Improved Montgomery Multiplication

Trenton J. Grale
Diligentia Technology LLC
Austin, TX USA
trenton.grale@diligentiatech.com

Earl E. Swartzlander, Jr.
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX USA
eswartzla@aol.com

***Abstract*—The Montgomery multiplication algorithm is used to perform multiplication coupled with modular reduction without the need to employ a division operation. Serial Montgomery implementations may operate at a bit, digit, or word level. An established classification scheme for serial implementations considers two dimensions: whether the multiplication and reduction computations are separated or integrated, and whether operand or product words are prioritized for scanning. Presented here is an augmented version of the taxonomy, which adds a third dimension. The new dimension characterizes the degree of parallelism in performing low-level (bit or digit) computations. Introducing a small degree of bit or digit level parallelism to a formerly serial approach can enhance performance, both through the typical benefit of parallel computation, and by opportunistically avoiding unnecessary computations. This can be achieved for a modest incremental cost in increased area in lieu of the larger cost of realizing a fully parallel solution. In this way, a new region on the latency-area curve becomes available for tradeoff considerations. The novel Rescheduled Montgomery Multiplier is presented as an experimental realization of the augmented taxonomy.**

Keywords—modular arithmetic, modular multiplication, digit multiplication, Serial Montgomery Model, Rescheduled Montgomery Multiplier

I. INTRODUCTION

Modular arithmetic with large operands is employed in common public key cryptographic systems such as RSA and ECC. Operations such as addition, subtraction, multiplication, division, and exponentiation are performed, followed by computing the modulus function on the result with respect to some pre-selected modulus, which is often a prime number. The Montgomery multiplication algorithm replaces division by the arbitrary modulus M with division by a power of two, which is simply a right shift [1]. The key to this method is the prior transformation of the field elements into *M-residues* in what is referred to as the *Montgomery domain*. Algorithm 1 lists the steps in computing a Montgomery product.

Algorithm 1. Montgomery Multiplication [1].
Input: A, B
Output: $P = ABR^{-1} \bmod M$

1. $T = AB$ $\quad T_0 = T \bmod R$
2. $Q = T_0M'$ $\quad Q_0 = Q \bmod R$
3. $U = Q_0M$
4. $P = (T + U) / R$
5. if $(P > M)$:
6. $\quad P = P - M$
7. end if
8. return P

Over the preceding three decades, a variety of Montgomery multiplier implementations have been proposed. These implementations have ranged from software to hardware, and have included bit-serial, word-serial, and fully parallel designs.

Ordinary multiplication can be implemented with iterative algorithms that operate on smaller portions of the operands. Operands may be split into multiple words or digits. Then, the arithmetic components can be smaller but operate in a reduced cycle time. Taken to the extreme, operands may be processed at the bit level. Bit implementations can have a bare minimum of complexity and extremely short cycle time, at the cost of performing a large number of iterations. Where operands are divided into k digits, a total of k^2 digit multiplications must be performed in computing the product.

This paper is organized as follows. Section II reviews prior serial Montgomery architectures and a widely employed taxonomy for characterizing them. Section III proposes an extension to the taxonomy. Section IV describes a novel algorithm and hardware architecture for optimizing Montgomery multiplication. Section V covers experimental results. Finally, Section VI draws conclusions.

II. RELATED WORK

A. Previous Serial Architectures

H. Eberle, *et al.* describe a serial Montgomery architecture that operates at the digit level [2]. Large operands are split into k digits of size d, where $k = \lceil n/d \rceil$. A single digit multiplier builds up digit-word partial products sequentially. Reduction is interleaved with partial product row computation. This approach might be termed Montgomery Micro Reduction.

Großschädl, *et al.* present a bit-word serial multiplier [3]. This architecture operates on single bits of operand A (denoted as a_i) and the entire full-width operand B to compute bit-word partial products. A partial product a_iB can simply be computed by a row of n AND gates. It is accumulated in place in a register with the previous bit-word product. Montgomery reduction is performed on each iteration of the running partial product P_i. The quotient term is a single bit q_i, which is used to gate the modulus as q_iM for the reduction step. No computation is

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

required to generate q_i, because it can be shown mathematically to resolve to bit 0 of the current partial product word P_i before reduction.

An architecture presented by A. F. Tenca and Ç. K. Koç employs a simple bit-digit multiplier [4], [5]. It computes a sequence of bit-digit products to form a bit-word product. Reduction occurs concurrently. Subsequent bit-word products may be computed in a cascaded block or in the same block if the previous bit-word product is fed back.

B. Koç Serial Montgomery Model

A taxonomy proposed by Koç, *et al.* is used to characterize serial Montgomery algorithms [6]. Most research on Montgomery multiplication references it and classifies proposed architectures into one of its categories [3], [7], [8], [9], [10], [11], [12], [13], [14], [15]. The taxonomy considers two dimensions. The first dimension is reduction mode, and specifies whether reduction is separated from or integrated with partial product generation. Integrated reduction is further subdivided into coarse and fine. The second dimension is digit scanning priority, and specifies whether operand digits or product digits are prioritized for scanning.

Algorithm 1 represents a trivial example of separated reduction. The initial product T is first computed in full. Subsequent computations of Q and U are then performed to effect the reduction. Conversely, in integrated reduction, a partial product is computed, and then a Montgomery reduction is performed on it. This process continues for each partial product, so that a series of partial products and reductions is performed and the results accumulated into one final Montgomery product.

Digit scanning may prioritize the digits of either the input operands or of the product. The choice of priority affects the order in which digits are read from the input operands and the order in which product digits are written. If input operands are given priority (operand scanning), then they are scanned in a regular order. A partial product is built up from right to left. The next partial product, accumulated with the previous one, starts in the next higher digit position, rewriting product digits, so that most product digits are written more than once. If product scanning is employed, all computations that target a particular product digit are executed close together in time, in adjacent cycles or phases. The operands are scanned only for the digits that will contribute to that targeted product digit. The i and j loop bounds are adjusted such that all partial products accumulated to a product digit occur in immediate succession. Once the product digit is fully computed, it is not revisited; the product digits are written in order as $P[0]$, $P[1]$, $P[2]$, …, *etc.* Altogether the Koç, *et al.* taxonomy lists five classifications. They are listed in Table I.

TABLE I. KOÇ, *ET AL.* SERIAL MONTGOMERY MODEL [6]

Reduction Mode		**Digit Scanning Priority**		
		Operand	***Hybrid***	***Product***
Separated		SOS		
Integrated	Coarse	CIOS	CIHS	
	Fine	FIOS		FIPS

The first three cover operand scanning for both separated and integrated reduction: Separated Operand Scanning (SOS), Coarsely Integrated Operand Scanning (CIOS), and Finely Integrated Operand Scanning (FIOS). Finely Integrated Product Scanning (FIPS) prioritizes product digit scanning, and integrates Montgomery reduction finely on a digit basis. There is a hybrid method, which they term Coarsely Integrated Hybrid Scanning (CIHS). It performs product scanning for the low word of the full product, and then switches to operand scanning for the integrated Montgomery reduction of the high word.

The Koç taxonomy provides useful expressions for evaluating performance and storage requirements. To a first order it assumes that both input operands are split in the same way, *i.e.* that both are split into k digits of d bits each. It specifies the number of digit operations that must be performed. These operations include multiplication, addition, reads, and writes. In all categories, the required number of digit multiplications is $2k^2 + k$, while additions, reads, and writes vary. Since digit multiplications are performed serially, it follows that the minimum number of cycles required to compute a Montgomery product is $2k^2 + k$. The minimum storage requirements for most categories is $k + 3$ digits, while for SOS it is $2k + 2$.

C. Koç Classification of Previous Architectures

Table II lists pertinent characteristics of the Eberle, Großschädl, and Tenca and Koç designs and their Koç classifications.

TABLE II. MONTGOMERY ARCHITECTURE CLASSIFICATION

Architecture	**Base Operand**	**# Base Ops.**	**# Cycles**	**Koç Classi-fication**
Eberle	Digit	$2k^2 + k$	$2k^2 + k$	CIOS
Großschädl	Bit/word	n	$n + k$	FIOS[a]
Tenca & Koç	Bit/digit	nk	$2n + k - 1$	FIOS[a]

[a]Closest fit.

The Eberle digit-based architecture integrates reduction operations, once per each digit-word partial product, and may be classified as Coarsely Integrated Operand Scanning (CIOS). In the Großschädl architecture, priority is given to scanning the bits of operand A in succession, and reduction is performed combinationally (finely) during partial product computation. Therefore it can be classified as FIOS. The Tenca and Koç architecture scans the bits of operand A and the digits of operand B, and performs reduction combinationally without a separate reduction step. Therefore it also can be classified as FIOS.

III. EXTENDED SERIAL MONTGOMERY MODEL

Despite its wide acceptance, the serial Montgomery taxonomy suffers from two limitations. It omits some possible reduction and scanning combinations, and it neglects to consider opportunities for parallelization, particularly at the digit level. A major upgrade to the taxonomy broadens its reach and enhances its utility.

Architectures that perform integrated reduction include both operand and product scanning instances (CIOS, FIOS, CIHS, FIPS). However, the category for separated reduction only considers operand scanning (SOS). It is possible to devise architectures which prioritize product scanning and still perform

reduction in a separated manner, as will be demonstrated below. Accordingly, the taxonomy can be broadened to include a new category denoted as Separated Product Scanning (SPS).

The serial Montgomery taxonomy can also be expanded to encompass *digit level parallelism*. At present, the classification scheme considers only serial realizations in which a single digit multiplier or multiply-accumulate (MAC) unit performs each multiplication in sequence. For operation phases that can be parallelized, simply adding a second digit multiplier can cut those phases' latency in half at relatively low cost. Adding a third dimension of digit level parallelism converts the taxonomy from a flat surface to a large volume of descriptive and analytic power.

Where multiple operand or product digits are being processed concurrently, a more accurate term than digit scanning would be digit *scheduling*. Each category from the Serial Montgomery Model can be expanded to apply to realizations with two or more digit multipliers. This is termed the Extended Serial Montgomery Model. Table III lists the categories from the improved classification scheme.

TABLE III. EXTENDED SERIAL MONTGOMERY MODEL

Reduction Mode		Digit Scanning Priority		
		Operand	***Hybrid***	***Product***
Separated		SOS/m	[a]	SPS/m[b]
Integrated	Coarse	CIOS/m	CIHS/m	[a]
	Fine	FIOS/m	[a]	FIPS/m

[a]Review of relevant literature has not revealed any architectures with these combinations.
[b]New categories applicable to the Montgomery architecture proposed in this research.

Each scheduling method is split into two subcategories. One column applies to the original category with no added parallelism, which usually indicates a single instance of a digit multiplier or MAC unit. The adjacent column applies where a degree of digit level parallelism has been added. The category abbreviation is suffixed with "/m," where m indicates the number (> 1) of instantiated digit multipliers. For example, a CIOS architecture employing two digit multipliers would be denoted as CIOS/2, for 2-digit parallelism. The new SPS and SPS/m categories are also listed.

Adding the third dimension of digit level parallelism aids in estimating performance and resource requirements, and permits comparing two disparate architectures that employ it in different ways. The unique characteristics and dependency relationships of each architecture determine where digit parallelism can be employed, and whether operand or product scanning is better.

For utility in making performance estimates, the categories from the existing serial Montgomery taxonomy list the number of cycles required assuming a single digit multiplier. Recall that $2k^2 + k$ digit multiplications are required. Although it is possible to construct an SPS architecture which also requires $2k^2 + k$ multiplications, the preferred realization uses $2.5k^2 + 0.5k$ multiplications. Despite a higher digit multiplication count, it offers more opportunities for parallel optimization, partly because of the dependency ordering. As a result, it can offer higher performance than other categories with the same number of digit multipliers. It is described in Section IV.

Employing $m > 1$ digit multipliers increases parallelism, but only for those digit operations that can be performed concurrently. Table IV lists the digit scheduling sequences for selected categories for both the strictly serial mode ($m = 1$) and modes with some degree of digit parallelism ($m > 1$).

TABLE IV. SELECTED EXTENDED SERIAL MONTGOMERY MODEL DIGIT SCHEDULES AND CYCLES

Category		Schedule Order	# Cycles
SOS	$m = 1$	$k^2, k(1, k)$	$2k^2 + k$
	$m > 1$	$\lceil k^2/m \rceil, k(1, \lceil k/m \rceil)$	$\lceil k^2/m \rceil + k\lceil k/m \rceil + k$
CIOS	$m = 1$	$k(k, 1, k)$	$2k^2 + k$
	$m > 1$	$k(\lceil k/m \rceil, 1, \lceil k/m \rceil)$	$2k\lceil k/m \rceil + k$
FIOS	$m = 1$	$k[1, 1, 1, 2(k-1)]$	$2k^2 + k$
	$m > 1$	$k[1, 1, 1, \lceil 2(k-1)/m \rceil]$	$k\lceil 2(k-1)/m \rceil + 3k$
SPS	$m = 1$	$k^2, (k^2 + k)/2, k^2$	$2.5k^2 + 0.5k$
	$m > 1$	$\lceil [k^2, (k^2 + k)/2, k^2]/m \rceil$	$\lceil (2.5k^2 + 0.5k)/m \rceil$

Schedule order for the proposed SPS category differs subtly from that for SOS, CIOS, and FIOS. In those three categories, the schedule follows a strict digit order that is imposed by partial product/reduction dependency ordering. By contrast, the SPS has only a macro level dependency order; there is no alternating dependency chain of the form (partial product, reduction, partial product, …). Because SPS employs product digit scheduling, the phases may be overlapped to some degree. The dependency order guarantees that all digit dependencies from one phase to the next are available before the next phase begins, as long as $m \leq k^2$.

To illustrate the degree to which certain categories can benefit from digit parallelism, let $k = 4$ and consider SOS, CIOS, FIOS, and the proposed SPS. The expressions in Table IV determine how many cycles are required for different values of m. Table V lists the cycle count for the respective categories for m, where $1 \leq m \leq 5$.

TABLE V. CYCLE COUNTS FOR SOS, CIOS, FIOS, AND SPS FOR $K = 4$ AND $1 \leq M \leq 5$

m	SOS	CIOS	FIOS	SPS
1	36	36	36	42
2	20	20	24	21
3	18	20	20	14
4	12	12	20	11
5	12	12	20	9

The table shows the relative benefit of increasing digit parallelism. For $m = 1$, categories SOS, CIOS, and FIOS all require 36 cycles, whereas proposed SPS category is slower at 42 cycles. If m is increased to 2, SOS and CIOS improve to 20 cycles, SPS improves to 21 cycles, and FIOS only improves to 24 cycles. For $m = 3$ SOS requires 18 cycles, while CIOS and FIOS both require 20 cycles. In this case, however, SPS is faster than the other three in requiring only 14 cycles. For $m = 4$ and 5, SPS continues to improve relative to the other three categories.

IV. Montgomery Algorithm Optimization

Closer examination of the Montgomery algorithm listed as Algorithm 1 suggests a few possible ways to improve performance. The steps include three multiplications, two modulus functions, an addition, and a division which can be computed as a simple right shift. Fig. 1 graphically illustrates Algorithm 1, along with some annotations.

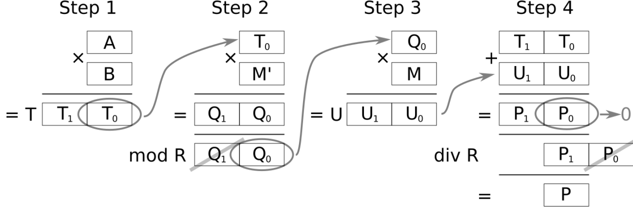

Fig. 1. Montgomery computation steps.

Step 1 computes an initial product T of M-residues A and B. T's upper and lower halves are designated as T_1 and T_0 respectively, and this is represented as $T = (T_1, T_0)$. In Step 2, the quotient Q is computed from T_0 and M'. Only the lower half, Q_0, is used as the operand in the following step. In Step 3, the reduction term $U = (U_1, U_0)$ is computed. In Step 4, U is added to the initial product T, followed by a divide by R (a right shift by n bits where $R = 2^n$) to compute the final Montgomery product P.

Some potential optimizations are readily apparent. After T is computed in Step 1, only its lower half, T_0, is immediately required by Step 2. T_1 is not required until Step 4, so its computation could be delayed or overlapped with Step 2. Because Q_1 is discarded at the end of Step 2, simply not computing it would save time and energy. This property is rarely mentioned explicitly in the literature; brief exceptions are found in [3] and [16]. The Montgomery algorithm guarantees that in Step 4 P_0 will resolve to zero. It is only necessary to know whether in Step 4 there would be a carry generated from the addition of T_0 and U_0, to compute P_1 correctly. Other research that has acknowledged this is [17]. In fact, it is not even necessary to add T_0 and U_0 at all. If $T_0 = 0$, then Step 2 ensures that $Q_0 = 0$, and as a result $U = 0$ and therefore $U_0 = 0$. Conversely, if $T_0 \neq 0$, U_0 is guaranteed to be nonzero. Because $P_0 = T_0 + U_0$ always resolves to zero, a carry into $P_1 = T_1 + U_1$ is invariably generated for $T_0 \neq 0$. Only $P_1 = T_1 + U_1 + 1$ needs to be computed. It is merely necessary in Step 4 to know whether T_0 is nonzero.

A. *Rescheduled Montgomery Multiplication*

Rescheduled Montgomery Multiplication enables efficient computation of the Montgomery product by minimizing unnecessary computations and deferring some other computations. It completely separates product computation from reduction at a macro level. Partial products are computed and summed before reduction starts, similarly to a word-level architecture, except that digit multiplication is used. It employs the novel Separated Product Scheduling (SPS).

Using digit multiplication, computing T, Q, and U in Algorithm 1 as full products would require $3k^2$ digit multiplications. However, in the Rescheduled Montgomery algorithm, the top half of the Q product (Q_1) is not computed. Fig. 2 illustrates the computation of Q_0 for $k = 2$ digits. The shaded areas indicate unused computations. The $T[0]\times M'[1]$ and $T[1]\times M'[0]$ digit products are required for Q_0, but their upper halves are not used. Similarly $T[1]\times M'[1]$ contributes only to Q_1, and so can be skipped altogether.

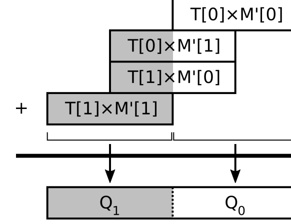

Fig. 2. Digit multiplication for Q.

The number of digit multiplications N_Q to compute Q_0 then is as follows:

$$N_Q = (k^2 + k)/2 \tag{1}$$

Thus, the total number of digit multiplications N_M for a Montgomery product using the Rescheduled Montgomery algorithm requires $N_P = k^2$ for each of T and U, and N_Q for Q_0:

$$N_M = 2k^2 + N_Q = 2.5k^2 + 0.5k \tag{2}$$

Despite requiring more digit multiplications than other categories such as CIOS, FIOS, *etc.*, the Rescheduled Montgomery Multiplier exploits digit level parallelism more efficiently using SPS. Because dependencies occur only at the macro level between products T, Q_0, and U, digit level parallelism can be fully applied within a phase without interruption by an intervening dependency. Furthermore, interphase dependencies can be broken down into digit level dependencies, which means that the start of the next phase can overlap with the completion of the prior phase.

Fig. 3 illustrates the RMM digit multiplication and accumulation schedule for $k = 2$ and using a single digit multiplier ($m = 1$). Each row corresponds to a clock cycle. Digit products appear on the left. To the right of each digit product are the result digits that are being computed during that cycle. The third column lists, in italics, the result digits that have been fully computed in the previous cycles and are available in the accumulator. Finally, the clock cycle count is shown in the rightmost column.

A[0]×B[0]	T[1:0]		0
A[0]×B[1]	T[2:1]	*T[0]*	1
A[1]×B[0]	T[2:1]		2
A[1]×B[1]	T[3:2]	*T[1:0]*	3
T[0]×M'[0]	Q[1:0]	*T[3:0]*	4
T[0]×M'[1]	Q[2:1]	*Q[0]*	5
T[1]×M'[0]	Q[2:1]		6
Q[0]×M[0]	U[1:0]	*Q[1:0]*	7
Q[0]×M[1]	U[2:1]	*U[0]*	8
Q[1]×M[0]	U[2:1]		9
Q[1]×M[1]	U[3:2]	*U[1:0]*	10
		U[3:0]	

Fig. 3. Digit product scheduling for $k = 2$, $m = 1$.

The figure shows the three phases in which the same hardware is reused: T in Cycles 0-3, Q_0 in Cycles 4-6, and finally U in Cycles 7-10. For k = 2 and m = 1, computing the Montgomery product requires only 11 instead of 12 digit multiplications, or about 91.7%.

Dividing operands into a larger number of smaller digits increases granularity, which permits eliminating a higher percentage of digit multiplications that would otherwise contribute to Q_1. In the example above, a full product requires $2^2 = 4$ digit multiplications, while Q_0 only requires 3, or 75%. In the overall scheme, the savings from reducing Q_1 computations approaches 17% with increasing k. Employing a plurality of multipliers permits concurrent digit product scheduling for increased parallelism and lower latency. At the same time, the increased granularity afforded by digit multiplication to minimize Q_0 computations remains.

B. *Architecture*

The proposed architecture consists of a four-stage pipeline with stages designated as Load (L), Multiply (M), Accumulate (A), and Final Sum. Partial products are computed in the first three stages, LMA. In Load, the input operands are selected and loaded into the digit multiplier input registers. In Multiply, the actual digit multiplication occurs and is written to the digit multiplier output register. Finally, in Accumulate, the digit partial product, with appropriate bit offset, is summed with the accumulator. Fig. 4 depicts a pipeline diagram of the LMA stages for $k = 2$, $m = 1$.

Required:					*T[0]*	*T[0]*	*T[1]*	*Q[0]*	*Q[0]*	*Q[1]*	*Q[1]*			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	T[1:0]	T[2:1]	T[2:1]	T[3:2]	Q[1:0]	Q[2:1]	Q[2:1]	U[1:0]	U[2:1]	U[2:1]	U[3:2]			
M		T[1:0]	T[2:1]	T[2:1]	T[3:2]	Q[1:0]	Q[2:1]	Q[2:1]	U[1:0]	U[2:1]	U[2:1]	U[3:2]		
A			T[1:0]	T[2:1]	T[2:1]	T[3:2]	Q[1:0]	Q[2:1]	Q[2:1]	U[1:0]	U[2:1]	U[2:1]	U[3:2]	
			0	1	2	3	4	5	6	7	8	9	10	11
Ready:				*T[0]*		*T[1:0]*	*T[3:0]*	*Q[0]*		*Q[1:0]*	*U[0]*		*U[1:0]*	*U[3:0]*

Fig. 4. RMM pipeline for $k = 2$, $m = 1$.

Fig. 5 depicts the microarchitecture. The Load stage consists of n-bit input registers A, B, M', and M, and selection logic. These registers and the output of the accumulator are multiplexed at the inputs to one or more digit multipliers, which comprise the Multiply stage. Each digit multiplier has two d-bit input registers X and Y and computes a $2d$-bit digit product P in one clock cycle. This is the critical timing path of the entire design because of the large size of the digits. Next, Accumulate adds the digit products to the contents of the accumulator register ACCUM. Lastly, the Final Sum stage computes the reduction. This stage is only active after all the intermediate operands T, Q_0, and U have been fully computed in the LMA pipeline.

The accumulator data path is designed to minimize carry chain delays. As much as is practicable for a given (k, m) configuration, the same accumulation schedule is used for the three T, Q_0, and U computation phases. This maximizes gate reuse and minimizes combinational area growth in the accumulator datapath.

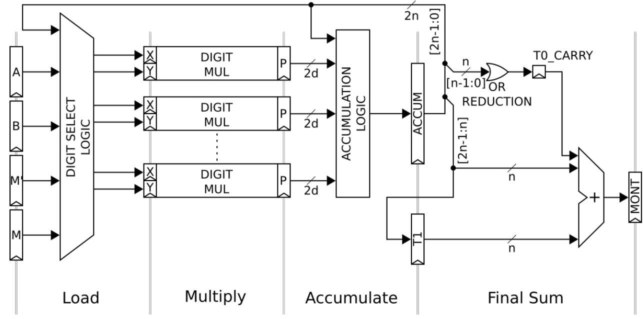

Fig. 5. Rescheduled Montgomery Multiplier architecture.

V. RESULTS

Several variants of the RMM architecture were designed, with a varying number of digits k, and of digit multipliers m. In all cases, the operand size n was set to 256, or to a nearby value to permit a split into k uniform digits each. k was varied from 2 to 8. Depending on k, m was varied from 1 to 14.

Other architectures were also designed for comparison purposes. At one end of size scale, the Eberle, Großschädl, and Tenca and Koç designs comprised the relatively small, intensely iterative realizations. At the other end, various designs that operate on full word-size operands were also built. These included a basic 256×256 synthesized multiplier, and variants (both fully parallel and pipelined versions) of the Montgomery algorithm realized directly in hardware. Other large designs that were implemented included a pipelined Karatsuba-Ofman multiplier [18] and a pipelined arithmetic processor proposed by McIvor, *et al.* in [19].

All designs were written in Verilog HDL and synthesized using Synopsys Design Compiler in the Nangate 45 nm research library [20]. Static timing analysis was performed with Synopsys PrimeTime.

Fig. 6 plots latency versus area for all RMM implementations. The Pareto frontier for area-latency tradeoff is indicated.

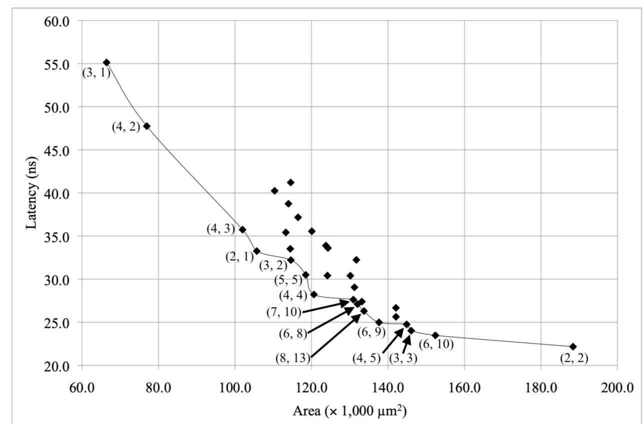

Fig. 6. RMM latency versus area with Pareto frontier.

Table VI summarizes the results for RMM instances which lie on the Pareto frontier of the latency-area plot, ordered by area.

TABLE VI. RMM RESULTS (PARETO FRONTIER)

d	(k, m)	Area (μm^2)	Per. (ns)	f (MHz)	# Cyc.	Total Latency (ns)	A·L Prod.
86	(3, 1)	66.4k	1.969	508	28	55.1	3.66
64	(4, 2)	76.9k	1.910	524	25	47.8	3.67
64	(4, 3)	102.0k	1.881	532	19	35.7	3.64
128	(2, 1)	105.7k	2.217	451	15	33.3	3.52
86	(3, 2)	114.6k	2.013	497	16	32.2	3.69
52	(5, 5)	118.5k	1.794	557	17	30.5	3.61
64	(4, 4)	120.6k	1.881	532	15	28.2	3.40
37	(7, 10)	130.9k	1.625	615	17	27.6	3.62
43	(6, 8)	132.0k	1.694	590	16	27.1	3.58
32	(8, 13)	133.7k	1.547	646	17	26.3	3.52
43	(6, 9)	137.6k	1.666	600	15	25.0	3.44
64	(4, 5)	144.9k	1.904	525	13	24.8	3.59
86	(3, 3)	146.1k	2.002	500	12	24.0	3.51
43	(6, 10)	152.4k	1.677	596	14	23.5	3.58
128	(2, 2)	188.4k	2.217	451	10	22.2	4.18

Of course, increasing area generally purchases a reduction in latency. The trend is not monotonic, because other variables in the architecture and scheduling contribute to achievable performance, beyond aggregate area. This is evident from the variations in achievable latency in the central region of the plot. Between 110k μm^2 and 145k μm^2 there are 16 configurations that are not at Pareto minimum.

For $k \in \{3, 4, 5\}$, configurations in which $m = k$ have the lowest area-latency product for that k. For example, the RMM (4, 4) area-latency product is the minimum of all RMM (4, m) configurations, at 3.40. This implementation requires 121k μm^2 and has a latency of 28 ns. Considering other $k = 4$ configurations, it is possible to reduce area to just over 100k μm^2 by switching to RMM (4, 3) for a 16% area reduction and 8 ns (29%) of additional latency. In the opposite direction, the (4, 5) configuration saves 3 ns (11%) of latency (speedup = 1.12) but at an additional area cost of over 20k μm^2, 20% larger. For the smaller digit sizes in which the operands are subdivided into 6 to 8 digits, the minimum area-latency product is achieved closer to $m = 1.5k$. For example, for $k = 6$, the minimum area-latency product of 3.44 is obtained configuration (6, 9).

A few reasons for this shift include the following. As the digit size d decreases and the number of digits k increases, the number of digit multiplications increases quadratically relative to k. More digit multipliers are required to keep the number of cycles under control. RMM (5, 5) computes a result in 17 cycles of about 1.8 ns each in 118k μm^2. For RMM (6, 6), although the clock period improves to about 1.7 ns, the number of cycles jumps to 20, an 18% increase, in an area of 114k μm^2. RMM (5, 5) has 52×52×5 = 13,520 digit multiplication bits in flight, whereas RMM (6, 6) has 43×43×6 = 11,094 bits in flight. This is a lower degree of digit level parallelism. Conversely, RMM (6, 9) only requires 15 cycles (25 ns) in 138k μm^2, because it has 16,641 bits in flight in any given multiplication cycle. Increasing the number of multipliers with large digits is costly because those multipliers are relatively large. With small digits, adding another multiplier results in a marginal increase in area but improves performance by reducing cycles.

The latency-area curves of Fig. 6 suggest that a point of diminishing returns has been reached with respect to further increases in k. Using an increasing number k of smaller digits becomes more costly relative to less complex designs, despite the fact that the smaller digits permit a higher granularity in computing Q_0 more efficiently. The quadratic relationship of the number of digit multiplications to k means that m must also grow quadratically to keep cycle count down. Although the timing paths within the smaller digit multipliers are shorter, the accumulator logic complexity must grow to handle more vertically stacked digit products.

Fig. 7 plots latency versus area for the serial architectures along with several configurations of the Rescheduled Montgomery Multiplier.

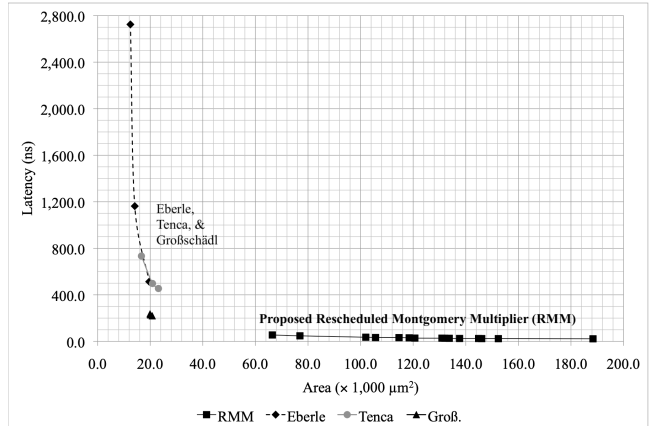

Fig. 7. Latency versus area, serial architectures and RMM.

The serial architectures are clustered near the left side of the plot with low areas and varied latencies, while the RMM architectures vary in size but all have very low latencies. The Eberle, Tenca and Koç, and Großschädl serial designs have areas on the order of 23k μm^2 or less. The latencies of the Eberle and Tenca designs are over 400 ns, approaching 2,800 ns for the worst Eberle instance. Operating at the bit or digit level requires a substantial number of clock cycles, and this tends to overwhelm any performance benefit of reduced cycle time resulting from less complex logic. In contrast, the Großschädl architecture, while still small on the order of 20k μm^2, has latencies all clustered just above 200 ns. The three configurations built compute the Montgomery product identically, and the only difference is the size of the final digit multiplier used for converting the carry save result to nonredundant form. This architecture gives the best performance for area among the serial designs, with area-latency products under 5. The RMM designs all have latencies on the order of 50 ns and less, but area varies widely. The smallest RMM is just over three times the size of the Großschädl architectures, but is five times as fast. Fig. 8 plots the results of all designs. The latency axis uses a logarithmic scale.

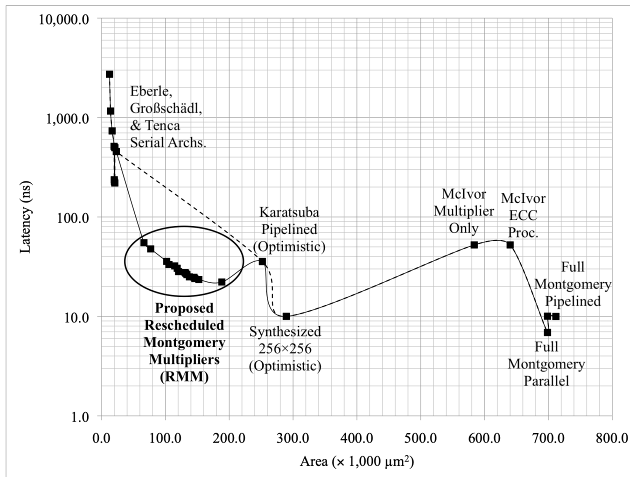

Fig. 8. Latency versus area for implemented Montgomery multipliers.

The plot shows the area versus latency points for all the architectures, along with two curves fit to those points. The dashed line is the curve fit for the points of the prior architectures but excludes the proposed Rescheduled Montgomery Multiplier results. It has a downward slope from the serial architectures to the full size architectures. The solid line shows the curve fit to all points, including the RMM architectures. The RMM latencies fall well below the original, dashed curve fit. Within the context of the architectures that were implemented, it suggests that the RMM establishes a new minimum on the Pareto frontier of the latency-area plot.

VI. Conclusions

This research presents the Extended Serial Montgomery Model, a fundamental expansion of an established taxonomy for categorizing serial realizations of the Montgomery algorithm. The extension adds comprehension of digit level parallelism; it permits the designer to assess the performance and area effects of employing a variable degree of digit level parallelism in an otherwise serial architecture. It also augments the prior taxonomy with a new type of digit scheduling termed Separated Product Scheduling (SPS). Finally, it provides expressions to estimate performance by accounting for the number of digit multiplications, dependency relationships, and the number of digit multipliers.

A novel hardware architecture for Montgomery multiplication, termed the Rescheduled Montgomery Multiplier (RMM), is presented. The architecture synthesizes techniques from multiple sources. It borrows the concept of digit multiplication from serial approaches, and augments it by exploiting digit level parallelism to compute multiple digit products concurrently. Employing the novel SPS approach, it orders digit multiplications to simplify the dependency chain and to minimize stalls and resource underutilization. The digit-centric approach allows it to exploit opportunities in the canonical Montgomery algorithm to eliminate unnecessary computation. This brings two benefits: reducing the number of digit multiplications that must be performed, and permitting opportunistic deferral of some digit multiplications until later in the process. Moreover, it permits a greater degree of parallelization, and a wider range of parallelization options, than are available to prior serial architectures.

The RMM establishes a new region of possible area-latency tradeoffs between, on the one hand, small digit- or bit-oriented serial architectures, and large word-size architectures that perform the canonical Montgomery algorithm in a more conventional way.

References

[1] P. L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519-521, April 1985.

[2] H. Eberle, N. Gura, S. Chang-Shantz, V. Gupta, and L. Rarick, "A Public-key Cryptographic Processor for RSA and ECC," *Proc. 15th IEEE Conf. Appl.-Specific Syst., Arch., and Processors* (ASAP'04), 2004.

[3] J. Großschädl, E. Savas, and K. Yumbul, "Realizing Arbitrary-Precision Modular Multiplication with a Fixed-Precision Multiplier Datapath," *Proc. 2009 International Conference on Reconfigurable Computing and FPGAs*, pp. 261-266, Cancun, Mexico, December 9-11, 2009.

[4] A. F. Tenca and Ç. K. Koç, "A Scalable Architecture for Montgomery Multiplication," *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems* (CHES '99), Lecture Notes in Computer Science (LNCS), vol. 1717, pp. 94-108, Worcester, MA, August 12-13, 1999.

[5] A. F. Tenca and Ç. K. Koç, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm," *IEEE Trans. Computers*, vol. 52, no. 9, pp. 1215-1221, September 2003.

[6] Ç. K. Koç, T. Acar, and B. S. Kaliski, Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26-33, June 1996.

[7] C. McIvor, M. McLoone, and J. V. McCanny, "Improved Montgomery Modular Inverse Algorithm," *Electronics Letters*, vol. 40, no. 18, pp. 1110-1112, September 2, 2004.

[8] G. Gallin and A. Tisserand, "Generation of Finely-Pipelined *GF*(*P*) Multipliers for Flexible Curve Based Cryptography on FPGAs," *IEEE Trans. Computers*, vol. 68, no. 11, pp. 1612-1622, November 2019.

[9] A. F. Tenca and Ç. K. Koç, "A Scalable Architecture for Montgomery Multiplication," *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, Lecture Notes in Computer Science (LNCS), vol. 1717, pp. 94-108, Worcester, MA, August 12-13, 1999.

[10] A. F. Tenca and Ç. K. Koç, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm," *IEEE Trans. Computers*, vol. 52, no. 9, pp. 1215-1221, September 2003.

[11] M. Q. Huang, K. Gaj, and T. El-Ghazawi, "New Hardware Architectures for Montgomery Modular Multiplication Algorithm," *IEEE Trans. Computers*, vol. 60, no. 7, pp. 923-935, July 2011.

[12] A. F. Tenca, G. Todorov, and Ç. K. Koç, "High-Radix Design of a Scalable Modular Multiplier," *Proc. Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES '01), Lecture Notes in Computer Science (LNCS),* vol. 2162, pp. 185-201, Paris, France, May 14-16, 2001.

[13] E. Savas, A. F. Tenca, M. E. Çiftçibasi, and Ç. K. Koç, "Multiplier Architectures for $GF(p)$ and $GF(2^n)$," *IEE Proc. – Computers and Dig. Techniques*, vol. 151, no. 2, pp. 147-160, March 2004.

[14] D. Harris, R. Krishnamurthy, M. Anders, S. Mathew, and S. Hsu, "An Improved Unified Scalable Radix-2 Montgomery Multiplier," *Proc. 17th IEEE Symposium on Computer Arithmetic (ARITH '05),* pp. 172-178, Cape Cod, MA, June 27-29, 2005.

[15] Gabriel Gallin and Arnaud Tisserand, "Hyper-Threaded Multiplier for HECC," *51st Asilomar Conference on Signals, Systems, and Computers,* Pacific Grove, CA, October 29-November 1, 2017, pp. 447-451.

[16] R. R. Liu and S. G. Li, "A Design and Implementation of Montgomery Modular Multiplier," *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, Sapporo, Japan, 2019, pp. 1-4.

[17] J. N. Ding and S. G. Li, "Broken-Karatsuba Multiplication and Its Application to Montgomery Modular Multiplication," *Proc. 27th*

International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, September 4-6, 2017.

[18] A. Karatsuba and Yu. Ofman, “Multiplication of Multidigit Numbers on Automata,” *Proc. USSR Academy of Sciences*, vol. 145, no. 2, pp. 293-294, July 1962. Translation by USSR Academy of Sciences, 1962 from: А. Карацуба и Ю. Офман, «Умножение многозначных чисел на автоматах», *Докл. Академии Наук СССР*, 1962 г., том 145, № 2, с. 293-294.

[19] C. McIvor, M. McLoone, and J. V. McCanny, “Hardware Elliptic Curve Cryptographic Processor over *GF*(*p*),” *IEEE Trans. Circuits and Systems I—Regular Papers*, vol. 53, no. 9, pp. 1946-1957, September 2006.

[20] *Nangate FreePDK45 Generic Open Cell Library*, http://projects.si2.org/openeda.si2.org/projects/nangatelib.