Coarsely integrated operand scanning (CIOS) architecture for high-speed Montgomery modular multiplication

Conference Paper · January 2005

DOI: 10.1109/FPT.2004.1393267 · Source: IEEE Xplore

3 authors:

Maire Mcloone
Queen's University Belfast
91 PUBLICATIONS **2,577** CITATIONS
SEE PROFILE

Corinne McIvor
University of the West of Scotland
11 PUBLICATIONS **428** CITATIONS
SEE PROFILE

J.V. Mccanny
Queen's University Belfast
210 PUBLICATIONS **3,341** CITATIONS
SEE PROFILE

Coarsely Integrated Operand Scanning (CIOS) Architecture For High-Speed Montgomery Modular Multiplication

Máire McLoone, Ciaran McIvor, John V McCanny
The Institute of Electronics, Communications & Information Technology (ECIT)
Queen's University Belfast, Northern Ireland Science Park,
Queen's Road, Queen's Island, Belfast BT3 9DT

Abstract

A generic Coarsely Integrated Operand Scanning (CIOS) architecture that provides high speed Montgomery modular multiplication is presented in this paper. The architecture is capable of supporting varying operand sizes. It achieves a throughput of 210 Mbps, 289 Mbps and 334 Mbps for 128-bit, 256-bit and 512-bit operand sizes respectively, when implemented on a Virtex XC2VP50 FPGA. Throughputs of up to 400 Mbps are achieved if the final subtraction in the Montgomery algorithm is excluded. To the authors' knowledge this is the fastest Montgomery multiplication architecture reported in the literature.

1. Introduction

The CIOS algorithm is one of five Montgomery multiplication algorithms proposed by Koç *et al* [1]. In the algorithm, multiplication and reduction are integrated and are carried out using operand scanning, where the operations are performed on words of one of the operands.

Montgomery multiplication was developed in 1985 [2] and has emerged as the most efficient way to perform modular multiplication in both hardware and software. It is highly efficient as it replaces division by the modulus, *n*, with division modulo a power of 2, an operation which is easily implemented on hardware and software platforms.

Elliptic Curve Cryptosystems [7, 8] comprise modular multiplications, and modular exponentiation is an integral part of the Diffie-Hellman key exchange protocol [3], the Digital Signature Algorithm (DSA) [4] and of public key algorithms, such as RSA [5] and ElGamal [6]. Modular exponentiation comprises multiple modular multiplications and this proves to be the bottleneck in implementations of these algorithms. Therefore, a high-speed efficient modular multiplication implementation is essential in order to provide the security requirements of present day real-time communication applications.

This paper describes a high-speed modular multiplication architecture, which can support numerous multiplication data lengths. The architecture utilizes a state machine methodology which allows the complex CIOS algorithm to be easily implemented. Although a significant amount of research work has been published on architectures which comprise Montgomery multiplication [9-13], they fail to provide specific performance metrics. A number of publications do provide comparative results and these are discussed in section 4 of this paper. The only known previous hardware architecture which uses the CIOS algorithm is the work of the authors [14]. This paper improves on the previous work by supporting multiple operand sizes and achieving a higher throughput.

In section 2, the CIOS algorithm is explained. Section 3 describes the architecture that has been developed, and performance results are provided in section 4. Finally, conclusions are given in section 5.

2. The Coarsely Integrated Operand Scanning (CIOS) Algorithm

The CIOS algorithm is a representation of the Montgomery modular multiplication algorithm.

2.1 Montgomery Multiplication

Montgomery multiplication is performed on integers in *n*-residue format, where the modulus *n* is a *k*-bit integer, $r = 2^k$, and *n* and *r* are relatively prime. The *n*-residue of an integer *y* is given by,

$$\overline{y} = y.r \pmod n \qquad (1)$$

and the Montgomery multiplication of two *n*-residue integers, *x* and *y*, is,

$$\overline{z} = \overline{x}.\overline{y}.r^{-1} \pmod n \qquad (2)$$

where, r^{-1} is the inverse of *r* modulo *n*. To obtain the original integer given the *n*-residue form, it is necessary to multiply the *n*-residue product by the integer r. In Montgomery multiplication, a value n' is also required, such that,

$$r.r^{-1} - n.n' = 1 \quad (3)$$

The overall Montgomery modular multiplication algorithm can be described as follows [15]:

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

2.2 The CIOS Algorithm

The CIOS algorithm alternates between the multiplication and reduction steps involved in Montgomery modular multiplication, rather than carrying out full multiplication followed by reduction. These multiplications and reductions are conducted using two loops, where an outer loop cycles through words of one of the operands. The CIOS algorithm is described in Figure 1, where s is the number of words in the operand, C and S are the carry and sum respectively, t is the output product, a and b are the input operands, n' is as defined by equation 3, $W = 2^w$, where w is the wordsize of the computer, and n is the modulus.

Step 4 of the Montgomery multiplication algorithm is not included in the CIOS algorithm description. Under certain conditions, this subtraction step is not necessary. It has been proven that with a correct hardware implementation and within particular bounds, Montgomery exponentiation can be carried out without utilising any modular subtraction [16-18]. It has also been shown that the subtraction step can be avoided in software implementations of Montgomery exponentiation with even stricter bounds [19]. However, this does not hold for hardware implementations when the modulus length is smaller than the multiplier length.

In this paper, we present and compare Montgomery modular multiplication architectures, which include final subtraction, and architectures, which do no include this step.

3. CIOS Algorithm Architecture

The CIOS architecture described in this paper assumes that the number of words, s, in each operand is 4. Therefore, from Figure 1, it is evident that the i loop and both j loops need to be unrolled 4 times, leading to quite a complex algorithm to implement. This paper proposes the use of a state machine methodology for ease of implementation. This involves outlining the core components required at any one stage of the algorithm's execution, and therefore, during any one state of the state machine.

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

Figure 1. CIOS Algorithm

The components required by the CIOS algorithm and the inputs and outputs of each component, are outlined in Table 1. Thus, the components needed to implement the CIOS algorithm per state include a multiplication, an addition and two j loop components. A j loop component is the realization of the j loops in the CIOS algorithm description of Figure 1, and comprises a multiplication and two additions. These components are performed in parallel and the iterative architecture cycles through them during each state. Overall, an 18-state state machine is utilized to execute the complete CIOS algorithm and to obtain the 4-word output product, t = $t[0]$, $t[1]$, $t[2]$, $t[3]$. Finally, if step 4 of the Montgomery multiplication algorithm is to be implemented, a subtract block is required. This subtract block is the critical path of the overall architecture since it involves a comparison of the full-length modulus with the full-length product operand. In the architecture described here, the full-length product and modulus are split into s = 4 words and the individual words are compared. As such, step 4 of the Montgomery multiplication algorithm is implemented according to the pseudo-code outlined in Figure 2. This allows the individual word comparisons to be registered, significantly improving the speed of the subtraction block at the expense of a 4-cycle latency. An outline of the CIOS architecture incorporating the subtract block is depicted in Figure 3.

Table 1. Components Required by CIOS Architecture (s = 4)

State	Comp.	Inputs	Outputs
0	*j* loop	a(0), b(0), '0', '0'	C_{out0}, t_{out0_0}
1	*j* loop	a(1), b(0), C_{out0}, '0'	C_{out1}, t_{out0_1}
	Mult	t_{out0_0}, n'(0)	m_0
2	*j* loop	a(2), b(0), C_{out1}, '0'	C_{out2}, t_{out0_2}
	j loop	m_0, n(0), '0', t_{out0_0}	C_{out3}, *null*
3	*j* loop	a(3), b(0), C_{out2}, '0'	C_{out4}, t_{out0_3}
	j loop	m_0, n(1), C_{out3}, t_{out0_1}	C_{out5}, t_{out1_0}
4	Add	C_{out4}, '0'	t_{out0_4}, t_{out0_5}
	j loop	m_0, n(2), C_{out5}, t_{out0_2}	C_{out6}, t_{out1_1}
	j loop	a(0), b(1), '0', t_{out1_0}	C_{out7}, t_{out2_0}
5	*j* loop	m_0, n(3), C_{out6}, t_{out0_3}	C_{out8}, t_{out1_2}
	j loop	a(1), b(1), C_{out7}, t_{out1_1}	C_{out9}, t_{out2_1}
	Mult	t_{out2_0}, n'(0)	m_1
6	Add	C_{out8}, t_{out0_4}	*Csig*, t_{out1_3}
	j loop	a(2), b(1), C_{out9}, t_{out1_2}	C_{out10}, t_{out2_2}
	j loop	m_1, n(0), '0', t_{out2_0}	C_{out11}, *null1*
7	Add	*Csig*, t_{out0_5}	t_{out1_4}
	j loop	a(3), b(1), C_{out10}, t_{out1_3}	C_{out12}, t_{out2_3}
	j loop	m_1, n(1), C_{out11}, t_{out2_1}	C_{out13}, t_{out3_0}
8	Add	C_{out12}, t_{out1_4}	t_{out2_4}, t_{out2_5}
	j loop	m_1, n(2), C_{out13}, t_{out2_2}	C_{out14}, t_{out3_1}
	j loop	a(0), b(2), '0', t_{out3_0}	C_{out15}, t_{out4_0}
9	*j* loop	m_1, n(3), C_{out14}, t_{out2_3}	C_{out16}, t_{out3_2}
	j loop	a(1), b(2), C_{out15}, t_{out3_1}	C_{out17}, t_{out4_1}
	Mult	t_{out4_0}, n'(0)	m_2
10	Add	C_{out16}, t_{out2_4}	*Csig1*, t_{out3_3}
	j loop	a(2), b(2), C_{out17}, t_{out3_2}	C_{out18}, t_{out4_2}
	j loop	m_2, n(0), '0', t_{out4_0}	C_{out19}, *null2*
11	Add	*Csig1*, t_{out2_5}	t_{out3_4}
	j loop	a(3), b(2), C_{out18}, t_{out3_3}	C_{out20}, t_{out4_32}
	j loop	m_2, n(10), C_{out19}, t_{out4_1}	C_{out21}, t_{out5_0}
12	Add	C_{out20}, t_{out3_4}	t_{out4_4}, t_{out4_5}
	j loop	m_2, n(2), C_{out21}, t_{out4_2}	C_{out22}, t_{out5_1}
	j loop	a(0), b(3), '0', t_{out5_0}	C_{out23}, t_{out6_0}
13	*j* loop	m_2, n(3), C_{out22}, t_{out4_3}	C_{out24}, t_{out5_2}
	j loop	a(1), b(3), C_{out23}, t_{out5_1}	C_{out25}, t_{out6_1}
	Mult	t_{out6_0}, n'(0)	m_3
14	Add	C_{out24}, t_{out4_4}	*Csig2*, t_{out5_3}
	j loop	a(2), b(3), C_{out25}, t_{out5_2}	C_{out26}, t_{out6_2}
	j loop	m_3, n(0), '0', t_{out6_0}	C_{out27}, *null3*
15	Add	*Csig2*, t_{out4_5}	t_{out5_4}
	j loop	a(3), b(3), C_{out26}, t_{out5_3}	C_{out28}, t_{out6_3}
	j loop	m_3, n(1), C_{out27}, t_{out6_1}	C_{out29}, t[0]
16	Add	C_{out28}, t_{out5_4}	t_{out6_4}, t_{out6_5}
	j loop	m_3, n(2), C_{out29}, t_{out6_2}	C_{out30}, t[1]
17	*j* loop	m_3, n(3), C_{out30}, t_{out6_3}	C_{out31}, t[2]
18	Add	C_{out31}, t_{out6_4}	t[3]

4. Performance of CIOS Architecture

The generic CIOS architecture described in this paper supports varying operand sizes. If a different multiplication length is required, the architecture does not need to be re-designed - the generic operand length is simply changed and the design re-synthesised.

```
for $i = 0$ to 3 loop
    if $t(i) \geq n(i)$
        $flag(i) = 1$
    else
        $flag(i) = 0$
    end if
end loop
for $i = 0$ to 3 loop
    if $flag(i) = 1$
        $Product = t - n$
    else
        $Product = t$
    end if
end loop
```

Figure 2. Pseudo-Code for Subtract Block

To analyse the architecture's performance, 128-bit, 256-bit and 512-bit multiplier designs, with and without the subtract block, were implemented on XC2VP50 Virtex-II Pro devices. The designs were simulated using Modelsim and synthesised using Synplfiy Pro v7.3.1 and Xilinx Foundation software v5.2i. In order to verify correct operation of the CIOS multiplier architectures, test vectors were generated using the Big Number Calculator developed by Reinhold [20]. The following calculations are performed:

$$r = 2^k \tag{5}$$

where k is the operand length, and,

$$r^{-1}(\text{mod } n) \tag{6}$$

Next, n' is found, where,

$$n' = [(r * r^{-1}) - 1] / n \tag{6}$$

Finally, the Montgomery product, P, of two k-bit operands, A and B is calculated such that,

$$P = A * B * r^{-1}(\text{mod } n) \tag{6}$$

The 256-bit and 512-bit Montgomery multiplication test vectors are given in the Appendix. The 128-bit test-vectors are provided in [15].

The 128-bit, 256-bit and 512-bit operands, A, B and n are input in 32-bit blocks, thus reducing the overall IOB count.

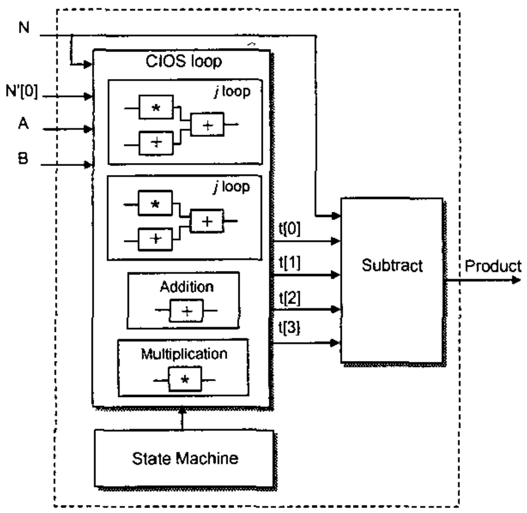

Figure 3. CIOS Algorithm Architecture

The full-length operand n' is not input since only the first word, $n'[0]$ is utilised in the algorithm. When $s = 4, n'[0]$ is 32-bits, 64-bits or 128-bits in length and this is also input in 32-bit blocks.

The multipliers in the architecture are implemented using the 18x18-bit multiplier blocks available on the Virtex-II devices. Also, since there are a number of high fanout signals in the design, Synplify performs replication to improve timing. The performance results for the 128-bit, 256-bit and 512-bit multiplier architectures, with and without the subtract block, are outlined in Table 2.

Removing the subtract block from the Montgomery multiplication architecture does not significantly improve the overall clock speed. It does reduce the overall latency since the 4-cycle latency of the individual word comparisons is not incurred and hence, the throughput is improved. Evidently, the area is also reduced.

The five Montgomery multiplication methods described by Koç et al [1], are the CIOS algorithm, the Separated Operand Scanning (SOS) algorithm, the Finely Integrated Operand Scanning (FIOS) algorithm, the Finely Integrated Product Scanning (FIPS) algorithm and the Coarsely Integrated Hybrid Scanning (CIHS) algorithm. In their paper they provide performance figures for software implementations of these algorithms and conclude that the CIOS method is the most efficient. A 512-bit CIOS multiplication implemented in Assembly code on a Pentium-60 Linux system is executed in 0.122 ms [1]. In work previously carried out by the authors [14], hardware architectures of the SOS, CIOS and FIOS algorithms were implemented on Xilinx Virtex-II Pro devices and compared. The CIOS multiplier proved to be the fastest with the 128-bit and 256-bit multiplier architectures both achieving speeds of 100 Mbps. Satoh and Takano [21] describe a scalable dual-field elliptic curve processor which comprises a Montgomery multiplication architecture. When implemented using a 0.13 μm CMOS standard cell library, 256-bit multiplication is executed in 162 clock cycles. No throughput or area metrics for the multiplication architecture are given. Tenca and Koç [22] present a modular multiplication design which is scalable in terms of both area and operand size. Their 256-bit multiplication design is implemented using 0.5 μm CMOS technology and for a wordsize of 8 and 40 pipeline stages, it executes in 7.4 μs. All of these hardware Montgomery multiplication implementations exclude the conditional subtract.

Table 2. CIOS Architecture Performance Results

Architecture Type	Clock Speed (MHz)	Area (slices & multipliers)	No. of clock cycles	Through-put (Mbps)
128-bit: Subtract	70.7	1522 slices 11 mults	43	210 (0.6 μs)
128-bit: No Subtract	72	1341 slices 11 mults	39	236.3 (0.54 μs)
256-bit: Subtract	53	3512 slices 42 mults	47	289 (0.89 μs)
256-bit: No Subtract	56	3208 slices 42 mults	43	334 (0.77 μs)
512-bit: Subtract	35.8	8299 slices 164 mults	55	333.6 (1.5 μs)
512-bit: No Subtract	40	7704 slices 164 mults	51	401.5 (1.3 μs)

The CIOS architectures presented here, and even those incorporating the subtract block, outperform these previously reported Montgomery multiplication architectures and in particular, they are capable of executing any length of multiplication in a significantly lower number of clock cycles. The difference in latency for the 128-bit, 256-bit and 512-bit operand lengths lies in the extra number of clock cycles necessary to input the values in 32-bit blocks. Therefore, the execution times for the 256-bit and 512-bit multiplications will be 4 and 12 clock cycles, respectively, more than the execution time for the 128-bit multiplication.

5. Conclusion

This paper describes a high speed Montgomery modular multiplication architecture. The CIOS algorithm, which is the most efficient [1, 14] of the five algorithms proposed by Koç *et al*, is utilised to perform the Montgomery multiplication. The CIOS algorithm hardware architecture supports varying operand lengths. When the final subtraction of Montgomery's algorithm is included in the design, it runs at speeds of up to 334 Mbps. When the subtraction is not incorporated, the architecture achieves speeds of up to 400 Mbps. The final subtraction step can be removed if the multiplication architecture is used in implementations of Montgomery exponentiation, which is required by commonly used public key algorithms, such as RSA and El Gamal.

The high speed Montgomery multiplication architecture is accomplished through implementation using a state machine methodology. This enables the complex CIOS algorithm to be presented in a simple, straightforward manner and core components identified and designed in parallel. It also allows the multiplication to be executed in a fixed number of clock cycles regardless of the operand length. Variations in execution time for the different multiplier inputs only arise due to the latency incurred in loading the operand lengths.

The CIOS multiplication architecture presented is 3 times faster than previously reported CIOS hardware implementations and 80 times faster than equivalent software implementations.

6. Acknowledgements

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

7. References

[1] C.K. Koç, T. Acar, B.S. Kaliski Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms", *IEEE Micro.*, Vol. 16, No. 3, pp 26-33, June 1996.

[2] P.L. Montgomery, "Modular Multiplication without Trial Division", *Mathematics of Computation*, Vol. 44, pp 512-521, April 1985.

[3] W. Diffie and M. E. Hellman, "New directions in cryptography", *IEEE Transactions on Information Theory*, IT-22:644-654, 1976.

[4] Digital Signature Standard (DSS), US Federal Information Processing Standards Publication 186-2, http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf

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

[6] T. ElGamal, "A Public-key cryptosystem and a Signature Scheme based on Discrete Logarithms", *IEEE Trans. on Information Theory*, vol.31, no.4, pp 469-472, '85

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

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

[9] S.E. Eldridge, C.D. Walter, " Hardware Implementation of Montgomery's Modular Multiplication Algorithm", *IEEE Trans. on Computers*, Vol 42, Issue 6, pp 693-699, 1993.

[10] T. Blum and C. Paar, "Montgomery Modular Exponentiation on Reconfigurable Hardware", 14th IEEE Symposium on Computer Arithmetic (ARITH-14), April 14-16, Adelaide, Australia, 1999.

[11] A. Tiountchik, E. Trichina, " RSA Acceleration with Field Programmable Gate Arrays", Proc. of 4th Australasian Conference on Information Security and Privacy, pp: 164 – 176, 1999

[12] G. Orlando and C. Paar, "A scalable GF(p) elliptic curve processor architecture for programmable hardware", Cryptographic Hardware and Embedded Systems, CHES 2001, May 14-16, France, 2001

[13] S. B. Ors, L. Batina, B. Preneel, J. Vandewalle, ``Hardware Implementation of an Elliptic Curve Processor over GF(p)", Proc. of IEEE 14th International Conference on Application-specific Systems, Architectures and Processors, The Hague, The Netherlands, June 24-26, 2003, pp. 433-443.

[14] C. McIvor, M. McLoone, J.V. McCanny, "FPGA Montgomery Multiplier Architectures - A Comparsion", IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), April 2004.

[15] M. McLoone, C. McIvor, J.V. McCanny, "Montgomery Modular Multiplication Architecture for Public Key Cryptosystems", IEEE Workshop on Signal Processing Systems (SiPS'04), October 2004.

[16] C.D. Walter, "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.

[17] C.D. Walter, "Montgomery Exponentiation Needs no Final Subtractions", *Electronics Letters*, 35(21):1831-1832, October 1999.

[18] C.D. Walter, "Precise Bounds for Montgomery Modular Multiplication and some Potentially Insecure RSA Moduli", Topics in Cryptology – CT-RSA 2002, LNCS 2271, pp 30-39, 2002.

[19] G. Hachez, J. Quisquater, "Montgomery Exponentiation with no Final Subtractions: Improved Results", Workshop on Cryptographic Hardware and Embedded Systems (CHES'00), LNCS 1965, pp 293-301, July 2000.

[20] A. Reinhold, "Big Number Calculator Applet", URL: http://world.std.com/~reinhold/BigNumCalc.html, Mar 2004.

[21] A. Satoh, K. Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor", *IEEE Transactions on Computers*, vol. 52, no. 4, April 2003.

[22] A. Tenca, C.K. Koç, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm", *IEEE Transactions on Computers*, vol. 52, no. 9, Sept 2003.

8. Appendices

Appendix 1: 256-bit Montgomery Modular Multiplication Test Vectors

```
TEST1:
------
n  = b6fb5f6da48f54fc63af7b4b3e9c3631
     cd781a526fb464a66bb3e127e74e2c25
r  = 10000000000000000000000000000000
     000000000000000000000000000000000
r^-1 = 308f7cea017ae7f66a3a9acb8076c329
     d2f3d2d547ce4bc5d0ee15b3b206c092
n' = 43f036f32f67e3e7ffb5f855b5f8a1e6
     438057aa826f82b65a6eb41c91a1f053
A  = e41be5bde54ea01c5fd8132dae3c50bd
     9f96c5af1324a68d08d978048f69bf76
B  = f9852cd21de6a57f70ba175bea2fffc2
     a40a26a7d424cf6f3cc8843f9135d1ed

MonMult =
     412093eaf12d0d4733be5966aa7ad29c
     9172b3877641ecda4a75afb9bb90ba87

TEST2:
------
n  = baa9614c0aff9d805f10ab09561f9bb9
     879bcf08bd25aa6955db00b696ac2173
r  = 10000000000000000000000000000000
     000000000000000000000000000000000
r^-1 = b658e95836328487a156c809d349aa63
     86f75bbe8e67f4c12831e83a7a13dc15
n' = fa15426cc24e25b51bb5a930e608a5b2
     58d9fbfb140d53ef7629b630338a1445
A  = a9bf5dff81ef3871f8206ecdd8b673cf
     b0b6737d5a983628babdca532e76f1ef
B  = c893cd7b64513b7a2652d8e4865dc4d0
     7af5bb3bd1ef10aa28197d81e622029f

MonMult =
     3389330a008175aef80264f09b38b8f2
     9dae79564599e0c2f3b0de7974a3a5cd

TEST3:
------
n  = a8c219a6b9410befd065464c689949dd
     21bed322a0851311b10b84f1ba87eccd
r  = 10000000000000000000000000000000
     000000000000000000000000000000000
r^-1 = 1103ea9926a61bb951bcdd82ece51e3c
     8e27c17d6d9e85edd20ba86cac2995ac
n' = 19cfc3e8846fb3901f1ab4cf9165c3ed
     85bef32703ab799f5da81a5083461ffb
A  = b071fcf379e962002c54d59662641609
     f870ddfb50601d5e2da14f6de24da602
B  = fa5cdcb31e308c75ddfb1f57812e4c1c
     f20dc45cabef6b25ea9b707e6e0bc147

MonMult =
     487448233b556c30a1820ef6bb44893d
     d67cd631adf7af48c86c667306f04cb2

TEST4:
------
n  = be0887bad260f54a528fb17779e03a7c
     7bea418f3fb4ced2a5cc8f841c7e28a5
r  = 10000000000000000000000000000000
     000000000000000000000000000000000
r^-1 = 47b253269cf8fb58acc6e3bb9dfe6b8a
     6b23f39eed8caef9d6a4a5b9411fd677
n' = 6095b38e3eec01f9696d7528270b5132
     ed9afb91e47d29293ec6c32ff04a80d3
A  = adb09a026dd44879f2105d553469fcaa
     036f4cf64450f1d7f36be42497926400
B  = 80b2a5ad7645210135d2908799f573ea
     b3a2dbe497f669ccc907f65b47e8c205

MonMult =
     349a9b139d4796e8d9077d162d175fcb
     b5f5721a17a35410625487eee5146778

TEST5:
------
n  = 880d640184375378e15e975b2b7550a9
     74dccab541a5addd9ce5ff01128a7005
r  = 10000000000000000000000000000000
     000000000000000000000000000000000
r^-1 = 47ea2a9c21a77a7d5ce488a8001c6def
     d26ea842d668414fe935b1e11f7fc6a7
n'=  87511b89e14d2f942a91553e67851f60
     899d0c2e797214f15c5e4d20f69f2333
A  = f8f89a1014078fe4c1c403074dd57d2e
     2af0ea5ff74467d58f3ff278749cd6e6
B  = ae435f91aad89bf845fba8b69cc0da52
     d6c5776714d63f6d3823f98b1712c2cc

MonMult =
     5e002a4b652c9b3cf0cbbb062fecbb96
     79e3c25ada22092d5461b760eb7ed05b
```

Appendix 2: 512-bit Montgomery Modular Multiplication Test Vectors

```
TEST1:
------
n  = c2b4f82ee693407aa871d788f2d87429
     8cefb19c4a396e671b6e9beef6ac8974
     7f6de818d6d7d6e19a71940f6b5084b5
     73fb0f65712f41da7a73891399729741
r  = 10000000000000000000000000000000
     00000000000000000000000000000000
     00000000000000000000000000000000
     000000000000000000000000000000000
r^-1 = 2928b25f1b3ef3e411c28c12e3100430
     44a0fecc0758e508a55bcce648948d74
     7c1f0f904312412c714de5922325954d
     69691f6a28b951b2e38e9b8f3417937d
n' = 361d9d24f6b5b6f9c27210efae12b86c
     03be9cfd7d894df308bf71a22db8646a
     598f6cdb94439f98cf29ab868dbe1e8b
     a6f10402402628da2b179a95ca2a073f
A  =  919425cebf5038fd6692568fc1801452
     9d527f6e2eae2ca7adc83a73e4398416
     ba139cbd0f1b1a7171f60e7b6bbd0864
     61460d8d3eebc517c5ca407d38ec7b31
B  = 86c345f271827b60c4df90e44ef9e8af
     ce5b6364efb558aee9ff7aaed0538a21
     4b5cfb536cc005243cf7838b5ac5b7ca
     c8f49ecdbc137f2635f71ce640a303bc
```

```
MonMult  =
     27a977d7874a1b7af9f51be9986184b3
     23e42ec47b8f00cac4af836d27cfb309
     dd68289ebb5d8ee5b20ad3ce65ed2928
     afc0cf06fc4f8cf80948b8c96f2e983a

TEST2:
------
n    = b73ae3e1bb5c756c2a550dc96a549057
       97040a8c93816c9271ad0c1d3115f219
       406d5c1013623046ecd13e7c41037540
       9dc360de2d9271a685617cc0121d8ec7
r    = 10000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       000000000000000000000000000000000
r^-1 = 3e771fb4c82429f429beb05b5557d6a2
       8bff23d8b1b64b6b0675de952e397a6d
       f3598b7b731867a6d5a8bf0b40b4b9ec
       419738705191c3de4ead7f11e0b20fb4
n'   = 5746030448ddc9592f80a861aa4e39ce
       ad9d30271c522442582d422fe7ca88f6
       23b1b35a209d2dc15c5ad34a2d737ce0
       a2867a1e405cd7e877272bf6a53b2d09
A    = d62097cebd3dce1096e056e7b418a257
       f16e31457e667a54dca66bdf6b533f32
       ef79c4e864566a7c2e06b58670f4a0a1
       39f3bca33081c61b48a46faeba2d9077
B    = 8a9b46f130a25ce8dc3e8a76ac00318b
       2bd864d7dd1eec4a8ea93b5114e55e23
       0f32cdf1c37a78c340499c8385e39abd
       767f2fbebbf717092ba80711bf6dbdfe

MonMult  =
     1bf407dc9042a4b653de6577bd83669e
     17e1baa3fb8c2ebc1e9863f93aa04b1b
     17cc153b2fd3489a31f3e55bd8123025
     770786053c3d1d2926cd88334925f642

TEST3:
------
n    =  cd86300a395dde2937c9610b5442d3c1
       41bf5069cf4caba2b3e0b07b7bf22e06
       32195360a6c5c8f69a11aadca2f831c2
       9c8343140a275f899193c739ee5d6119
r    = 10000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       000000000000000000000000000000000
r^-1 = 752c7d9f8c2e23840b08886a406270b2
       816ef7524e0a6addb499e4b7c21daa1b
       e5c1bbfa8fed685a622e2c300bda35d7
       6801c199a223e64fc38a577ffc908267
n'   = 91f37f19a479ceac680c96cd55829e1b
       c6a2dbd3c87ee14030b116a5cfd822f4
       dc70a54039a9b9b3b4ba9abb621f3276
       7375de3e16212d720689bb22181994d7
A    = f5e66bb5f19d9bab2025df6ed5a84b4c
       b36b61a54143fa5a09dd4cbc0570ad55
       a416594d23bf9438722dafb96824de9e
       3f9169670b55cba60290afb3df8686b9
B    = d615915ab93019a7420b38686b83f769
       c82ec1d85a5ab3183a44d3d1aff468c6
       bedb4e0d706bb8a97957958e1e932548
       c9c2c78c90763d1ead66e650d036e8ef

MonMult  =
     808275fa70ae35b3da376a28e53a8f48
     204ded9b33d12c3edb0fe8b1b800e396
     155b187bfa247f9701617040f5ebd740
     f9415bc59c2a08d2a4566a0c7ba7f8fd

TEST4:
------
n    = 959d0296f86a6fde340509ce489f2b39
       fa83ef317482d1893f748a4b486d8111
       d7a4e6640864823b2760a7a4b359dc86
       45de851a28e1eb3d997ddf40ea4fb5a3
r    = 10000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       000000000000000000000000000000000
r^-1 = 69bd494046651c64788f80aa3170c318
       9bc6fbf6b2c64f7f93d7f8146eab60fe
       4dc7e8267f8b535e8394d9a735aaedf2
       4bdc76f2cdb17a66ac9357643aef2106
n'   = b4eda6198560cd954d5a3165325dc05b
       c5296026fe094af34ee277f0f2baad49
       b06072c804bd7d0197a46e279e4f6b3a
       5eeaddc17041e1f53c65e5549840d9f5
A    = eb63a1f8c173dfb22fb06c830d93602a
       840c2ec3bc7a375a2ff80f68ad856b6e
       ef641c7f0d694685bed758239caad60b
       4d1a261a3985d04b6be492351a3fefb3
B    = a162d9a386bb7729d6ef07eb3dfadb46
       e5563cdf78f66a36b8b4c98b8cff4836
       6c61a2fa4a9cd4ba697c4a69cc93c2e6
       7dcce9d1f727a4816941050c62cf1c89

MonMult  =
     1c4ad14eea1ce6f6ab05993cf3c6908c
     df5d9a5b3db414869a5d2a99834dfe88
     b82e769100dcb2e0f2f99799e3a07a9a
     0e7fb1a6b87cf0e7afe9b34937fd9b3c

TEST5:
------
n    = 8e9956d1f4e4d2b109a47b95dfa622bd
       d87d034e902253a50c3b32983d408134
       4093ef192ddf1de888f4ebed14f03643
       8edc0a606afc322ae1c1e2312fc86745
r    = 10000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       000000000000000000000000000000000
r^-1 = 2d24c0ce40f463dac511f60c02a06079
       2cc95126ec1c0cb28db63df7f3420478
       e688b50b1eacd88d159cfcd0c16b4591
       7a57c6273e98a6196cc5ef296d1e1c49
n'   = 510b31cfc8c7c6b6b261a3d464537842
       eb2cf342f754f1c7b0054bf190376771
       cd1300f727caa41eb25ee18dcce25ac1
       c88586b238201d46fdb7f9e0d40eec73
A    = b2e2cf75f5e14942916eba99916df29c
       fa7bf7757ee2b077a755eb13c435dfaa
       d72b65c5b8321dd03c4d0b9c86b79aff
       55cef2e695851e2f5521c3e211ec3eb3
B    = afa5aa5469c75415e674ca307437cc75
       bb38b7a345f1c72fcadb7d9a8f93da80
       8cd75535afae77f5acd8b526262c0f46
       37eb42cdb0019b766482471a8f40c6d1

MonMult  =
     2bf7b842b3120b90612cc148bf03e087
     90824461f0f365513319ec707a0221a8
     dc7604d1228427d681ecba9b8ac93fc7
     a5f3eacab24dfb42850613bc7d20684
```