A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm

Alexandre F. Tenca, *Member*, *IEEE*, and
Çetin K. Koç, *Senior Member*, *IEEE*

Abstract—This paper presents a scalable architecture for the computation of modular multiplication, based on the Montgomery multiplication (MM) algorithm. A word-based version of MM is presented and used to explain the main concepts in the hardware design. The proposed multiplier is able to work with any precision of the input operands, limited only by memory or control constraints. Its architecture gives enough freedom to select the word size and the degree of parallelism to be used, according to the available area and/or desired performance. Design trade offs are analyzed in order to identify adequate hardware configurations for a given area or bandwidth requirement.

Index Terms—Cryptography, Montgomery multiplication, modular multiplication, modular multiplier, scalable multiplier.

1 Introduction

MONTGOMERY multiplication [1] is an efficient method for modular multiplication with an arbitrary modulus, particularly suitable for implementation on general-purpose computers and embedded microprocessors. The method is based on a representation of the residue class modulo M. The algorithm uses simple divisions by a power of two instead of divisions by M, which are used in a conventional modular operation.

Modifications of the original method [2], [3], [4], [5], [6] target more efficient software implementations on specific processors, arithmetic coprocessors, or specific hardware implementations. Many algorithms and hardware implementations based on Montgomery multiplication were developed for a fixed precision of the operands [4], [2], [7], [5], [6]. The disadvantage of such designs is to disregard the variable-precision multiplication feature that is common in software implementations. In order to get improved performance, high-radix algorithms and designs have also been proposed [2], [8]. However, these designs are usually complex and it is not so evident whether they provide the desired speed gain. A theoretical investigation of the design trade offs for high-radix modular multipliers is given in [9]. Low-radix designs are usually more attractive for hardware implementation [10].

The Montgomery multiplication (MM) is the basic operation used in modular exponentiation [11], [12], which is required in the Diffie-Hellman and RSA public-key cryptosystems [13], [14]. Recent implementations of the Montgomery Multiplication are focused on elliptic curve cryptography [15] over the finite fields $GF(p)$ and $GF(2^m)$ [16].

A major design concern for multiplication units used in cryptography is the large number of operand bits, which causes large fan-out of signals, large wire delays, and complex routing. These problems are reduced in systolic architectures [17], [18], at the cost of extra hardware resources. However, these architectures are usually tailored for fixed-precision computation and they are not equivalent to the architecture presented in this work. A possible approach to obtain a similar architecture using systolic arrays would consist of generating a systolic array for MM, applying a partitioning method [19] to break the array into a smaller network, and creating the infrastructure required to reuse it. This approach was not followed in this work.

The word-based version of the MM algorithm and a scalable architecture derived from it are described in more detail in this paper than in [20]. Both deal with arithmetic operations performed in radix-2 Montgomery multiplication (presented in Section 3), but they manipulate operand words, as done in high-radix algorithms used in software. However, differently from high-radix algorithms, the proposed algorithm avoids the use of costly digit multiplications and, this way, it allows the generation of simple hardware implementations and the exploration of several design trade offs to obtain the best performance in a limited chip area, without limiting the operand precision. Practical limits to the precision are imposed by the control and memory subsystems, but not the data path. The algorithm is used to explain the hardware design, thus it is not proposed for software implementation. High-radix designs for our scalable architecture were reported in [21], [22].

Section 2 contains a brief discussion on the scalability requirement imposed to our design. A presentation of the general theoretical aspects of the Montgomery multiplication is given in Section 3. The word-based algorithm is presented in Section 4, followed by a discussion about its mapping to hardware (Section 5). The scalable architecture for modular multiplication is described in Section 6. The results obtained using a design synthesis tool for $0.5\mu m$ CMOS technology are presented in Section 7 and then used to create a first-order system model to evaluate the design trade offs.

2 A Scalable Architecture

The problem solved with the proposed word-based architecture is related to the inability of most designs to handle more precision than the one for which the system was designed. This aspect is considered in this paper as a *scalability* feature. For example, a multiplier designed for 768 bits [10] cannot be immediately used in a system which needs 1,024 bits. The functions performed by such designs are not consistent with the ones required in the larger precision system and the multiplier needs to be redesigned. In order to make the hardware scalable, it is necessary to have the ability to reuse it in both space and time until the desired result is obtained. The usual solution is to use software and standard digit multipliers. The algorithms for software implementation of Montgomery multiplication are presented in [16], [3]. The complexity of software-oriented algorithms is much higher than the complexity of the radix-2 algorithm and implementation [4], making a direct hardware implementation of a software-oriented algorithm unattractive.

3 Montgomery Multiplication

The application of the Montgomery Multiplication (MM) algorithm on two integers X and Y, with required parameters for n bits of precision, will result in the number $Z = \mathrm{MM}(X,Y) = XYr^{-1} \bmod M$, where $r = 2^n$ and M is an integer in the range $2^{n-1} < M < 2^n$ such that $\gcd(r, M) = 1$. Since $r = 2^n$, it is sufficient that the modulus M be an odd integer. For cryptographic applications, M is usually a prime number or a product of primes, thus this condition is easily satisfied. The *image or the M-residue* of an integer a is defined as $\overline{a} = ar \bmod M$. It is easy to show that the Montgomery multiplication over the images $\overline{a}$ and $\overline{b}$ computes the image $\overline{c} = \mathrm{MM}(a,b)$, which corresponds to the integer $c = ab \bmod M$ [3]. The transformation between the image and the integer set is accomplished using MM as follows:

• *The authors are with the Department of Electrical and Computer Engineering, Oregon State University, Corvallis, OR 97331-3211.
E-mail: {tenca, koc}@ece.orst.edu.*

*Manuscript received 11 Feb, 2002; revised 11 July, 2002; accepted 10 Dec, 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115877.*

```
$S = 0$
for $i = 0$ to $n - 1$
    if $(S + x_i Y)$ is even
        then $S := (S + x_i Y)/2$
        else $S := (S + x_i Y + M)/2$
    end if
end for
if $S \geq M$ then $S := S - M$
```

Fig. 1. Radix-2 Montgomery Multiplication algorithm.

- From the integer value to the M-residue: $\overline{a} = \mathrm{MM}(a, r^2) = ar \bmod M$.
- From the M-residue to the integer value: $a = \mathrm{MM}(\overline{a}, 1) = arr^{-1} \bmod M = a \bmod M$.

Usually $r^2 \pmod M$ is precomputed and saved. Thus, only a single MM is needed to perform either one of these transformations. The MM algorithm has a much lower complexity than a regular modular multiplication, which requires a division operation. The conversion overhead is greatly compensated by the fast multiplication operation over the M-residue values. Another important advantage of MM over conventional multiplication is exposed in modular exponentiation, when multiple MMs are computed over the M-residues before the result is converted back to the original integer set.

The radix-2 Montgomery multiplication algorithm for n-bit operands $X = (x_{n-1}, ..., x_1, x_0)$, Y, and M is given in Fig. 1. Higher radices may be used but the radix-2 provides a simple hardware implementation.

The simple operations in this algorithm are easily implemented in hardware. However, these operations are performed with full-precision operands and, in this sense, they have an intrinsic limitation. Once a hardware based on this algorithm is defined for n bits, it cannot work with more bits. To remove this limitation and keep the simple operations of the radix-2 algorithm, a modified algorithm is proposed, as presented in the next section.

4 A Word-Based Radix-2 Montgomery Multiplication Algorithm

Given two operands Y (multiplicand) and X (multiplier) and the modulus M, the algorithm presented in this section executes a series of operations to generate $XYr^{-1} \bmod M$, scanning Y and M word-by-word and scanning X bit-by-bit. This characteristic enables us to derive a hardware implementation that is very regular and based on simple operations. The *Multiple Word Radix-2 Montgomery Multiplication algorithm (MWR2MM)* is presented in Fig. 2 and it is a refined version of the one presented in [20]. The final reduction step was intentionally omitted. The work in [23] describes the conditions for which the reduction is eliminated when multiple multiplications are performed.

In this algorithm, the n-bit operands are split into w-bit words. For now, suppose that e words are used. The actual value of e is discussed in Section 5. Word and bit vectors are represented as: $M = (0, M^{(e-1)}, ..., M^{(1)}, M^{(0)})$, $Y = (0, Y^{(e-1)}, ..., Y^{(1)}, Y^{(0)})$, $S = (0, S^{(e-1)}, ..., S^{(1)}, S^{(0)})$, and $X = (x_{n-1}, ..., x_1, x_0)$, where the words are marked with superscripts and the bits are marked with subscripts. M, Y, and S are extended to $e + 1$ words by a most-significant zero word.

The concatenation of two vectors A and B is represented as (A, B). A particular range of bits in a vector A from position i to position j, $j > i$, is represented as $A_{j..i}$. The bit position i of the kth word of an operand A is represented as $A_i^{(k)}$.

```
$S = 0$
for $i = 0$ to $n - 1$
    $(C_a, S^{(0)}) := x_i Y^{(0)} + S^{(0)}$
    if $S_0^{(0)} = 1$ then
        $(C_b, S^{(0)}) := S^{(0)} + M^{(0)}$
        for $j = 1$ to $e$
            $(C_a, S^{(j)}) := C_a + x_i Y^{(j)} + S^{(j)}$
            $(C_b, S^{(j)}) := C_b + M^{(j)} + S^{(j)}$
            $S^{(j-1)} := (S_0^{(j)}, S_{w-1..1}^{(j-1)})$
        end for
    else
        for $j = 1$ to $e$
            $(C_a, S^{(j)}) := C_a + x_i Y^{(j)} + S^{(j)}$
            $S^{(j-1)} := (S_0^{(j)}, S_{w-1..1}^{(j-1)})$
        end for
    end if
    $S^{(e)} = 0$
end for
```

Fig. 2. The MWR2MM algorithm.

The total carry-out value generated in each j loop iteration corresponds to $C_a + C_b$ (one for each addition) and it is in the range $[0, 2]$. This range for the carry value satisfies the condition imposed by the addition of three w-bit words and a maximum carry value C_{max} (generated by previous word additions), which is determined as : $3(2^w - 1) + C_{max} = C_{max}2^w + 2^w - 1 \Rightarrow C_{max} = 2$.

The algorithm computes a new partial sum S for each bit of X, scanning the words of the present S, Y, and M. Once Y is completely read, another bit of X is taken and the scan is repeated. The arithmetic operations are performed in w bits of precision. The number of loop iterations is adjusted to accomplish the modular multiplication in the required final precision, without modifications to the inner structure. This is the main feature we explore in the modular and scalable architecture shown in this paper.

A shift right operation must be performed in each i loop. In the multiple-word computation of the shifted value, a new $S^{(j-1)}$ word is computed only when the least-significant bit of the new $S^{(j)}$ word is obtained. This operation is described in the MWR2MM algorithm by the expression: $S^{(j-1)} := (S_0^{(j)}, S_{w-1..1}^{(j-1)})$, which is inside each of the inner loops. Inserting an extra most-significant word with value 0 allows the computation of $S^{(e-1)}$ once the loop is completed. Since e words are enough to represent the values in S, the word $S^{(e)}$ will be always zero. Thus, $e + 1$ words are scanned in each i loop.

5 Mapping the MWR2MM Algorithm to Hardware

The dependency between operations within the j loop restricts their parallel execution due to the dependency on the carry bits. However, instructions in different i loops may be executed in parallel. The dependency graph for the algorithm is shown in Fig. 3a. An atomic task is represented by a circle and it is labeled according to the type of action it performs. Tasks A and B execute basically the following steps:

1. Add one word from each one of the vectors S, $x_i Y$, and M (the addition of M depends on a test), and

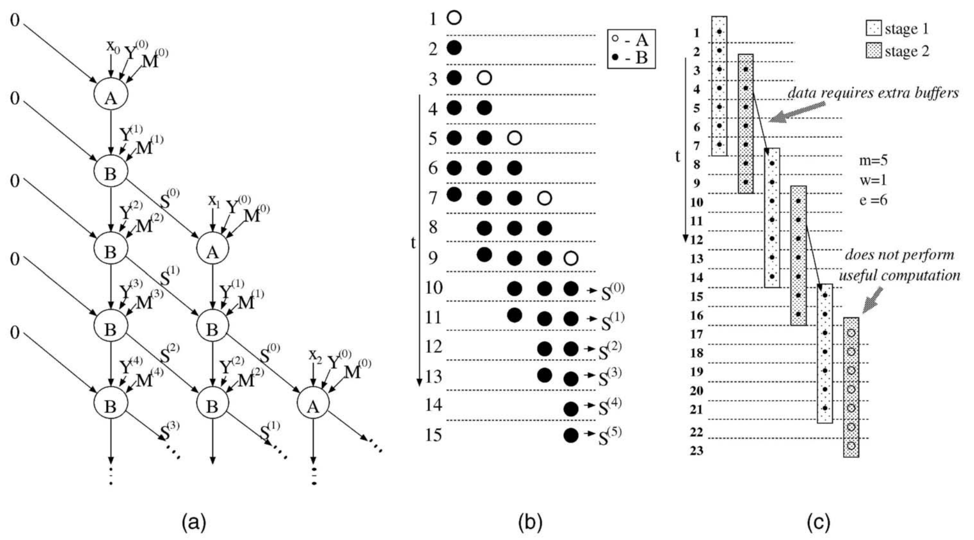

○ - A
● - B
stage 1
stage 2
data requires extra buffers
m=5
w=1
e=6
does not perform useful computation
(a)
(b)
(c)

Fig. 3. (a) Dependency graph for the MMR2MM algorithm, (b) example of its computation for 5-bit operands, where $w = 1$ bit and $L = 2$, and (c) same computation with different number of PEs.

2. One-bit right shift of an S word. For this operation, the generation of the shifted $S^{(j-1)}$ is possible only after computing the least significant bit of $S^{(j)}$.

Task A differs from task B because, additionally to these two steps, it also needs to test the *even* condition for the result of the addition $x_iY^{(0)} + S^{(0)}$ (shown in Fig. 2, Step 4), and store this test result for the other tasks dealing with the next words of the same operands (same column in the graph).

The dependency graph (Fig. 3a) has $e + 1$ tasks per column. Each column may be computed by a separate *processing element* (PE) and the data generated by one PE may be passed to another PE in a pipelined fashion. Each task is computed in one clock cycle.

The actual value of e depends on the representation of S. The range of S ($[0, 2M - 1]$) must be taken into account to determine e. When S is represented in nonredundant form, then $e = \lceil \frac{n+1}{w} \rceil$. When S is represented in redundant Carry-Save (CS) form, then $e = \lceil \frac{n}{w} \rceil$. The use of nonredundant representation, depending on the word size, may be appropriate for FPGA implementation where small carry-ripple adders are as fast as CS adders and occupy significantly less area. CS adders are used to make the design almost insensitive to variations in the word size (limited carry chain). In CS form, a number (S) is represented by two bit-vectors: carry vector (SC) and partial-sum vector (SS) such that the nonredundant form of S is obtained computing $SC + SS$.

An example of the computation executed on 5-bit operands using five PEs (one for each column) is shown in Fig. 3b for the word size of $w = 1$ bit ($e = 6$ words when S is in nonredundant form). The example considers that the final result is available without register delay at the end of each time unit (least-significant word of the result—$S^{(0)}$—is ready at the end of $t = 10$). Observe that there is a latency (L) of two clock cycles between processing a column for x_i and a column for x_{i+1}. In this case, there are no combinational logic paths starting in one PE and ending into another. It would be possible to make PE_i use the data coming from PE_{i-1} earlier ($L = 1$). However, the critical path in the design would be almost twice that for the case $L = 2$, which wouldn't bring any advantage in performance. For this reason, $L = 2$ is used in this work.

Fig. 3c shows what happens when only two processing modules are used for the same computation shown in Fig. 3b. In this case, the intermediate data vector S must pass through the two PEs several times. Each pass is called a *pipeline cycle* in this work. Once the first pipeline cycle starts, the time it takes to have the first word of S coming out of the pipeline (*pipeline latency*) is Lp clock cycles. Once this first word is generated, another pipeline cycle may start. However, the PEs in the pipeline may still be busy when that happens, as shown in Fig. 3c, and then the data generated by the last PE in the pipeline must be buffered for a while. Observe that the computation during the last pipeline cycle could be done with a single stage in this example. It may be the case that some PEs will be performing a nonrequired operation during the last pipeline cycle just because n is not a multiple of the number of available PEs. What happens when the precision of the X (n bits) is not a multiple of the number of PEs in the pipeline (p)?

The same problem happens in software implementations of Montgomery multiplication, when the precision is not equal to an exact multiple of the word size. It is equivalent to having an algorithm with an outer loop with $kp > n$ iterations. When p PEs are used in the pipeline, it is the same as saying that $r = 2^{kp}$, where $n \leq kp < n + p$, since $k = \lceil \frac{n}{p} \rceil$ corresponds to the number of pipeline cycles required to compute the multiplication for operands in precision n. However, the originally stated condition of $r = 2^n$, based on $2^{n-1} < M < 2^n$, was given only to reduce the number of iterations in the algorithm. Using a different value of r does not cause any problems in the computation as far as r is kept constant the whole time (which is the case in the hardwire design). In fact, it is shown in [24] that the selection of r as an integer multiple of the word size produces more efficient software implementations since bit-level operations within words are avoided. Thus, the architecture corresponds to a special case of the MWR2MM algorithm.

5.1 Performance Estimate

The total computation time T in clock cycles (assuming that each task consumes one clock cycle) when p stages are used in the pipeline to compute the MM with n bits of precision is

$$T = \begin{cases} Lkp + e - 1 & \text{if}(e + 1) \leq Lp \\ k(e + 1) + L(p - 1) & \text{otherwise.} \end{cases} \qquad (1)$$

The first case shown in the equation represents the situation when the first PE in the pipeline cannot start its computation with another bit of X because the least significant word of S didn't show up at the pipeline output yet. The second case models the condition when the number of words in the operands is large

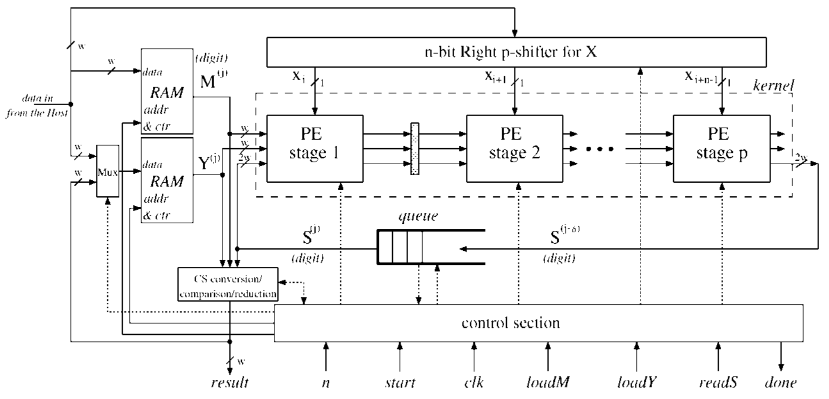


Fig. 4. Pipelined organization for the multiplier.

enough to keep all the PEs working all the time. As the word size increases, there is a reduction in the total execution time up to a lower bound that can be obtained from (1). The best parameters in terms of number of PEs and word size for a given operand precision and chip area depends on the clock cycle time of the final implementation and the total number of clock ticks. The evaluation of an ASIC implementation of the architecture is presented in Section 7.

6 A Scalable Architecture for MM

A pipelined organization for the system is shown in Fig. 4. The pipeline itself was named *kernel* in the figure and it is composed of p PEs. The other blocks represent memory, data conversion, and control unit. Each processing element in the pipeline relays the received words to the next downstream unit. All paths are w-bits wide, except for the x_i inputs (only 1 bit). The kernel itself does not limit the final computation precision. If more precision is required, it is only necessary to feed more words. The final and intermediate results are stored in the queue. Gray boxes indicate registers.

The control block function can be inferred from the algorithm description that was provided, combined with other data manipulation tasks that must be done to transfer data between the multiplier and the host system. The other portions of this design are presented in the following sections.

6.1 Memory Organization

Since the data is received word-serially by the kernel, the memory for Y, M, and S must allow the scan of operands' words. In this implementation, RAM modules are used to store Y and M. These RAMs are loaded with the operand values from the host system. The x_i bits come from a n-bit p-shift register, where p is the number of processing elements in the pipeline. In order to make the system flexible enough for several values of precision, the memory element for S is designed as a queue. The maximum length of the queue (Q_{max}) depends on the maximum number of words that may be stored in the memory (e_{max}) and the number of stages (p) in the pipeline. This length is determined as:

$$Q_{max} = \begin{cases} e_{max} + 2 - Lp & \text{if}(e_{max} + 2) > Lp \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$

The size of memory components becomes the limiting factor for the operand precision. The memory space should be previously allocated with a provision for the maximum number of words to be used in the system. In order to avoid this limitation, one alternative to compute $e > e_{max}$ words is to keep e_{max} words inside the multiplier and the other $e - e_{max}$ words in the system main memory. The host processor would then load the words in main memory into the kernel when needed. Such a solution would imply a reduction in the processing speed when the words in main memory are required.

The memory space in this design is not more than what is normally used in a conventional radix-2 design of the MM algorithm for similar maximum precision of the operands.

6.2 Result Conversion

To reduce storage and arithmetic hardware complexity, M, X, and Y are in nonredundant form. However, the internal accumulated product and final result S is received and generated in CS form ($2w$ bits per word). This design decision forces the conversion of the result from CS to nonredundant form in order to reuse it as input for another multiplication. However, the conversion does not impact the system performance significantly. For example, in a multiplication of 512-bit operands using an 8-PE kernel with $w = 8$, the multiplication takes 4,174 clock cycles (from (1)). The CS conversion module may be designed to compute one nonredundant 8-bit word every two clock cycles (assuming a fast 4-bit carry-propagate adder). Thus, since $e = 64$ words are used in this case, only 128 extra clock cycles would be needed for conversion, which represents a worse case of only 3 percent increase in the total number of clock cycles. In addition to that, this overhead could be almost eliminated if words of the converted result are immediately used as inputs for the next multiplication. A design that takes X or Y in CS form uses much more hardware, has a longer clock cycle, and requires more memory resources.

6.3 Processing Element

The block diagram of the processing element is shown in Fig. 5. The data path receives words $M^{(j)}$, $Y^{(j)}$, and $S^{(j)}$ from the previous stage in the pipeline and computes the new value of $S^{(j-1)}$. Delaying inputs M and Y, the module provides as outputs the words $M^{(j-1)}$, $Y^{(j-1)}$, and $S^{(j-1)}$. In fact, since the signals provided

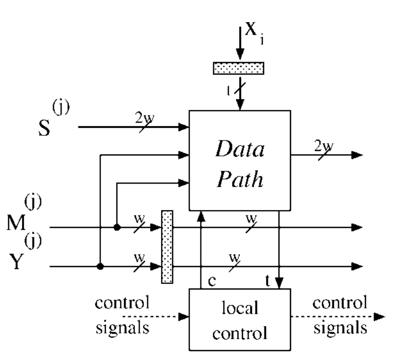
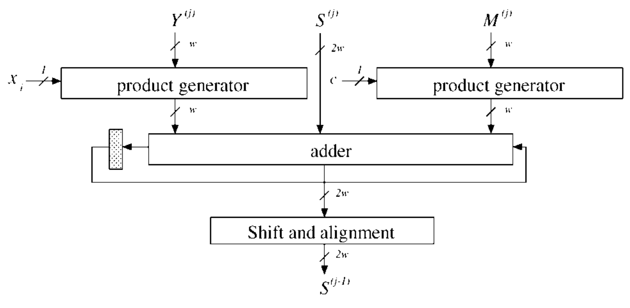
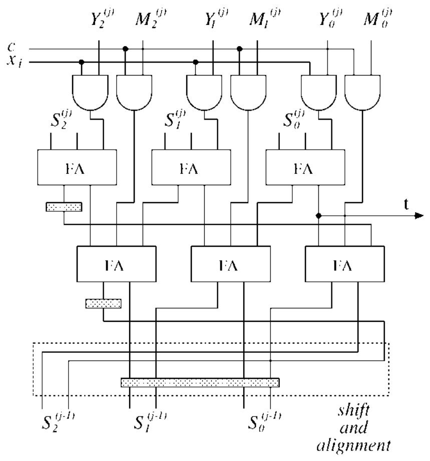


Fig. 5. The block diagram of the processing element (PE).



(a)



(b)

Fig. 6. PE data path (a) block diagram and (b) logic diagram for $w = 3$ bits.

by the PE pass by an interstage register, these signals will reach the next PE one clock cycle later. That means, when one PE is working on word j, the next PE is working on word $j-2$.

The data path needs to make the information on the least significant bit (t) of the computation $S^{(0)} + x_iY^{(0)}$ available to the local control. This bit is used to generate the control signal c (controls addition of M). The local control is responsible for generating and keeping c during a pipeline cycle, and also relay some control signals to downstream PEs. The basic operations executed in the data path are: 1) generation of the product $x_iY^{(j)}$, 2) generation of the product $cM^{(j)}$, and 3) addition of three words ($S^{(j)}$, $x_iY^{(j)}$, and $cM^{(j)}$) and a carry digit in the set $\{0, 1, 2\}$. The basic organization of the data path is shown in Fig. 6a, for a w-bit word. Observe that we are considering a 4-input CS adder, but any other adder could be used. The shaded box represents a register.

The partitioning of a long-precision adder into a w-bit adder follows the idea presented in [25] modified for least-significant-digit-first type of computation. The basic organization of the 4-input CS adder (CSA) for $w = 3$ bits is shown in Fig. 6b. It is expected that the cycle time increase with w as a result of the increase in the fan-out on x_i and c.

The data path has an alignment section to generate the output words. When computing bits of word j (step j), the circuit generates $w-1$ bits of $S^{(j)}$ and the new most significant bit of $S^{(j-1)}$. Bits of $S^{(j-1)}$ computed at step $j-1$ must be delayed and concatenated with the most significant bit generated at step j.

7 Design Evaluation

The scalable architecture has area/time trade offs that result from different values of operand precision n, word size w, and number of stages in the pipeline (p). The area used by registers for the intermediate sum, input operands, and modulus is the same for all cases discussed in this section and it was not included in the area calculations.

7.1 Experimental Results

The performance evaluation presented in this section is based on area and time estimates obtained with Mentor Graphics design tools and libraries for $0.5\mu m$ CMOS technology (ADK - ASIC Design Kit). Table 1 shows the area and time results from experiments.

Given an area A, there are many possible kernel configurations with different number of PEs and word size that satisfies the area constraint. Table 1 shows the estimated area for several kernel configurations to a maximum around 20,000 gates. This limitation of area was done arbitrarily to show a situation when a full-precision design would not be feasible. The PE area (in equivalent gates) depends only on the word size w. The experimental results obtained with the synthesis tools mentioned above allows an estimation of the PE area as: $A_{PE}(w) = 50w + 25$, which includes the local control logic. The area of each interstage latch was obtained as $A_{latch}(w) = 34w$ and, therefore, the area of a pipeline with p units may be approximated as:

$$\begin{aligned} A_{kernel}(p, w) &= (p-1)A_{latch}(w) + pA_{PE}(w) - corr \\ &= 84wp + 25p - 22w, \end{aligned}$$

where $corr$ accounts for simplifications in the last PE in the pipeline. The interstage latches can be removed at the cost of longer clock cycles (Section 5).

The clock period values shown in Table 1 consider the impact of load and wires. Observe that the increase in word size is the only parameter that effects the clock period since the architecture is very modular. Increasing the number of stages shouldn't impact the clock period after placement and routing if neighboring modules in the pipeline are kept close to each other. As a consequence of using Carry-Save adders, the increase in the word size does not have a significant impact on this design parameter.

The time to compute n bits using the scalable architecture depends on the kernel configuration (w and p). Given w and p, the total number of clock cycles to execute MM for n-bit operands is obtained from (1). This data, combined with the results shown in Table 1 are used to calculate the total execution time in microseconds shown in Fig. 7. Only two important values of

TABLE 1
Area and Clock Cycle Time for Different Kernel Configurations

Kernel Area (equivalent gates)						
p	Word size - w					
	4	8	16	32	64	128
1	233	425	825	1627	3225	6412
4	1328	2539	5009	9897	19767	
8	2792	5350	10567	20893		
16	5719	10972	21684			
32	11572					
	Clock period (ns)					
	Word size - w (bits)					
	4	8	16	32	64	128
	4.9	5.2	5.8	6.0	6.0	6.8

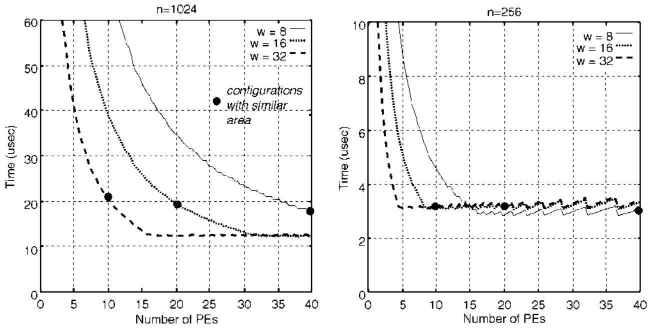


Fig. 7. Execution time of several configurations

operand precision are presented. Observe that, when the operand precision is small, the number of PEs may be small and, when the precision is high, the number of PEs should be as high as possible. Thus, the final decision on the actual configuration depends on the precision for which the hardware will be used the most and the available area. In order to make the design efficient for a large range of operand precision, the optimal solution for the highest expected precision is the best choice. Dots in the figure show configurations with equivalent area of 27-28K gates, such as $(w, p) = (8, 40)$, $(16, 20)$, and $(32, 10)$. It is easy to see that going from $p = 40$ to $p = 30$ will not affect the computation time for $n = 256$, but will impact the time to compute the multiplication for $n = 1,024$. Thus, the concept of optimal design in this case is relative to the precision of the operands and the available area. A detailed evaluation of this problem is given in [26].

7.2 Comparison with Other Approaches

The comparison with other hardware implementations of the Montgomery multiplication algorithm is not straightforward since, to the best of the authors' knowledge, there is no other hardware design that presents the same scalability features. Systolic implementations of the Montgomery multiplier such as the one in [6] are done for full precision of the operands. A systolic multiplier for $n = 512$ bits consumes about 50K gates and performs the operation in approximately $2n = 1,024$ clock cycles. Our design with a configuration of $w = 8$ and $p = 40$ uses an area of 28K and computes the multiplication in 1,103 clock cycles. The design is 8 percent slower using only slightly more than half the area. Besides, the systolic design couldn't directly compute with more than 512 bits, while our design could.

The best comparison can be made against another scalable system (software based) that uses an ARM processor and software to obtain a variable precision implementation [3]. Table 2 shows the performance of the scalable multiplier designed for an area of 28K (kernel area) and an ARM system (running at 80MHz) and operand precision of 256 and 1,024 bits. For a configuration of the scalable multiplier kernel using $w = 8$, $p = 40$, and the clock cycle period of 12.5ns (slower clock than the scalable hardware can operate in this configuration, see Table 1), the total computation time is calculated as $7.4\mu s$ and $43\mu s$, for operand precision of 256 and 1,024 bits, respectively. The speedup promoted by the scalable multiplier is significant and increases as the operand precision increases, using only a fraction of the area that would be required by a full-precision design.

TABLE 2
Performance Comparison between Scalable Multiplier Hardware ($w = 8, p = 40$) and Software on a Microprocessor System (Both with 80MHz Clock)

Operand Precision (bits)	Scalable Hardware (μs)	Software on ARM system (μs)	Speedup
256	7.4	42.3	5.7
1024	43	570	13.3

8 Conclusion

The fundamental advantage of this new architecture to compute Montgomery multiplication is its ability to manipulate any operand precision while using even a small chip area. The proposed architecture is highly flexible and allows the investigation of several design trade offs. Some possible configurations were discussed in this paper and others can be found in [26]. The proposed multiplier kernel was synthesized for a $0.5\mu m$ CMOS technology and the experimental results show that the circuit is able to work at high clock frequencies. The total time to compute the Montgomery multiplication for a given precision of the operands depends on the kernel configuration. The upper limit on the operands' precision is imposed only by the memory available to store the operands and internal results. Significant speedups can be achieved over software algorithms executing on a microprocessor system, using only a small extra chip area, which makes this design a good solution for embedded systems. An extension of this architecture to work in both $GF(p)$ and $GF(2^m)$ was proposed in [27], increasing the applicability of this architecture.

Acknowledgments

This work was partially supported by Secured Information Technology, Inc., and the US National Science Foundation under Grant CCR-0093434. The authors would like to thank Erkay Savas, Georgi Todorov, and the anonymous reviewers for their contributions. This is an extended and improved version of the paper published in the *Proceedings of CHES '99*, Lecture Notes in Computer Science, vol. 1717, Springer-Verlag.

References

[1] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. of Computation,* vol. 44, no. 170, pp. 519-521, Apr., 1985.
[2] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W.H. McAllister, eds., pp. 193-199, July 1995.
[3] Ç.K. Koç, T. Acar, and B.S. Kaliski, "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro,* vol. 16, no. 3, pp. 26-33, June 1996.
[4] A. Bernal and A. Guyot, "Design of a Modular Multiplier Based on Montgomery's Algorithm," *Proc. 13th Int'l Conf. Design of Circuits and Integrated Systems (DCIS '98),* Nov. 1998.
[5] C.-C. Yang, T.-S. Chang, and C.-W. Jen, "A New RSA Cryptosystem Hardware Design Based on Montgomery's Algorithm," *IEEE Trans. on Circuits and Systems - II: Analog and Digital Signal Processing,* vol. 45, no. 7, pp. 908-913, July 1998.
[6] C.-Y. Su, S.-A. Hwang, P.-S. Chen, and C.-W. Wu, "An Improved Montgomery's Algorithm for High-Speed RSA Public-Key Cryptosystem," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 7, no. 2, pp. 280-284, June 1999.
[7] 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.
[8] P. Kornerup, "High-Radix Modular Multiplication for Cryptosystems," *Proc. 11th IEEE Symp. Computer Arithmetic,* E. Swartzlander Jr., M.J. Irwin, and G. Jullien, eds., June 1993.
[9] C.D. Walter, "Space/Time Trade-Offs for Higher Radix Modular Multiplication Using Repeated Addition," *IEEE Trans. Computers,* vol. 46, no. 2, Feb. 1997.
[10] A. Royo, J. Moran, and J.C. Lopez, "Design and Implementation of a Coprocessor for Cryptography Applications," *Proc. European Design and Test Conf.,* pp. 213-217, Mar. 1997.
[11] T. Hamano, N. Takagi, S. Yajima, and F.P. Preparata, "O(n)-Depth Circuit Algorithm for Modular Exponentiation," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W.H. McAllister, eds., pp. 188-192, July 1995.
[12] Ç.K. Koç and T. Acar, "Fast Software Exponentiation in GF (2^k)," *Proc. 13th IEEE Symp. Computer Arithmetic,* T. Lang, J.-M. Muller, and N. Takagi, eds., pp. 225-231, July 1997.

[13] W. Diffie and M.E. Hellman, “New Directions in Cryptography,” *IEEE Trans. Information Theory,* vol. 22, pp. 644-654, Nov. 1976.

[14] 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, Feb. 1978.

[15] A.J. Menezes, *Elliptic Curve Public Key Cryptosystems.* Boston: Kluwer Academic Publishers, 1993.

[16] Ç.K. Koç and T. Acar, “Montgomery Multiplication in GF (2^k),” *Design, Codes, and Cryptography,* vol. 14, no. 1, pp. 57-69, Apr. 1998.

[17] T. Blum and C. Paar, “Montgomery Modular Exponentiation on Reconfigurable Hardware,” *Proc. 14th IEEE Symp. Computer Arithmetic,* pp. 70-77, Apr. 1999.

[18] C. Walter, “Systolic Modular Multiplication,” *IEEE Trans. Computers,* vol. 42, no. 3, pp. 376-378, Mar. 1993.

[19] D.I. Moldovan and J.A.B. Fortes, “Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays,” *IEEE Trans. Computers,* vol. 35, no. 1, pp. 1-12 Jan. 1986.

[20] A.F. Tenca and Ç.K. Koç, “A Scalable Architecture for Montgomery Multiplication,” *Proc. First Int'l Workshop Cryptographic Hardware and Embedded Systems—CHES '99,* Ç.K. Koç and C. Paar, eds., pp. 94-108, Aug. 1999.

[21] G. Todorov, “ASIC Design, Implementation, and Analysis of a Scalable High-Radix Montgomery Multiplier,” MS thesis, Oregon State Univ., Dec. 2000.

[22] A.F. Tenca, G. Todorov, and Ç.K. Koç, “High-Radix Design of a Scalable Modular Multiplier,” *Proc. Workshop Cryptographic Hardware and Embedded Systems,* Ç.K. Koç, D. Naccache, and C. Paar, eds., pp. 185-201, 2001.

[23] G. Hachez and J.-J. Quisquater, “Montgomery Exponentiation with No Final Subtractions: Improved Results,” *Lecture Notes in Computer Science,* Ç.K. Koç and C. Paar, eds., vol. 1965, pp. 293-301, 2000.

[24] T. Yanik, E. Savas, and Ç.K. Koç, “Incomplete Reduction in Modular Arithmetic,” *IEE Proc.-Computers and Digital Techniques,* vol. 149, no. 2, pp. 46-52, Mar. 2002.

[25] A.F. Tenca, “Variable Long-Precision Arithmetic (VLPA) for Reconfigurable Coprocessor Architectures,” PhD thesis, Univ. California Los Angeles, 1998.

[26] B. Kurniawan, “ASIC Design and Implementation of a Parallel Exponentiation Algorithm Using Optimized Scalable Montgomery Multipliers,” MS thesis, Oregon State Univ., Corvallis, 2002.

[27] E. Savas, A.F. Tenca, and Ç.K. Koç, “A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$,” *Proc. Second Int'l Workshop Cryptographic Hardware and Embedded Systems—CHES 2000,* Ç.K. Koç and C. Paar, eds., pp. 277-292, Aug. 2000.