Parallelized Radix-4 Scalable Montgomery Multipliers

Nathaniel Pinckney and David Money Harris[1]

[1] Harvey Mudd College, 301 Platt. Blvd., Claremont, CA, USA
e-mail: npinckney@hmc.edu

ABSTRACT

This paper describes a parallelized radix-4 scalable Montgomery multiplier implementation. The design does not require hardware multipliers, and uses parallelized multiplication to shorten the critical path. By left-shifting the sources rather than right-shifting the result, the latency between processing elements is shortened from two cycles to nearly one. Multiplexers are used to select precomputed products. Carry-save adders propagate carry bits before words are discarded. The new design can perform 1024-bit modular exponentiation in 9.4 ms and 256-bit exponentiation in 0.38 ms using 4997 Virtex2 4-input lookup tables, while consuming 30% fewer LUTs than a previous parallelized radix-4 design. This is comparable to radix-2 for long multiplies and nearly twice as fast for short ones.

Index Terms: Cryptography, RSA, Montgomery Multiplication.

1. 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. But, 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 will perform modular multiplication of Montgomery residues without the need of a division step. Hence, they can dramatically 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 [3, 2], radix-4 [4], radix-8 [5], radix-16 [6], and very high radix [7, 8]. A scalable radix-2^v design processes v bits of the multiplier and bits of multiplicand per step. The scalable very high radix designs commonly use dedicated $w \times v$ hardware multipliers. These multipliers are efficient on FPGAs containing high-speed multipliers, but may be undesirable for application-specific integrated circuits.

Conventional scalable Montgomery multipliers right-shift the result after each PE. This leads to two-cycle latency between PEs. By left-shifting the operands rather than right-shifting the result, the latency can be reduced to nearly one cycle at the expense of a small increase in the number of iterations through the PEs [3].

The critical path through a PE can be shortened by reordering the steps of the Montgomery multiplication algorithm, which parallelizes multiplications within the PE [9, 7, 6].

This paper improves a previous parallelized radix-4 design [13] by reducing the required hardware and presents a novel solution for carry bit propagation when using redundant form for partial words of the result. As with the previous design, it left-shifts operands and parallelizes multiplications within the PE. For short operands the multiplier is nearly twice as fast as radix-2 designs and for long operands it is comparable.

2. MONTGOMERY MULTIPLICATION

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. Hence, no information is lost during the reduction step.

The algorithm involves three dependent multiplications. Orup showed that it can be sped up by reordering steps and doing a precomputation, to eliminate one of the multiplications and to allow the other two to occur in parallel [9].

Note that we can also skip the normalization step for successive Montgomery multiplications because if $R > 4M$ and $X, Y < 2M$ then $Z < 2M$ [9, 10, 11, 12]. 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 $Z = Z - M$

Figure 1. Montgomery multiplication algorithm

A. Parallelized Radix-4 Scalable Design

The parallelized radix-4 scalable algorithm is a hybrid of two previous Montgomery multiplier designs: the improved unified scalable radix-2 design [3] and the parallelized very high radix scalable design [7]. Fig. 2 shows the parallelized radix-4 scalable algorithm derived by [9, 7]. Parallel radix 2^v algorithms require extending the operands by another v bits, so $n_1 = n + 3$ for radix 4. R also increases by 2^v. The variables are defined below.

n_1: $n + 3$
n_2: $n + 4$ (or larger; see Section III)
M: n-bit odd modulus
M': n_2-bit integer satisfying $(-MM') \bmod 2^{n_2} = 1$
$\hat{M}$: n-bit integer $[(M' \bmod 2^2) \times M + 1)/2^2$
Y: n_1-bit multiplicand
X: n_1-bit multiplier
C: 3-bit carry
w: scalable inner word length
f: outer loop length ceiling of $n_2 / 2$
e: inner loop length ceiling of n_1 / w

The precomputed $\hat{M}$ is used so that no multiplication is needed to calculate Q. The algorithm is scalable because it iterates over words of the operands using fixed-sized PEs. The superscripts denote 2-bit words for X and w-bit words for Y, Z, and $\hat{M}$. There are e w-bit words of Y, $\hat{M}$, and Z, and f 2-bit words of X in a radix-4 design with w-bit PEs

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

Figure 2. Parallelized radix-4 scalable Montgomery algorithm

3. HARDWARE IMPLEMENTATION

As Tenca proposed [2], the Montgomery multiplier is built from a systolic array of processing elements (PEs), as shown in Fig. 3. The architecture includes memories for X, Y, and $\hat{M}$, a FIFO to store partial words of Z and Q, and a sequence controller. The memory also holds precomputed $3\hat{M}$ and $3Y$ values for multiplications within the PEs. A FIFO holds results of the last PE until the first PE has completed processing the current operands. The FIFO has a latency of b (typically 1) cycles. Bold lines in the figure indicate variables in carry-save redundant form.

A. Processing Elements

The parallelized radix-4 processing element design is shown in Fig. 4. It is similar to the design from [13] but optimizes out two w-bit carry-save adders (CSAs).

Each PE receives a different 2-bit word of X and thus handles a different iteration of the outer loop of the radix-4 algorithm. The number of outer loop iterations typically exceeds the number PEs, thus the kernel pipeline may be used multiple times during a

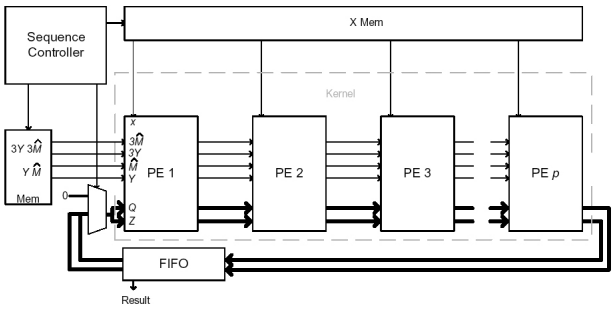

Figure 3. Scalable Montgomery multiplier architecture

single multiply. For a kernel with p PEs, $k = f/p$ pipeline cycles are needed to process all of X. The PEs also receive w bits of Y, $\hat{M}$, and Z in each clock cycle. Hence each PE requires e cycles to process all the iterations of the inner loop.

In one pipeline cycle, $2p$ bits of X are processed. Unlike a previous design [7], we ensure that the final result is always taken from the last PE in the kernel to simplify the hardware design. For this to be true, $n_2 = 2pk \geq n + 4$, where k is an integer number of pipeline cycles.

Z and Q are represented in carry-save redundant form for speed. Each PE contains two multiplexers to select the apprpropate multiples of Y and M, two 3:2 CSAs to add these multiples to Z, datapath registers (with control signals), and a feedback register for the carry between iterations of the inner loop. Because the CSAs do not have identical length operands, they are optimized into combinations of half adders and full adders to reduce the amount of hardware. Recall that X^i and Q^i can range from 0 to 3 for radix-4. It is trivial to compute $X^i \times Y^j$ or $Q^i \times \hat{M}^j$ when either multiplier is 2, because Y^j and $\hat{M}^j$ shift left by 1 bit. Likewise, when the multiplier is 0 or 1 the product is also trivial. When the multiplier is 3, computing the product in real-time would be costly. Instead of including multipliers in the PEs, precomputed $3Y^j$ and $3\hat{M}^j$ are stored in the memory and bussed to each PE, where multiplexers are used to select the product. The drawback of this is extra registers are added to accommodate $3Y^j$ and $3\hat{M}^j$ in the PE. Since Q is stored in redundant form, it must first be converted to non-redundant form, using an XOR, to select the appropriate multiple of $\hat{M}^j$. The multiplexer select lines drive a fanout of $w + 1$, so they must be buffered for adequate drive.

The PE shifts Y^j and $\hat{M}^j$ left by two bits instead of right by two after each step. This reduces cycle latency of the PE from two cycles to a single cycle by removing the dependence on the lower bits of the next Z^{j+1} word [3]. As with previous designs, the PE is pipelined for single-cycle throughput. Every $w/2$ steps the lowest word of Z is discarded because it is not in the final result. To simplify implementation, a word is always discarded in the FIFO between pipeline cycles. Hence, our design requires that $2p$ must be divisible by w.

So that X^i and Q^i and are constant for an entire pipeline cycle, *xptr* is asserted at the start of a pipeline cycle, to enable the X^i and Q^i registers. A shift register, outside of the PE, sequentially asserts PE *xptrs* as words transverse through the kernel. It similarly asserts *discard* to suppress the input registers and discard words of Z.

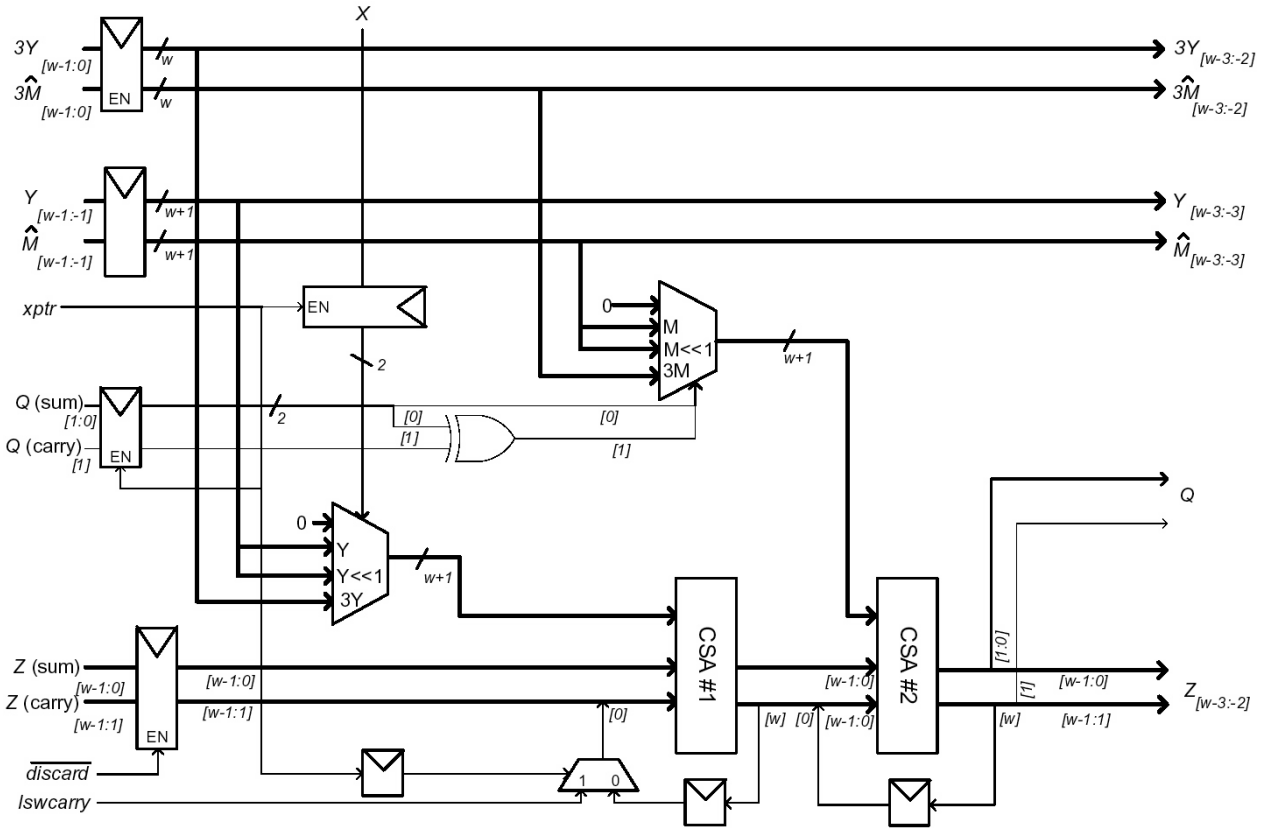

Figure 4. Parallelized Radix-4 scalable Montgomery multiplier processing element

For conventional Montgomery multiplications using M instead of $\hat{M}$, the discarded word is all zeros. However, in the parallel version, the word is usually non-zero. Since the result is stored in redundant form, conversion to non-redundant form could produce a carry-out. The FIFO discards a word, so logic in the FIFO calculates a carry out from the first word and outputs to the first PE's *lswcarry*. During the first word in each PE, the carry feedback registers are always zero. Hence, *lswcarry* is selected by a multiplexer and added to the first word by the first CSA. This logic is only included for the first PE.

Subsequent PEs rely on the CSAs to propagate carry bits of discarded words. If a least significant bit of an input to a 3:2 CSA is zero, the least significant bit of the result will always be in non-redundant form. This can be extended to the two PE CSAs. Fig. 5 shows a dot diagram of the first word Z^0 when $X^i \times Y^j$ and $Q^i \times \hat{M}^j$ have been shifted left by two bits during a previous step. Note that the two LSBs of the results are in non-redundant form. Because each PE contains two CSAs, two bits of Z^0 will be converted to non-redundant form after each step if the corresponding bits of $X^i \times Y^j$ and $Q^i \times \hat{M}^j$ are also zero. By delaying discarding of words between PEs by one step, the CSAs can be used to propagate carry bits. Hence, the *lswcarry* multiplexer is not needed except in the first PE and *lswcarry* does not need to be generated between PEs.

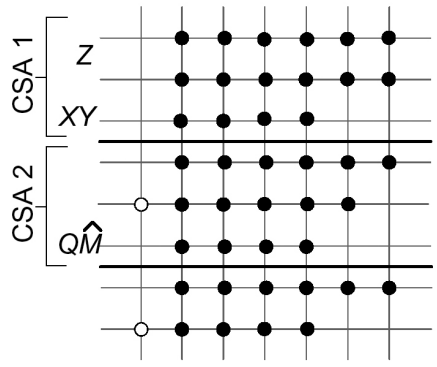

Figure 5. Dot diagram of first word of Z processed by CSAs

B. FIFO

The FIFO stores partial words Z^j and Q^i in non-redundant form to reduce FIFO memory size. Since the FIFO discards a word of Z^j, it must calculate the carry out *lswcarry* from the discarded word. The FIFO design includes a 2-bit CPA, feedback registers, multiplexers, AND gates, and memory, as shown in shown in Fig. 6.

In this design, the LSB of the Z redundant form carry word is always zero. Hence, instead of adding the two upper bits of sum and carry together to generate *lswcarry*, an AND gate is used because only the upper bit of each is in redundant form. The CPA converts Z^j to non-redundant form

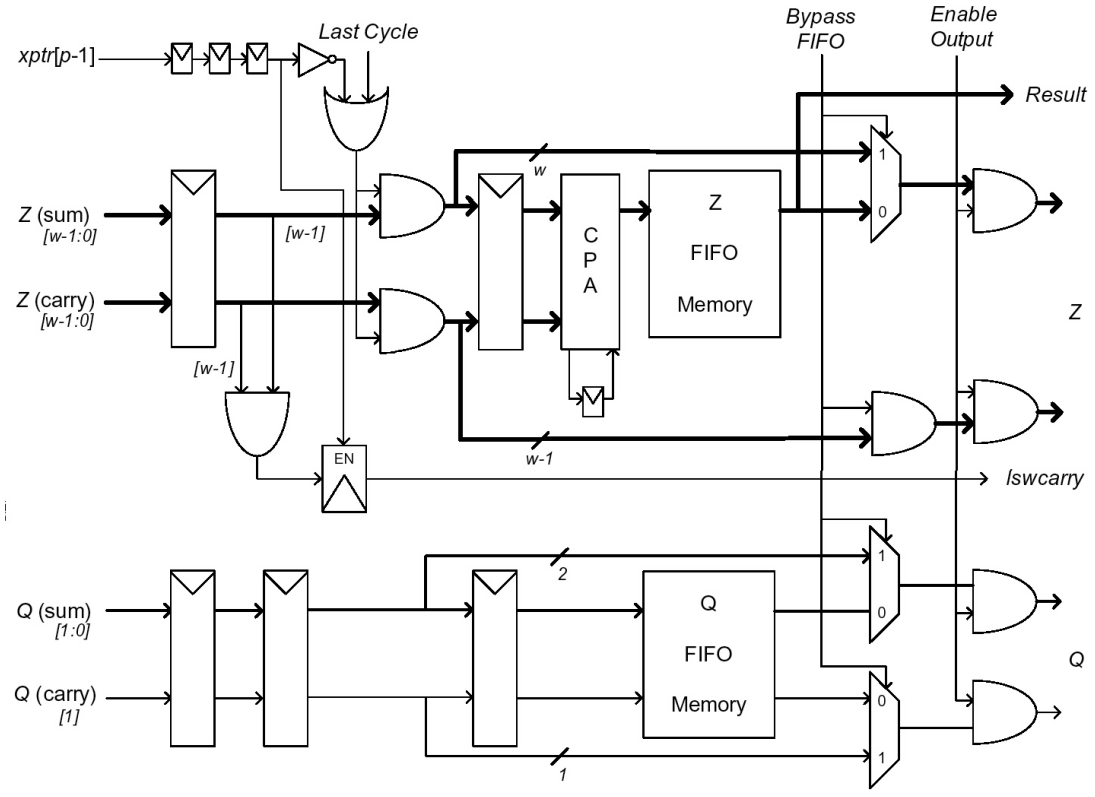

Figure 6. Parallelized Radix-4 scalable Montgomery multiplier FIFO

for the final result and for partial words stored in the FIFO memory. When *Bypass FIFO* is asserted, the FIFO memory is bypassed to minimize FIFO cycle latency to $b = 1$. A 2:1 multiplexer is used to select between the stored partial word and the current partial word. Since partial words in the FIFO memory are stored in non-redundant form, an AND gate is used instead of a multiplexer to zero the partial Z^j and Q^i carry words.

Two AND gates after the first set of flip flops are used to discard words of Z, except during the last cycle for the result. AND gates at the end of the FIFO suppress the output to the PEs during the first and last cycles.

C. Latencies

A hardware pipeline diagram of the radix-4 design is shown in Fig. 7. The diagram assumes the minimum FIFO cycle latency $b = 1$. Each PE completes a pipeline cycle in e cycles plus an additional cycle to handle overflow because Z is not shifted right after each step.

There are two cases for the time between pipeline cycles. Case I occurs when the PEs are used continuously and is $e + 1$ cycles per pipeline cycle. Case II occurs when the first PE must wait for the result from the last PE. For the first word to transverse through all of the PEs, lp cycles are needed, where l is the cycle latency between PEs (1 in this design). An additional vp/w are needed to handle discarded words of Z^j as Y^j and $\hat{M}^j$ are shifted. Lastly, an extra b cycles are needed due to latency through the FIFO. Therefore, Case II is $lp + vp/w + b$ cycles per pipeline cycle. The figure is simplified to not show vp/w extra cycles for Case II.

Because k pipeline cycles are needed to process all bits of X, $k(e + 1)$ total cycles are needed in Case I for Montgomery multiplication. Let T_c denote the clock cycle period. The total time for Montgomery multiplication is then

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

In Case II, total cycles are needed for Montgomery multiplication. Hence, the total Montgomery multiplication time is

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

Let $m = vwp$ be a metric for the amount of hardware in the multiplier. We can rewrite the above delays in terms of the design parameters n, w, v, p, l, and m, and the low order terms can be dropped, so that the approximate number of cycles is

$$d_1 = n_2/m \text{ for } n \geq lwp$$
$$d_2 = nl/v \text{ for } n < lwp$$

Hence, for Case I the time required for Montgomery multiplication decreases linearly with amount of hardware in the multiplier. For Case II, where operand lengths are short compared to the hardware available, the time does not change with the amount of hardware.

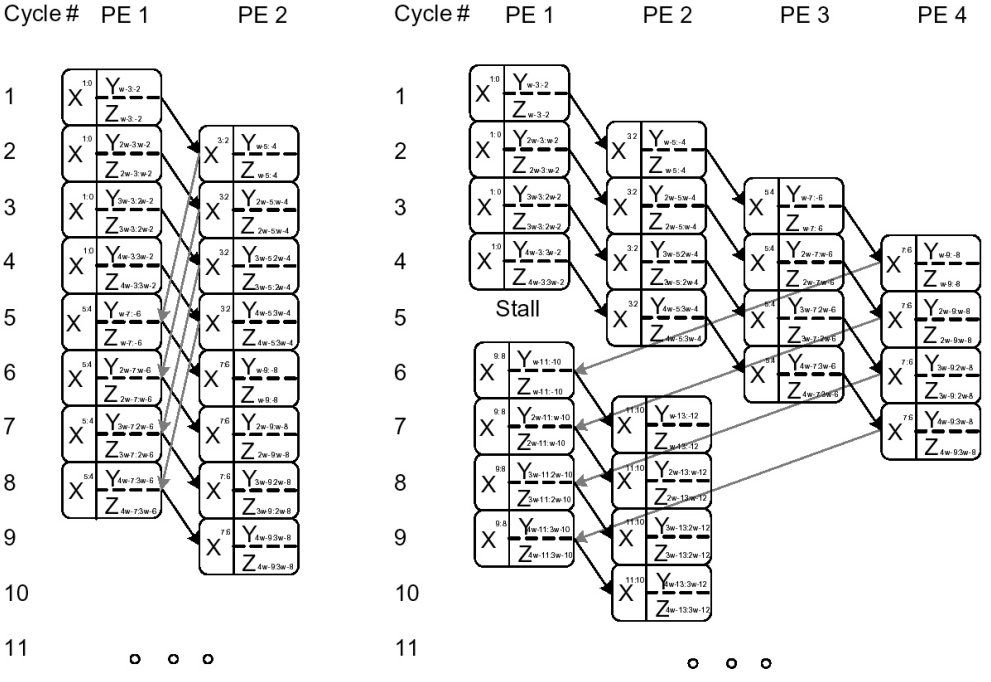

Figure 7. Hardware pipeline diagram for $e = 3$, $p = 2$ (Case I, left) and $p = 4$ (Case II, right)

4. RESULTS

The processing elements were coded in Verilog and simulated in ModelSim. The parallelized radix-4 design, along with previous 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. A comparison of the parallelized scalable radix-4 design with other Montgomery multiplier designs is shown in Table I. Clock speed for the new design was obtained by synthesizing kernel with $p = 2$. The critical path for the radix-4 design is identical to [13]. It includes multiplexer, a buffer, two CSAs, and a register, which limits the speed to 248 Mhz for $w = 16$. Compared to [13] this design consumes 30% fewer LUTs and comparable number of REGs.

A comparison of hardware usage and exponentiation time for the parallelized radix-4 design with others is shown in Table II. The data includes the hardware in the PEs and controller, but not the RAM bits or logic in the memories and FIFO. The modular exponentiation time was calculated by multiplying the time of a single Montgomery multiply with the number of multiplies, $2n + 2$, for a modular exponentiation.

The 32 PE parallelized scalable radix-4 design has the same number of full adder bits $m = vwp$ as a 64 PE radix-2 design. For $w = 16$, the radix-4 design includes 17% fewer LUTs and 12% fewer registers than the parallelized left-shifting scalable radix-2 design of [14]. It performs 256-bit modular exponentiation in 42% less time, but 1024-bit modular exponentiation in 18% more time. The short exponentiation ($n < lwp$) is part of Case II, in which the cycle count scales inversely with v, so we would expect about half the number of cycles. The long exponentiation ($n \geq lwp$) is part of Case I, in which the cycle count scales inversely with m, so we would expect comparable cycle counts. The cycle time is 22% slower for radix-4, so the total time is correspondingly longer. The parallelized radix-4 design also come at the expense of precomputing and storing $3Y$ and $3\hat{M}$.

In a system whose clock period is limited by other factors, the radix-4 design is clearly superior to radix-2 because it uses half the number of cycles for short operands and a comparable number for long operands, while requiring fewer LUTs and registers.

Table I. Comparison of FPGA resource usage and clock speed

Architecture	Reference	*w*	*v*	4-input LUTs / PE	Registers / PE	16 x 16 Mults / PE	Critical Path	Clock Speed (Mhz)
Parallel radix-4 scalable	This work	4	2	35	45	0	2CSA + BUF + MUX + REG	248
		8	2	66	69	0		
		16	2	132	120	0		
Parallel radix-4 scalable	[13]	16	2	187	121	0	2CSA + BUF + MUX + REG	266
Parallel radix-2 scalable	[14]	16	1	94	72	0	AND + 2CSA + BUF + REG	318
Improved radix-2 scalable	[3]	16	1	95	72	0	2AND + 2CSA + BUF + MUX + REG	285
Tenca-Koç radix-2 scalable	[2]	16	1	95	72	0	2AND + 2CSA + BUF + MUX + REG	285

Table II. Comparison of modular exponentiation times

Description	Ref	Technology	Freq. (MHz)	*w*	*v (radix-2^v)*	*p*	LUTs	REGs	*n*	*T* (ms)
Parallel radix-4 scalable	This work	Xilinx XC2V2000-06	248	16	2	16	3079	2048	256	0.35
									1024	18
						32	4997	4051	256	0.38
									1024	9.4
						64	8428	7543	256	0.45
									1024	5.4
Parallel radix-4 scalable	[13]	Xilinx XC2V2000-06	266	16	2	16	2996	1955	256	0.33
									1024	17
						32	5916	3863	256	0.36
									1024	8.7
						64	11684	7642	256	0.42
									1024	5.1
Parallel radix-2 scalable	[14]	Xilinx XC2V2000-06	318	16	1	16	1575	1189	256	0.52
									1024	28
						64	6006	4597	256	0.66
									1024	8.0
Improved radix-2 scalable	[3]	Xilinx XC2V2000-06	285	16	1	16	1408	1205	256	0.55
									1024	30
						64	6317	4844	256	0.62
									1024	8.4

5. CONCLUSIONS

This paper describes a parallelized scalable radix-4 Montgomery multiplier design that reduces PE cycle latency and critical path length. The design is suitable for situations where a dedicated hardware multiplier is undesirable or not available. The design performs 42% faster or half as many clock cycles as previous radix-2 designs for small multiplies, but about 18% slower for large multiplies because the cycle time is longer. The kernel uses 12% fewer REGs and 17% fewer LUTs compared to a radix-2 kernel. This design improves upon a previous parallelized radix-4 design [13] by using 30% fewer LUTs and comparable REGs in each PE for $w = 16$.

Future radix-4 designs could use Booth encoding instead of precomputed $3Y$ and $3\hat{M}$ values. This might deliver similar performance with fewer registers needed to store precomputed values. FPGA synthesis results for Montgomery multiplier architectures are compared in this paper. Future research could compare ASIC implementations of these architectures.

ACKNOWLEDGEMENTS

The authors thank the Clay-Wolkin Family Foundation fellowship and Intel Circuit Research Lab for funding the research.

REFERENCES

[1] P. Montgomery, "Modular multiplication without trial division," *Math. of Computation*, vol. 44, no. 170, pp. 519-521, April 1985.
[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] 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, Lecture Notes in Computer Science, No. 1717, pp. 189-206, Springer, Berlin, Germany.
[6] 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.
[7] K. Kelley and D. Harris, "Parallelized very high radix scalable Montgomery multipliers," *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 1196-1200, Nov. 2005.
[8] K. Kelley and D. Harris, "Very high radix scalable Montgomery multipliers," *Proc. 5th Intl. Workshop on System-on-Chip*, pp. 400-404, July 2005.
[9] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," *Proc. 12th IEEE Symp. Computer Arithmetic*,pp. 193-199, July 1995.
[10] T. Blum and C. Paar, "High-radix Montgomery multiplication on reconfigurable hardware," *IEEE Trans. Computers*, vol. 50, no. 7, July 2001, pp. 759-764.
[11] C. Walter, "Montgomery exponentiation needs no final subtractions," *Electronics Letters*, vol. 35, no. 21, pp. 1831-1832, 14 October 1999.
[12] 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.
[13] 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.
[14] N. Jiang and D. Harris, "Parallelized Radix-2 Scalable Montgomery Multiplier," *IFIP Intl. Conf. on VLSI*, 2007.