Scalable and Unified Hardware to Compute Montgomery Inverse in GF(p) and GF(2^n)

Adnan Abdul-Aziz Gutub[1], Alexandre F. Tenca, Erkay Savaş[2], and Çetin K. Koç

Department of Electrical & Computer Engineering
Oregon State University, Corvallis, Oregon 97331, USA
{gutub,tenca,savas,koc}@ece.orst.edu

Abstract. Computing the inverse of a number in finite fields GF(p) or GF(2^n) is equally important for cryptographic applications. This paper proposes a novel scalable and unified architecture for a Montgomery inverse hardware that operates in both GF(p) and GF(2^n) fields. We adjust and modify a GF(2^n) Montgomery inverse algorithm to accommodate multi-bit shifting hardware, making it very similar to a previously proposed GF(p) algorithm. The architecture is intended to be scalable, which allows the hardware to compute the inverse of long precision numbers in a repetitive way. After implementing this unified design it was compared with other designs. The unified hardware was found to be eight times smaller than another reconfigurable design, with comparable performance. Even though the unified design consumes slightly more area and it is slightly slower than the scalable inverter implementations for GF(p) only, it is a practical solution whenever arithmetic in the two finite fields is needed.

1 Introduction

The modular inversion is an essential arithmetic operation for many cryptographic applications, such as Diffe-Hellman key exchange algorithm, decipherment operation of RSA algorithm, elliptic curve cryptography (ECC) [1,5], and the Digital Signature Standard as well as the Elliptic Curve (EC) Digital Signature algorithm [4,5]. The arithmetic performed in cryptographic applications consists mainly in modular computations of addition, subtraction, multiplication, and inversion. Although inversion is not as performance critical as all the others, it is the most time consuming arithmetic operation [1,2,8-10,12,13]. Therefore, most of the practical implementations try to avoid the use of inversion as much as possible. However, it is not possible to avoid it completely [1,2,5], what motivates the implementation of inversion as a hardware module in order to gain speed. In addition to that, hardware implementations provide an increased level of security for cryptographic systems, as discussed in [15].

Cryptographic inverse calculations are normally defined over either prime or binary extension fields [5], more specifically *Galois Fields GF(p)* or *GF(2^n)*. All

[1] Now with King Fahd University, Dhahran, Saudi Arabia, gutub@kfupm.edu.sa

[2] Now with Sabanci University, Istanbul, Turkey, erkays@sabanciuniv.edu

available application-specific integrated circuit (ASIC) implementations for inversion computation [8-10,12,13] are modeled strictly for one finite field, either GF(p) or GF(2^n). If the hardware at hand is for GF(2^n) calculations, such as [9,10,12,13], and the application this time needs GF(p) computation, a completely different hardware is required [5]. It is inefficient to have two hardware designs (one for GF(p) and another for GF(2^n)) when only one is needed each time. This issue motivated the search for a single unified hardware architecture used to compute inversion in either finite field GF(p) or GF(2^n), similar, in principle, to the multiplier idea proposed in [4].

Cryptography is heavily based on modular multiplication [4,5], which involves division by the modulus in its computations. Division, however, is a very expensive operation [6]. P. Montgomery proposed an algorithm to perform modular multiplication [7] that replaces the usual complex division with divisions by two, which is easily performed in the binary representation of numbers. The cost behind using Montgomery's method is paid in some extra computations to convert the numbers into Montgomery domain and vice-versa [7]. Once the numbers are transformed into Montgomery domain, all operations (addition, subtraction, multiplication, and inversion) are performed in this domain. The result is then converted back to the original integer values. Few methods were aimed to compute the inverse in the Montgomery domain [1-3] and are named *Montgomery modular inverse algorithms* [1].

The GF(p) Montgomery inverse (MonInv) algorithm [18] is an efficient method for doing inversion with an odd modulus. The algorithm is particularly suitable for implementation on application specific integrated circuits (ASICs). For GF(2^n) inversion, the original inverse procedure (presented in [17]) has been extended to the finite field GF(2^n) in [16]. It replaces the modulus (p) by an irreducible polynomial ($p(x)$), and adjusts the algorithm according to the properties of polynomials. We implemented the inversion algorithms in hardware based on the observation that the Montgomery inverse algorithm for both fields GF(p) and GF(2^n) can be very similar. We show that a unified architecture computing the Montgomery inversion in the fields GF(p) and GF(2^n) is designed at a price only slightly higher than the one for only the field GF(p), providing major savings when both types of inverters are required.

A scalable Montgomery inverter design methodology for GF(p) was introduced in [18]. This methodology allows the use of a fixed-area Montgomery inverter ASIC design to perform the inversion of unlimited precision operands. The design tradeoffs for best performance in a limited chip area were also analyzed in [18]. We use the design approach as in [14,18] to obtain a scalable hardware module. Furthermore, the scalable inverter described in this paper is capable of performing inversion in both finite fields GF(p) and GF(2^n) and is for this reason called a *scalable and unified Montgomery inverter*.

There are two main contributions of this paper. First, we show that a unified architecture for inversion can be easily designed without compromising scalability and without significantly affecting delay and area. Second, we investigate the effect of word length (w) and the actual number of bits (n) on the hardware area, based on actual implementation results obtained by synthesis tools. We start with a brief explanation of scalability in Section 2. In Section 3, we propose the GF(2^n) extended Montgomery inverse procedure that has several features suitable for an efficient hardware implementation. The unified architecture and its operation in both types of finite fields, GF(p) and GF(2^n), are described in Section 4. Section 5 presents the area/time tradeoffs and appropriate choices for the word length of the scalable module. Finally, a summary and conclusions are presented in Section 6.

2 Scalable Architecture

Hardware architectures are generally designed for an exact number of operand bits. If this number of bits needs to be increased, even by one bit, the complete hardware needs to be replaced. In addition to that, if the design is implemented for a large number of bits, the hardware will be huge and usually slow. These issues motivated the search for the scalable inversion hardware proposed in [14].

The scalable architecture [14] solves the previous problems with the following three hardware features. First, the design's longest path should be short and independent of the operands' length. Second, it is designed in such a way that it fits in restricted spaces (flexible area). Finally, it can handle the computation of numbers in a repetitive way, up to a certain limit that is usually imposed by the size of the memory in the design. If the amount of data exceeds the memory capacity, the memory unit is replaced while the scalable computing unit may remain the same. Therefore, the scalable hardware design is built of two main parts, a memory unit and a computing unit. The memory unit is not scalable because it has a limited storage that imposes an upper bound on the number of bits that can be handled by the hardware (n_{max}). The computing unit read/write the data bits using another word size of w bits, normally much smaller than n_{max}. The computing unit is completely scalable. It is designed to handle w bits every clock cycle. The computing unit does not know the total number of bits that the memory is holding. It computes until the actual number of operand bits (n) is processed.

3 Montgomery Inverse Procedures for *GF(p)* and *GF(2^n)*

In order to design a unified Montgomery inverse architecture, the GF(p) and GF(2^n) algorithms need to be very similar and this way consume the least amount of extra hardware. Extending the GF(p) Montgomery inverse algorithm to GF(2^n) is practical due to the removal of carry propagation required in GF(p) and simple adjustments of test conditions. In other words, the GF(2^n) algorithm is like a simplification of the GF(p) algorithm. The converse (modifying GF(2^n) algorithms for GF(p)), on the other hand, is very difficult [4,5,16].

The scalable GF(p) Montgomery inverse (*MonInv*) procedure suitable for this work consists in two phases: the almost Montgomery inverse (*AlmMonInv)* and the correction phase (*CorPh*) [18]. The *AlmMonInv* has $a2^m$ as input and produces r and k, where $r = a^{-1}2^{k-m} \bmod p$, $2^{n-1} \leq p < 2^n$ and $n < k < 2n$. The factor 2^m (of the AlmMonInv input $a2^m$) is related to Montgomery arithmetic [4,5,16]. The only restriction on the value of m is that it should not be less than the number of bits (n), i.e., $m \geq n$, as discussed in [1]. The *CorPh* takes r and k to generate the Montgomery inverse $a^{-1}2^m \bmod p$. Both GF(p) AlmMonInv and CorPh algorithms were mapped to hardware features and further modified for multi-bit shifting, a concept discussed in [18], which resulted in an efficient implementation of the GF(p) Montgomery inverse. The GF(p) multi-bit shifting AlmMonInv and CorPh hardware algorithms (HW-Alg1 and HW-Alg2, respectively), are outlined in Figure 1.

GF(p) Multi-Bit Shifting AlmMonInv HW Algorithm (HW-Alg1)

```
Registers: u, v, r, s, x, y, z, and $p$ (all registers hold $n_{max}$ bits)
Input:  $a2^m \in [1, p-1]$; Where $p$ = modulus, and $m \geq n$ ($2^{n-1} \leq p < 2^n$)
Output:   $result \in [1, p-1]$ & $k$; Where $result = a^{-1}2^{k-m} mod\ p$ & $n < k < 2n$
1.   u = $p$; v = $a2^m$; r = 0; s = 1; x = 0; y = 0; z = 0; $k$ = 0
2.   if ($u_2u_1u_0$=000) then {u=ShiftR(u,3);s=ShiftL(s,3);$k$=$k$+3}; goto 8
2.1. if ($u_2u_1u_0$=100) then {u=ShiftR(u,2);s=ShiftL(s,2);$k$=$k$+2}; goto 8
2.2. if ($u_2u_1u_0$=110) then {u=ShiftR(u,1);s=ShiftL(s,1)}; goto 7
3.   if ($v_2v_1v_0$=000) then {v=ShiftR(v,3);r=ShiftL(r,3);$k$=$k$+3}; goto 8
3.1. if ($v_2v_1v_0$=100) then {v=ShiftR(v,2);r=ShiftL(r,2);$k$=$k$+2}; goto 8
3.2. if ($v_2v_1v_0$=110) then {v=ShiftR(v,1);r=ShiftL(r,1)}; goto 8
4.   x = Subtract(u, v); y = Subtract(v, u); z = Add(r, s)
5.   if ($x_{borrow}$=0) then {u=ShiftR(x,1); r=z; s=ShiftL(s,1)}; goto 7
6.   s = z; v = ShiftR(y,1); r = ShiftL(r,1)
7.   $k = k + 1$
8.   if (v $\neq$ 0) goto step 2
9.   x = Subtract($p$, r); y = Subtract($2p$, r)
10.  if ($x_{borrow}$ = 0) then {result=x}; else{result = y}
```

GF(p) Multi-Bit Shifting CorPh HW Algorithm (HW-Alg2)

```
Registers: $r$, $u$, $v$, x, y, z, and $p$ (all registers hold $n_{max}$ bits)
Input:     $r, p, n, k$; Where $r$ ($r=a^{-1}2^{k-m} mod\ p$)&$k$ from HW-Alg1
Output:  result; Where result = $a^{-1}2^m$ (mod $p$).
11.   $j = 2m-k$; x = 0; y = 0; z = 0
12.   $v = 2p$; $u = 3p$
13.   While $j > 0$
14.       if $j = 1$ then {$r$ = ShiftL($r$,1); $j=j-1$}
15.       else {$r$ = ShiftL($r$,2); $j=j-2$}
16.       x=Subtract($r$,$p$); y=Subtract($r$,$v$); z=Subtract($r$,$u$)
17.       if ($z_{borrow}$ = 0) then  {$r$ = z}
18.       else if ($y_{borrow}$ = 0) then {$r$ = y}
19.       else if ($x_{borrow}$ = 0) then  {$r$ = x}
20.   result = $r$
```

Fig. 1. Montgomery inverse hardware algorithm for GF(p)

Differently from what normally happens in a full-precision hardware design, the scalable hardware, as in [4,14,18], has multi-precision operators for shifting, addition, subtraction, and comparison. Observe the AlmMonInv algorithm in Figure 1, for example, the scalable subtraction (step 4) is also used for comparison ($u > v$), which is performed on a word-by-word basis (w-bit words) until all the actual data words (all n bits) are processed. Then, borrow-out bit of the most-significant word is used to decide on the result. Also, depending on the subtraction's completion, variable r or s has to be shifted. All variables, u, v, r and s, need to remain as is until the subtraction process is complete, and the borrow-out bit appears. For this reason, eight registers are required, as shown in Figure 1.

3.1 Representation and Manipulation of Elements in GF(2^n)

The inversion algorithm for GF(2^n) used in this work was presented in [16]. Although prime and binary extension fields, GF(p) and GF(2^n), have different properties, the elements of either field are represented using similar data structures. The elements of the field GF(2^n) can be represented in several different ways [5]. The polynomial representation, however, is a useful and appropriate form to the unified implementation, as used for the unified multiplier in [4]. According to the GF(2^n) polynomial representation, an element $a(x) \in$ GF(2^n) is a polynomial of length n, i.e., of degree less than or equal to $n-1$, written as $a(x)=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+ \ldots +a_2x^2+a_1x+a_0$, where $a_i \in$ GF(2). These coefficients a_i are represented as bits in the computer and the element $a(x)$ is represented as a bit vector $a = (a_{n-1}\ a_{n-2} \ldots a_2\ a_1\ a_0)$.

The addition/subtraction of two elements $a(x)$ and $b(x)$ in GF(2^n) is performed by adding/subtracting the polynomials $a(x)$ and $b(x)$, where the coefficients are added/subtracted in the field GF(2). As a consequence, both addition and subtraction operations are exactly the same and equivalent to bit-wise XOR operations on the bit-vectors a and b ($a_i \oplus b_i$). In order to compute the inverse of element $a(x)$ in GF(2^n), we need an irreducible polynomial of degree n. Let the irreducible polynomial be $p(x)= x^n+p_{n-1}x^{n-1}+p_{n-2}x^{n-2}+ \ldots +p_2x^2+p_1x+p_0$. Whenever the degree of a polynomial obtained in intermediate inversion calculations equals n, the polynomial is reduced (XORed) by $p(x)$. For example, if $\|r(x)\| = \|p(x)\|$ (degree of $r(x)$ equals degree of $p(x)$) then r is replaced by $p \oplus r$. Note that in some cases $\|r(x)\| = \|p(x)\|$ while $r < p$. These cases restrict the comparison of r to 2^n only (x^n not $p(x)$) to indicate if $r(x)$ needs to be reduced by $p(x)$ ($r = p \oplus r$); where 2^n is the binary representation of x^n.

3.2 Montgomery Inverse in GF(2^n)

The GF(2^n) Montgomery inverse of $a(x)x^m \bmod p(x)$ is $a(x)^{-1}x^m \bmod p(x)$ [5]. The Montgomery factor 2^m of GF(p) is replaced by x^m in GF(2^n), which is exactly equal to 2^m in a binary representations [4,5,16], where $m \geq n$. The elements of GF(p) and GF(2^n) are represented using similar binary data structures, a for both GF(p) and GF(2^n) equals ($a_{n-1}\ a_{n-2}\ \ldots\ a_2\ a_1\ a_0$) while $p = (p_{n-1}\ p_{n-2} \ldots p_2\ p_1\ p_0)$ for GF(p) and $p=(1\ p_{n-1}\ p_{n-2} \cdots p_2\ p_1\ p_0)$ for GF(2^n) [5]. Our adjusted binary GF(2^n) Montgomery inverse (MonInv) procedure consists in a GF(2^n) AlmMonInv and a GF(2^n) CorPh routines as outlined in Figure 2.

For more clarification of the GF(2^n) MonInv computation, see the numerical example in Figure 3. It takes as inputs the polynomial $a(x)= x^3+1$, represented into Montgomery domain as $a(x)x^9 \bmod p(x)= x^4+x^2$ ($m=9 \geq n=5$), and $p(x)= x^5+x^2+1$ as the irreducible polynomial. All the data are shown in its binary representation ($a=1001$, $a2^m=10100$, and $p=100101$). The example (Figure 3) follows the convention:

Met condition ➔ *affected registers with their updated values.*

The AlmMonInv routine generates the results $a^{-1}2^{k-m} = 1000$, and $k = (10)_{10}$ (k is a normal decimal counter), which are used by the CorPh to provide the Montgomery inverse result 111 (x^2+x+1 in the polynomial form). The reader is referred to the Appendix for checking the result of this example.

```
GF(2^n) AlmMonInv Algorithm
Input:  $a2^m \in GF(2^n)$ & $p$; ($p$=irreducible polynomial & $m \geq n$)
Output: $result \in [1, p-1]$ & $k$ ($result = a^{-1}2^{k-m} mod\ p$ & $n < k < 2n$)
1.  u = p; v = $a2^m$; r = 0; s = 1; k = 0
2.  While (v > 0)
3.      if $u_0$ = 0 then {u = u/2; s = 2s}
4.      else if $v_0$ = 0 then {v = v/2; r = 2r}
5.      else if u>v then {u = (u⊕v)/2; r = r⊕s; s = 2s}
6.      else {v = (u⊕v)/2; s = r⊕s; r = 2r}
7.      k=k+1
8.  if r ≥ $2^{n+1}$ (||r|| > ||p||) then {result = 2p⊕r}
9.  else if r ≥ $2^n$ (||r|| = ||p||) then {result = p⊕r}
10. else result = r

GF(2^n) CorPh Algorithm
Input:   r, p, m, & cowherd r & k from AlmMonInv
Output:  result, where $result = a^{-1}2^m$ (mod p)
11. j = 2m-k
12. While j > 0
13.     r = 2r
14.     if r ≥ $2^n$ (||r|| = ||p||) then {r = p⊕r}
15.     j = j-1
16. result = r
```

Fig. 2. GF(2^n) Montgomery inverse algorithm in its binary representation

Observe on Figure 2 the several hardware operations applied to compute the MonInv in finite field GF(2^n). For example, the division and multiplication by two are equivalent to one bit shifting the binary representation of polynomials to the right and to the left, respectively. Checking the condition of step 5, if $u>v$, is performed through normal (borrow propagate) subtraction and test of the borrow-out bit. The subtraction result is completely discarded, only the borrow bit is observed. If the borrow bit is zero, then $u(x)$ is greater than $v(x)$. Similarly, the conditions in steps 8, 9, and 14 demand normal subtraction. However, the subtraction this time is used to check $\|r(x)\|$, which requires the availability of x^n (2^n in binary).

3.3 Multi-bit Shifting

A further improvement on the GF(2^n) MonInv algorithm is performed based on a multi-bit shifting method making it similar to the GF(p) algorithm in Figure 1. After comparing different multi-bit shifting distances applied to reduce the number of iterations of the GF(p) MonInv algorithm [18,19], the best maximum distance for multi-bit shifting was found to be three, as clarified in [18,19]. The GF(2^n) inverse algorithm (Figure 2) is mapped to hardware involving multi-bit shifting and making it very similar to the GF(p) algorithm (Figure 1) as shown in Figure 4. Note that x^n is required in the GF(2^n) algorithm as an extra variable that is needless in the GF(p) MonInv algorithm; x^n (2^n) is saved in register y in HW-Alg3 (used in step 9), and in register s in HW-Alg4 (used in step 16.1). These registers (y in HW-Alg3 and s in HW-Alg4) are not changed during the algorithms' execution.

GF(2^n) AlmMonInv Numerical Example

$a = 1001 \in \mathrm{GF}(2^5)$, $p=100101$, $m= 9$, $n=5$

$a2^m \bmod p = 10100 \in \mathrm{GF}(2^6)$ (a in Montgomery domain)

$u = p = 100101,\ v = a2^m = 10100,\ s = 1,\ r = k = 0$

$v_0 = 0 \rightarrow v = 1010,\ r = 0,\ k=1$

$v_0 = 0 \rightarrow v = 101,\ r = 0,\ k=2$

$u > v \rightarrow u = 10000,\ r = 1,\ s = 10,\ k=3$

$u_0 = 0 \rightarrow u = 1000,\ s = 100,\ k=4$

$u_0 = 0 \rightarrow u = 100,\ s = 1000,\ k=5$

$u_0 = 0 \rightarrow u = 10,\ s = 10000,\ k=6$

$u_0 = 0 \rightarrow u = 1,\ s = 100000,\ k=7$

$v > u \rightarrow v = 10,\ s = 100001,\ r = 10,\ k=8$

$v_0 = 0 \rightarrow v = 1,\ r = 100,\ k=9$

$u = v \rightarrow v = 0,\ r = 1000,\ s = 100101,\ k=10$

$||r||<||p|| \rightarrow$ result $= r$

GF(2^n) CorPh Numerical Example

$p=100101,\ m= 9,\ n= 5$

$r = 1000 \in \mathrm{GF}(2^6)$, $k=10$ (from AlmMonInv)

$j = 8$

$r = 10000,\ j = 7$

$r = 100000,\ ||r|| = ||p|| \rightarrow r = 101,\ j = 6$

$r = 1010,\ j = 5$

$r = 10100,\ j = 4$

$r = 101000,\ ||r|| = ||p|| \rightarrow r =1101,\ j = 3$

$r = 11010, j = 2$

$r = 110100,\ ||r|| = ||p|| \rightarrow r =10001,\ j = 1$

$r = 100010,\ ||r|| = ||p|| \rightarrow r = 111,\ j = 0$

$\therefore$GF(2^n) MonInv of $10100 = 111$ ($a^{-1}2^m$); Where $m=9$ & $n = 5$

Fig. 3. GF(2^n) MonInv computation numerical example

For both GF(p) and GF(2^n) MonInv hardware algorithms (Figure 1 and Figure 4, respectively), the AlmMonInv algorithm needs to finish its computation completely before the CorPh begins processing. This data dependency allows the use of the same hardware to execute both algorithms, i.e., both the AlmMonInv and CorPh. The algorithms are implemented in the unified and scalable hardware architecture as described in the following section.

4 The Unified and Scalable Inverter Architecture

Taking into account the amount of effort, time, and money that must be invested in designing an inverter, a scalable and unified architecture that can perform arithmetic in two commonly used algebraic finite fields is clearly advantageous. In this section, we present the hardware design of a Montgomery inverse architecture that can be used for both types of fields following the design methodology presented in [14]. The proposed unified architecture is obtained from the scalable architecture given in [14] but with some modifications, which slightly increases the longest path propagation delay and chip area. The scalable GF(p) Montgomery inverse architecture presented in [14] consisted in two main units, a non-scalable memory unit and a scalable computing unit. The memory unit is not scalable because it has a limited storage defined by the value of n_{max}. The data values of a and p are first loaded in the memory unit. Then, the computing unit read/write (modify) the data using a word size of *w* bits. The computing unit is completely scalable. It is designed to handle *w* bits every clock cycle. The computing unit does not know the total number of bits, n_{max}, the memory is holding. It computes until the controller indicates that all operands' words were processed. Note that the precision of the actual numbers used may be way smaller than n_{max} bits. The user needs to identify the type of finite field his application needs at the beginning of the computation. An input signal *FSEL* (field select) is used to tell the architecture weather GF(p) or GF(2^n) is the desired arithmetic domain.

```
GF(2^n) Multi-Bit Shifting AlmMonInv HW Algorithm (HW-Alg3)
    Registers:  u, v, r, s, x, y, z, & $p$ (all registers hold $n_{max}$ bits)
    Input:      $a2^m$, $2^n \in [1, p-1]$ ($p$=irreducible polynomial & $m \geq n$)
    Output:     result$\in [1, p-1]$ & $k$ (result=$a^{-1}2^{k-m} \bmod p$ & $n<k<2n$)
    1.   u = $p$; v = $a2^m$; r = 0; s = 1; x = 0; y = $2^n$; z = 0; $k$ = 0
    2.   if ($u_2u_1u_0$=000) then{u=ShiftR(u,3);s=ShiftL(s,3);$k$=$k$+3}; goto 8
    2.1. if ($u_2u_1u_0$=100) then{u=ShiftR(u,2);s=ShiftL(s,2);$k$=$k$+2}; goto 8
    2.2. if ($u_2u_1u_0$=110) then{u=ShiftR(u,1);s=ShiftL(s,1)}; goto 7
    3.   if ($v_2v_1v_0$=000) then{v=ShiftR(v,3);r=ShiftL(r,3);$k$=$k$+3}; goto 8
    3.1. if ($v_2v_1v_0$=100) then{v=ShiftR(v,2);r=ShiftL(r,2);$k$=$k$+2}; goto 8
    3.2. if ($v_2v_1v_0$=110) then{v=ShiftR(v,1);r=ShiftL(r,1)}; goto 8
    4.   S1 = Subtract(u, v); x = v$\oplus$u; z = r$\oplus$s
    5.   if ($S1_{borrow}$=0) then {u=ShiftR(x,1); r=z; s=ShiftL(s,1)}; goto 7
    6.   s = z; v = ShiftR(x,1); r = ShiftL(r,1)
    7.   $k = k + 1$
    8.   if (v $\neq$ 0) go to step 2
    9.   x = $p \oplus$ r ; z = $2p \oplus$ r ; S1 = Subtract (y,x); S2 = Subtract (y,z)
    10.  if ($S1_{borrow}$=0) then {result=x}
    10.1 else if ($S2_{borrow}$=0) then {result=z}
    10.2 else {result = r}

GF(2^n) Multi-Bit Shifting CorPh HW Algorithm (HW-Alg4)
    Input:   $r$, $p$, $m$, $2^n$ & $k$; Where $r$ ($r=a^{-1}2^{k-m} \bmod p$)& $k$ from HW-Alg3
    Output:  result; Where result = $a^{-1}2^m \pmod p$.
    11.  $j = 2m-k-1$; x = 0; y = 0; z = 0
    12.  $v = 2p$; $u = 3p$; s = $2^n$
    13.  While $j > 0$
    14.      if $j = 1$ then {$r$ = ShiftL($r$,1); $j=j-1$}
    15.      else {$r$ = ShiftL($r$,2); $j=j-2$}
    16.      x = $p \oplus r$ ; y = $u \oplus r$ ; z = $u \oplus r$
    16.1     S1=Subtract(s,x); S2=Subtract(s,y); S3=Subtract(s,z)
    17.      if ($S3_{borrow}$ = 0) then {$r$ = z}
    18.      else if ($S2_{borrow}$ = 0) then {$r$ = y}
    19.      else if ($S1_{borrow}$ = 0) then {$r$ = x}
    20.  result = $r$
```

Fig. 4. Montgomery inverse hardware algorithm for GF(2^n)

The block diagram for the Montgomery inverter hardware is shown in Figure 5. The memory unit is connected to the computing unit components. The memory unit is not changed from what is presented in [14]. It contains a counter to compute variable *k* and eight first-in-first-out (FIFO) registers used to store the inversion algorithm's variables. All registers, *u*, *v*, *r*, *s*, *x*, *y*, *z* and *p*, are limited to hold at most n_{max} bits. Each FIFO register has its own reset signal generated by the controller. They have counters to keep track of *n* (the number of bits actually used by the application).

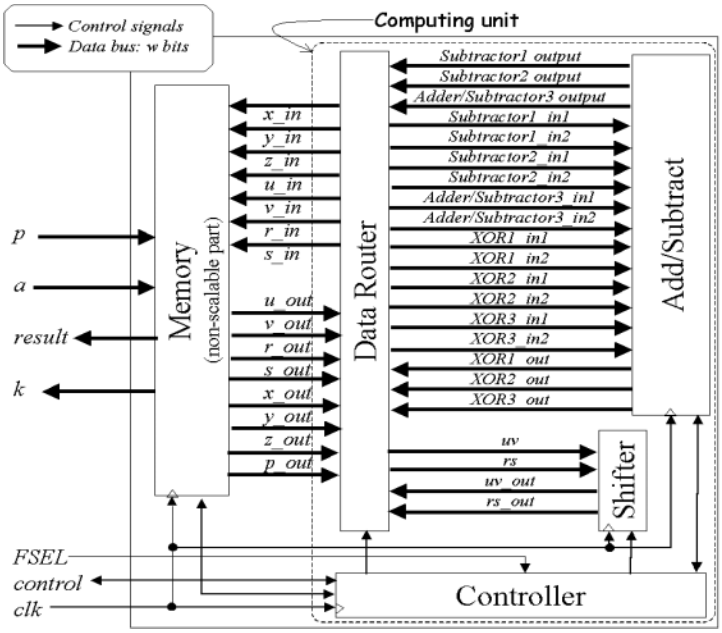

Fig. 5. Scalable and unified inverter hardware

The computing unit is made of four hardware blocks: add/subtract, shifter, data router, and controller block. The GF(p) add/subtract unit and the data router are the only components that need to be adjusted to make the inverter hardware unified for GF(p) and GF(2^n) finite fields.

The GF(p) add/subtract unit is originally built of two *w*-bit subtractors, a *w*-bit adder/subtractor, four flip-flops, one multiplexer, a *w*-bit comparator, and logic gates, as detailed in [14]. This unit is adjusted to operate for GF(2^n) by adding a set of *3w* parallel XOR gates used for steps 4 and 9 of HW-Alg3 and step 16 of HW-Alg4. The new add/subtract unit is shown in Figure 6. The signal *Control* makes the unit perform either two subtractions plus one addition (step 4 of HW-Alg1), or three subtractions (step 16 of HW-Alg2 and step 16.1 of HW-Alg4). Three flip-flops are used to hold the intermediate borrow bits of the subtractors and the carry bit of the adder to implement the multi-precision operations. The fourth flip-flop is used to store a flag that keeps track of the comparison between *u* and *v*, which is used to perform step 8 of HW-Alg1 and HW-Alg3. The subtractors borrow-out bits are connected to the controller through signals that are useful only at the end of each multi-precision addition/subtraction operation. Subtractor1 borrow-out bit will affect the flow of the operation to choose either step 5 or step 6 of both HW-Alg1 and HW-Alg3. It is also essential in electing the result observed in step 10 of HW-Alg1 and of HW-Alg2. The three subtractors borrow-out bits ($S1_{borrow}$, $S2_{borrow}$, $S3_{borrow}$) are likewise necessary for selecting the correct solution of the 'if' condition to be one of the steps 17, 18, or 19, from the HW-Alg2 and from the HW-Alg4 algorithms.

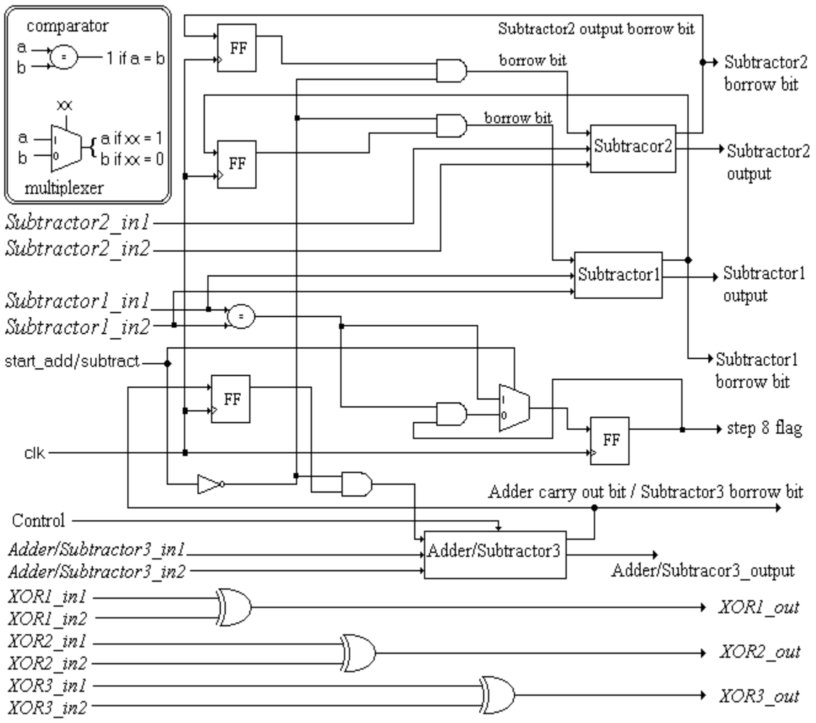

Fig. 6. Add/Subtract unit of the scalable and unified hardware

The shifter is made of two multiplexers and two registers with special mapping of some data bits, as shown in Figure 7. Depending on the controller signal *Distance*, the shifter acts as a one, two, or three-bit shifter. Two types of shifting operations are needed in the HW-Alg1 and the HW-Alg3 algorithms, shifting an operand (*u* or *v*) through the *uv* bus one, two, or three bits to the right, and shifting another operand (*r* or *s*) through the *rs* bus by a similar number of bits to the left. Shifting *u* or *v* is performed through Register1, which is of size *w*-1 bits. For each word, all the bits of *uv* are stored in Register1 except for the least significant bit(s) to be shifted, it is (or they are) read out immediately as the most significant bit(s) of the output bus *uv_out*. Shifting *r* or *s* to the left is performed via Register2, which is of size *w*+3 bits similar to shifting *uv* but to the other direction. When executing the HW-Alg2 or HW-Alg4, the shifting is performed either to one or two bits to the left only, which is via MUX2 and Register2 ignoring MUX1 and Register1.

The data router capabilities are extended to satisfy the unified architecture requirements. It interconnects the memory, add/subtract, and shifter units. The possible configurations of the data router are shown in Figure 8.

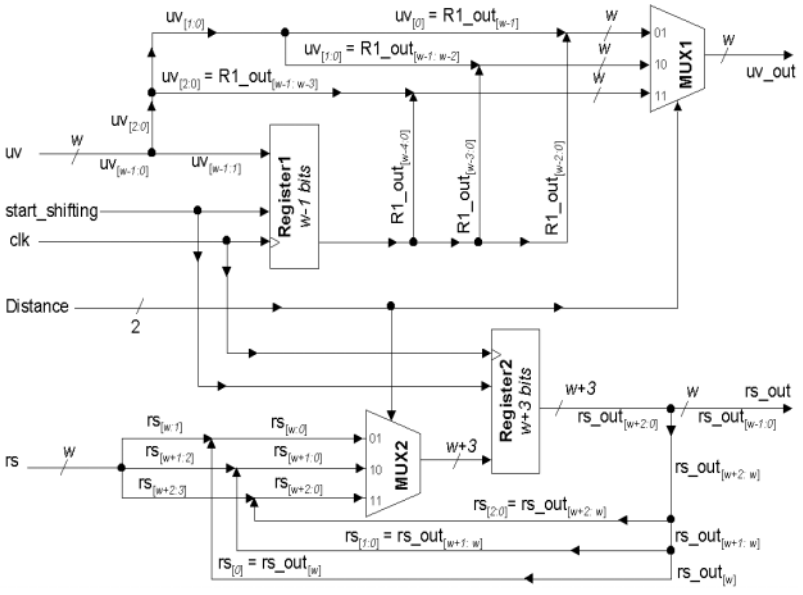

Fig. 7. Shifter unit hardware

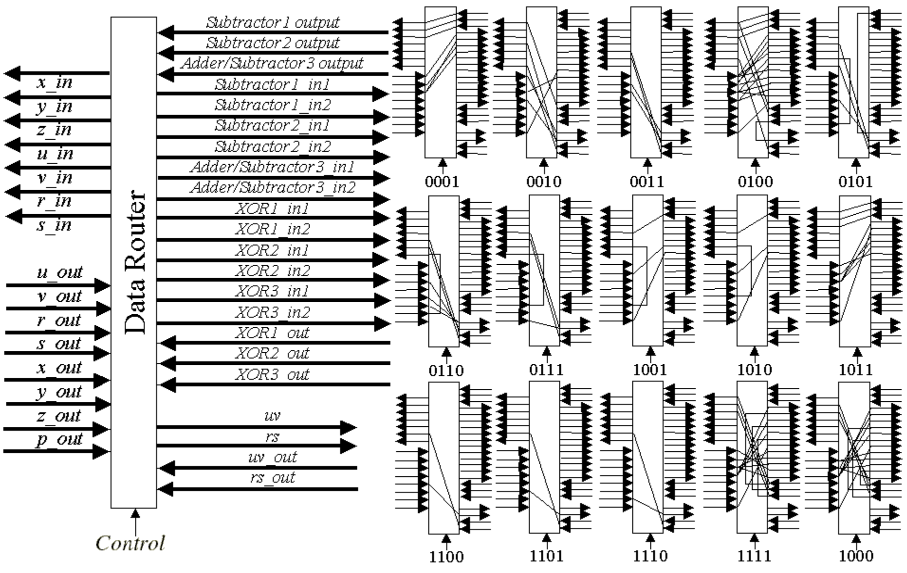

Fig. 8. Data router configurations

5 Modeling and Analysis

The unified and scalable inverter was modeled and simulated in VHDL. Previously, a fixed design (full precision) and other scalable inverter designs for inversion in GF(p) were also described in VHDL. All VHDL descriptions of the scalable designs, including the new unified ones, have two main parameters, namely n_{max} and w. The fixed hardware, however, is parameterized by n_{max} only. Their area and speed are presented in this section. Also a reconfigurable hardware [16] that can perform the inversion in both GF(p) and GF(2^n), besides other functions, is considered in the comparison. We didn't define a specific architecture for the adders and subtractors used in our VHDL implementations. Thus, the synthesis tool chooses the best option in terms of area from its library of standard cells. As a result, all proposed designs use the same type of adders and subtractors.

5.1 Area Comparison

The exact area of any design depends on the technology and minimum feature size. For technology independence, we use the equivalent number of NOT-gates as an area measure [6]. A CAD tool from Mentor Graphics (Leonardo) was used. Leonardo takes the VHDL design code and provides a synthesized model with its area and longest path delay. The target technology is a $0.5\mu m$ CMOS defined by the 'AMI0.5 fast' library provided in the ASIC Design Kit (ADK) from the same Mentor Graphics Company [11]. It has to be mentioned here that the ADK is developed for educational purposes and cannot be thoroughly compared to technologies adopted for marketable ASICs. It however, provides a framework to contrast all scalable hardware designs together and with the fixed one. The sizes of the designs are compared in Figure 9. Observe that the fixed design has a better area if the maximum number of bits used (n_{max}) is small which is useless in cryptographic applications [5]. The unified designs are larger than the GF(p) ones with a calculated average of 8.4% more hardware area.

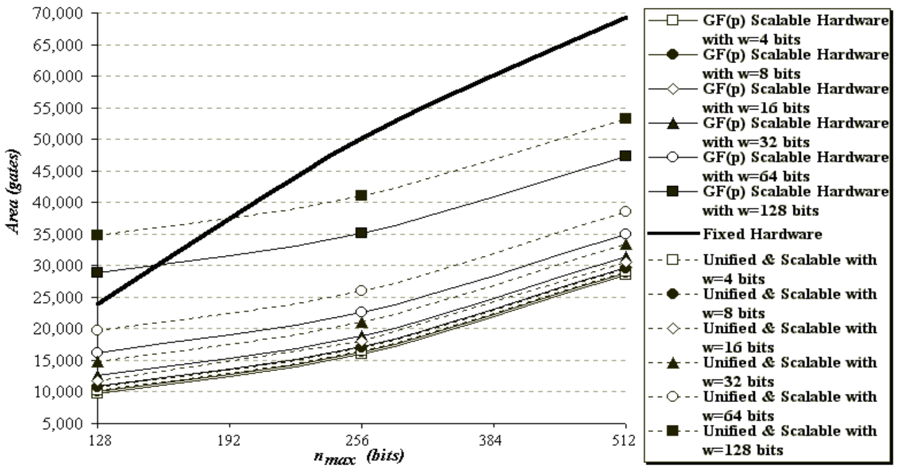

Fig. 9. Area comparison

The area of the unified designs were also compared with the reconfigurable hardware [16], but not shown in Figure 9. The reconfigurable design core is built of 880,000 devices [16]. Assume a device is corresponding to a transistor and our NOT-gate is equivalent to two transistors [6], so the reconfigurable hardware core is equivalent to 440,000 gates, which means that the reconfigurable design is eight times greater than the largest unified hardware shown in Figure 9. Of course, the design in [16] does more than inversion, but its datapath is responsible for most of the area, and would be used anyway for the inversion computation.

5.2 Speed Comparison

The total computation time is a product of the number of clock cycles the algorithm takes and the clock period of the final implementation. This clock period changes with the value of w in the unified and scalable hardware, and changes with the value of n_{max} in the fixed hardware. This is because $w = n_{max}$ in the fixed hardware. All VHDL coded designs clock cycle periods are generated automatically by Leonardo, which determines the longest path delay of the hardware circuits. The clock period of the reconfigurable design was considered as being 20ns/cycle (operates at 50MHz clock rate frequency) [16].

The number of clock cycles depends completely on the data and the algorithm. A probabilistic study described in [18] is used to estimate the average number of clock cycles. For the fixed design, the average number of clock cycles equal to $C_f = 1.525n$. For all scalable designs, the average number of clock cycles is $C_s=(2.4125n+1)\lceil n/w \rceil$, which is exactly the same for the unified designs presented in this paper. Hence, adjusting the scalable designs to be unified did not change the number of clock cycles of the inverse computation. However, the clock cycle period of the unified designs increased slightly, making the total computation time of the unified hardware different than what was given in [18]. The number of clock cycles for the reconfigurable hardware to complete the inversion process is $C_r=14.5n$ [16].

Similar to the GF(p) scalable hardware of [18], the unified and scalable hardware can have several designs for each n_{max}, depending on w. For example, Figure 10 shows the delay of several designs of the unified and scalable hardware compared to the reconfigurable, GF(p) scalable, and fixed hardware designs, all modeled for $n_{max}=512$ bits, which is a practical number for future cryptographic applications [5]. Observe how the actual data size (n) plays a big role on the speed of the designs. In other words, as n reduces and w is small, the number of clock cycles decrease significantly, which considerably reduces the overall computing time of all scalable designs (including the unified ones) compared to the others. This is a major advantage of the scalable hardware over the fixed [14,18] and reconfigurable ones.

The new unified designs when compared to the scalable design for GF(p) only have very similar characteristics. Overall, it needs an average of 19.8% more time than the designs for GF(p) [18]. Another observation from Figure 10 is that the unified designs are faster than the fixed one as long as:

$$n < \begin{cases} (\log_2 w)n_{max}/8 & when\ w < n_{max}/8 \\ n_{max} & when\ w \geq n_{max}/8 \end{cases}$$

which is generalized for different n_{max} values. Several experimental tests were done for n_{max}= *32, 64, 128, 512* and *1024* bits. Figure 10 also shows that the unified designs are comparable to the reconfigurable one giving better performance when:

$$n < \begin{cases} (\log_2 w)\, n_{max}/32 & \text{when } w < n_{max}/8 \\ (\log_2 w)\, n_{max}/25 & \text{when } w \geqslant n_{max}/8 \end{cases}$$

Consider the case when $n=n_{max}=512$ bits in Figure 10, the unified design with $w=64$ bits has almost the same speed as the fixed one, but the ones with $w=128$ bits remain faster. In fact, as w gets bigger the total time decreases, which is also true when comparing among the different unified designs while $n \geq w$, as also proven before in [18] for the GF(p) scalable designs. Whenever $n < w$ considering the unified and scalable designs, the scalability advantage of these designs is reduced since the number of words to be processed reached its lower limit, but still the unified and scalable designs are faster than the fixed one.

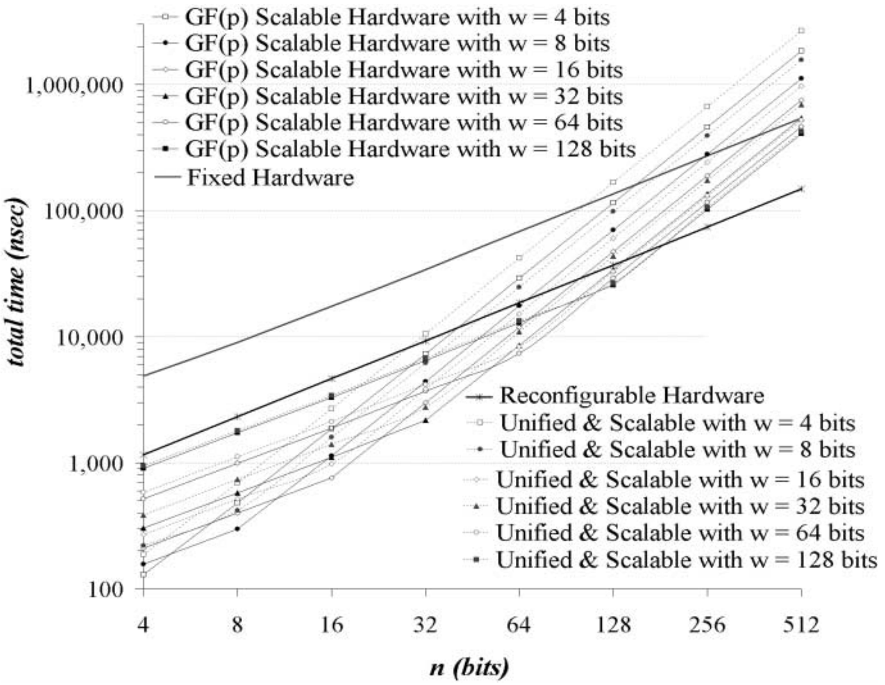

Fig. 10. Delay comparison of designs with n_{max} = 512 bits

6 Conclusion

This paper presents a scalable inverter for both finite fields GF(p) and GF(2^n) in a unified hardware module that applies the design approach proposed in [14,18,19]. The primary contribution of this research is to show that it is possible to design a unified hardware without compromising scalability and area efficiency. The unified inverter hardware is built of two main units, a memory unit and a computing unit. The memory unit defines the upper bound of the number of bits that the hardware can handle. The computing unit is the real scalable hardware, it is designed to fit in constrained areas and perform the computation of numbers in a repetitive way. Our analysis shows that as the word size of the scalable computing unit reduces, the hardware area decreases and the possible clock frequency increases. However, if we increase the computing unit word size, the clock frequency is reduced, but for $n > w$ the overall computing time is also reduced, which is considered a normal area-time tradeoff.

Several configurations of the proposed inverter hardware (different word lengths) were described and synthesized using Mentor Graphics CAD tools. They were compared with equivalent configurations of a previously proposed inversion hardware design for inversion in GF(p) only. The comparisons show that this unified and scalable structure is very attractive for cryptographic systems, particularly for ECC where there is a need for modular inversion of large numbers in both finite fields GF(p) and GF(2^n) depending on the application usage.

Acknowledgments. The authors would like to thank KFUPM-Saudi Arabia and NSF under the CAREER grant CCR-0093434-"Computer Arithmetic Algorithms and Scalable Hardware Designs for Cryptographic Applications" for providing financial support toward this research.

References

1. E. Savas and C. K. Koç. The Montgomery Modular Inverse – Revisited. *IEEE Trans. on Computers*, 49(7): 763-766, July 2000.
2. T. Kobayashi and H. Morita. Fast Modular Inversion Algorithm to Match Any Operation Unit. *IEICE Trans. Fundamentals*, E82-A(5):733-740, May 1999.
3. B. S. Kaliski. The Montgomery Inverse and its Applications. *IEEE Trans. on Computers*, 44(8):1064-1065, Aug. 1995.
4. E. Savas, A. F. Tenca, and C. K. Koç. A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^k). *In Cryptographic Hardware and Embedded Systems*, Lecture notes in Computer Science. Springer, Berlin, Germany, 2000.
5. I. Blake, G. Seroussi, and N. Smart. Elliptic Curves in Cryptography. *Cambridge University Press*: New York, 1999.
6. M. D. Ercegovac, T. Lang, and J. H. Moreno. Introduction to Digital System. *John Wiley & Sons, Inc.*, New York, 1999.
7. P. Montgomery. Modular Multiplication without Trail Division. *Mathematics of Computation*, 44(170): 519-521, April 1985.
8. N. Takagi. Modular Inversion Hardware with a Redundant Binary Representation. *IEICE Trans. on Information and Systems*, E76-D(8): 863-869, Aug. 1993.
9. J.-H. Guo, and C.-L. Wang. Hardware-Efficient Systolic Architecture for Inversion and Division in GF(2^m). *IEE Proceedings: Computers and Digital Techniques*, 145(4): 272-278, July 1998.
10. Choudhury, Pal, and Barua. Cellular Automata Based VLSI Architecture for Computing Multiplication and Inverses in GF(2^m). *Proceedings of the 7th IEEE International Conference on VLSI Design*, Calcutta, India, January 5-8 1994.
11. Mentor Graphics Co., *ASIC Design Kit*, http://www.mentor.com/partners/hep/AsicDesign Kit/dsheet/ami05data book.html

12. M. A. Hasan. Efficient Computation of Multiplicative Inverses for Cryptographic Applications. *Proceeding of the 15th IEEE Symposium on Computer Arithmetic*, June 2001.
13. M. Feng. A VLSI Architecture for Fast Inversion in GF(2^m). *IEEE Trans. on Computers*, 38(10):1383-1386, Oct. 1989.
14. A. A.-A. Gutub, A. F. Tenca, and C. K. Koç. Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation. *ISVLSI 2002: IEEE Computer Society Annual Symposium on VLSI*, Pittsburgh, Pennsylvania, April 25-26 2002.
15. J. R. Michener and S. D. Mohan. Clothing the E-Emperor. *IEEE Compute*, 34(9):116-118, Sep. 2001.
16. J. Goodman and A. P. Chandrakasan. An Energy-Efficient Reconfigurable Public-Key Cryptogrphy Processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808-1820, Nov. 2001.
17. D. Knuth. The Art of Computer Programming – Seminumerical Algorithms, 2nd ed. Vol. 2, Reading, MA: *Addison-Wesley*, 1981.
18. A. A.-A. Gutub and A. F. Tenca. A Scalable VLSI Architecture for Montgomery Inversion in GF(p). *Submitted for publication in March 2002 to IEEE Trans. on VLSI.*
19. A. A.-A. Gutub, New Hardware Algorithms and Designs for Montgomery Modular Inverse Computation in Galois Fields GF(p) and GF(2^n), *Ph.D. thesis, Oregon State University, 2002.*

Appendix

This Appendix details the computations and verifies the results used in the GF(2^n) MonInv numerical example shown in Figure 3. The example defines $m=9$ and $n=5$; where n is the degree of the irreducible polynomial and m (of the Montgomery constant 2^m) is any number as long as $m \geq n$. To simplify the arithmetic lets only use the binary representation of polynomials. The MonInv takes the inputs $a=1001$ and $p=100101$. However, a is represented into Montgomery domain as $a2^m$, which is calculated as follows:

$$a=1001 \Rightarrow a2^m=a2^9=1001000000000$$

but since 1001000000000 needs to be reduced by p or a multiple of p until the number of significant bits of $a2^9$ is less or equal to n (the degree of polynomial $a(x)x^m mod\, p(x)$ should be less than the degree of the irreducible polynomial ($p(x)$)), so

$$a2^9 \oplus 2^7p=1001000000000 \oplus 1001010000000=10000000$$

and 10000000 also needs reduction

$$10000000 \oplus 2^2p =10000000 \oplus 10010100 = 10100.$$

So

$$a2^m\, mod\, p = a2^9\, mod\, p = 1001000000000\, mod\, p \equiv 10100.$$

The fact that GF(2^n) MonInv of 10100 is $a^{-1}2^m$=111, can be similarly verified. The MonInv numerical example in Figure 3 calculated that $a^{-1}2^9 = 111 \Rightarrow a^{-1} = 111/2^9$.

Any congruent polynomial can be XORed with the irreducible polynomial, such as:

$$a^{-1}2^9=111\equiv 111 \oplus 100101 =100010 \rightarrow a^{-1}2^8=10001$$
$$a^{-1}2^8=10001\equiv 10001 \oplus 100101=110100 \rightarrow a^{-1}2^6=1101$$
$$a^{-1}2^6=1101\equiv 1101 \oplus 100101=101000 \rightarrow a^{-1}2^3=101$$
$$a^{-1}2^3=101\equiv 101 \oplus 100101=100000 \rightarrow a^{-1}=100$$

To confirm that the GF(2^n) MonInv of 10100 is 111, when $m=9$ and $n=5$, it is enough to show that $a\, .\, a^{-1}\, mod\, p= 1$, as follows:

$$a\, .\, a^{-1}= 1001\, .\, 100 = 100100$$
$$100100\, mod\, p = 100100 \oplus 100101=1$$