Efficient Scalable VLSI Architecture for Montgomery Inversion in GF(p)

Adnan Abdul-Aziz Gutub[a,*] *and Alexandre Ferreira Tenca*[b]

[a] Computer Engineering Department, King Fahd University of Petrolium & Minerals, Dhahran 31261, Saudi Arabia
[b] Electrical and Computer Engineering Department, Oregon State University, Corvallis, Oregon 97331 USA.

Abstract

The multiplicative inversion operation is a fundamental computation in several cryptographic applications. In this work, we propose a scalable VLSI hardware to compute the Montgomery modular inverse in GF(p). We suggest a new correction phase for a previously proposed almost Montgomery inverse algorithm to calculate the inversion in hardware. We also propose an efficient hardware algorithm to compute the inverse by multi-bit shifting method. The intended VLSI hardware is scalable, which means that a fixed-area module can handle operands of any size. The word-size, which the module operates, can be selected based on the area and performance requirements. The upper limit on the operand precision is dictated only by the available memory to store the operands and internal results. The scalable module is in principle capable of performing infinite-precision Montgomery inverse computation of an integer, modulo a prime number. This scalable hardware is compared with a previously proposed fixed (fully parallel) design showing very attractive results.

Keywords: Montgomery inverse, Elliptic curve cryptography, Scalable hardware design

1. Introduction

MODULAR inverse arithmetic is an essential arithmetic operation in public-key cryptography. It is used in the Diffie-Hellman key exchange method [5], and it was also adopted to calculate private decryption key in RSA [4]. Modular inversion is a basic operation in the elliptic curve cryptography (ECC) [1,2,9-13,20-24]. The main focus of this work is ECC because of its promise to replace older public-key cryptographic systems [9-12,20]. ECC arithmetic consists of mainly modular computations of addition, subtraction, multiplication, and inversion.

Inversion is generally known to be the slowest of all other field level arithmetic operations in ECC [1,2,11,16-18]. Many researchers propose minimizing the use of modular inversion by adopting elliptic curves defined for projective coordinates [9-12], which substitutes the inverse by several multiplication operations. Inversion, in the projective coordinate systems, is required only once at the end, to convert points in projective coordinates back to affine coordinates. If this inversion is made faster, it will improve the complete ECC system to be less time-consuming.

A high-speed modular inverse calculation is the main reason to do inversion in hardware instead of software [16-18]. If it is possible to compute the inverse in less time than nine multiplication operations, then it is more efficient to use the affine coordinate system instead of employing projective coordinate systems [2,10]. Even if the speed to compute the inverse is not that good to justify the use of affine coordinates, projective coordinate inversion computation when performed within hardware is still faster than software [6,13,16-18,20-24], which will provide better performance for the overall cryptographic system.

Another main reason to implement the inverse in hardware is security. For cryptographic applications, it is more secure to have all the computations handled in hardware, inside an IC-chip, instead of mixing some computations performed in software with others processed in hardware. Software-based systems can be interrupted and trespassed by intruders much easier than hardware, which can jeopardize the security of the whole application. Moreover, stealing information from software systems is easier than from hardware.

Modular inversion is often performed by algorithms based on the Extended Euclidean algorithm [11]. Several inversion VLSI designs are described in the literature [13,16-18,20-24]. Most of them [13,17,18,20-24] are for inversion in Galois

* Corresponding author. Tel. +966-3-860-1723, Fax: +966-3-860-3059
Email address: gutub@kfupm.edu.sa

Fields GF(2^k). Several [13,17,18,21-24] are based upon extensive combinational networks. The inversion in GF(2^k) is fast due to the elimination of the carry propagation in GF(2^k) calculations. However, the area used in these parallel organizations are very large, of order O(n^2). Hasan in [20] proposed to implement the GF(2^k) inversion algorithm in a smaller area but with slower speed. His hardware performs word-by-word computation on the operands instead of computing all the words in parallel. Since we focus on GF(p), the designs proposed for GF(2^k) in [13,17,18,20-24] have no direct link to this work.

Takagi in [16], proposed an inverse algorithm for hardware with a redundant binary representation. Each number is represented by a digit in the set {0,1,-1}. Redundant representation is used to avoid the carry propagation delay problem. However, the hardware in [16] requires more area than the design proposed here and also needs data transformations that are usually expensive.

The standard modular inverse over GF(p) can be defined by the following example. Assume a is an integer in the range [1, p-1]. Integer x is called the modular inverse, or modulo inverse, of integer a if-and-only-if: $ax \equiv 1 (mod\ p)$; where $x \in [1, p-1]$. It is normally represented as $x=a^{-1} mod\ p$ [1]. The Montgomery modular inverse algorithm suitable for our research is presented in [1]. The algorithm consists of two phases: first phase (almost Montgomery inverse) and correction phase [3]; in this work we suggest replacing the correction phase with a simpler one. A further modification to the inversion algorithm to use multi-bit shifting instead of single-bit shifting is also proposed. These alterations reduce the number of clock cycles without significantly increasing the clock period, which results in an overall speedup of the inverse computation.

Our improved algorithm is implemented in hardware using scalability features, which allows the use of a fixed-area scalable circuit to perform inversion of unlimited precision operands, as originally introduced in [7]. The hardware divides the long-precision numbers in words and each word is processed in a clock cycle. It is shown that this hardware is appropriate for cryptographic applications. The work shows the area and speed of several scalable hardware configurations compared with a fixed fully parallel design presented in [7]. It gives various practical improvement results to show how attractive this contribution is.

In the coming section, the reason behind choosing Montgomery modular method is described. Section 2 presents the Montgomery inverse algorithm including the correction phase proposed in this work. Section 3 explains the multi-bit shifting strategy and corresponding modifications to the hardware algorithm. In Section 4 the scalable hardware implementation is described in some detail. The comparison between several different hardware implementations is given in Section 5.

1.1 Why Montgomery Arithmetic?

Cryptography is heavily based on modular multiplication, which involves the division by the modulus in its computation. Division, however, is a very expensive operation [14]. This fact made researchers seek out for methods to reduce the division impact and make modulo multiplication less time consuming.

In 1985, P. Montgomery invented his clever algorithm to perform modular multiplication without trial division [15]. He replaced the normal division with divisions by two, which is easily performed in the binary number representation (shifting the binary representation of a number one bit to the right). The cost behind using Montgomery's method resides in some extra-required computations to represent the numbers into Montgomery domain and vice-versa [1,2,6,15]. The Montgomery multiplication function [15] of two numbers, symbolized as MonPro (Montgomery product) [1], is applied to a and b producing the result c=MonPro$(a,b)=ab2^{-n} mod\ p$; where $a,b \in [0,p-1]$, n is the actual number of bits of the prime modulus p, and $2^{n-1} \leq p < 2^n$. The reader is referred to [15] for more knowledge of Montgomery multiplication.

To use Montgomery's method for ECC, as an example, the integer input operands are first transformed into Montgomery domain, all the modular operations are performed in this Montgomery domain, and the result is converted back to the original integer values. Because the inversion is one of these modular operations, researchers propose to have dedicated procedures to compute the modular inverse in the Montgomery domain, i.e., Montgomery modular inverse algorithms [1,2], which has also been proven that it is faster than the conventional Extended Euclidean algorithm [3].

2. MONTGOMERY INVERSE ALGORITHM AND PROPOSED MODIFICATIONS

Two Montgomery modular inverse algorithms are found in the literature [1,2]. Both modify a technique proposed by Kaliski in 1995 [3], to make it more suitable and faster for cryptography using Montgomery's idea. The Montgomery inverse function is to calculate $x=a^{-1}2^n mod\ p$ from $a2^n$. Kaliski method however, takes an integer a and produces $x=a^{-1}2^n mod\ p$. If a is an integer, the algorithm will calculate the inverse of a, but represented in Montgomery domain, as shown in Fig. 1. However, in order to have fast ECC operations using Montgomery arithmetic, numbers should be

represented into Montgomery domain and all modular operations should be performed in this domain. In other words, if the number a is already in Montgomery domain, the application of Kaliski's routine will not give the needed Montgomery inverse result and some extra arithmetic operations are required to get it. Kaliski method is summarized next. Then, we propose a new correction phase that results in a speedup in its hardware implementation.

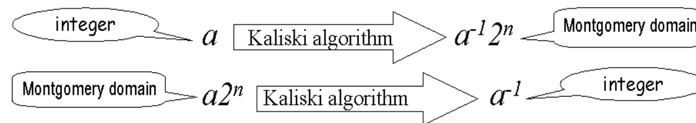

Fig. 1 Types of input/output numbers for Kaliski algorithm

2.1 Kaliski Algorithm

Kaliski algorithm [1,3] is derived from the extended Euclidean algorithm and is divided in two phases as shown below. Phase I, also called almost Montgomery inverse (AlmMonInv) [1], takes the inputs a and p, and give outputs r and k; where $r=a^{-1}2^k mod\ p$, and $n<k<2n$. Phase II takes the outputs of Phase I as its inputs, and gives the final result $x = a^{-1}2^n\ mod\ p$, where $2^{n-1} \le p<2^n$. Note that in both phases the values of a and $x \in [1, p-1]$.

Phase I: Almost Montgomery Inverse, *AlmMonInv(a)*
Input: a & p; where a is in the range [1, p-1].
Output: r & k; where $r = a^{-1}2^k\ mod\ p$ & $n < k < 2n$
1. $u = p$, $v = a$, $r = 0$, and $s = 1$
2. $k = 0$
3. while ($v > 0$)
4. if u is even then $u = u/2$, $s = 2s$
5. else if v is even then $v = v/2$, $r = 2r$
6. else if $u > v$ then $u = (u - v)/2$, $r = r+s$, $s = 2s$
7. else $v = (v - u)/2$, $s = s+r$, $r = 2r$
8. $k = k + 1$
9. if $r \geq p$ then $r = r-p$
10. return $r = p-r$

Phase II
Input: p, $r=a^{-1}2^k mod\ p$, k & n; where r & k from Phase I
Output: x; where $x = a^{-1}2^n\ mod\ p$
11. for $i = 1$ to $k - n$ do
12. if r is even then $r = r/2$
13. else $r = (r + p)/2$
14. return $x = r$

2.2 New Approach for Montgomery Inverse

Let's consider the main Montgomery inverse problem again. Our new way to calculate the Montgomery inverse is by first applying the *AlmMonInv* on the input $a2^n$ to produce r and k, then multiplying r ($r = a^{-1}2^{k-n} mod\ p$) by 2^{2n-k} to immediately generate the needed Montgomery inverse result $a^{-1}2^n mod\ p$.
Multiplying ($r = a^{-1}2^{k-n} mod\ p$) by (2^{2n-k}) is traditionally performed as follows:

$$([(((((a^{-1}2^{k-n}).2).\ 2).\ 2).\ 2)\ldots\ldots\ldots\ldots 2).\ 2)]\ mod\ p) = a^{-1}2^n\ mod\ p$$

2n-k times

This arrangement of applying the modular operation after completing the multiplication is very expensive because the result of the multiplication by 2^{2n-k} can go far above the modulus and a large amount of hardware to handle it [11]. However, the result can be simplified by introducing the modular reduction operation with each multiplication by 2 as the following:

$$[(((((a^{-1}2^{k-n}).2)\ mod\ p).2)\ \ldots\ 2)\ mod\ p).2)\ mod\ p)]=a^{-1}2^n mod\ p$$

The modular reduction operation is performed by a subtraction of p whenever the number exceeds p. The proposed correction phase consists then in performing a multiplication of the $a^{-1}2^{k-n}$ by $C = 2^{2n-k}$ as outlined below:

Correction Phase (Multiply by 2^{2n-k})
Input: r, p, n & k; where r & k are *AlmMonInv* outputs
Output: x; where $x = a^{-1}2^n\ mod\ p$
1. for $i = 2n-k$ to 0 do
2. $r = 2r$
3. if $r > p$ then $r = (r - p)$
4. return $x = r$

2.3 Evaluation of Alternatives

Several methods considered for hardware computation of the Montgomery inverse are shown in Fig. 2; including the procedures proposed by Savas and Koç in [1] using MonPro. Each path in the graph has its own set of routines and its total computation time. Fig. 2 presents the approximate number of iterations for each routine. Note that the number of iterations for multiplication is estimated considering serial-parallel multipliers, because fully parallel multipliers are impractically large [6].

All approaches of Fig. 2 lead to the same final result. However, the number of iterations in each path proves that our two-phase method, the *AlmMonInv* followed by the *correction phase* (the bold path shown in Fig. 2), is the fastest. It requires only $2n$ iterations to complete the inversion, the *AlmMonInv* needs $1.5n$ iterations, and the correction phase (*CorPh*) needs $0.5n$ iterations, assuming an average value of $k=1.5n$ [1].

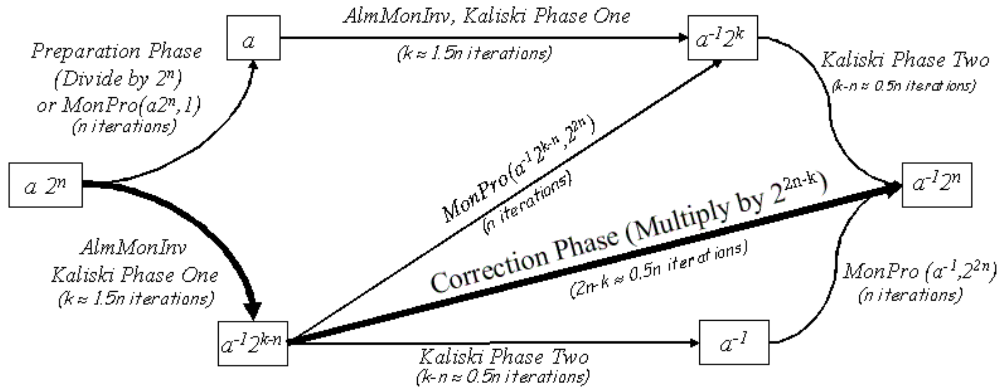

Fig. 2 Different ways to compute the Montgomery inversion

3. Multi-Bit Shifting

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 following section presents an improvement of the *AlmMonInv* and CorPh algorithms based on a multi-bit shifting method.

3.1 AlmMonInv Hardware Algorithm

The AlmMonInv algorithm, when observed from hardware point-of-view, contains operations that are easily mapped to hardware as clarified thoroughly in [7]. For example, one-bit right shifting the binary representation of number u (ShiftR(u,1)) is equivalent to perform division by two, or shifting s one bit to the left (ShiftL(s,1)) is equal to multiplication by two. Checking for a number to be even or odd requires a test of the least significant bit (LSB). The comparison of two numbers is performed after subtracting one from the other. If the subtraction result is positive (the borrow-bit is zero) the first number is greater, otherwise the opposite is true. Such hardware mapping is represented as the hardware algorithm below:

AlmMonInv Hardware Algorithm (HW-Alg1)

Registers: u, v, r, s, & p (all five registers hold n bits).
Input: $a \in [1, p-1]$, p = modulus; where $2^{n-1} \le p < 2^n$
Output: result$\in[1, p-1]$ & k; where result=$a^{-1}2^k \bmod p$ & $n \le k \le 2n$

1. u = p; v = a; r = 0; s = 1; k = 0
2. if (u_0 = 0) then { u = ShiftR(u,1) ; s = ShiftL(s,1)}; goto 7
3. if (v_0 = 0) then { v = ShiftR(v,1) ; r = ShiftL(r,1)}; goto 7
4. S1 = Subtract (u, v); S2 = Subtract (v, u); A1 = Add (r, s)
5. if($S1_{borrow}$=0)then{u=ShiftR(S1,1));r=A1;s=ShiftL(s,1)};goto 7
6. s = A1; v = ShiftR(S2,1); r = ShiftL(r,1)
7. $k = k + 1$
8. if (v ≠ 0) go to step 2
9. S1 = Subtract (p, r); S2 = Subtract ($2p$, r)
10. if($S1_{borrow}$=0)then{return result=S1}; else {return result=S2}

3.2 Best Maximum Distance for Multi-bit Shifter

The operation to shift numbers *u* and *s* (step 2), or *v* and *r* (step 3), in the HW-Alg1, are performed depending on u_0 and v_0. In fact, when either u_0 or v_0 is zero, only shift operations happen. We propose to observe several least significant (LS) bits of *u* and *v* and depending on the number of consecutive zeros in the LS positions perform a multi-bit shifting. This approach clearly reduces the number of iterations in the HW-Alg1 when steps 2 and 3 are modified to perform such multi-bit shift. The exact number of bits to be shifted depends on the data that is modified during the process. Thus a probabilistic analysis [8] is used to decide the maximum number of bits to be shifted, which also affected in the maximum distance of the multi-bit shifter hardware design. In the HW-Alg1 presented earlier, the loop (steps 2 through 8) is executed for *k* iterations. Based on experimental statistics collected with a software implementation of the inversion algorithm generated for this purpose, almost half of the *k* algorithm iterations are used executing step 4 (addition and subtraction) and the other half executing only steps 2 or 3 (shifting process). Applying the multi-bit shifting approach will reduce the number of iterations for the shifting process only. Reusing $p=0.5$ as the probability of performing a shift operation, we model the average number of iterations by probability equations. Let *x* be the maximum number of bits to be shifted, Table 1 shows probabilistic equations to compute the number of iterations when a multi-bit shifter of up to *x* bits is available. The first polynomial term stands for the number of iterations used for addition and subtraction (step 4 of HW-Alg1, which is not affected by *x*). The term/terms after that are used for the shifting, which represent the *x* multi-bit shifting modification. Given the value *p* that was defined before, the average number of iterations (*i*) is computed as listed in the last column of Table 1.

After comparing the different *i* values, the notable improvement is found for the case with $x=3$ (shifting up to three bits), which gives the average of 15% reduction to *k*. Note that increasing *x* over three is not giving significant improvement beyond the 15%. For this reason, we choose to modify the HW-Alg1 to shift up to three bits.

Table 1 Average number of iterations

x	Probabilistic Equations	*i*
1	$(1-p)k + pk$	1.00 *k*
2	$(1-p)k + p[(1-p)k + p\,k/2]$	0.88 *k*
3	$(1-p)k + p[(1-p)k + p((1-p)\,k/2 + p\,k/3)]$	0.85 *k*
4	$(1-p)k + p[(1-p)k + p((1-p)\,k/2 + p\,[(1-p)\,k/3 + p\,k/4])]$	0.849 *k*
5	$(1-p)k + p[(1-p)k + p((1-p)\,k/2 + p[(1-p)k/3 + p((1-p)k/4 + pk/5)])]$	0.847 *k*

3.3 Adjustments to HW-Alg1

The new capability to shift up to three bits per iteration requires a modification in the HW-Alg1 as shown below:

Multi-Bit Shifting HW-Alg1 (M.HW-Alg1)

Registers: u, v, r, s, & *p* (all five registers hold *n* bits)
Input: $a \in [1, p-1]$, p = modulus.
Output: result$\in[1, p-1]$ & *k*; where result=$a^{-1}2^k \bmod p$ & $n \le k \le 2n$

1. u = *p*, v = *a*, r = 0, s = 1, *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($u_2u_1u_0$=X10)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($v_2v_1v_0$=X10)then{v=ShiftR(v,1);r=ShiftL(r,1)};goto 8
4. S1 = Subtract (u, v); S2 = Subtract (v, u); A1 = Add (r, s)
5. if($S1_{borrow}$=0)then{u=ShiftR(S1,1);r=A1;s=ShiftL(s,1)};goto 7
6. s = A1; v = ShiftR(S2,1); r = ShiftL(r,1)
7. $k = k + 1$
8. if (v ≠ 0) go to step 2
9. S1 = Subtract (*p*, r); S2 = Subtract (2*p*, r)
10. if($S1_{borrow}$=0)then{return result=S1}; else{return result = S2}

The M.HW-Alg1 when implemented in hardware requires: two subtractors (used in steps 4 and 9), an adder (step 4), a *k*-counter (that variably increments up to three), two multi-bit shifters (to shift u and s or v and r up to three bits, steps 2 to 3.2), and five *n*-bit registers (to store all the variables: u, v, r, s and *p*).

3.4 Suitable Multi-Bit Shifting the CorPh

The *CorPh* algorithm contains operations that are easily mapped to hardware components as shown in the *CorPh* hardware algorithm (HW-Alg2) below:

<u>CorPh Hardware Algorithm (HW-Alg2)</u>
Registers: r & p (two registers to hold n bits).
Input: r,p,n,k; where r ($r= a^{-1}2^{k-n} mod\ p$)& k from *AlmMonInv*
Output: result; where result = $a^{-1}2^{n}\ (mod\ p)$.
11. $j= 2n-k-1$
12. While $j>0$
13. r = ShiftL(r,1); $j = j$-1
14. S1 = Subtract(r, p)
15. if ($S1_{borrow} = 0$) then {r = S1}
16. return result = r

To implement the HW-Alg2 we need: two n-bit registers (to store r and p), a subtractor (step 14), a shifter, and a counter (step 13). The one-bit shifter (step 13) can be easily modified to perform multi-bit shifting and clearly reduce the number of iterations. The ideal situation is to implement HW-Alg2 utilizing the same M.HW-Alg1 hardware components. Since the shift operation in the HW-Alg2 is followed by a subtraction, applying the multi-bit shifting technique to the algorithm demands extra subtractors to perform these operations in parallel and fully speedup the process.

The practical choice of the maximum shifting distance in the *CorPh* implementation is considered to be two. This decision is due to the need of three subtractors when shifting two bits, which are already found in the *AlmMonInv* hardware (assuming two's complement subtraction). If the maximum distance is three, seven subtractors are required, which is far beyond the *AlmMonInv* hardware capability. The *CorPh* algorithm is modified to accommodate the two-bit shifting as shown in the multi-bit shifting *CorPh* hardware algorithm below (M.HW-Alg2).

<u>Multi-Bit Shifting HW-Alg2 (M.HW-Alg2)</u>
Registers: r, u, v & p (all four registers are to hold n bits).
Input: r,p,n,k; where r ($r=a^{-1}2^{k-n} mod\ p$)&k from *AlmMonInv*
Output: result; where result = $a^{-1}2^{n}\ (mod\ p)$.
11. $j = 2n-k-1$
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. S1=Subtract(r,p);S2=Subtract(r,v);S3=Subtract(r,u)
17. if ($S3_{borrow} = 0$) then {r = S3}
18. else if ($S2_{borrow} = 0$) then {r = S2}
19. else if ($S1_{borrow} = 0$) then {r = S1}
20. return result = r

The three subtraction operations are performed in parallel, as step 16 of M.HW-Alg2. Four registers are needed to hold the variables r, u, v and p. The value of p is already available in register p, however, the values of $2p$ and $3p$ have to be computed once at the beginning of the *CorPh* and stored in registers v and u respectively (step 12). The counter, j, is set to $2n-k-1$ at step 11 (using the value k from *AlmMonInv*); it is used to keep track of the number of iterations in the algorithm.

4. THE SCALABLE DESIGN

Application specific hardware architectures are usually designed to deal with a specific maximum number of bits. If this number of bits has to be modified the complete hardware needs to be changed. In addition to that, if the design is implemented for a large number of bits, the hardware is huge and its' clock frequency tends to be very low. These issues motivated the search for a complete scalable (multi-precision) hardware for Montgomery inversion as an extension to what was originally presented by the authors in [7] since it adds the *CorPh* to compute the inverse completely. The scalable architecture solves the previous problems with the following four 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 hardware (flexible use of area). Third, it can handle the computation of numbers in a repetitive way up to a certain limit usually imposed by the size of the memory in the design. If the number of bits in the data exceeds the memory size, the

memory unit may be replaced while the scalable computing unit is not changed. Finally, the number of clock cycles required for an inverse operation depends on the actual size of the numbers used, not on the maximum operand size.

4.1 Scalable Hardware Issues Applied to the Algorithms

Differently from what normally happens in the full-precision hardware design, the scalable hardware, as in [7], has multi-precision operators for shifting, addition, subtraction and comparison. Consider the M.HW-Alg1, for example, the subtraction used for comparison ($u>v$) is performed on a word-by-word (w-bit slices) basis until all the data words (all n bits) are processed, as outlined below:

for $i = 1$ to $\lceil n/w \rceil$

$(x_{borrow}, x_{iw-1:iw-w}) = \text{Subtract}(u_{iw-1:iw-w}, v_{iw-1:iw-w}, x_{borrow})$

$(y_{borrow}, y_{iw-1:iw-w}) = \text{Subtract}(v_{iw-1:iw-w}, u_{iw-1:iw-w}, y_{borrow})$

$(z_{carry}, z_{iw-1:iw-w}) = \text{Add}(r_{iw-1:iw-w}, s_{iw-1:iw-w}, z_{carry})$

Then, the final word borrow out bit is used to decide on the result. Also, depending on the subtraction completion, variable r or s has to be shifted. All variables, u, v, r and s, cannot change until the subtraction processes complete, and the borrow-out bit appears. This forces the use of three more variables: x, y and z; where $x=u-v$, $y=v-u$ and $z=r+s$. These variables are stored in extra registers increasing the number of hardware registers to eight. All the registers hold n_{max} bits even though the actual number of bits in the numbers are $n \leq n_{max}$ bits. This n_{max} limit defines the memory capability and does not degrade the total computation time of the inversion process; i.e., the total delay of the computation depends on the actual number of bits (n) and not on n_{max}.

4.2 Scalable Hardware Design

The scalable hardware design is built of two main parts, a memory unit and a computing unit, as shown in Fig. 3. It is very similar, in principle, to the scalable hardware presented in [7]. 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 actual numbers used may be much smaller than n_{max} bits.

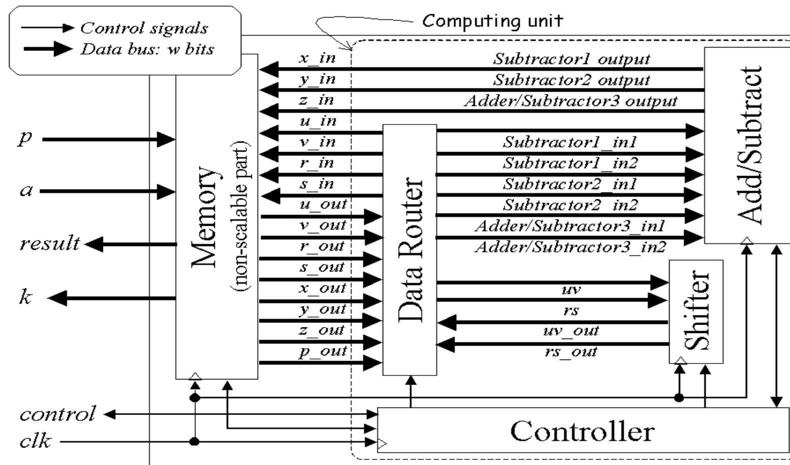

Fig. 3 Montgomery inverse scalable hardware block diagram

The memory unit 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).

The computing unit is made of four hardware blocks, the add/subtract, shifter, data router, and controller block. The add/subtract unit is built of two subtractors, an adder/subtractor, four flip-flops, one multiplexer, a comparator, and logic

gates, connected as shown in Fig. 4. This unit performs one of two operations, either two subtractions and one addition for the M.HW-Alg1, or three subtractions for the M.HW-Alg2. To execute M.HW-Alg1 the Adder/Subtractor3 is controlled to work as an adder (step 4 of M.HW-Alg1). The same Adder/Subtractor3 is used as subtractor to execute step 16 of the M.HW-Alg2. 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 M.HW-Alg1. The borrow-out bits from the subtractors are connected to the controller used only at the end of the each multi-precision addition/subtraction operation. Subtractor 1 borrow-out bit is used to test the condition in step 5 of M.HW-Alg1. It is also essential in electing the result observed in step 10 of M.HW-Alg1. The three subtractors borrow-out bits ($S1_{borrow}$, $S2_{borrow}$, $S3_{borrow}$) are likewise necessary to select the correct 'if' condition to be used in steps 17, 18, or 19, of the M.HW-Alg2 algorithm.

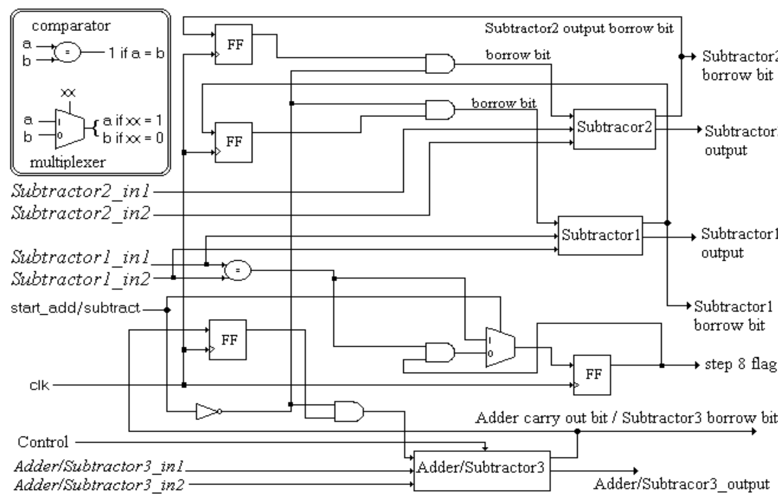

Fig. 4 Add/subtract unit

The multi-bit shifter is made of two multiplexers and two registers with special mapping of some data bits, as shown in Fig. 5. The two multiplexers are used to select the correct set to be used in the multi-bit shifter. Depending on the controller signal *Distance*, the shifter acts as a one, two, or three-bit shifter. Two types of shifting are needed in the M.HW-Alg1 algorithm, right shifting an operand (*u* or *v*) through the *uv* bus (one, two, or three bits) and left shifting

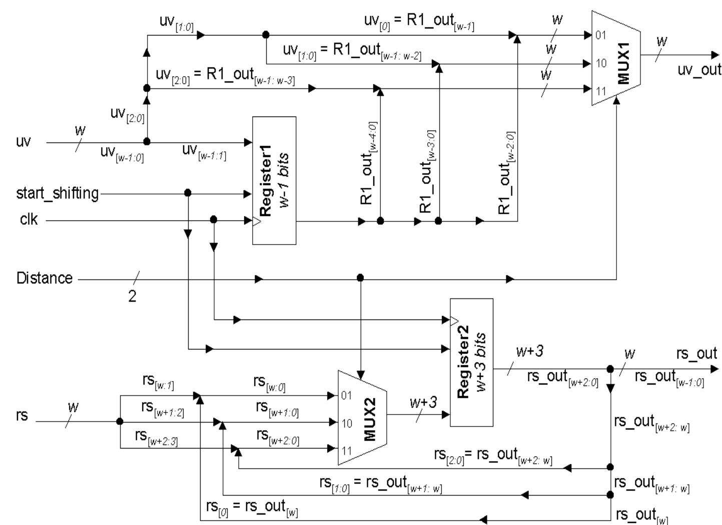

Fig. 5 Multi-bit shifter (max distance = 3)

another operand (*r* or *s*) through the *rs* bus (by similar number of bits). Right shifting *u* or *v* is performed through Register1, which is of size *w*-1 bits. For each word, *w-1* bits of *uv* are stored in Register1. The LS bit(s) of each word is (are) read out immediately as the most significant bit(s) of the output bus *uv_out*. Left shifting *r* or *s* is performed via

Register2, which is of size w+3 bits, in a similar fashion. When executing the M.HW-Alg2, the left shifting is performed to a distance by either one or two bits using the rs path only.

The data router shown in Fig. 3 is made of twelve multiplexers to connect the data going out of the memory unit to the inputs of the add/subtract unit or shifter and also transfers the shifted data values to their destination locations in the memory unit. The possible configurations of the data router are shown in Fig. 6.

The controller is the unit that coordinates the flow of data. It consists of a state machine easily derived from both M.HW-Alg1 and M.HW-Alg2. The controller does not include counters to avoid any dependency on the number of bits (n_{max}) that the system can handle. Such counters are located in the memory block, which is the non-scalable component in the system.

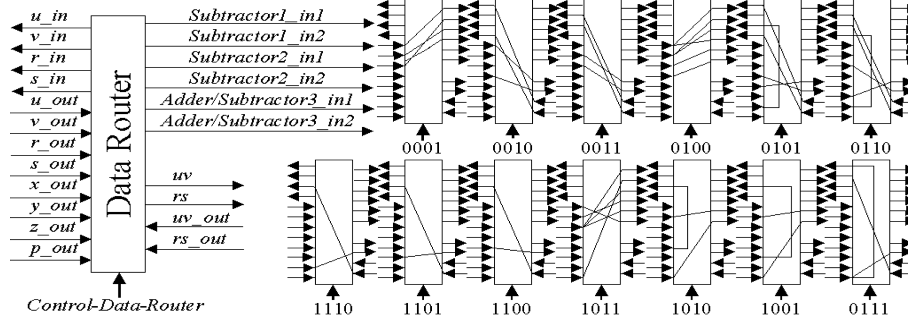

Fig. 6 Data router configurations

5. Modeling and Analysis

The proposed Montgomery inverse scalable design was modeled and simulated in VHDL. It has two main parameters, namely n_{max} and w, which define several hardware configurations. These design configurations are compared in this work with other *fixed designs* previously described in [7]. The *fixed designs*, in brief, are direct fully parallel implementation of HW-Alg1 only parameterized by n_{max} because $w=n_{max}$ in their case.

For both area and speed comparisons, we show the *fixed design* in [7] modified to execute both M.HW-Alg1 and M.HW-Alg2, to be realistic and functionally similar to the scalable hardware of this work. Note that the area presented in [7] is the same given here because modifying the AlmMonInv hardware to process both AlmMonInv and CorPh will increase the area with a negligible amount due to modifying the controller. However, the time of [7] is different than what is here considering executing the complete Montgomery inverse computation. We didn't define a specific architecture for the adders and subtractors used in the designs. Thus, the synthesis tool chooses the best option from its library of standard cells. Since all designs use the same type of adders and subtractors, the comparison is fair.

5.1 Area Comparison

The exact area of any design depends on the technology and minimum feature size. For technology independence, we use the number of equivalent gates as an area measure [14]. A CAD tool from Mentor Graphics (Leonardo) was used. Leonardo takes the VHDL design code and provides a synthesized design 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 [19].

The area of the scalable designs and the fixed one are compared in Fig. 7. As n_{max} increases the difference between the fixed hardware and scalable ones increases, which is expected because of the increasing burden of the computing unit of the fixed design. Observe that the fixed design has larger area than all scalable ones except for the configuration with w=128 and n_{max}<160 bits, because as w approaches n_{max} the scalable design's benefit reduces and the extra hardware used for multi-precision computation shows-up. i.e., the scalable design with $w=n_{max}$ has the same size of adder and subtractors as the fixed one with extra hardware for scalability features, making it more expensive.

5.2 Speed Comparison

The total computation time is the product of the number of clock cycles the algorithm takes and the clock period of the final VLSI implementation. This clock period changes with the value of w in the scalable hardware (Table 2), and changes

with the value of n_{max} in the fixed hardware (Table 3). Tables 2 and 3 lists the clock period for each design obtained from synthesis of the VHDL models.

Table 2 Clock period for scalable designs (nsec)

w	4	8	16	32	64	128
Period	12	14	19	28	47	82

Table 3 Clock cycle period for fixed designs (nsec)

n_{max}	32	64	128	256	512	1024
Period	50	93	178	351	694	1382

The number of clock cycles for all designs depends completely on the data and its computation. For the scalable design, the number of cycles is a function of three parameters: k, w and n. To compute any shifting, addition and/or subtraction, the number of cycles is calculated as $\lceil n/w \rceil$. The total number of clock cycles to execute step 2 or 3 is different than step 4. Step 4 needs extra $\lceil n/w \rceil$ cycles for the shifting operation after it (steps 5 or 6). The average number of clock cycles to perform each iteration of M.HW-Alg1 (step 2 through step 8) is calculated as $CPI_1=(0.5\lceil n/w \rceil)+(0.5(2\times\lceil n/w \rceil)$, (CPI stands for the clock cycles per iteration within the loop: step 2 to 8). The number of iterations of HW-Alg1 is originally equal k, but applying the multi-bit shifting of section 3.2 made the average number of iterations reduce to $0.85k$. An extra $\lceil n/w \rceil$ cycles are needed once after ending the loop of M.HW-Alg1 to perform steps 9 and 10. To sum up, the overall average number of cycles to execute M.HW-Alg1 equals $(CPI_1\times 0.85k)+\lceil n/w \rceil$.

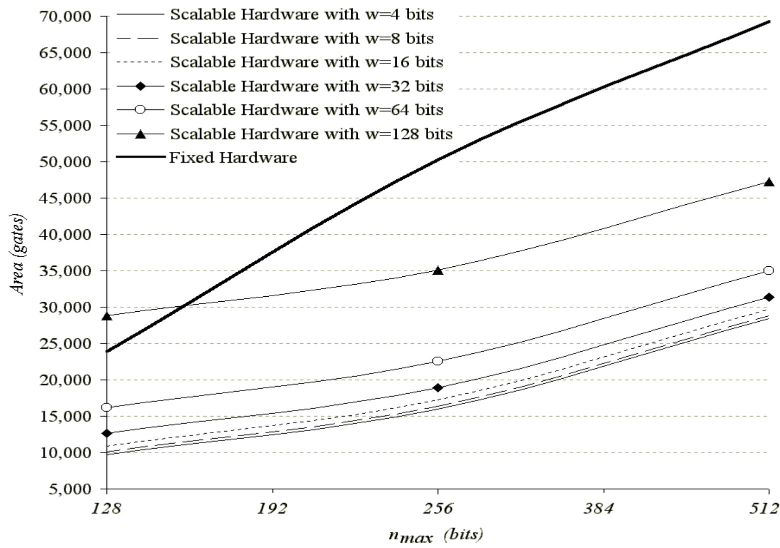

Fig. 7 Area comparison

Similarly, the average number of clock cycles of the scalable hardware to execute M.HW-Alg2 equals to $CPI_2\times(2n-k)/2$; where $CPI_2=2\times\lceil n/w \rceil$ and $(2n-k)/2$ is the average number of iterations shifting two bits per iteration, as explained in section 3.4. The value of k (M.HW-Alg1 and M.HW-Alg2) is within the range $[n,2n]$ [1], which justify the use of its average of $3n/2$, for comparison purposes. The total number of clock cycles required by the scalable design to complete Montgomery inverse computation is then calculated as $C_s=(2.4125n+1)\lceil n/w \rceil$, which was verified by several VHDL simulations.

For the fixed design to perform the CorPh after the AlmMonInv both using multi-bit shifting algorithms as M.HW-Alg1 and M.HW-Alg2, the total average number of clock cycles is $n+0.35k$; where $0.85*k$ cycles are used to execute M.HW-Alg1, and $(2n-k)/2$ cycles are allocated for M.HW-Alg2. If k is approximated to its average of $3n/2$ (similar to the scalable design), the number of the clock cycles will be given by the function $C_f=1.525n$.

Several scalable hardware configurations are designed depending on different n_{max} and w parameters. Each configuration can have different computation time depending on the actual number of bits, n, used. For example, Fig. 8 shows the delay of six scalable hardware designs compared to the fixed hardware, all modeled for $n_{max}=512$ bits, which is a practical number for future ECC applications [11]. 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 the scalable design compared to the fixed one. This is a major advantage of the scalable hardware over the fixed one.

Recall that the number of clock cycles of all designs depends on the actual size of the data used. However, the fixed hardware period always assume to have n_{max} bits to process. i.e., if the application is using $n=128$ bits, and all designs are

made for $n_{max}=512$ bits, as the example of Fig. 8, the fixed design frequency is not affected by n and all n_{max} bits are treated in the computation causing the fixed design to have a total time greater than all different scalable ones. This observation is found valid for all different n_{max} designs built and tested, which generalized the fact that all scalable designs are faster than the fixed one while

$$n < \begin{cases} (\log_2 w-1)n_{max}/4 & when\ \ w < n_{max}/16 \\ n_{max} & when\ \ w \geq n_{max}/16 \end{cases}$$

See Fig. 8 for example, as $n<n_{max}/2$ ($n=256$) the fixed hardware is faster than the scalable one with $w=4$ bits and very similar to the one with $w=8$ bits. As $n>3n_{max}/4$ ($n=384$) the scalable design with $w=16$ speed falls below the fixed one. When $n=n_{max}=512$ the scalable design with $w=32$ bits has almost the same speed as the fixed one, but the ones with $w>n_{max}/16$ bits remain faster. In fact, as w gets bigger the total time decreases, which is also true when comparing among the different scalable designs as long as $n\geq w$ (Fig. 8). Whenever $n<w$ considering the scalable designs only, the scalability advantage of the designs reduces indicating that the number of words to be processed reached its lower limit, but still the scalable designs are faster than the fixed one.

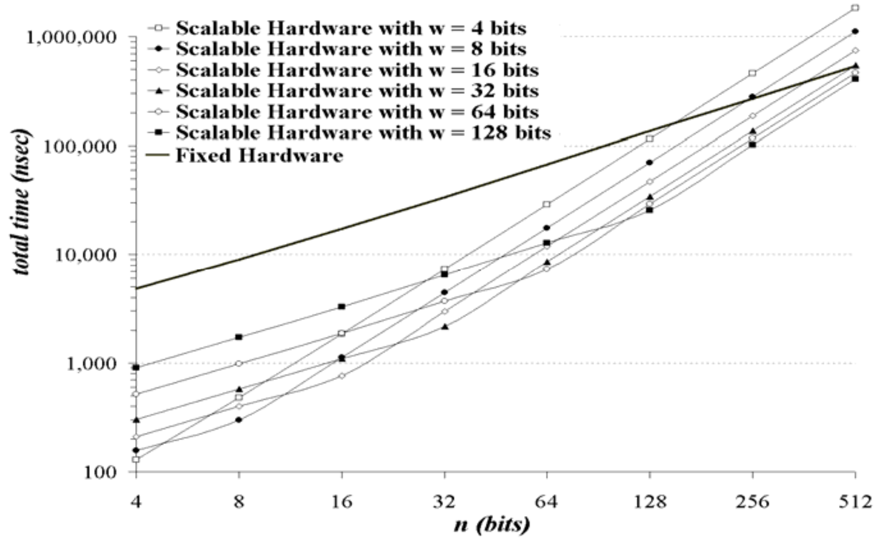

Fig. 8 Delay comparison of designs with n_{max} = 512 bits

6. CONCLUSION

This paper presents a scalable VLSI hardware implementation of a new procedure proposed for the computation of Montgomery modular inverse in hardware. The procedure used a previously published almost Montgomery inverse algorithm followed by a new correction phase, which resulted in the fastest approach to compute Montgomery inverse when compared with several other Montgomery inverse computation methods.

Our Montgomery inverse procedure was further improved by the introduction of multi-bit shifting instead of single bit shifting, which is very attractive for hardware implementations. The proposed architecture is scalable allowing a specific computing module to handle operands of any precision. The word-size that the module operates can be selected depending on the area and performance requirements. The maximum limit (n_{max}) on the operand precision of the entire inverter hardware is limited only by the available memory to store the operands and internal results. If the operand precision exceeds the memory size, the memory unit is the only part that needs to be modified, while the scalable computing unit does not change.

The scalable VLSI architecture was compared to a fully parallel fixed hardware. The scalable design showed area flexibility, depending on the number of bits used at each clock cycle (w), as w increase the scalable hardware area increase. Choosing $w=4$ bits (as smallest scalable design) and $n_{max}=512$ bits, the area of the scalable design is *60%* less than the fixed hardware. The speed, however, of this scalable hardware depends on the actual number (n) of bits used; if $n\leq n_{max}/4$, the scalable design is faster than the fixed one. The clock cycle period required to execute the algorithm on the scalable hardware relies on w, which is not the case for the fixed hardware. The comparisons show that our scalable structure is very attractive for cryptographic systems, particularly for ECC where there is a clear need for modular inversion of large numbers, which may differ in size depending on security requirements imposed by applications.

ACKNOWLEDGMENTS

The authors are thankful to the Information Security Laboratory researchers at Oregon State University. This research received financial support from King Fahd University of Petroleum & Minerals (KFUPM), Saudi Arabia, and NSF career grant 93434-ccr-"Computer Arithmetic Algorithms and Scalable Hardware Designs for Cryptographic Applications".

REFERENCES

[1] Savas, and Koç, "The Montgomery Modular Inverse – Revisited", *IEEE Trans. on Computers*, 49(7): 763-766, July 2000.
[2] Kobayashi, and Morita, "Fast Modular Inversion Algorithm to Match Any Operation Unit", *IEICE Trans. Fundamentals*, E82-A(5):733-740, May 1999.
[3] Kaliski, "The Montgomery Inverse and its Applications", *IEEE Trans. on Computers*, 44(8):1064-1065, Aug. 1995.
[4] Rivest, Shamir, and Adleman, "A Method for Obtaining Digital Signature and Public-Key Cryptosystems", Comm. ACM, 21(2):120-126, Feb. 1978.
[5] Diffie, and Hellman, "New Directions on Cryptography", *IEEE Trans. on Information Theory*, 22:644-654, Nov. 1976.
[6] Tenca, and Koç, "A Scalable Architecture for Montgomery Multiplication", *In Cryptographic Hardware and Embedded Systems*, no. 1717 in Lecture notes in Computer Science, Springer, Berlin, Germany, 1999.
[7] Adnan Abdul-Aziz 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.
[8] Charles J. Stone, *A course in probability and statistics*, Duxbury Press, Belmont, 1996.
[9] Chung, Sim, and Lee, "Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor", *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000*, Massachusetts, Aug. 2000.
[10] Atsuko Miyaji, "Elliptic Curves over F_P Suitable for Cryptosystems", *Advances in cryptology- AUSCRUPT'92*, Australia, Dec. 1992.
[11] Blake, Seroussi, and Smart, *Elliptic Curves in Cryptography*, Cambridge University Press: New York, 1999.
[12] Hankerson, Hernandez, and Menezes, "Software Implementation of Elliptic Curve Cryptography Over Binary Fields", *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000*, Massachusetts, Aug. 2000.
[13] Kovac, M., Ranganathan, N. and Varanasi M., "SIGMA: A VLSI Systolic Array Implementation of Galois Field $GF(2^m)$ Based Multiplication and Division Algorithm", *IEEE Trans. on VLSI*, 1(1):22-30, March 1993.
[14] Ercegovac, M. D., Lang, and Moreno, J., *Introduction to Digital System*, John Wiley & Sons, Inc., New York, 1999.
[15] Montgomery, P.L., "Modular Multiplication Without Trail Division", *Mathematics of Computation*, 44(170): 519-521, April 1985.
[16] Naofumi Takagi, "Modular Inversion Hardware with a Redundant Binary Representation", *IEICE Transactions on Information and Systems*, E76-D(8): 863-869, Aug. 1993.
[17] Guo, J.-H., and Wang, C.-L., "Hardware-Efficient Systolic Architecture for Inversion and Division in $GF(2^m)$", *IEE Proceedings: Computers and Digital Techniques*, 145(4): 272-278, July 1998.
[18] Choudhury, P. Pal., and Barua, R., "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.
[19] http://www.mentor.com/partners/hep/AsicDesignKit/dsheet/ami05databook.html, Mentor Graphics Co., *ASIC Design Kit*.
[20] Hasan, M. A., "Efficient Computation of Multiplicative Inverse for Cryptographic Applications", *Proceeding of the 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, June 11-13 2001.
[21] Guo, J.-H., and Wang, C.-L., "Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in $GF(2^m)$", *IEEE Trans. on Computers*, 47(10): 1161-1167, Oct. 1998.
[22] Fenn, S. T. J., Benaissa, M., and Taylor, D., "$GF(2^m)$ Multiplication and Division Over the Dual Basis", *IEEE Trans. on Computers*, 45(3): 319-327, March 1996.
[23] Wang, C. C., Truong, T. K., Shao, H. M., Deutsch, L. J., Omura, J. K., and Reed, I. S., "VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$", *IEEE Trans. on Computers*, C-34(8): 709-717, Aug. 1985.
[24] Feng, G.-L., "A VLSI Architecture for Fast Inversion in $GF(2^m)$", *IEEE Trans. on Computers*, 38(10):1383-1386, Oct. 1989.

Adnan Abdul-Aziz Gutub received his Ph.D. degree in June 2002 from the Department of Electrical and Computer Engineering at Oregon State University. He received his B.S. (1995) degree from the Electrical Engineering Department and M.S. (1998) degree from the Computer Engineering Department in King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia. Adnan's research interests are in modeling, simulating, and synthesizing VLSI hardware for computer arithmetic operations. Adnan is currently an assistant professor in the Computer Engineering Department at King Fahd University of Petroleum & Minerals in Saudi Arabia.

Alexandre F. Tenca received the B.S. and M.S. degrees in electrical engineering from University of Sao Paulo (USP) - Escola Politecnica (Polytechnic School), Brazil, in 1981 and 1990, and the M.S. and Ph.D. degrees from UCLA - Computer Science Department, in 1994 and 1998, respectively. Upon completion of his doctorate he became an assistant professor of computer engineering at the Oregon State University, Electrical and Computer Engineering Department. His research interests include computer architecture, computer networks, computer arithmetic and reconfigurable systems.