Parallelized Very High Radix Scalable Montgomery Multipliers

Kyle Kelley and David Harris
Harvey Mudd College
301 E. Twelfth St. Claremont, CA 91711
{Kyle_Kelley, David_Harris}@hmc.edu

Abstract — **This paper describes a parallelized very high radix scalable Montgomery multiplier designed for non-redundant FPGA implementations. It improves on the very high radix scalable architecture by using techniques to parallelize the two multiplications within each processing element. The new design can perform 1024-bit modular exponentiation in 5.0 ms and 256-bit modular exponentiation in 0.20 ms using 2593 4-input lookup tables and 32 16 × 16 multipliers, improving the fastest scalable design yet reported.**

I. Introduction

Many modern cryptographic techniques rely on modular exponentiation as the fundamental operation. Montgomery's modular multiplication algorithm [8] is attractive for hardware implementations of cryptographic accelerators because it can perform modular exponentiation without needing costly division steps.

There have been many implementations of Montgomery multipliers. Scalable designs are attractive because they allow a variable number of fixed-width processing elements to operate on n-bit operands. [13] presents a scalable radix-2 design based on a kernel of p processing elements, each $w \times 1$ – bit wide. [4] improves on this design by reducing the latency between processing elements. [5] presents a scalable radix 2^v design optimized for implementation on an FPGA, using $w \times v$ – bit processing elements.

This paper improves on the scalable radix-2^v design in [5] by transforming the modulus to avoid multiplication in the calculation of *reduce* [7] and by pre-scaling X by 2^v to allow the processing element multiplications to occur in parallel [2] and [6]. These improvements come at the expense of a single pre-computation for the new modulus and an extra iteration because of the additional factor of 2^v.

II. Montgomery Multiplication

Montgomery multiplication is defined as

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

with the notation

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

It is performed with the following steps [8]:

Multiply: $Z = X \times Y$
Reduce: $reduce = Z \times M' \bmod R$
$Z = [Z + reduce \times M] / R$
Normalize: if $Z \geq M$ then $Z = Z - M$

reduce has the important property that $Z + reduce \times M$ has 0's in the n least significant positions. The mod R and divide by R steps are trivial because R is a power of 2, so Montgomery multiplication avoids difficult divisions. The normalize step can be skipped in certain repeated Montgomery multiplies, and so we ignore it for the rest of this paper.

A. Parallelized very high radix

The Montgomery multiplication algorithm above can be rewritten to allow a hardware implementation using $v \times n$ - bit multipliers instead of $n \times n$ - bit multipliers [10]. Multiplication in the calculation of *reduce* can be avoided by transforming the modulus M to $\hat{M}$ at the expense of a single precomputation. Furthermore, the multiplication and reduction steps can occur in parallel if X is prescaled by 2^v. The tradeoff here is that the result Z can now be up to 2^v times larger than M, so the normalization step is no longer a single subtraction. However, it can still be skipped in certain repeated Montgomery multiplies, and so we ignore it for the rest of this paper.

The following notation is used to describe this parallelized very high radix Montgomery multiplication. Fig. 1 is equivalent to Algorithm 4 with zero stages of delay [11].

v: outer digit length, radix = 2^v
M: n-bit odd modulus
M': n-bit integer satisfying $(-MM') \bmod 2^n = 1$
$\widetilde{M}$: $(M' \bmod 2^v)M$
n': $n + v$, length of prescaled X
Y: n'-bit multiplicand
R: $2^{n'}$

R^{-1}: modular multiplicative inverse of R, $(RR^{-1}) \bmod M = 1$

$$\hat{M} : \frac{\tilde{M} + 1}{2^v}$$

$$f : \left\lceil \frac{n'}{v} \right\rceil$$

$Z = 0$
for $i = 0$ to f
  $reduce = Z_{v-1:0}$
  $Z = (Z >> v) + reduce \times \hat{M} + X_{(i+1)v-1:iv} \times Y$

Fig. 1. Parallelized radix-2^v algorithm

B. Parallelized very high radix scalable design

The algorithm in Fig. 1 can be rewritten to perform $v \times w$ – bit multiplication instead of $v \times n$ – bit, making it a scalable design. The following additional notation will be used to describe the parallelized very high radix scalable Montgomery multiplication algorithm.

w: inner word length

$$e : \left\lceil \frac{n}{w} \right\rceil + 1$$

C: $(v+1)$-bit carry digit

Note that very high radix designs should use $w \geq v$ because each w-bit word is right-shifted by v bits in the reduction step.

$Z = 0$
for $i = 0$ to f
  $C = 0$
  $reduce = Z_{v-1:0}$
  for $j = 0$ to $e - 1 + \left\lfloor \frac{v}{w} \right\rfloor$
    $(C, Z_{(j+1)w-1:jw}) = (Z_{(j+1)w+v-1:(j+1)w}, Z_{(j+1)w-1:jw+v}) +$
    $reduce \times \hat{M}_{(j+1)w-1:jw} +$
    $X_{(i+1)v-1:iv} \times Y_{(j+1)w-1:jw} + C$

Fig. 2. Parallelized scalable radix-2^v algorithm

III. Hardware Implementation

This improved algorithm allows similar hardware architectures to those presented in [4], [5], and [13]. Fig. 3 shows the architecture of a scalable Montgomery multiplier with a kernel of p PEs. Each PE receives v bits of X and *reduce* and w bits of $\hat{M}$, Y, and Z on each step. In one kernel cycle, p v-bit digits of X are processed. Hence, $k = n'/pv$ full kernel cycles are necessary to process all the bits of X, with an additional partial kernel cycle to account for the factor of 2^v. A tristate bus allows the output of any PE to be written to the result.

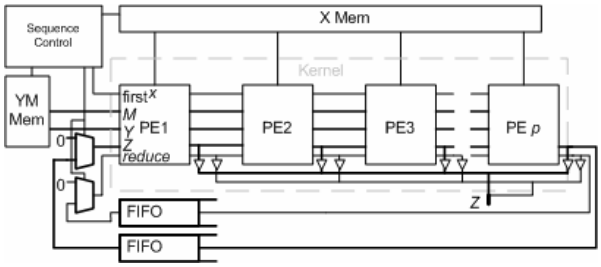

Fig. 3. Scalable very high radix Montgomery multiplier architecture

A. Processing elements

Fig. 4 shows the implementation of a processing element. The weights of the lines indicate the bus widths. The PE contains a pair of $w \times v$ - bit multipliers, a pair of 3:2 carry-save adders, and one $w + v$ – bit carry-propagate adder. A feedback register holds the running carry C. A control path at the top of the PE indicates when X and *reduce* should be latched. Compared to [5], the pre-computation of $\hat{M}$ allows two multiplexers to be removed from the PE, one of which was in the critical path.

The PE is pipelined to offer single-cycle throughput but two-cycle latency, compared to four-cycle latency in [5]. This latency improvement further decreases the area of each PE by removing pipeline registers.

B. Latencies

Fig. 5 shows the pipeline timing for a system with four processing elements. The vertical axis represents time and the horizontal represents PEs. On cycle 1, PE 1 computes $Z_{w-1:0} = X_{(i+1)v-1:iv} \times Y_{(j+1)w-1:jw}$. On each of the e-1 subsequent cycles, it processes the same digit of X but the next words of Y, Z, and $\hat{M}$. For example, on cycle 2, PE 1 computes $Z_{2w-1:w}$ and right shifts Z by v bits to produce the new least significant word of Z. On cycle 3, PE 2 can begin using this least significant word of Z.

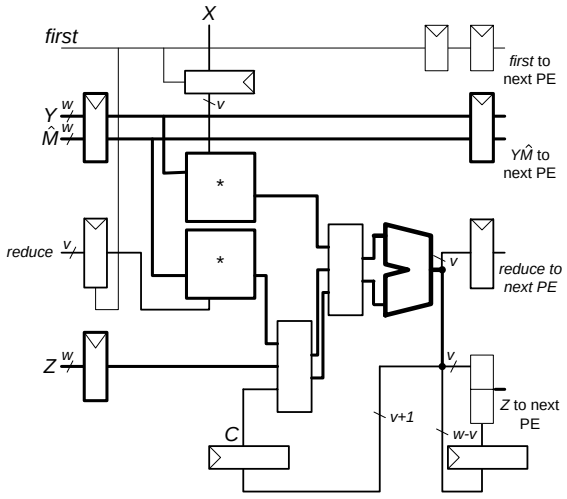

Fig. 4. Processing element

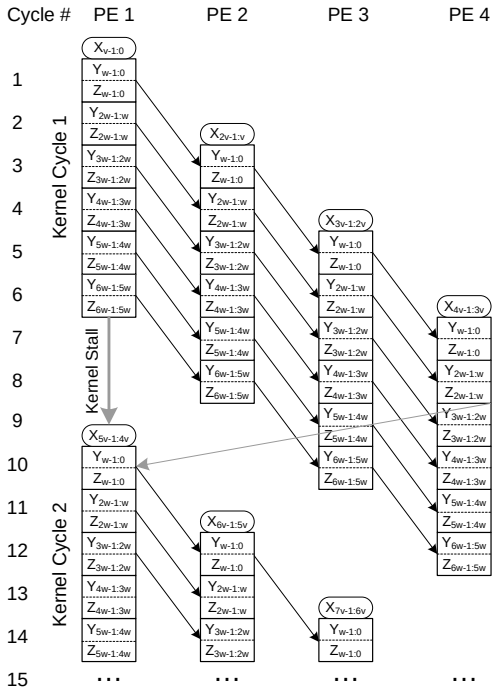

Fig. 5. Hardware pipeline diagram p=4, e=5

Recall that an entire multiplication requires $k = n'/pv$ full kernel cycles, plus a partial cycle. The kernel cycle time is the number of clock cycles until PE 1 can begin processing the next digit of X. PE 1 cannot begin the next kernel cycle until it has processed all the words of Z and until PE p has produced the first word of Z. The output of PE p is bypassed back to PE 1 through a FIFO, adding one cycle of latency. This leads to two cases to determine the multiplication latency. Case I corresponds to a large number of words, e, relative to the number of processing elements, p. Here there is no stall between kernel cycles, and so the PE hardware is used with maximal efficiency. Case II corresponds to a large number of processing elements relative to the number of words. As shown in Fig. 5, the first PE must be stalled until the last PE finishes calculating the first word of Z.

In general, to handle all the words of Y, a particular PE must perform (e) clock cycles in one kernel cycle (or in one iteration of the outer loop).

Note that in cases where w=v, this (e) is replaced with (e+1) because an additional cycle is then necessary to handle the v+1 carry bits. There is a 2 clock cycle latency between PEs. Thus, with p PEs, there is a 2p delay before the first PE may begin processing again.

Therefore Case I occurs when (e)>=2p+1 and Case II occurs when (e)< 2p+1.

Case I: The first PE is used continuously (e) times per kernel cycle for k full kernel cycles. The length of the final partial kernel cycle depends on f and p. Assuming f is evenly divisible by p, the output of PE 1 during the additional kernel cycle is the final result, and so the partial kernel cycle is (e) clock cycles. Otherwise (e) + 2*(f mod p) additional clock cycles are necessary. Therefore the total delay d_{I} is

$$d_{\mathrm{I}} = k(e) + (e) + 2*(f \bmod p) \qquad (2)$$

Case II: Each kernel cycle takes 2p clock cycles until the first word of Z is ready, plus 1 to bypass the result back to the first PE through the queue. Thus k(2p+1) full kernel cycles are needed. Again, the last partial kernel cycle has an additional delay of (e) + 2*(f mod p) clock cycles. Therefore the total delay d_{II} is

$$d_{\mathrm{II}} = k(2p+1) + (e) + 2*(f \bmod p) \qquad (3)$$

Rewriting these delays in terms of the design parameters n, w, v, and p, and assuming integer divisibility, we obtain

$$d_I = \frac{n^2}{pwv} + \frac{2n+v}{pw} + \frac{n+v}{w} \text{ for } n > 2pw - v \qquad (4)$$

$$d_{II} = \frac{2n}{v} + \frac{n}{vp} + \frac{n+v}{w} + \frac{1}{p} + 2 \text{ for } n \le 2pw - v \qquad (5)$$

These delays are similar those in [5], except the 4 cycle latency between processing elements is now 2 because multiplication and reduction take place in parallel. The area grows as the product pwv. When there are relatively few small PEs, the area-delay product is approximately n^2, and so this design is efficient. In Case II the latency approaches n(2/v + 1/w) with large amounts of hardware, and so the area-delay product is approximately np(2w+v). This parallelized design performs comparably to the old design when the number of words dominates (Case I), and outperforms significantly (approaching a factor of 2 speedup) when a large amount of hardware is available (Case II).

4. Results

The very high radix Montgomery multiplier was coded in Verilog parameterized by p, w, and v, and verified against a C reference model. It was synthesized using Synplicity Pro targeting a Xilinx Virtex-II speed grade 6 XC2V2000-6 FPGA [16]. The results were not verified on an actual chip. Each PE uses two dedicated 18×18 block multipliers, two carry-save adders, and one carry-propagate adder. The intrinsic size of the multipliers makes w = v = 16 a sweet spot for this design.

The complete 16-PE radix 2^{16} Montgomery multiplier (including sequencing hardware) contains 2593 LUTs, 32 multipliers, and approximately 5n bits of RAM for the FIFO and operand storage. It operates at a worst-case 135 MHz, limited by the critical path through the multiplier and adders.

Table I compares the times for 256-bit and 1024-bit modular exponentiations for various Montgomery multiplier hardware implementations. The exponentiation times are 2n + 2 times that of a single modular multiplication.

TABLE I. COMPARISON OF MODULAR EXPONENTIATION TIMES

Description	Technology	Hardware	Clock Speed	Scalable	Reference	256-bit time (ms)	1024-bit time (ms)
Parallel scalable radix 2^{16} 16 PEs x 16 bits	Xilinx Virtex II	2593 LUTs + 32 mults + ~5*n* RAM	135 MHz	Yes	This work	0.20	5.0
Parallel scalable radix 2^{16} 4 PEs x 16 bits	Xilinx Virtex II	695 LUTs + 8 mults + ~5*n* RAM	135 MHz	Yes	This work	0.35	17
Scalable radix 2^{16} 16 PEs x 16 bits	Xilinx Virtex II	2847 LUTs + 32 mults + ~5*n* RAM	102 MHz	Yes	[5]	0.40	6.6
Scalable radix 2^{16} 4 PEs x 16 bits	Xilinx Virtex II	780 LUTs + 8 mults + ~5*n* RAM	102 MHz	Yes	[5]	0.45	22
Improved radix 2 64 PEs x 16 bits	Xilinx Virtex II	5598 LUTs + ~5*n* RAM	144 MHz	Yes	[4]	1.0	16
Improved radix 2 16 PEs x 16 bits	Xilinx Virtex II	1514 LUTs + ~5*n* RAM	144 MHz	Yes	[4]	1.1	59
General radix 16 1 PE x 64 bits	0.11µm CMOS synthesized	61 Kgates	250 MHz	Yes	[9]	n/a	7.3
Scalable radix 8 16 PEs x 16 bits	0.5µm CMOS synthesized	28 Kgates	64 MHz	Yes	[15]	1.6	46
Systolic radix 16 1024-bit	Xilinx XC40250XV	3317 LUTs	45 MHz	No	[1]	n/a	12
Systolic radix 16 256-bit	Xilinx XC40150XV	909 LUTs	47 MHz	No	[1]	0.73	n/a
Tenca-Koç radix 2 40 PEs x 8 bits	0.5µm CMOS synthesized	28 Kgates	80 MHz	Yes	[13]	3.8	88
Scalable high radix	0.5µm CMOS estimated	33 Kgates (estimated)	44 MHz	Yes	[3]	1.8	82

The scalable 16 PE design from this work consumes about 10% fewer Virtex II LUTs than the equivalent design from [5], and performs 1024-bit modular exponentiation in 5.0 ms as compared to 6.6 ms. It performs 256-bit modular exponentiation in 0.20 ms as compared to 0.40 ms. Thus, the parallelized very high radix scalable Montgomery multiplier performs significantly faster when there are many PEs relative to the number of words. The speedup observed in the 1024-bit case can be attributed to the faster clock frequency, which was the result of a shorter critical path in the processing elements. These simplifications within each PE further allow a reduced kernel area.

5. CONCLUSIONS

In summary, this paper has improved the scalable very high radix Montgomery multiplier by replacing X by $2^v X$ to allow the multiplication and reduction steps to occur in parallel within each processing element and by a modulus transformation. The parallel operation reduces latency from 4 cycles to 2 between PEs and saves pipeline registers at the expense of extra postcomputation cycles. The reduction modification eliminates a multipliexer from the critical path. Using a tristate bus within the kernel was a further improvement, allowing the design to work with any choice of p. This design was targeted for a non-redundant FPGA implementation. An implementation with 16 16×16 PEs uses 32 dedicated 18×18 block multipliers and 2593 lookup tables to perform 1024-bit modular exponentiation in 5.0 ms and 256-bit modular exponentiation in 0.20 ms.

There remain several opportunities for investigation of very high radix design Montgomery multiplication. If $v < w$, a subsequent PE may begin operating on the w-v bits that are immediately available without waiting for a shift [4]. By conditionally killing carries within the MAC, the algorithm extends to unified multipliers for GF(2^n) as well as GF(p). It would also be interesting to investigate a quotient-pipelined [11], [12] ASIC implementation in redundant form in which tradeoffs may be made among w, v, and p to affect cycle time and cycle count.

REFERENCES

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

[2] S. Eldridge and C. Walter, "Hardware implementation of Montgomery's modular multiplication algorithm," *IEEE Trans. Computers*, vol. 42, no. 6, pp. 693-699, June 1993.

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

[4] D. Harris *et al*., "An improved unified scalable radix-2 Montgomery multiplier", to appear in *IEEE Symp. Computer Arithmetic*, 2005.
[5] K. Kelley and D. Harris, "Very high radix scalable Montgomery multipliers", to appear in *IEEE IWSOC Conference*, July 2005.
[6] P. Kornerup, "High-radix modular multiplication for cryptosystems," *Proc 11th IEEE Symp. Computer Arithmetic*, pp. 277-283, 1993.
[7] P. Kornerup, "A systolic, linear-array multiplier for a class of right-shift algorithms," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 892-898, August 1994.
[8] P. Montgomery, "Modular multiplication without trial division," *Math. Of Computation*, vol. 44, no. 170, pp. 519-521, April 1985.
[9] 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.
[10] H. Orup and P. Kornerup, "A high-radix hardware algorithm for calculating the exponential M^E Modulo N," *Proc. 10th IEEE Symp. Computer Arithmetic*, pp. 51-56, 1991.
[11] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 193-199, 1995.
[12] M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," *Proc. 11th IEEE Symp. Computer Arithmeti*c, pp. 252-259, 1993.
[13] A. Tenca and Ç. Koç, "A scalable architecture for modular multiplication based on Montgomery's algorithm," *IEEE Trans. Computers*, vol. 52, no.9, pp. 1215-1221, Sept. 2003.
[14] 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.
[15] G. Todorov, "ASIC design, implementation and analysis of a scalable high-radix Montgomery multiplier," M.S. Thesis, Oregon State University, June 2001.
[16] Xilinx, Virtex-II Pro and Virtex-II Pro X Platform FPGAs Datasheet, June 30, 2004, www.xilinx.com