Parallelized Booth-Encoded Radix-4 Montgomery Multipliers

Nathaniel Pinckney, Philip Amberg, and David Money Harris
Harvey Mudd College, Claremont, CA, USA
npinckney@hmc.edu

Abstract— **This paper proposes two parallelized radix-4 scalable Montgomery multiplier implementations. The designs do not require precomputed hard multiples of the operands, but instead uses Booth encoding to compute products. The designs use a novel method for propagating the sign bits for negative partial products. The first design right shifts operands to reduce critical path length when using Booth encoding. The second design left shifts operands to improve latency between processing elements and to decrease hardware usage. An FPGA implementation of the right-shifting design consumes 17% more lookup tables (LUTs) and 25% to 33% more flip-flops than a comparable non-Booth encoded design. It performs 1024-bit modular exponentiation in 9.1 ms using 5959 LUTs and 5079 flip-flops. The left-shifting design consumes 3% fewer LUTs and 29% to 33% fewer REGs than non-Booth. Its clock speed is 25% slower than non-Booth, and it performs 1024-bit modular exponentiation in 13 ms using 4852 LUTs and 2887 flip-flops.**

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], radix-8 [7], radix-16 [8], and very high radix [9, 10]. A scalable radix-2^v design processes v bits of the multiplier and w bits of the multiplicand per step.

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 [11, 9, 8]. The authors recently proposed a radix-4 multiplier [5, 6] that extends the parallelized radix-2 design of [12] to consume twice as many bits of the multiplier per step. For short operands the multiplier was nearly twice as fast as the radix-2 design and comparable for long ones. However, this design required precomputed $3Y$ and $3\hat{M}$ multiples, which requires a carry propagate adder (CPA). Another previous radix-4 design [4] uses Booth encoding [13] to eliminate these multiples, but it did not use the parallelization technique.

This paper improves the Booth-encoded radix-4 design [4] with parallelization to shorten the critical path. Two designs are presented, one right-shifting and one left-shifting. As compared to [6], the left-shifting design avoids the need for precomputed multiples and has comparable performance for long multiplications. However, the right-shifting is slower for short multiplications and requires more flip-flops to pipeline the results and more LUTs for Booth encoding. The left-shifting design has a much longer critical path, resulting in slow performance. But, it consumes comparable LUTs and fewer flip-flops and thus may be beneficial if clock speed is externally constrained.

II. BOOTH ENCODING

Consider a multiplication $X\times Y$, where X and Y are an arbitrary length. For a radix-4 multiplication algorithm, 2 bits of X are consumed per step. The possible multiples of Y per step are then 0, Y, $2Y$, or $3Y$. Without precomputed multiples, the only available multiples are 0 and $1Y$. Computing $2Y$ is a trivial shift, but $3Y$ traditionally requires an adder to sum Y and $2Y$. Booth encoding is a technique to compute the $3Y$ without an additional adder by using negative partial-products [12]. Note that on the next step of radix-4 multiplication, a multiple of Y becomes $4Y$ because the next two bits of X are being processed. Observe that

$$3Y = 4Y - Y$$

and hence a $3Y$ multiple may be generated with a $-Y$ in the

TABLE I. RADIX-4 BOOTH ENCODING

x_{2i+1}	x_{2i}	x_{2i-1}	Partial Product
0	0	0	0
0	0	1	Y
0	1	0	Y
0	1	1	2Y
1	0	0	-2Y
1	0	1	-Y
1	1	0	-Y
1	1	1	0

current step and a $4Y$ in the next step. Booth encoding, summarized in Table I, examines the current bits of X and the previous most significant bit of X to add negative and extra multiples of Y, avoiding hard $3Y$ multiples. The index i refers to the current two-bit word of X.

III. Montgomery Multiplication

The section below 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. 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 [10].

Note that we can also skip the normalization step for successive Montgomery multiplications because if $R > 4M$ and X, $Y < 2M$ then $Z < 2M$ [10, 11, 12, 13]. To do this we increased the size of the operands to $n_1 = n + 1$ bits and let $R = 2^{n_2}$, where $n_2 = n + 2$.

A. Parallelized Radix-4 Scalable Booth Algorithm

The parallelized radix-4 scalable Booth algorithm, shown in Fig. 2, is a modification of the parallelized radix-4 scalable non-Booth algorithm [5, 6]. Parallel radix-2^v algorithms require extending the operands by another v bits. The variables are defined below.

- n_1: $n + v + 1$
- n_2: $n + v + 2$ (or larger; see Section IV)
- 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 $\lceil n_1 / w \rceil$

As with the previous radix-4 design, 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, $\hat{M}$, and Z. There are

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

$$
\begin{aligned}
& Z = 0 \\
& \text{for } i = 0 \text{ to } f-1 \\
& \quad Q^i = Z^0 \bmod 2^2 \\
& \quad Q'^i = Booth(Q^i, Q_1^{i-1}) \\
& \quad X'^i = Booth(X^i, X_1^{i-1}) \\
& \quad C = 0 \\
& \quad \text{for } j = 0 \text{ to } e-1 \\
& \quad\quad (C, Z^j) = (Z_{1:0}^{j+1}, Z_{w-1:2}^j) + C + Q'^i \times \hat{M}^j + X'^i \times Y^j \\
& \text{If } Q_1^{f-1} == 1 \text{ then} \\
& \quad C = 0 \\
& \quad \text{for } j = 0 \text{ to } e-1 \\
& \quad\quad (C, Z^j) = Z^j + C + 4\hat{M}^j
\end{aligned}
$$

Fig. 2. Parallelized radix-4 scalable Booth algorithm

$e = \lceil n_1 / w \rceil$ w-bit words Y, $\hat{M}$, and Z, and $f = \lceil n_2 / 2 \rceil$ 2-bit words of X in a radix-4 design with w-bit PEs.

Encoding is indicated by the Booth() function. A $4\hat{M}$ 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_1^{f-1} is the $n_2 = n + 4$ bit of X, which will always be zero.

IV. Hardware Implementation

As Tenca proposed [2], the scalable 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 products, and a sequence controller. The FIFO holds results of the last PE until the first PE has completed processing the current operand. Like previous designs [5, 6, 12], this architecture needs to propagate words of Q between PEs and in the FIFO. Bold lines in Fig. 3 indicate variables in carry-save redundant form. Only the large operand buses are drawn between PEs. The last $4\hat{M}$ multiple is added in the FIFO before storing the result.

The authors have previously used left-shifting [5, 6] to reduce the cycle count for short multiplies and to save pipeline flip-flops. With Booth encoding, this technique adds complexity because Z is stored in redundant form, as explained below.

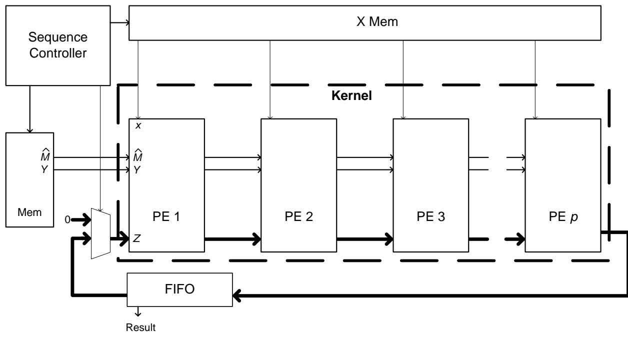

Fig. 3. Parallelized radix-4 architecture

A. *Right-shifting Processing Element*

The parallelized radix-4 right-shifting Booth processing element design is shown in Fig. 4. Each PE receives a different 2-bit word of X, so each PE is assigned a different iteration of the outer loop of the radix-4 algorithm. For a kernel with p PEs, $k = \lceil f / p \rceil$ 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 iterations of the inner loop.

In one pipeline cycle, $2p$ bits of X are processed. Like the previous designs [5, 6, 12], 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.

Booth encoding is implemented with a pair of Booth encoders and Booth selectors. Booth encoders take the multiplier's current word and the most significant bit of the multiplier's previous word, and output control signals to the Booth selector. In our implementation, the control signals are positive (Y or $2Y$), negative ($-Y$ or $-2Y$), and double ($2Y$ or $-2Y$). If the partial product is to be $0Y$, positive and negative are both 0. The Booth selector is essentially a 5-input multiplexer with inverters on two of its inputs, used to select between $-2Y$, $-Y$, 0, Y, or $2Y$. A negative partial product is in two's complement form, so a one must be added to the LSB of the inverted and shifted multiplicand. This is done in the carry-save adders (CSAs), and will be described below. We label the negative signals S_X and S_Q, for the $X^i \times Y^j$ and $Q^i \times \hat{M}^j$ partial products, respectively.

Each PE contains two w-bit Booth selectors, two w-bit 3:2 CSAs, 8 w-bit registers, a 2-bit CPA, two Booth encoders, and a small amount of other logic. Z is represented in carry-save redundant form for speed. Because this algorithm shifts Z right two bits before each step, there is a two-cycle latency within each PE to allow the next word of Z to be shifted into the two MSBs of the current word. This doubles the number of registers from the $4w$ that would be required for a left-shifting design. Because Z^j is stored in redundant form, the two least significant bits that are discarded after each right shift may produce a carry-out. A CPA is used to compute the carry-out H. It also converts Q to non-redundant form for the Booth encoder. The CPA and Booth encoders are moved ahead of a pipeline register to reduce the critical path.

The 3:2 CSAs are used to add partial products and carries to the shifted Z^j. There are three more bits we must add in to the LSB of Z^j: the carry-out H from the CPA and the two bits for the partial products to be in two's complement form, S_X and S_Q. The carries in the feedback registers will always be zero for the first words. Hence, we can multiplex the extra bits into the carry positions. But, there are only two CSAs and consequently only two carry spots, while we need three; this was a key challenge in this design. A nonparallelized design [4] was able to precompute S_Q into $\hat{M}^j$, because $\hat{M}^j$ was always odd. However, this is not possible in the parallelized design with the modified modulus, which may be even. Instead we send S_Q to the next PE, which then propagates it through the carry in of the CPA. The CSAs then add in S_x and H for the first word and the carries, labeled C_X and C_Q, for the subsequent words. Fig. 5 is a dot diagram summarizing this.

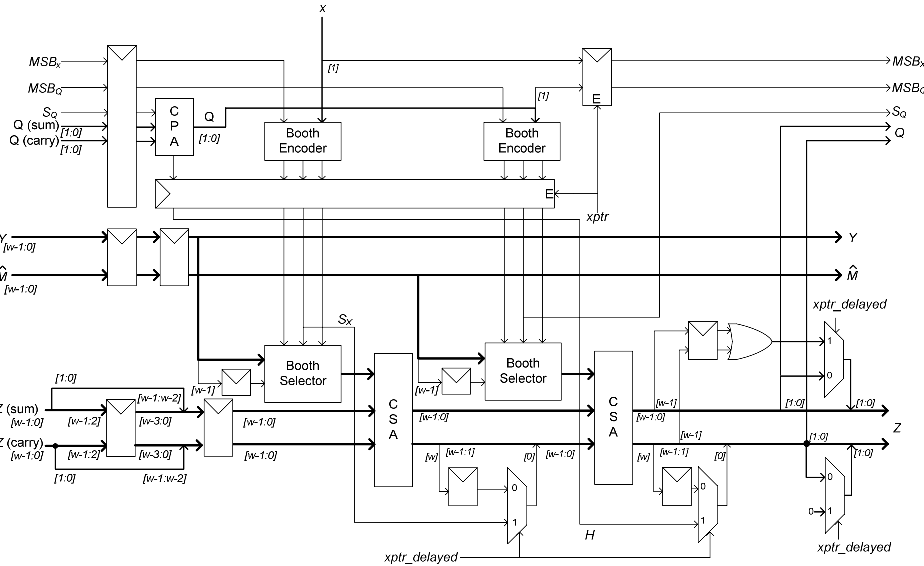

Fig. 4. Radix-4 right-shifting Booth Montgomery multiplier processing element

Since S_Q is added in the next PE, after all iterations have completed the FIFO must add S_Q to Z before storing the result.

Before Z is output to the next PE, it is shifted right and sign extended. As shown in the bottom right of Fig. 4, an OR gate computes the sign of Z by summing the most significant bits of sum and carry. A multiplexer sets or resets the least significant two bits of Z during the last word in a pipeline cycle. The least significant two bits of Z will then be shifted right into the two upper bits in the next PE.

xptr is a control signal from a shift register, which goes high for the first word of the pipeline cycle. It is used for enable inputs on some of the flops, and as select signals to the sign extension multiplexers. *xptr_delayed* is *xptr* delayed by one cycle. Because xptr comes from a shift register, *xptr_delayed* does not consume extra registers.

B. *Left-shifting Processing Element*

The parallelized radix-4 left-shifting Booth processing element is shown in Fig. 6. It is similar to the right-shifting design, but removes the registers needed for two cycle latency.

In left-shifting, the lowest word of Z needs to be discarded when it is no longer used. Therefore, it must be converted to non-redundant form to determine if a carry is generated before discarding it. [6] did this by relying on CSAs to propagate carries within words. In this design, carries are propagated with the hidden bit H, which uses a CPA to process two bits of Z into non-redundant form per step.

As the operands are shifted left, their least significant bits become zero. As a result, the effective LSB changes, creating problems for Booth encoding, and carry injection. For negative multiples ($-Y/-\hat{M}^{j}$ or -2$Y/-2\hat{M}^{j}$), the LSBs are inverted, making them all 1. Since the 2 effective LSBs of Z are zeroed each cycle, adding a negative multiple to Z will cause the actual LSBs of Z to be non-zero, preventing the discarding of the word. This is fixed by adding AND gates and multiplexers to select which bit is treated as the LSB of Z. The index *pos* in the figure indicates the effective LSB, which is PE dependent.

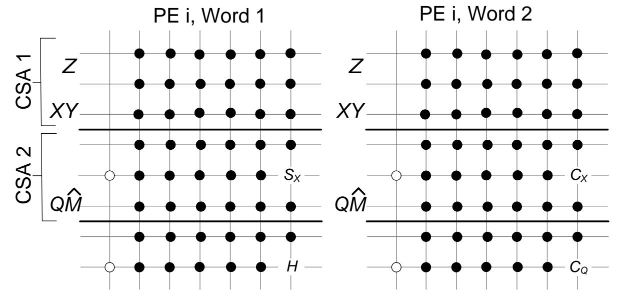

Fig. 5. Dot diagram of intermediate results in processing elements

Thus, S_Q, S_X, and H are added in the same effective positions as the right-shifting design except shifted left by *pos* bits.

AND gates at the output of the Booth selectors zero any gates below the LSB for the first word. Similarly, AND gates for Z at the input to the first CSA reset the two bits below *pos* so that carries are not added twice, once from H and once from the CSAs. *discardword* is similar to *xptr_delayed*, except it occurs the cycle before *xptr* instead of the cycle after and only on PEs which discard words of Z.

C. *Latencies*

There are two cases for the time between pipeline cycles. Case I occurs when the PEs are used continuously, thus e cycles per pipeline cycle for right shifting. 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 (2 for right shiting, 1 for left shifting). Lastly, an extra b cycles are needed due to latency through the FIFO. Therefore, Case II is $lp + b$ cycles per pipeline cycle in right shifting.

Fig. 7 shows a hardware pipeline diagram of the right-shifting design. The diagram assumes the minimum FIFO cycle latency $b = 1$. Case I is shown on the left, with $e = 4$ and $p = 2$. Case II is shown on the right, with $e = 4$ and $p = 4$.

Because k pipeline cycles are needed to process all bits of

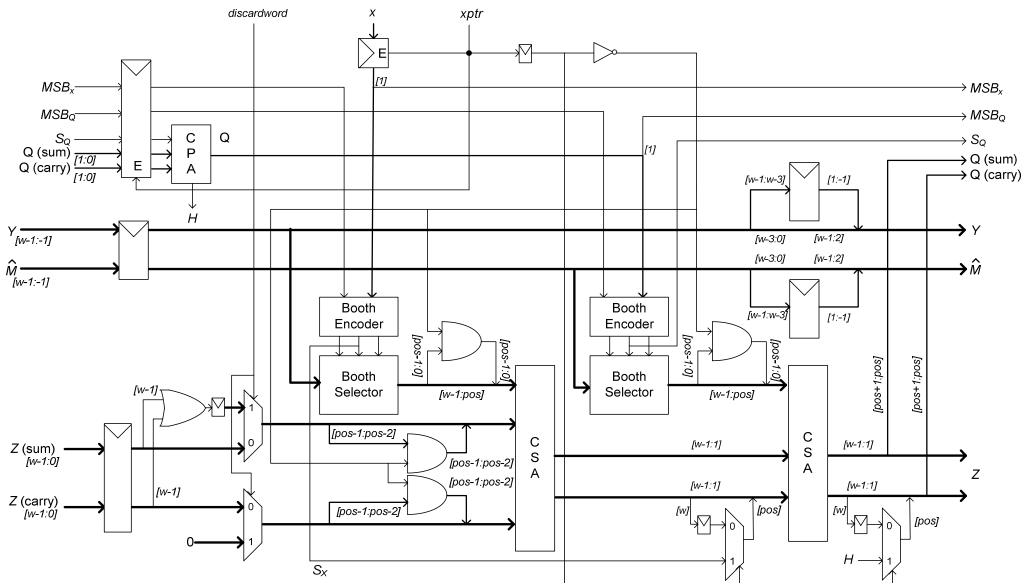

Fig. 6. Radix-4 left-shifting Booth Montgomery multiplier processing element.

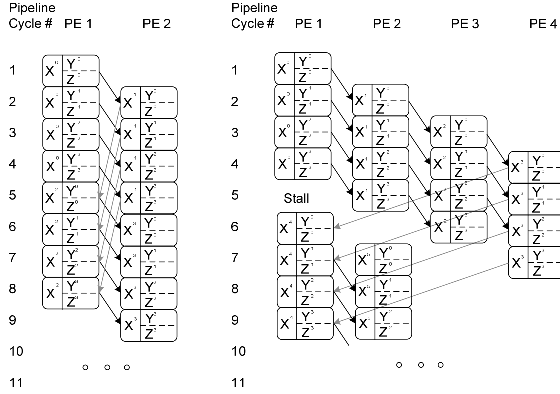

Fig. 7. Pipeline diagram for right shifting parallelized radix-4 Booth processing element

X, $k(e)$ total cycles are needed in Case I for Montgomery multiplication. Let T_c denote the clock cycle period. The total time for Montgomery multiplication for right shifting is then

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

In Case II, $k(lp + b)$ total cycles are needed for Montgomery multiplication. Hence, the total Montgomery multiplication time for right shifting is

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

Let $m = vwp$ be a metric for the amount of hardware in the multiplier [5, 6]. We can rewrite the above delays in terms of the design parameters n, w, v, p, l, and m, and drop the low order terms, so that the approximate number of cycles is

$$d_1 = n^2 / m \quad \text{ for } \ n \geq lwp$$
$$d_2 = nl / v \quad \text{ for } \ n < lwp$$

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 reaches a minimum beyond which no more hardware helps.

In the left-shifting design, an extra word is needed for overflow since the lower word of Z is not discarded immediately. Thus if the PEs are used continuously, $e + 1$ cycles are needed per pipeline cycle. In Case II, an extra vp/w cycles are needed for discarding words of Z. Thus the number of cycles per pipeline cycle is $lp + vp/w + b$. The total Montgomery multiplication times for left-shifting are then

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

V. Results

The processing elements were coded in Verilog and simulated in ModelSim. Verilog for the parallelized radix-4 Booth 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 radix-4 scalable Booth designs with other Montgomery multiplier designs is shown in Table II. Critical paths were obtained by synthesizing kernel with $p = 2$. The critical path for the right-shifting design is an inverter and 5-input multiplexer (for the Booth selector), two CSAs, a 2-input multiplexer, and a register. This limits the clock speed to 248 MHz for $w = 8$ and 16. The critical path for the left-shifting design adds a booth encoder, multiplexer, and an AND gate. This decreases the clock speed to 186 MHz for $w = 16$.

A comparison of hardware usage and exponentiation time for the parallelized radix-4 Booth designs with others is shown in Table III. 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 modular exponentiation time is the time for $2n$+2 Montgomery multiplies.

When the parallelized radix-4 scalable right-shifting Booth design is compared to the non-Booth design, there are 19% more LUTs and 25% more REGs in the $w = 16$, $p = 32$ case. The non-Booth designs [5, 6] had $4w$ registers to hold Y, $\hat{M}$, $3Y$, and $3\hat{M}$, and $2w$ registers to hold Z in redundant form, for a total $6w$ registers (plus some overhead). Because of the extra cycle of latency to shift Z right, the Booth design needs a pair of $4w$ registers for Y, $\hat{M}$, and Z, for a total of $8w$ registers. Therefore, for large w, the right-shifting Booth design using $8w/6w$ = 1.33 or 33% more registers than non-Booth. Its clock speed is identical to non-Booth.

The left-shifting design has $4w$ registers and approximately 33% fewer registers than non-Booth. In the $w = 16$, $p = 32$ case, it consumes 3% fewer LUTs and 29% fewer registers than non-Booth. However, the left-shifting Booth design's clock speed is 25% slower than non-Booth because of the longer critical path.

The right-shifting Booth design can complete a long (1024-bit) exponentiation in 9.1 ms, about the same speed as non-Booth. For a short (256-bit) exponentiation, the right-shifting Booth design needs twice the amount of time compared to the left-shifting non-Booth design because of the two cycle latency within PEs. The left-shifting design's long exponentiation time increases to 13 ms because of slower clock speed. However, its short exponentiation time decreases to 0.52 ms because of the one cycle latency within PEs. This is still slower than the 0.38 ms time for non-Booth, but it consumes less hardware.

VI. Conclusions

This paper described two parallelized scalable radix-4 Montgomery multiplier designs that uses Booth encoding to eliminate the precomputed $3Y/3\hat{M}$ multiples of the previous parallelized radix-4 design [5], making the design beneficial for modular exponentiation implementations. Left-shifting and right-shifting Booth designs were compared. Because of the parallelized design, previous methods [4] of adding sign bits used for Booth encoding do not work. Instead, the design propagates a bit using the carry-in of a 2-bit CPA. Right-shifting decreases the critical path length when compared to [4], but increases the number of flip-flops used when compared to [6]. The design consumes about 19% more LUTs and 25%

TABLE II. 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-4 scalable Booth	This work	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	This work	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	[12]	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

TABLE III. COMPARISON OF MODULAR EXPONENTIATION TIMES AND TOTAL FPGA RESOURCE USAGE

Architecture	**Ref.**	**Freq**	**Shift Dir.**	***w***	***v***	***p***	**LUTs**	**REGs**	***n***	***T* (ms)**
Parallel radix-4 scalable Booth	This work	248	R	16	2	16	3069	2640	256	0.62
									1024	18
						32	5959	5079	256	0.68
									1024	9.1
						64	11853	10292	256	0.80
									1024	9.7
Parallel radix-4 scalable Booth	This work	186	L	16	2	16	2469	1458	256	0.47
									1024	24
						32	4852	2887	256	0.52
									1024	13
						64	9637	5690	256	0.60
									1024	7.2
Parallel radix-4 scalable non-Booth	[6]	248	L	16	2	32	4997	4051	256	0.38
									1024	9.4
Parallel radix-2 scalable	[12]	318	L	16	1	64	6006	4597	256	0.66
									1024	8.0
Improved radix-2 scalable	[3]	285	L	16	1	64	6317	4844	256	0.62
									1024	8.4

to 33% more registers, and performs modular exponentiations in comparable time to a non-Booth encoded radix-4 design [6] for long multiplies. Left-shifting uses comparable LUTs and 29% to 33% fewer flip-flops, but is 25% slower, than non-Booth. If cycle time is limited by other system elements, the left-shifting radix-4 Booth-encoded design offers the lowest hardware cost and requires no precomputed multiples while offering latency comparable to the best alternative.

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] 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] 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.

[8] 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.

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

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

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

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

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

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

[15] C. Walter, "Montgomery exponentiation needs no final subtractions," *Electronics Letters*, vol. 35, no. 21, pp. 1831-1832, 14 October 1999.

[16] 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.