The Dissertation Committee for Trenton John Grale certifies that this is the approved version of the following dissertation:

Rescheduled Montgomery Multiplication: Digit Level Parallelism in Serial Architectures

Committee:

Earl E. Swartzlander, Jr., Supervisor

Andreas Gerstlauer

Lizy K. John

Michael E. Orshansky

Michael J. Schulte

Rescheduled Montgomery Multiplication: Digit Level Parallelism in Serial Architectures

by

Trenton John Grale

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2021

Dedication

Ad maiorem Dei gloriam.

Acknowledgements

It is with great pleasure that I thank my supervisor, Dr. Earl E. Swartzlander, Jr. He made quite a positive impression on me when I first took his Computer Architecture class, and later, High Speed Computer Arithmetic when pursuing my master's degree. After several years in industry, I returned to academia to pursue a doctorate, and Dr. Swartzlander kindly and generously agreed to be my supervising professor again. He has always encouraged me and given practical recommendations for my research and my writing.

I would also like to thank Dr. Andreas Gerstlauer, Dr. Lizy John, Dr. Michael Orshansky, and Dr. Michael Schulte for taking the time to serve on my dissertation committee. I sincerely appreciate their valuable feedback and suggestions to improve my research and dissertation. Their rigorous demands have been challenging, and they have helped drive me to produce a far better dissertation than I could have without them.

Dr. Jason T. Arbaugh has always been available to discuss my ideas for this research in depth. He has reviewed several sections of the dissertation and given me well thought-out, genuinely insightful feedback and advice. The recommendations and advice have not always been what I wanted to hear, but were what I needed to hear. Jason's practical assistance has been vital to my success in this work.

Joel W. Page was my first manager in the chip design business. I am grateful for all that I learned from him, both on a practical engineering level and in collaborating with others on complex technical projects. Like Jason, Joel has very generously reviewed parts of the dissertation. He has asked challenging questions and provided original insights. His assistance has been invaluable to my success.

Other people whom I consider good friends also deserve recognition. Dr. Edwin De Angel first suggested I consider returning to graduate school to earn a Ph.D. He has given me good insights and suggestions about the process of completing a dissertation. Dr. K'Andrea Bickerstaff, whom I met while pursuing my Master's degree many years ago, has always been kind and caring. She was encouraging and welcoming when I decided to return for a Ph.D.

Finally, I am grateful to close friends who have offered their support throughout this journey. John G. Chertude has been a constant source of encouragement and faith in my ability to succeed. Steven T. Vogel has also encouraged me and given me much practical advice. My longtime friend Dr. Mark E. Swartzburg has always given me unqualified cheerleading and a sympathetic ear, as well as insights on dissertation writing. Finally, one of my oldest friends, Dr. James E. Hamman, has helped me by suggesting alternative ways of thinking about and confronting the tasks at hand.

Abstract

Rescheduled Montgomery Multiplication: Digit Level Parallelism in Serial Architectures

Trenton John Grale, Ph.D.

The University of Texas at Austin, 2021

Supervisor: Earl E. Swartzlander, Jr.

Two well-known cryptographic protocols, RSA and ECC, employ modular multiplication on large integers or binary polynomial bit strings of hundreds or thousands of bits. The modulus may be an odd integer (usually prime), or an irreducible polynomial. Large products exceeding the value of the modulus must be reduced to congruent values smaller than the modulus. In simple terms, this is done by taking the remainder with respect to the modulus. However, reduction by integer or polynomial division is computationally expensive.

The Montgomery Multiplication transform and algorithm replace arbitrary division by the modulus with division by a power of two. Both hardware and software realizations of the Montgomery algorithm have been proposed over the past three decades. These range from serial algorithms that perform bit or digit level operations to large full word parallel architectures.

A widely-adopted classification scheme categorizes and characterizes serial Montgomery architectures. This dissertation introduces the Serial Montgomery Model

by fundamentally upgrading the existing scheme to accurately characterize a rich universe of architectures that employ digit level parallelism.

Certain properties of the Montgomery computation present optimization opportunities that have been little noted by prior researchers. This dissertation presents a novel Rescheduled Montgomery Multiplication architecture that targets those opportunities to drive a new level of optimal tradeoffs between area cost and latency. The architecture exploits digit level parallelism and dependency scheduling to attain higher performance than is attainable with serial architectures while avoiding the high area cost associated with large parallel architectures.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Motivation

Ensuring the privacy and integrity of sensitive electronic information continues to gain importance as mobile and interconnected devices become more prevalent. Encryption of data scrambles it in such a way that an unauthorized recipient cannot read it, but an authorized recipient can. The data to be encrypted, called *plaintext*, is input to an encryption algorithm with a *key*. The output is the encrypted data, called *ciphertext*. On receipt of the ciphertext, the receiver applies it and a key to a decryption algorithm, which outputs the plaintext.

Broadly speaking, modern cryptographic systems can be categorized as either symmetric or asymmetric. In symmetric encryption, the same key is used for both encryption and decryption. Two parties who wish to communicate with each other must first share this private key between themselves. In asymmetric encryption, two separate keys must be used: one for encryption, and one for decryption. A party that wants to communicate securely with others will generate both keys. This is referred to as a *key pair*. The encryption key is usually termed a *public key*, because the party generating it makes it available to the public. Any person may use the public key to encrypt and send a message to the generating party. It cannot be used to decrypt the message. The decryption key is usually termed the *private key*, because the generating party will keep it private, or secret. Only the private key can be used to decrypt the message. Because the usual convention is for the encryption key to be made public, asymmetric encryption is also referred to as a public key (PK) cryptographic system.

Public key cryptographic systems work because they are difficult to crack. They are built upon what are termed one-way trap door functions. A function is one-way in that it operates on the message and the encryption key to produce an encrypted output,

but the reverse operation is computationally difficult. Given only the encrypted message and the encryption key, it is a computationally hard problem to retrieve the original message. The private encryption key provides the trap door: with the encrypted message and the decryption key, the original message can be computed relatively easily. The public encryption and private decryption keys have a mathematical relationship to each other. Given a public encryption key, it is computationally difficult to determine from it the private decryption key [1].

Two currently prevalent public key cryptosystems are the Rivest-Shamir-Adleman (RSA) algorithm [2] and Elliptic Curve Cryptography (ECC) [3]. In both systems, larger key sizes are correlated with higher levels of security. Currently, RSA uses key sizes on the order of 2,048 bits or more. To achieve a comparable level of security, ECC uses key sizes in the hundreds of bits.

At a basic level, both RSA and ECC employ modular arithmetic extensively. That is, arithmetic operations such as addition, subtraction, multiplication, division, and exponentiation are performed, followed by computing the modulo function on the result with respect to some pre-selected modulus. In some cases the operands are integers, and the modulus is often a prime number. In other cases the operands are binary polynomials, and the modulus is an irreducible (nonfactorable) polynomial. Multiplication and exponentiation operations present a challenge because their intermediate results can be large relative to the operand size. Taking the modulus of a large result can be computationally expensive. P. L. Montgomery proposed an efficient algorithm for performing modular multiplication [4]. It plays a prominent role in this dissertation.

Although public key cryptography employs well-understood mathematical principles, high-performance implementation remains a challenge. One obvious reason

for this is the relatively large operand sizes, on the order of hundreds or thousands of bits. A software implementation on a general purpose 64-bit CPU can require many cycles of loads, computations, and stores because the register file space is unlikely to be large enough to hold the operands. A dedicated hardware implementation, with large registers, can have high performance, but will have a high complexity and relatively low utilization since it is dedicated to a specific function.

For a well-defined problem set, a hardware implementation will generally be faster than a software implementation on a general purpose microprocessor. This remains true for PK cryptosystems, for which there are pure software implementations at one extreme, and direct hardware implementations for a specified set of parameters, at the other. Between those extremes, there are relatively fast hardware implementations that still provide flexibility in parameters such as key size and choice of field modulus.

Over the past several decades, researchers have proposed numerous hardware implementations of the Montgomery multiplication algorithm, either standalone or as components of comprehensive cryptographic engines. These architectures have made different tradeoffs among such parameters as area, performance, power, and operand size, according to the targeted application and other constraints such as cost. Some operate at a fine level of granularity, splitting up operands into smaller pieces and employing a sequence of small-scale mathematical computations which are then merged into a final whole. If performed sequentially, the small computations can require a large number of cycles, but with a very fast clock. Others operate at a coarser granular level, in which each major step of the algorithm is performed whole. This usually results in a much slower clock, but only a small number of cycles are required.

In general, the Montgomery algorithm is taken as given, and all its steps performed. Proposals for optimizing performance or area typically focus on

microarchitectural improvements to the chosen architecture. They may include reordering parts of the algorithm, but do not change it substantively.

1.2 Proposal

When architecting a cryptographic processor, the designer will have a set of criteria that must be met. One criterion is whether the architecture should be specific to a particular algorithm, or general to support a multitude of algorithms. Other criteria include the targeted operand sizes, and performance requirements such as latency (less is generally better) and throughput (higher is generally better). At the same time, the designer is confronted with limitations and constraints, such as allowable die area and maximum energy consumption (especially, but not solely, in the case of mobile deployments). Therefore, there is seldom a "one size fits all" solution. The requirements and constraints guide the designer in choosing an architecture that best fits the particular set of tradeoffs for the targeted deployment.

This dissertation presents a comprehensive classification scheme for Montgomery architectures that employ digit level parallelism. It demonstrates that a widely accepted, existing taxonomy for serial Montgomery multiplication lacks an important dimension. It fundamentally revises the taxonomy to incorporate that dimension, and thereby expands the taxonomy's reach and analytic utility. The proposed Serial Montgomery Model provides expressions for estimating the performance of realizations employing varying degrees of digit level parallelism.

This dissertation presents a novel hardware architecture for performing Montgomery multiplication, termed the Rescheduled Montgomery Multiplier. It demonstrates that the proposed architecture achieves a new set of latency-area tradeoffs

for hardware Montgomery multiplication, and shifts a targeted region of the latency-area Pareto frontier to new minima.

Conceptually, this dissertation demonstrates optimization opportunities present in the Montgomery algorithm to reduce latency in computing a Montgomery product. It is the first research to identify and analyze those opportunities systematically and comprehensively. Optimizations include avoiding unnecessary computations, deferring some computations where possible, and short-circuiting other computations by replacing them with equivalent simpler ones. The Rescheduled Montgomery Multiplier employs parallelism and targeted instruction ordering to enable concurrent processing of different parts of the overall computation, despite macro level dependencies. It achieves a new level of performance while minimizing area.

Compared to various previous architectures, the Rescheduled Montgomery Multiplier achieves at least *one order of magnitude* of latency reduction without invoking a drastic area penalty. Compared to some other architectures, the Rescheduled Montgomery Multiplier fits into less than 25% of the area, while having *lower* latency.

1.3 Organization

This dissertation is organized as follows. Chapter 2 introduces RSA and ECC and describes the fundamental importance of modular multiplication to their realization. It describes the underlying mathematical principles on which modular multiplication is based. It presents the Montgomery multiplication algorithm and related algorithms. Chapter 3 reviews the prior work of other researchers on Montgomery multiplication, particularly in hardware realizations. Chapter 4 introduces a novel Montgomery taxonomy for serial architectures that encompasses a variable degree of digit level parallelism. Chapter 5 analyzes the Montgomery multiplication algorithm and

systematically identifies opportunities for optimization. It proposes a family of hardware architectures for dedicated Montgomery multipliers that exploit digit level parallelism coupled with the optimization opportunities. Chapter 6 establishes the experimental methodology and criteria for evaluation. For comparison purposes, Chapter 7 reviews some possible naive Montgomery realizations. Chapter 8 reviews results of the experiments. Chapter 9 draws conclusions and describes opportunities for further research.

Chapter 2 Background

2.1 Public Key Cryptographic Systems

Both RSA and ECC systems employ a large but restricted set of operands and make extensive use of modular arithmetic. RSA employs modular exponentiation over integer fields. This is often implemented as repeated modular multiplication and squaring. ECC uses elliptic curve (EC) point operations on elliptic curves defined over either finite integer fields $GF(p)$ or finite polynomial fields $GF(2^n)$. In turn, the point operations consist of modular addition, subtraction, multiplication, and division. Modular division is implemented as a product of the dividend and the multiplicative inverse (modular inversion) of the divisor.

Because a computation can produce a result that is outside the allowable range of elements of the field, an important step is reduction of the result to a congruent value that is within the permitted range. As previously discussed, modular addition and subtraction are relatively easy. Modular multiplication is more computationally intensive. Finally, modular inversion is the most computationally intensive. Accordingly, some ECC implementations employ alternative coordinate systems that permit the inversion operation to be deferred until the very end of a computation. The tradeoff is usually a substantially larger number of intermediate computations.

Deploying an RSA cryptographic system begins with generating a public and private key pair. Two fairly large primes p and q of equal width are selected. Typical sizes are over one thousand bits. These primes are multiplied to yield the modulus n: $n = pq$. n is large enough such that it is computationally expensive to factor. Next a term ϕ is computed: $\phi = (p - 1)(q - 1)$. The user then chooses a random integer e, which is relatively prime to ϕ. The term "relatively prime" means that e and ϕ have no common divisors. The pair (n, e) constitutes the public key, and is freely shared. The user

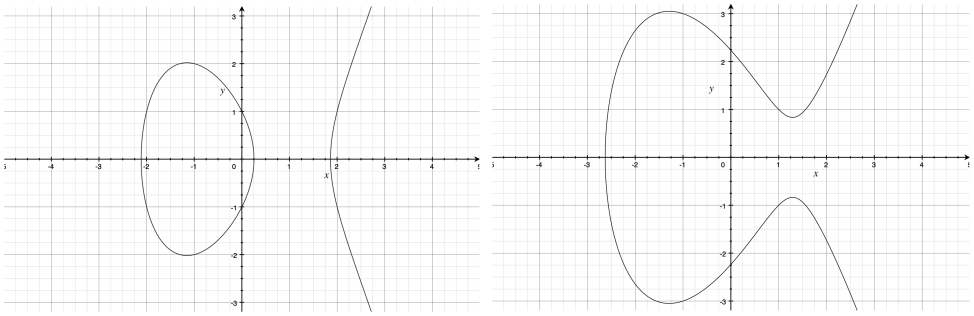
generates a private key $d = e^{-1} \pmod{\phi}$. d is relatively prime to n, and $ed \equiv 1 \pmod{\phi}$. The user keeps d secret. One component of RSA's security is that, while it is straightforward for the user to compute d from e and ϕ, it is computationally very expensive to compute d from e and n. Factoring n to find p and q, and thus ϕ, is expensive [1], [2].

Given a message m that a second person wishes to send privately to the person who generated the key pair, the second person encrypts it as follows. If the message is large, it can be broken up into smaller blocks. The second person encrypts, or transforms the message to ciphertext c using $c = m^e \bmod n$, and transmits it to the first person. The first person receives c, and decrypts it to plaintext p using $p = c^d \bmod n$. p is identical to the original message m because $p = (m^e)^d \pmod{n} = m^{ed} \pmod{n} \equiv m \pmod{n}$ for all m [1], [2]. The product ed itself is not necessarily congruent to 1 (mod n). In both cases, encryption and decryption, modular exponentiation is employed. It is often implemented as a sequence of repeated modular multiplication and squaring operations.

Elliptic curve cryptography (ECC) employs operations on a geometric construct called an elliptic curve (EC). An EC is represented as a graph in a 2-dimensional plane and is described most generally by the following expression, called the Weierstrass equation [3]. In most curves of interest for ECC, some of these terms, *e.g.* a_1xy, are absent.

$$E: \; y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

Fig. 1 shows some examples of elliptic curves defined over the set of real numbers [5].



(a) $y^2 = x^3 - 4x + 1$ (b) $y^2 = x^3 - 5x + 5$

Fig. 1. Some elliptic curves over the set of real numbers [5].

A point P on elliptic curve E has x and y coordinates that satisfy (2.1), but E is not necessarily defined for all possible values of x and y. ECC algorithms define and employ three operations for all points P on E. These operations are termed point negation, point addition, and point doubling. The operations have specific geometric meanings, and their computations are nontrivial. Let the points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$. Table 1 lists the point operations and their underlying computations.

Table 1. Elliptic curve point operations for three points P_1, P_2, and P_3.

Point Operation	**Expression**	**Computation**
Negation	$P_3 = -P_1$	$x_3 = x_1$ $y_3 = x_1 + y_1$
Addition	$P_3 = P_1 + P_2$	$\lambda = \frac{y_2 + y_1}{x_2 + x_1}$ $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$
Doubling	$P_3 = 2P_1$	$\lambda = x_1 + \frac{y_1}{x_1} = \frac{x_1^2 + y_1}{x_1}$ $x_3 = \lambda^2 + \lambda + a = x_1^2 + b/x_1^2$ $y_3 = x_1^2 + (\lambda + 1)x_3 = x_1^2 + \lambda x_3 + x_3$

Although Fig. 1 depicts ECs defined over the real numbers, ECC employs curves defined over a finite field. It is implied that the computations in the third column of Table 1 are performed over the underlying finite field, *i.e.* modulo a field modulus M. As evident from the table, the computations consist of modular addition, multiplication, and division. Modular division is implemented as a multiplicative inversion (reciprocal) followed by a multiplication. For example, $A/B \bmod M$ is implemented as $A \times (1/B) \bmod M = AB^{-1} \bmod M$. Furthermore, modular squaring is often treated as a separate operation from multiplication, because just as in integer implementations, its computation is usually cheaper than general multiplication.

Modular inversion is typically expensive to compute relative to addition, multiplication, and squaring. Consequently, ECC researchers have devised alternative coordinate systems which defer inversion to the end, by carrying redundant information and performing a larger set of computations along the way. The two-dimensional (x, y) coordinates used in Table 1 are referred to as affine coordinates. Other coordinate systems that have been proposed include Projective, Lopez-Dahab, and Jacobian [3]. Table 2 lists transforms between affine and these other coordinate systems.

Table 2. Transforms between affine and alternate coordinate systems [3].

Alternate Coordinate System	**Affine Coordinates**	**Transform to Alternate**	**Alternate Coordinates**	**Transform to Affine**
Projective	(x, y)	$(x, y, 1)$	(X, Y, Z)	$(X/Z, Y/Z)$
Lopez-Dahab	(x, y)	$(x, y, 1)$	(X, Y, Z)	$(X/Z, Y/Z^2)$
Jacobian	(x, y)	$(x, y, 1)$	(X, Y, Z)	$(X/Z^2, Y/Z^3)$

The third, Z-coordinate carries the redundant information. The point operation computations for the alternate coordinate systems are omitted here. However, the only

inversion that must be performed is on the final value of Z at the end of the computations to return to affine coordinates.

Two entities **A** and **B** that wish to communicate mutually choose a particular elliptic curve and a point P on that curve. **A** randomly selects a private key k_A, which it does not reveal. **B** randomly selects its own private key k_B, which it also keeps secret. **A** computes a point multiplication $P_A = k_A P$, and shares it with **B**. In turn **B** computes a point multiplication $P_B = k_B P$, and shares it with **A**. Both **A** and **B** can now compute a shared secret Q, as follows. **A** computes $k_A(P_B) = k_A(k_B P)$, and **B** computes $k_B(P_A) = k_B(k_A P)$. $k_A(k_B P) = k_B(k_A P) = Q$. Both can now use Q as a key to a separate symmetric cryptographic algorithm to use for sharing payload messages.

2.2 Mathematical Foundation

RSA and ECC employ operations on a finite set of elements. The results of those operations are also elements of the finite set. In many cases the set consists of a sequence of integers. The operations are modular arithmetic functions, such as addition and multiplication, which are performed with respect to a modulus.

At the foundational level is the concept of a group. A group G is defined as a set that additionally has some binary operation defined, denoted generically as *. As a rule, a group has the following properties. First, the group is *associative*, such that for set members (elements) $a, b, c \in G$, $a * (b * c) = (a * b) * c$. Second, there is an *identity* element e such that for all $a \in G$, $a * e = e * a = a$. Third, there is an *inverse* element a^{-1} such that for all a, $a * a^{-1} = a^{-1} * a = e$. Additionally, if the group is commutative, such that $a * b = b * a$, then the group is termed *abelian* [6]. As an example, consider the set $\mathbb{Z}$ of all integers and the binary operation addition, denoted by +. Given arbitrarily-chosen integers 3, 5, and 9, it is evident that all of the preceding properties apply:

associativity, such that $3 + (5 + 9) = (3 + 5) + 9 = 17$; there is an identity element 0, such that $3 + 0 = 0 + 3 = 3$; an inverse element -3, such that $3 + -3 = -3 + 3 = 0$; and finally, commutativity, such that $3 + 5 = 5 + 3 = 8$.

At the next level is the *ring*. It has two operations and it is denoted as $(R, +, \cdot)$, where R indicates the set, and the two binary operations + and · are defined. For present purposes, the operations can be assumed to be addition (+) and multiplication (×), and thus the ring is $(R, +, \times)$. A ring has the following properties. First, the set R is abelian with respect to addition. Second, multiplication is associative: $(a \times b) \times c = a \times (b \times c)$. Third, multiplication is distributive: $a \times (b + c) = a \times b + a \times c$ [6].

Next is the *field*. If the nonzero elements of the set R constitute a *group* for the × operation, and if the ring is commutative, it is termed a field. For practical purposes, the additive identity element is termed the *zero element* and is 0. The *multiplicative identity element* is 1. That is, for $a \in R$, $a + 0 = a$, and $a \times 1 = a$. Given a prime number p, and the finite set of integers $\{0, 1, \ldots, p - 1\}$ as elements, a finite field is termed a Galois Field of order p, and is typically denoted as $GF(p)$ [6].

In addition to integers, it is possible to apply the foregoing concepts to polynomials. For purposes of this research, a polynomial field is defined over a polynomial in 2, where the coefficients $a_i \in \{0, 1\}$. This is termed a binary polynomial, and is denoted as $GF(2^n)$. The maximum degree of a polynomial element is $n - 1$. The modulus is an irreducible polynomial of degree n.

Operations over finite fields are characterized by the concept of congruence. What this means is that the result of a defined operation, even if the result initially falls outside the bounds of the field, always has an equivalent corresponding element within the field. Consider the field $GF(5)$. The integer 7 is not an element of the field, but it is congruent to field element 2. This is computed by taking the modulus of 7 with respect

to 5, or 7 mod 5 = 2. Symbolically, the congruence is represented as $7 \equiv 2 \pmod 5$. One can think of the field elements as wrapping around once they exceed $5 - 1 = 4$. The process of computing a congruence within the field is termed modular reduction.

The following arithmetic operations are defined for $GF(p)$ fields: addition (+) and multiplication (×). These operations, when correctly and completely executed, produce results that are themselves elements of the field. $GF(p)$ arithmetic is the same as integer arithmetic, except that the initial result may be a value outside the range of the field and must be transformed (reduced) to the congruent element within the field.

Addition is straightforward. Consider a finite field $GF(M)$, where M is used in place of p to denote the modulus. The sum S of two elements A and B is $S = A + B$. If $S \geq M$, then it is easy to compute the congruent field element S' by subtracting M from S: $S' = S - M$. For example, in $GF(5)$, $2 + 4 = 6 \geq 5$. Thus to compute the congruence, 5 is subtracted from 6, with 1 as the result.

Multiplication is more complicated, because an initial product $T = AB$ can be many times as large as the modulus M, *i.e.* as large as $(M - 1)^2$. Reduction of T can be performed by repeated subtraction of the modulus until the result is less than the modulus. Alternatively, if an integer divider is available, it can be used to compute the quotient and remainder of T/M, taking the remainder as the congruent result.

Two additional properties defined for $GF(p)$ fields are an additive inverse ($-A$) and a multiplicative inverse (A^{-1}). Given a finite field with modulus M and an arbitrary field element A, the additive inverse of A is another field element, denoted by $-A$, such that the sum $A + -A$ is congruent to 0. Symbolically: $A + -A \equiv 0 \pmod M$. For example, in $GF(7)$, let $A = 2$. Then, $-A = 5$ because $2 + 5 = 7 \equiv 0 \pmod 7$. The multiplicative inverse, $1/A$, denoted by A^{-1}, is the field element such that $A \times A^{-1} \equiv 1 \pmod M$. Again in $GF(7)$, let $A = 2$. Then, $A^{-1} = 4$ because $2 \times 4 = 8 \equiv 1 \pmod 7$.

Although the preceding introduction describes modular operations over prime integer fields $GF(p)$, they are also applicable in cases where the modulus itself is not prime, subject to some limitations. One such use case is RSA, whose modulus is actually the product of two large primes: $n = pq$ where p and q are both prime [2], [1].

2.3 Montgomery Multiplication

As previously demonstrated, modular multiplication in integer fields is complicated by the difficulty of computing the modular reduction. P. L. Montgomery offers an ingenious solution to this problem [4]. The Montgomery multiplication algorithm replaces division by the arbitrary modulus M with division by a power of two, which is simply a right shift. 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*.

At a high level, the algorithm works as follows. Assume a modulus M of n bits, such that $n = \lceil \log_2 M \rceil$. Let $R = 2^n$. It is usually beneficial that R be a multiple of the machine word size. The modular multiplicative inverse of R with respect to M is $R^{-1} \pmod{M} = 2^{-n} \pmod{M}$. Consider two field elements A and B whose modular product is to be computed. These can be transformed to the M-residues A' and B' via conventional modular multiplication of A and B, respectively, by R with respect to M. Thus: $A' = AR \bmod M$ and $B' = BR \bmod M$. When Montgomery Multiplication is invoked with A' and B', it computes the following product: $A'B'R^{-1} \bmod M$:

$$\begin{aligned} \text{MontMult}(A'B') &= (A'B')R^{-1} \bmod M \\ &= (AR)(BR)R^{-1} \bmod M \\ &= (AB)R \bmod M \\ &= (AB)' \end{aligned}$$

The Montgomery Multiplication function computes the M-residue of the product of the M-residues of the multiplicands. In other words, given two elements in the Montgomery domain, the function computes a product which itself is in the Montgomery domain. In addition to the M-residues, the Montgomery algorithm requires that two other terms be precomputed:

$$R^{-1} \pmod{M} \quad \text{where } RR^{-1} \bmod M = 1$$
$$M' \quad \text{where } RR^{-1} - MM' = 1$$

The Montgomery Multiplication Algorithm is listed as Algorithm 2.1. Assume that A and B are already M-residues.

Algorithm 2.1. Montgomery Multiplication Algorithm [4].
Input: A, B
Output: $P = ABR^{-1} \bmod M$

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

In Step 1, the input operands A and B are multiplied to produce initial product T, which is $2n$ bits wide. For purposes of this discussion, T can be decomposed into two n-bit words, a high word T_1 and a low word T_0. Word concatenation is illustrated with parentheses, so $T = (T_1, T_0)$. In Step 2, Q is computed as the product of T_0 and M'. The low word Q_0 of this product is selected ($Q \bmod R$). Q_0 may be described as a quotient term, and it is what makes it possible to avoid performing a conventional division

operation, as will be seen presently. In Step 3, Q_0 is multiplied with M to produce U. In Step 4, T and U are summed, and the entire result is divided by R to produce a preliminary product result P. The division by R is merely a right-shift by n bits. It is possible that after the shift $P > M$ (however $P < 2M$), and Step 5 tests for this case. If it occurs, the subtraction in Step 6 ensures that $0 \leq P < M$.

Replacing the division by R with an n-bit right shift in Step 4 is possible without any loss of information because the low word of the sum computes to zero. M' is the additive inverse of the multiplicative inverse of M with respect to R. In other words $M' = -M^{-1} \pmod{R}$. In effect then, $Q_0 = -T_0 M^{-1} \pmod{R}$. Because of the foregoing, $U = -T_0 M^{-1} M$ and $U_0 = -T_0 \pmod{R}$. The addition of T and $Q_0 M = (T_1, T_0) + (U_1, U_0) = (T_1, T_0) + (U_1, -T_0) = (T_1 + U_1 + c, 0)$, where c is the carry out of the sum $T_0 + U_0 = T_0 + -T_0$. In this way, the quotient term Q_0 and the foregoing computations have zeroed out T_0, thus making the right shift possible.

Clearly, there is a cost to using Montgomery multiplication. Prior to multiplication, the operands must be transformed to M-residues, and then the Montgomery product must be transformed back to the integer domain. For a single modular multiplication this overhead may confer no benefit. On the other hand, where a sequence of several multiplications is required, the overhead may be acceptable. This is the case in RSA implementations which perform modular exponentiation as a sequence of modular multiplications and squarings. It is also invariably the case in ECC whose point operations consist of several modular multiplications.

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.

2.4 Montgomery Inverse

Multiplicative modular inversion is a vital function for elliptic curve cryptography. For a field element X, its inverse is defined as $Y = X^{-1} \pmod{M}$ such that $XY = XX^{-1} \pmod{M} \equiv 1 \pmod{M}$. In terms of processing time, modular inversion can be relatively expensive, even when compared to multiplication. Furthermore, depending on the algorithm, the computation time can vary with the input. Nevertheless several novel and efficient inversion algorithms and hardware have been proposed.

In a foundational paper, B. S. Kaliski describes an algorithm to compute the Montgomery inverse of an arbitrary integer A [7]. In this context, let M be the prime modulus, $R = 2^n$, where $2^{n-1} \leq M < 2^n$. The algorithm computes $A^{-1}R \bmod M$, the M-residue of the multiplicative inverse of A. As originally presented, the algorithm uses a sequence of additions and right shifts; no general multiplication or division operations are required. In describing the algorithm, the author first refers to a greatest common divisor algorithm (GCD) which uses halvings (right shifts) and subtractions. He then refers to a related algorithm for the modular inverse which uses halving. His contribution is twofold: First, he notes that it is possible to defer the halvings to the end, such that an intermediate result is $A^{-1}2^z \bmod M$ where $z \geq n$. Second, he observes that, for $z \geq n$, one of the intermediate values is the M-residue of the inverse, *i.e.* $A^{-1}2^n \bmod M = A^{-1}R \bmod M$. The algorithm consists of two processing phases, designated Phase 1 and Phase 2.

Algorithm 2.2. Kaliski's Montgomery Inverse Algorithm, Phase 1 [7].
Input: A, M
Output: $r = A^{-1}2^z \bmod M$, z
1. $u = M$
2. $v = A$
3. $r = 0$
4. $s = 1$
5. $z = 0$
6. while ($v > 0$) do:
7. if (u is even) then:
8. $u = u / 2$
9. $s = 2s$
10. else if (v is even) then:
11. $v = v / 2$
12. $r = 2r$
13. else if ($u > v$) then:
14. $u = (u - v) / 2$
15. $r = r + s$
16. $s = 2s$
17. else:
18. $v = (v - u) / 2$
19. $s = r + s$
20. $r = 2r$
21. end if
22. $z = z + 1$
23. end while
24. if ($u \neq 1$) then:
25. return "Not relatively prime."
26. end if
27. if ($r \geq M$) then:
28. $r = r - M$
29. end if
30. return r, z

Algorithm 2.3. Kaliski's Montgomery Inverse Algorithm, Phase 2 [7].
Input: z, n, r, M
Output: $Z = A^{-1}2^n \bmod M = A^{-1}R \bmod M$
1. for $i = 1$ to ($z - n$) do:
2. if (r is even) then:
3. $r = r / 2$
4. else:
5. $r = (r + M) / 2$
6. end if
7. end do
8. return $Z = M - r$

If $z > n$, then this algorithm computes the Montgomery inverse faster than it would take to first compute the ordinary inverse and then convert it to the Montgomery domain (M-residue) [7].

The paper by E. Savas and Ç. K. Koç is an important work is a that analyzes Kaliski's Montgomery inversion algorithm in detail and extends it [8]. The authors introduce a modification to the definition of the Montgomery inverse which they term the New Montgomery Inverse. The change makes it possible to replace Kaliski's Phase 2 with a short sequence of Montgomery multiplications. The authors propose algorithms to compute the following:

- Classical modular inverse
- Kaliski-Montgomery inverse
- New Montgomery inverse

A. Gutub, *et al.* extend the Savas and Koç basic algorithm into a scalable architecture in which operands are not limited to a fixed machine word size [9]. C. McIvor, *et al.* propose and employ a unified architecture for both multiplication and inversion [10], [11].

2.5 Iterative Algorithms

Ordinary multiplication can be implemented with iterative algorithms that operate on smaller portions of the operands. An operand 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.

Assume that operands are n bits wide and that they are divided into digits, each of which is d bits wide. Each operand then consists of $k = \lceil n/d \rceil$ digits. An arbitrary operand A can be expressed as a concatenation of its digits: ($A[k–1]$, $A[k–2]$, …, $A[1]$, $A[0]$), where $A[i]$ indicates the i-th digit from the right. Algorithm 2.4 illustrates how a product of two large operands A and B can be computed via serial digit multiplication and accumulation.

Algorithm 2.4. Digit-Based Iterative Multiplication.
Input: A, B
Output: $P = AB$
1. $P = 0$
2. for $i = 0$ to $k – 1$ do:
3. $C = 0$
4. for $j = 0$ to $k – 1$ do:
5. $(C, S) = A[i] \times B[j] + P[i+j] + C$
6. $P[i+j] = S$
7. end for
8. $P[i+k] = C$
9. end for

Inspection of Algorithm 2.4 shows that to compute the full product, $N_P = k^2$ digit multiplications must be performed:

$$N_P = k^2 \tag{2.2}$$

For example, let $P = AB$, and A and B are split into $k = 2$ digits each, denoted by $A[1]$, $A[0]$ and $B[1]$, $B[0]$ respectively. The subscripts used here are digit indices, not bit indices. For $k = 2$, $d = n/2$. The product is computed by adding the digit products as follows:

$$P = 2^n(A[1] \times B[1]) + 2^{n/2}(A[1] \times B[0] + A[0] \times B[1]) + A[0] \times B[0] \tag{2.3}$$

Fig. 2 illustrates how the digit products are aligned and accumulated to produce the product *P*.

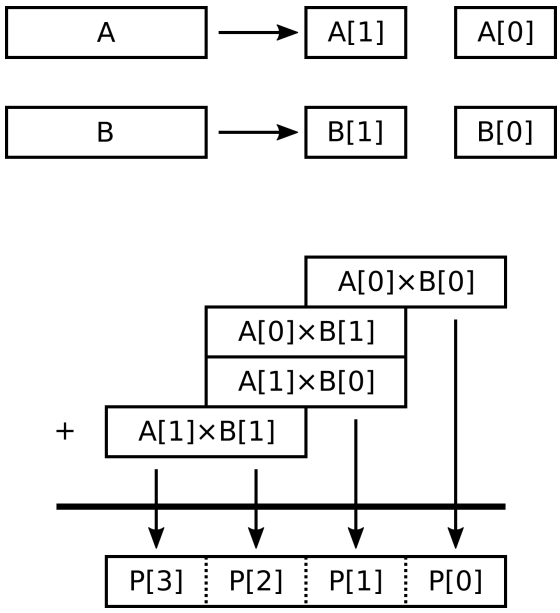


Fig. 2. Digit multiplication.

Chapter 3 Related Work

Hardware Montgomery multiplier architectures are not new. The paper by S. E. Eldridge and C. D. Walter [12] conducts a systematic review of implementations extant in the early 1990s. H. Orup proposes a digit-serial architecture based on optimizing quotient determination (Step 2 of the Montgomery Algorithm) [13]. O. Nibouche, *et al.* present a parallel implementation, and a systolic variant, that incorporates finely interleaved Montgomery reduction [14].

Some recent research has targeted field programmable gate arrays (FPGAs) for hardware implementation. The proposed architectures typically take advantage of the chosen FPGA family's resources to drive high performance. The digit-serial implementation by Erdem, *et al*. targets the Xilinx Virtex-7 [15]. A tool to generate VHDL code for several different Xilinx FPGAs is described in [16].

The remainder of this chapter reviews a specific, diverse set of architectures for computing Montgomery products. The objectives of these architectures vary among minimizing area, maximizing performance, offering flexibility in operand size, and potentially enabling other related computations. These architectures also vary in how they divide up the work. Division may be spatial, in the sense that parts of the computation may be split up and performed concurrently in duplicated hardware. Division may also be temporal, in that sequential processing and pipelining may be employed.

3.1 McIvor ECC Processor with Pipelined Multiplier

C. McIvor, *et al.* present a complete ECC coprocessor that employs several innovative techniques [11]. It performs Montgomery multiplication and uses a novel

implementation of modular inversion. The processor is developed for FPGA implementation and includes a 256×256-bit pipelined multiplier.

The architecture places considerable emphasis on efficient modular inversion. The authors present a unified inversion algorithm in [10]. It builds upon prior algorithms proposed by Savas and Koç which substitute Montgomery multiplication for some of Kaliski's iterative computations [8]. The unified inversion algorithm can compute the Montgomery inverse of an ordinary integer, or the ordinary inverse of a Montgomery element, as well as convert between the ordinary domain and the Montgomery domain.

The coprocessor can be configured to perform any of the following four operations: Montgomery multiplication, multiplicative inversion (whether conventional or Montgomery), modular addition, and modular subtraction.

The 256×256 multiplier is hierarchical in composition. At the top level, the array is divided into four 256-bit partial product quadrants. Its 512-bit output register is fed by the sum of the four registered 256-bit partial products, properly aligned. The same hierarchical grouping is recursively applied to the 256-bit quadrants on down. At the lowest level the McIvor multiplier consists of 256 16×16 digit multipliers. Successively larger partial products are formed by adder logic and registered in a kind of bottom-up pipelining.

Although the McIvor ECC Processor is a complete acceleration engine that employs Montgomery multiplication, it does not attempt to optimize the Montgomery algorithm as such. It performs the algorithm step by step to compute the word products T, Q, and U sequentially as listed in Algorithm 2.1. Since the multiplier circuit is pipelined, it could achieve a hypothetical throughput of one product per cycle, but this is not available for a Montgomery operation. Interproduct dependencies among the products $T \rightarrow Q \rightarrow U$ preclude overlapping them for increased throughput.

3.2 Eberle Serial Digit-Digit Architecture

H. Eberle, *et al.* describe a coprocessor extension to general purpose processors on the server side to support high performance public key cryptography [17]. The focus is on high performance as opposed to low power. The coprocessor supports both RSA and ECC.

This architecture operates serially and its computation unit is the digit. Large operands are split into k digits of size d, where $k = \lceil n/d \rceil$. Input operands are stored in memory and fetched sequentially digit by digit instead of in a large parallel load operation. Likewise, intermediate outputs and final results are stored to memory sequentially digit by digit. For a change in operand size, only the loop counter values for operand addressing need to be changed, whereas the digit datapath and data buses remain unchanged. There is no register file, only an accumulator register, so operands are read from and written back to memory with each instruction. The architecture builds upon one previously described by the authors in [18].

Digit multiplication is performed as described by Algorithm 2.4, except that Montgomery reduction is interleaved with partial product row computation. Instead of computing the entire $T = AB$ product followed by the Q and U steps, this architecture computes a digit-word product $A[i] \times B$ and then performs the Montgomery reduction on that. A digit-word product is computed sequentially by accumulating $A[i] \times B[j]$, $0 \leq j < k$. Once Q_i is computed, the partial product is reduced sequentially by accumulating with $Q_i \times M[j]$, $0 \leq j < k$. All digit-word partial products are accumulated in a sliding region of product memory P. This approach might be termed Montgomery Micro Reduction. A formal mathematical definition of the computations is (3.1).

$$ABR^{-1} \bmod M \equiv \left(\sum_{i=0}^{k-1} 2^{id} A[i] \times B + \sum_{i=0}^{k-1} 2^{id} Q_i[0] \times M \right) 2^{-kd}, 0 < Q_i[0] < 2^d \qquad (3.1)$$

The first summation represents the conventional partial product summation of digit-word products. The second summation represents the Montgomery reduction digit-word products. This sum is effectively divided by $R = 2^n = 2^{kd}$, not by shifting the partial products right, but by incrementing the P address.

Because the Montgomery reduction is interleaved with partial product computation, there are dependencies both within partial product words and between partial product words. Fig. 3 depicts a dependency graph for the Eberle digit-serial architecture in which operands are split into $k = 2$ digits. Numbers along the top of the graph indicate computation dependency steps.

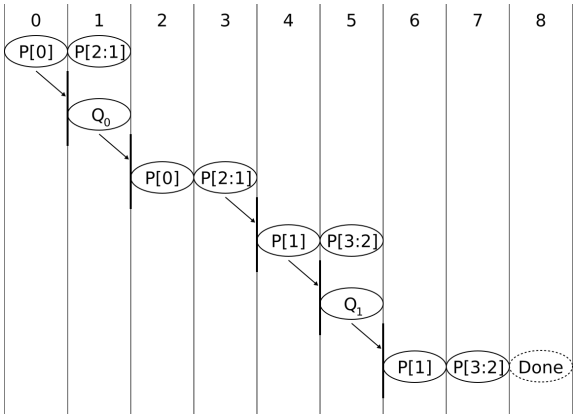


Fig. 3. Eberle architecture dependency chain ($k = 2$).

The figure shows that the Eberle architecture has a chain of seven dependency steps. The first partial product digit stored in $P[0]$ in Step 0 is required to compute Q_0 in Step 1. Q_0 is required before the reduction can start in Step 2. Reduction starts in Step 2 and modifies the digits stored in $P[2:0]$. $P[2:1]$ is required for the next partial product accumulation. With sequential digit accumulation, $P[1]$ is not fully computed until Step 3. Accordingly, the next partial product computation cannot start until Step 4. The same

dependencies exist for the second partial product as for the first, and so $P[1]$ in Step 6 precedes $P[3:2]$ in Step 7. There is a cyclic dependency pattern centered on the product memory P. Partial product computation is followed by in-place reduction, which is followed by in-place computation of the next partial product, and so on.

3.3 Großschädl Serial Bit-Word Architecture

Großschädl, *et al.* present a bit-word serial multiplier which can compute an n-bit Montgomery product in n cycles, plus some overhead [19]. As a bonus, the architecture can compute a conventional product $P = AB$ in $n/2$ cycles plus the overhead.

In contrast to the digit-digit architecture proposed by Eberle, *et al.* in [17], 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. No actual multiplier circuit is required, as a partial product a_iB can simply be computed by a row of n AND gates. This is accumulated in place in an accumulator with previous the 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 required to generate q_i, it can be shown mathematically to resolve to bit 0 of the current partial product word P_i before reduction. Computed in this way, the Montgomery product of A and B is defined by (3.2).

$$AB2^{-n} \bmod M \equiv \left(\sum_{i=0}^{n-1} 2^i a_i B + \sum_{i=0}^{n-1} 2^i q_i M \right) 2^{-n}, q_i \in \{0,1\} \tag{3.2}$$

The first summation represents the conventional partial product summation using bit-word products. The second summation represents the Montgomery reduction terms, one for each q_i, $0 \le i < n$.

Accumulation of partial products is performed in two cascaded n-bit carry save adders (CSAs). The accumulator register complex consists of two registers, RS and RC, which hold the partial product in redundant carry save form. After all partial products are computed and Montgomery-reduced, RS and RC are summed sequentially in a small digit adder of size d, and shifted back into RS. In this mode, RS and RC are right-shifted digit by digit instead of bit by bit. Typical sizes of the d-bit adder may be 8, 16, or 32 bits. Computing the nonredundant form requires n/d cycles.

This architecture has both parallel and sequential aspects to operation. The parallelism is in computing and accumulating the whole bit-word product $P_i = P_{i-1} + a_iB + q_iM$. This is computed and then right-shifted by one bit in a single cycle. q_i is determined in combinational logic in the same clock cycle. The sequential aspect is in accumulating all n Montgomery-reduced products P_i for $0 \leq i < n$, followed by converting the product to nonredundant form.

The dependency chain in the Großschädl architecture is simple. At a macro level, a sequential dependency exists from one bit-word product to the next. There is an intra-word dependency chain within a bit-word partial product: bit 0 of P_i before reduction $\rightarrow$ $q_i \rightarrow$ final P_i), but the simplicity of the q_i and q_iM computations means it can be resolved combinationally in one cycle. Fig. 4 illustrates the sequential dependency chain.

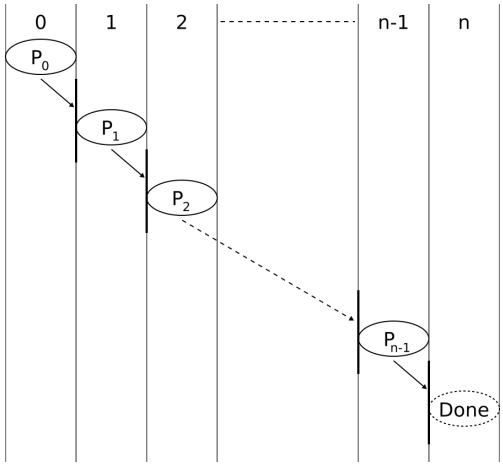


Fig. 4. Großschädl architecture dependency chain.

The figure shows that the first, reduced partial product P_0 is required to compute P_1, which is then required for P_2, and so on. Because this is a sequential architecture, the dependency exists from one cycle to the next, one for each of the n partial products.

3.4 Tenca and Koç Serial Hybrid Bit-Digit Architecture

A. F. Tenca and Ç. K. Koç present a Montgomery architecture that employs a simple bit-digit multiplier for a minimal area footprint [20], [21]. The architecture is scalable in two ways. First, it can accommodate arbitrary operand sizes. Second, multiple computation blocks can easily be cascaded to drive increased performance and throughput by increasing parallelism.

At a high level, this architecture computes Montgomery products in the same way as the Großschädl architecture described in the previous section [19], and (3.2) applies. The difference in this architecture is that both A and B are subdivided. As in the Großschädl architecture, A is split into n individual bits, denoted by a_i, $0 \leq i < n$. B is split into k uniform digits d bits in width, where $k = n/d$.

The main compute block is termed a Processing Element (PE). It is a two-stage miniature pipeline. In the first stage it sequentially computes partial product $P_i[j]$ digits from bit a_i and digits $P_{i-1}[j]$, $B[j]$, and $M[j]$ received from queues. The Montgomery division by two is performed on a digit basis on each $P_i[j]$ digit as it is transferred to the second stage register. This happens concurrently with the first stage computation of $P_i[j+1]$, whose LSB is copied into the MSB position of $P_i[j]$.

While being computed, P_i digits may be fed back to the same PE via a queue, or can be can be routed to another PE in cascade which is computing P_{i+1}. There is a two-cycle delay until the next PE can start because of the two stage PE pipeline. Other researchers have proposed optimizations to reduce the two-cycle delay to one cycle [22]. Any number of PEs may be cascaded in this way for increased throughput. There is generally little performance benefit if the number of PEs exceeds $k/2$.

The architecture must compute a total of n bit-word partial products and n partial product reductions. For a bit-digit partial product P_i, a PE computes k partial product digits and k digit reductions, but the reduction is performed combinationally. The entire design must compute a total of nk bit-digit products.

Dependencies in the Tenca and Koç architecture occur both between sequential partial products P_i and within those products. As with the Großschädl architecture, each partial product P_i requires the previous product P_{i-1}. The difference is that P_i does not need to wait for P_{i-1} to complete before starting, due to the bit-digit serialization. Within a product P_i there are both backward and forward dependencies. As used here, "backward dependency" indicates a typical dependency of a computation on the result of a previous computation. "Forward dependency" indicates that a computation has a dependency on a *subsequent* result. The backward dependencies are that the reduction digits $q_iM[j]$ require the quotient bit q_i, which in turn is a function of $P_{i-1}[0] + a_iB[0]$, but

all of this is resolved in combinational logic. Each partial product digit $P_i[j]$ depends on the carry out of $P_i[j–1]$. The forward dependency is caused by the division by two (right shift) step in Montgomery reduction. In this case the $P_i[j]$ digit is right-shifted, dropping its LSB, and receiving as its new MSB a copy of the LSB of the subsequent digit $P_i[j+1]$. Consequently, two cycles are required to effectively compute a $P_i[j]$ digit, but the throughput is one digit per cycle.

The sequential digit processing of the Tenca and Koç architecture is what enables concurrent computation of bit-word partial products P_i and P_{i+1}. The PE computing P_{i+1} is merely offset two cycles later than the one computing P_i. Fig. 5 shows a dependency graph of the Tenca and Koç architecture.

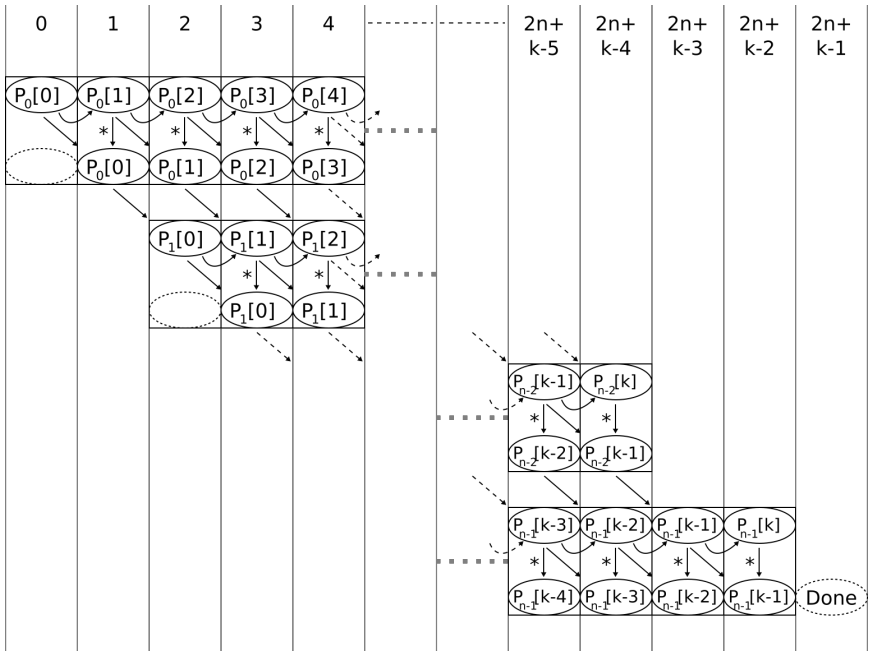


Fig. 5. Tenca and Koç architecture dependency chain.

The first partial bit-word product P_0 starts computation in Step 0, with digit 0 ($P_0[0]$). The next digit $P_0[1]$ cannot start until Step 1, because it depends on the carry out of $P_0[0]$. Also, $P_0[0]$ is not completely finished yet and so it transitions to Step 1 and down. There, its forward dependency on the LSB from $P_0[1]$ is shown by an arrow pointing down and annotated with an asterisk (*). Partial bit-word product P_1 is being computed in an adjacent PE. Because it requires P_0, it cannot start processing until Step 2.

The effective length of the dependency chain in this architecture is $2(n - 1) + k + 1 = 2n + k - 1$. The reason is that all n bits of one operand are processed in sequence to produce n bit-word partial products, as in the Großschädl architecture, and there is a two-step delay from the start of one partial product to the next. Processing k bit-digit products instead of a single bit-word product adds k steps. This imposes a lower bound on the number of cycles required to compute a Montgomery product, after which there is no way to increase performance by adding more PE resources.

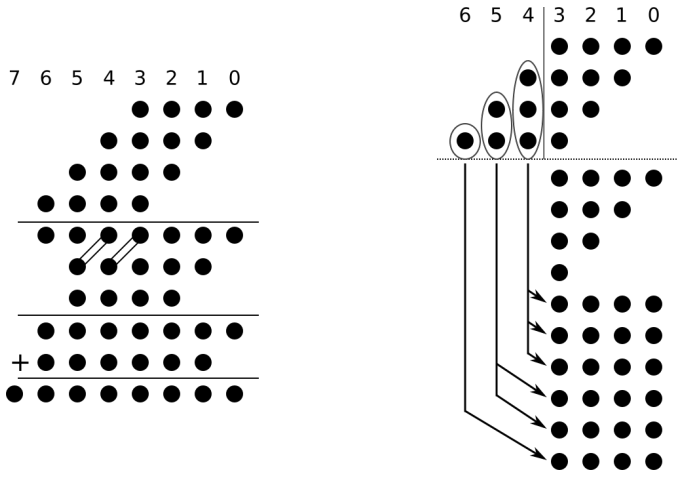
3.5 Sanu Parallel Architecture

Sanu, *et al.* describe a fully combinational Montgomery multiplier architecture in [23]. This design integrates the Montgomery reduction terms into the partial product reduction array. The authors demonstrate a mathematical transform that permits the Montgomery reduction algorithm to be performed by a vector summation of h n-bit numbers. The key is that the sum is congruent to the $n \times n$-bit product. The summation and reduction may be computed in $\log_{1.5}(h)$ time [23].

First, the authors present the congruence expression, which is the same as (3.2). Second, the authors suggest a possible substitution:

$$\sum_{j=0}^{h-1} X_j \equiv \left(\sum_{i=0}^{n-1} 2^i a_i B \right) \bmod M \tag{3.3}$$

The left side of the substitution shows the summation of h n-bit numbers from the set $\{X_0, X_1, \ldots, X_{h-1}\}$. In this case the subscripts are merely identifiers and do not signify digits or bits of X. If for all X_j, $0 \leq X_j < 2n$, then the authors show mathematically that the overall summation result will grow minimally beyond n bits. A conventional multiplier array grows to $2n$ bits wide. Montgomery reduction performed as in Algorithm 2.1 effectively zeroes out the lower n bits, leaving an n-bit Montgomery product in the upper bits. The transform used by this architecture keeps the reduction array more "vertical" with the final result growing minimally beyond n bits. Fig. 6 illustrates the transform with a trivial 4-bit example.



(a) Initial Dadda array. (b) Transform.

Fig. 6. Sanu Montgomery multiplier array [23].

Fig. 6(a) shows a 4×4 Dadda dot array of bit partial products, and how they are reduced through succeeding stages of half adders and full adders [24]. The final two

rows are summed in a conventional carry propagate adder. The first row of dots indicates the a_0B partial product in columns n–1 down to 0. The second row indicates a_1B in columns n down to 1, and so on. The array grows to eight bits wide., and then a_2B and a_3B, each shifting one bit further to the left. Ultimately the final product will be eight bits wide. Not shown explicitly are the interleaved Montgomery reduction rows between partial product reduction rows.

Fig. 6(b) shows the substitutions designed to minimize horizontal growth in the array. Effectively, the product bits to the left of column 3 in the initial array are replaced with congruent terms fetched from a lookup table (LUT). The original array has been transformed to a series of summands derived from the left hand side of the array and the LUT. The authors propose variants in which the MSBs are combined in different ways so as to minimize the number of congruent terms and LUT size.

A key point is that Montgomery reduction steps are interleaved with the summation. The interleaving is similar but not identical to that employed by the Großschädl and Tenca and Koç architectures, and is effected spatially instead of sequentially. Groups of three summation rows are Montgomery reduced with interleaved modulus M terms. This is omitted form the figure for clarity. It is what makes it possible to compute the modular product in logarithmic time.

This architecture is fully parallel (combinational). It uses a traditional Dadda type multiplier array for the partial products [24]. A simple Dadda type multiplier starts with n rows, and has approximately $\lceil \log_{1.5}(n) \rceil$ reduction stages. A naive implementation of interleaved Montgomery reduction would double the original n rows to $2n$. The Sanu, *et al.* architecture cleverly applies the interleaving to groups of three rows for approximately $\lceil n/3 \rceil$ modular reduction rows, for an initial total of $n + \lceil n/3 \rceil$.

Furthermore, it minimizes array growth (both width and height) by replacing the upper bits with a smaller number of congruent reduction terms fetched from a lookup table.

Dependencies exist within groups of rows and between stages. Each group of three or fewer partial product rows will potentially be Montgomery-reduced, based on the sum of the LSBs of the three rows. This bit selects either the modulus M or a row of zeroes, and so it is dependent on the sum of those three rows. Each group of rows can be taken separately from the other groups, at that stage. The output rows of that stage then become the input rows of the next partial product and modular reduction stage.

Chapter 4 Serial Montgomery Design Space

The preceding review of several Montgomery hardware architectures reveals a wide range of fundamental approaches to the problem, as well as specific implementation choices that can be made. Some architectures employ arithmetic circuits that operate on full, word-sized operands. For a serial approach, other architectures split the operands into smaller parts, such as digits or bits. Digit or bit scale arithmetic circuits of relatively low complexity are employed in a serial algorithm to build up the final result. As the preceding review has shown, there is no requirement for symmetry in operand division—one operand may be split into bits while the other operand is split into digits. The number of basic operations that must be performed, such as digit multiplication, determines the performance of an architecture.

4.1 Koç Montgomery Classification

Koç, *et al.* review and classify serial Montgomery multiplication algorithms [25]. The classification scheme they devised has become the *de facto* taxonomy for characterizing serial Montgomery realizations. Most research on Montgomery multiplication references it and classifies proposed architectures into one of its categories [10], [16], [19], [20], [21], [22], [26], [27], [28], [29]. It is applicable to both software and hardware realizations, and may be termed the serial Montgomery classification or taxonomy.

The classification scheme has two major dimensions. First, it considers whether Montgomery reduction is separated from or integrated with product computation. Integrated reduction may be further classified as either coarse or fine. Second, it considers whether input operand digits or product digits are prioritized for scanning.

Computing a Montgomery product step by step as listed in Algorithm 2.1 is one example of separated reduction. The initial product T is computed first. Then the quotient term Q is computed. Finally, the reduction term U is computed and added to T to perform the reduction. Each of these steps may be considered to be atomic, no matter whether performed in one cycle in a large multiplier, or over several cycles using a digit-sized multiplier.

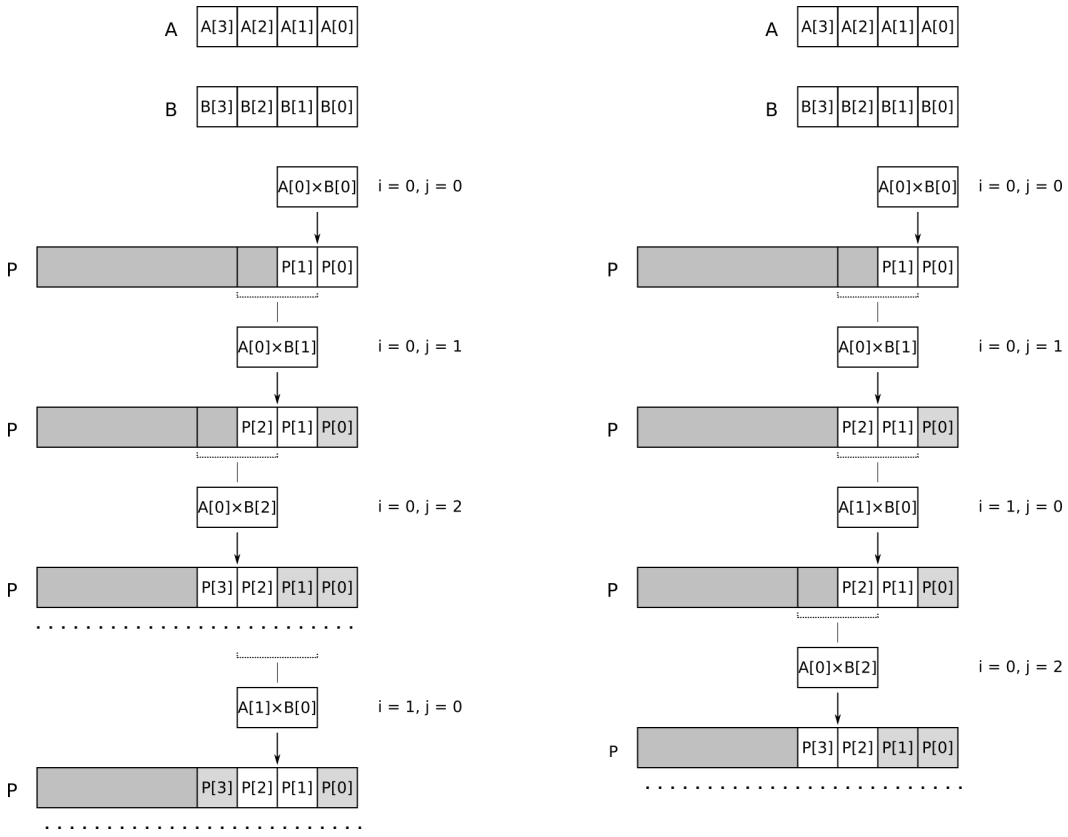
As an alternative to separated reduction, reduction can be integrated with product computation. A partial product is computed, and then a Montgomery reduction is performed on it. This process continues for each partial product, so that a series of partial products and reductions are performed and accumulated into one final Montgomery product. In coarsely integrated scanning, the partial product computation is followed sequentially by a separate reduction operation. Partial products alternate with reduction operations. With finely integrated scanning, both the partial product computation and reduction are computed in the same step.

Digit scanning is treated as a separate, orthogonal parameter. It 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 operands are given scanning 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.

Consider an example of operand scanning. Assume operands are split into four digits each, and let multiplier A be indexed by i in an outer loop, and multiplicand B indexed by j in an inner loop. For the first $i = 0$ loop iteration digit $A[0]$ is fetched and the digits $B[j]$ of B are scanned in succession for $j = 0, 1, 2, 3$. The product is

accumulated into $P[k]$ where $k = i + j$. The five-digit digit-word partial product will reside in $P[4:0]$. For the next iteration of i, partial product accumulation is moved one digit to the left, so $A[1]\times B$ will be accumulated into $P[5:1]$.

Alternatively, priority may be assigned to product digits. In this case, 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. Thus the product digits are written in order as $P[0]$, $P[1]$, $P[2]$, …, *etc.* Fig. 7 illustrates operand priority scanning and product priority scanning.



(a) Operand scanning. (b) Product scanning.

Fig. 7. Scan priority for partial product assembly [25].

Observe that in Fig. 7(a), product digit $P[1]$ is first computed in iterations $(i, j) = (0, 0)$ and $(0, 1)$ when partial product $A[0]\times B$ is being computed. Later in iteration $(i, j) = (1, 0)$, it is written again when partial product $A[1]\times B$ is being computed. Conversely, in Fig. 7(b), all digit computations that contribute to product digit $P[1]$ are computed in immediate succession, and index variables i and j are cycled accordingly. After $P[1]$ is completed, subsequent iterations need never revisit it.

Altogether, Koç, *et al.* identify five iterative Montgomery algorithm classes. They are listed in Table 3.

Table 3. Koç, *et al.* serial Montgomery architecture taxonomy.

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.

Beyond functioning as a shorthand for concisely describing a Montgomery realization, the Koç taxonomy provides useful expressions for evaluating performance and storage requirements. 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 $2k + 2$.

For an architecture that fits into the classification scheme, this provides a useful starting point for making performance estimates. Table 4 lists the architectures reviewed in Chapter 3 along with their Koç, *et al.* classification, if applicable.

Table 4. Montgomery architecture classification.

Architecture	Base Operand	# Base Operations	# Cycles	Koç Classification	Notes
McIvor[a]	Word	3	$3p + 1$	—	
Eberle	Digit	$2k^2 + k$	$2k^2 + k$	CIOS	
Großschädl	Bit/word	n	$n + k$	FIOS[b]	Bit-word
Tenca & Koç	Bit/digit	nk	$2n + k - 1$	FIOS[b]	Bit-digit
Sanu	Word	1	1	—	

[a]Number of pipeline stages p.
[b]Closest fit.

The McIvor and Sanu architectures operate on word size operands, so computing their respective numbers of operations is straightforward. Because the McIvor architecture has a pipeline of depth p and computes the three intermediate products T, Q, and U sequentially (with no overlap possible), it requires $3p$ cycles, plus an additional cycle for the final $T + U$ sum. The Sanu architecture is fully combinational and requires only a single cycle, although that cycle may be of long duration. Neither of these fits into the Koç serial classification scheme.

The Eberle architecture operates at the digit level and integrates reduction operations, once per each digit-word partial product. It may be classified as Coarsely Integrated Operand Scanning (CIOS), and it requires a minimum of $2k^2 + k$ cycles.

The Großschädl and Tenca and Koç architectures use mixed operand sizes, bit-word and bit-digit respectively. Because they do not divide A and B operands symmetrically, they do not precisely fit the Koç classification scheme. The Großschädl architecture computes n bit-word products, and requires $n + k$ cycles to complete. The

final k cycles are required to convert the carry save product into nonredundant form in a digit multiplier. Priority is given to scanning the bits of operand A in succession, and reduction is performed combinationally (finely) during partial product computation. Because of the foregoing, it can be classified as FIOS. The Tenca and Koç architecture computes nk bit-digit products and requires $2n + k - 1$ cycles to complete. It 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.

4.2 Serial Montgomery Model

Despite its universal acceptance, the serial Montgomery taxonomy suffers from two major limitations. First, it omits possible reduction and scanning combinations. Second and more importantly, 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 in a subsequent chapter. 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*. In its present form, the classification scheme considers only serial realizations in which a single digit multiplier or multiply-accumulate (MAC) unit performs each multiplication in sequence. Indeed, use of the term *scanning* in reference to operand or product digit priority reflects this one-dimensional conception of digit

processing. It can be shown, however, that many digit computations can be performed concurrently. For portions of a particular Montgomery realization that can be parallelized, simply adding a second digit multiplier can cut those portions' latency in half. Employing digit level parallelism, a Montgomery architecture's performance can therefore receive a substantial performance boost at relatively low cost. At present the serial Montgomery taxonomy is strictly two dimensional. Its two axes represent separated versus integrated reduction, and operand versus product priority digit scanning. Expanding the classification scheme to encompass digit level parallelism provides it with a third dimension, converting the scheme from a flat surface to a large volume of descriptive and analytic power.

It is possible to parallelize portions of any serial Montgomery architecture selectively for targeted performance enhancement. 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 taxonomy can be expanded to apply to realizations with two or more digit arithmetic units. Accordingly, this Serial Montgomery Model can be used effectively to classify and characterize Montgomery architectures that schedule multiple concurrent digit operations. Table 5 lists the categories from the improved classification scheme. Novel categories are indicated in bold font.

Table 5. Serial Montgomery Model.

Reduction		Digit Scheduling					
		Operand		Hybrid		Product	
		Serial ($m = 1$)	Parallelized ($m > 1$)	Serial ($m = 1$)	Parallelized ($m > 1$)	Serial ($m = 1$)	Parallelized ($m > 1$)
Separated		SOS	**SOS/*m***	[a]	[a]	**SPS**[b]	**SPS/*m***[b]
Integrated	Coarse	CIOS	**CIOS/*m***	CIHS	**CIHS/*m***	[a]	[a]
	Fine	FIOS	**FIOS/*m***	[a]	[a]	FIPS	**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 dissertation.

Each scanning 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 original category from the Koç serial taxonomy is carried forward to this column. 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.

Enhancing the taxonomy comprises more than merely adding and modifying category names. The entire purpose of adding the third dimension of digit level parallelism is to provide a means to estimate performance and resource requirements. This can be especially useful when comparing two disparate architectures that employ digit level parallelism in different ways.

Estimating the performance impact of increased parallelism in a serial Montgomery architecture is nontrivial. It is not as simple as adding a digit multiplier and dividing the cycle count by the total number of multipliers. Dependencies may prevent some digit computations from being performed concurrently. The unique characteristics and dependency relationships of each architecture determine where and to what degree

digit parallelism can be increased. Furthermore, dependency relationships can determine whether operand or product scanning is best.

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. In all cases, $2k^2 + k$ digit multiplications are required, and it can be assumed that at least that number of cycles is required. It would be possible to construct an SPS architecture that requires $2k^2 + k$ cycles. The preferred realization, however, requires $2.5k^2 + 0.5k$ cycles. It is described in Chapter 5. Although this digit multiplication count is higher, 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.

Estimating the performance effect of adding parallel digit multipliers requires analysis of the underlying digit operations and their dependency relationships. For example, consider architectures in the CIOS category. CIOS requires a total of $2k^2 + k$ digit multiplications, which are performed sequentially with a single digit multiplier. This category employs operand scheduling to compute and reduce a digit-word partial product $A[i] \times B$ in each iteration i via three dependent steps:

1. Compute and accumulate k digit products $A[i] \times B[j]$, $0 \le j < k$.
2. Compute one quotient digit Q_i.
3. Compute and accumulate k reduction digit products $Q_i \times M[j]$, $0 \le j < k$.

The first step proceeds, and indeed all the digit products could be computed in parallel if sufficient digit multipliers are available. The second step for Q_i cannot be performed concurrently because Q_i depends on the first step. The third step depends on Q_i, and so it cannot be executed in parallel with Q_i. These dependencies thus impose a sequential ordering of digit multiplications as follows: k, 1, k. The group of three steps

must in turn be performed for all k digits of A. The total sequence then consists of $(k, 1, k)_0$, $(k, 1, k)_1$, …, $(k, 1, k)_{k-1}$ digit multiplications. Because each group depends on the previous group's result, they must be performed sequentially. This requires $k(k + 1 + k) = 2k^2 + k$ cycles.

Employing $m > 1$ digit multipliers increases parallelism, but only for those digit operations that can be performed concurrently. In the preceding example, the partial product and reduction steps can each benefit from parallelization, whereas the quotient term Q_i cannot. The sequence for a single digit-word partial product becomes $\lceil k/m \rceil$, 1, $\lceil k/m \rceil$. Because k sequences must be performed, the total number of cycles required to compute a Montgomery product is $k(\lceil k/m \rceil + 1 + \lceil k/m \rceil)$. If $m = k$, then this reduces to $k(1 + 1 + 1) = 3k$ cycles. Table 6 lists the digit scheduling sequences for selected Serial Montgomery Model categories for both the strictly serial mode ($m = 1$) and modes with some degree of digit parallelism ($m > 1$).

Table 6. Selected Serial Montgomery Model digit schedules and cycles.

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

Schedule order for the proposed SPS category differs subtly from that for SOS, CIOS, and FIOS. In the latter three categories, the schedule follows a strict digit order that is imposed by partial product/reduction dependency ordering. By contrast, the SPS has a macro level dependency order only. There is no alternating dependency chain of the form (partial product, reduction, partial product, …) as in the integrated scheduling

categories. Because SPS employs product digit scheduling, one phase need not complete before the next one can begin. In this way, the phases may be overlapped, though they need not be. If they are overlapped, the dependency order guarantees that all digit dependencies from one phase to the next are available before the next phase begins, as long as $m \leq k^2$. Chapter 5 explains the proposed SPS dependency chain more fully.

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 6 determine how many cycles are required for different values of m. Table 7 lists the cycle count for the respective categories for m, where $1 \leq m \leq 5$.

Table 7. 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 the listed categories. Some categories benefit more than others. For $m = 1$, the categories SOS, CIOS, and FIOS all require 36 cycles, whereas the proposed SPS category is worse in that it requires 42 cycles. If m is increased to 2, however, SOS and CIOS improve to 20 cycles and SPS improves to 21 cycles, but 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 only requiring 14 cycles. In fact, for $m = 4$ and 5, SPS is again faster than the other three categories.

Chapter 5 Montgomery Algorithm Optimization

Closer examination of the McIvor, *et al.* multiplier and the Montgomery algorithm listed as Algorithm 2.1 suggests a few possible ways to improve performance. Consider the computation steps. The steps include three multiplications, two modulus functions, an addition, and a division which can be computed as a simple right shift. Fig. 8 illustrates the steps.

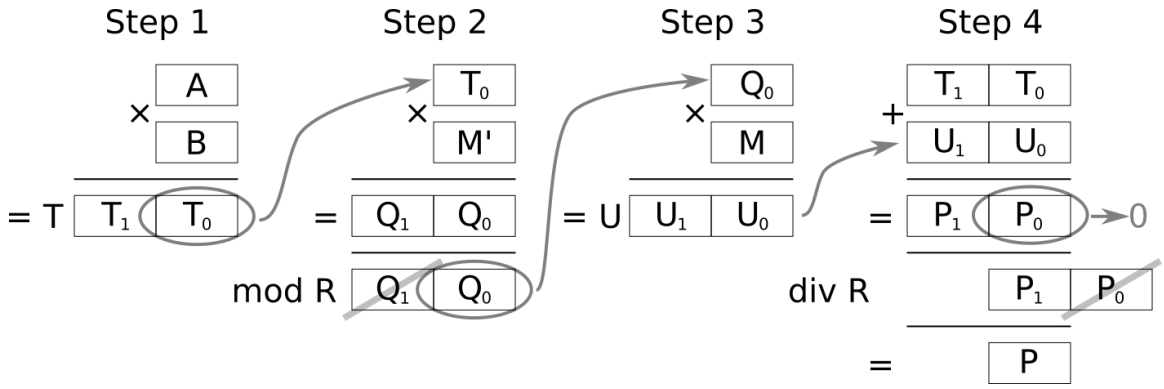


Fig. 8. Montgomery computation steps [11].

From the figure, it is evident that some portions of the intermediate results are not required immediately, or are not used at all. Step 1 computes an initial product T of M-residues A and B. Its upper and lower halves are designated as T_1 and T_0 respectively. In Step 2, the quotient Q is computed from T_0 and M'. The upper half, designated as Q_1, is discarded, and only the lower half Q_0 is used as the operand in the following step. Next in Step 3, the Montgomery reduction term U (U_1 and 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. The upper half, T_1, is not required until Step 4. It may be possible to reduce overall latency if the computation of T_0 can be expedited, and the computation of T_1 overlapped with Step 2. Because Q_1 is discarded at the end of Step 2, simply not computing it could save time and energy. This property is rarely mentioned explicitly in the literature; brief exceptions are found in [19] and [30]. 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 [31]. 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 $Q = 0$, 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.

5.1 Rescheduled Montgomery Multiplication

Rescheduled Montgomery Multiplication enables efficient computation of the Montgomery product by minimizing unnecessary computations and deferring some other computations. The Montgomery algorithm as listed in Algorithm 2.1 requires three multiplications of n-bit operands to produce three $2n$-bit products T, Q, and U. It has already been noted that the more significant half of Q, designated as Q_1, is not used and so need not be computed. Furthermore, the more significant half of T (T_1), does not need to be computed in full until U_1 is computed.

In order to exploit the foregoing properties, the architecture devised here uses digit multiplication. Input operands are split into k digits of d bits each. One or more

digit multipliers compute partial products, which are then summed in an accumulator. Generally, there are fewer digit multipliers than there are digit products to compute, even as few as one multiplier. Multiple phases of digit multiplication and accumulation are required.

According to (2.2) and Algorithm 2.1, computing T, Q, and U as full products would require $3N_P = 3k^2$ digit multiplications. However, in the Rescheduled Montgomery algorithm, the top half of the Q product (Q_1) need not be computed. Fig. 9 illustrates the computation of Q_0. 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.

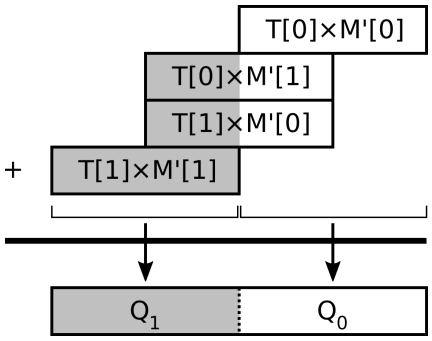


Fig. 9. Digit multiplication for Q.

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

$$N_Q = (k^2 + k)/2 \tag{5.1}$$

$N_Q < N_P$ and the ratio of digit multiplications to compute Q_0 to the number to compute a full product is $r_Q = N_Q/N_P$. It is evident that as k increases, this ratio approaches 1/2.

$$r_Q = \frac{N_Q}{N_P} = \frac{\left(k^2 + k\right)/2}{k^2} = \frac{k+1}{2k} \tag{5.2}$$

$$\lim_{k\to\infty} \frac{k+1}{2k} = 0.5 \tag{5.3}$$

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

$$N_M = 2.5k^2 + 0.5k \tag{5.4}$$

The ratio r_R of digit multiplications in the Rescheduled Montgomery algorithm relative to the full approach is described in (5.5).

$$r_R = \frac{2.5k^2 + 0.5k}{3k^2} = \frac{5k+1}{6k} \tag{5.5}$$

As k grows large, the ratio asymptotically approaches 5/6.

$$\lim_{k\to\infty} \frac{5k+1}{6k} = \frac{5}{6} = 0.8\bar{3} \approx 83\% \tag{5.6}$$

The Rescheduled Montgomery Multiplier employs the novel Separated Product Scheduling method from the Serial Montgomery Model. From (5.4), it is evident that it requires a larger number of digit multiplications than the other categories such as CIOS, FIOS, *etc.* Despite this, the Rescheduled Montgomery architecture is inherently equipped to employ digit level parallelism more efficiently than the previous categories. Both fundamental aspects of the SPS mode, separated reduction, and product scheduling, make this possible. Separated reduction as employed here ensures that the dependencies occur only at the macro level between products T, Q_0, and U. There is no alternating dependency chain between partial product and reduction phases. Digit level parallelism can therefore be fully applied within a phase without interruption by an intervening dependency. Nevertheless, dependencies from one phase to the next can be broken down into digit level dependencies, which means that one phase need not complete in full before the next phase can begin. As a result, it is possible to schedule digit operations from two phases concurrently straddling the temporal boundary between the two phases. It also minimizes stalls when operations are pipelined. Fig. 10 illustrates the SPS dependency chain for the Rescheduled Montgomery Multiplier with $k = 2$.

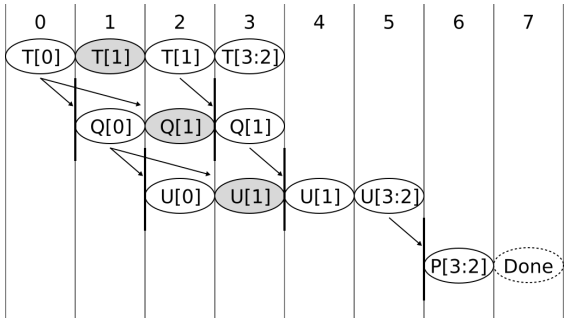


Fig. 10. Rescheduled Montgomery Multiplier SPS dependency chain, $k = 2$.

The first three rows of bubbles in Fig. 10 correspond to the intermediate product phases T, Q_0, and U. As depicted, each of these phases is split into steps corresponding to digit multiplications, with the product digits that are being worked on indicated. Refer to Fig. 2 for T and U, and Fig. 9 for Q_0. Because reduction via Q_0 and U is *separated* from T partial product generation, there is no alternating dependency chain in which a later partial product depends on a previous reduction. The CIOS dependency chain of (partial product, quotient, reduction, partial product, …) does not apply. In the Rescheduled Montgomery Multiplier, the SPS dependency chain is strictly a single sequence of (product, quotient, reduction), full stop.

Moreover, as the figure illustrates, the dependencies broken down to the digit level make it possible to overlap T, Q_0, and U computation. This feature can be useful in some cases. Separated product scheduling largely permits increased digit level parallelism within relatively large phases uninterrupted by alternating dependency chains. If k^2 is not divisible by m, then at the end of one phase fewer than m multipliers are required for the remaining digit products in that phase. The remaining digit multipliers need not be idled; instead they can be scheduled to begin computation of the next phase. The new phase's first few digit products depend on digits already computed early in the phase that is just completing.

Fig. 11 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.

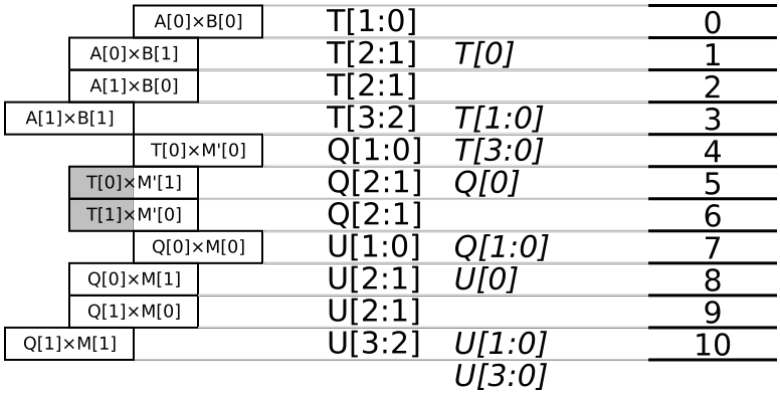


Fig. 11. Digit product scheduling for $k = 2$, $m = 1$.

The figure shows only the scheduling of the digit products and their offsets, and does not indicate partial product accumulation. It should not be read to imply that all of the digit products shown are summed together. There are, in fact, three phases in which the same hardware is reused: computing T in Cycles 0-3, computing Q_0 in Cycles 4-6, and finally computing U in Cycles 7-10.

A detailed, step-by-step description of the schedule is as follows. The first row shows Cycle 0 in which digit product $A[0]\times B[0]$ is computed, contributing to result digits $T[1:0]$. The digit product is latched in the accumulator register. The $T[0]$ digit is complete and available *after* this cycle; it is listed in italics on the next row, *i.e.* in Cycle 1. In Cycle 1, $T[1]$ is not yet complete, because it also depends on accumulation of the digit products $A[0]\times B[1]$ and $A[1]\times B[0]$. These latter two are computed in Cycles 1 and 2, and $T[1]$ finally becomes available in the accumulator register in Cycle 3. This process continues until all four digits of T are available in the accumulator register in Cycle 4. In the second phase, computation of Q_0 starts in Cycle 4 with digit product $T[0]\times M'[0]$. The digit products $T[0]\times M'[1]$ and $T[1]\times M'[0]$ are computed in Cycles 5 and 6, and straddle the line between the unused Q_1 ($Q[3:2]$) and Q_0 ($Q[1:0]$). Their upper halves are shaded to indicate that they are not used. The final digit product $T[1]\times M'[1]$, which contributes

only to the unused Q_1, is skipped. Finally, computation of U occurs in Cycles 7-10. The figure shows that, for $k = 2$ digits and employing $m = 1$ digit multiplier, computing the Montgomery product requires only 11 instead of 12 digit multiplications, or about 91.7%.

Increasing the number of digits into which the operands are subdivided permits a higher degree of granularity. As shown by (5.3), higher granularity permits eliminating an increasing 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, (5.6) shows that the savings from reducing Q_1 computations approaches 17%.

Another example will illustrate the increasing savings. Suppose now that $k = 4$. Computing a full product requires $N_P = 4^2 = 16$ digit multiplications (T and U), while computing Q_0 requires only $N_Q = (16 + 4) / 2 = 10$ multiplications, or 62.5%. Thus for $k = 4$, the digit multiplication approach results in a further savings by eliminating unnecessary computations. From (5.5), $r_R = 42 / 48 = 87.5\%$. This compares even more favorably to the $k = 2$ case where $r_R = 91.7\%$.

It is, of course, possible to employ more than a single digit multiplier. A plurality of multipliers can be scheduled concurrently for increased parallelism and lower latency. At the same time, the increased granularity afforded by digit multiplication to minimize Q_0 computations remains.

In general, a digit-based architecture must perform the following operations for each set of digit products. First, digits are selected from the full-sized operands; this can be done with a multiplexer tree. Next, the digits are multiplied together. Finally, the digit product is summed with the previous products' running sum in an accumulator register. Due to the relatively large digit sizes, with 32 bits being the minimum, it is expected that the digit multiplier will be the critical timing path. Accordingly, pipelining

the digit operations can increase the throughput and improve the overall latency of a Montgomery product computation.

5.2 Architecture

The Rescheduled Montgomery Multiplier architecture consists of a three-stage reusable pipeline with stages designated as Load (L), Multiply (M), and Accumulate (A). 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. 12 depicts the pipeline for the $k = 2$, $m = 1$ case.

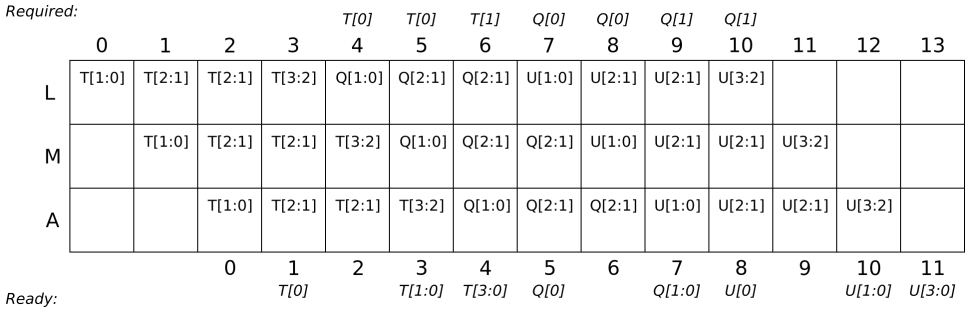


Fig. 12. RMM pipeline for $k = 2$, $m = 1$.

Data progress through the pipeline occurs vertically from top to bottom, through stages L, M, and A. Time is indicated horizontally, with the Load cycle count on top, ranging from 0 to 10. The pipeline boxes indicate the result computation that is being performed. Computation of $T[1:0]$ begins in Cycle 0, with digits $A[0]$ and $B[0]$ being loaded in Stage L. The actual digit multiplication $A[0]\times B[0]$ occurs in Stage M during Cycle 1. Finally, accumulation occurs in Stage A during Cycle 2. The 0 underneath the

stage indicates that accumulation is being performed on the result of the computation that started in Stage L during Cycle 0. Alternatively it can be considered as Accumulate Cycle 0. The accumulated result then becomes available in the accumulator register in Cycle 3 (Accumulate Cycle 1). The completed part of that result is $T[0]$, so it is shown below in italics. Recall that $T[1]$ requires two more digit multiplications before it is complete.

Because it is a pipelined architecture, the entire Montgomery product requires more cycles than the 11 implied by the digit product scheduling. Two additional cycles account for the pipeline depth. Two more cycles represent the final Montgomery $T_1 + U_1$ operation, which may include a subtraction of M if the initial result is out of range, for a total of 15 cycles.

Fig. 13 depicts the Rescheduled Montgomery Multiplier microarchitecture. Computation progresses from left to right. At the front end are the n-bit input registers A, B, M', and M. These registers, along with a feedback path from the accumulator register, feed digit selection logic. This logic is controlled by a counter in a separate finite state machine. It selects digits from the inputs to feed to one or more digit multipliers. The input registers and select logic form the Load stage of the pipeline. The Multiply stage contains the digit multipliers. 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, the Accumulate stage adds one or more digit products to the contents of the accumulator register ACCUM. Finally, computation of the Montgomery product occurs in the Final Sum stage. This stage is only active after all the intermediate operands T, Q_0, and U have been fully computed in the L-M-A pipeline.

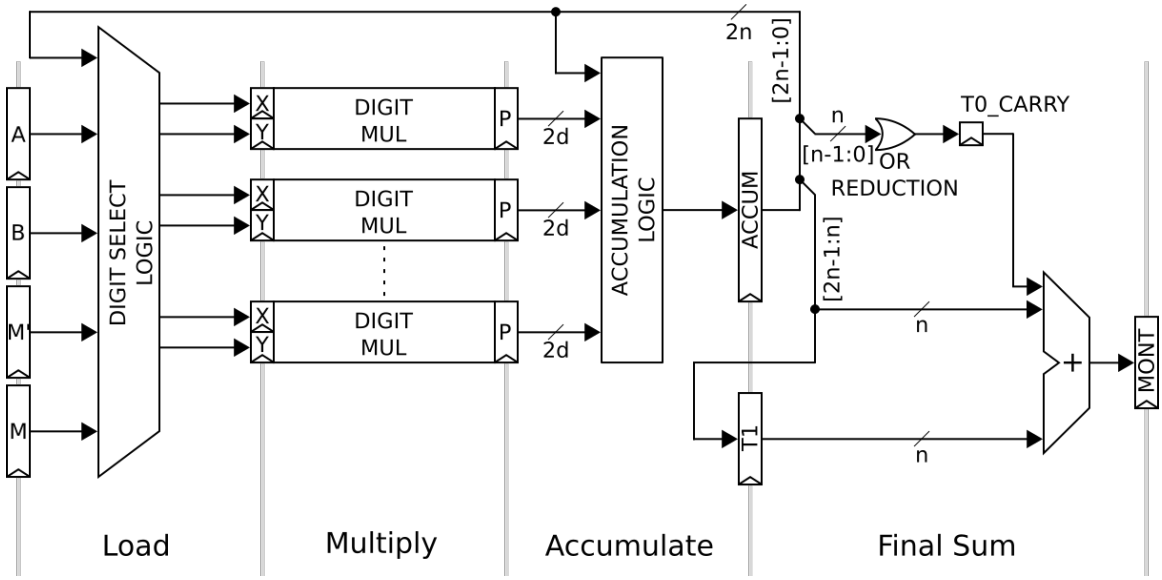


Fig. 13. Rescheduled Montgomery Multiplier architecture.

During the computation of T in Step 1, digits from A and B are selected according to the schedule. After some number of cycles determined by the number of digits and available number of digit multipliers, the computation of T_0 is complete and occupies the low order bits of the accumulator register ACCUM. T_1 computation may still be in progress. T_0 is checked for any non-zero bits by means of an OR reduction. If any of its bits is 1, the single-bit T0_CARRY register is set; if they are all zero, the register is cleared. This register is used later during final Montgomery computation and indicates whether a carry out of the $T_0 + U_0$ sum would have occurred. When the entire T word has been computed in the Accumulate stage, its upper word is moved to the T1 register, because subsequent phases for Q_0 and U will eventually overwrite ACCUM.

When the last digits of A and B have been clocked into the digit multipliers, Step 2 for computing Q_0 can begin in the Load stage. Q_0 is a function of T_0 and M'. T_0 digits are selected from the low order word of the accumulator register which is fed back to the

Load stage. M' digits are selected from its input register. When Q_0 is complete, it occupies the low order bits of the accumulator register just as T_0 had when it was completed during Step 1. Depending on the (k, m) design configuration, and due to the pipelining, it is possible that all of the T_0 digits will be read before the first partial result of Q arrives in ACCUM. In this case, T_0 would not need to be moved from the accumulator register. In other cases where T_0 digits are still being selected in the Load stage while the ACCUM is being written with intermediate Q results, T_0 can be saved in and then fetched from an auxiliary register T0. Because only Q_0 instead of the full value of Q is computed, Step 2 requires fewer cycles to complete than either Step 1 or 3. It stops after slightly more than half the number of digit multiplications that would be required to compute a complete Q.

After the last T_0 and M' digits have been input into the digit multipliers, Step 3 to compute U can begin. In the Load stage, the Q_0 digits are selected from the fed-back low word of the accumulator, while M digits are selected from the M input register. This computation proceeds just as in Step 1. Depending on the (k, m) design configuration, and due to the pipelining, it is possible that all of the Q_0 digits will be read before the first partial result of U arrives in ACCUM. In this case, Q_0 would not need to be moved from the accumulator register. In other cases where Q_0 digits are still being selected in the Load stage while the ACCUM is being written with intermediate U results, Q_0 can be saved in and then fetched from an auxiliary register Q0. At the end of this phase, the entire value of $U = (U_1, U_0)$ is present in the accumulator register ACCUM.

The accumulator datapath varies from configuration to configuration. The number of digits k into which the full sized operands are divided, and the number of digit multipliers m that are instantiated, along with the digit product scheduling, directly determine the structure and complexity of the datapath.

Consider the case where $k = 2$ and $m = 1$. Refer to Fig. 11 for the digit product scheduling for T. When any product computation phase starts, the accumulator register is first cleared to zero. The first digit product $A[0]\times B[0]$ is computed in Cycle 0 and written to the least significant bits of the accumulator register ACCUM. In Cycle 1, the next digit product $A[0]\times B[1]$ is left-shifted by d bits and added to ACCUM. In Cycle 2, $A[1]\times B[0]$ is also left-shifted by d bits and added. Finally in Cycle 3, $A[1]\times B[1]$ is left-shifted by $2d$ bits and added. Fig. 14 illustrates this process.

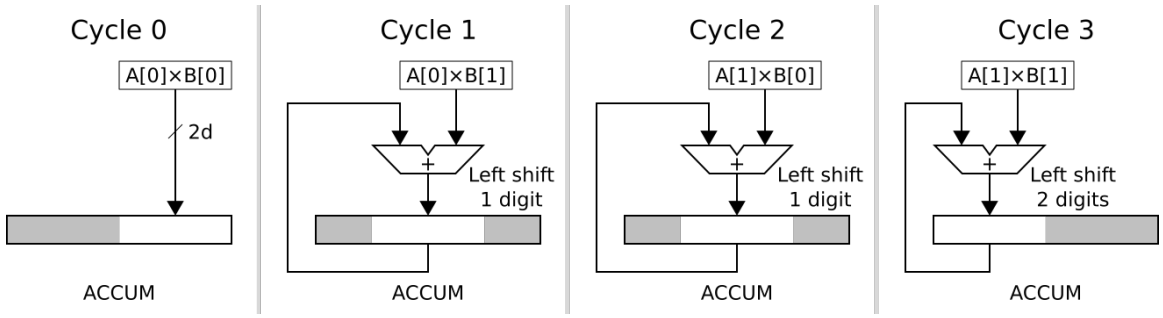


Fig. 14. Digit product accumulation for T ($k = 2$, $m = 1$).

The accumulator datapath logic thus must support three configurations, or modes: (1) in Cycle 0, no shift and no addition; (2) in Cycles 1 and 2, a one-digit left shift of the digit product, accumulated with the contents of ACCUM; and (3) in Cycle 3, a two-digit left shift and accumulation with ACCUM. This same logic can be reused exactly when computing Q_0 and U. Q_0 is computed in Cycles 4-6, corresponding to Cycles 0-2, while the accumulator mode of Cycle 3 is skipped. U is computed in Cycles 7-10, corresponding to Cycles 0-3.

As much as is practicable for a given (k, m) configuration, the same accumulation schedule should be used for the three T, Q_0, and U computation phases. This maximizes

gate reuse and minimizes combinational area growth in the accumulator datapath. Consider an architecture with $k = 3$ digits and $m = 2$ digit multipliers. From (5.4), 24 digit multiplications must be performed. Fig. 15 illustrates one possible schedule.

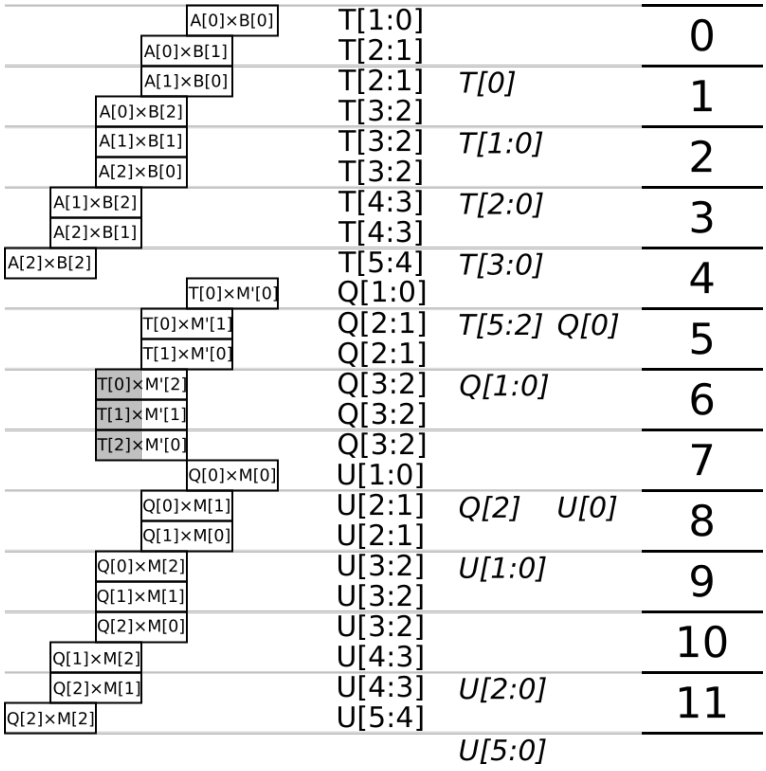


Fig. 15. Digit multiplication schedule for $k = 3$, $m = 2$.

In the figure, observe that the final digit product for T is computed in Cycle 4, as is the initial digit product for Q_0. There is a disconnect in that these two partial products are not added together. Moreover, in Cycle 0 the first *two* partial products for T are accumulated, but in Cycle 4 only the first Q_0 partial product is written to the accumulator. The second Q_0 partial product is not written until Cycle 5, during which it is added with the third partial product. Thus, the accumulation logic must be different for Cycle 4 from

that in Cycle 0. Cycle 7 is yet another case where the accumulation requirements diverge and are not consistent with Cycles 0 and 4. In all, the accumulator datapath requires ten different states. Cycles 0 through 7, 10, and 11 each requires a unique accumulation state. Cycles 8 and 9 can reuse the same states as Cycles 5 and 6 respectively.

By maximizing uniformity of accumulation for the three phases, the number of accumulator states can be reduced. Using a slightly different digit product and accumulation schedule, the $k = 3$, $m = 2$ architecture only requires five accumulator states. Fig. 16 depicts the revised schedule.

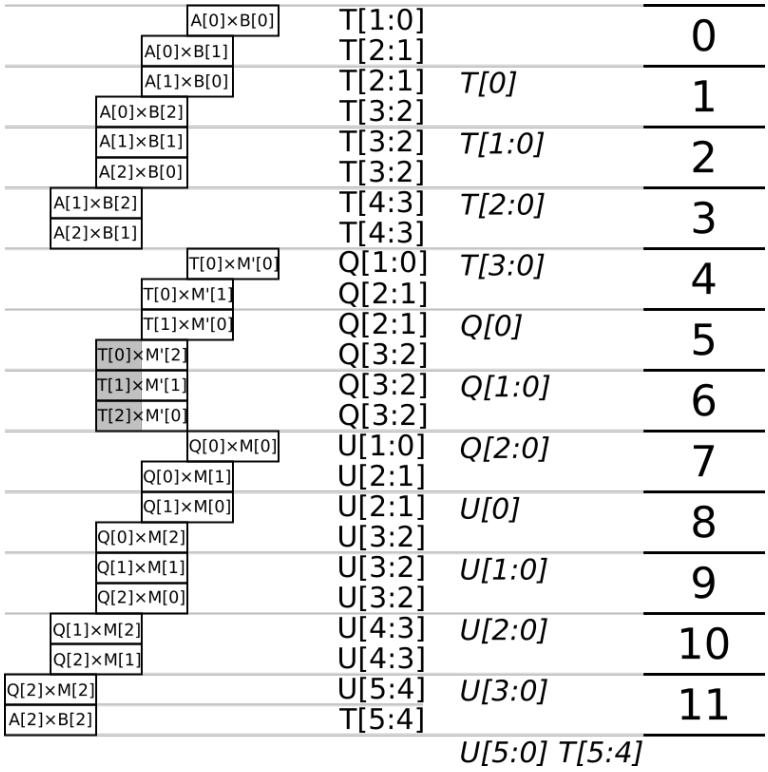


Fig. 16. Revised digit multiplication schedule for $k = 3$, $m = 2$.

Step 1 for computing most of T proceeds through Cycles 0-3. However, the $A[2]\times B[2]$ contribution to the final digit products $T[5:4]$ is not computed in Cycle 4. The fact that T_1 (= $T[5:3]$) is not required until the end makes it possible to defer computing the $T[5:4]$ digits for now. Thus in Cycle 4, Step 2 begins for computing Q_0. Observe that the two digit products $T[0]\times M'[0]$ and $T[0]\times M'[1]$ are computed and accumulated in exactly the same way as were $A[0]\times B[0]$ and $A[0]\times B[1]$ for T in Cycle 0. This means that the exact same accumulator state logic can be reused but with the new digit product inputs. The same result obtains for the Step 3 U computation that begins in Cycle 7. Looking at the schedule as a whole, it is evident that the same accumulator logic state is reused for Cycles 0, 4, and 7. Likewise, another accumulator logic state is reused for Cycles 1, 5, and 8; another for Cycles 2, 6, and 9; and yet another for Cycles 3 and 10. This constitutes only four accumulator states. Finally, Cycle 11 is a special case. The $Q[2]\times M[2]$ partial product is added to the accumulator as expected, but the $A[2]\times B[2]$ partial product is concurrently added not to the accumulator, but to the T1 save register. This does require some duplicated accumulator logic, but it is more than offset by the logic reuse in earlier cycles.

After all intermediate quantities have been computed, the final addition to compute the Montgomery product may proceed in the Final Sum stage. T_1, which has been saved in register T1, is added to U_1 in the high word of ACCUM. Also added is the bit from the T0_CARRY register. Recall that the Montgomery algorithm guarantees that the low word of the final product will compute to zero. If T_0 computed to zero, U_0 will also be zero. If T_0 computed to nonzero, U_0 will have the value that cancels it out as well as generating a carry into the high word. Therefore, there is no need to actually add T_0 and U_0—just T_1 and U_1 plus the potential carry bit from T_0. This results in an addition

path that is half as wide as would be required for a direct implementation of the Montgomery algorithm.

5.3 Implementation

For 256-bit operands, the number of digits was varied from two to eight, in steps of one. For any given architecture the digit size was kept uniform. Thus, the digit sizes were varied from 128 down to 32 bits, according to the number of digits chosen for a particular architecture. For example, choosing a two-digit architecture meant that the 256-bit operands were divided into two digits of 128 bits each. A four-digit architecture employed 64-bit digits. In the case of a digit count that was not an integral divisor of 256, the operand size was increased slightly so that all digits would be of equal size. In a three-digit architecture, 256 is not divisible by three, so in that case the datapath size was increased slightly to 258 bits, and the digit size was 86 bits ($3 \times 86 = 258$).

In addition, the number of digit multipliers was varied. In the simplest realization of the Rescheduled Montgomery Multiplier architecture, a single digit multiplier could be employed to compute all digit partial products. With the operands divided into only a few large digits, this would be sufficient and provide acceptable performance. However, the number of cycles would be proportional to the square of the number of digits. Therefore with a larger number of smaller digits, the performance would be too slow.

Consider a four-digit architecture. Recall from (5.4) that the total number of required digit multiplications is $N_M = 2.5k^2 + 0.5k$. For $k = 4$, this computes to 42. With only a single digit multiplier, 42 cycles plus the 4-cycle overhead would be required to compute the Montgomery product. Employing more than one digit multiplier permits computation of multiple digit products concurrently, reducing the number of cycles.

Assume two digit multipliers are employed. This reduces the number of cycles to 42/2 = 21.

Chapter 6 Methodology

6.1 Architectural Comparisons

Several alternative architectures were built, including those of other researchers, to evaluate the relative impact of the Rescheduled Montgomery Multiplier optimizations on area and performance. The proposed alternative architectures are listed below.

This research is focused on algorithmic optimization of the Montgomery steps, and not on designing the fastest multiplier possible. As such it allows the synthesis tool to build multiplier circuits that meet, as much as practicable, the applied design and optimization constraints.

The McIvor, *et al.* full-word Montgomery multiplier and ECC coprocessor architecture originally motivated this research because it dispensed with the interleaved reduction employed by the serial architectures. It also incorporated new algorithms to speed up multiplicative inversion, and therefore could conceivably be used to perform ECC point operations using affine coordinates.

The Eberle architecture [17] operates at the digit level like the Rescheduled Montgomery Multiplier, although without any parallelism. Digit sizes for the experiment range from 8 to 32 bits.

The Großschädl architecture [19] offers the potential to perform fast sequential bit-word computation using less complex logic than a multibit multiplier circuit.

The Tenca and Koç architecture [20], [21] uses a hybrid bit-digit approach. This permits multiple bit-word partial products to be computed concurrently. It can be implemented in a very small area if a minimum number of the relatively simple processing elements are instantiated.

The prior Montgomery multiplier architectures were originally designed and evaluated on a varied set of platforms and technology nodes. For example, the McIvor, *et al.* Elliptic Curve Coprocessor was targeted to an FPGA.

This research focuses on bottom-up IC design using standard cells and synthesis. It targets the Nangate 45 nm research process node [32]. In order to make fair comparisons, the preceding Montgomery architectures have been built and simulated in the same way as the proposed Rescheduled Montgomery Multiplier. In all cases, the RTL design adheres as closely as possible to the architectures described by the respective authors. The objective is to remain as faithful as possible to those architectures as described by the authors in their publications, in the absence of actual RTL or circuits.

6.2 RTL Design and Simulation

Initial algorithm analysis and development is performed in software using Python. Python is widely available and fast. Its inherent capacity for arbitrarily large operands suits it well to exploring and analyzing the $GF(p)$ and $GF(2^n)$ algorithms of interest here with operands on the order of hundreds of bits. In many cases Python functions are employed to generate test data for RTL simulations.

All architectures that were evaluated for area and performance were designed at the register transfer level (RTL) in the Verilog hardware definition language (HDL). The circuits were partitioned using a strict hierarchical design methodology. In particular, the objective was to evaluate overall design performance and complexity, not merely raw datapath performance. For the most part a structural design approach has been employed, especially in organizing and connecting well-defined submodules. In some cases, low level modules such as integer digit multipliers were defined with simple behavioral statements (*e.g.* of the form $P = A \times B$). In other cases, such as with $GF(2^n)$ multiplier

circuits, a structural approach was employed down to the gate level. At higher levels of abstraction, such as with finite state machines (FSMs), behavioral RTL was used.

All designs were simulated in a conventional Verilog testbench. The open source Icarus Verilog simulator has proven to be reliable and fast, and was employed here [33]. Result checking was automated, and an accompanying waveform viewer aided in debugging.

6.3 Synthesis and Static Timing Analysis

Synthesis translates an RTL design to a circuit consisting of standardized gates (referred to as standard cells) that implement Boolean logic functions, simple arithmetic, and memory functions in the form of sequential cells, or registers. The gates have been designed for a particular semiconductor process technology and characterized over a wide range of process variation, voltage, and temperature (PVT). The synthesis tool retrieves the gates from a library, and combined with various constraints, builds the circuit.

Synthesis constraints are varied and derive from multiple sources. For example, the standard cell library imposes constraints such as cell area, capacitance, transitions (slew), maximum fanout, and a range of output loads. For optimization purposes additional constrains are placed on the design, most fundamentally the target clock period for sequential logic. The synthesis tool translates the RTL to a circuit which consists of combinational and sequential logic gates to effect the function of the design.

Each combinational logic gate has a propagation delay. A source sequential gate (register) has a delay from the launching clock edge to when the Q data output pin changes (referred to as clock-to-Q delay). These delays are partly a function of the input transition and output load of the pins. Finally, a destination register has a setup time

requirement, usually positive, which is the amount of time before the capturing clock edge by which the input data must be stable in order to be captured correctly.

In most cases, timing paths are measured from a rising, launching clock edge at a *startpoint* register, to the data's arrival at the D pin of an *endpoint* register. If data arrives sufficiently early at the endpoint register's D pin before the next rising clock edge, it is captured correctly. The path does not have a timing violation. Fig. 17 illustrates such a case. The output function of the combinational logic gates has settled to its final value at the point denoted as *arrival time*. This is earlier than the required arrival time, a function of the setup time of the destination register. The difference between the two is the timing path's *positive slack*. For a first order approximation, slack is computed by (6.1).

$$t_{slack} = t_{CLK} - t_{CLK2Q} - t_{prop} - t_{SU} \tag{6.1}$$

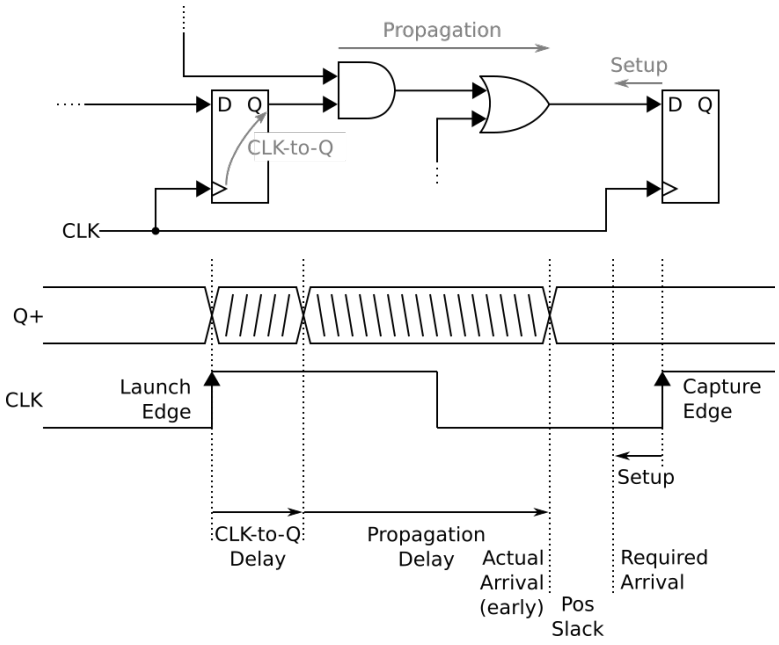


Fig. 17. Gate-level circuit timing with positive slack.

If the combinational path is large, such as having many levels of logic, the result data may resolve later than the required time. In such a case, an incorrect value will be captured by the endpoint register. Thus the circuit cannot operate as fast as the required clock frequency. Fig. 18 illustrates this condition. The combinational logic function resolves, and it arrives later the required time. The difference between arrival time and required time is termed *negative slack*. In this case, the circuit can still be run without timing-related errors by switching to a lower clock frequency. For example, given a clock period constraint of 2.0 ns, if the worst timing path has 0.2 ns of negative slack (–0.2 ns), it means that the data has arrived 0.2 ns too late to be correctly captured by the endpoint register. However, an increased clock period of 2.2 ns could be selected and the circuit operated without errors caused by timing violations.

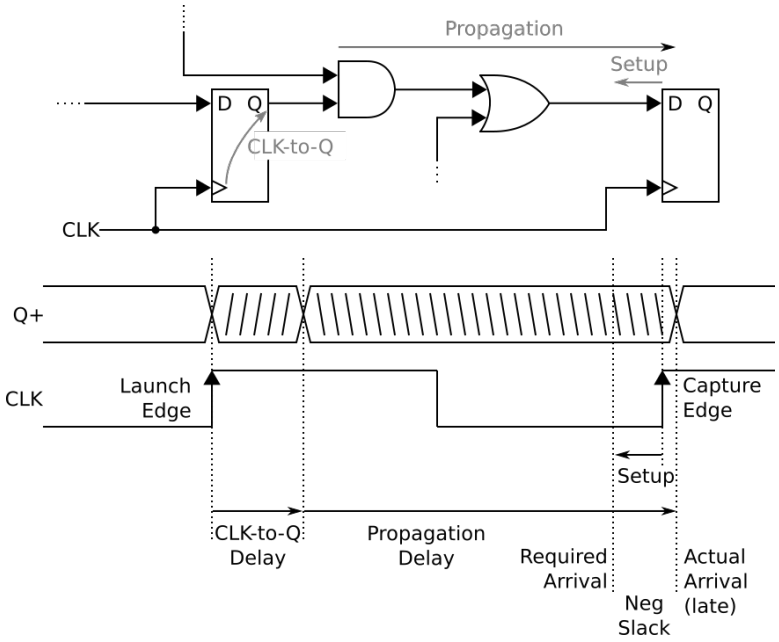


Fig. 18. Gate-level circuit timing with negative slack.

During synthesis, the tool can be driven, within reason, to certain optimization constraints. Operating conditions, design size, and the capabilities of the standard cell library all interact to determine how fast the design can function. Often the optimization constraints are more aggressive than what can realistically be achieved in a particular process corner. Synthesis tools operate on cost functions that determine a balance between building a design that meets the required performance specifications, and actually finishing the build. The optimization effort (duration) is thus limited to ensure the synthesis job does not run forever attempting to optimize a circuit beyond what is physically possible. Consequently, it is possible that upon completion of synthesis, the circuit may operate at a lower level of performance than what was requested. Although the synthesis tool has its own timing engine to handle the timing optimization, industry practice is to evaluate timing using a dedicated static timing analysis (STA) tool.

Static timing analysis (STA) of a synchronous digital circuit uses the standard cell libraries and operating conditions to compute the minimum and maximum time for signals to travel from all startpoints (input ports and registers) to their endpoints (output ports and registers). This determines the effective achievable minimum clock period for a design. A synthesized design may be timed with wireload models to model signal delays between cells. On a design that has been fully routed, STA can provide an even more accurate estimate of performance by taking into account routing delays, clock tree effects, and signal crosstalk.

All of the architectures implemented for this research were built and timed in the same integrated circuit technology. This allowed apples-to-apples comparisons of factors such as area and performance. The Nangate 45 nm standard cell library was chosen for the target library [32]. All designs were synthesized with the Synopsys *Design Compiler* synthesis tool [34]. Finally, static timing analysis (STA) was performed using Synopsys

PrimeTime [35]. During synthesis designs were driven with a clock period of 2.0 ns, *i.e.* a targeted clock frequency of 500 MHz. The clock period was also set to 2.0 ns for STA. Where a design had positive slack, it was subtracted from 2.0 ns to derive the possible clock period at which the circuit could run faster. Similarly where a design had timing violations, *i.e.* negative slack, the worst negative slack figure was added to the 2 ns clock period to compute a slower clock period at which the circuit could run without data errors.

6.4 Evaluation Criteria

In broad terms, a fundamental objective of the $GF(p)$ portion of this research is to exploit properties of the Montgomery algorithm to maximize efficiency. A large component of that is to optimize computation by avoiding unnecessary operations. This applies in computing the Q_0 quotient term as well as in performing the final $T + U$ reduction. Where opportunities arise, operations or portions of operations can be deferred, such as computing the high word T_1. This may permit reduction in the number of clock cycles as these operations can be rescheduled to run concurrently with others.

Given the foregoing, performance (latency) is of prime importance. Even as transistors become increasingly cheaper in succeeding generations of deep submicron process technologies, however, design complexity (area) should not be ignored. Both area and latency also contribute to power, or perhaps more important, energy consumption. To assist in evaluating design tradeoffs between performance and area (which is a cost) across many diverse architectures, many researchers have employed a figure of merit that takes into account both performance and area. This typically is expressed as an area-latency product. Simply, a design's latency in some unit of time is multiplied with its size in some unit of area. For purposes of this research, design area is

reported in square microns (μm^2), while latency is expressed in nanoseconds (ns). Thus the area-latency product would be expressed in units of $\mu m^2 \cdot ns$. In fact, the area results show fairly large numbers when expressed in μm^2 (on the order of 100,000 μm^2), so an adjustment is made by dividing the area-latency product by 10^6 to keep the figures of merit simple with one or two whole digits. The figure of merit computation is shown in (6.2) below, where A denotes area and L denotes latency.

$$\text{Area} \cdot \text{Latency FOM} = A \times L / 10^6 \tag{6.2}$$

From (6.2), the units of this figure of merit are $mm^2 \cdot ns$, but in practice the units are dropped.

Chapter 7 Elementary Montgomery Realizations

It can be useful to establish a baseline for what is possible using only basic resources such as a process design kit, a library of standard cells, and a synthesis tool. The steps in the Montgomery algorithm set out in Algorithm 2.1 are regular and uncomplicated, consisting of integer multiplication, addition/subtraction, and comparison.

7.1 Synthesized Parallel Multipliers

A single multiplier circuit may be instantiated and reused to perform Steps 1, 2, and 3 of the Montgomery algorithm. Modern logic synthesis tools are equipped with a wide variety of algorithms for design partitioning and logic optimization. These include algorithms for constructing highly efficient multipliers to meet process design rules as well as area and performance optimization targets. A parallel multiplier may be specified with simple RTL of the form $P = A \times B$. From this, the synthesis tool can build an appropriate architecture to meet the design and optimization constraints. In addition to the 256-bit operand size target of this research, it can be instructive to build multipliers of various sizes such as 128×128, 64×64, and even 32×32, and examine different ways of employing them.

For each multiplier size, two hypothetical deployments can be analyzed for area and performance. In the first deployment, a single instance of the multiplier is scheduled as many times as necessary to compute all the partial products, culminating in a final Montgomery product. In the second deployment, a baseline multiplier is treated as instantiated as many times as necessary to form a 256×256 multiplier. The first deployment may be termed *multiple-scheduling*. For example, given an operand size of 256 bits, a single 128×128 multiplier can be scheduled sequentially four times to compute

a single 512-bit product. Because a Montgomery product requires three intermediate products T, Q, and U, the latency would be multiplied by a factor of three. The second deployment may be termed *multiple-instantiation*. Four copies of a 128×128 multiplier could be instantiated along with addition logic to sum the partial products and produce the 512-bit result.

This approach is only intended to give a first order estimate of the area and performance results that may be possible with a simple synthesized multiplier. It would not be a comprehensive solution as it does not take into account other datapath logic such as addition/subtraction, multiplexing, and storage registers. Furthermore it ignores control logic resource requirements.

7.2 Pipelined Karatsuba-Ofman Multiplier

The Karatsuba-Ofman multiplication algorithm is a divide-and-conquer approach to integer multiplication. In simplest form, it splits up the input operands into two digits each and performs digit multiplication and accumulation. This permits the use of a smaller multiplication unit. Furthermore, it employs a mathematical identity to reduce the number of digit multiplications that must be performed from four to three [31], [36].

Let A and B be n-bit operands whose product $P = AB$ is to be computed. They are decomposed into two digits of $n/2$ bits each. Digits are indicated with subscripts. For example, $A = 2^{n/2}A_1 + A_0$. Digit concatenation is shown with parentheses: $A = (A_1, A_0)$ and $B = (B_1, B_0)$. Using the present notation, (2.3) is reproduced as (7.1).

$$P = 2^n A_1 B_1 + 2^{n/2}(A_1 B_0 + A_0 B_1) + A_0 B_0 \tag{7.1}$$

Let $Z_2 = A_1B_1$, $Z_0 = A_0B_0$, and $Z_1 = (A_1B_0 + A_0B_1)$. Substituting the Z terms into (7.1) gives (7.2).

$$P = 2^n Z_2 + 2^{n/2} Z_1 + Z_0 \tag{7.2}$$

Fig. 19 illustrates (7.2) graphically.

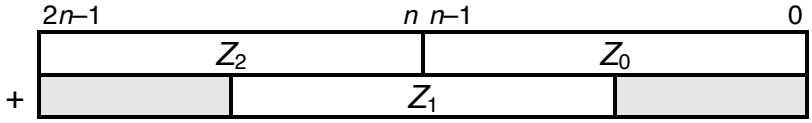


Fig. 19. Karatsuba-Ofman *Z* term summation.

The middle term of (7.1) and (7.2), $Z_1 = (A_1B_0 + A_0B_1)$, requires two (n/2)-bit digit multiplications. Z_1 can be replaced via an identity which requires only a single (n/2+1)-bit multiplication. The following steps show how the identity is derived. It begins by factoring $(A_1 + A_0)(B_1 + B_0)$ and culminates in (7.3).

$$(A_1 + A_0)(B_1 + B_0) = A_1B_1 + A_1B_0 + A_0B_1 + A_0B_0$$

$$(A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0 = A_1B_0 + A_0B_1$$

$$A_1B_0 + A_0B_1 = (A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0$$

$$Z_1 = (A_1 + A_0)(B_1 + B_0) - Z_2 - Z_0$$

$$Z_1 = (A_1 + A_0)(B_1 + B_0) - Z_2 - Z_0 \tag{7.3}$$

Fig. 20 depicts the final partial product summation. The Z_0 product A_0B_0 is in the least significant position, while the Z_2 product A_1B_1 is shifted up by n bits. From (7.2) the Z_1 term is shifted up by $n/2$ bits. Z_2 and Z_0 are subtracted, while the $(A_1 + A_0)(B_1 + B_0)$ product term is added as indicated in (7.3).

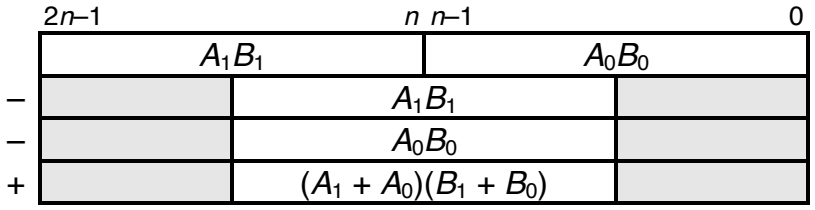


Fig. 20. Karatsuba-Ofman partial product summation.

Thus, performing an extra two additions and two subtractions makes it possible to compute only three digit-magnitude products instead of four: $(A_1 + A_0)(B_1 + B_0)$, A_1B_1, and A_0B_0. The Z_1 identity may be defined and computed in a few different ways, and it is added or subtracted depending on which one is chosen.

$$Z_1 = (A_1 + A_0)(B_1 + B_0) - Z_2 - Z_0 \quad \rightarrow \quad P = 2^n Z_2 + 2^{n/2} Z_1 + Z_0$$
$$Z_1 = (A_1 - A_0)(B_1 - B_0) - Z_2 - Z_0 \quad \rightarrow \quad P = 2^n Z_2 - 2^{n/2} Z_1 + Z_0$$
$$Z_1 = -(A_1 - A_0)(B_1 - B_0) + Z_2 + Z_0 \quad \rightarrow \quad P = 2^n Z_2 + 2^{n/2} Z_1 + Z_0$$

A hierarchical pipelined Karatsuba-Ofman 256×256 multiplier circuit was constructed in similar manner to the pipelined multiplier in the McIvor, *et al.* ECC Processor. At the top level, the three Z terms were further decomposed into smaller Karatsuba-Ofman type multipliers whose outputs were registered. The decomposition

was applied at three hierarchical levels in all. The lowest level used 32×32 conventional multipliers of RTL form $P = A \times B$.

As with the simple synthesized multipliers, the Karatsuba-Ofman realization is only intended as a point of reference for the three multiplication steps in the Montgomery algorithm. It does not include the other datapath circuitry, register storage, or control logic required for a full Montgomery deployment.

7.3 Full Direct Montgomery Multipliers

Algorithm 2.1 lists the Montgomery computations as a list of steps. It is tempting to think of these steps as being performed sequentially. With this in mind, the same integer multiplier can be reused to compute the three intermediate products in Steps 1-3. Scheduling the multiplier in this way maximizes its utilization.

An obvious alternative is to perform the sequence of operations in space as opposed to time, if resource constraints are ignored. In other words, using a brute force approach it is certainly possible to design a fully combinational Montgomery multiplication circuit. Operands A, B, M, and M' can be applied to a set of input registers, and the Montgomery product P can be captured one cycle later in an output register. Three multiplier units for intermediate products T, Q_0, and U can be built and connected such that the output of one cascades to the next. Because only the low word Q_0 of the quotient term is required, its multiplier unit will necessarily be less complex than the other two multipliers. Fig. 21 depicts the parallel architecture.

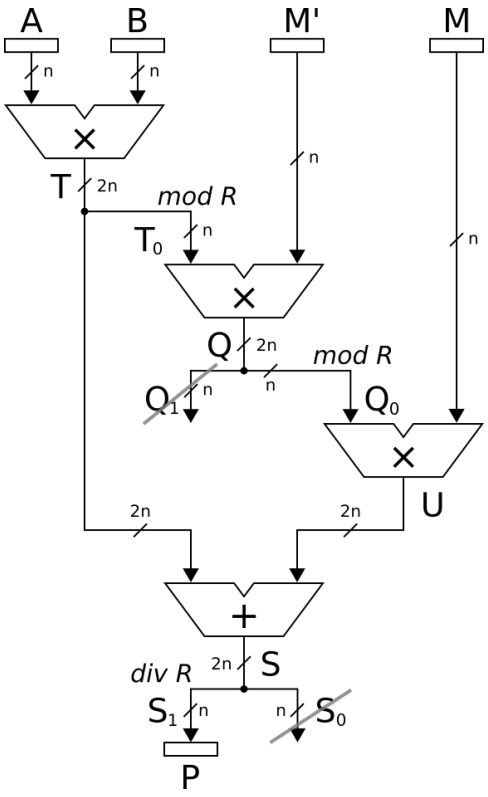


Fig. 21. Full direct parallel Montgomery architecture.

The next obvious step is to partition the combinational datapath into stages such that it is pipelined. Pipelining can increase latency, even with the shorter clock period that it typically enables. Conversely, the throughput gain of pipelining can easily be exploited for both RSA and ECC. An RSA application can use a sequence of repeated multiplications to compute a modular exponent. The ECC point operations necessarily employ a sequence of modular multiplications as well. Fig. 22 depicts the pipelined architecture.

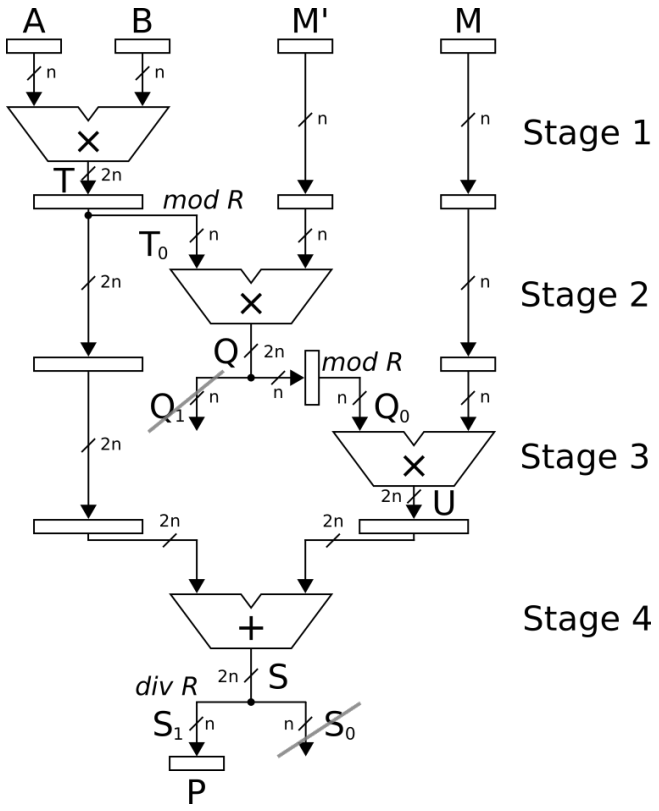


Fig. 22. Full direct pipelined Montgomery architecture.

Fully parallel and pipelined Montgomery circuits with n = 256 were built. The fully parallel circuit was a direct implementation of the Montgomery multiplication algorithm. Inputs and output were registered. Next, pipeline registers were added to this design to break up the large combinational arithmetic logic. A relatively simple pipeline consisting of four stages was built. Each of the three multiplication steps was segregated into a dedicated pipeline stage, and the final add, compare, and subtract operations were placed into a fourth stage.

Finally, a second pipelined version was built. This version had the remaining optimizations applied. First, a ones detector was applied to the T_0 output of the first multiplier in Stage 1. It took the form of a large OR gate structure to detect any 1s in T_0.

Only the single bit needed to be registered, saving register area for T_0, which could be discarded. The second optimization was to reduce the size of the adder circuit in Stage 4. Instead of computing the sum of double words T and U, only the upper word must be computed from T_1, U_1, and the single bit from the ones detector in Stage 1, in a classic add-with-carry approach. Fig. 23 depicts the pipelined architecture with the opportunistic Montgomery optimizations applied.

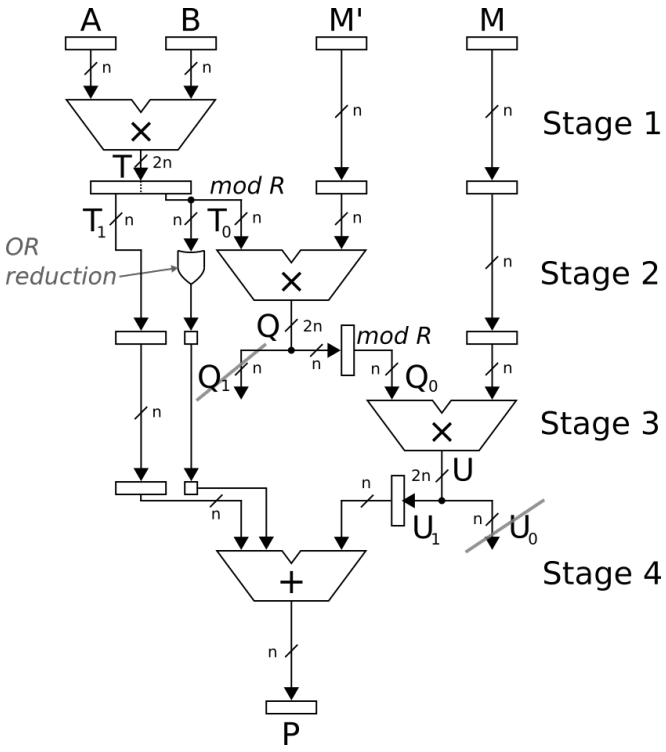


Fig. 23. Full direct optimized pipelined Montgomery architecture.

Chapter 8 Results

8.1 Synthesized Parallel Multipliers

Parallel multipliers for input operand sizes of 32, 64, 128, and 256 bits were built. Each multiplier had two n-bit input registers for the multiplier and multiplicand, and one $2n$-bit product output register.

Results for the *multiple-scheduling* deployment are as follows. The smallest simple multiplier is the 32×32. This multiplier has an area of just under 6.5k μm^2, and an effective period of 1.5 ns. To compute a complete Montgomery product, this multiplier would need to be scheduled for 164 cycles, plus an additional cycle for the final $P - M$ computation. Its total Montgomery latency is 249.2 ns. While this is large, the relatively moderate size gives this multiplier a small area-latency product of about 1.6.

Moving up to 64×64, its size is 3.4 times as large as the smaller one, at just under 22k μm^2. Its effective clock period is longer as well, at 1.849 ns. Conversely, because of the number of partial products is quadratically related to the number of digits, this multiplier only requires about 25% the number of cycles as the 32×32 version. Even with the slightly larger clock period, its latency is reduced by 68% to 79.5 ns.

As expected, the trend continues with the 128-bit and 256-bit variants. Each variant's area increases by a factor of between 3 and 4 of that of the next smaller one, and the latency decreases by about a factor of 3 each time.

Table 8 lists the results for the simple parallel multipliers deployed under multiple-scheduling. It should be kept in mind that the data are somewhat ideal and not completely realistic. They are based on the area of the multiplier itself and do not account for partial product summation logic. Likewise, the latencies only consist of the latency through the multiplier itself, plus one cycle at the end for the final add, test, and

possible subtract in the Montgomery algorithm. Additional delays for accumulation, which would affect the overall timing in a fully constructed circuit, and not included here.

Table 8. Synthesized parallel multiplier (multiple scheduling) area and latency.

n*×*n	**Total Area (μm^2)**	**Effective Period (ns)**	**Effective Frequency (MHz)**	**# Cycles**	**Total Latency (ns)**	**Area· Latency Product**
32×32	6,438	1.510	662	165	249.2	1.60
64×64	21,935	1.849	541	43	79.5	1.74
128×128	78,307	2.124	471	12	25.5	2.00
256×256	289,483	2.504	399	4	10.0	2.90

For the *multiple-instantiation* deployment estimates, area is traded for performance. In all cases the basic synthesized multiplier is considered to be tiled in such a way as to construct a full 256×256 parallel multiplier. Thus, the full size multiplier circuit consists of a single 256×256 multiplier itself, four 128×128 multipliers, or 16 64×64 multipliers, or 64 32×32 multipliers. Table 9 lists the results.

Table 9. Synthesized parallel multiplier (multiple instantiation) area and latency.

n*×*n	**# Instances**	**Total Area (μm^2)**	**Effective Period (ns)**	**Effective Frequency (MHz)**	**# Cycles**	**Total Latency (ns)**	**Area· Latency Product**
32×32	64	411,998	1.510	662	4	6.0	2.49
64×64	16	350,954	1.849	541	4	7.4	2.60
128×128	4	313,227	2.124	471	4	8.5	2.66
256×256	1	289,483	2.504	399	4	10.0	2.90

Based on this simplistic approach, the data might suggest that the best performance would be obtained from tiling 32×32 multipliers, because the presumed latency is only 6 ns. However, these data are not realistic and are useful only as an

approximate first order estimate of performance and area. It is certainly the case that the 32-bit multiplier by itself has the lowest latency, at 1.51 ns, but in the big picture this datum is incomplete. It ignores the fact that the 32×32 partial products must themselves be summed, and that the accumulation circuitry has not actually been built. Furthermore, all of the synthesized multipliers are designed such that the inputs and outputs are registered. These intermediate registers add to the overall area and would not be present in a purely combinational design. Either the data must be pipelined through each rank of 32×32 multipliers or the internal registers would have to be removed. In the latter case, of course, the combinational logic paths are increased in depth and thus in delay. As the size of the base multiplier increases, the area and performance estimates become more accurate. Finally, the algorithms available to synthesis tools are at present quite sophisticated. It is very difficult for an engineer to devise a more efficient combinational circuit than the tools themselves. Accordingly, the results in Table 9 can only be considered a very general starting point. For completeness, the 256×256 is repeated here. Its results are the most "realistic" because it lacks the data artifacts caused by the simple tiling assumption.

8.2 Pipelined Karatsuba-Ofman Multiplier

The results for the Karatsuba-Ofman multiplier are listed in Table 10.

Table 10. Pipelined Karatsuba-Ofman multiplier area and latency.

n	**Total Area (μm^2)**	**Effective Period (ns)**	**Effective Frequency (MHz)**	**# Cycles**	**Total Latency (ns)**	**Area· Latency Product**
256	251,949	1.620	617	22	35.6	8.98

Area is substantial, nearly a fourth of a square millimeter. Effective clock period is on the order of 1.6 ns, and the single product latency is seven cycles. As employed to perform the Montgomery algorithm, a total of 22 cycles would be required for a latency of 35.6 ns.

These data are only for the Karatsuba-Ofman multiplier circuit itself, and do not reflect a complete Montgomery design. Omitted are the actual circuitry to store the intermediate products T, Q, and U, as well as the final add, compare, and potential subtract.

8.3 Full Direct Montgomery Multipliers

All three full directly implemented Montgomery circuits have similar area cost and performance. Total areas are in the vicinity of 700k μm^2, the largest being the pipelined design at 711,744 μm^2. The pipeline registers account for this growth. The area for the optimized pipelined design falls to 698,660 μm^2. Shrinking the final adder circuit from 512 to 256 bits, as well as eliminating the 256-bit T_0 pipeline registers accounts for this saving. Table 11 lists the results.

Table 11. Direct parallel and pipelined Montgomery area and latency.

Design	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product	Area· Throughput Product
Parallel	698,628	6.895	145	1	6.9	4.82	4.82
Pipelined	711,744	2.496	401	4	10.0	7.11	1.78
Optimized Pipelined	698,660	2.512	398	4	10.0	7.02	1.76

While performance is similar, there is a significant difference between that of the parallel design and that of the pipelined designs. In the parallel design, a great deal of

arithmetic combinational logic must be built between the input and output registers for a one-cycle computation. The achievable clock cycle time is about 6.9 ns. While this is slower than the targeted clock period of 2 ns, it is the entire latency of the design. Its area-latency product is only 4.82. The pipelined designs, on the other hand, can be run with a faster clock of approximately 2.5 ns. With four stages their latency is 10 ns, and their area-latency products are worse than that of the parallel design, at just over 7. Conversely, the pipelining brings with it the usual advantage of higher throughput if a series of Montgomery products are to be computed. In this case, a Montgomery product can be output every 2.5 ns. An area-throughput figure of merit can be computed as the area-throughput product. For both pipelined designs it is just under 1.8.

8.4 McIvor ECC Processor with Pipelined Multiplier

The McIvor, *et al.* ECC Processor provides the original motivation for this research [11]. Its integration of related functions for ECC computations, such as the optimized inversion algorithm, make it interesting as a comprehensive solution in the application space. Table 12 lists the area and performance results for the ECC Processor.

Table 12. McIvor, *et al.* ECC Processor area and latency.

Design	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
ECC Processor	640,106	1.496	668	35	52.4	33.52

As the table shows, the entire ECC Processor is large, well over half a square millimeter at 640k μm^2. The multiplier itself occupies 584k μm^2, or 91% of the total. Of

the multiplier's area, 26% consists of noncombinational logic, mainly the pipeline registers.

In this design, the multiplier simply computes the 512-bit product of its inputs. It is scheduled to perform the sequential steps of computing T, Q, and U of the basic Montgomery algorithm as listed in Algorithm 2.1. The multiplier computes the 512-bit product in nine clock cycles. Despite the pipelining, the operations for a single Montgomery product cannot be overlapped because each integer product depends on the prior one. Additional cycles are required for overhead (such as loading input registers) and for computing the final product P from the original product T and the reduction word U. The total number of cycles to perform a Montgomery multiplication is 35. At the typical process, voltage, and temperature (PVT) corner, PrimeTime STA results showed that the design could run with a minimum clock period of 1.496 ns. Total latency for a Montgomery product then is 35 × 1.496 ns, or 52.4 ns.

There are some obvious ways to improve the performance. First, some stages of the multiplier pipeline could be merged. It is possible that a different tradeoff between clock period and cycle count would result in faster computation. In this instance the critical paths are in the final 512-bit carry propagate addition at the end. For some of the phases this could be reduced. For example, in computing the quotient word Q in multiplication Step 2, only the less significant word Q_0 is used while the more significant Q_1 is discarded. Resource scheduling in the datapath could be changed so that in this case only the lower 256-bit sum need be computed. Thus it should be possible to reduce the cycle time of the design further.

8.5 Eberle Serial Digit-Digit Architecture

The digit-digit Montgomery multiplier proposed by H. Eberle, *et al.* [17] was implemented in three separate configurations. Each configuration was determined by the digit size, *d*, from the set {8, 16, 32}. The design consists of three major units: a controller, a multiplier-accumulator (MAC) unit, and a memory for operand storage. The memory is implemented in standard cell registers but could also be implemented as a register file.

In contrast to the architecture presented in [17], the controller designed for this research is not a fully programmable coprocessor. The objective is to analyze Montgomery multiplication performance specifically, and a fixed hardware control is sufficient and entails less implementation risk. The controller is comprised of a hierarchical set of two finite state machines (FSMs). The primary FSM controls the outer loop of the algorithm, denoted by index *i*, and the secondary FSM controls the inner loop, denoted by index *j*. Operands are fetched from memory into registers, and digit computations are performed in accumulator register ACC, which is two digits wide. The two halves of the accumulator are denoted by ACC[1] and ACC[0]. Partial product digits are written back to memory. Digit operations are pipelined.

With the digit sizes from the set {8, 16, 32} used in this realization, the MAC datapath is small relative to the 256-bit operand size. As a result, total areas are all under 20k μm^2. In all cases the noncombinational area is a substantial portion of the total area, varying between 9k and 11k μm^2. It mainly consists of standard cell registers that are used for the memory. Table 13 lists the area and latency results.

Table 13. Eberle, *et al.* multiplier area and latency.

Digit Size d	# Digits k	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
8	32	12,462	0.967	1,034	2,817	2,724.0	33.95
16	16	14,125	1.296	772	897	1,162.5	16.42
32	8	19,605	1.599	625	321	513.3	10.06

As might be expected, increasing digit size has two opposing effects on performance. First, at larger digit sizes, the MAC circuitry grows more complex. This results in more and larger combinational timing paths. Consequently, the minimum achievable clock period grows. The 32-bit version runs at a lower frequency than the 8-bit version. Conversely, increasing digit size dramatically reduces the number of cycles required. From (2.2), the number of multiplications is a quadratic function of the number of digits. For a fixed operand size, doubling the digit size cuts the number of digits in half. Therefore, to a first order approximation, a 75% cycle count reduction might be expected. In practice, the reduction is approximately 67% for each doubling of digit size. Some handshaking between the primary and second FSMs and the fixed cycle count to compute the quotient digit Q_i do not scale down. With small digits ($d = 8$), 2,817 cycles are required. For $d = 16$, the number of cycles drops to 897, and for $d = 32$ only 321 cycles are required.

Despite the area minimization afforded by a digit-centric approach, the high latencies make this architecture unattractive. The area-latency products range from 10 ($d = 32$) to almost 34 ($d = 8$).

It would be possible to make microarchitectural changes to reduce the number of cycles. One way would be to merge the partial product and reduction phases, converting it from a CIOS to FIOS architecture. The product phase could be interrupted after inner loop iteration $j = 0$ to compute Q_i. At this point, iterations could resume for $1 \le j \le k - 1$,

and the partial product $A[i]\times B[i+j]$ and reduction $Q_i\times M[j]$ computations could be performed concurrently and summed. The cost would be an additional multiplier and more complex addition circuitry, but the separate reduction phase would be eliminated.

Closer examination of the Montgomery reduction step suggests another possible approach, within some bounds. Once the reduction term U_i is computed, adding it to the initial product effectively cancels out the least significant word, digit, or bit to zero. Applying Algorithm 2.1 to the digit case, U_i depends only on $T_i[0]$, M', and M. Therefore, a particular bit pattern of $T_i[0]$ will always generate the same reduction word U_i.

If $T_i[0]$ is relatively small, a lookup table (LUT) of precomputed reduction terms U_i could feasibly be constructed. Instead of computing Q_i in phase 2 followed by in phase 3, U_i digits could be fetched from the table while phase 1 is iterating. As a result, the number of cycles could be reduced by more than 50%, without adding a second digit multiplier, but at the cost of storage for the reduction terms.

In the present digit-digit architecture, assume digit size $d = 8$. Using the eight bits of partial product digit $T_i[0]$ as an address translates to a LUT size of $2^8 = 256$ entries. For $n = 256$, this would require a 65,536-bit memory.

The reduction terms are partly a function of the radix $R = 2^n$ and the modulus M. In a completely programmable circuit in which the user can choose these parameters, the reduction terms must be precomputed for the chosen R and M. For maximum flexibility, then, the LUT would reside in some type of writable storage, such as registers or RAM. The LUT can be programmed with the precomputed terms prior to deployment.

For larger sizes of d, such as 16 and 32, an address width equal to d becomes infeasible. Consider the $d = 16$ case. The LUT entries still must be 256 bits wide, but now the number of entries has increased from 256 to 65,536. This translates to a LUT

size of 16 megabits (Mb). Instead, an approach could be to reuse the LUT but for each 8-bit subset of the least significant digit. The upper eight bits cannot be used directly, because the reduction for the lower eight bits will change the upper eight bits. This means that the reduction must first start on the lower eight bits to reduce the remainder of the digit. Now, the upper eight bits can be used as an address into the LUT to fetch the next reduction term. Thus throughout the reduction operations there are two overlapping reductions. In this way the partial product can be reduced, with the least significant 16-bit digit zeroed out. A similar technique may be applied to the $d = 32$ case, only in this case there must be four 8-bit reduction term fetches with each digit. In an ideal case a register file with four read ports could be employed. If this is impractical, four identical register files could be tiled. and independently read.

In broad terms, such a variant of the Eberle, *et al.* digit-digit Montgomery multiplier was designed to use a lookup table as described above. There were actually six implementations considered, according to two parameters. The first parameter determined the design of the LUT storage, which could be standard cell registers, or a compiled RAM. The second parameter was the digit size d, whose values were chosen from the set {8, 16, 32}.

Table 14 lists the results for the modified Eberle, *et al.* architecture with standard cell register LUTs for the cases $d = 8$, 16, and 32. In all cases the register LUT increases the area massively by factors between 30 and 40 of that of the canonical design. This result is not unexpected, because standard cell registers are large. In the Nangate 45 nm library used here, the SDFF_X1 register is over 6 μm^2 in area, and the LUT alone requires 65,536 of them.

Table 14. Eberle, *et al.* multiplier with register LUT area and latency.

Digit Size *d*	# Digits *k*	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
8	32	422,329	1.127	887	1,409	1,587.9	670.6
16	16	486,569	1.468	681	513	753.1	366.4
32	8	615,327	1.784	561	257	458.5	282.1

In the "best" area case, for $d = 8$, the LUT design latency is 1,588 ns, compared to 2,724 ns for the canonical design. The register LUT speedup over canonical design is merely 1.7. For larger digit sizes, performance gains become even less impressive. For $d = 16$, the LUT design has a latency of 753 ns, versus 1,153 ns for the canonical design. This is only about a 35% reduction. For $d = 32$ there is virtually no performance benefit at all, only improving from 513 to 459 ns. Clearly this is not a compelling tradeoff.

Dedicated memory circuits are much more space efficient for storage than a bank of standard cell registers. The bit density of even a 6-transistor static random access memory (SRAM) is higher than equivalent storage using registers. OpenRAM is an open source RAM compiler tool. It can be employed to generate compiled RAMs of almost any configuration [37]. The open source FreePDK 45 nm physical design kit contains a library of bit cells that can be used by OpenRAM [38].

OpenRAM was used to generate RAMs with different configurations of word size and word count, and the area results compared. Using a compiled RAM in lieu of registers for the LUT did result in lower area. Table 15 lists the results.

Table 15. Eberle, *et al.* multiplier with RAM LUT area and latency.

Digit Size d	# Digits k	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
8	32	100,884	1.127	887	1,409	1,587.9	160.2
16	16	103,413	1.468	681	513	753.1	77.9
32	8	109,929	1.784	561	257	458.5	50.4

The only practical benefit of the RAM over the registers is in smaller area. Latencies are the same as with the register LUT. Critical paths are in the MAC circuit itself, not the memory access, so there is no overall performance difference between the register LUT and RAM LUT. A 64k-bit RAM is still massive, however. Even with the smallest available configuration for a RAM LUT, the total circuit areas are between five and eight times as large as those in the canonical Eberle, *et al.* design. The minor performance benefit offered by a LUT is negated by the large area growth. In any case, the high latency and massive area result in massive double or triple digit area-latency products.

8.6 Großschädl Serial Bit-Word Architecture

This section describes the implementation results for the bit-word multiplier architecture proposed in [19]. As in the other architecture implementations, the digit size d is varied. In this case, however, the digit size pertains only to the size of the carry propagate adder used at the end for merging the working carry save words. The multiplier datapath itself remains $1 \times n$ in all cases.

There is a large fanout of 256 from the a_i bit of the A register to the AND gates computing a_iB. The first CSA's least significant output bit is used as the quotient bit q_i to compute q_iM. It too has a fanout of 256 to the AND gates computing q_iM. It could help

to add a pipeline stage with duplicated/multiple a_i registers between A and the AND gates to reduce the fanout-caused delays.

Because only the size of the carry propagate adder varies, all three variants of the Großschädl architecture have similar area, on the order of 20k μm^2. Performance is similar as well. In all cases, the product computation requires 260 cycles, consisting of 256 cycles for the bit-word multiply-accumulate plus four cycles of overhead. Performing the final carry propagate addition is what varies with the digit size. For example, for a digit size of $d = 8$, $k = 32$ digits need to be summed and therefore require 32 cycles to compute, for a total cycle count of 292. For $d = 32$, only eight additional cycles are required to compute the final nonredundant product. Table 16 lists the results.

Table 16. Großschädl, *et al.* multiplier area and latency.

Digit Size d	# Digits k	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
8	32	19,934	0.810	1,235	292	236.5	4.72
16	16	20,093	0.825	1,212	276	227.7	4.58
32	8	20,760	0.820	1,220	268	219.8	4.56

The effective clock period is roughly identical for all configurations. The smaller number of digit additions for $k = 8$ provides the highest performance, with a total latency of 219.8 ns for one Montgomery multiplication. The area-latency product figure of merit for all three variants is less than 5.

8.7 Tenca and Koç Serial Hybrid Bit-Digit Architecture

Multiple versions of the Tenca and Koç architecture [20], [21] were designed, simulated, and built. Digit size d was chosen from the set {8, 16, 32}. For each digit

size, the number of processing elements (m) was varied from 1 to 20. Fig. 24 plots the latency versus PE count (m = 1 to 20) for each digit size d.

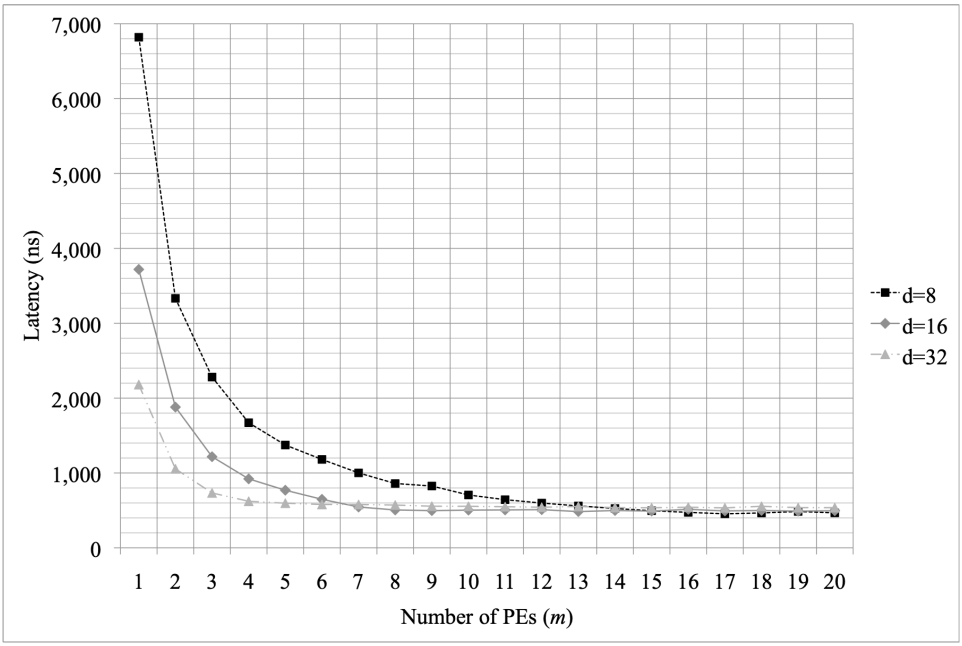


Fig. 24. Tenca and Koç multiplier latency versus number of PEs (m).

Increasing the number of PEs increases parallelism, and results in improved performance up to a point, after which continuing to add PEs yields no additional improvement. For example, for d = 8 (k = 32) and m = 1, latency is approximately 6,820 ns. Doubling m to 2 reduces the latency in half to 3,333 ns. Latency incrementally improves until there are about 17 PEs, and then levels out. A similar trend is evident for d = 16 and d = 32. The d = 16 design's latency stops improving after m = 8 PEs, and the d = 32 design reaches its minimum even sooner.

For each digit size d, the minimum hypothetical latency is achieved when the number of PEs $m = k/2$, where k indicates the number of digits and $k = n/d$. The reason is

that at that point, just enough PEs are available to process all partial products in sequence, and no partial products are stalled waiting for a PE to complete its previous partial product. In the actual implementations, adding another PE or two beyond $k/2$ nevertheless yields some slight improvement because PE-PE handoff is slightly faster than the handoff from PE_{m-1} via circular buffer to PE_0. The circular buffer design introduces a few cycles of delay.

A bit-digit multiplier with $d = 32$ effectively computes a result twice the size of one with $d = 16$, and four times the size of one with $d = 8$. Accordingly, a configuration with $d = 32$ and $m = 1$ is roughly equivalent to one with $d = 16$ and $m = 2$, or $d = 8$ and $m = 4$. For $d = 32$ and $m = 1$, area is 12.4k μm^2 and latency is 2,180 ns. For $d = 16$ and $m = 2$, area is 12.4k μm^2 and latency is 1,880 ns. Finally, for $d = 8$ and $m = 4$, area is 12.7k μm^2 and latency is 1,671 ns. The trend appears to favor smaller digit sizes.

For $d = 8$, the minimum latency approaches 460 ns, corresponding to 570 cycles, for 18 or more PEs. For $d = 16$, the minimum approaches just under 500 ns, about 550 cycles, for eight and more PEs. Finally, for $d = 32$, the minimum approaches 540 ns, approximately 540 cycles, for nine or more PEs. In general the smaller digit size results in a lower latency because, despite a higher cycle count, the clock period can be shorter.

Fig. 25 plots latency versus area for most of the configurations. Some $d = 16$ and $d = 32$ variants with a larger number of PEs (and thus higher area) are omitted to prevent the x-axis from growing too large to be legible on the page.

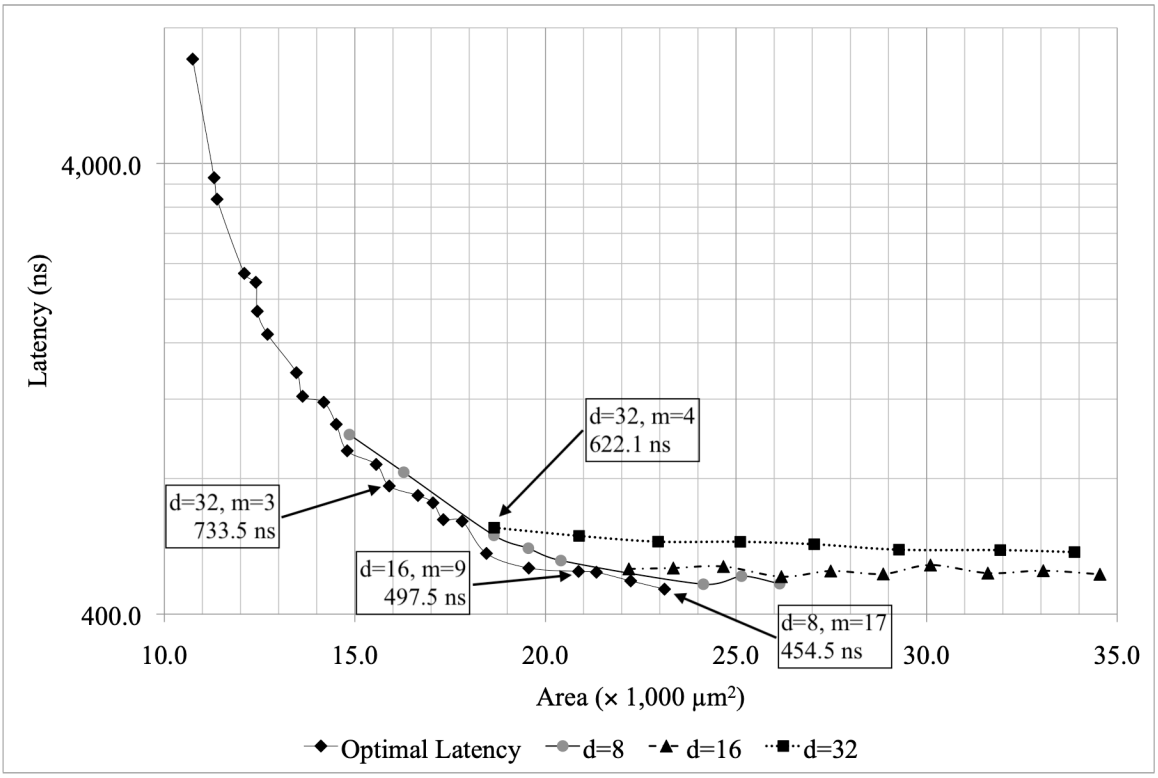


Fig. 25. Tenca and Koç architecture latency versus area.

The plot labeled Optimal Latency is the Pareto frontier of the latency versus area curve, and the lowest-latency for each d on the frontier is called out. For $d = 32$, the best Pareto latency is a relatively slow 733.5 ns, with $m = 3$. The next $d = 32$ configuration ($m = 4$) is also indicated, with a 622 ns latency, although it is not on the frontier. The ($d = 16$, $m = 9$) configuration's latency is just under 500 ns, and the ($d = 8$, $m = 17$) configuration has a latency of 454.5 ns.

Table 17 list the results for three Pareto frontier implementations of the Tenca and Koç Montgomery architecture. For each digit size d, the fastest configuration on the Pareto frontier is listed.

Table 17. Tenca and Koç multiplier area and latency.

Digit Size *d*	# Digits *k*	# PEs *m*	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
8	32	17	23,124	0.789	1,267	576	454.5	10.51
16	16	9	20,869	0.849	1,178	586	497.5	10.38
32	8	3	16,655	0.944	1,059	777	733.5	12.22

The first configuration uses d = 8-bit digits. This particular instance of the design employs m = 17 PEs in cascade and occupies 23k μm^2 of area. Each PE has a single 1×8 bit-digit multiplier which requires 32 cycles to compute a 1×256 partial product. Thus this instance employs 17 multipliers concurrently. Its latency of 454.5 ns is the smallest of any of the Tenca and Koç designs that were implemented. The next configuration uses d = 16-bit digits. Nine PEs are cascaded in this configuration, so there are nine 1×16 multipliers. Its total area is 20.9k μm^2, and it can produce a 256-bit Montgomery product in 497.5 ns. This is not the fastest 16-bit architecture, but it is the fastest one that lies on the Pareto frontier. Finally, the third configuration uses digit size d = 32. It uses only three PEs in cascade, so there are three 1×32 multipliers computing partial products concurrently. It is the fastest 32-bit design on the Pareto frontier, with a latency of 733.5 ns. There are faster 32-bit designs. For example, the k = 4 variant has a latency of 622 ns, an area of 18.7k μm^2, and an even better area-latency product of 11.60. But as the plot in Fig. 25 shows, 32-bit designs larger than this are all undercut by faster d = 16 and d = 8 designs for equivalent area.

8.8 Rescheduled Montgomery Multiplier

The Rescheduled Montgomery Multiplier (RMM) architecture was built in 31 different configurations. 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. One or more RMMs were designed for

each value of k from 2 to 8. The number of digit multipliers, m, was also varied for each k configuration. Table 18 is a matrix indicating which combinations of k and m were built. As used in this dissertation, a Rescheduled Montgomery Multiplier configuration is designated as RMM (k, m).

Table 18. Rescheduled Montgomery Multiplier (k, m) combinations.

		# Digit Multipliers *m*													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Digits *k*	2	•	•												
	3	•	•	•											
	4		•	•	•	•									
	5				•	•	•								
	6					•	•	•	•	•	•				
	7						•	•	•	•	•	•			
	8								•	•	•	•	•	•	•

From (5.4), computing the intermediate products T, Q_0, and U requires a total of $N_M = 2.5k^2 + 0.5k$ digit multiplications. With only a single digit multiplier, that many cycles is required to compute the digit products. If ($m > 1$) digit multipliers are employed, digit products can be scheduled concurrently. In such a case, the number of cycles for product computation is approximately $\lceil (2.5k^2 + 0.5k)/m \rceil$. The three-stage pipeline adds two cycles of latency. One cycle is then required to compute the $P = T_1 + U_1$ sum. One final cycle is used to test for $P \geq M$ and conditionally compute $P = P - M$. These four cycles are considered overhead and are invariant regardless of the specific RMM (k, m) microarchitecture chosen.

8.8.1 RMM (2, *m*)

The first set of RMMs were designed with $k = 2$. For this configuration, a total of $N_M = 11$ digit multiplications must be performed to compute T, Q_0, and U. Two RMMs

were built, one with a single (m = 1) digit multiplier, and one with m = 2 digit multipliers, denoted as RMM (2, 1) and RMM (2, 2) respectively. These variants provided straightforward, first-order insights into the area-latency tradeoff. Table 19 lists the area and performance results for k = 2.

Table 19. RMM (2, m) area and latency.

d	(k, m)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
128	(2, 1)	105,697	2.217	451	15	33.3	3.52
128	(2, 2)	188,374	2.217	451	10	22.2	4.18

With only a single digit multiplier, RMM (2, 1) requires 11 cycles to compute the N_M = 11 digit products, plus the four-cycle overhead, for a total of 15 cycles. The synthesized area of this design is 105,697 μm^2. A substantial portion consists of the 128×128 digit multiplier, at over 80k μm^2. As synthesized, this design can run at an effective clock period of 2.217 ns, or 451 MHz. For the 15 cycles required, the total latency is 33.3 ns.

Adding a second digit multiplier for a (2, 2) configuration permits concurrent computation of two digit products during a cycle. This configuration reduces the digit product cycles from eleven to six, but with the four-cycle overhead, overall cycle latency drops by only one third from 15 to 10 cycles. These digit multipliers are large, so adding the second one increases area by slightly more than 80k μm^2 to 188,374 μm^2. Because the critical path is in the digit multipliers, and not in other areas such as the accumulator datapath, the effective clock period is not degraded further, and remains at 2.217 ns. Therefore, the total latency is 22.2 ns. This is a 33% reduction in latency relative to RMM (2, 1), but the area cost of 83k μm^2 constitutes a 78% increase over RMM (2, 1).

8.8.2 RMM (3, *m*)

RMMs with operands split into k = 3 digits were built next. For this configuration, a total of N_M = 24 digit multiplications must be performed. Three configurations were built, with m = 1, 2, and 3 digit multipliers: RMM (3, 1), (3, 2), and (3, 3). The word size n was increased slightly to 258 bits to permit a uniform digit size d = 86 bits. Table 20 lists the area and performance results for k = 3.

Table 20. RMM (3, *m*) area and latency.

d	(*k*, *m*)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
86	(3, 1)	66,429	1.969	508	28	55.1	3.66
86	(3, 2)	114,610	2.013	497	16	32.2	3.69
86	(3, 3)	146,060	2.002	500	12	24.0	3.51

Smaller digit multipliers result in area savings. The entire area of RMM (3, 1) is only 66,429 μm^2. Increasing the number of digit multipliers naturally increases the area, sometimes dramatically. RMM (3, 2) is almost twice the size of RMM (3, 1), at over 114k μm^2. This is caused by the more complex accumulator logic required to add two 172-bit digit products with the accumulator. The overall increase for instantiating a third multiplier is less dramatic, at just under 32k μm^2 to 146k μm^2.

All three RMMs have improved cycle time compared to the k = 2 case. This is a direct result of reducing the digit multiplier width from 128×128 to 86×86. For the single-digit multiplier design RMM (3, 1), the effective clock period of 1.969 ns and 28 cycles gives a latency of 55.1 ns. The two- and three-multiplier designs RMM (3, 2) and (3, 3) have slightly larger clock periods at just over 2 ns, which is caused by additional propagation delay through the more complex accumulator circuit.

The RMM (3, 2) configuration has a similar latency (32.2 ns) to that of RMM (2, 1) (33.3 ns), but is larger by approximately 9k μm^2. The next configuration, RMM (3, 3) has a latency of 24 ns, only slightly more than RMM (2, 2) at 22.2 ns, but its area is 42k μm^2 *smaller*. RMM (3, 3) has an area-latency product of 3.5, the best of the $k = 3$ designs. If a (3, 4) configuration were built, it would require only 10 cycles for an estimated latency of 20 ns.

8.8.3 RMM (4, *m*)

RMMs with operands split into $k = 4$ digits were built. For this configuration, a total of $N_M = 42$ digit multiplications must be performed. Four variants were built, with $m = 2$, 3, 4, and 5 digit multipliers: RMM(4, 2), RMM (4, 3), RMM (4, 4), and RMM (4, 5). All could be optimized to run at a clock period of less than 2 ns. As before, the digit multipliers determined the critical timing path. Table 21 lists the area and performance results for $k = 4$.

Table 21. RMM (4, *m*) area and latency.

d	(*k*, *m*)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
64	(4, 2)	76,933	1.910	524	25	47.8	3.67
64	(4, 3)	101,974	1.881	532	19	35.7	3.64
64	(4, 4)	120,631	1.881	532	15	28.2	3.40
64	(4, 5)	144,877	1.904	525	13	24.8	3.59

With two digit multipliers, RMM (4, 2) requires $N_M/2 = 21$ cycles, plus the four overhead cycles, for a total of 25 cycles to compute a full Montgomery product. With an effective clock period of 1.91 ns, this translates to a latency of 47.8 ns. The implementation is relatively compact at just under 77k μm^2. As expected, increasing the

number of digit multipliers reduces the number of cycles and overall latency, to just under 25 ns with $m = 5$. However, the area-latency product minimum is achieved in RMM (4, 4), with a value of 3.40. It can perform a full Montgomery multiplication in 28.2 ns in an area of 120,631 μm^2.

It can be instructive to compare the RMM (4, m) configurations to the RMM (2, m) configurations, because the digit size of the former is exactly half that of the latter. It is possible to examine two different designs that nevertheless have the same number of multiplication bits "in flight" during any cycle. For example, RMM (2, 1) has a single digit multiplier, so during each cycle one 128×128 product is being computed, for 16,384 partial bit products in flight. It has an area of 106k μm^2 and a latency of 33.3 ns. RMM (4, 4) computes four 64×64 digit products concurrently, for 4×64×64 = 16,384 partial bit products in flight. Its area is 121k μm^2 but it has a latency of only 28.2 ns.

For a full product, halving the digit size results in squaring the number of digit multiplications that must be performed. A single 256×256 product computed with 128-bit digits requires four digit multiplications. If those digits are reduced to $d = 64$ bits, $k = 4$ and the number is increased to 4^2 digit multiplications. With the RMM, however, the Q_0 computation makes it possible to avoid some of those digit multiplications. In the $d = 128$ case, (5.1) provides that Q_0 requires only three digit multiplications—one digit multiplication is saved compared to computing the full $Q = (Q_1, Q_0)$, a 25% reduction. For $d = 64$, Q_0 requires only 10 instead of 16 digit multiplications, a 37.5% reduction relative to a full Q product.

The scheduling algorithm ensures that the product bits being computed and accumulated are more "vertical" in the (4, 4) configuration than in the (2, 1) configuration. In the (2, 1) configuration more of the "vertical" partial bit product accumulation occurs in the digit multiplier (with higher complexity). In the (4, 4)

configuration, a portion of that vertical accumulation is moved out of the digit multipliers and into the accumulator datapath.

RMM (2, 1) requires 106k μm^2 of die area and has a latency of 33.3 ns. RMM (4, 4) is larger, at 121k μm^2, but only requires 28.2 ns. Switching from RMM (2, 1) to RMM (4, 4) costs a 14% increase in area, but purchases a speedup of 1.18. The overall area-latency product declines from 3.51 to 3.40. Although the area-latency product improves, it can be argued that RMM (4, 4)'s resource utilization is not as efficient as that of RMM (2, 1). Configuring $m = 4$ digit multipliers does not evenly divide the $N_M = 42$ (or $N_Q = 10$) digit multiplications required. That means that during one cycle, two of the digit multipliers are not used. It is partially compensated for by the algorithmically more efficient Q_0 computation enabled by employing smaller digits.

8.8.4 RMM (5, *m*)

RMMs with operands split into $k = 5$ digits were built. For this configuration, a total of $N_M = 65$ digit multiplications must be performed. m was varied between 4 and 6: RMM (5, 4), RMM (5, 5), and RMM (5, 6). As with the $k = 3$ configurations, $n = 256$ is not divisible by k. Therefore for $k = 5$, n is set to 260 to allow uniform digit sizes of $d = 52$ bits. Table 22 lists the area and performance results for $k = 5$.

Table 22. RMM (5, *m*) area and latency.

d	(*k*, *m*)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
52	(5, 4)	116,492	1.770	565	21	37.2	4.33
52	(5, 5)	118,482	1.794	557	17	30.5	3.61
52	(5, 6)	131,231	1.816	551	16	29.1	3.81

RMM (5, 5) is only 2k μm^2 (1.7%) larger than RMM (5, 4) but is nearly 7 ns faster, for a speedup of 1.22. The initial RMM (5, 6) implementation grew quite large, approximately 45k μm^2 (38%) larger than RMM (5, 5), for only a two cycle (4 ns) decrease in latency and a speedup of 1.15. That implementation did not employ uniform accumulator scheduling through the three products T, Q_0, and U. As a result, the accumulator datapath did not have much reuse between phases and grew large. The RMM (5, 6) design was reworked to use uniform accumulator scheduling. Area shrank from 163k μm^2 to 131k μm^2, at the cost of an additional clock cycle. Total latency was 29.1 ns. Compared to RMM (5, 5), RMM (5, 6) is 11% larger, for a speedup of only 1.05.

8.8.5 RMM (6, *m*)

RMMs with operands split into $k = 6$ digits were built. For this configuration, a total of $N_M = 93$ digit multiplications must be performed. m was varied from 5 to 10. Word size n was increased to 258, which is divisible by 6, for a uniform digit size of $d = 43$ bits. Uniform accumulator scheduling was employed for all variants. Table 23 lists the area and performance results for $k = 6$.

Table 23. RMM (6, *m*) area and latency.

d	(*k*, *m*)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
43	(6, 5)	110,362	1.677	596	24	40.2	4.44
43	(6, 6)	114,452	1.676	597	20	33.5	3.84
43	(6, 7)	124,165	1.690	592	18	30.4	3.78
43	(6, 8)	132,035	1.694	590	16	27.1	3.58
43	(6, 9)	137,644	1.666	600	15	25.0	3.44
43	(6, 10)	152,388	1.677	596	14	23.5	3.58

Due to the relatively smaller digit multiplier size, areas increase moderately with increasing m. For all configurations the effective clock period is just under 1.7 ns, determined by the digit multiplier datapath delay. Minimum area-latency product is achieved for RMM (6, 9), with a value of 3.44. Although its 138k μm^2 area is not small, it achieves a latency of 25 ns.

8.8.6 RMM (7, *m*)

Several RMMs were built with operands split into $k = 7$ digits. For this configuration, a total of $N_M = 126$ digit multiplications must be performed. The number of digit multipliers m was varied from 6 to 11. Operand size n was set to 259 bits to ensure a uniform digit size $d = 37$. Table 24 lists the area and performance results for $k = 7$.

Table 24. RMM (7, *m*) area and latency.

d	(*k*, *m*)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
37	(7, 6)	114,560	1.585	631	26	41.2	4.72
37	(7, 7)	113,293	1.610	621	22	35.4	4.01
37	(7, 8)	123,762	1.614	620	21	33.9	4.20
37	(7, 9)	130,158	1.600	625	19	30.4	3.96
37	(7, 10)	130,941	1.625	615	17	27.6	3.62
37	(7, 11)	142,091	1.602	624	16	25.6	3.64

RMM (7, 6) has an area of 115k μm^2, and the area increases to 142k μm^2 for RMM (7, 11). A 37×37 digit multiplier requires approximately 7.7k μm^2 of area. Thus, incrementing m by one should normally increases area by about that much. However, that does not necessarily hold here. For example, RMM (7, 7) is actually slightly *smaller* than RMM (7, 6), at just over 113k μm^2. That area for the additional digit multiplier in

RMM (7, 7) is more than offset by a decrease in accumulator datapath logic. Scheduling the six digit multipliers in RMM (7, 6) requires nine distinct states of the accumulation logic. For RMM (7, 7), only seven states are required. Similarly, RMM (7, 10) is only larger than RMM (7, 9) by about 800 μm^2. The RMM (7, 10) accumulator datapath has only five different states, versus seven for RMM (7, 9).

In all cases the attainable clock period is approximately 1.6 ns. For the $k = 7$ configurations, RMM (7, 10) has the minimum area-latency product of 3.62. It computes the Montgomery product in 17 cycles, for a total latency of 27.6 ns.

8.8.7 RMM (8, *m*)

Seven variants of RMM with $k = 8$ were built and analyzed. For this configuration, a total of $N_M = 164$ digit multiplications must be performed. The number of digit multipliers for concurrent multiplications had to be sufficiently high in order to have a reasonably low cycle count. Accordingly, m was varied within the range 8 to 14. Table 25 lists the area and performance results for $k = 8$.

Table 25. RMM (8, *m*) area and latency.

d	(*k*, *m*)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
32	(8, 8)	113,970	1.550	645	25	38.8	4.42
32	(8, 9)	120,079	1.546	647	23	35.6	4.27
32	(8, 10)	131,725	1.535	652	21	32.2	4.25
32	(8, 11)	124,298	1.769	565	19	33.6	4.18
32	(8, 12)	133,169	1.522	657	18	27.4	3.65
32	(8, 13)	133,720	1.547	646	17	26.3	3.52
32	(8, 14)	142,074	1.667	600	16	26.7	3.79

With increasing m, total area does not increase monotonically. For example, RMM (8, 10) requires 131 μm^2 of die area, but in RMM (8, 11) the area actually shrinks

to 124 μm^2. Moving from RMM (8, 12) to RMM (8, 13), the overall area increase is on the order of a few hundred square microns. This is similar to the area change from RMM (7, 9) to RMM (7, 10) previously shown, whereby area savings from the simpler accumulation logic offset the area of an additional digit multiplier.

Effective clock period hovers around 1.5 ns, except for RMM (8, 11), at which it increases to almost 1.8 ns, and RMM (8, 14) at which it increases to almost 1.7 ns. In fact, overall latency is *degraded* slightly by about 400 ps moving from RMM (8, 13) to RMM (8, 14), even though the latter requires one less cycle. Of these designs, the best latency is provided by RMM (8, 13), at 26.3 ns. It also has the minimum area-latency product for $k = 8$ of 3.52.

8.8.8 Rescheduled Montgomery Multiplier Summary

Table 26 summarizes the results for RMM instances which lie on the Pareto frontier of the latency-area plot, ordered by area.

Table 26. RMM results (Pareto frontier).

d	(*k*, *m*)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
86	(3, 1)	66,429	1.969	508	28	55.1	3.66
64	(4, 2)	76,933	1.910	524	25	47.8	3.67
64	(4, 3)	101,974	1.881	532	19	35.7	3.64
128	(2, 1)	105,697	2.217	451	15	33.3	3.52
86	(3, 2)	114,610	2.013	497	16	32.2	3.69
52	(5, 5)	118,482	1.794	557	17	30.5	3.61
64	(4, 4)	120,631	1.881	532	15	28.2	3.40
37	(7, 10)	130,941	1.625	615	17	27.6	3.62
43	(6, 8)	132,035	1.694	590	16	27.1	3.58
32	(8, 13)	133,720	1.547	646	17	26.3	3.52
43	(6, 9)	137,644	1.666	600	15	25.0	3.44
64	(4, 5)	144,877	1.904	525	13	24.8	3.59
86	(3, 3)	146,060	2.002	500	12	24.0	3.51
43	(6, 10)	152,388	1.677	596	14	23.5	3.58
128	(2, 2)	188,374	2.217	451	10	22.2	4.18

Table 27 lists the results for the remaining RMM instances.

Table 27. RMM results (non Pareto).

d	(*k*, *m*)	Total Area (μm^2)	Effective Period (ns)	Effective Frequency (MHz)	# Cycles	Total Latency (ns)	Area· Latency Product
43	(6, 5)	110,362	1.677	596	24	40.2	4.44
37	(7, 7)	113,293	1.610	621	22	35.4	4.01
32	(8, 8)	113,970	1.550	645	25	38.8	4.42
43	(6, 6)	114,452	1.676	597	20	33.5	3.84
37	(7, 6)	114,560	1.585	631	26	41.2	4.72
52	(5, 4)	116,492	1.770	565	21	37.2	4.33
32	(8, 9)	120,079	1.546	647	23	35.6	4.27
37	(7, 8)	123,762	1.614	620	21	33.9	4.20
43	(6, 7)	124,165	1.690	592	18	30.4	3.78
32	(8, 11)	124,298	1.769	565	19	33.6	4.18
37	(7, 9)	130,158	1.600	625	19	30.4	3.96
52	(5, 6)	131,231	1.816	551	16	29.1	3.81
32	(8, 10)	131,725	1.535	651	21	32.2	4.25
32	(8, 12)	133,169	1.522	657	18	27.4	3.65
32	(8, 14)	142,074	1.667	600	16	26.7	3.79
37	(7, 11)	142,091	1.602	624	16	25.6	3.64

Fig. 26 plots latency versus area for all RMM implementations. The Pareto frontier for area-latency tradeoff is indicated.

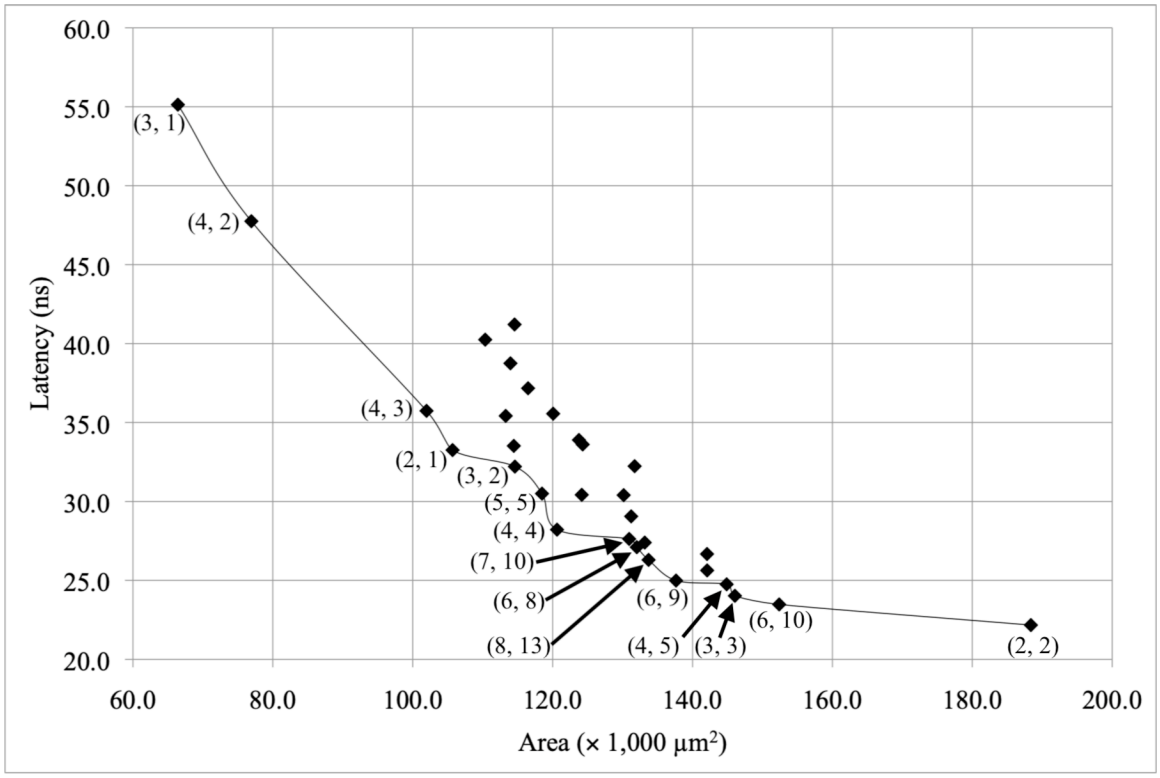


Fig. 26. RMM latency versus area with Pareto frontier.

Unsurprisingly, increasing area generally purchases a reduction in latency. Of course, 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.

All configurations with $k \in \{2, 3, 4\}$ lie on the Pareto frontier, whereas only one configuration each for $k \in \{5, 7, 8\}$ lies on the frontier. Each of those has the minimum area-latency product for that k. There are three configurations with $k = 6$ on the frontier: RMM (6, 8), (6, 9), and (6, 10), with the $k = 6$ minimum area-latency product of 3.44 achieved in configuration (6, 9).

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 $m = 1.5k$. Thus, for $k = 6$, the minimum area-latency product of 3.44 is obtained configuration (6, 9). It has a 25 ns latency in an area of 138k μm^2. For $k = 7$, configuration (7, 10) has an area-latency product of 3.62. For $k = 8$, the best configuration is (8, 13) with area-latency product 3.52.

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 $52 \times 52 \times 5 = 13{,}520$ digit multiplication bits in flight, whereas RMM (6, 6) has $43 \times 43 \times 6 = 11{,}094$ 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 central clustering is also partly a function of the configurations that were chosen for analysis. For example, for the $k \geq 5$ RMM configurations, designs could have been built with small values of m (1, 2, *etc.*) as well as large values of m further beyond $1.5k$. The latency curve would approach vertical for smaller and smaller values of k with small m because of the digit count to digit product quadratic relationship. Conversely, it would flatten toward horizontal for a given value of k and with increasing values of m as more and more circuitry were added and the cycles approach a 5-cycle minimum.

The latency-area curves of Fig. 26 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. This is the case even 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. For higher performance the plot suggests that the better tradeoff is in configurations with a small number of large digits ($k = 4, 3, 2$).

8.9 Montgomery Multiplier Comparisons

Table 28 lists the area and performance results for most of the fully functional Montgomery multiplier architectures, plus the 256×256-bit synthesized and the pipelined Karatsuba-Ofman multipliers. The Eberle, *et al.* design variants that employ lookup tables are excluded, because their poor area-latency tradeoffs preclude their utility. Only

those RMM instances lying on the Pareto frontier are listed. The table orders the results from lowest to highest area. In all cases the performance figures are for computing a 256-bit Montgomery product from two 256-bit operands.

Table 28. Montgomery 256-bit multipliers ranked by area.

Architecture	Total Area (μm^2)	Effective Period (ns)	# Cycles	Total Latency (ns)	Area· Latency Product
Eberle Digit-Digit 8×8, $k = 32$	12,462	0.967	2,817	2,724.0	33.95
Eberle Digit-Digit 16×16, $k = 16$	14,125	1.296	897	1,162.5	16.42
Tenca Bit-Digit 1×32, $k = 8$, $m = 3$	16,655	0.944	777	733.5	12.22
Eberle Digit-Digit 32×32, $k = 8$	19,605	1.599	321	513.3	10.06
Großschädl Bit-Word 1×256, $d = 8$	19,934	0.810	292	236.5	4.72
Großschädl Bit-Word 1×256, $d = 16$	20,093	0.825	276	227.7	4.58
Großschädl Bit-Word 1×256, $d = 32$	20,760	0.820	268	219.8	4.56
Tenca Bit-Digit 1×16, $k = 16$, $m = 9$	20,869	0.849	586	497.5	10.38
Tenca Bit-Digit 1×8, $k = 32$, $m = 17$	23,124	0.789	576	454.5	10.51
RMM (3, 1), $d = 86$	66,429	1.969	28	55.1	3.66
RMM (4, 2), $d = 64$	76,933	1.910	25	47.8	3.67
RMM (4, 3), $d = 64$	101,974	1.881	19	35.7	3.64
RMM (2, 1), $d = 128$	105,697	2.217	15	33.3	3.52
RMM (3, 2), $d = 86$	114,610	2.013	16	32.2	3.69
RMM (5, 5), $d = 52$	118,482	1.794	17	30.5	3.61
RMM (4, 4), $d = 64$	120,631	1.881	15	28.2	3.40
RMM (7, 10), $d = 37$	130,941	1.625	17	27.6	3.62
RMM (6, 8), $d = 43$	132,035	1.694	16	27.1	3.58
RMM (8, 13), $d = 32$	133,720	1.547	17	26.3	3.52
RMM (6, 9), $d = 43$	137,644	1.666	15	25.0	3.44
RMM (4, 5), $d = 64$	144,877	1.904	13	24.8	3.59
RMM (3, 3), $d = 86$	146,060	2.002	12	24.0	3.51
RMM (6, 10), $d = 43$	152,388	1.677	14	23.5	3.58
RMM (2, 2), $d = 128$	188,374	2.217	10	22.2	4.18
Pipelined Karatsuba-Ofman 256×256[a]	251,949	1.620	22	35.6	8.98
Synthesized 256×256[a]	289,483	2.504	4	10.0	2.90
McIvor MUL256×256 only	583,830	1.496	35	52.4	30.57
McIvor ECC Processor	640,106	1.496	35	52.4	33.52
Full Direct Parallel	698,628	6.895	1	6.9	4.82
Full Direct Optimized Pipelined[b]	698,660	2.512	4	10.0	7.02
Full Direct Pipelined[c]	711,744	2.496	4	10.0	7.11

[a]Optimistic and does not reflect actual implementation of full algorithm.
[b]Throughput 2.51 ns, area-throughput product 1.76.
[c]Throughput 2.50 ns, area-throughput product 1.78.

Visual representations of the results collected in Table 28 can aid in better understanding the strengths and weaknesses of the various architectures and the tradeoffs among them. Fig. 27 plots latency versus area for the serial architectures along with several configurations of the Rescheduled Montgomery Multiplier.

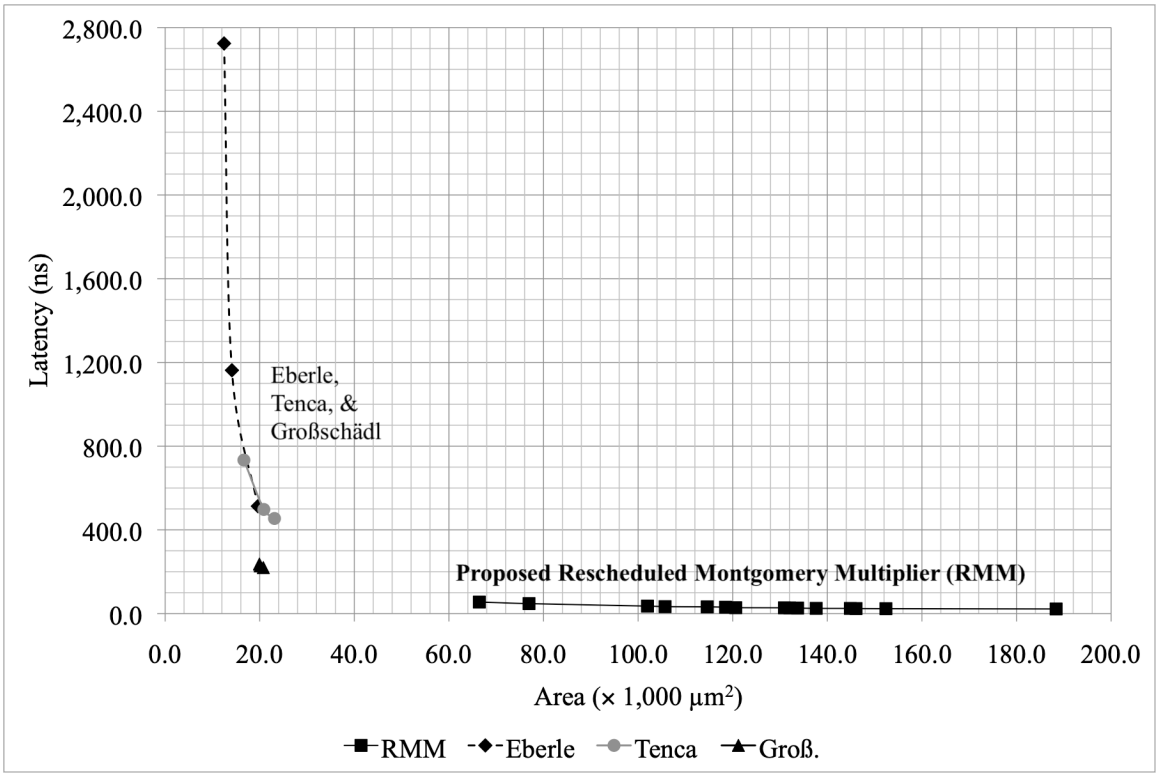


Fig. 27. Latency versus area, serial architectures and RMM.

The serial architectures are clustered near the left side of the plot with low areas and varied, relatively high latencies, while the RMM architectures vary in size but all have low latency well under 100 ns. The Eberle, Tenca, 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 necessarily 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 other words, the cycle count tends to increase faster than the clock period decreases. In contrast, the Großschädl architecture, while still small on the order of 20k μm^2, has latencies all clustered just above 200 ns. Its three configurations compute the Montgomery product identically. 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 Eberle and Tenca architectures possess an advantage in that they can readily support arbitrary operand sizes. The RMM designs all have latencies on the order of 50 ns and less, but of course the area varies greatly. The smallest RMM is just over three times the size of the Großschädl architectures, but is five times as fast.

Fig. 28 plots latency versus area for the Rescheduled Montgomery Multiplier and the word size architectures.

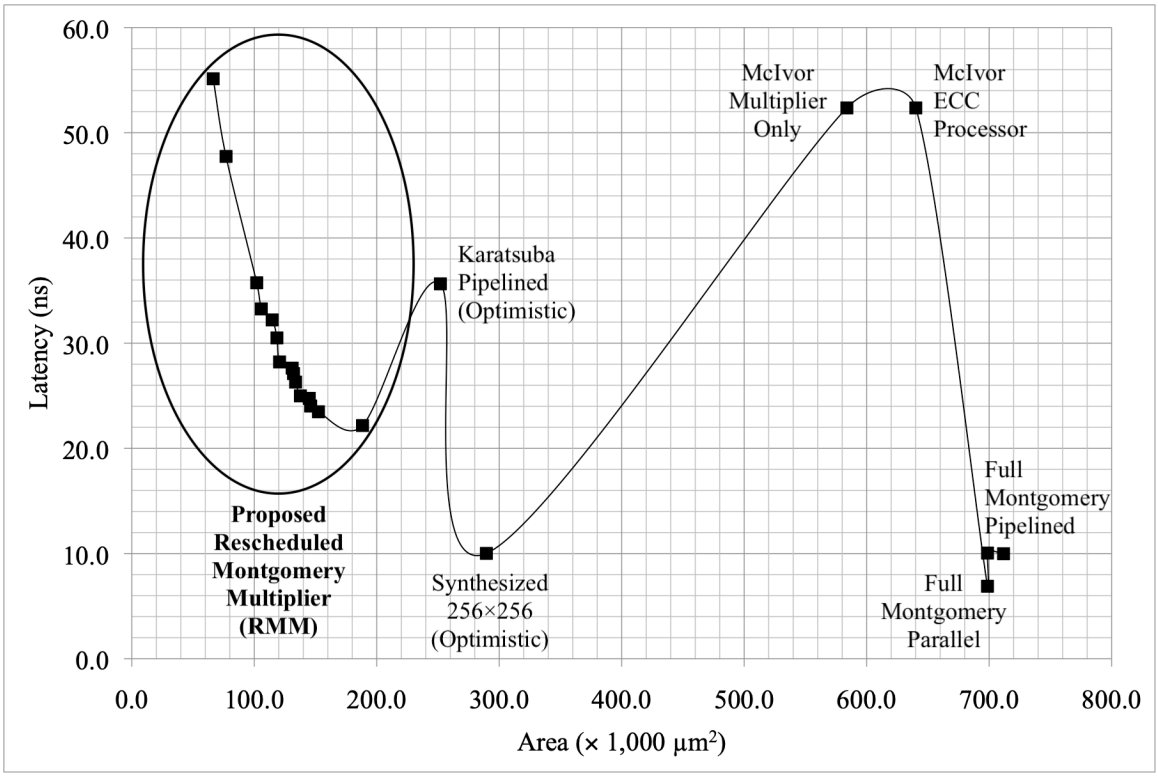


Fig. 28. Latency versus area, RMM and full word architectures.

The RMM designs are all smaller than the word size architectures, and in some cases even have superior performance. Most of the RMMs are faster than the pipelined Karatsuba-Ofman multiplier, despite being smaller. The latter's pipelining contributes to its area bloat and performance degradation similarly to the McIvor design. The 256-bit simple synthesized multiplier is potentially faster, but its performance is overly optimistic. Neither the Karatsuba-Ofman nor the simple synthesized multiplier is complete, in that each is strictly the multiplier component. Both lack the additional circuitry required to compute a complete Montgomery result. Furthermore, since they operate on full size operands, neither of these multipliers can be further optimized for the Montgomery Q_0 computation.

The McIvor ECC Processor, as well as its constituent multiplier considered in isolation, are massive, on the order 0.5 mm^2. Even with the large scale of resources devoted to Montgomery multiplication, they have substantially worse performance than most of the RMM designs. The results here suggest that it is overpipelined for an ASIC realization. The large number of pipeline registers contributes to area growth, and the resulting large number of cycles contributes to high latency. Overlapping the intermediate products T, Q, and U is also not possible due to their mutual dependencies.

The full directly-implemented Montgomery architecture, in both parallel and pipelined versions, is capable of very high performance. Both pipelined versions can support a high throughput of one Montgomery product every 2.5 ns. This would be especially advantageous for RSA's modular exponentiation and ECC's point operations, both of which employ many modular multiplications in sequence. Their performance is costly, requiring 0.7 mm^2 of die area at the 45 nm process node.

Finally, Fig. 29 combines the results from the preceding two plots and depicts the results of all designs. The latency axis uses a logarithmic scale.

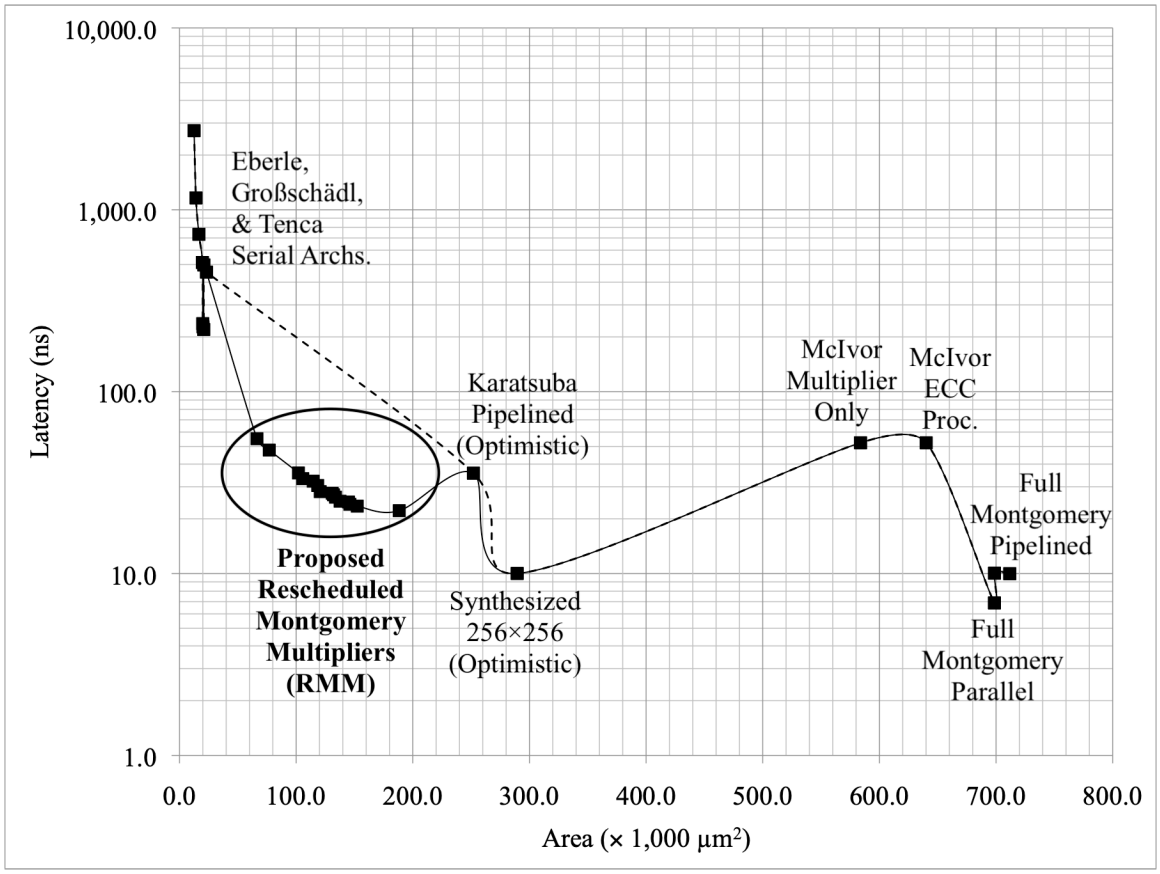


Fig. 29. Latency versus area for implemented Montgomery multipliers.

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.

Chapter 9 Conclusions

This dissertation presents the Serial Montgomery Model, a fundamental expansion of an established taxonomy commonly used to categorize serial realizations of the Montgomery algorithm. The Serial Montgomery Model encompasses 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 augments the prior taxonomy with a new type of digit scheduling termed Separated Product Scheduling (SPS). The Serial Montgomery Model provides expressions that take account of the number of digit multiplications, dependency relationships, and the number of digit multipliers to estimate the number of cycles a particular realization will require to compute a result.

This dissertation also presents a novel hardware architecture for Montgomery multiplication, termed the Rescheduled Montgomery Multiplier. The architecture synthesizes techniques from a diverse set of sources. It borrows the concept of digit multiplication from serial approaches. It then improves on that 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. This minimizes 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.

9.1 Results

The Rescheduled Montgomery Multiplier 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.

9.2 Future Research

Although the Rescheduled Montgomery Multiplier architectures proposed in this research are limited to 256-bit operands and have fixed control circuits, it is not hard to envision adding programmability to support larger operands. In lieu of a fixed accumulator block with a finely-tuned set of accumulation states, a standardized accumulator block could be designed. During computation of the T, Q_0, and U products, the accumulator path could effectively "slide" right to left under microprogram control. Outputs from the plurality of digit multipliers would be correctly aligned via multiplexers and fed into the accumulator block during each cycle. Such an approach would require a more complex accumulator block but would offer flexibility for arbitrary operand sizes while still avoiding "unnecessary" reduction accumulations.

In the current technological era with the ubiquity of mobile devices, energy consumption is a primary concern. The designs realized here were taken through logic synthesis to gates in a 45 nm process node. This provides a first order estimate of area and performance, especially for comparison among different architectures. An obvious next step would be to take the synthesized designs and run them through the place and route flow. Not only would this provide even more accurate area and performance estimates, it would facilitate making valid estimates of energy consumption. In modern deep submicron technologies, the interconnect between gates and the effects of parasitic

capacitance become more prevalent. With a routed database, effects of crosstalk on performance and power can be estimated and incorporated into making more accurate predictions.

With its capacity to analyze digit level parallelism, the Serial Montgomery Model can be used to evaluate the effects of adding parallelism to previous serial architectures. For example, a second digit multiplier can be added to the Eberle serial architecture, placing it into the CIOS/2 category of the new classification scheme. Using the expressions provided, it will be straightforward to estimate the performance of the revised architecture. In many cases it is expected that the Rescheduled Montgomery Multiplier can exploit digit level parallelism more efficiently than the previous serial architectures. Accordingly, serial designs to a degree of concurrent digit multiplication is added can be compared to equivalent RMM designs with the same number of digit multipliers.

References

[1] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed., New York: John Wiley and Sons, 1996.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM*, vol. 21, no. 2, pp. 120-126, 1978.

[3] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, New York: Springer-Verlag, 2004.

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

[5] M. Amara and A. Siad, "Elliptic Curve Cryptography and Its Applications," *7th International Workshop on Systems, Signal Processing and their Applications (WOSSPA)*, pp. 247-250, Corne d'Or, Tipaza, Algeria, May 2011.

[6] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications*, Cambridge University Press, Cambridge, UK, 1986.

[7] B. S. Kaliski, Jr., "The Montgomery Inverse and Its Applications," *IEEE Trans. Computers*, vol. 44, no. 8, pp. 1064-1065, August 1995.

[8] E. Savas and Ç. K. Koç, "The Montgomery Modular Inverse—Revisited," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 763-766, July 2000.

[9] A. A.-A. Gutub, A. F. Tenca, and Ç. K. Koç, "Scalable VLSI Architecture for *GF*(*p*) Montgomery Modular Inverse Computation," *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2002)*, pp. 46-51, Pittsburgh, PA, April 25-26, 2002.

[10] 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.

[11] 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.

[12] S. E. Eldridge and C. D. Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm," *IEEE Trans. Computers*, vol. 42, no. 6, pp. 693-699, June 1993.

[13] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," *Proc. 12th IEEE Symposium on Computer Arithmetic*, pp. 193-199, Bath, England, UK, July 19-21, 1995.

[14] O. Nibouche, A. Bouridane, and M. Nibouche, "Architectures for Montgomery's Multiplication," *IEE Proc. – Computers and Dig. Techniques*, vol. 150, no. 6, pp. 361-368, November 2003.

[15] S. S. Erdem, T. Yanik, and A. Çelebi, "A General Digit-Serial Architecture for Montgomery Modular Multiplication," *IEEE Trans. Very Large Scale Integration (VLSI) Sys.*, vol. 25, no. 5, pp. 1658-1668, May 2017.

[16] 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.

[17] 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.

[18] H. Eberle, N. Gura, and S. Chang-Shantz, "A Cryptographic Processor for Arbitrary Elliptic Curves over *GF*(2^m)," *Proc. Appl.-Specific Syst., Arch., and Processors (ASAP'03)*, pp. 444-454, The Hague, Netherlands, June 2003.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] M. O. Sanu, E. E. Swartzlander, Jr., and C. M. Chase, "Parallel Montgomery Multipliers," *Proc. 15th IEEE Conf. Appl.-Specific Syst., Arch., and Processors (ASAP '04)*, pp. 63-72, Galveston, TX, September 27-29, 2004.

[24] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, vol. 34, pp. 349-356, 1965.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] S. Williams. *Icarus Verilog*. http://iverilog.icarus.com.

[34] Synopsys, Inc. *Design Compiler*. https://www.synopsys.com.

[35] Synopsys, Inc. *PrimeTime*. https://www.synopsys.com.

[36] 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.

[37] *Open Source Static RAM Compiler*, University of California at Santa Cruz, https://openram.soe.ucsc.edu.

[38] *FreePDK Open Source 45 nm Physical Design Kit (PDK)*. https://www.eda.ncsu.edu/wiki/FreePDK.

Vita

Trenton J. Grale was born in suburban Cleveland, Ohio. He earned a Bachelor of Arts (with honors) at Colgate University in Hamilton, New York and a Master of Science in Engineering at the University of Texas at Austin in Austin, Texas. He is a member of Phi Beta Kappa, Tau Beta Pi, and Eta Kappa Nu. He holds five patents in integrated circuit design.

Permanent address: Trenton J. Grale
P.O. Box 9118
Austin, TX 78766-9118

tgrale@utexas.edu

This dissertation was typed by the author.