Quotient Pipelined Very High Radix Scalable Montgomery Multipliers

Nan Jiang and David Harris
Harvey Mudd College
301 E. Twelfth St. Claremont, CA 91711
{Nan_Jiang, David_Harris}@hmc.edu

Abstract— **This paper describes the FPGA implementation of a scalable very high radix Montgomery multiplier using quotient pipelining. It improves upon previous designs by removing critical dependencies between successive processing elements. This design can perform 1024-bit modular exponentiation in 5.1 ms using 3825 4-input lookup tables and 32 18×18 multipliers, a 20% speed increase over a comparable design without quotient pipelining.**

I. Introduction

Modular exponentiation is widely used in modern cryptography algorithms, such as RSA and digital signatures. However the operation usually involves 256 to 2048-bit numbers and is time-consuming due to the long divisions involved in the modulo calculation. Montgomery multiplication transforms this difficult division into a simple bit shift, and is therefore more attractive for hardware implementation.

Since the advent of Montgomery's algorithm in [4], there have been many implementations of the Montgomery multiplier. In a conventional radix-2 implementation, an n-bit multiplication is performed using n steps, each acting on 1 bit of the multiplier and all n bits of the multiplicand. These designs only support one choice of n. Scalable radix-2 designs [6, 2] break the multiplicand into w-bit chunks. They contain processing elements (PEs) organized in a systolic array. Each PE handles 1 bit of the multiplier and w bits of the multiplicand at a time. The scalable multiplier iterates until the entire multiplication completes. We have recently proposed scalable very high radix designs [3, 1] that handle v bits of the multiplier and w bits of the multiplicand in each PE. These designs require $w \times v$ bit multiplication in each PE and are well suited to FPGAs containing dedicated multipliers.

One major delay that exists in very high radix Montgomery multiplication is the calculation of *reduce*. *Reduce* is the crucial value that transforms the long division into a simple shift. Past implementations such as [3] required two extra cycles per PE to handle the *reduce* calculation. Reference [5] proposed several variations of Montgomery's algorithm that decreased the delay caused by *reduce*. In particular, the parallel algorithm was implemented in [1], eliminating the two extra cycles that were present in [3]. This paper intends to further improve the *reduce* calculation through the quotient pipelining algorithm presented in [5] and completely eliminate the *reduce* dependency between successive PEs.

II. Background

The basic Montgomery multiplication algorithm is

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

With the notation

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

Montgomery showed how to perform this multiplication without dividing by M in [4]:

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

The *reduce* term has the property such that it forces the numerator of the reduce step's second equation to be divisible by R, simplifying the division to a shift.

A. Parallelized very high radix design

In [1], an alternative implementation of the Montgomery algorithm based on [5] is discussed. This parallel algorithm, which prescales X by 2^v and uses a pre-calculated value of $\hat{M}$, allows the reduce and multiply steps to occur simultaneously.

Each PE has a two-cycle latency, which is half the latency of the previous implementation [3]. The block diagram of the PE is shown in Fig. 1.

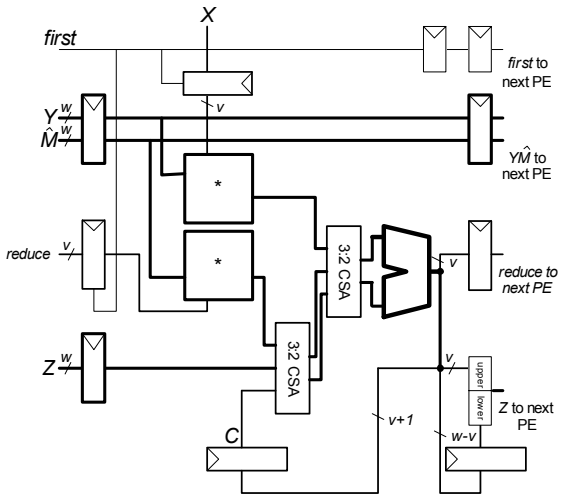

Fig. 1: Parallel Design Processing Element

The critical path for this design is from the *reduce* input to the *reduce* output and consists of a register delay, a multiplier, two carry save adders and a carry propagation adder. The CPA is necessary in the design, because the reduce output of a PE must be in non-redundant form for the $reduce \times \hat{M}$ calculation of the next PE. Removing this CPA from the critical path will be at the core of the speed increase offered by quotient pipelining.

III. QUOTIENT PIPELINING

The cycle time of the parallel design could be significantly improved by removing the CPA from the critical path. However, since the *reduce* output of each PE is immediately used in the successive PE, the CPA is required for the conversion to non-redundant form. This problem would be trivial if the *reduce* output is not immediately used, and an entire PE cycle could be dedicated for the conversion to non-redundant form.

Quotient pipelining accomplishes exactly this task. A d-stage quotient pipeline design lets the *reduce* output of a PE to be used $d+1$ PEs later, allowing d PE cycles to be used for non-redundant conversion. Since CPA delay is always less than that of a PE cycle, d is set to 1 for the reminder of the paper. The general quotient pipeline algorithm and its proof of correctness are presented in Algorithm 4 of [5]. Its operation is very similar to the parallel algorithm, except that X is now prescaled by $2^{(d+1)v}$ and the *reduce* values are now stored and used d PEs later. In addition, this algorithm is one PE cycle longer than the parallel algorithm due to the delay in *reduce* utilization. In fact, the parallel design is the $d = 0$-stage implementation of the quotient pipeline algorithm.

The 1-stage quotient pipelined scalable very high radix algorithm is shown in **Fig. 2**.

v: outer digit length, radix $= 2^v$
d: stages of delay
M: n-bit odd modulus
M': n-bit integer satisfying $(-MM') \bmod 2^n = 1$
$\tilde{M}$: $(M' \bmod 2^{v(d+1)})M$
n': $n + (d+1)v$, length of prescaled X
Y: n'-bit multiplicand
R: $2^{n'}$
R^{-1}: modular multiplicative inverse of R, $(RR^{-1}) \bmod M = 1$

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

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

$Z = 0$
$oldreduce = 0$
for $i = 0$ to f
  $reduce = Z^0$
  $Z = Z >> v + oldreduce \times \hat{M} + X^i \times Y$
  $oldreduce = reduce$
$Z = Z << v + oldreduce$

Fig. 2: Quotient Pipelined Algorithm

The idea that makes delaying reduce possible is the new method from which $\hat{M}$ is calculated. $\tilde{M}$ is now $d \times v$ bits longer, but later is right shifted by $d \times v$, resulting in $\hat{M}$ remaining the same length. But the value of $\hat{M}$ is now different, and can be multiplied with *reduce* d PEs later to yield the correct result. A rigorous proof is provided in [5].

The very high radix quotient pipelined algorithm can be easily converted into a scalable design show in Fig. 3 by iterating over w-bit words of Y and M:

w: inner word length

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

C: $(v+1)$-bit carry digit
All other notations are defined on Fig. 3

$Z = 0$
$oldreduce = 0$
for $i = 0$ to f
  $C = 0$
  $reduce = Z_0$
  for $j = 0$ to e
    $(C, Z^j) = (Z^j_{v-1:0}, Z^{j-1}_{w-1:v}) + oldreduce \times M^j + X^i \times Y^j + C$
  $oldreduce = reduce$
$Z = Z << v + oldreduce$

Fig. 3: Scalable Quotient Pipeline Algorithm

IV. HARDWARE IMPLEMENTATION

The overall hardware architecture of the scalable 1-stage quotient pipelined multiplier is similar to those presented in [1], [2], [3], and [6]. **Fig. 4** provides the overview architecture of a scalable Montgomery multiplier using p PEs. Every PE receives v bits of X and w bits of $\hat{M}$, Y, and Z on each step,

and also receives the $reduce \times \hat{M}$ product from two PEs earlier. In one kernel cycle, $p \times v$ 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 X scaling factor of $2^{(d+1)v}$ and the extra stage of *reduce* delay. A tristate bus allows the output of any PE to be written to the result.

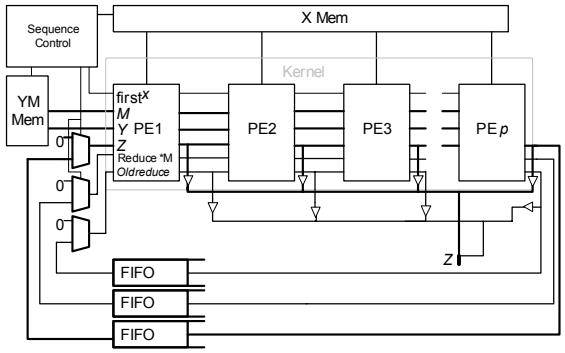

Fig. 4: Overall Scalable Montgomery Multiplier Architecture

A. Processing Elements

In comparison with the parallel algorithm PE diagram in Fig. 1, there are several significant changes in the new PE shown in Fig. 5. First, notice that the *reduce* CPA is no longer in the first cycle of the PE. The result from the last CSA is stored and resolved by the CPA on the next cycle. This result is now labeled as the *oldreduce* value to be used two PEs down the pipeline. In essence, two sequential tasks that used to be performed in one clock cycle are now distributed over two clock cycles. In addition, because the *reduce* and $\hat{M}$ bits for the *i*th PE is available at *i-1*th PE, the $reduce \times \hat{M}$ operation has been moved up one PE cycle to further reduce the critical path by removing a CSA. The final critical path of this PE design is a register, multiplier, and CSA, eliminating the CPA and one CSA from the critical path of the parallel design.

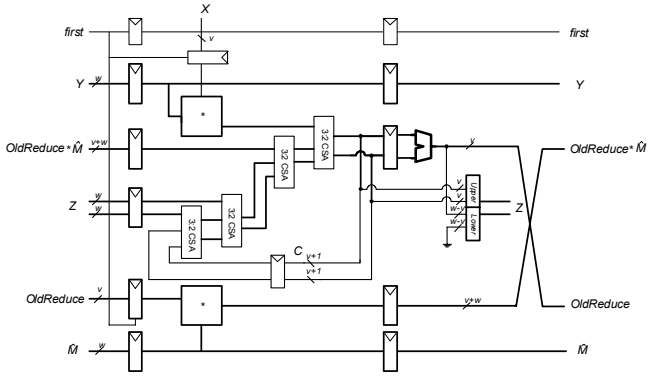

Fig. 5: Quotient Pipelined PE Architecture

B. Latency

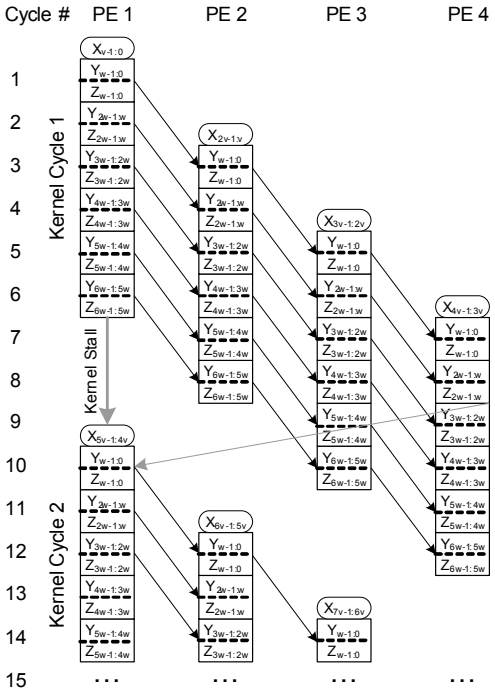

Fig. 6: Quotient Pipeline Latency Graph

The latency of the quotient pipeline design is similar to that of [1]. One PE cycle consists of two clock cycles. Each PE multiplies v bits of X with w bits of Y on the first cycle, then right shifts the result on the second cycle. When a PE has processed all the bits of Y, a kernel cycle has completed, and it will wait for a new set of X bits to start the cycle all over again. Modification introduced by the quotient pipelining algorithm shows no visible effect on the latency graph in Fig. 6.

An entire multiplication using p PEs takes n'/pv kernel cycles plus an additional partial cycle. When a PE has finished its kernel cycle, it cannot begin the next cycle until the last PE has completed the first word of Z. This leaves the PE in two possible situations when determining overall latency. Case I corresponds to a large number of words, e, relative to the number of processing elements, p. In this situation, when the first PE has finished its kernel cycle, the first word of Z from the last PE is already waiting in the FIFO, there is no stall between kernel cycles, and 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 wait until the last PE finishes calculating the first word of Z.

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 two-cycle latency between PEs. Thus, with p PEs, there is a $2p$ cycle delay before the first PE may begin processing again.

Therefore Case I occurs when $(e) \geq 2p+1$ 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

$$d_I = k(e) + (e) + 2(f \bmod p) \quad (2)$$

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

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

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

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

$$d_{II} = \frac{2n}{v} + \frac{n}{vp} + \frac{n+2v}{w} + 4 \text{ for } n \leq 2pw - v \quad (5)$$

V. Results

The quotient pipelined algorithm was coded in Verilog and synthesized onto a Virtex II XC2V2000-6 FPGA using Synplify Pro. A complete radix 2^{16} Montgomery multiplier unit with 16 PEs uses 3825 LUTs and 32 18×18 multipliers. The synthesized unit operates at 135.7 Mhz with a critical path through the multiplier and CSA, matching our prediction.

Table 1 compares the performance of the quotient pipelined design with that of previous scalable designs. Eliminating the CPA from the critical path gives the quotient pipelined design a 30% faster cycle time. Even though the quotient pipeline design requires a few more PE cycles to complete the overall multiplication, these extra clock cycles are outweighed by the faster clock. For 1024-bit modular exponentiation using 16 PEs, the quotient pipelined design is 20% faster than the parallel design.

Of course, the tradeoff of the quotient pipeline design is not only increase in PE cycles, but also increase in hardware and the overall complexity of design. Most of the temporary values within each PE are in redundant form and require extra registers and CSAs. More registers are also needed to store *oldreduce*. Even though these extra components do not affect the critical path of the design, they do increase the overall hardware requirement. Comparing to the parallel design, Table 1 shows a 45% increase in LUTs.

VI. Conclusions

This paper has shown an implementation of a scalable very high radix quotient pipelined Montgomery multiplier. This design improved upon methods used in [1] by removing the *reduce* dependency of successive PEs. As a result, we were able to remove a CPA and a CSA from the critical path of [1] at the cost of up to one extra kernel cycle. Synthesis results show that the quotient pipeline design clocks 30% faster than previous very high radix designs. It performs 1024-bit modular exponentiation in 5.1 ms using 16 PEs, 20% faster

TABLE 1: SYNTHESIS PERFORMANCE COMPARISON OF VARIOUS MONTGOMERY MULTIPLIERS

Description	Hardware	Technology	Clock Speed	Reference	256-bit time (ms)	1024-bit time (ms)
Quotient pipeline scalable radix 2^{16} 16 PEs x 16 bits	3825 LUTs + 32 mults + ~5*n* RAM	Xilinx Virtex II	135.7 MHz	This work	0.21	5.1
Quotient pipeline scalable radix 2^{16} 4 PEs x 16 bits	920 LUTs + 8 mults + ~5*n* RAM	Xilinx Virtex II	135.7MHz	This work	0.37	17.4
Parallel scalable radix 2^{16} 16 PEs x 16 bits	2608 LUTs + 32 mults + ~5*n* RAM	Xilinx Virtex II	106.3MHz	[1]	0.25	6.3
Parallel scalable radix 2^{16} 4 PEs x 16 bits	640 LUTs + 8 mults + ~5*n* RAM	Xilinx Virtex II	106.3 MHz	[1]	0.43	21.6
Scalable radix 2^{16} 16 PEs x 16 bits	2336 LUTs + 32 mults + ~5*n* RAM	Xilinx Virtex II	106.3 MHz	[3]	0.38	6.31
Scalable radix 2^{16} 4 PEs x 16 bits	584 LUTs + 8 mults + ~5*n* RAM	Xilinx Virtex II	106.3MHz	[3]	0.43	21
Improved radix 2 64 PEs x 16 bits	5598 LUTs + ~5*n* RAM	Xilinx Virtex II	144 MHz	[2]	1.0	16
Improved radix 2 16 PEs x 16 bits	1514 LUTs + ~5*n* RAM	Xilinx Virtex II	144 MHz	[2]	1.1	59

than the parallel design. However, the quotient pipelined design uses 45% more LUTs in each PE than the parallel design.

Removing the CPA from the critical path is the most significant modification of the quotient pipelined design. It allowed more even distribution of operations over the two clock cycles in each PE cycle. We are presently investigating other modifications of the parallel design to remove the CPA from the critical path without increasing the number of kernel cycles or the hardware requirements.

REFERENCES

[1] K. Kelly and D. Harris, "Parallelized Very High Radix Scalable Montgomery Multipliers," *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 1196-1200, 2005.

[2] D. Harris *et al*., "An improved unified scalable radix-2 Montgomery multiplier", *IEEE Symp. Computer Arithmetic*, pp. 172-178, 2005.

[3] K. Kelley and D. Harris, "Very high radix scalable Montgomery multipliers", *IEEE IWSOC Conference*, pp. 400-404, July 2005.

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

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

[6] 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.

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