An Improved Unified Scalable Radix-2 Montgomery Multiplier

David Harris
Harvey Mudd College
David_Harris@hmc.edu

Ram Krishnamurthy, Mark Anders, Sanu Mathew, and Steven Hsu
Intel Circuits Research Laboratory
Ram.Krishnamurthy@intel.com

Abstract

This paper describes an improved version of the Tenca-Koç unified scalable radix-2 Montgomery multiplier with half the latency for small and moderate precision operands and half the queue memory requirement. Like the Tenca-Koç multiplier, this design is reconfigurable to accept any input precision in either GF(p) or GF(2^n) up to the size of the on-chip memory. An FPGA implementation can perform 1024-bit modular exponentiation in 16 ms using 5598 4-input lookup tables, making it the fastest unified scalable design yet reported.

1. Introduction

Multiplication in a finite field is essential to many encryption algorithms including RSA, Diffie-Hellman key exchange, the Digital Signature Algorithm, and elliptic curve cryptography [1]. The two common finite Galois fields are GF(2^n), used for elliptic curves, and GF(p), used for most other algorithms. Multiplication in a prime field GF(p) is performed modulo some prime p. Multiplication in a binary extension field GF(2^n) is performed modulo some irreducible polynomial f(x) of degree n. It is implemented identically to GF(p) except that carries are not propagated. Therefore addition reduces to the XOR operation.

Cryptographic computations are time-consuming because they operate on precisions of 256 to 2048 or more bits and require large numbers of multiplications to perform exponentiation. The Montgomery multiplication algorithm [2] is commonly used because it avoids division by the modulus. Many software and hardware implementations of Montgomery multiplication have been proposed. Software uses repeated multiplication and addition instructions [3, 4]. Radix-2 hardware designs operate in a word-serial fashion with addition as the basic operation [5]. Higher-radix designs use fewer cycles at the expense of requiring multiplications or memories containing precomputed multiples [6, 7, 8]. Hardware designs are said to be *scalable* if they can work on variable precision limited only by memory capacity. They are *unified* if they handle both GF(p) and GF(2^n) on the same array [9].

This paper proposes an improvement on the Tenca-Koç scalable unified radix-2 Montgomery multiplier [5] with half the latency for small and moderate-precision operands. The paper begins by reviewing Montgomery multiplication and the Tenca-Koç algorithm. It then describes how to left-shift input operands rather than right-shift results to avoid a bottleneck waiting for the most significant bit of each result word. The queue size is also cut in two by converting results to nonredundant format before storing them. Delay, area, and power results for a Verilog implementation synthesized to a Xilinx FPGA are discussed.

2. Montgomery Multiplication

We would like to compute $Z = X \times Y \bmod M$, where the operands have n bits of precision and M is an odd number in the range $2^{n-1} < M < 2^n$. In GF(p), M is the prime p. In GF(2^n), M is a binary representation of an irreproducible polynomial and carries are not propagated between columns in the multiplication. The modulo operation is expensive because it involves division.

Montgomery [2] observed that the divisions can be converted into simple shifts if multiplication is instead performed on so-called *Montgomery residues* (*M-residues*). The M-residue of an integer a ($0 = a < M$) is defined to be $\bar{a} = ar \bmod M$ where $r = 2^n$. For example, if r = 16 and M = 11, then we see $\bar{3} = 3 \times 16 \bmod 11 = 4$. There is an isomorphism between integers in this range and their Montgomery residues.

The modular multiplicative inverse b^{-1} of an integer b is that number such that $bb^{-1} \bmod M = 1$. For example, if $r = 16$ and $M = 11$, then $r^{-1} = 9$ because $rr^{-1} = 16\times 9 \bmod 11 = 1$. Montgomery multiplication (MM) of residues is defined as

$$\bar{z} = MM(\bar{x}, \bar{y}) = \overline{xy}r^{-1} \bmod M$$

Observe that

$$\bar{z} = \overline{xy}r^{-1} \bmod M = xryrr^{-1} \bmod M$$
$$= (xy)r \bmod M = zr \bmod M$$

so the Montgomery product of two Montgomery residues is the Montgomery residue of the product of the two corresponding integers.

It may not be immediately obvious that multiplying by r^{-1} mod M is an easier problem than simply multiplying mod M. However, there is a simple Radix-2 algorithm for doing so, given in Figure 1. This algorithm computes $S = \mathrm{MM}(X, Y) = XYr^{-1} \bmod M$. It uses n steps, as in a word-serial multiplication algorithm. On step i, it adds the Y to the running sum if the ith bit of X (x_i) is true. Also on each step, it divides by two. If the running sum was odd, it first adds M so the result can be divided by two without loss of information. This is permissible because adding M mod M does not change the result. After n steps of dividing by two, the algorithm has divided by $r = 2^n$. The algorithm might produce a result as large as $2M$-1, so it concludes by subtracting M if necessary to restore the result to the legal range. The final subtraction can be avoided for repetitive multiplications used in exponentiation [10], so we will ignore it through the rest of this paper. In summary, the algorithm computes the Montgomery product using only $2n$ n-bit additions and n one-bit right shifts, which is substantially simpler than conventional modular multiplication with division.

Conversion to and from M-residues is accomplished using Montgomery multiplication:

$$\bar{x} = MM(x, r^2) = xr^2r^{-1} \bmod M = xr \bmod M$$
$$x = MM(\bar{x}, 1) = \bar{x}1r^{-1} \bmod M = xr1r^{-1} \bmod M = x$$

Note that r^2 mod M should be precomputed to make this efficient; this is easy for cryptographic systems that change M infrequently.

Now a long sequence of multiplications, like those required in exponentiation, can be performed by converting the operands to M-residues, performing Montgomery multiplications, and converting the result back to an integer.

Z is commonly stored in carry-save redundant format for fast addition. In a unified design, we use a modified carry-save adder that forces the carry to zero when operating in GF(2^n) [9]. Define a bit cell to perform addition for one bit of the partial product of x_i and y_j. In a typical design, a bit cell would contain two full adders, two AND gates, and some registers, as

```
Z = 0
for i = 0 to n-1
    Z = Z + $x_i \times$ Y
    if Z is odd then Z = Z + M
    Z = Z/2
if Z = M then Z = Z - M
```

Figure 1. Simple radix-2 Montgomery multiplication algorithm

```
Z = 0
for i = 0 to n-1
    $C_a = C_b = 0$
    for j = 0 to e
        $(C_a, Z^j) = C_a + x_i \times Y^j + Z^j$
        if (j == 0) odd = $z_0$
        if odd then
            $(C_b, Z^j) = C_b + M^j + Z^j$
        $Z^{j-1} = \left(Z_0^j, Z_{w-1:1}^{j-1}\right)$
```

Figure 2. Tenca-Koç multiple word radix-2 Montgomery multiplication algorithm

will be shown later in Figures 5 and 8. Assume the bit cell requires a full cycle to operate. We will compare the number of bit cells and the number of cycles required by various algorithms. The Radix-2 Montgomery Multiplication Algorithm uses n bit cells and n cycles so its area-delay product is n^2.

3. The Tenca-Koç Algorithm

The algorithm of Figure 1 requires an adder with a precision of n so it is not scalable to different values of n. Tenca and Koç present a multiple word radix-2 Montgomery multiplication algorithm that uses hardware with a fixed word width w [5]. We will review this algorithm and its performance before describing how to improve it. For $n = ew$, the hardware is reused e times. Let $M = (M^{(e-1)}, \ldots, M^1, M^0)$, $Y = (Y^{(e-1)}, \ldots, Y^1, Y^0)$, $Z = (Z^{(e-1)}, \ldots, Z^1, Z^0)$, $X = (x_{n-1}, \ldots, x_1, x_0)$, where words are indicated with superscripts and bits with subscripts. M, Y, and Z are zero-extended to e+1 words to avoid overflow.

The algorithm is given in Figure 2. The outer loop iterates over all n bits of X. The inner loop iterates over e+1 words of M, Y, and Z. Z is odd if the least significant bit is 1. Z is right-shifted by one bit at each step to divide by two. Note that the least significant bit of Z^j must be computed before it can be right-shifted into the most significant position of Z^{j-1} on the jth step of the inner loop. This is a critical limitation of the algorithm.

Observe that the only dependency in the outer loop is that Z^j for iteration i must be known before Z^j for iteration i+1 can be computed. A hardware implementation of the Tenca-Koç algorithm unrolls the outer loop to use a pipelined *kernel* of p w-bit processing elements (PEs). Each PE contains two w-bit adders and two banks of w AND gates to add $x_i \times Y^j$ and $odd \times M^j$ to Z^j and registers to hold the results. A PE must wait two cycles to kick off after its predecessor until Z^0 is available because Z^1 must first be computed and shifted. In one kernel cycle, p bits of x are processed. Hence $k = n/p$ kernel cycles are required to do the entire computation. We will assume for simplicity that n is divisible by p and w; this is usually true because all three parameters are typically powers of 2.

Figure 3 shows a block diagram of the scalable Montgomery multiplier. The kernel contains p w-bit PEs for a total of wp bit cells. Z is stored in carry-save redundant form. If PE p completes Z^0 before PE1 has finished Z^{e-1}, the result must be queued until PE1 becomes available again. The design in [5] queues the results in redundant form, requiring $2w$ bits per entry. For large n the queue consumes significant area, so we propose converting Z to nonredundant form to save half the queue space, as shown in Figure 4. On the first cycle, Z is initialized to 0. When no queuing is needed, the carry-save redundant Z' is bypassed directly to avoid the latency of the carry-propagate adder. The nonredundant Z result is also an output of the system.

Figure 5 shows a design of the processing element. It uses $2w$ carry-save adders and 2-input AND gates, a 2:1 multiplexer, and $4w$+5 register bits. The odd parity of the least-significant word of Z is stored to determine whether M should be added. On each cycle, Z is right-shifted so that the most significant bit of the previous word becomes the least significant bit of the next word.

Figure 6 shows a pipeline diagram of the Tenca-Koç architecture indicating which bits are processed on each cycle. There are two dependencies for PE1 to begin a kernel cycle, indicated by the gray arrows: PE1 must be finished with the previous cycle, and the $Z_{w-1:0}$ result of the previous kernel cycle must be ready at PE p. We assume that there is a two-cycle latency to bypass the result from PE p to account for the FIFO and routing. The computation time in clock cycles is

$$k(e+1)+2(p-1) \quad e>2p-1 \quad \text{(Case I)}$$

$$k(2p+1)+e-2 \quad e\le 2p-1 \quad \text{(Case II)}$$

The first case corresponds to a large number of words. Each kernel cycle requires e+1 clock cycles for the first PE to handle one bit of X. The output of PE p must be queued until the first PE is ready again.

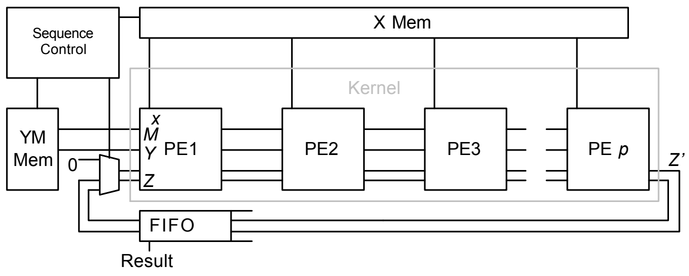

Figure 3. Scalable Montgomery multiplier architecture

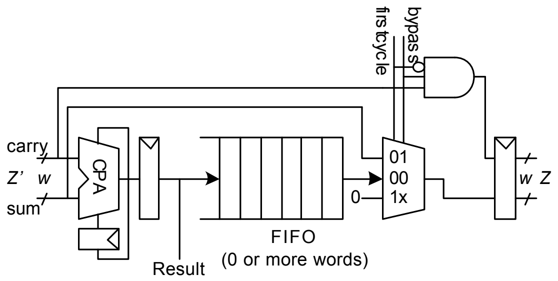

Figure 4. Improved queue design

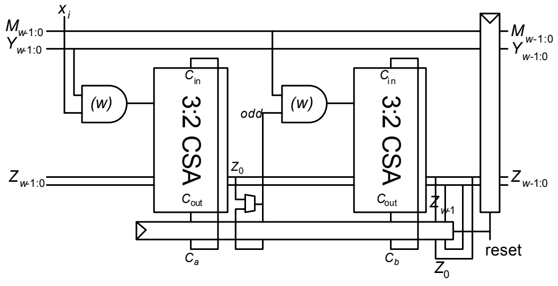

Figure 5. PE schematic

There are k kernel cycles. Finally, $2(p-1)$ cycles are required for the subsequent PEs to complete on the last kernel cycle.

The second case corresponds to lower precision where a small number of words are necessary. Each kernel cycle takes $2p$ clock cycles before final PE produces its first word and one more cycle to bypass the result back. k kernel cycles are needed. Finally e-2 cycles are required to obtain the more significant words at the end of the final kernel cycle.

In other words, if there are relatively few small PEs, the latency is approximately $ke = n^2/wp$, so the area-delay product is n^2; the design is efficient. On the other hand, if there are many PEs or the PEs are too wide (wp exceeding approximately $n/2$), the latency is approximately $2kp = 2n$ and the area-delay product is $2nwp > n^2$. When large amounts of hardware are available, the minimum latency is a factor of two worse than the simple radix-2 design. In the next section, we will see how to improve the latency.

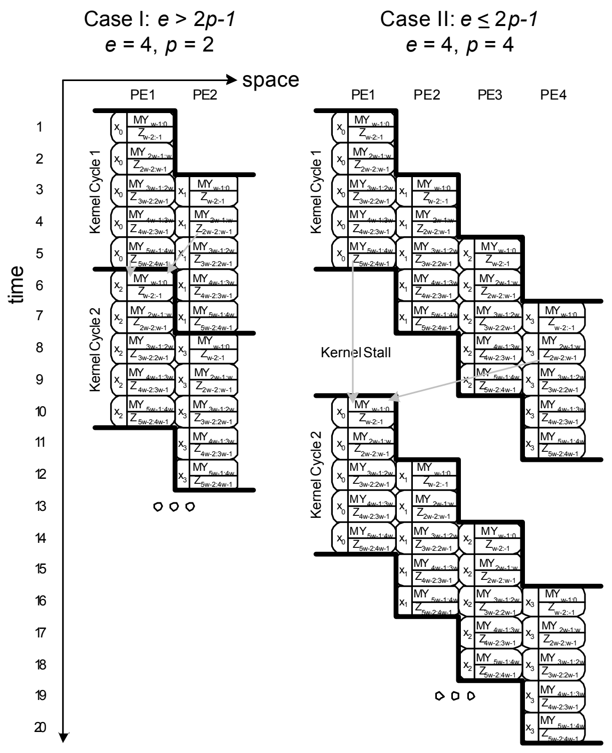

Figure 6. Tenca-Koç pipeline diagram

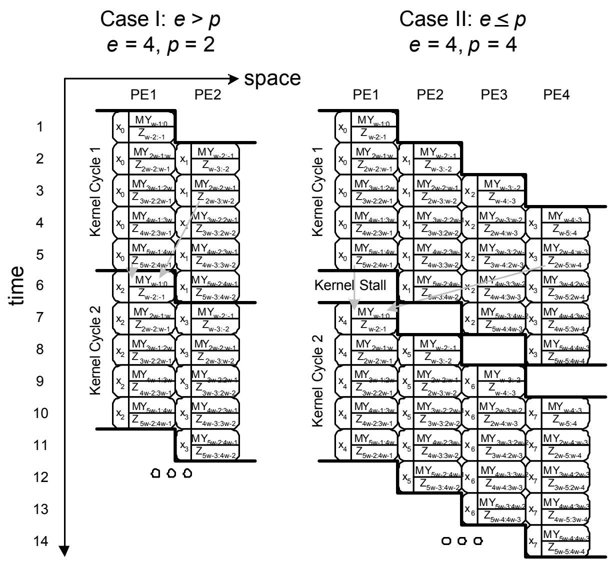

Figure 7. Improved pipeline diagram

4. Improved Scalable Architecture

The fundamental problem with the Tenca-Koç architecture is the two-cycle latency from one PE to the next caused by waiting to right shift Z_0^1 into Z_{w-1}^0 before processing Z^0. Instead, we propose to begin operating on the least significant bits of Z as soon as they are available. Rather than right-shifting *Z*, we left-shift *Y* and *M* at each step. Now, each PE can begin immediately after its predecessor. At the end of each kernel cycle, *p/w* additional cycles are necessary to complete the most significant words rather than the single cycle previously used to handle overflow. For convenience, we assume that *p* is a multiple of *w*.

Figure 7 shows a pipeline diagram for the improved scalable Montgomery multiplier. The bits with negative indices are insignificant trailing zeros. Each of the first *k*-1 kernel cycles takes max(*e, p)* + *p/w* + 1 clock cycles before the next can begin (cases I and II, respectively). The final kernel cycle takes *e* + *p* + *p*/*w* cycles to produce the last word. In case I, the system is still limited by the time for PE1 to complete and the latency is only slightly better. In case II, the system is limited by the time for PE *p* to complete and speeds up significantly. Substituting *kp*/*w* = *e*, the latencies simplify to

$$(k+1)(e+1)+p-2 \quad e>p \quad \text{(Case I)}$$

$$k(p+1)+2e-1 \quad e\leq p \quad \text{(Case II)}$$

Again, for Case I the latency simplifies to approximate n^2/wp so the area-delay product is n^2. In case II, the latency is approximately *n* so the area-delay product is *nwp*, which is ideal. Both the latency and area-delay product improve by a factor of two.

The overall architecture is unchanged from Figure 3; the differences lie in the shifting done by the PEs and the timing from the sequence controller. Figure 8 shows a schematic of the improved *w*-bit processing element. It contains the same critical path and amount of hardware as the Tenca-Koç design.

5. Results

Table 1 lists the cycle count for various choices of *w*, *p*, and *n* for the Tenca-Koç and improved multipliers. For operand precisions *n* up to the number of bit cells *wp*, the new design is about twice

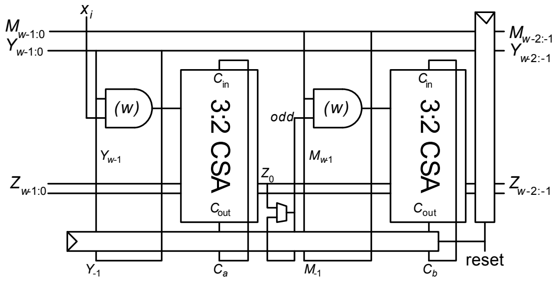

Figure 8. Improved PE schematic

as fast. For larger operands, the two designs are comparable. Selecting $w = p$ gives a good balance for latency, but large w leads to a long CPA delay in the result adder (although this delay could be pipelined). Hence, we evaluate designs with $w = 16$.

The improved design was coded in Verilog parameterized by p and w. It was verified against a C reference based on the simple radix-2 algorithm of Figure 1. It was synthesized using Synplify Pro targeting a Xilinx Virtex-II speed grade 6 XC2V250-6 FPGA [11]. The results were not verified on an actual chip. Each PE has $4w + 6$ flip-flops and $5w + 1$ 4-input lookup tables (LUTs). The sequencer is designed with 16-bit counters to handle e and k up to 2^{15}. The sequencer accounts for another 53 flip-flops and 95 LUTs. The final CPA and queue control logic contains 64 flip-flops and 90 LUTs. RAMs for the input and output operands and queue depend on the operand precision and are not considered, but would add approximately $4n$ bits of storage for X, Y, M, and Z, and n-wp bits for the queue.

Putting this all together, the complete 16×16 Montgomery multiplier contains 1233 flip-flops and 1514 LUTs including buffers and taking advantage of optimization across modules. A 64×16 Montgomery multiplier contains 4466 flip-flops and 5598 LUTs. Both designs operate at a worst-case 6.9 ns clock period, limited by the 16-bit counter in the sequencer.

n-bit modular exponentiation requires at most $2n + 2$ modular multiplications including the conversion to and from M-residues. Table 2 compares the time for 256-bit and 1024-bit exponentiation using various recent hardware and software implementations. For reference, a CLB in a Xilinx 4000XV-series chip contains 32 bits of RAM or two flip-flops and two LUTs.

The improved scalable design is significantly faster than the Tenca-Koç design because of both the architecture and the faster clock rate. It appears to be comparable in performance and to use less hardware than the nonscalable radix-2 systolic design of Blum [12]. However, Blum's radix-16 nonscalable design [6] is faster because it processes four times as many bits of X per cycle per PE using 1.5x as much hardware. This suggests that it would be interesting to further investigate higher-radix scalable designs, although [13, 14] did not achieve as dramatic an area-delay improvement.

Mukaida [7] uses an entirely different approach based on a large multiplier for GF(p). The approach scales to multiple word lengths by reusing the multiplier, but does not support GF(2^n). The paper does not report times for modular exponentiation, but it appears to be extremely fast at generating digital signatures.

A 16×16 Montgomery multiplier has also been simulated in a 90 nm process using $V_{DD} = 1.2$ V [15]. Static circuits with high V_t devices are used exclusively. The clock frequency of 2.4 GHz is limited by the critical path through the 16-bit CPA. If this were pipelined, the limiting path through the bit cell operates at 3.2 GHz. The kernel has an area of 354 μm × 146 μm based on a trial layout. The complete unit draws 69 mW on a random test case, of which 23 mW is leakage power.

6. Conclusion

This paper has described an improvement on the Tenca-Koç unified scalable radix-2 Montgomery multiplier. The design left-shifts the input operands rather than right-shifting the result to reduce the latency by nearly a factor of two for operand precisions up to the size of the array. It also converts intermediate results to nonredundant form to cut the queue memory requirement in half. The proposed multiplier has been synthesized for a Xilinx Virtex-II FPGA. It is the fastest scalable unified design reported in the literature.

This work might be extended to higher radix multipliers. It would also be useful to better understand the tradeoffs between architectures with a large number of bit cells and those with a large conventional multiplier array using reduction steps from [3].

Acknowledgments

This work was sponsored by and done at the Intel Circuit Research Lab, Hillsboro, OR.

Table 1. Montgomery multiplier latencies (clock cycles)

Bit cells	w	p	n	e	k	Tenca-Koç	This work	% Improvement
256	8	32	256	32	8	550	327	41
			512	64	16	1102	1135	-3
			1024	128	32	4190	4287	-2
			2048	256	64	16510	16735	-1
	16	16	256	16	16	542	303	44
			512	32	32	1086	1103	-2
			1024	64	64	4190	4239	-1
			2048	128	128	16542	16655	0
1024	8	128	256	32	2	544	321	41
			512	64	4	1090	643	41
			1024	128	8	2182	1287	41
			2048	256	16	4366	4495	-3
	16	64	256	16	4	530	291	45
			512	32	8	1062	583	45
			1024	64	16	2126	1167	45
			2048	128	32	4254	4319	-2
	32	32	256	8	8	526	279	47
			512	16	16	1054	559	47
			1024	32	32	2110	1119	47
			2048	64	64	4222	4255	-1

Table 2. Comparison of modular exponentiation times

Description	Technology	Hardware	Clock Speed	Scalable / Unified	Source	256-bit time (ms)	1024-bit time (ms)
Improved 16 PEs x 16 bits	Xilinx Virtex Pro	1514 LUTs + ~5n bits RAM	144 MHz	Yes / Yes	This work	1.1	59
Improved 64 PEs x 16 bits	Xilinx Virtex Pro	5598 LUTs + ~5n bits RAM	144 MHz	Yes / Yes	This work	1.0	16
Tenca-Koç 40 PEs x 8 bits	0.5 µm CMOS synthesized	28 Kgates (kernel only)	80 MHz	Yes / Yes	[5]	3.8	88.2
Scalable radix 8 16 PEs x 16 bits	0.5 µm CMOS synthesized	28 Kgates	64 MHz	Yes / No	[14]	1.6	46
Scalable high radix	0.5 µm CMOS estimated	33 Kgates (estimated)	44 MHz	Yes / Yes	[13]	1.8	82
32-bit ARM processor		n/a	80 MHz	Yes / No	[5, 3]	21.8	117
Systolic Radix-2 256-bit	Xilinx XC40150XV-08	1307 CLBs	57 MHz	No / No	[12]	2.4	n/a
Systolic Radix-2 1024-bit	Xilinx XC40150XV-08	4865 CLBs	52 MHz	No / No	[12]	n/a	40.0
Systolic Radix-16 256-bit	Xilinx XC40150XV-08	1818 CLBs	47 MHz	No / No	[6]	0.73	n/a
Systolic Radix-16 1024-bit	Xilinx XC40250XV-09	6633 CLBs	45 MHz	No / No	[6]	n/a	12.0

References

[1] B. Schneier, *Applied Cryptography*, New York: John Wiley, 1996.

[2] P. Montgomery, "Modular multiplication without trial division," *Math. of Computation*, vol. 44, no. 170, pp. 519-521, April 1985.

[3] Ç. Koç, T. Acar, and B. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, June 1996, pp. 26-33.

[4] B. Phillips and N. Burgess, "Implementing 1024-bit RSA exponentiation on a 32-bit processor core," *Proc. IEEE Intl. Conf. Application-Specific Systems, Architectures, and Processors*, pp. 127-137, July 2000.

[5] A. Tenca and Ç. Koç, "A scalable architecture for modular multiplication based on Montgomery's algorithm," *IEEE Trans. Computers*, vol. 52, no. 9, Sept. 2003, pp. 1215-1221.

[6] T. Blum and C. Paar, "High-radix Montgomery multiplication on reconfigurable hardware," *IEEE Trans. Computers*, vol. 50, no. 7, July 2001, pp. 759-764.

[7] K. Mukaida, M. Takenaka, N. Torii, and S. Masui, "Design of high-speed and area-efficient Montgomery modular multiplier for RSA algorithm," *IEEE Symp. VLSI Circuits*, pp. 320-323, 2004.

[8] A. Tenca and L. Tawalbeh, "An efficient and scalable radix-4 modular multiplier design using recoding techniques," *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 1445-1450, 2003.

[9] A. Savas, A. Tenca, M. Çiftçibasi, and Ç. Koç, "Multiplier architectures for GF(p) and GF(2^n)," *IEE Proc. Comput. Digit. Techn.*, vol. 151, no. 2, March 2004, pp. 147-160.

[10] G. Hachez and J. Quisquater, "Montgomery exponentiation with no final subtractions: improved results," *Lecture Notes in Computer Science*, C. Koç and C. Paar, eds., vol. 1965, pp. 293-301, 2000.

[11] Xilinx, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs Datasheet*, June 30, 2004, www.xilinx.com.

[12] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," *Proc. 14th Symp. Computer Arithmetic*, pp. 70-77, 1999.

[13] G. Gaubatz, *Versatile Montgomery multiplier architectures*, M. S. Thesis, Worcester Polytechnic Institute, Dept. of Electrical Engineering, April 2002.

[14] G. Todorov, *ASIC design, implementation and analysis of a scalable high-radix Montgomery multiplier*, M. S. Thesis, Oregon State University, June 2001.

[15] S. Thompson, *et al.*, "A 90 nm logic technology featuring 50 nm strained silicon channel transistors, 7 layers of Cu interconnects, low k ILD, and 1 μm^2 SRAM cell," *Proc. Intl. Electron Device Meeting*, pp. 61-64, Dec. 2002.