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 very high radix scalable Montgomery multiplier. It extends the radix-2 Tenca-Koç scalable architecture using w × v – bit integer multipliers in place of AND gates. The new design can perform 1024-bit modular exponentiation in 6.6 ms using 2847 4-input lookup tables and 32 16 x 16 multipliers, making it the fastest scalable design yet reported.

1. Introduction

Many cryptographic algorithms require modular exponentiation of very long n-bit operands, which is performed with repeated modular multiplications. Typical values of n are 256-2048. Montgomery's modular multiplication algorithm [5] is widely used in these applications because it avoids costly division steps.

Much research has been devoted to hardware implementations of Montgomery multipliers. [9] describes a scalable radix-2 design that handles word lengths independent of the width of the processing elements and processes the multiplier one bit at a time. [10] extends the scalable approach to radix-4, handling two bits at a time. [1] proposes radix-4 and radix-16 designs using precomputed multiples of the operands, but the design is not scalable. [6] presents a very high radix implementation, but focuses on optimizing the memory interface to a single bandwidth-constrained operand memory and uses a single processing element.

This paper presents a very high radix scalable Montgomery multiplier. It extends the scalable architecture of [9] to radix 2^v by using a w × v-bit multiplier. It also improves on [6] by allowing for a variable number of processing elements. We begin by defining our notation and reviewing two algorithms for very high radix Montgomery multiplication. Both algorithms can be unrolled onto the same hardware pipeline. We present the design of the processing elements in the pipeline and analyze the system latency. The very high radix designs have been mapped onto Xilinx FPGAs and the results are compared with competing implementations.

2. Montgomery multiplication

Montgomery multiplication is defined as

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

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 [5]:

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

2.1. Tenca-Koç scalable radix-2 multiplier

The algorithm above can be implemented in a straightforward fashion if n-bit adders and n x n–bit multipliers are available. This becomes impractical for large n (e.g. 1024). Moreover, we would like a scalable design that can handle arbitrary values of n by reusing fixed-width hardware.

Tenca and Koç [9] describe a scalable radix-2 implementation with w-bit processing elements (PEs) using the algorithm in Fig. 1. Each PE iterates over v = 1 bit of X at a time. It requires $e = \lceil n/w \rceil$ steps for the PE to handle all n bits of Y and M. In radix 2^v, only the v least significant bits of *reduce* are necessary because Z is only right-shifted by v bits [2]. *M'* is always odd, so for radix 2, *reduce* simplifies to the least significant bit of Z, Z_0.

```
$Z = 0$
for $i = 0$ to $n$-1
    $(C^A, Z_{w-1:0}) = Z_{w-1:0} + X_i \times Y_{w-1:0}$
    $reduce = Z_0$
    $(C^B, Z_{w-1:0}) = Z_{w-1:0} + reduce \times M_{w-1:0}$
    for $j = 1$ to $e$
        $(C^A, Z_{(j+1)w-1:jw}) = Z_{(j+1)w-1:jw} + X_i \times Y_{(j+1)w-1:jw} + C^A$
        $(C^B, Z_{(j+1)w-1:jw}) = Z_{(j+1)w-1:jw} + reduce \times M_{(j+1)w-1:jw} + C^B$
        $Z_{jw-1:(j-1)w} = (Z_{jw}, Z_{jw-1:(j-1)w+1})$
```

Fig. 1. Tenca-Koç scalable radix-2 Montgomery multiplication algorithm

2.2. Very high radix

Scalable Montgomery algorithms can be generalized to higher radices. [4] describes radix 2^w designs using w-bit processing elements, suitable for software implementations with square multipliers. This paper extends the generalization to radix 2^v designs using w-bit processing elements. These designs require $w \times v$ – bit rectangular multipliers.

The following notation will be used to describe the very high radix Montgomery multiplication algorithms. Figures 2 and 3 extend two of the most efficient algorithms from [4]. In *coarsely integrated operand scanning* (CIOS), the multiplication and reduction steps are separated for each v-bit digit of X. In *finely integrated operand scanning* (FIOS), the steps are combined for each digit of X and word of Y. The algorithms use the following parameters.

w:	word length
v:	digit length
e:	$\lceil n/w \rceil$, the number of words of *Y, M* to process
f:	$\lceil n/v \rceil$, the number of digits of X to process
C^A:	v-bit carry digit
C^B:	v-bit carry digit

```
$Z = 0$
for $i = 0$ to $f$-1
    $C^A = 0$
    for $j = 0$ to $e + \lceil \frac{v+1}{w} \rceil$-1
        $(C^A, Z_{(j+1)w-1:jw}) = Z_{(j+1)w-1:jw} + X_{(i+1)v-1:iv} \times Y_{(j+1)w-1:jw} + C^A$
    $C^B = 0$
    $reduce = (M'_{v-1:0} \times Z_{w-1:0})_{v-1:0}$
    for $j = 0$ to $e + \lceil \frac{v+1}{w} \rceil$-1
        $(C^B, Z_{(j+1)w-1:jw}) = Z_{(j+1)w-1:jw} + reduce \times M_{(j+1)w-1:jw} + C^B$
        $Z_{jw-1:(j-1)w} = (Z_{jw+v-1:jw}, Z_{jw-1:(j-1)w+v})$
```

Fig. 2. Scalable radix-2^v coarsely integrated operand scanning algorithm

```
$Z = 0$
for $i = 0$ to $f$-1
    $(C^A, Z_{w-1:0}) = Z_{w-1:0} + X_{(i+1)v-1:iv} \times Y_{w-1:0}$
    $reduce = (M'_{v-1:0} \times Z_{w-1:0})_{v-1:0}$
    $(C^B, Z_{w-1:0}) = Z_{w-1:0} + reduce \times M_{w-1:0}$
    for $j = 1$ to $e + \lceil \frac{v+1}{w} \rceil$-1
        $(C^A, Z_{(j+1)w-1:jw}) = Z_{(j+1)w-1:jw} + X_{(i+1)v-1:iv} \times Y_{(j+1)w-1:jw} + C^A$
        $(C^B, Z_{(j+1)w-1:jw}) = Z_{(j+1)w-1:jw} + reduce \times M_{(j+1)w-1:jw} + C^B$
        $Z_{jw-1:(j-1)w} = (Z_{jw+v-1:jw}, Z_{jw-1:(j-1)w+v})$
```

Fig. 3. Scalable radix-2^v finely integrated operand scanning algorithm

Very high radix designs should use $w = v$ because each w-bit word is right-shifted by v bits in the reduction step.

3. Hardware implementation

Fig. 4 shows the architecture of a scalable Montgomery multiplier with a kernel of p PEs. Each PE receives v bits of X and w bits of M, Y, and Z on each step. In one kernel cycle, p v-bit digits of X are processed. Hence, k = n/pv kernel cycles are necessary to process all the bits of X.

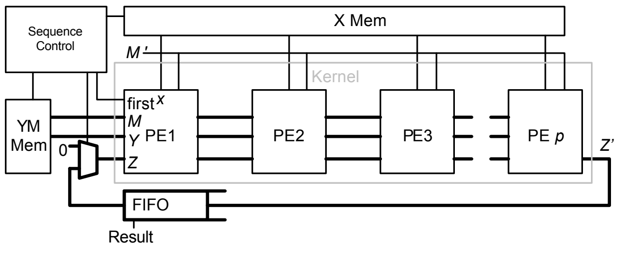

Fig. 4. Scalable very high radix Montgomery multiplier architecture

3.1. Processing elements

Fig. 5 shows the implementation of a processing element. The weights of the lines indicate the bus

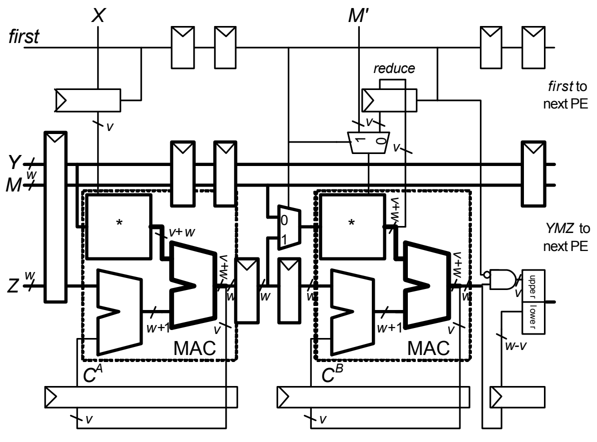

Fig. 5. Processing element

widths. The PE contains a pair of multiply-accumulate (MAC) circuits for the multiplication and reduction steps. Feedback registers hold the running carries C^A and C^B. At the beginning of the kernel cycle, the second multiplier is also used to compute reduce. A control path at the top of the PE indicates when X should be latched and when reduce should be computed and latched.

The PE is pipelined to offer single-cycle throughput but four-cycle latency. This compares to two-cycle latency in [9]. The greater latency permits the cycle time to be limited to that of a single MAC.

3.1. Latencies

Fig. 6 shows the pipeline timing for a system with three processing elements. The vertical axis represents time and the horizontal represents PEs (with two MAC columns per PE). On cycle 1, the first MAC in PE 1 computes $Z_{w-1:0} = Z_{w-1:0} + X_{v-1:0} \times Y_{w-1:0}$. On each of the $e+1$ subsequent cycles, it processes the same digit of X but the next word of Y and Z. On cycle 2, the second MAC in PE 1 computes $reduce = M'_{v-1:0} \times Z_{w-1:0}$ and saves the v-bit result. On cycle 3, it reduces $Z_{w-1:0} = Z_{w-1:0} + reduce \times M_{w-1:0}$. On cycle 4, it reduces the next word $Z_{2w-1:w} = Z_{2w-1:w} + reduce \times M_{2w-1:w}$ and right shifts Z by v bits to produce a new least significant word of Z. On cycle 5, PE 2 can begin using this least significant word of Z.

Recall that an entire multiplication requires $k = n/pv$ kernel cycles. 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.

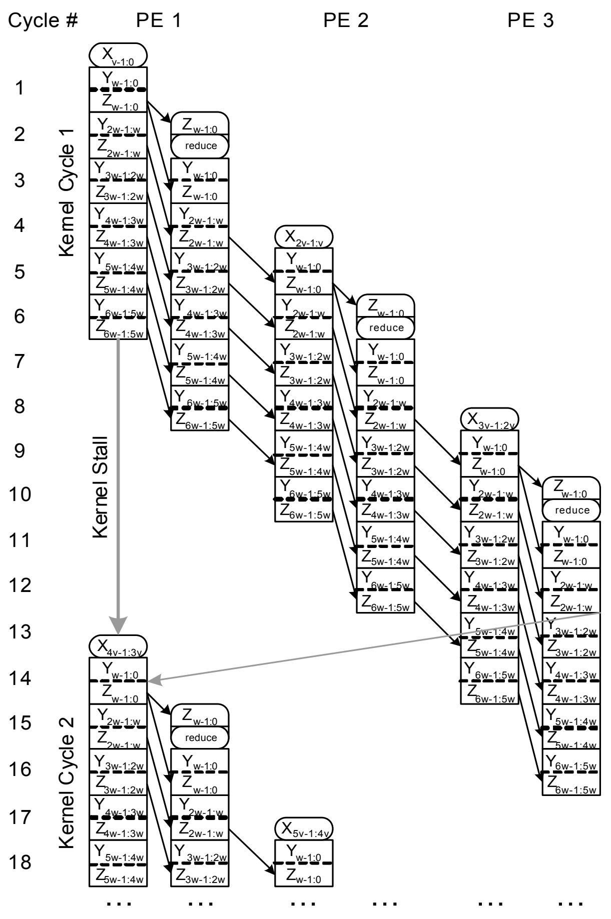

Fig. 6. Hardware pipeline diagram p=3, e=4

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. 6, 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 $(e+2)$ cycles in one kernel cycle (or in one iteration of the outer loop). There is a 4 clock cycle latency between PEs. Thus, with p PEs, there is a $4p$ delay before the first PE may begin processing again. Therefore Case I occurs when $(e+2)>4p$ and Case II occurs when $(e+2)<=4p$.

Table I. Comparison of modular exponentiation times

Description	Technology	Hardware	Clock Speed	Scalable	Reference	256-bit time (ms)	1024-bit time (ms)
Scalable radix 2^{16} 16 PEs x 16 bits	Xilinx Virtex II	2847 LUTs + 32 mults + ~5*n* RAM	102 MHz	Yes	This work	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	This work	0.45	22
Improved radix 2 64 PEs x 16 bits	Xilinx Virtex II	5598 LUTs + ~5*n* RAM	144 MHz	Yes	[3]	1.0	16
Improved radix 2 16 PEs x 16 bits	Xilinx Virtex II	1514 LUTs + ~5*n* RAM	144 MHz	Yes	[3]	1.1	59
General radix 16 1 PE x 64 bits	0.11μm CMOS synthesized	61 Kgates	250 MHz	Yes	[6]	n/a	7.3
Scalable radix 8 16 PEs x 16 bits	0.5μm CMOS synthesized	28 Kgates	64 MHz	Yes	[12]	1.6	46
Systolic radix 16 1024-bit	Xilinx XC40250XV	3317 LUTs	45 MHz	No	[1]	n/a	12
Tenca-Koç radix 2 40 PEs x 8 bits	0.5μm CMOS synthesized	28 Kgates	80 MHz	Yes	[9]	3.8	88
Systolic radix 16 256-bit	Xilinx XC40150XV	909 LUTs	47 MHz	No	[1]	0.73	n/a
Scalable high radix	0.5μm CMOS estimated	33 Kgates (estimated)	44 MHz	Yes	[11]	1.8	82

Case I: The first PE is used continuously e+3 times per kernel cycle for k kernel cycles (the kernel cycle time has been increased by one to avoid resource contention of the multiplier in MAC 2, which is used in the reduce calculation). Then each remaining PE requires 4 more cycles to complete. Finally there are 2 extra cycles needed for the reduction MAC to finish and shift. Therefore the total delay d_I is

$$d_I = k(e+3) + 4(p-1) + 2$$

Case II: Each kernel cycle takes $4p$ cycles until the first word of Z is ready, plus 1 to bypass the result back to the 1st PE through the queue. Thus $k(4p+1)$ cycles are needed. The last kernel cycle has an additional delay for the final PE to finish its processing. Since this PE takes e+4 cycles (including 2 cycles for the reduction MAC to finish and shift), but 5 of them have already been accounted for, there are e+4-5 = e-1 cycles remaining. Therefore the total delay d_{II} is

$$d_{II} = k(4p+1) + (e - 1)$$

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

$$d_I = \frac{n^2}{wvp} + \frac{3n}{pv} + 4p - 2 \text{ for } n > 4pw - 2w$$

$$d_{II} = \frac{4n}{v} + \frac{n}{vp} + \frac{n}{w} - 1 \text{ for } n \le 4pw - 2w$$

These delays are similar those in [9], except the 2 cycle latency between processing elements is now 4 because multiplication and reduction take place in separate cycles. 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 $n(4/v + 1/w)$ with large amounts of hardware, and so the area-delay product is approximately $np(4w+v)$.

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 [13]. The results were not verified on an actual chip. Each PE requires 2 MACs. Each MAC uses a dedicated 18×18 block multiplier and two carry-propagate adders. The intrinsic size of the multipliers makes $w = v = 16$ a sweet spot for this design.

The complete radix 2^{16} Montgomery multiplier (including sequencing hardware) contains 2847 LUTs, 32 multipliers, and approximately 5n bits of RAM for the FIFO and operand storage. It operates at a worst-case 102 MHz, limited by the MAC.

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.

The scalable 16 PE design from this work consumes about half as many Virtex II LUTs as the improved radix-2 64 PEs × 16 design from [3], but performs 1024-bit modular exponentiation in 6.6ms as compared to 16ms. Thus, FPGAs with dedicated multipliers are very well suited to very high radix Montgomery multiplication.

5. Conclusions

In summary, this paper has extended the scalable Tenca-Koç Montgomery multiplier to very high radices using a MAC in place of a carry-save adder. It also extended the Mukaida-Tanenaka design by supporting multiple PEs. The very high radix approach is well-suited to FPGA implementations because of their rich set of dedicated multipliers. An implementation with 16 16×16 PEs uses 32 dedicated 18×18 block multipliers and 2847 lookup tables to perform 1024-bit modular exponentiation in 6.6 ms.

While the specific implementation used a square multiplier, the paper has generalized analysis for $w \times v$-bit rectangular multipliers. CIOS and FIOS offer tradeoffs in software implementations, but they both unroll onto the identical hardware pipeline.

Several opportunities still exist for improving the very high radix design. Quotient pipelining [8], [7] might be used to reduce the latency between processing elements. If $v < w$, a subsequent PE may begin operating on the w-v bits that are immediately available without waiting for a shift [3]. By conditionally killing carries within the MAC, the algorithm extends to unified multipliers for GF(2^n) as well as GF(p). It would be interesting to investigate an ASIC implementation of this design in which tradeoffs may be made among w, v, and p to affect cycle time and cycle count.

10. References

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

[2] S. Dusse and B. Kaliski, "A cryptographic library for the Motorola DSP56000," *Eurocrypt 90, Lecture Notes in Computer Science*, No. 473, I. B. Damgaard, ed., Springer-Verlag, New York, 1990, pp. 230-244.

[3] D. Harris *et al.*, "An improved unified scalable radix-2 Montgomery multiplier", submitted to *IEEE Symp. Computer Arithmetic*, 2005.

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

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

[6] 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.

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

[8] M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 252-259, 1993.

[9] 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.

[10] 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.

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

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

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