# Parallel High-Radix Montgomery Multipliers

Philip Amberg, Nathaniel Pinckney, and David Money Harris
Harvey Mudd College
301 Platt Blvd. Claremont, CA 91711
{pamberg, npinckney, David_Harris}@hmc.edu

*Abstract*— **This paper describes the algorithm and design tradeoffs for multiple hardware implementations of parallel high-radix scalable Montgomery multipliers. Hardware implementations of Montgomery multipliers require choosing a radix, shift direction, and whether to use Booth encoding. Presented are processing element designs exploring combinations of radices 2, 4, and 8, right vs. left shifting, and Booth encoding. A radix-4, left-shifting, non-Booth encoded design performs a 1024-bit modular exponentiation in 9.4 ms using 4997 LUTs and 4051 REGs and appears to maximize performance/hardware in an FPGA implementation. A Booth encoded version of the above multiplier performs a 1024-bit modular exponentiation in 13 ms using 4852 LUTs and 2887 REGs. This design may be beneficial for systems constrained by the cycle time of other elements because the design minimizes hardware usage and requires no precomputed multiples. The radix-8, right-shifting, Booth-encoded design offers no performance/hardware advantage over a comparable radix-4 design.**

## I. Introduction

Public key encryption schemes, including RSA, use modular exponentiation of large numbers to encrypt data. This is secure because factoring large numbers is computationally intensive and becomes intractable for very large numbers. Modular exponentiation of large numbers is slow because of repeated modular multiplications with division steps to calculate the remainder. Montgomery multipliers [1] are useful because they replace the costly division with a simple right shift. Hence, they can increase the speed of encryption systems.

Older Montgomery multipliers are hard-wired to support a particular operand length, $n$. Scalable Montgomery multipliers reuse $w$-bit processing elements (PEs) many times to handle the entire $n$-bit operands, making them suitable to arbitrary-length operands [2]. Previous scalable Montgomery multiplier designs include radix-2 [2, 3], radix-4 [4, 5, 6, 7], radix-8 [8], radix-16 [9], and very high radix [10, 11]. A scalable radix-$2^v$ design processes $v$ bits of the multiplier and $w$ bits of the multiplicand per step.

The critical path through a PE can be shortened by reordering the steps of the Montgomery multiplication algorithm, which parallelizes multiplications within the PE and simplifies quotient determination [12, 10].

In developing a parallel high radix Montgomery multiplier, a designer has three primary design choices: radix, left or right shifting, and use of pre-computed multiples or Booth encoding. These choices have direct impact on the hardware footprint and exponentiation time. This paper discusses these design options and evaluates the tradeoffs in terms of hardware cost and exponentiation time. This paper also proposes a novel design for a parallel radix-8, right-shifting, Booth-encoded Montgomery multiplier.

## II. Montgomery Multiplication

This section summarizes Montgomery multiplication, based on the treatment from [5, 6]. Montgomery multiplication is defined as

$$Z = (XYR^{-1}) \bmod M$$

where

- $X$: $n$-bit multiplier
- $Y$: $n$-bit multiplicand
- $M$: $n$-bit odd modulus, typically prime
- $R$: $2^n$
- $R^{-1}$: modular multiplicative inverse of $R$ $(RR^{-1}) \bmod M = 1$.

The steps of Montgomery multiplication are shown in Fig. 1. Because $R=2^n$, dividing by $R$ is equivalent to shifting right by $n$ bits. $Q$ has the property that the lower $n$ bits of $[Z + Q \times M]$ are 0, so no information is lost in this step.

Note that we can skip the normalization step for successive Montgomery multiplications because if $R > 4M$ and $X, Y < 2M$ then $Z < 2M$ [14]. To do this we increased the size of the operands to $n_1 = n + 1$ bits and let $R = 2^{n_2}$, where $n_2 = n + 2$.

**Multiply:** $Z = X \times Y$

**Reduce:** $Q = Z \times M' \bmod R$

$Z = [Z + Q \times M] / R$

**Normalize:** if $Z \geq M$ then $Z = Z - M$

Fig.1: Montgomery multiplication algorithm

*A. Parallel Radix-$2^v$ Scalable Algorithm*

The parallel radix-$2^v$ Booth and non-Booth algorithms are shown in Figs. 2 and 3. These are equivalent to previous algorithms [5, 6, 7, 10] extended to $v$ bits. The variables are defined below.

$n_1$: $n + v + 1$
$n_2$: $n + v + 2$ (or larger [7])
$M$: $n$-bit odd modulus
$M'$: $n_2$-bit integer satisfying $(-MM') \bmod 2^{n_2} = 1$
$\hat{M}$: $n$-bit integer $((M' \bmod 2^v) \times M + 1) / 2^v$
$Y$: $n_1$-bit multiplicand
$X$: $n_1$-bit multiplier
$C$: 2-bit carry
$w$: scalable inner word length
$f$: outer loop length $\lceil n_2 / v \rceil$
$e$: inner loop length $\lceil n_1 / w \rceil$

$Z = 0$
for $i = 0$ to $f - 1$
  $Q^i = Z^0 \bmod 2^v$
  $Q^{'i} = \text{Booth}(Q^i, Q_v^{i-1})$
  $X^{'i} = \text{Booth}(X^i, X_v^{i-1})$
  $C = 0$
  for $j = 0$ to $e - 1$
    $(C, Z^j) = (Z_{v-1:0}^{j+1}, Z_{w-1:v}^{j}) + C + Q^{'i} \times \hat{M}^j + X^{'i} \times Y^j$
If $Q_v^{f-1} == 1$
  $C = 0$
  for $j = 0$ to $e - 1$
    $(C, Z^j) = Z^j + C + 2^v \hat{M}^j$

Fig. 2: Parallelized, radix-$2^v$ scalable Booth algorithm

$Z = 0$
for $i = 0$ to $f - 1$
  $Q^i = Z^0 \bmod 2^v$
  $C = 0$
  for $j = 0$ to $e - 1$
    $(C, Z^j) = (Z_{v-1:0}^{j+1}, Z_{w-1:v}^{j}) + C + Q^i \times \hat{M}^j + X^i \times Y^j$

Fig. 3: Parallelized, radix-$2^v$ scalable non-Booth algorithm

The algorithms are scalable because they iterate over words of the operands using fixed-sized PEs. The superscripts denote $v$-bit words for $X$ and $w$-bit words for $Y$, $\hat{M}$, and $Z$. There are $e = \lceil n_1 / w \rceil$ $w$-bit words of $Y$, $\hat{M}$, and $Z$, and $f = \lceil n_2 / v \rceil$ $v$-bit words of $X$ in a radix-$2^v$ design with $w$-bit PEs.

Encoding is indicated by the *Booth()* function. A $2^v \hat{M}$ multiple must be added to the result at the end of the algorithm, if the last Booth encoding for $Q \times \hat{M}$ was negative. This is not needed for $X \times Y$ because $X < 2M$ and $X_{v-1}^{f-1}$ is the $n_2 = n + v + 2$ bit of $X$, which will always be zero.

## III. Hardware Implementation

As Tenca proposed [2], the scalable Montgomery multiplier is built from a systolic array of $p$ processing elements (PEs). The architecture includes memories for $X$, $Y$, and $\hat{M}$, a FIFO to store partial products, and a sequence controller. The FIFO holds results of the last PE until the first PE has completed processing the current operand. For Booth encoded designs, the last $2^v \hat{M}$ multiple is conditionally added in the FIFO before storing the result.

The original Montgomery multiplication algorithm involves three dependent multiplications. Orup showed that the algorithm can be sped up by reordering steps and doing a precomputation, which eliminates one of the multiplications and allows the other two to occur in parallel [12]. Tenca's designs [8] do not use Orup's parallelization method. These designs use a lookup table to determine $Q$ from the previous PE's result, $X$, and $Y$. The three designs presented in this paper both parallelize the multiplications in each PE and simplify quotient determination. Orup's method prescales $X$ by $2^v$, by setting $Q^0 = 0$ and setting subsequent $Q$'s to be the $v$ LSBs of the result, simplifying the hardware implementation.

Before attempting the design of the specific multipliers in [6, 7] and this paper, the datapaths were evaluated to get an initial estimate of hardware usage, shown in Fig. 4. The radix-4, non-Booth encoded architecture was considered the baseline. When adding Booth encoding to this design, no precomputed hard-multiples are required, saving registers, and the additional hardware required is a Booth encoder and an additional input on the product selection multiplexer. These were deemed to be acceptable tradeoffs for study. The next designs considered were radix-8, non-Booth and Booth encoded designs. The radix-8 non-Booth design was not pursued, because we think it would not be competitive due to the extra flip-flops needed to store all four hard multiples. However, the Booth-encoded design would require the same amount of registers to store the multiples as the radix-4, non-Booth encoded design, making this design also worthwhile to study. A radix-16 design was not considered because there did not appear to be any additional advantage scaling to radix $v = 4$.

*A. Radix*

As radix increases, the number of bits consumed per step increases, resulting in decreased cycle count at the expense of more hardware. The complexity of multipliers within each PE also increases with radix. The smallest radix design is a radix-2 design, which consumes one bit of the multiplier $X$ in each processing element. Generating products for this processing element requires only an AND2 gate, since the

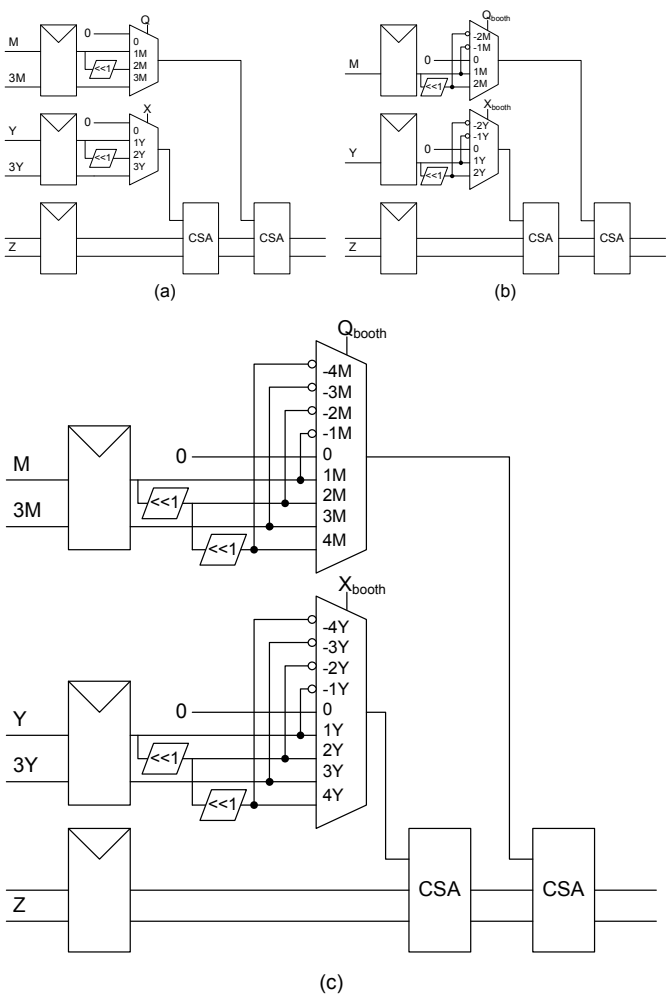


Fig. 4: Parallel High Radix Datapaths. (a) Radix-4 non-Booth. (b) Radix-4 Booth. (c) Radix-8 Booth

multiples can only be $0Y$ or $1Y$. Radix-4 designs process two bits of the multiplier in each processing element, leading to four possible multiples. For non-Booth designs, this changes the multiplier from an AND2 gate to a 4:1 MUX, which chooses between the products. The hard multiple $3Y$ now has to be pre-computed and stored. In general, non-Booth encoded radix-$2^v$ designs will have $2^v$ possible multiples, requiring a $2^v$:1 MUX to choose products and registers to store $2^{v-1}$ hard multiples. The choice of radix also modifies quotient determination. As shown in Fig. 2, the quotient $Q$ is found by taking the bottom $v$ bits of the least significant word from each PE. To generate the selector for $Q \times \hat{M}$ in the next PE, a $v$-bit CPA is needed to convert the results from redundant to non-redundant form.

*B. Booth Encoding*

Booth encoding is a technique for avoiding hard multiples, such as $3Y$, by using negative partial products [15]. Consider a multiplication $X \times Y$. For a radix-$2^2$ multiplication algorithm, 2-bits of $X$ are consumed each step. Therefore the possible multiples of $Y$ are 0, $Y$, $2Y$, or $3Y$. Note that in radix-$2^2$ multiplication, a multiple of $Y$ becomes $4Y$ because the next two bits of $X$ are being processed. Observe that

$$3Y = 4Y - Y$$

so the $3Y$ multiple can be generated with a $-Y$ in the current cycle and a $4Y$ in the next step. In general, Booth encoding will recode the possible multiples from $\{0, Y, \ldots, (2^v\text{-}1)Y\}$ to $\{-2^{v-1}Y, \ldots, 0, \ldots, 2^{v-1}Y\}$.

For a radix-4 design, the multiple set is recoded from $\{0, 1Y, 2Y, 3Y\}$ to $\{-2Y, -1Y, 0, 1Y, 2Y\}$. Therefore with Booth encoding, it no longer becomes necessary to precompute and store the $3Y$ multiple. For right-shifting designs, this reduces the required registers for storing multiples from $8w$ to $4w$. The cost is additional hardware for the Booth encoding and moving from a 4:1 MUX to a 5:1 MUX to choose the partial product. Additionally, special hardware is required to inject bits to generate the 2s complement form of negative multiples. For left-shifting designs, Booth encoding reduces the required registers for storing multiples from $4w$ to $2w$. However, the Booth encoder and its support logic increase the critical path, leading to a 29% reduction in clock speed.

For a radix-8 design, the multiple set is recoded from $\{0, 1Y, 2Y, 3Y, 5Y, 6Y, 7Y\}$ to $\{-4Y, -3Y, -2Y, -1Y, 0, 1Y, 2Y, 3Y, 4Y\}$. Without Booth encoding, it would be necessary to precompute and store the $3Y$, $5Y$, and $7Y$ multiples. Booth encoding reduces this requirement so only the $3Y$ multiple must be stored. For a right-shifting design, this reduces the required registers for storing multiples from $16w$ to $8w$. The cost is additional hardware for the Booth encoding and increasing the product selector size from an 8:1 MUX to a 9:1 MUX. Radix-8 designs could in principle achieve similar clock speed as radix-4 designs because the only critical path change is MUX5 to MUX9.

Higher radices become difficult because an increasing number of hard multiples must be stored, requiring additional registers for each multiple or each multiple must be dynamically generated within each PE. Generating the multiples within each PE would require large multiplexers to choose between the appropriate partial products and using two 4:2 CSAs rather than two 3:2 CSAs to add them together [8, 9].

*C. Left vs. Right Shifting*

Traditional Montgomery multipliers shift the result $Z$ right $v$ bits between each processing element. This leads to a cycle latency $l = 2$ between PEs, because the bottom bits of the next word must be computed before they are shifted into the top bits of the previous word. For this case, the Montgomery multiplication times [7] are

$$T_1 = k(e)T_c \quad \text{for } e \geq lp + b$$
$$T_2 = k(lp + b)T_c \quad \text{for } e < lp + b$$

Instead of right-shifting the result, the operands $Y$ and $M$ can be left shifted $v$ bits, reducing the latency to $l = 1$ cycle

between PEs. The total Montgomery multiplication time [5, 6, 7] is then

$$T_1 = k(e+1)T_c \text{ for } e \geq lp + vp/w + b$$
$$T_2 = k(lp + vp/w + b)T_c \text{ for } e < lp + vp/w + b$$

For large multiplications, left-shifting results in a reduction of cycle count.

An additional benefit of left-shifting is the reduction of the number of pipeline registers. In a right-shifting design, each multiple requires $2w$ registers and the result requires $4w$ registers due to the two cycle latency. When the latency is reduced to one cycle, each multiple only requires $w$ registers and only $2w$ registers are needed to store the result. Therefore left-shifting reduces the number of pipeline registers by approximately half.

However, left shifting adds additional design complexity and places strict constraints on word length, and number of processing elements in the pipeline. Left-shifting the operands changes the effective least significant bit in the result. Therefore, each PE must be customized to its particular position in the pipeline to handle carries correctly. When an operand has been completely shifted out of the least significant word, that word must be discarded. For this to happen cleanly, $w$ should be a power of 2, $p$ must be a power of 2, and $vp$ must be divisible by $w$. This limits the possible combinations of word length and processing elements. These strict constraints allow for radix 2, 4, and 16 multipliers but prevent a radix-8 multiplier ($v$ = 3) from being built with left-shifting. An odd $v$ will never satisfy the necessary constraints.

## IV. Radix-8 Processing Element

The parallelized, right-shifting, Booth-encoded, Radix-8 processing element is shown in Fig 5. This design processes $v$ = 3 bits of the multiplier $X$ in each PE.

Each PE contains twelve $w$-bit registers, two $w$-bit Booth selectors, two $w$-bit CSAs, a 3-bit CPA and assorted support logic. This PE design stores the hard $3\hat{M}$ /$3Y$ multiple in a register. Negative multiples from Booth encoding and carry-injection are handled the same as in [7].

## V. Results

The processing elements were coded in Verilog and simulated in ModelSim. Verilog for all the Montgomery multiplier designs has been synthesized in Synplify Pro onto the Xilinx XC2V2000-6 Virtex II FPGA with "Sequential Optimizations" disabled to prevent flip-flops from being optimized into shift registers. Critical paths were obtained by synthesizing the kernel with $p$ = 2. A comparison of the parallelized high radix scalable Booth designs is shown in Table I.

For right shifting, Booth encoded designs, the critical path is through an inverter, the Booth selector ($2^v$ + 1-input multiplexer), two CSAs, a 2-input sign extension multiplexer, and a register.

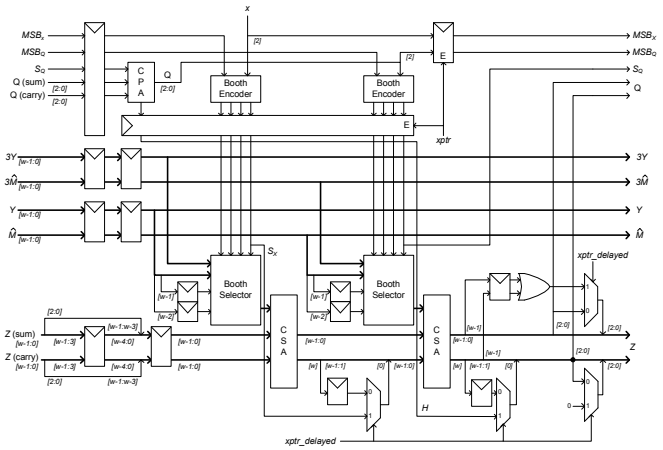


Fig. 5: Radix-8 PE

For left shifting, Booth encoded designs, the critical path is through a Booth encoder, the Booth selector, an AND gate for masking the lower bits, and two CSAs. The longer critical path for this design increases the cycle time over the right-shifting design. The right shifting design is able to pipeline the Booth encoder and CPA into the first stage to reduce the critical path.

For left shifting, non-Booth encoded designs, the critical path is through a product selector ($2^v$-input multiplexer), and 2 CSAs. Left shifting, non-Booth encoded designs have the shortest critical path and simplest hardware implantation at the cost of having to store $2^{v-1}$ hard multiples.

A comparison of hardware usage and exponentiation time for the parallelized high radix Booth designs is shown in Figures 6 and 7. The data includes the hardware in the kernel and controller, but not RAM bits or logic in the memories and FIFO (which are roughly the same for all designs). The amount of hardware was increase by increasing the pipeline length. The curves end when the pipeline reaches 100% utilization and more PEs do not increase performance. The modular exponentiation time is the time for $2n$+2 Montgomery multiplies.

## VI. Conclusions

This paper described the algorithms and design tradeoffs for high radix Montgomery multipliers. Left-shifting designs are uniformly better than right-shifting designs because fewer cycles and pipeline registers are needed. Of these designs, a radix-4, left-shifting, non-Booth encoded design has the best performance per unit of hardware.

For systems constrained by the cycle time of other elements, a radix-4, left-shifting, Booth encoded design is best. It has virtually identical performance per unit of hardware to the non-Booth encoded design but does not require precomputing the $3Y$ and $3\hat{M}$ multiples.

## References


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

TABLE I
COMPARISON OF PE FPGA RESOURCE USAGE AND CLOCK SPEED

| **Architecture** | **Ref.** | **Shift Dir.** | ***w*** | ***v*** | **4-input LUTs/ PE** | **Registers /PE** | **Critical Path** | **Clock Speed (MHz)** |
|---|---|---|---|---|---|---|---|---|
| Parallel radix-8 scalable Booth | This work | R | 4 | 3 | 80 | 71 | INV + MUX9 + 2CSA + MUX2 + REG | 263 |
| | | | 8 | 3 | 179 | 130 | | 240 |
| | | | 16 | 3 | 295 | 222 | | 212 |
| Parallel radix-4 scalable Booth | [7] | R | 4 | 2 | 50 | 49 | INV + MUX5 + 2CSA + MUX2 + REG | 259 |
| | | | 8 | 2 | 91 | 87 | | 249 |
| | | | 16 | 2 | 154 | 149 | | 248 |
| Parallel radix-4 scalable Booth | [7] | L | 4 | 2 | 54 | 41 | ENC + INV + MUX5 + 2CSA + AND + 2MUX2 + REG | 216 |
| | | | 8 | 2 | 102 | 60 | | 213 |
| | | | 16 | 2 | 176 | 89 | | 186 |
| Parallel radix-4 scalable non-Booth | [6] | L | 16 | 2 | 132 | 120 | 2CSA + BUF + MUX4 + REG | 248 |
| Parallel radix-2 scalable | [14] | L | 16 | 1 | 94 | 72 | AND + 2CSA + BUF + REG | 318 |
| Improved radix-2 scalable | [3] | L | 16 | 1 | 95 | 72 | 2AND + 2CSA + BUF + MUX2 + REG | 285 |

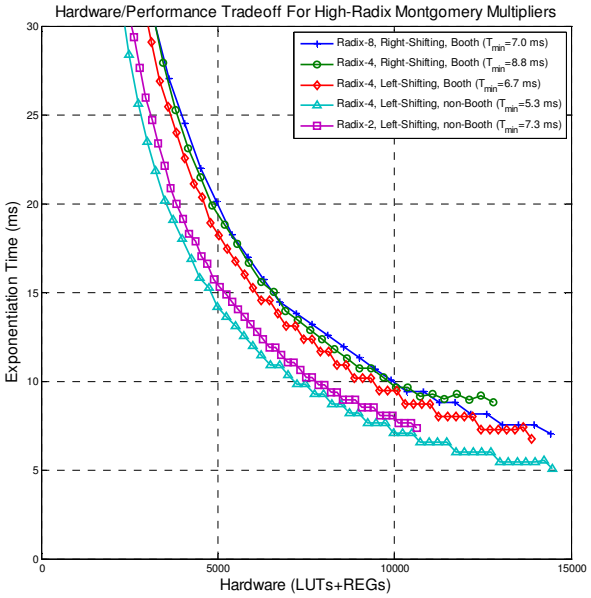


Fig. 6: Hardware/performance tradeoff for high-radix Montgomery multipliers.

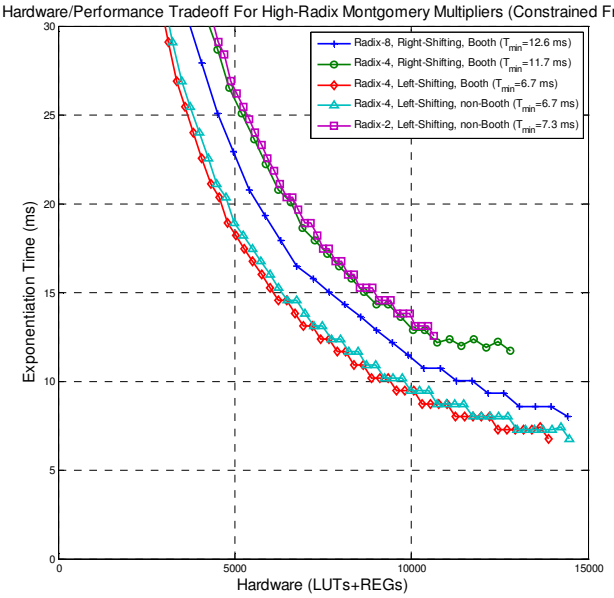


Fig. 7: Hardware/performance tradeoff for high-radix Montgomery multipliers. All PEs are set to same operating frequency.


[2] 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.

[3] D. Harris, R. Krishnamurthy, M. Anders, S. Mathew, and S. Hsu, "An improved unified scalable radix-2 Montgomery multiplier," *Proc. 17th IEEE Symp. Computer Arithmetic*, pp. 172-178, 2005.

[4] 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.

[5] N. Pinckney and D. Harris, "Parallelized radix-4 scalable Montgomery multipliers," *Proc. 20th SBCCI Conf. on Integrated Circuits and Systems Design*, pp. 306-311, 2007.

[6] N. Pinckney and D. Harris, "Parallelized radix-4 scalable Montgomery multipliers," submitted to *Journal of Integrated Circuits and Systems*, Feb. 2008.

[7] N. Pinckney, P. Amberg, and D. Harris, "Parallelized Booth-Encoded Radix-4 Montgomery Multipliers," submitted to *VLSI SOC* 2008.

[8] A. Tenca, G. Todorov, and Ç. Koç, "High--radix design of a scalable modular multiplier," Cryptographic Hardware and Embedded Systems, Ç. Koç and C. Paar, eds., 2001, *Lecutre notes in Computer Science*, No. 1717, pp. 189-206, Springer, Berlin, Germany.

[9] Y. Fan, X. Zeng, Y, Yu, G. Wang, and Q. Zhang, "A modified high-radix scalable Montgomery multiplier," *Proc. Intl. Symp. Circuits and Systems*, pp. 3382-3385, 2006.

[10] K. Kelley and D. Harris, "Parallelized very high radix scalable Montgomery multipliers," *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 1196-1200, Nov. 2005.

[11] K. Kelley and D. Harris, "Very high radix scalable Montgomery multipliers," *Proc. 5th Intl. Workshop on System-on-Chip*, pp. 400-404, July 2005.

[12] H. Orup, "Simplified quotient determination in high-radix modular multiplication," *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 193-199, July 1995.

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

[14] N. Jiang and D. Harris, "Parallelized Radix-2 Scalable Montgomery Multiplier," *IFIP Intl. Conf. on VLSI*, 2007.

[15] A. Booth, "A signed binary multiplication technique," *Quarterly J. Mechanics and Applied Mathematics*, vol. IV, part 2, pp. 236-240, June 1951.