Montgomery modular multiplication architecture for public key cryptosystems

Conference Paper · November 2004

DOI: 10.1109/SIPS.2004.1363075 · Source: IEEE Xplore

3 authors:

Maire Mcloone
Queen's University Belfast
91 PUBLICATIONS **2,364** CITATIONS
SEE PROFILE

Corinne McIvor
University of the West of Scotland
11 PUBLICATIONS **407** CITATIONS
SEE PROFILE

J.V. Mccanny
Queen's University Belfast
210 PUBLICATIONS **3,241** CITATIONS
SEE PROFILE

MONTGOMERY MODULAR MULTIPLICATION ARCHITECTURE FOR PUBLIC KEY CRYPTOSYSTEMS

Máire McLoone, Ciaran McIvor, John V McCanny

Institute of Electronics, Communications and Information Technology,
Queen's University Belfast, Northern Ireland
maire.mcloone@ee.qub.ac.uk, c.mcivor@ee.qub.ac.uk, j.mccanny@ee.qub.ac.uk

ABSTRACT

This paper describes a novel hardware architecture of the Coarsely Integrated Hybrid Scanning (CIHS) algorithm which performs Montgomery modular multiplication. The CIHS algorithm integrates the multiplication and reduction steps involved in modular multiplication. When implemented on a Virtex XC2VP50 device, the architecture can perform 128-bit modular multiplication at a data-rate of 160 Mbps and 256-bit modular multiplication at 216 Mbps. To the authors' knowledge, these are the first reported performance figures for a hardware CIHS algorithm architecture to be reported in the literature. A methodology for generating Montgomery multiplication test vectors is also described.

1. INTRODUCTION

Two of the most common public key cryptosystems are RSA and Elliptic Curve Cryptosystems (ECCs). The RSA algorithm [1], was developed by Rivest, Shamir and Adleman in 1977 and is used in numerous security protocols, such as the IP Security Protocol (IPSec), the Transport Layer Security Protocol (TLS) and the Secure Electronic Transactions (SET) protocol for digital signatures, digital enveloping and to securely exchange keys. The algorithm is based on the number theory problem of factorising the product of two large prime numbers, which is currently computationally infeasible. ECCs were independently proposed by Miller [2] and Koblitz [3] in 1986 and 1987 respectively. They utilise smaller key sizes than RSA, while still maintaining equivalent security levels. A 256-bit ECC key size is equivalent to a 6000-bit RSA key size in terms of security [4]. Modular multiplication is an integral part of both these important cryptosystems. The RSA algorithm utilises multiple modular multiplications, namely modular exponentiation. The critical function in ECCs over GF(p) is scalar multiplication, which comprises a series of doubling and addition operations. Modular multiplication is performed in both the addition and doubling.

One of the most efficient methods which has emerged for performing modular multiplication is Montgomery's algorithm [5]. In modular multiplication, the most time consuming operation is trial division by the modulus. However, in Montgomery's algorithm this trial division is replaced by binary additions and divisions, which are efficient when implemented in hardware and indeed in software. In 1996, Koç, Alcar and Kaliski [6] carried out research on Montgomery's multiplier and developed alternative algorithms for describing the multiplication. In this paper, a hardware architecture is described for one of these alternative algorithms, the Coarsely Integrated Hybrid Scanning (CIHS) algorithm. In the next section Montgomery multiplication is outlined while section 3 provides a detailed description of the CIHS algorithm. In section 4, the CIHS hardware architecture is presented and in section 5 its performance is discussed and compared with equivalent architectures. The generation of Montgomery multiplication test vectors is also discussed in this section. Finally, conclusions are provided in section 6.

2. MONTGOMERY MODULAR MULTIPLICATION

Montgomery multiplication is carried out in *n*-residue format, where the modulus *n* is a *k*-bit integer, *r* is equal to 2^k, and *n* and *r* are relatively prime. The *n*-residue of an integer *x* is defined as,

$$\bar{x} = x.r \,(\mathrm{mod}\, n) \tag{1}$$

The Montgomery product of two *n*-residue values, *x* and *y*, is,

$$\bar{z} = \bar{x}.\bar{y}.r^{-1} \,(\mathrm{mod}\, n) \tag{2}$$

where, r^{-1} is the inverse of *r* modulo *n*. To convert back from *n*-residue format, the product is multiplied by the integer *r*. For Montgomery reduction, a value n' is defined, where

$$r.r^{-1} - n.n' = 1 \tag{3}$$

Therefore, the Montgomery modular multiplication algorithm is defined as follows:

$$\underline{MontMult\ (\bar{x}, \bar{y})}$$
1. $t := \bar{x}.\bar{y}$
2. $m := t.n' \bmod r$
3. $u := (t + m.n)/r$ (4)
4. *if* $u \geq n$ *then return* $u - n$
   else return u

Much analysis has been carried out on Montgomery's algorithm since its proposal. Walter [7, 8] illustrated that it is possible to implement Montgomery exponentiation without the need for modular reduction. He also provided bounds for Montgomery modular multiplication so that the subtraction in step 4 of the algorithm could be avoided [9]. Hachez and Quisquater [10] proved that for software implementations, final reduction in Montgomery exponentiation could still be avoided with better bounds. However, their result did not hold in hardware implementations for small modulo. In this paper, the reduction step is not avoided and the implementations include the subtraction step of the Montgomery modular multiplication algorithm.

3. CIHS ALGORITHM DESCRIPTION

Coarsely Integrated Hybrid Scanning (CIHS) Montgomery multiplication utilises both operand scanning and product scanning [11] methods. Operand scanning comprises a loop, which moves through words of one of the operands, while in product scanning the loop moves through words of the product. The reduction step of Montgomery multiplication is carried out using the operand scanning method only. The overall CIHS algorithm is outlined in Figure 1 [6], where C and S represent the carry and sum respectively, a and b are the input variables to be multiplied, t is the output product, s is the wordsize, n is the modulus and n' is as defined by equation 3.

4. CIHS HARDWARE ARCHITECTURE

The Coarsely Integrated Hybrid Scanning Algorithm contains numerous loops and as such, is a very complex algorithm to implement in hardware. The CIHS architecture comprises three main components, the first CIHS loop, the second CIHS loop and a subtraction component. An overall block diagram of the CIHS algorithm architecture is provided in Figure 2.

$\underline{CIHS\ (a,b,n,n')}$
for $i = 0$ *to* $s-1$ *loop*
  $C := 0$
  for $j = 0$ *to* $s-i-1$ *loop*
    $(C,S) := t[i+j] + a[j]*b[i] + C$
    $t[i+j] := S$
  end loop
  $(C,S) := t[s] + C$
  $t[s] := S$
  $t[s+1] := C$
end loop

Fig 1(a): First CIHS loop

for $i = 0$ *to* $s-1$ *loop*
  $m := t[0]*n'[0] \bmod W$
  $(C,S) := t[0] + m*n[0]$
  for $j = 1$ *to* $s-1$ *loop*
    $(C,S) := t[j] + m*n[j] + C$
    $t[j-1] := S$
  end loop
  $(C,S) := t[s] + C$
  $t[s-1] := S$
  $t[s] := t[s+1] + C$
  $t[s+1] := 0$

  for $j = i+1$ *to* $s-1$ *loop*
    $(C,S) := t[s-1] + b[j]*a[s-j+i]$
    $t[s-1] := S$
    $(C,S) := t[s] + C$
    $t[s] := S$
    $t[s+1] := C$
  end loop
end loop

Fig 1(b): Second CIHS loop

In this paper, a state machine methodology has been developed to implement the two CIHS algorithm loops. The architecture assumes a wordsize, $s = 4$. Therefore, for a 128-bit modular multiplication, the inputs a and b will be divided into 4 x 32-bit blocks, $a(0)$ to $a(3)$ and $b(0)$ to $b(3)$. The contents of the j loops within each of the two CIHS algorithm loops shown in Figure 1, are implemented as a core block, with inputs x, y, C_{in} and t_{in} and outputs, C_{out} and t_{out}, as illustrated in Figure 3.

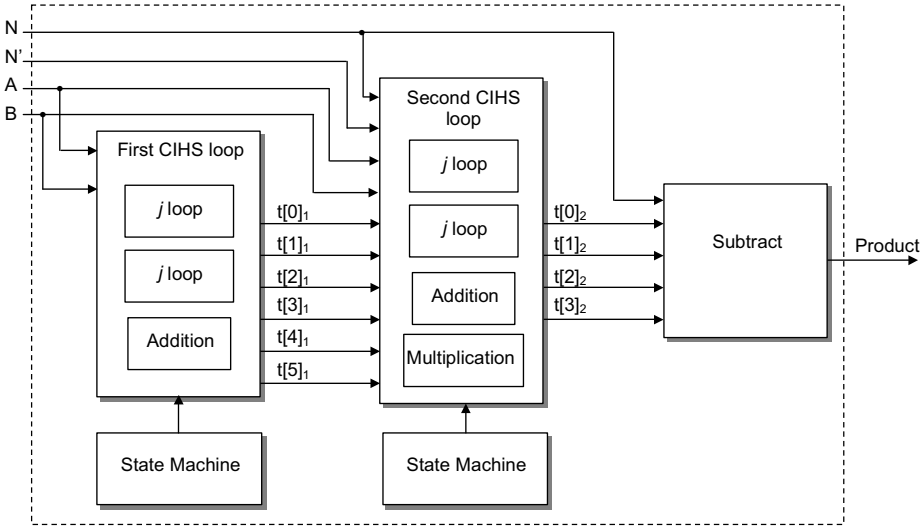

Figure 2: Overall CIHS Algorithm Architecture

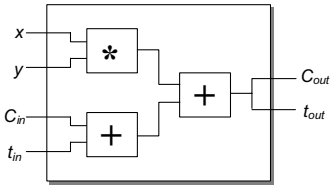

Figure 3: CIHS Algorithm *j* loop component

Multiplication and addition blocks are also utilised. The algorithm is studied to determine the maximum number of components required within one stage of the state machine. The first CIHS loop requires a 7-stage state machine, in which each state performs a combination of a *j* loop component, *j* loop components in parallel and/or additions. Table 1 shows the inputs and outputs of each component required in the first loop. The second CIHS loop is more complicated and requires a 20-stage state machine. The inputs and outputs of each component required in the second loop are given in Table 2. In each state of the second loop a combination of a *j* loop component, *j* loop components in parallel and/or multiplications and additions will be performed.

The reduction step of the Montgomery multiplication algorithm proves to be the critical path of the overall architecture if implemented as a full-length subtraction. This is avoided in the architecture described in this paper by splitting the data length into the assumed wordsize blocks of $s = 4$. Therefore, a 128-bit subtraction input, u, will be divided into 4 x 32-bit blocks, $u(0)$ to $u(3)$. Similarly, the modulus, n is split into four parts and a comparison of the individual parts is conducted, with each stage separated by registers to improve the overall timing. This concept is outlined in Table 3 and an 8-bit data length example provided, where the input u = '00001010' and the input n = '11100000'.

State	Comp.	Inputs	Outputs
0	*j* loop	a(0), b(0), '0', '0'	C_{out0}, $t[0]_1$
1	*j* loop	a(1), b(0), C_{out0}, '0'	C_{out1}, t_{out0}
2	*j* loop	a(2), b(0), C_{out1}, '0'	C_{out2}, t_{out1}
	j loop	a(0), b(1), '0', t_{out0}	C_{out3}, $t[1]_1$
3	*j* loop	a(3), b(0), C_{out2}, '0'	C_{out4}, t_{out2}
	j loop	a(1), b(1), C_{out3}, t_{out1}	C_{out5}, t_{out3}
4	*j* loop	a(2), b(1), C_{out5}, t_{out2}	C_{out6}, t_{out4}
	j loop	a(0), b(2), '0', t_{out3}	C_{out7}, $t[2]_1$
5	Addition	C_{out6}, C_{out4}	A_0
	j loop	a(1), b(2), C_{out7}, t_{out4}	C_{out8}, t_{out6}
6	Addition	A_0, C_{out8}	A_1
	j loop	a(0), b(3), '0', t_{out6}	C_{out9}, $t[3]_1$
7	Addition	A_1, C_{out9}	$t[4]_1$& $[5]_1$

Table 1: Components Required by First CIHS loop

5. GENERATION OF TEST VECTORS & PERFORMANCE EVALUATION

In order to verify the operation of the CIHS algorithm architecture, it was necessary to generate Montgomery multiplication test vectors, as these, to the author's knowledge, have not been provided in the literature to date. Test vectors are developed for 128-bit and 256-bit Montgomery multiplications. In order to perform calculations on such large data lengths, the Big Number Calculator developed by Reinhold [12] was required. Utilising this calculator, the 128-bit test vectors are generated as follows:

Step 1: Find the value of $r = 2^{128}$.

Step 2: Select a 128-bit prime number, n, as the modulus. Set as modulus on Big Number Calculator.
Step 3: Using Calculator, calculate r^{-1}.
Step 4: Remove n as modulus.
Step 5: Calculate n', where:

$$n' = \frac{(r \times r^{-1}) - 1}{n}$$

Step 6: Find two 128-bit numbers, A and B.
Step 7: Calculate the Montgomery Product, P, where,

$$P = A \times B \times r^{-1} \pmod{n}$$

The 256-bit Montgomery multiplication test vectors are generated in a similar manner, where $r = 2^{256}$, and the modulus, n and prime numbers A and B are 256-bits in length. The 128-bit test vectors are provided in the Appendix.

State	Component	Inputs	Outputs
0	Multiplication	$t[0]_1$, n'(0)	m_0
1	*j* loop	m_0, n(0), '0', $t[0]_1$	C_{out0}, *null*
2	*j* loop	m_0, n(1), C_{out0}, $t[1]_1$	C_{out1}, t_{out0}
3	Multiplication	t_{out0}, n'(0)	m_1
4	*j* loop *j* loop	m_0, n(2), C_{out1}, $t[2]_1$ m_1, n(0), '0', t_{out0}	C_{out2}, t_{out1} C_{out3}, *null*
5	*j* loop *j* loop	m_0, n(3), C_{out2}, $t[3]_1$ m_1, n(1), C_{out3}, t_{out1}	C_{out4}, t_{out2} C_{out5}, t_{out3}
6	Addition *j* loop Multiplication	$t[4]_1$, C_{out4} m_1, n(2), C_{out5}, t_{out2} t_{out4}, n'(0)	A_{0_0}, A_{0_1} C_{out6}, t_{out4} m_2
7	Addition *j* loop *j* loop	$t[5]_1$, A_{0_1} a(3), b(1), '0', A_{0_0} m_2, n(0), '0', t_{out3}	A_1 C_{out7}, t_{out5} C_{out8}, *null*
8	Addition *j* loop *j* loop	C_{out7}, A_1 a(2), b(2), '0', t_{out5} m_2, n(1), C_{out8}, t_{out4}	A_2 C_{out9}, t_{out6} C_{out10}, t_{out7}
9	Addition Multiplication	C_{out9}, A_2 t_{out7}, n'(0)	A_3 m_3
10	*j* loop *j* loop	a(1), b(3), '0', t_{out6} m_3, n(0), '0', t_{out7}	C_{out11}, t_{out8} C_{out12}, *null*
11	Addition *j* loop	C_{out11}, A_3 m_1, n(3), C_{out6}, t_{out8}	A_{4_0}, A_{4_1} C_{out13}, t_{out9}
12	Addition *j* loop	C_{out13}, A_{4_0} m_2, n(2), C_{out10}, t_{out9}	A_{5_0}, A_{5_1} C_{out14}, t_{out10}
13	Addition *j* loop *j* loop	A_{4_1}, A_{5_1} a(3), b(2), '0', A_{5_0} m_3, n(1), C_{out12}, t_{out10}	A_6 C_{out15}, t_{out11} C_{out16}, $t[0]_2$
14	Addition *j* loop	C_{out15}, A_6 a(2), b(3), '0', t_{out11}	A_7 C_{out17}, t_{out12}
15	Addition *j* loop	C_{out17}, A_7 m_2, n(3), C_{out14}, t_{out12}	A_{8_0}, A_{8_1} C_{out18}, t_{out13}
16	Addition *j* loop	A_{8_0}, C_{out18} m_3, n(2), C_{out16}, t_{out13}	A_{9_0}, A_{9_1} C_{out19}, $t[1]_2$
17	Addition *j* loop	A_{8_1}, A_{9_1} a(3), b(3), '0', A_{9_0}	A_{10} C_{out20}, t_{out14}
18	Addition *j* loop	A_{10}, C_{out20} m_3, n(3), C_{out19}, t_{out14}	A_{11} C_{out21}, $t[2]_2$
19	Addition	A_{11}, C_{out21}	$t[3]_2$

Table 2: Components Required by Second CIHS loop

Condition (Let $t = u(3)$ & $u(2)$ & $u(1)$ & $u(0)$)		**Example:** u = '00 00 10 10' n = '11 10 00 00'
Check if: $u(0) \geq n(0)$	*If yes*: $u0\_TRUE = 1$ *If no*: $u0\_TRUE = 0$	$u0\_TRUE = 1$
Check if: $u(1) \geq n(1)$	*If yes*: $u1\_TRUE = 1$ *If no*: $u1\_TRUE = 0$	$u1\_TRUE = 1$
Check if: $u(2) \geq n(2)$	*If yes*: $u2\_TRUE = 1$ *If no*: $u2\_TRUE = 0$	$u2\_TRUE = 0$
Check if: $u(3) \geq n(3)$	*If yes*: $u3\_TRUE = 1$ *If no*: $u3\_TRUE = 0$	$U3\_TRUE = 0$
If $u0\_TRUE = 1$ *then* *else*	Output $= t - n$ Output $= t$	Output $= t - n$
If $u1\_TRUE = 1$ *then* *else*	Output $= t - n$ Output $= t$	Output $= t - n$
If $u2\_TRUE = 1$ *then* *else*	Output $= t - n$ Output $= t$	Output $= t$
If $u3\_TRUE = 1$ *then* *else*	Output $= t - n$ Output $= t$	Output $= t$

Table 3: Reduction Step of Montgomery Multiplication Algorithm

In order to evaluate the performance of the CIHS algorithm hardware architecture 128-bit and 256-bit designs were implemented on Xilinx Virtex-II Pro FPGA devices. The designs were simulated using Modelsim and synthesised using Synplify Pro v7.3.1 and Xilinx Foundation software v5.2i. The algorithm inputs, A, B and n were input in 32-bit blocks to reduce the overall IOB count. The 18x18 multipliers available on the Virtex-II devices were used to implement the multiplication operations. Also, since a number of the component signals have a significantly high fanout, Synplify performed replication in order to improve timing. Future work will involve carrying out manual replication.

The 128-bit CIHS architecture implemented on an XC2VP50 device requires 2203 CLB slices and 19 18x18 multiplier blocks. It achieves a clock speed of 77 MHz and hence a throughput of 160 Mbps. The 256-bit CIHS architecture implemented on the same device utilises 5067 of 23616 slices and 74 of 232 18x18 multipliers. It runs at a throughput of 216 Mbps.

Although there are no previous implementations of the CIHS algorithm to the authors' knowledge, hardware implementations of the other alternative algorithms proposed by Koç *et al* do exist. The alternative Separated Operand Scanning (SOS), Coarsely Integrated Operand Scanning (CIOS) and Finely Integrated Operand Scanning (FIOS) algorithms all utilise operand scanning, and either separated or integrated multiplication and reduction. Satoh and Takano [13] use the FIOS Montgomery multiplication algorithm in their implementation of a scalable elliptic curve cryptographic

processor. Their 160-bit FIOS multiplication is carried out in 72 clock cycles for a wordsize of 32-bits. Throughput figures for the multiplication are not provided. The architecture described in this paper performs 128-bit and 256-bit multiplication in only 62 and 66 clock cycles respectively. Previous work by the authors' [14] examined hardware architectures of the CIOS, FIOS and SOS algorithms on XC2VP50 devices. The performance results of these algorithms are compared with those achieved by the CIHS architecture in Table 4.

Algorithm Wordsize	Clock Speed (MHz)	Area (slices & multipliers)	No. of clock cycles	Throughput (Mbps)
128-bits:				
CIHS: 4 x 32bits	77	2203 slices 19 MULTs	62	160
SOS: 4 x 32bits[14]	129.6	1216 slices 4 MULTs	177	93.75
CIOS: 4 x 32bits[14]	126.7	936 slices 4 MULTs	162	100
FIOS: 4 x 32bits[14]	129.6	1216 slices 4 MULTs	177	93.75
256-bits:				
CIHS: 8 x 32bits	55.8	5067 slices 74 MULTs	66	216
SOS: 8 x 32bits[14]	83	2070 slices 4 MULTs	609	34.95
CIOS: 8 x 32bits[14]	101.8	1433 slices 4 MULTs	582	44.8
FIOS: 8 x 32bits[14]	66.9	1709 slices 4 MULTs	590	29.06

Table 4: Performance Comparison of CIHS, SOS, CIOS & FIOS Algorithms

The 128-bit CIHS architecture is 1.6 times faster and the 256-bit CIHS architecture 4.8 times faster than the equivalent alterative algorithm architectures previously reported. This is due to the proposed state machine methodology used in implementing the CIHS algorithm. This concept results in the Montgomery multiplication being performed in a shorter number of cycles in comparison to the other algorithms. The main advantage of the method is that when the data length is increased, the latency only increases by the number of extra cycles required to input the 32-bit blocks. For example, the 128-bit data length is input over 4 clock cycles, whereas the 256-bit data length is input over 8 cycles. Therefore, 256-bit multiplication will take only 4 extra clock cycles over 128-bit multiplication. However, the state machine methodology requires additional area in terms of both slices and multipliers since both the first and second CIHS loop architectures have components which operate in parallel.

6. CONCLUSIONS

Modular multiplication is a critical operation in numerous public key cryptosystems such as RSA, El Gamal and ECCs, and also in key exchange protocols such as Diffie-Hellman key exchange. In this paper a highly efficient CIHS algorithm architecture is described which performs Montgomery modular multiplication. The Montgomery modular multiplication method is well-suited for hardware implementations as it performs modular division using binary additions and divisions.

Since the CIHS algorithm is very complex, with numerous loops, a state machine methodology was developed for ease of implementation. This state machine methodology is a systematic approach which can also be utilised to implement the other complex algorithms (SOS, CIOS and FIOS) described by Koç *et al* [6]. The CIHS algorithm architecture implemented on a Xilinx Virtex-II Pro FPGA performs 128-bit and 256-bit multiplication at 160 Mbps and 216 Mbps respectively. These results are faster than previous hardware implementations of the SOS, CIOS and FIOS algorithms and to the authors' knowledge, are the first hardware performance figures provided for the CIHS algorithm.

The principle advantage of the modular multiplication architecture described in this paper over other implementation methods is its ability to perform multiplication in a small number of clock cycles regardless of the data-length.

Future work will involve utilizing the state machine methodology to implement the SOS, CIOS and FIOS algorithms. Also, the CIHS architecture will be further developed to perform 512-bit and 1024-bit multiplications.

7. ACKNOWLEDGEMENTS

This research has been funded under a Royal Academy of Engineering/EPSRC post-doctoral research fellowship.

8. REFERENCES

[1] Rivest, R.L., Shamir, A., Adleman, L.: "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". Communications of the ACM, 21(2): 120–126, February 1978.

[2] Miller, V.S.: "Use of Elliptic Curves in Cryptography". Advances in Cryptology (Crypto' 85), pp. 417-426, 1986.

[3] Koblitz, N.: "Elliptic Curve Cryptosystems". Math. Computing, Vol. 48, pp. 203-209, 1987.

[4] Vanstone, S.A., "Next Generation Security for Wireless: Elliptic Curve Cryptography" , http://www.compseconline.com/hottopics/hottopic20_8/ Next.pdf, March 2004.

[5] Montgomery, P.L.: "Modular Multiplication without Trial Division", Mathematics of Computation, Vol. 44, pp 512-521, April 1985.
[6] Koç, C.K., Acar, T., Kaliski Jr., B.S., "Analyzing and Comparing Montgomery Multiplication Algorithms", *IEEE Micro*., Vol. 16, No. 3, pp 26-33, June 1996.
[7] Walter, C. D., "Montgomery Exponentiation Needs no Final Subtractions", *Electronics Letters*, 35(21):1831-1832, October 1999.
[8] Walter, C. D., "Montgomery's Multiplication Technique: How to make it Smaller and Faster", Workshop on Cryptographic Hardware and Embedded Systems (CHES'99), LNCS 1717, pp 80-93, August 1999.
[9] Walter, C. D., "Precise Bounds for Montgomery Modular Multiplication and some Potentially Insecure RSA Moduli", Topics in Cryptology – CT-RSA 2002, LNCS 2271, pp 30-39, 2002.
[10] Hachez, G., Quisquater, J., "Montgomery Exponentiation with no Final Subtractions: Improved Results", Workshop on Cryptographic Hardware and Embedded Systems (CHES'00), LNCS 1965, pp 293-301, July 2000.
[11] Kaliski Jr., B.S., "The Z80180 and Big-Number Arithmetic", Dr.Dobb's, pp 50-58, September 1993.
[12] Reinhold, A., "Big Number Calculator Applet", URL: http://world.std.com/~reinhold/BigNumCalc.html, Mar 2004.
[13] Satoh, A., Takano, K., "A Scalable Dual-Field Elliptic Curve Cryptographic Processor", *IEEE Transactions on Computers*, vol. 52, no. 4, April 2003.
[14] McIvor, C, McLoone, M, McCanny, J., "FPGA Montgomery Multiplier Architectures - A Comparsion", IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), April 2004.

9. APPENDICES

128-bit Montgomery Modular Multiplication Test Vectors

```
TEST 1:
-------
n    = eee74404d129949520704c5bf5814703
r    = 100000000000000000000000000000000
r-1  = 9acc3fdac783b519dd82e86481aa4f41
n'   = a5e0296c5b1d29d6b905ba8ad65d2455
A    = 223520375bd184b2bac64c9d1a6c55fa
B    = d857ed0720d590d61f05c150e1e40917
MonMult = e3bd635debc8021ea0208d75df078ea6


TEST 2:
---------
n    = f0fe9f3c608d779379bb3676fdb85071
r    = 100000000000000000000000000000000
r-1  = bec378d4cd1ac607faef5b46eb7928cd
n'   = caa42ecf90315ae71b89b6193d868f6f
A    = adfdb6089758064a69aad900ad18274b
B    = 828994fe60ddcab48671399fd349b0ff
MonMult = d848da961b4c3092def8bdaca7a73bee


TEST 3:
---------
n    = dd54fd6aa41a0dbcb7550b284862b7a5
r    = 100000000000000000000000000000000
r-1  = 16761e2f9128508ad135cc6b6d6218d
n'   = 19fac77336272731ae87fe0e59ea7d3
A    = c9d5398bf1cec1342f3c7cca33ea04a6
B    = 2f5bf07df1f473c46318eea49d7f1de3
MonMult = 7dccdc7deb3d1a1b3afb2b7c0ca5c53a


TEST 4:
---------
n    = 9848765c71a41c12921ca63ca42a203d
r    = 100000000000000000000000000000000
r-1  = 4756c6a22f112c1b804bb540fb47b820
n'   = 77ed2ff5fed15b1c611423518c4488eb
A    = c52c1e570e620174fcbb51063ecbfcb1
B    = 4b1a9caa8e183aabd89e8f3ab7561273
MonMult = 55352eb5475ce94fefcce0a8b687044f


TEST 5:
---------
n    = c71a6ffca861df3a175c6eb226581289
r    = 100000000000000000000000000000000
r-1  = b12e9e5600d31abbdf15fd645ec00471
n'   = e3d08525994d7aa7d1556631cbb4fc47
A    = b253687d5f73bacf1fbd542d5d604272
B    = af9764ea40aa14153e8af13177851ab1
MonMult = 8b89addf457e084b752d716c73d29821
```