EFFICIENT SCALABLE HARDWARE ARCHITECTURE FOR MONTGOMERY INVERSE COMPUTATION IN GF(P)

Adnan Abdul Aziz Gutub

Computer Engineering Department
King Fahd University of Petroleum & Minerals
Dhahran 31261, SAUDI ARABIA
Email: gutub@kfupm.edu.sa

Alexandre Ferreira Tenca

Electrical & Computer Engineering Department
Oregon State University
Corvallis, Oregon 97331, USA
Email: tenca@ece.orst.edu

ABSTRACT

The Montgomery inversion is a fundamental computation in several cryptographic applications. In this work, we propose a scalable hardware architecture 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. The intended architecture 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.

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) [1,9-12,20-25]. This work is targeted mainly toward the use of ECC because of its promise to replace older public-key cryptographic systems [9-12, 20]. ECC arithmetic consists mainly in modular computations of addition, subtraction, multiplication, and inversion.

Inversion is well known to be the lowest computation among all other arithmetic calculations in ECC [1, 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 the projective coordinate points back to affine coordinates. However, if this single inversion is not fast enough, it will cause the complete ECC system to be slow.

Speed and security are the driving trends to do inversion in hardware instead of software [16-18]. It is more secure, practical and fast to have all the computations handled in hardware, inside a VLSI IC-chip, instead of mixing some computations performed in software with others processed in hardware [6, 16-18, 20-25].

Modular inversion is often performed by algorithms based on the Extended Euclidean algorithm [11]. Several inversion designs are described in the literature [4, 16-18, 20-25]. Most of them [17, 18, 20-25] are for inversion in Galois Fields GF(2^k), which are out of the scope of 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 requires two main operations and in this work we suggest replacing one of them with a simpler correction phase, 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 discussed in [4]. 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 [4], similar in principle to the scalable multipliers presented in [6-8]. Our scalable inverter gives various practical results to show how attractive this contribution is.

In the coming section, Section 2, the reason behind choosing Montgomery modular method is briefly described. Section 3 presents the Montgomery inverse algorithm including the correction phase proposed in this work. In Section 4 the scalable hardware procedure and VLSI implementation is presented. The comparison between several different hardware implementations is given in Section 5 followed by the conclusion, Section 6.

2. 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 [13]. 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 an 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, 6, 7, 15]. 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, some researchers propose to have dedicated procedures to compute the modular inverse in the Montgomery domain, i.e., Montgomery modular inverse algorithms [1,2].

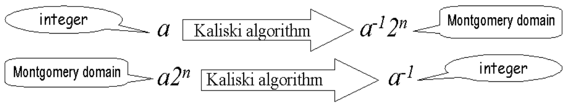
3. MONTGOMERY INVERSE ALGORITHM AND PROPOSED MODIFICATIONS

A modified Montgomery modular inverse algorithm is found in the literature [1]. It modifies a technique proposed by Kaliski in 1995 [2], to make it more suitable and faster for cryptography using Montgomery's idea. The Montgomery inverse function is to calculate $x = a^{-1}2^n \bmod p$ from $a2^n$. Kaliski method however, takes an integer a and produces $x = a^{-1}2^n \bmod p$. 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. It is followed by a very brief explanation of the modification to make it compute the Montgomery inverse (input and output in the Montgomery domain) and to make it faster. We then propose a new correction phase that results in further speedup in hardware implementations.

3.1 Kaliski Algorithm

Kaliski algorithm [1, 2] 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 $r=a^{-1}2^k \bmod p$, and $n < k < 2n$. Phase II takes the outputs of Phase I as its inputs, and gives the final result $x=a^{-1}2^n \bmod p$, where $2^{n-1} \leq p < 2^n$. Note that in both phases values of a and $x \in [1, p\text{-}1]$.



$$\text{integer}\ a \xrightarrow{\text{Kaliski algorithm}} a^{-1}2^n\ \text{Montgomery domain}$$
$$\text{Montgomery domain}\ a2^n \xrightarrow{\text{Kaliski algorithm}} a^{-1}\ \text{integer}$$

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

<u>**Phase I: Almost Montgomery Inverse, *AlmMonInv*(*a*)**</u>

```
Input:  $a$ & $p$; where $a$ is in the range [1, $p$-1].
Output: $r$ & $k$; where $r = a^{-1}2^k \bmod p$ & $n < k < 2n$
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 $r = p-r$
```

<u>**Phase II**</u>

```
Input: $p, r=a^{-1}2^k \bmod p, k$ & $n$; where $r$ & $k$ from PhaseI
Output: $x$; where $x = a^{-1}2^n \bmod p$
1.  for $i = 1$ to $k - n$ do
2.      if $r$ is even then $r = r/2$
3.      else $r = (r + p)/2$
4.  return $x = r$
```

3.2 Modification to Kaliski Algorithm

In July 2000, Savas and Koç [1] proposed to replace Phase II of Kaliski's algorithm with Montgomery multiplication, which resulted in a faster execution process for software implementations on general-purpose microprocessors. They presented a complete Montgomery modular inverse algorithm using two main procedures, the *almost Montgomery inverse* (AlmMonInv) and the *Montgomery product* (MonPro). This modification was targeted toward software implementations. However, the work in [1] acted as the seed of our study of a correction phase that completely eliminates the need for MonPro in the inversion computation.

3.3 New Approaches for Montgomery Inverse

Let's consider the main Montgomery inverse problem again. An approach to calculate $x=a^{-1}2^n \bmod p$ from $a2^n$ can be to compute a first and then calculate the *AlmMonInv* (*Kaliski Phase I*) followed by *Kaliski Phase II* to get the desired inverse result. The first computation of a from $a2^n$ is performed by a modular division by 2^n named *Preparation Phase* as shown below.

<u>**Preparation Phase (Divide by 2^n)**</u>

```
Input:  $r = a2^n, n$ & $p$; where $p$=modulus & $2^{n-1} \leq p < 2^n$
Output:   $x$; where $x = a \bmod p$
1.  for $i = 1$ to $n$ do
2.      if $r$ is even then $r = r/2$
3.      else $r = (r + p)/2$
4.  return $x = r$
```

Note that calculating a from $a2^n$ is obtained in [1] by a Montgomery product (MonPro) as: $a2^n\ (2^{-n}) \bmod p = a \bmod p$. However, we preferred the preparation phase to using MonPro because it clearly can be implemented utilizing the same hardware components of the AlmMonInv.

Another new way to calculate the Montgomery inverse is by applying the *AlmMonInv* on the input $a2^n$ to produce r and k according to the formula: $(r,k) = AlmMonInv\ (a2^n)$
where: $r = (a2^n)^{-1}2^k \bmod p = a^{-1}2^{k-n} \bmod p$

Recall that Montgomery inverse of $a2^n$ is $a^{-1}2^n \bmod p$, which implies that the *AlmMonInv* result ($a^{-1}2^{k-n} \bmod p$) must be corrected. It is possible to find a constant C such that:

$$C \times (a^{-1}2^{k-n} \bmod p) = a^{-1}2^n \bmod p.$$

Applying some algebra we get: $C = (a^{-1}2^n \bmod p)/(a^{-1}2^{k-n} \bmod p)$
$= (a^{-1}2^n)/(a^{-1}2^{k-n}) = (2^n)/(2^{k-n}) = 2^{n-(k-n)} = 2^{2n-k}$
multiplying ($a^{-1}2^{k-n} \bmod p$) by (2^{2n-k}) in a modular fashion can be performed as the following:

$$([(((((a^{-1}2^{k-n})\underbrace{.2).\ 2).\ 2).\ 2)\ldots\ldots\ldots\ldots.\ 2).\ 2)]}_{2n\text{-}k\ times}\ \bmod p) = a^{-1}2^n \bmod p$$

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:

$$[(((((a^{-1}2^{k-n}).2) \bmod p).2)\ldots2) \bmod p).2) \bmod p)] = a^{-1}2^n \bmod p$$

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 $x = a^{-1}2^n \bmod p$
1.  for $i = 2n-k$ to $0$ do
2.      $r = 2r$
3.      if $r > p$ then $r = (r - p)$
4.  return $x = r$
```

3.4 Evaluation of Alternatives

Several methods considered for hardware computation of the Montgomery inverse are shown in Fig. 2; including the procedures proposed 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].

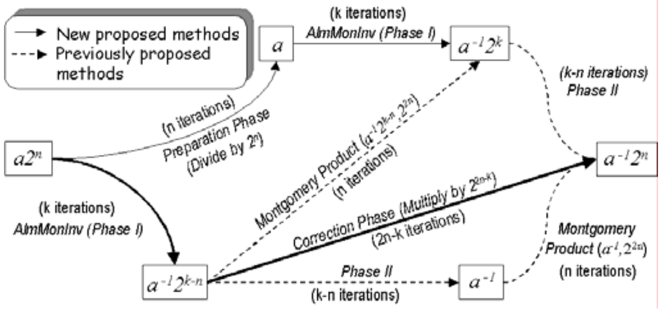


Fig. 2. Ways to compute the Montgomery inversion

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].

Observe that the other approach proposed in Section 3.3 (Fig. 2 path: Preparation Phase, *AlmMonInv* and Kaliski Phase II) would require *3n* iterations to complete the inversion; it is the slowest alternative and for this reason will not receive further attention. For the previously proposed methods using *MonPro* multipliers [1], even if the multipliers are completely parallel (one iteration instead of n), they need more than *2n* iterations, which is still slower than using our new two-phase method.

4. THE SCALABLE DESIGN

4.1 Why 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 increased, even by one bit, the complete hardware needs to be replaced. What's more, 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 [4] since it adds the CorPh to compute the inverse completely (no need for MonPro).

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.

4.2 Scalable Hardware Issues Applied to Algorithms

The AlmMonInv algorithm, when observed from hardware point-of-view, contains operations that are easily mapped to hardware. 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 shown in 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 ($v \neq 0$) 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}
```

Consider step 6 of the AlmMonInv algorithm (HW-Alg1), if $u>v$ then a subtraction (u-v) takes place. Otherwise, as in step 7, the subtraction of (v-u) is calculated. For the worst case, two subtraction operations are performed, because the comparison of u and v is also performed through subtraction of u and v. Note that (v-u) can be generated from (u-v) by the use of an adder module to compute the two's complement. We rather perform these two subtractions in parallel, as step 4 of HW-Alg1 (S1 and S2 stands for the subtraction results signals). The same case applies to step 9 and step 10 of *AlmMonInv* algorithm, both subtractions are performed in parallel in the HW-Alg1.

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

CorPh Hardware Algorithm (HW-Alg2)

```
Registers: r & p (two registers to hold n bits).
Input:    r,p,n,k; where r ($r = a^{-1}2^{k-n} \bmod p$)& k from AlmMonInv
Output: result; where result = $a^{-1}2^{n} \pmod p$.
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
```

4.3 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 [4]. 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 way smaller than n_{max} bits.

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 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. It worth's mentioning that 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.

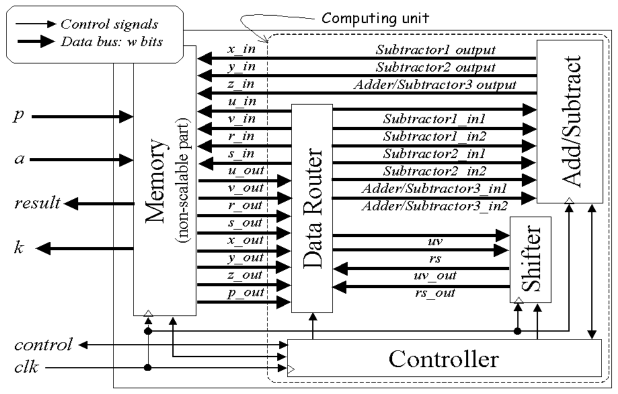


Fig. 3. Montgomery inverse scalable hardware block diagram

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 [4]. The *fixed designs*, in brief, are direct fully parallel implementation only parameterized by n_{max} because w=n_{max} in their case.

For both area and speed comparisons, we show the *fixed design* in [4] modified to execute the same Montgomery inverse algorithm proposed here to be realistic and functionally similar to the scalable hardware of this work. Note that the area presented in [4] is the same given in this paper because modifying the AlmMonInv hardware to process both AlmMonInv and CorPh will increase the area with a negligible amount due to modifying the controller only. However, the time of [4] 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 [13]. 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. 4. 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.

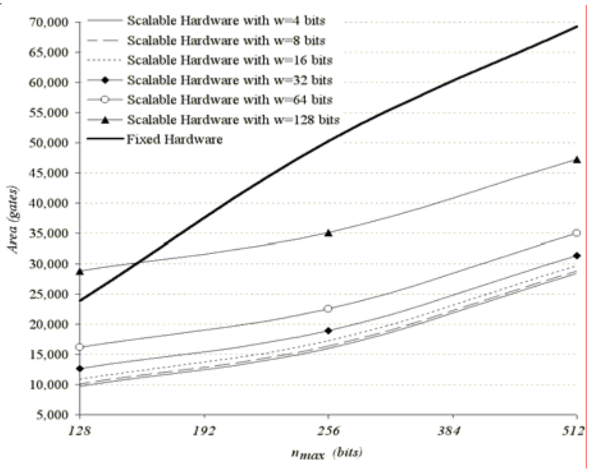


Fig. 4. Area comparison

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 1), and changes with the value of n_{max} in the fixed hardware (Table 2). Tables 1 and 2 lists the clock period for each design obtained from synthesis of the VHDL models.

Table 1 Clock cycle period for scalable designs (nsec)

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

Table 2 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 (HW-Alg1) 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 HW-Alg1 (step 2 through step 10) is calculated as $(\text{CPI}\times 0.85k)+\lceil n/w \rceil$; where $\text{CPI}=(0.5\lceil n/w \rceil)+(0.5(2\times\lceil n/w \rceil)$, (CPI stands for the clock cycles per iteration within the loop: steps 2 to 8). Similarly, the average number of clock cycles of the scalable hardware to execute HW-Alg2 equals to $2\times\lceil n/w \rceil\times(2n\text{-}k)/2$. The total number of clock cycles required by the scalable design to complete Montgomery inverse computation is 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 using HW-Alg1 and HW-Alg2, the total average number of clock cycles is $n+0.35k$; where $0.85*k$ cycles are used to execute HW-Alg1, and $(2n\text{-}k)/2$ cycles are allocated for HW-Alg2. If k is approximated to its average of $3n/2$ (similar to the scalable design [4]), 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. 5 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. 5, 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

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

See Fig. 5 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. 5). 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.

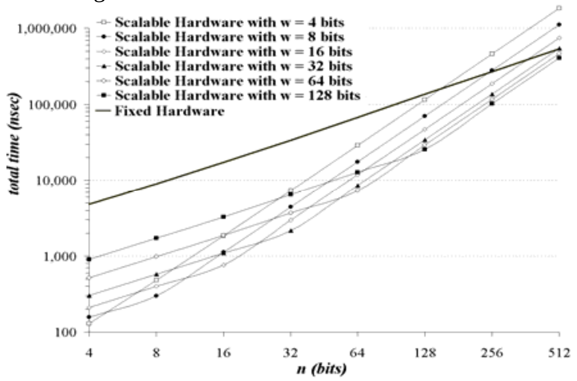


Fig. 5. Delay comparison of designs with n_{max} = 512 bits

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.

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.

7. ACKNOWLEDGMENTS

This research received financial support from King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia.

8. REFERENCES

[1] Savas, and Koç, "The Montgomery Modular Inverse – Revisited", *IEEE Trans. Computers*, 49(7):763-766, July 2000.

[2] Kaliski, "The Montgomery Inverse and its Applications", *IEEE Trans. on Computers*, 44(8):1064-1065, Aug. 1995.

[3] Rivest, Shamir, and Adleman, "A Method for Obtaining Digital Signature and Public-Key Cryptosystems", Comm. ACM, 21(2):120-126, Feb. 1978.

[4] Adnan Abdul-Aziz Gutub, A. Tenca, and C. Koç, "Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation", *ISVLSI 2002: IEEE Computer Society Annual Symposium On VLSI*, Pittsburgh, Pennsylvania, April 2002.

[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] Savas, Tenca, and 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.

[8] Tenca, Todorov, and Koç, "High-Radix Design of a Scalable Modular Multiplier", *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2001*, France, May 14-16 2001.

[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] Tocci, and Widmer, "*Digital Systems: Principles and Applications*", Eighth Edition, Prentice-Hall Inc., NJ 2001.

[14] Charles J. Stone, *A course in probability and statistics*, Duxbury Press, Belmont, 1996.

[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] Mentor Graphics Co., *ASIC Design Kit*, http://www.mentor. com/partners/hep/AsicDesignKit/dsheet/ami05databook.html

[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, and Wang, "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, Benaissa, and Taylor, "GF(2^m) Multiplication and Division Over the Dual Basis", *IEEE Trans. on Computers*, 45(3): 319-327, March 1996.

[23] Wang, Truong, Shao, Deutsch, Omura, and Reed, "VLSI Architectures for Computing Multiplications and Inverses in GF(2^m)", *IEEE Trans. on Computers*, C-34(8):709-717, Aug. 1985.

[24] Feng, "A VLSI Architecture for Fast Inversion in GF(2^m)", *IEEE Trans. on Computers*, 38(10):1383-1386, Oct. 1989.

[25] Kovac, Ranganathan, and Varanasi, "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.