A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$ [*†‡]

E. Savaş, A. F. Tenca, and Ç. K. Koç
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Abstract

We describe a scalable and unified architecture for a Montgomery multiplication module which operates in both types of finite fields $GF(p)$ and $GF(2^m)$. The unified architecture requires only slightly more area than that of the multiplier architecture for the field $GF(p)$. The multiplier is scalable, which means that a fixed-area multiplication module can handle operands of any size, and also, the wordsize can be selected based on the area and performance requirements. We utilize the concurrency in the Montgomery multiplication operation by employing a pipelining design methodology. We also describe a scalable and unified adder module to carry out concomitant operations in our implementation of the Montgomery multiplication. The upper limit on the precision of the scalable and unified Montgomery multiplier is dictated only by the available memory to store the operands and internal results, and the module is capable of performing infinite-precision Montgomery multiplication in both types of finite fields.

Key Words: Prime fields, binary extension fields, multiplication, Montgomery multiplication, scalability, hardware implementation.

1 Introduction

The basic arithmetic operations (i.e., addition, multiplication, and inversion) in prime and binary extension fields, $GF(p)$ and $GF(2^m)$, have several applications in cryptography, such as decipherment operation of RSA algorithm [18], Diffie-Hellman key exchange algorithm [3], elliptic curve cryptography [7, 12], and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm [15]. The most important of these three arithmetic operations is the field multiplication operation since it is the core operation in many cryptographic functions.

The Montgomery multiplication algorithm [13] is an efficient method for doing modular multiplication with an odd modulus. The Montgomery multiplication algorithm is a very useful for obtaining fast software implementations of the multiplication operation in prime fields $GF(p)$. The algorithm replaces division operation with simple shifts, which are particularly suitable for implementation on general-purpose computers. The Montgomery multiplication operation has been

[*] *Cryptographic Hardware and Embedded Systems - CHES 2000*, Ç. K. Koç and C. Paar, editors, Lecture Notes in Computer Science No. 1965, pages 281-296, Springer Verlag, Berlin, Germany, 2000.

[†] The reader should note that Oregon State University has filed US and International patent applications for inventions described in this paper.

[‡] This work is supported by Secured Information Technology, Inc.

extended to the finite field $GF(2^k)$ in [9]. Efficient software implementations of the multiplication operation in $GF(2^k)$ can be obtained using this algorithm, particularly when the irreducible polynomial generating the field is chosen arbitrarily. The main idea of the architecture proposed in this paper is based on the observation that the Montgomery multiplication algorithm for both fields $GF(p)$ and $GF(2^k)$ are essentially the same algorithm. The proposed unified architecture performs the Montgomery multiplication in the field $GF(p)$ generated by an arbitrary prime p and in the field $GF(2^m)$ generated by an arbitrary irreducible polynomial $p(x)$. We show that a unified multiplier performing the Montgomery multiplication operation in the fields $GF(p)$ and $GF(2^k)$ can be designed at a cost only slightly higher than the multiplier for the field $GF(p)$, providing significant savings when both types of multipliers are needed.

Several variants of the Montgomery multiplication algorithm [17, 10, 2] have been proposed to obtain more efficient software implementations on specific processors. Various hardware implementations of the Montgomery multiplication algorithm for limited precision operands are also reported [2, 17, 4]. On the other hand, implementations utilizing high-radix modular multipliers have also been proposed [17, 11, 19]. Advantages and disadvantages of using high-radix representation have been discussed in [22, 21]. Because high-radix Montgomery multiplication designs introduce longer critical paths and more complex circuitry, these designs are less attractive for hardware implementations.

A scalable Montgomery multiplier design methodology for $GF(p)$ was introduced in [21] in order to obtain hardware implementations. This design methodology allows to use a fixed-area modular multiplication circuit for performing multiplication of unlimited precision operands. The design tradeoffs for best performance in a limited chip area were also analyzed in [21]. We use the design approach as in [21] to obtain a scalable hardware module. Furthermore, the scalable multiplier described in this paper is capable of performing multiplication in both types finite fields $GF(p)$ and $GF(2^k)$, i.e., it is a scalable and unified multiplier.

The main contributions of this paper are summarized below.

- We show that a unified architecture for multiplication module which operates both in $GF(p)$ and $GF(2^m)$ can be designed easily without compromising scalability, time and area efficiency.

- We analyze the design considerations such as the effect of word length, the number of the pipeline stages, and the chip area, etc., by supplying implementation results obtained by Mentor graphics synthesis tools.

- We describe the design of a dual-field, scalable adder circuit which is suitable for the pipeline organization of the multiplier. This adder is necessary for the final reduction step in the Montgomery algorithm and in the final addition for converting the result of the multiplication operation (which is in the Carry-Save form) to the nonredundant form. Naturally, the adder operates both in $GF(p)$ and $GF(2^m)$. We give an analysis of the time and area cost of the adder circuit.

We start with a short discussion of scalability in §2 and explain the main idea behind the unified multiplier architecture in §3. We then present the methodology to perform the Montgomery multiplication operation in both types of finite fields using the unified architecture. We give the original and modified definitions of Montgomery algorithm for $GF(p)$ and $GF(2^m)$ in §4. We discuss concurrency in the Montgomery multiplication and show the methodology to design a pipeline module utilizing the concurrency in §5. We present the processing unit and the modifications needed to make the unit operate in prime and binary extension fields in §6. We then provide a multi-purpose word adder/subtractor module in §7, which can be integrated into the main Montgomery multiplier module in order to perform the field addition and subtraction operations. In §8, we discuss the area/time tradeoffs and suitable choices for word lengths, the number of pipeline stages, and typical chip area requirements. Finally, we summarize our conclusions in §9.

2 Scalable Multiplier Architecture

An arithmetic unit is called scalable if it can be reused or replicated in order to generate long-precision results independently of the data path precision for which the unit was originally designed. To speed up the multiplication operation, various dedicated multiplier modules were developed in [19, 1, 14]. These designs operate over a fixed finite field. For example, the multiplier designed for 155 bits [1] cannot be used for any other field of higher degree. When a need for a multiplication of larger precision arises, a new multiplier must be designed. Another way to avoid redesigning the module is to use software implementations and fixed precision multipliers. However, software implementations are inefficient in utilizing inherent concurrency of the multiplication because of the inconvenient pipeline structure of the microprocessors being used. Furthermore, software implementations on fixed digit multipliers are more complex and require excessive amount of effort in coding. Therefore, a scalable hardware module specifically tailored to take advantage of the concurrency of the Montgomery multiplication algorithm becomes extremely attractive.

3 Unified Multiplier Architecture

Even though prime and binary extension fields, $GF(p)$ and $GF(2^m)$, have dissimilar properties, the elements of either field are represented using almost the same data structures inside the computer. In addition, the algorithms for basic arithmetic operations in both fields have structural similarities allowing a unified module design methodology. For example, the steps of the Montgomery multiplication algorithm for binary extension field $GF(2^m)$ given in [9] only slightly differs from those of the integer Montgomery multiplication algorithm [13, 10]. Therefore, a scalable arithmetic module, which can be adjusted to operate in both types of fields, is feasible, provided that this extra functionality does not lead to an excessive increase in area or a dramatic decrease in speed. In addition, designing such a module must require only a small amount of extra effort and no major modification in control logic of the circuit.

Considering the amount of time, money and effort that must be invested in designing a multiplier module or more generally speaking a cryptographic coprocessor, a scalable and unified architecture which can perform arithmetic in two commonly used algebraic fields is definitely beneficial. In this paper, we show the method to design a Montgomery multiplier that can be used for both types of fields following the design methodology presented in [21]. The proposed unified architecture is obtained from the scalable architecture given in [21] after minor modifications. The propagation time is unaffected and the increase in chip area is insignificant.

4 Montgomery Multiplication

Given two integers A and B, and the prime modulus p, the Montgomery multiplication algorithm computes

$$C = \mathsf{MonMul}(A, B) = A \cdot B \cdot R^{-1} \pmod{p} \ , \tag{1}$$

where $R = 2^m$ and $A, B < p < R$, and p is an m-bit number. The original algorithm works for any modulus n provided that $\gcd(n, R) = 1$. In this paper, we assume that the modulus is a prime number, thus, we perform multiplication in the field defined by this prime number. This issue is also relevant when the algorithm is defined for the binary extension fields.

The Montgomery multiplication algorithm relies on a different representation of the finite field elements. The field element $A \in GF(p)$ is transformed into another element $\bar{A} \in GF(p)$ using the formula $\bar{A} = A \cdot R \pmod p$. The number $\bar{A}$ is called Montgomery image of the element, or $\bar{A}$ is said to be in the Montgomery domain. Given two elements in the Montgomery domain $\bar{A}$ and $\bar{B}$, the Montgomery multiplication computes

$$\bar{C} = \bar{A} \cdot \bar{B} \cdot R^{-1} \pmod p = (A \cdot R) \cdot (B \cdot R) \cdot R^{-1} \pmod p = C \cdot R \pmod p \ , \qquad (2)$$

where $\bar{C}$ is again in the Montgomery domain. The transformation operations between the two domains can also be performed using the MonMul function as

$$\begin{aligned} \bar{A} &= \mathsf{MonMul}(A, R^2) = A \cdot R^2 \cdot R^{-1} = A \cdot R \pmod p \ , \\ \bar{B} &= \mathsf{MonMul}(B, R^2) = B \cdot R^2 \cdot R^{-1} = B \cdot R \pmod p \ , \\ C &= \mathsf{MonMul}(\bar{C}, 1) = C \cdot R \cdot R^{-1} = C \pmod p \ . \end{aligned}$$

Provided that $R^2 \pmod p$ is precomputed and saved, we need only a single MonMul operation to carry out each of these transformations. However, because of these transformation operations, performing a single modular multiplication using MonMul might not be advantageous, however, there is a method to make it efficient for a few modular multiplications by eliminating the need for these transformations [16]. The advantage of the Montgomery multiplication becomes much more apparent in applications requiring multiplication-intensive calculations, e.g., modular exponentiation or elliptic curve point operations. In order to exploit this advantage, all arithmetic operations are performed in the Montgomery domain, including the inversion operation [6, 20]. Furthermore, it is also possible to design cryptosystems in which all calculations are performed in the Montgomery domain eliminating the transformation operations permanently.

Below, we give bitwise Montgomery multiplication algorithm for obtaining $C := ABR^{-1} \pmod p$, where $A = (a_{m-1}, \ldots, a_1, a_0)$, $B = (b_{m-1}, \ldots, b_1, b_0)$, and $C = (c_{m-1}, \ldots, c_1, c_0)$.

```
Input:    $A, B \in GF(p)$ and $m = \lceil \log_2 p \rceil$
Output:   $C \in GF(p)$
Step 1:   $C := 0$
Step 2:   for $i = 0$ to $m - 1$
Step 3:       $C := C + a_i B$
Step 4:       $C := C + c_0 p$
Step 5:       $C := C/2$
Step 6:   if $C \geq p$ then $C := C - p$
Step 7:   return $C$
```

In the case of $GF(2^m)$, the definitions and the algorithms are slightly different since we use polynomials of degree at most $m - 1$ with coefficients from the binary field $GF(2)$ to represent the field elements. Given two polynomials

$$\begin{aligned} A(x) &= a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0 \\ B(x) &= b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_1 x + b_0 \ , \end{aligned}$$

and the irreducible monic degree-m polynomial

$$p(x) = x^m + p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1 x + p_0$$

generating the field $GF(2^m)$, the Montgomery multiplication of $A(x)$ and $B(x)$ is defined as the field element $C(x)$ which is given as

$$C(x) = A(x) \cdot B(x) \cdot R(x)^{-m} \pmod{p(x)} . \tag{3}$$

We note that, as compared to Equation (1), $R(x) = x^m$ replaces $R = 2^m$. The representation of x^m in the computer is exactly the same as the representation of 2^m, i.e., a single 1 followed by 2^m zeros. Furthermore, the elements of $GF(p)$ and $GF(2^m)$ are represented using the same data structures. For example, the elements of $GF(7)$ for $p = 7$ and the elements of $GF(2^3)$ for $p(x) = x^3 + x + 1$ are represented in the computer as follows:

$$\begin{aligned} GF(7) &= \{000, 001, 010, 011, 100, 101, 110\} , \\ GF(2^3) &= \{000, 001, 010, 011, 100, 101, 110, 111\} . \end{aligned}$$

Only the arithmetic operations acting on the field elements differ. The Montgomery image of a polynomial $A(x)$ is given as $\bar{A}(x) = A(x) \cdot x^m \pmod{p(x)}$. Similarly, before performing Montgomery multiplication, the operands must be transformed into the Montgomery domain and the result must be transformed back. These transformations are accomplished using the precomputed variable $R^2(x) = x^{2m} \pmod{p(x)}$ as follows:

$$\begin{aligned} \bar{A}(x) &= \mathsf{MonMul}(A, R^2) = A(x) \cdot R^2(x) \cdot R^{-1}(x) = A(x) \cdot R(x) \pmod{p(x)} , \\ \bar{B}(x) &= \mathsf{MonMul}(B, R^2) = B(x) \cdot R^2(x) \cdot R^{-1}(x) = B(x) \cdot R(x) \pmod{p(x)} , \\ C(x) &= \mathsf{MonMul}(\bar{C}, 1) = C(x) \cdot R(x) \cdot R^{-1}(x) = C(x) \pmod{p(x)} . \end{aligned}$$

The bit-level Montgomery multiplication algorithm for the field $GF(2^m)$ is given below:

```
Input:    $A(x), B(x) \in GF(2^m)$, $p(x)$, and $m$
Output:   $C(x)$
Step 1:   $C(x) := 0$
Step 2:   for $i = 0$ to $m - 1$
Step 3:       $C(x) := C(x) + a_i B(x)$
Step 4:       $C(x) := C(x) + c_0 p(x)$
Step 5:       $C(x) := C(x)/x$
Step 6:   return $C(x)$
```

We note that the extra subtraction operation in Step 6 of the previous algorithm is not required in the case of $GF(2^m)$, as proven in [9]. Also, the addition operations are different. While addition in binary field is just bitwise mod 2 addition, the addition in $GF(p)$ requires carry propagation.

Our basic observation is that it is possible to design a unified Montgomery multiplier which can perform multiplication in both types of fields if an adder module, equipped with the property of performing addition with or without carry, is available. The design of an adder with this property is provided in the following sections.

The algorithms presented in this section require that the operations be performed using full precision arithmetic modules, thus, limiting the designs to a fixed degree. In order to design a scalable architecture, we need modules with the scalability property. The scalable algorithms are word-level algorithms, which we give in the following sections.

4.1 The Multiple-Word Montgomery Multiplication Algorithm for $GF(p)$

The use of fixed precision words alleviates the broadcast problem in the circuit implementation. Furthermore, a word-oriented algorithm allows design of a scalable unit. For a modulus of m-bit precision, $e = \lceil m/w \rceil$ words (each of which is w bits) are required. The algorithm proposed in [21] scans the operand B (multiplicand) word-by-word, and the operand A (multiplier) bit-by-bit. The vectors involved in multiplication operations are expressed as

$$\begin{aligned} B &= (B^{(e-1)}, \ldots, B^{(1)}, B^{(0)}) \ , \\ A &= (a_{m-1}, \ldots, a_1, a_0) \ , \\ p &= (p^{(e-1)}, \ldots, p^{(1)}, p^{(0)}) \ , \end{aligned}$$

where the words are marked with superscripts and the bits are marked with subscripts. For example, the ith bit of the kth word of B is represented as $B_i^{(k)}$. A particular range of bits in a vector B from position i to j where $j > i$ is represented as $B_{j..i}$. Finally, 0^m represents an all-zero vector of m bits. The algorithm is given below:

```
Input:     $A, B \in GF(p)$ and $p$
Output:    $C \in GF(p)$
Step 1:    $(TC, TS) := (0^m, 0^m)$
Step 2:    $(Carry0, Carry1) := (0, 0)$
Step 3:    for $i = 0$ to $m-1$
Step 4:        $(TC^{(0)}, TS^{(0)}) := a_i \cdot B^{(0)} + TC^{(0)} + TS^{(0)}$
Step 5:        $Carry0 := TC_{w-1}^{(0)}$
Step 6:        $TC^{(0)} := (TC_{w-2..0}^{(0)}|0)$
Step 7:        $parity := TS_0^0$
Step 8:        $(TC^{(0)}, TS^{(0)}) := parity \cdot p^{(0)} + TC^{(0)} + TS^{(0)}$
Step 9:        $TS_{w-2..0}^{(0)} := TS_{w-1..1}^{(0)}$
Step 10:       for $j = 1$ to $e-1$
Step 11:           $(TC^{(j)}, TS^{(j)}) := a_i \cdot B^{(j)} + TC^{(j)} + TS^{(j)}$
Step 12:           $Carry1 := TC_{w-1}^{(j)}$
Step 13:           $TC_{w-1..1}^{(j)} := TC_{w-2..0}^{(j)}$
Step 14:           $TC_0^{(j)} := Carry0$
Step 15:           $Carry0 := Carry1$
Step 16:           $(TC^{(j)}, TS^{(j)}) := parity \cdot p^{(j)} + TC^{(j)} + TS^{(j)}$
Step 17:           $TS_{w-1}^{(j-1)} := TS_0^{(j)}$
Step 18:           $TS_{w-2..0}^{(j)} := TS_{w-1..1}^{(j)}$
Step 19:       end for
Step 20:       $TS_{w-1}^{(e-1)} := 0$
Step 21:   end for
Step 22:   $C := TC + TS$
Step 23:   if $C > p$ then
Step 24:       $C := C - p$
Step 25:   return $C$
```

As suggested in [21], we use the Carry-Save form in order to represent the intermediate results in the algorithm. The result of an addition is stored in two variables $(TC^{(j)}, TS^{(j)})$, thus, they can grow as large as $2^{m+1} + 2^m - 3$ which is exactly equal to the result of the addition of three numbers at the right hand side of equations in Steps 4, 8, 11, and 16. Recall that $TC^{(j)}$ and $TS^{(j)}$ are m-bit numbers, but $TC^{(j)}$ must be seen as a number multiplied by 2 since it represents the carry vector in the Carry-Save notation. At the end of Step 21, we obtain the result in the Carry-Save form which needs an extra addition to get the final result in the nonredundant form. If the final result is greater than the modulus p, one subtraction operation must be performed as shown in Step 24.

4.2 Multiple-Word Montgomery Multiplication Algorithm for $GF(2^m)$

The Montgomery multiplication algorithm for $GF(2^m)$ is given below. Since there is no carry computation in $GF(2^m)$ arithmetic, the intermediate addition operations are replaced by bitwise XOR operations, which are represented below using the symbol $\oplus$.

```
Input:     $A, B \in GF(2^m)$ and $p(x)$
Output:    $C \in GF(2^m)$
Step 1:    $TS := 0^m$
Step 2:    for $i = 0$ to $m$
Step 3:        $TS^{(0)} := a_i B^{(0)} \oplus TS^{(0)}$
Step 4:        $parity := TS_0^{(0)}$
Step 5:        $TS^{(0)} := parity \cdot p^{(0)} \oplus TS^{(0)}$
Step 6:        $TS_{w-2..0}^{(0)} := TS_{w-1..1}^{(0)}$
Step 7:        for $j = 1$ to $e - 1$
Step 8:            $TS^{(j)} := a_i B^{(j)} \oplus TS^{(j)}$
Step 9:            $TS^{(j)} := parity \cdot p^{(j)} \oplus TS^{(j)}$
Step 10:           $TS_{w-1}^{(j-1)} := TS_0^{(j)}$
Step 11:           $TS_{w-2..0}^{(j)} := TS_{w-1..1}^{(j)}$
Step 12:       end for
Step 13:       $TS_{w-1}^{(e-1)} := 0$
Step 14:   end for
Step 15:   $C := TS$
Step 15:   return $C$
```

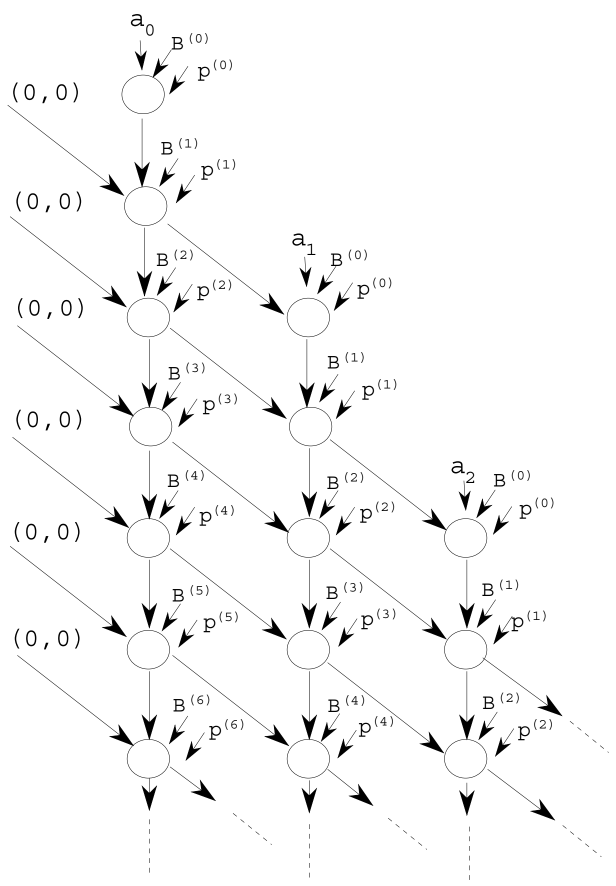
Notice that in the outer loop the index i runs from 0 to m. Since $(m + 1)$ bits are required to represent irreducible polynomial of $GF(2^m)$, we prefer to allocate $(m + 1)$ bits to express the field elements. We can also modify the algorithm for $GF(p)$ accordingly for sake of uniformity. Therefore, the formula for the number of words to represent a field element for both cases is given as $e = \lceil (m + 1)/w \rceil$ where w is the selected wordsize.

5 Concurrency in Montgomery Multiplication

In this section, we analyze the concurrency in Montgomery multiplication algorithms as given in the subsections §4.1 and §4.2. In order to accomplish this task, we need to determine the inherent data dependencies in the algorithm and describe a scheme to allow the Montgomery multiplication to be computed on an array of processing units organized in a pipeline.

We prefer to accomplish concurrent computation of the Montgomery multiplication by exploiting the parallelism among the instructions across the different iterations of i-loop of the algorithms, as proposed in [21]. We scan the multiplier one bit at a time, and after the first words of the intermediate variables (TC, TS) are fully determined, which takes two clock cycles, the computation for the second bit of A can start. In other words, after the inner loop finishes the execution for $j = 0$ and $j = 1$ in ith iteration of the outer loop, the $(i + 1)th$ iteration of outer loop starts its execution immediately. The dependency graph shown in Figure 1 illustrates these computations.

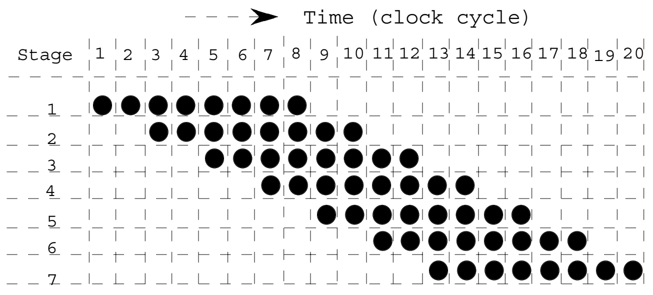
Figure 1: The dependency graph of the MonMul algorithm.



Each circle in the graph represents an elementary computation performed in each iteration of the j-loop. We observe from this graph that these computations are very suitable for pipelining. Each column in the graph represents operations that can be performed by separate processing units (PU) organized as a pipeline. Each PU takes only one bit from multiplier A and operates on each word of multiplicand, B, each cycle. Starting from the second clock cycle, a PU generates one word of partial sum $T = (TC, TS)$ in the Carry-Save form at each cycle, and communicates it to the next PU which adds its contribution to the partial sum, when its turn comes. After $e + 1$ clock cycles, the PU finishes its portion of work, and becomes available for further computation. In case there is no available PU and there is work to do, the pipeline must stall and wait for the working PUs to finish their jobs. Since the PU at the end of the pipeline has no way of communicating its result to another PU, we need to provide extra buffers for them. In the worst case, which happens when there is only one PU, there must be $2e$ extra buffers of w length to hold these partial sum words. In the last clock cycle of each column, the The PU responsible for this column must receive $p^{(e)} = B^{(e)} = 0$. Elementary computations represented by circles in Figure 1 are performed on the same hardware module. Local control module in the PU must be able to extract $TS_0^{(0)}$ and keep this value for the entire operand scanning. Each PU, in other words, has to obtain this value and use it to decide whether to add the modulus p to the partial sum. This value is determined in the first clock cycle of the each stage.

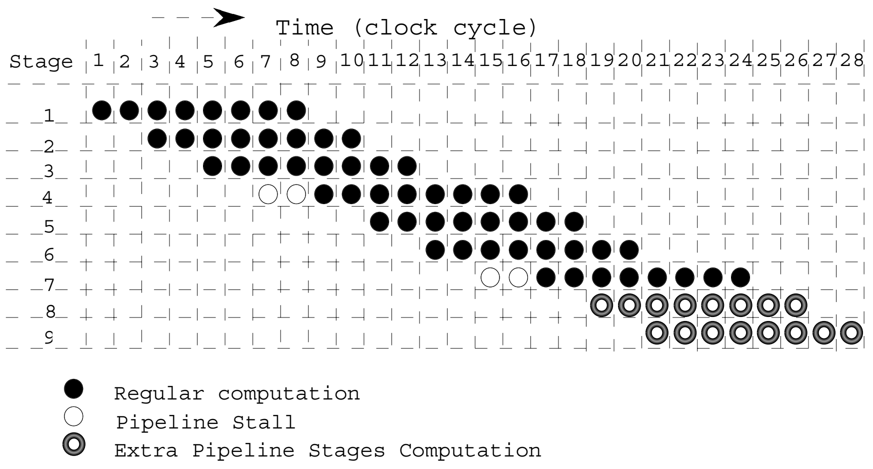
An example of the computation for 7-bit operands is shown in Figure 2 for the word size $w = 1$ provided that there are sufficient number of PUs preventing the pipeline to stall. Note that there is a delay of 2 clock cycles between the stage for x_i and the stage for x_{i+1}. The total execution time for the computation takes 20 clock cycles in this example.

Figure 2: An example of pipeline computation for 7-bit operands, where $w = 1$.



If there are at least $\lceil (e+1)/2 \rceil$ PUs in the pipeline organization the pipeline stalls do not take place. For the example in Figure 2, less than $\lceil 8/2 \rceil = 4$ PUs cause the pipeline to stall. Figure 3 shows what happens if there are only three PUs available for the same example.

Figure 3: An example of pipeline computation for 7-bit operands, illustrating the situation of pipeline stalls, where $w = 1$.



At the clock cycles 7 and 15, the pipeline cannot engage a PU, and thus, it must stall for 2 extra cycles. At the 9th and 17th cycles, the first PU becomes available and computation proceeds.

We need a buffer of 4-bit length to store the partial sum bits during the stall. Because the 8 is not a multiple of 3, the last two pipeline stages perform extra computations. Since it is a pipeline organization, it is not possible to stop the computations at any time. In [21], these extra cycles are treated as waste cycles. However, it is possible to perform useful computation without complicating the circuit. Recall that $C = A \cdot B \cdot 2^{-m} \pmod{p}$ where m is the number of bits in the modulus p. If we continue the computations in these extra pipeline cycles, we calculate $C = A \cdot B \cdot 2^{-n} \pmod{p}$ where $n > m$ is the smallest integer multiple of the number of PUs in the pipeline organization. It is always easy to rearrange the Montgomery settings according to this new Montgomery exponent, namely $R = 2^n$, or $R = x^n$ for the field $GF(2^m)$ case.

The total computation time, CC (clock cycles), is slightly different from the one in [21] and is given as

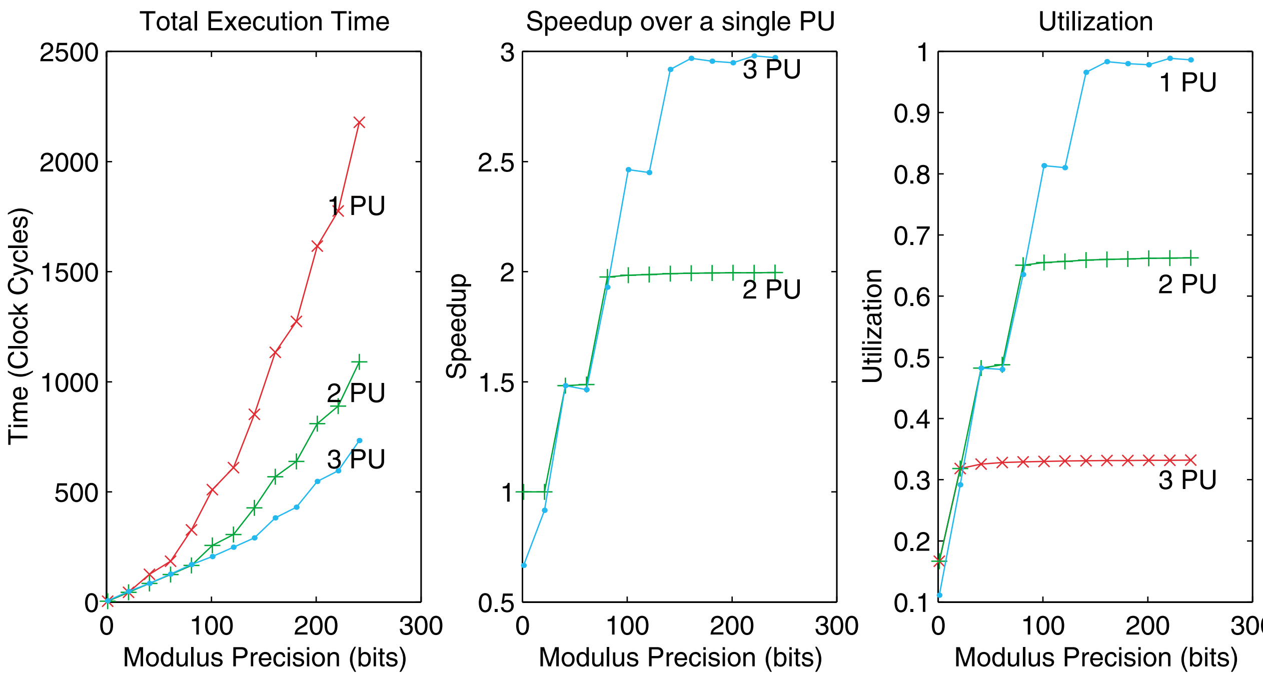
$$CC = \begin{cases} (\lceil \frac{m+1}{k} \rceil - 1)2k + e + 1 + 2(k-1) & \text{if } (e+1) < 2k \text{ ,} \\ (\lceil \frac{m+1}{k} \rceil)(e+1) + 2(k-1) & \text{otherwise ,} \end{cases}$$

where k is the number of PUs in the pipeline. Notice that the first line of the formula gives the execution time in clock cycles when there are sufficiently many PUs while the second line corresponds to the case when there are stalls in the pipeline. At certain clock cycles some of the PUs become idle, and this affects the utilization of the unit, which can be formulated as

$$U = \frac{\text{Total number of clock cycles per bit of } A \times m}{\text{Total number of clock cycles } \times k} = \frac{(e+1) \cdot m}{CC \cdot k} \text{ .}$$

Figure 4 shows (from left to right) the total execution time CC, the speedup introduced by use of more units over a single unit, and the hardware utilization U for a range of precision. We prefer to select the wordsize as $w = 32$ in order to provide a realistic example, considering that most multi-purpose microprocessors have 32-bit datapaths.

Figure 4: The performance of multiple units with $w = 32$.



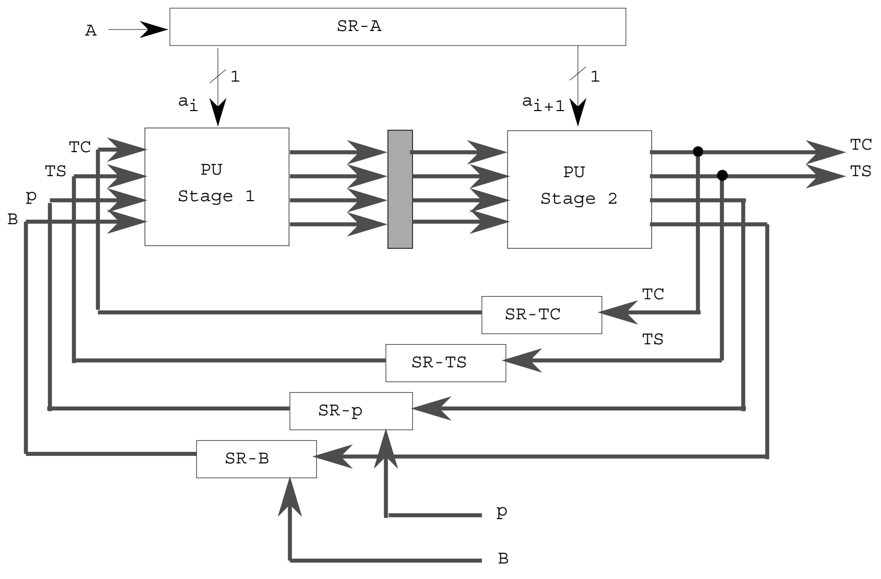
6 Scalable Architecture

An example of pipeline organization with 2 PUs is shown in Figure 5. An important aspect of this organization is the register file design. The bits of multiplier a_i are given serially to the PUs, and are not used again in later stages and can be discarded immediately. Therefore, a simple shift register would be sufficient for the multiplier. The registers for the modulus p and multiplicand B can also be shift registers. When there is no pipeline stall, the latches between PUs forward the modulus and multiplicand to next PU in the pipeline. However, if pipeline stalls occur, the modulus and multiplicand words generated at the end of the pipeline enter the $SR-p$ and $SR-B$ registers. The length of these shift registers are of crucial importance and determined by the number of pipeline stages (k) and the number of words (e) in the modulus. By considering that $SR-p$ and $SR-B$ values require one extra register to store the all-zero word needed for the last clock cycle in every stage (recall that $p^{(e)} = B^{(e)} = 0$) the length of these registers can be given as

$$L_1 = \begin{cases} e+1-2\cdot(k-1) & \text{if } (e+1) < 2k \ , \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

The width of the shift registers is equal to w, the wordsize. Once the partial sum (TC,TS) is generated, it is transmitted to the next stage without any delay. However, we need two shift registers, $SR-TC$ and $SR-TS$, to hold the partial sums from the last stage until the job in the first stage is completed. The length (L_2) of the registers TC and TS is equal to L_1.

Figure 5: Pipeline organization with 2 PUs.



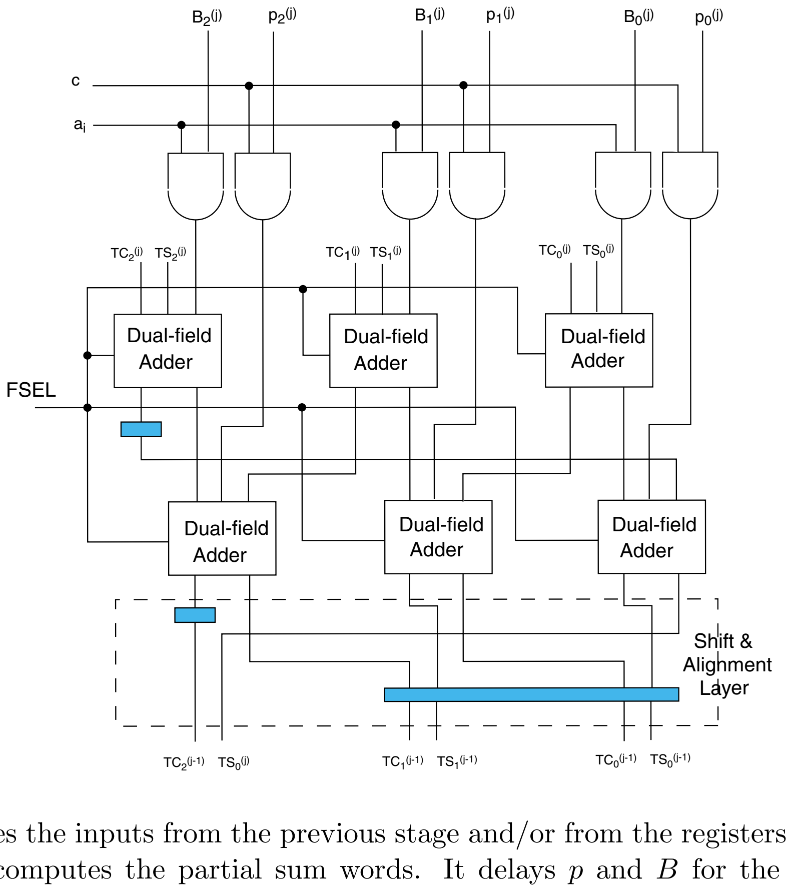
We observe that only at most one word of each operand is used in every clock cycle. This makes different design options possible. Since we intend to design a fully scalable architecture, we need to avoid restrictions on the operand size or deterioration of the performance. Also we assume that no prior knowledge is available about the prospective range of the operand precision. Since the length of the shift registers can vary with the precision, designing full-precision registers within the multiplier might not be a good idea. Instead, one can limit the length of these registers within the chip and use memory for the excessive words. If this method is adopted, the length of the registers no longer would depend on the precision and/or the number of stages. The words needed earlier are brought from memory to the registers first, and the successive groups of words are transferred during the computation. If the memory transfer rate is not sufficient, however, pipeline might stall.

The registers for TC, TS, B, and p must have loading capability which can complicate the local control circuit by introducing several multiplexers (MUX). The delay imposed by these MUXes will not create a critical path in the final circuit. The global control block was not mentioned since its function can be inferred from the dependency graph and the algorithms.

6.1 Processing Unit

The processing unit (PU) consists of two layers of adder blocks, which we call *dual-field adders*. A dual-field adder is basically a full adder which is capable of performing addition both with carry and without carry. Addition with carry corresponds to the addition operation in the field $GF(p)$ while addition without carry corresponds to the addition operation in the field $GF(2^m)$. We give the details about the dual-field adder in the next subsection. The block diagram of a processing unit (PU) for $w = 3$ is shown in Figure 6.

Figure 6: Processing Unit (PU) with $w = 3$.


$B_2^{(j)}$
$p_2^{(j)}$
$B_1^{(j)}$
$p_1^{(j)}$
$B_0^{(j)}$
$p_0^{(j)}$
c
a_i
$TC_2^{(j)}$ $TS_2^{(j)}$
$TC_1^{(j)}$ $TS_1^{(j)}$
$TC_0^{(j)}$ $TS_0^{(j)}$
Dual-field Adder
FSEL
Shift & Alignment Layer
$TC_2^{(j-1)}$ $TS_0^{(j)}$
$TC_1^{(j-1)}$ $TS_1^{(j-1)}$
$TC_0^{(j-1)}$ $TS_0^{(j-1)}$

The unit receives the inputs from the previous stage and/or from the registers $SR-A$, $SR-B$ and $SR-p$, and computes the partial sum words. It delays p and B for the first cycle, then, it transmits them to the next stage along with the first partial sum word (which is ready at the second clock cycle) if there is an available PU. The data path for partial sum $T = (TC, TS)$ (which is expressed in the redundant Carry-Save form) is $2w$ bits long while it is w bits long for p and B and 1 bit long for a_i. At the first cycle, the decision to add the modulus to the partial sum is determined, and this information is kept during the following e clock cycles. The computations in a PU for $e = 5$ are illustrated in Table 1 for both types of fields $GF(p)$ and $GF(2^m)$.

Table 1: Inputs and outputs of the ith pipeline stage with $w = 3$ and $e = 5$ for both types of fields $GF(p)$ (top) and $GF(2^m)$ (bottom).

Cycle No	Inputs	Outputs
1	$TC^{(0)}, TS^{(0)}, a_i, B^{(0)}, P^{(0)}$	$(0, TS_0^{(0)})$; $(0, 0)$; $(0, 0)$
2	$TC^{(1)}, TS^{(1)}, a_i, B^{(1)}, P^{(1)}$	$(TC_2^{(0)}, TS_0^{(1)})$; $(TS_2^{(0)}, TC_1^{(0)})$; $(TS_1^{(0)}, TC_0^{(0)})$
3	$TC^{(2)}, TS^{(2)}, a_i, B^{(2)}, P^{(2)}$	$(TC_2^{(1)}, TS_0^{(2)})$; $(TS_2^{(1)}, TC_1^{(1)})$; $(TS_1^{(1)}, TC_0^{(1)})$
4	$TC^{(3)}, TS^{(3)}, a_i, B^{(3)}, P^{(3)}$	$(TC_2^{(2)}, TS_0^{(3)})$; $(TS_2^{(2)}, TC_1^{(2)})$; $(TS_1^{(2)}, TC_0^{(2)})$
5	$TC^{(4)}, TS^{(4)}, a_i, B^{(4)}, P^{(4)}$	$(TC_2^{(3)}, TS_0^{(4)})$; $(TS_2^{(3)}, TC_1^{(3)})$; $(TS_1^{(3)}, TC_0^{(3)})$
6	0, 0, 0, 0, 0	$(TC_2^{(4)}, 0)$; $(TS_2^{(4)}, TC_1^{(4)})$; $(TS_1^{(4)}, TC_0^{(4)})$
1	$TC^{(0)}, TS^{(0)}, a_i, B^{(0)}, P^{(0)}$	$(0, TS_0^{(0)})$; $(0, 0)$; $(0, 0)$
2	$TC^{(1)}, TS^{(1)}, a_i, B^{(1)}, P^{(1)}$	$(0, TS_0^{(1)})$; $(TS_2^{(0)}, 0)$; $(TS_1^{(0)}, 0)$
3	$TC^{(2)}, TS^{(2)}, a_i, B^{(2)}, P^{(2)}$	$(0, TS_0^{(2)})$; $(TS_2^{(1)}, 0)$; $(TS_1^{(1)}, 0)$
4	$TC^{(3)}, TS^{(3)}, a_i, B^{(3)}, P^{(3)}$	$(0, TS_0^{(3)})$; $(TS_2^{(2)}, 0)$; $(TS_1^{(2)}, 0)$
5	$TC^{(4)}, TS^{(4)}, a_i, B^{(4)}, P^{(4)}$	$(0, TS_0^{(4)})$; $(TS_2^{(3)}, 0)$; $(TS_1^{(3)}, 0)$
6	0, 0, 0, 0, 0	$(0, 0)$; $(TS_2^{(4)}, 0)$; $(TS_1^{(4)}, 0)$

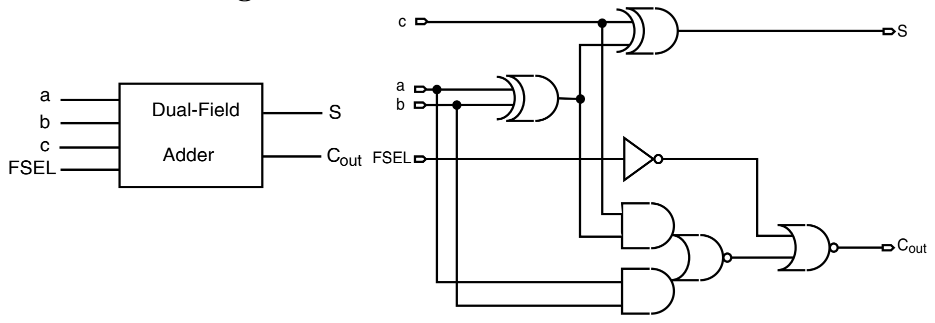
Notice that partial sum words in $GF(2^m)$ case are also in the redundant Carry-Save form. However, one of the components of the Carry-Save representation is always zero and the actual value of the result is the modulo-2 sum of the two. Since consecutive operations are all additions and the Carry-Save form is already aligned by the shift and alignment layer, this does not lead to any problem. We need to recall, however, that one extra addition is necessary at the end of the multiplication process. In the next section, we introduce a multi-purpose word adder/subtractor module which performs this final addition at the cost of an extra clock cycle.

6.2 Dual-Field Adder

Dual-field adder (DFA) shown in Figure 7a, as mentioned before, is basically a full-adder equipped with the capability of doing bit addition both with and without carry. It has an input called $FSEL$ (field select) that enables this functionality. When $FSEL = 1$, the DFA performs the bit-wise addition with carry which enables the multiplier to do arithmetic in the field $GF(p)$. When $FSEL = 0$, on the other hand, the output $Cout$ is forced to 0 regardless of the values of the inputs. The output S produces the result of bitwise modulo-2 addition of three input values. At most 2 of 3 input values of dual-field adder can have nonzero values while in the $GF(2^m)$ mode.

An important aspect of designing the dual-field adder is not to increase the critical path of the circuit which can have an effect on the clock speed which would be against our design goal. However, a small amount of extra area can be sacrificed. We show in the following section that this extra area is very insignificant. Figure 7b shows the actual circuit synthesized by Mentor Graphics tools using the $1.2\mu m$ CMOS technology.

Figure 7: The dual-field adder circuit.



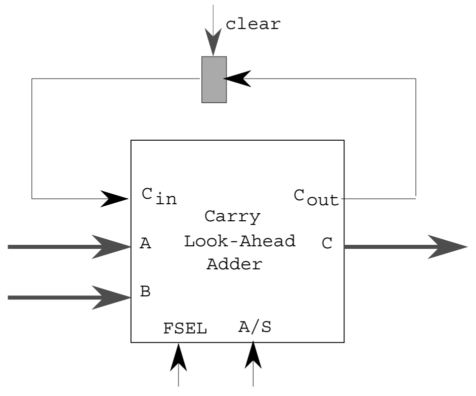
(a) Dual-Field Adder (b) Synthesized circuit by Mentor

In the circuit, the two XOR gates are dominant in terms of both area and propagation time. As in the standard full-adder circuit, the dual-field adder has two XOR gates connected serially. Thus, propagation time of the dual-field adder is not larger than that of full adder. Their areas differ slightly, but this does not cause a major change in the whole circuit.

7 Multi-purpose Word Adder/Subtractor

The proposed Montgomery multiplier generates results in the redundant Carry-Save form, hence we need to perform an extra addition operation at the end of the calculation to obtain the nonredundant form of the result. Therefore, a field adder circuit that operates in both $GF(p)$ and $GF(2^m)$ is necessary. A full-precision adder would increase the critical path delay and the area, and would also be hard to scale. A word adder of the type given in Figure 8 would be suitable for our implementation since the multiplier generates only one word at each clock cycle in the last stage of pipeline, thus we need to perform one word addition at a time.

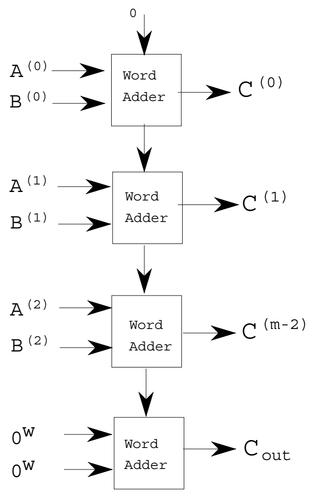
Figure 8: The word adder for field addition.



The word adder has two control inputs $FSEL$ and A/S, which enable to select the field ($GF(p)$ or $GF(2^k)$) and to choose between the addition and subtraction when in $GF(p)$ mode, respectively. The adder propagates the carry bit to the next word additions while working in $GF(p)$ mode (i.e.,

$FSEL = 1$). Thus, the carry from a word addition operation is delayed using a latch and fed back into the C_{in} input of the adder for the next word addition at the next clock cycle. In the $GF(2^m)$ mode, the module performs only bitwise modulo-2 addition of two input words and the A/S input is ineffective. An addition operation of two e-word long numbers takes $e + 1$ clock cycles. The last cycle generates the carry and prepares the circuit for another operation by zeroing the output of latch. Figure 9 shows an example of addition operation with operands of 3 words.

Figure 9: An example of multiprecision addition operation with $e = 3$.



We added subtraction functionality in the field $GF(p)$ to the word adder because the result might be larger than the modulus, and hence one final subtraction operation is necessary as shown in Step 23 of the algorithm in Section 4.1. We do not need this reduction in the $GF(2^m)$ case. The final subtraction operation takes place only if the result is larger than the modulus. Thus, a comparison operation, which can also be performed utilizing the multi-purpose word adder/subtractor, is required. However, the control circuitry to perform this conditional subtraction might be complicated, therefore, it might be placed outside of the Montgomery multiplier unit.

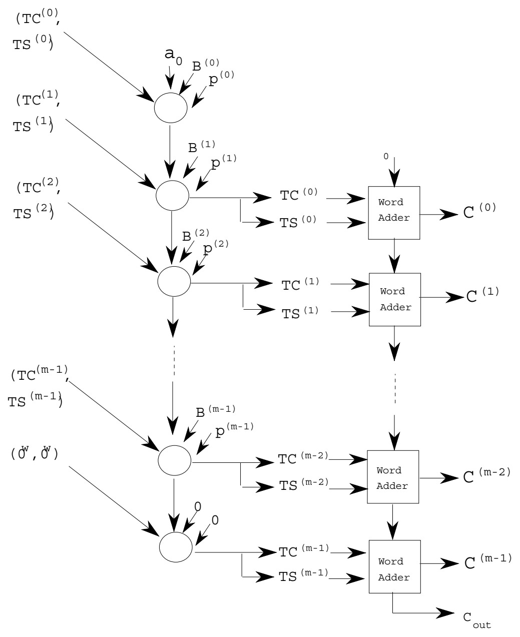
Another reason to include a multi-purpose word adder unit in the multiplier circuit is the fact that the field addition operation is also needed in many cryptographic applications. For example, in elliptic curve cryptosystems, the field addition and multiplication operations are performed successively, hence having the multiplier and adder in the same hardware unit will decrease the communication overhead. A word adder that has these properties is synthesized using the Mentor Graphics tools and the time and space requirements are obtained, which are given in Table 2.

Table 2: Time and area costs of a multi-purpose word adder for $w = 16, 32, 64$.

bitsize	Propagation Time (ns)	Area (in NAND gates)
16	6.87	254
32	9.22	534
64	12.55	1128

Finally, Figure 10 illustrates what happens in last stage of the pipeline. A pair of redundant words $(TC_j^{(i)}, TS_j^{(i)})$ are generated each cycle for e clock cycles. The word adder can be used to add these pairs in order to obtain the result words $C^{(i)}$. Note that only one extra cycle is needed to convert the result from the Carry-Save form to the nonredundant form.

Figure 10: Converting the result from the Carry-Save form to the nonredundant form in the last stage of the pipeline.



8 Design Considerations

In [21], an analysis of the are and time tradeoffs is given for the scalable multiplier. The architecture allows designs with different word lengths and different pipeline organizations for varying values of operand precision. In addition, the area can be treated as a design constraint. Thus, one can adjust the design according to the given area, and choose appropriate values for the word length and the number of pipeline stages, in accordance. We give a similar analysis for the scalable and unified architecture. We are targeting two different classes of ranges for operand precision:

- *High precision range* which includes 512, 768 and 1024, is intended for applications requiring the exponentiation operation.
- *Moderate precision range* which includes 160, 192, 224, and 256, is typical for elliptic curve cryptography.

The propagation delay of the PU is independent of the wordsize w when w is relatively small, and thus all comparisons among different designs can be made under the assumption that the clock cycle is the same for all cases. The area consumed by the registers for the partial sum, the operands, and modulus is also the same for all designs, and we are not treating them as parts of the multiplier module.

The proposed scheme yields the worst performance for the case $w = m$ in the high precision range, since some extra cycles are introduced by the PU in order to allow word-serial computation, when compared to other full-precision conventional designs. On the other hand, using many pipeline stages with small wordsize values brings about no advantage after a certain point. Therefore, the performance evaluation reduces into finding an optimum organization for the circuit.

In order to determine the optimum selection for our organization, we obtain implementation results by synthesizing the circuit with Mentor Graphics tools using $1.2\mu m$ CMOS technology. The cell area for a given word size w is obtained as

$$A_{cell}(w) = 48.5w \tag{5}$$

units, and is slightly different from the one found in [21], where the multiplication factor in the formula is the area cost provided by the synthesis tool for a single bit slice. Note that a 2-input NAND gate takes up 0.94 units of area. In the pipelined organization, the area of the inter-stage latches is important, which was measured as

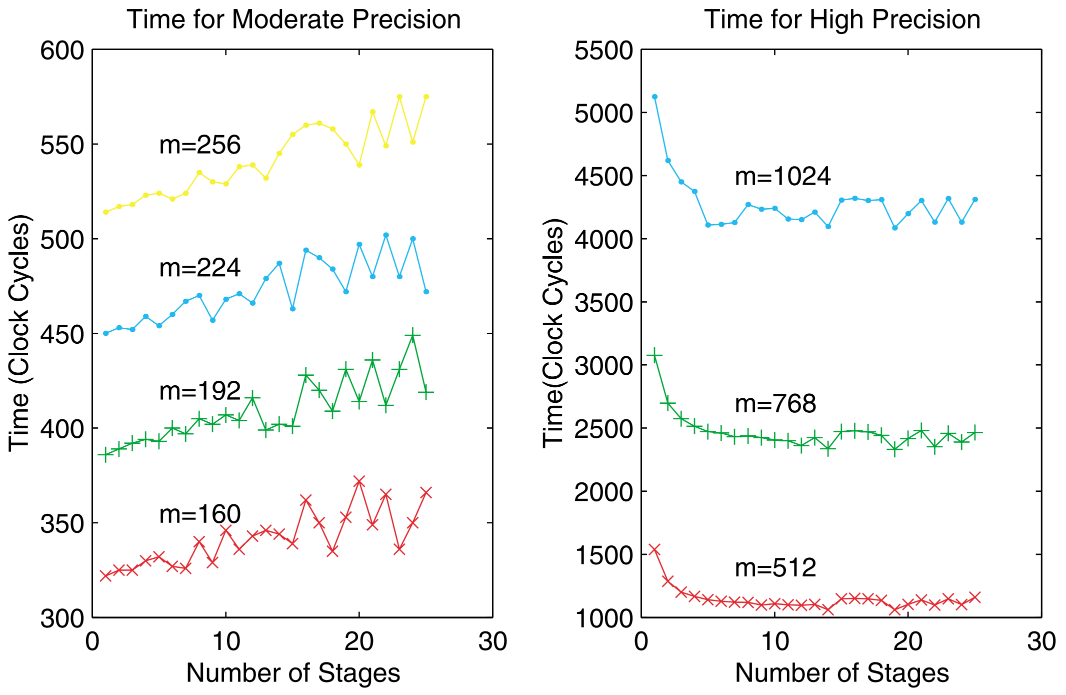
$$A_{latch}(w) = 8.32w \tag{6}$$

units. Thus, the area of a pipeline with k processing elements is given as

$$A_{pipe}(k, w) = (k - 1)A_{latch}(w) + kA_{cell}(w) = 56,82kw - 8.32w \tag{7}$$

units. For a given area, we are able to evaluate different organizations and select the most suitable one for our application. The graphs given in Figure 11 allow to make such evaluations for a fixed area of 15,000 gates.

Figure 11: Time efficiency for different configurations with a fixed area of 15,000 gates.



For both moderate and high precision ranges, the number of stages between 5 and 10 are likely to give the best performance. For the high precision cases, fewer than 5 stages yields very poor performance since the fixed area becomes insufficient for large wordsizes and the performance degradation due to pipeline stalls becomes a major problem. The small number of stages with very long word sizes seem to provide a reasonable performance in the moderate range, however, because of the incompatibility issues about using very long word sizes and inefficiency when the precision increases, using fewer than 5 stages is not advised. We avoid using many stages for two reasons:

- high utilization of the PUs will be possible only for very high precision, and
- the execution time may have undesirable oscillations.

The behavior mentioned in the latter category is the result of the facts that

- extra stages at the end of the computations, and
- there is not a good match between the number of words e and the number of stages k, causing a underutilization of stages in the pipeline.

From the synthesis tool we obtained a minimum clock cycle time of 11 nanoseconds, which allows to use a clock frequency of up to 90MHz with $1.2\mu m$ CMOS. Using the CMOS technology with smaller feature size, we can attain much faster clock speeds. It is very important to know how fast this hardware organization really is when comparing it to a software implementation. The answer to this would determine whether it is worth to design a hardware module. In general, it is difficult to compare hardware and software implementations. In order to obtain realistic comparisons, a processor which uses similar clock cycles and technology must be chosen. We selected an ARM microprocessor [5] with 80 MHz clock which has a very simple pipeline. We compare the $GF(p)$ multiplication timing on this processor against that of our hardware module. We use the same clock frequency 80 MHz for the module of the pipeline organization with $w = 32$ and $k = 7$ for the hardware module. On the other hand, the Montgomery multiplication algorithm is written in the ARM assembly language by using all known optimization techniques [8, 10]. Table 3 shows the multiplication timings and the speedup.

Table 3: The execution times of hardware and software implementations of the $GF(p)$ multiplication.

precision	Hardware (μs) (80 MHz, $w = 32$, $k = 7$)	Software (μs) (on ARM with Assembly)	speedup
160	4.1	18.3	4.46
192	5.0	25.1	5.02
224	5.9	33.2	5.63
256	6.6	42.3	6.41
1024	61	570	9.34

9 Conclusion

Using the design methodology proposed in [21], we obtained a scalable field multiplier for $GF(p)$ and $GF(2^m)$ in unified hardware module. The methodology can also be used to design separate modules for $GF(p)$ and $GF(2^m)$ which are fast, scalable and area-efficient. The fundamental contribution of this research is to show that it is possible to design a dual-field arithmetic unit without compromising scalability, the time performance and area efficiency. We also presented a dual-field addition module which is suitable for the pipeline organization of the multiplier. The adder is scalable and capable of performing addition in both types of fields. Our analysis shows that a pipeline consisting of several stages is adequate and more efficient than a single unit processing very long words. Working with relatively short words diminishes data paths in the final circuit, reducing the required bandwidth.

The proposed multiplier was synthesized using the Mentor tools, and a circuit capable of working with clock frequencies up to 90 MHz is obtained. Except for the upper limit on the precision which is dictated only by the availability of memory to store the operands and internal results, the module is capable of performing infinite-precision Montgomery multiplication in $GF(2^m)$ and $GF(p)$.

References

[1] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[2] A. Bernal and A. Guyot. Design of a modular multiplier based on Montgomery's algorithm. In *13th Conference on Design of Circuits and Integrated Systems*, pages 680–685, Madrid, Spain, November 17–20 1998.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[4] S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, June 1993.

[5] Steve Furber. *ARM System Architecture*. Addison-Wesley, Reading, MA, 1997.

[6] B. S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, August 1995.

[7] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[8] Ç. K. Koç. High-Speed RSA Implementation. Technical Report TR 201, RSA Laboratories, 73 pages, November 1994.

[9] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

[10] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[11] P. Kornerup. High-radix modular multiplication for cryptosystems. In E. Swartzlander, Jr., M. J. Irwin, and G. Jullien, editors, *Proceedings, 11th Symposium on Computer Arithmetic*, pages 277–283, Windsor, Ontario, June 29 – July 2 1993. IEEE Computer Society Press, Los Alamitos, CA.

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

[13] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[14] D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, June 1996.

[15] National Institute for Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-2, January 2000.

[16] J.-H. Oh and S.-J. Moon. Modular multiplication method. *IEE Proceedings: Computers and Digital Techniques*, 145(4):317–318, July 1998.

[17] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In S. Knowles and W. H. McAllister, editors, *Proceedings, 12th Symposium on Computer Arithmetic*, pages 193–199, Bath, England, July 19–21 1995. IEEE Computer Society Press, Los Alamitos, CA.

[18] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.

[19] A. Royo, J. Moran, and J. C. Lopez. Design and implementation of a coprocessor for cryptography applications. In *European Design and Test Conference*, pages 213–217, Paris, France, March 17-20 1997.

[20] E. Savaş and Ç. K. Koç. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, to appear, 2000.

[21] A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 94–108. Springer, Berlin, Germany, 1999.

[22] C. D. Walter. Space/Time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2):139–141, February 1997.