montgomery multiplier implementations for RSA exponentiation

In order to verify the functionality of the modified architectures described in Sections 2.2 and 2.3, silicon designs based on these were captured generically in VHDL and implemented using the Xilinx Virtex2 series of FPGAs using varying bit lengths. Tables 2 and 3 provide performance results for the various Montgomery multiplier implementations considered.

It should be noted that in the worst case, an increase in the operand bit length by a factor of four from 512 to 2048 bits results in an increase of only a factor of 1.4 in the critical path delay through the multipliers. This relatively small increase is simply a consequence of additional routing delay. On the other hand, the area of the multipliers approximately doubles as the bit length doubles. Thus, the architectures are highly scalable and regular, which is a useful characteristic as it ensures that the multiplier designs will be of use in future RSA cryptosystems when operand sizes will need to be greater than 2048 bits in length.

Table 2: Performance results for five-to-two CSA multiplier

Xilinx device	Bit length	Clock speed, MHz	Area (number of slices)	Throughput rate, Mb/s
XC2V1500	512	126.71	5 170	126.46
XC2V3000	1024	101.71	10 332	101.61
XC2V6000	2048	90.09	20 986	90.05

Table 3: Performance results for four-to-two CSA multiplier

Xilinx device	Bit length	Clock speed, MHz	Area (number of slices)	Throughput rate, Mb/s
XC2V1500	512	122.03	5 782	121.55
XC2V3000	1024	111.32	11 520	111.10
XC2V6000	2048	90.73	23 108	90.64

The results given in Tables 2 and 3 can be directly compared with each other. Figure 4 provides a graphical representation of this comparison.

For all operand bit lengths shown, the four-to-two multipliers use a larger silicon area than the equivalent five-to-two circuits. For 1024-bit and 2048-bit operand lengths there is an increase in the data throughput rate of 9.3 and 0.7%, respectively, using the four-to-two multipliers rather than five-to-two circuits. However, the increase in silicon area usage in these cases is 11.5 and 10.1% respectively. In this case there is therefore no great advantage, in using the four-to-two multipliers over a five-to-two circuit as the small increase in data throughput rate is outweighed by a larger increase in silicon area usage (Fig. 4).

The practical application of this research has been demonstrated by implementing special purpose RSA processing units with 512-bit and 1024-bit key sizes. As a result of the findings highlighted in this Section, the focus is on algorithm 2 with a modified modular exponentiation algorithm being used, as discussed in Section 3. The same principles can be employed using algorithm 3.

3 RSA processing units

3.1 RSA and modular exponentiation

The RSA encryption and decryption functions are given by $C = M^e \pmod n$ and $M = C^d \pmod n$ respectively, where

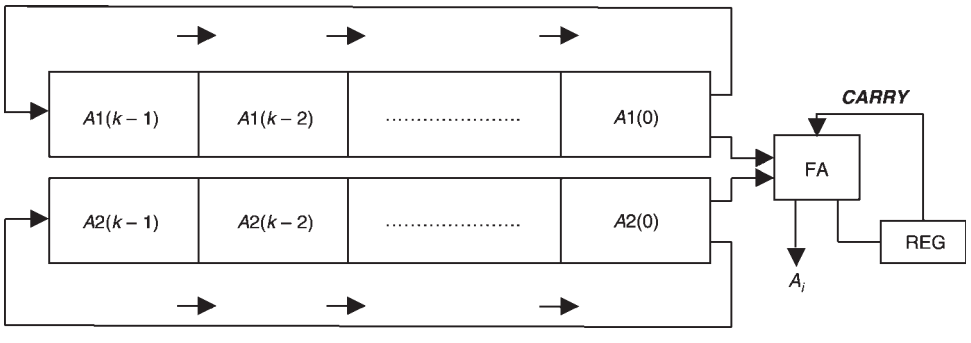

Fig. 3 *The BRFA*

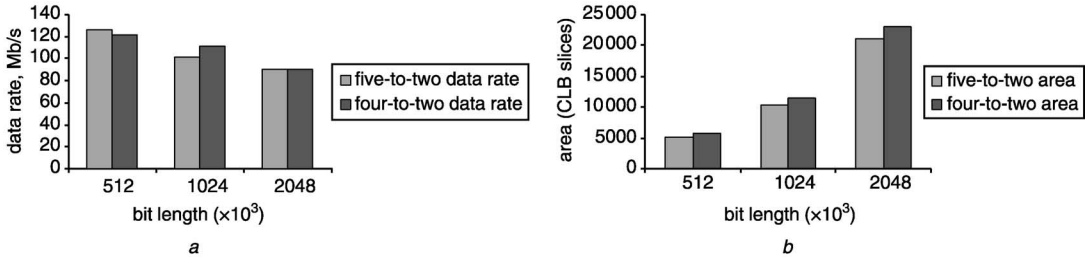

Fig. 4 *Implementation comparisons*

a Data rate comparisons
b Area comparisons

M is a plaintext message block, C is a ciphertext block, n is the k-bit modulus, and e and d are the public and private exponents respectively. The equation $ed = 1(\text{mod}(p-1) \times (q-1))$ must also hold, where p and q are two large prime numbers ($\sim k/2$-bits in length) and $n = pq$.

Thus, an RSA operation is a modular exponentiation with operands satisfying the conditions stated above. A well known and widely used modular exponentiation algorithm is the square and multiply algorithm given as algorithm 4 below. This computes $M = C^d(\text{mod } n)$, where k_d is the bit length of the private exponent d.

Algorithm 4: Modular exponentiation (C, d, n)

$\text{K} = 2^{2\text{k}}(\text{mod n});$ (computed externally)

$\text{P}[0] = \text{MontMult}(\text{K}, \text{C}, \text{n});$

$\text{R}[0] = \text{MontMult}(\text{K}, 1, \text{n});$

for i in 0 to $\text{k}_\text{d} - 1$ **loop**
  $\text{P}[\text{i}+1] = \text{MontMult}(\text{P}[\text{i}], \text{P}[\text{i}], \text{n});$
  if $\text{d}[\text{i}] = 1$ **then**
    $\text{R}[\text{i}+1] = \text{MontMult}(\text{R}[\text{i}], \text{P}[\text{i}], \text{n});$
  end if;

end loop;

$\text{M} = \text{MontMult}(1, \text{R}[\text{k}], \text{n});$

return M;

3.2 Modified modular exponentiation algorithm and RSA processing units

By using algorithm 2 to calculate the MontMult (Montgomery multiplication) stages of algorithm 4, we can rewrite this as algorithm 5. This allows special purpose RSA processing units to be created that have impressive data throughput rates. These can be implemented using FPGA, PLD or ASIC technology.

Algorithm 5: Five-to-two multiplier modular exponentiation (C, d, n)

$\text{K} = 2^{2\text{k}}(\text{mod n});$ (computed externally)

$\text{P1}[0], \text{P2}[0] = \text{5to2\_MontMult}(\text{K}, 0, \text{C}, 0, \text{n});$

$\text{R1}[0], \text{R2}[0] = \text{5to2\_MontMult}(\text{K}, 0, 1, 0, \text{n});$

for i in 0 to $\text{k}_\text{d} - 1$ **loop**
  $\text{P1}[\text{i}+1], \text{P2}[\text{i}+1] = \text{5to2\_MontMult}(\text{P1}[\text{i}], \text{P2}[\text{i}], \text{P1}[\text{i}], \text{P2}[\text{i}], \text{n});$
  if $\text{d}[\text{i}] = 1$ **then**
    $\text{R1}[\text{i}+1], \text{R2}[\text{i}+1] = \text{5to2\_MontMult}(\text{R1}[i], \text{R2}[i], \text{P1}[i], \text{P2}[i], \text{n});$
  end if;

end loop;

$\text{M1}, \text{M2} = \text{5to2\_MontMult}(1, 0, \text{R1}[\text{k}], \text{R2}[\text{k}], \text{n});$

$\text{M} = \text{M1} + \text{M2};$

return M;

An analysis of the total number of clock cycles required to perform a full RSA encryption and decryption operation is given in Table 4. Again, k is the bit length of the modulus and k_e and k_d are the bit lengths of the public and private exponents respectively. An extra clock cycle is now required to complete one of the successive modular multiplications required for RSA exponentiation, as shown in Table 4. This is to allow the results of the intermediate multiplications and squarings in the 'for' loop in algorithm 5 to be registered and reused as inputs to the next iteration of that loop. The pre- and post-processing operations indicated are required to convert the operands in algorithm 5 to and from n-residue format respectively, as required by Montgomery multiplication, described in Section 2. The final addition operation then makes use of the BRFA, as shown in Fig. 3, to add the components of the carry save result, $M1$ and $M2$ (i.e. to convert from carry save format to

Table 4: Clock cycles to compute one RSA exponentiation

Type of operation	Corresponding algorithm 5 calculations	Number of clock cycles for encryption	Number of clock cycles for decryption
Pre-processing	$P1[0]$, $P2[0]$	$k+2$	$k/2+2$
	$R1[0]$, $R2[0]$		
For loop	$P1[i+1]$, $P2[i+1]$	$k_e(k+2)$	$k_d/2(k/2+2)$
	$R1[i+1]$, $R2[i+1]$		
Post-processing	$M1$, $M2$	$k+2$	$k/2+2$
Final addition	M	$k+2$	$k/2+2$
		total: $(k+2)(k_e+3)$	total: $(k/2+2)(k_d/2+3)$

Table 5: RSA encryption results

Xilinx device	Bit length	Clock speed, MHz	Number of CLB slices	Encryption throughput rate, Mb/s	One RSA encryption, ms
XC2V3000	512	102.31	11 304	5.10	0.10
XC2V6000	1024	95.90	23 208	4.79	0.21

Table 6: RSA decryption results

Xilinx device	Bit length	Clock speed, MHz	Number of CLB slices	Decryption throughput rate, kb/s	One RSA decryption, ms
XC2V3000	512	115.75	13 910	886.92	0.58
XC2V6000	1024	97.08	26 136	375.54	2.73

ordinary format). This is the only full addition that is not carried out 'on-the-fly' throughout the whole exponentiation process. However, this requires a negligible number of extra clock cycles when compared with the total number of cycles required to perform a full RSA exponentiation (Table 4). Also, the critical path of any RSA processing unit, based on algorithm 5, is not increased as a result of this final addition, as the BRFA has a logic depth of only one full adder (Fig. 3), which is less than, and does not add to, the critical delay through the modified multipliers (Table 1). Thus, the RSA units presented have a very low critical path delay, independent of the word length chosen.

By using the CRT technique [10], two half-size decryption operations can be computed in parallel. Hence, the number of cycles required is decreased by approximately a factor of four (Table 4), again increasing the data throughput rate of the RSA decryption function significantly.

RSA processing units based on algorithm 5 were captured generically in VHDL and implemented using the Xilinx Virtex2 series of FPGAs for 512-bit and 1024-bit key sizes. Tables 5 and 6 provide performance results for these implementations. A public exponent of $2^{16} + 1$ and a k-bit private exponent were used during testing.

The very high clock speeds achieved reflect the very low critical path delay involved. It will be noted that the delay through the 1024-bit circuit for encryption (decryption) is only a factor of 1.1 (1.2) greater than its 512-bit equivalent. This small increase is again simply a consequence of additional routing delay. These throughput values are significantly greater than would be achievable had these RSA exponentiation circuits been based on the Montgomery multipliers described in [8] and [9], as the critical delay in those cases are much larger than the delays through the circuits presented (Table 1). In addition, a significant increase in critical delay would be expected in moving from a 512-bit to a 1024-bit key size in those cases. This is because the length of the conventional adders used would need to be doubled (Table 1), thus significantly increasing the critical delay through the multipliers.

3.3 Discussion

As discussed, the designs presented are technology independent and thus can be ported to other silicon technologies without difficulty. Thus, even higher data rates should be achievable when migrated to modern ASIC technology. Previous experience indicates [12], for example, that a further doubling in performance should be achievable if these architectures were implemented in 0.18 μm CMOS. The Montgomery multiplier designs presented can also be implemented as arithmetic co-processors on modern microprocessor-based smart cards. Typical RISC processors for such applications require around 80 000 gates with associated co-processors requiring between 80 000 and 165 000 gates, depending on the bit length k, as shown in Table 7.

Table 7: Montgomery multiplier area results (number of gates)

Multiplier	Bit length	Area (number of gates)
Five-to-two CSA	512	82 333
Five-to-two CSA	1024	165 048

The figures in Table 7 are based on the use of a single Montgomery multiplier. Furthermore, by exploiting certain time-area trade-offs, such as reducing the amount of CSL used to evaluate (5) and (6), then the silicon area of our multiplier architectures can be reduced significantly albeit at the expense of some additional clock cycles. For example, the amount of CSL used in the five-to-two CSA multiplier required to compute (5) can be reduced by two-thirds at the expense of two extra clock cycles. It is also expected that the proposed Montgomery multiplier algorithms and techniques can be used to perform the modular multiplication operations required in other public-key cryptosystems which require modular exponentiation or, with some additional thought, elliptic curve cryptosystems. The architectures and techniques presented are therefore highly flexible and can be used in a multitude of different applications, including high-speed network security processors and low silicon area embedded smart cards.

4 Conclusions

Modified Montgomery multiplication and associated RSA modular exponentiation algorithms and circuit architectures have been presented. CSAs are used to perform the large word length additions required by the modified multiplication algorithms. Two such algorithms have been presented, one using a five-to-two CSA and the other a four-to-two CSA with an additional multiplexer. Previous modular multiplication approaches [8, 9] for RSA modular exponentiation use CSAs but require that a lengthy conventional addition be carried out to perform output-to-input conversion at the end of each successive modular

multiplication. A key contribution has been to reformulate and solve the problem of modular multiplication as it relates to RSA exponentiation in such a way as to avoid this costly and previously necessary conversion addition. Consequently, the critical path delay and, hence, the data throughput rate of the resulting Montgomery multiplier architectures are word length independent. The practical application of this approach has been demonstrated by applying this to the design of special purpose RSA processing units with 512-bit and 1024-bit key sizes. The resulting RSA units presented have a very low critical path delay, which is independent of the word length. For the purposes of the work our focus has been on FPGA implementations. However, the approach is general and equally suited to ASIC implementation with further performance benefits. Nevertheless, the RSA performance results presented are the fastest reported to date in the literature.

5 Acknowledgments

Amphion Semiconductor Ltd. and a Northern Ireland Department of Learning postgraduate studentship in the form of a CAST award have funded this research.

6 References

1 Stallings, W.: 'Network and Internetwork Security Principles and Practice' (Prentice Hall, New York, USA, 1995)
2 Menezes, A., Oorschot, P., and Vanstone, S.: 'Handbook of Applied Cryptography' (CRC Press, Florida, USA, 1997)
3 Rivest, R.L., Shamir, A., and Adleman, L.: 'A Method for Obtaining Digital Signatures and Public-Key Cryptosystems', *Commun. ACM*, 1978, **21**, (2), pp. 120–126
4 Montgomery, P.L.: 'Modular Multiplication without Trial Division', *Math. Comput.*, 1985, **44**, pp. 519–521
5 Eldridge, S.E., and Walter, C.D.: 'Hardware Implementation of Montgomery's Modular Multiplication Algorithm', *IEEE Trans. Comput.*, 1993, **42**, pp. 693–699
6 Elbirt, A.J., and Paar, C.: 'Towards an FPGA Architecture Optimized for Public-Key Algorithms'. Presented at the SPIE Symp. on Voice, Video and Communications, Sept. 1999
7 Blum, T., and Paar, C.: 'Montgomery Modular Exponentiation on Reconfigurable Hardware'. Proc. 14th Symp. on Computer Arithmetic, 1999, pp. 70–77
8 Kim, Y.S., Kang, W.S., and Choi, J.R.: 'Implementation of 1024-bit modular processor for RSA cryptosystem'. http://www.ap-asic.org/2000/proceedings/10-4.pdf
9 Bunimov, V., Schimmler, M., and Tolg, B.: 'A Complexity-Effective Version of Montgomery's Algorithm'. Presented at the Workshop on Complexity Effective Designs (WECD02), May 2002
10 Quisquater, J.-J., and Couvreur, C.: 'Fast Decipherment Algorithm for RSA Public-Key Cryptosystem', *Electron. Lett.*, 1982, **18**, pp. 905–907
11 Walter, C.D.: 'Montgomery Exponentiation Needs No Final Subtractions', *Electron. Lett.*, 1999, **35**, (21), pp. 1831–1832
12 Amphion Semiconductor Ltd.: 'Data Security Data Sheets'. http://www.amphion.com