# High Speed Hardware Architecture to Compute GF(p) Montgomery Inversion with Scalability Features

*Adnan Abdul-Aziz Gutub*

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

**Abstract:**

Modular inversion is a fundamental process in several cryptographic systems. It can be computed in software or hardware, but hardware computation has been proven to be faster and more secure. This research focused on improving an old scalable inversion hardware architecture proposed in 2004 for finite field GF(p). The architecture comprises two parts, a computing unit and a memory unit. The memory unit holds all the data bits of computation whereas the computing unit performs all the arithmetic operations in word (digit) by word bases such that the design is scalable.

The main objective of this paper is to show the cost and benefit of modifying the memory unit to include shifting, which was previously one of the tasks of the scalable computing unit. The study included remodeling the entire hardware architecture removing the shifter from the scalable computing part and embedding it in the non-scalable memory unit instead. This modification resulted in a speedup to the complete inversion process with an area increase due to the new memory shifting unit. Several design schemes have been compared giving the user the complete picture to choose from depending on the application need.

## 1 Introduction

Modular inverse arithmetic is an essential arithmetic operation in public-key cryptography. It is a basic operation in the elliptic curve cryptography (ECC), which is the main focus of this work because of its promise to replace older public-key cryptographic systems [1]. ECC arithmetic consists of mainly modular computations: addition, subtraction, multiplication, and inversion.

Inversion is well known to be the slowest computation among all other arithmetic calculations in ECC [2, 3]. Many researchers propose minimizing the use of modular inversion by adopting elliptic curves defined for projective coordinates [1, 4], 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 inversion is not fast enough, it will cause the complete ECC system to be slow.

A fast modular inverse calculation is the main reason to do inversion in hardware instead of software [5, 6]. 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 going to the projective coordinate systems [3]. Even if the speed to compute the inverse is not that good to justify the use of affine coordinates, the computation with hardware is still faster than software [7, 8, 9, 10, 11, 12], which will provide better performance for the overall cryptographic system based on projective coordinates.

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 [1]. Several inversion VLSI designs are described in the literature [5, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Most of them [7, 8, 9, 10, 11, 12] are for inversion in Galois Fields GF($2^k$). Several [8, 9, 10, 11, 12] 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 [7] 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 [7, 8, 9, 10, 11, 12] have no direct link to this work. It is to be noted that GF(p) inversion architecture area decrease is traded for speed. And indeed, speed is one of the motivations for doing such a work as this scalable hardware.

Previously, Takagi in 1993 [5], 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. In 1998, Takagi [6] modified a binary method for calculating the greatest common divisor (GCD) using the plus-minus algorithm. This plus-minus algorithm is extended for performing modular division using similar redundant binary representation. Takagi's work in [5] and [6], however, required more area than the design proposed here and also needs data transformations that are usually expensive and time consuming.

Zhou *et al.* [14, 15], designed a VLSI implementation for GF(p) inversion computation using one simple adder. Zhou's hardware suffers from the long propagation carry chain which made the operation clock frequency limited and the design area and complexity not flexible to accommodate the changing demand of the crypto applications.

Several attempts [13, 16, 17, 18] have investigated the GF(p) inversion targeted to field programmable gate array (FPGA) implementations. Fiaz *et al.* [18] described an FPGA divider which can be used for inversion by representing the dividend by '1'. Daly *et al.* [13] and Dormale *et al.* [17] shortened the critical path and carry-chain addition within the inversion process according to the FPGA column limitations. The designs minimize the extra delay of the top to bottom carry chain mapping between different FPGA columns by specific physical routing. Daly *et al.* [13] presented an architecture technique "for implementation on any FPL (field programmable logic) device which has dedicated carry logic capability". Dormale *et al.* [17] presented an FPGA carry conditional adder implementation that demonstrates improvement especially when the carry-chain exceeds the specific FPGA column height.

An ECC arithmetic hardware unit has been proposed by Feldhofer *et al.* [19]. It contained asynchronous modules to compute all prime field computations including inversion. The inversion hardware was slow and complex depending on Fermat's Theorem. Feldhofer idea did not consider the inversion as a main problem, assuming all computations are performed through projective coordinates, and the inversion was calculated by a number of multiplications. The inversion process needed multiplication operations that can be equal to double the number of bits of the data length used.

McIvor *et al.* [16] improved the inversion algorithm presented by Savas *et al.* [2] involving modular multiplication. McIvor reduced the number of needed multiplication operations to speedup the complete computation [16]. McIvor benefited from the built-in carry-look-ahead adders on the FPGA to gain in area reduction.

Tawalbeh *et al.* [20] presented a unified inversion hardware for both GF(p) and GF($2^k$). Their unified design used a scheduling method to reduce the number of hardware resources without significantly increasing the total execution time. They replaced all comparisons hardware by the use of counters to keep track of the difference between field elements which are usually expensive and time-consuming. The counters, also, limited the flexibility of scalability which made their work suitably far from our main focus of this work.

The GF(p) modular inverse problem 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 $x \in [1, p\text{-}1]$. It is normally represented as $x = a^{-1} mod\ p$ [2]. The modular inverse algorithm and hardware suitable for this research is related to Montgomery multiplication as presented in [21, 22]. Montgomery algorithm is originally proposed to simplify modular multiplication [23]. It replaces the normal division with divisions by two, which is performed in the binary number representation (shifting the binary representation of a number one bit to the right). 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 [2, 3]. The Montgomery inverse algorithm of this work is implemented in hardware using scalability features, which allows the use of a fixed-area scalable circuit to perform inversion of unlimited precision operands. The hardware divides the long-precision numbers in words and each word is processed in a clock cycle.

This research aimed to investigate the possibility of speeding the process by modifying the registers of the non-scalable part to incorporate the shifting operation. The shifting operation will be part of the memory unit instead of the scalable computing unit. This feature is predicted to reduce the shifting operation delay which will improve the total computation performance. Therefore, the main objective has been to investigate the advantages and disadvantages behind this modification and its practicality to be implemented. The measurements of the trade-off between speed and area, and the hardware modification criteria will be expressed. The results will show the speedup gained and the extra hardware area needed. The conclusion will indicate whether the extra hardware is worth the expected speedup with quantitative measurements.

In the following section, the original scalable hardware is introduced. Section 3 is concerned about the modification to the original hardware and the algorithms used. Section 4 details the speed approximation method adopted. Section 5 compares all the designs schemes involving the old ones. Section 6 concludes the paper and summarizes the results.

## 2 Original Scalable Inversion Hardware

The original scalable inversion hardware is presented in [21]. It is built of two main parts, a memory unit and a computing unit, as shown in Fig. 1. The memory unit is not scalable because it has a limited storage defined by the value of $n_{max}$ (the maximum number of bits to be handled by the hardware). The data values of $a$ and $p$ are first loaded into 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 carries on through the computation iterations until the controller indicates that all the words of the operands 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, $u$, $v$, $r$, $s$, $x$, $y$, $z$ 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. All these blocks functions and hardware design are detailed in [21, 22]. Our focus of this research is about the *shifter*. The original shifter is made of two multiplexers and two registers with special mapping of some data bits, as shown in Fig. 2. 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, as clarified in [22]. Two types of shifting are needed in the inversion algorithm, right shifting an operand (*u* or *v*) through the *uv* bus (one, two, or three bits) and left shifting 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 least significant (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.

The modification in this work is to redesign both the scalable and non-scalable hardware units. The shifter will be removed from the computation unit. It will be embedded into the non-scalable memory unit as clarified in the next section.

## 3 Proposed Hardware and Algorithms

Several methods can be used to compute the Montgomery inverse as detailed in [24]. The most efficient technique [24] uses two-phases, the *Almost Montgomery Inverse (AlmMonInv)* followed by the *correction phase (CorPh)*. Assuming *n* as the operand number of bits, the process requires *2n* iterations to complete the inversion, the *AlmMonInv* needs *1.5n* iterations, and the *CorPh* needs *0.5n* iterations, assuming an average value of *k=1.5n*, as detailed in [22].

Two hardware algorithms of the *AlmMonInv* procedure are shown in [22], depending on the number of bits of shifting used. We will start this study by single bit shifting since the shifter will be eliminated from the computing unit. The *AlmMonInv* algorithm of single-bit shifting is shown 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 $\ne$ 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}
```

The *CorPh* [22] algorithm is shown as 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
```

The hardware is modified as shown in Fig. 3. The memory block is improved to perform shifting by adding $n_{max}$ multiplexers to each FIFO. The multiplexers will reroute the data to one of four options as shown in Fig. 4. The multiplexers can control passing the data to it self or to the next cell for FIFO shifting as in the original architecture, or it directs the data to shift right or left. The memory & shifter unit will follow the operations as needed by the HW-Alg1 and HW-Alg2.

## 4 Speedup Estimates of Proposed Scalable Design

The study will concentrate on the *AlmMonInv* algorithm first, then the *CorpPh* one, similar to our old scalable design presented in [22]. The new hardware speed will be estimated depending on the number of bits to be shifted through the non scalable 'memory & shifter' unit.

### *4.1 AlmMonInv Single Bit Shifting*

The number of clock cycles for all designs depends completely on the data and its computation. The computation time of the new hardware to run the AlmMonInv algorithm is estimated by a probability study as in [22]. See the AlmMonInv algorithm (HW-Alg1) represented earlier. Simulating this algorithm proved that almost 25% of the *k* cycles is consumed by step 2 and 25% is for step 3. Steps 4, 5, and 6 are a sequence that runs consuming 50% of the *k* iteration. After the *k* iterations, step 9 is performed once which needs to considered in the time estimation too. Note that each shifting operation is performed in one cycle independent to the number of words the hardware is having, while the addition and subtraction needs to be performed within $\lceil n/w \rceil$ cycles. These points made the AlmMonInv Computation Time as follows:

*Cycles for steps 4,5,6* $= 0.5\, k\, (\lceil n/w \rceil + 1)$
*Cycles for step 9* $= \lceil n/w \rceil$
*Cycles for steps 2,3* $= 0.5\, k$
*Total AlmMonInv Cycles* $= 0.5\, k\, (\lceil n/w \rceil + 1) + \lceil n/w \rceil + 0.5\, k$

### *4.2 AlmMonInv Two Bit Shifting*

The new hardware, shown in Figure 4, is improved to have its memory & shift unit to perform two bit shifting in addition to its original ability of single bit shifting. The shifting can now be performed as one bit shifting right, one bit shifting left, two bits shifting right, and two bits shifting left. This will modify the routing multiplexer inserted between the memory cells increasing the multiplexer size.

The AlmMonInv computation time will be similar to the single bit shifting except in steps 2, and 3, which will be reduced by 6% each. The overall time reduction of steps 2, and 3, together is estimated by 12%, as described in detail in [22]. This two bit shifting made the AlmMonInv computation time as follows:

*Cycles for steps 4,5,6 =* $0.5\,k\,(\lceil n/w \rceil+1)$
*Cycles for step 9 =* $\lceil n/w \rceil$
*Cycles for steps 2,3 =* $0.38\,k$
*Total AlmMonInv Cycles =* $0.5\,k\,(\lceil n/w \rceil+1)+\lceil n/w \rceil+0.38\,k$

## 4.3 AlmMonInv Multi Bit (Three) Shifting

In [22], it was shown that increasing the multi-bit shifting over three-bits is not beneficial. The time reduction probability will be too low compared to three-bit shifting making three bit shifting as the appropriate hardware to build. The AlmMonInv computation time is affected similarly as to the two-bits shifting (section 4.2) with the difference in time reduction to calculate steps 2, and 3. This three-bit shifting made the AlmMonInv computation time as follows:

*Cycles for steps 4,5,6 =* $0.5\,k\,(\lceil n/w \rceil+1)$
*Cycles for step 9 =* $\lceil n/w \rceil$
*Cycles for steps 2,3 =* $0.35\,k$
*Total AlmMonInv Cycles =* $0.5\,k\,(\lceil n/w \rceil+1)+\lceil n/w \rceil+0.35\,k$

## 4.4 CorPh Single Bit Shifting

The correction phase algorithm (HW-Alg2) can run on the new hardware with single bit shifting and two bits shifting. It cannot benefit from three bits shifting since it will need an impractical increase in the number of adders of the scalable design as clarified in [22]. The area of the hardware design is not affected when running HW-Alg2 while the computation time is. The computation time of HW-Alg2 depend on the total number of iterations and some extra cycles within the iterations due to scalability. The single bit shifting number of iterations is $2n-k-1$, assuming on average $k=1.5n$, will result:

*number of iterations =* $2n-1.5n-1\approx 0.5n$.

HW-Alg2 will need this number of iterations to process step 13 followed by step 14. Step 14 needs the extra scalability cycles of $\lceil n/w \rceil$ as detailed below:

*Cycles for step 13 =* $0.5\,n$
*Cycles for step 14 =* $0.5\,n * \lceil n/w \rceil$
*Total CorPh Cycles =* $0.5\,n + 0.5\,n * \lceil n/w \rceil$

## 4.5 CorPh Multi-Bit Shifting

When two-bits shifting method is involved within HW-Alg2, the average computation time will be halved. The average number of cycles to compute HW-Alg2 using the new hardware with multi-bit shifting is as follows:

*Cycles for step 13* $= 0.5\, n/2$
*Cycles for step 14* $= 0.5\, n/2 * \lceil n/w \rceil$
*Total CorPh Cycles* $= (0.5\, n + 0.5\, n * \lceil n/w \rceil)/2$

The exact computation time is computed by the number of cycles multiplied by the clock cycle period. It was found that the new hardware clock period is not affected by the shifting modification of this work, which made the clock period of the new hardware depend on the value of *w*, exactly as the clock period of the original scalable hardware of [22] as listed in Table 1.

**Table 1 Clock cycle period for all scalable designs (nsec)**

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

## 5 Comparisons and Analysis

### *5.1 Area Comparison*

The hardware area of any VLSI architecture depends on the technology and minimum feature size. For technology independence, the number of equivalent gates are used as area measure [25]. 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 [26].

The areas of all scalable designs are shown in Fig. 5. This figure shows the area of the two types of new scalable designs, single and multi-bit shifting, compared to the old scalable designs of [22]. As $n_{max}$ and *w* increase, all designs areas are getting larger. Observe that as $n_{max}$ is very low, i.e. $n_{max}$ around 128 bits, the multi-bit shifting hardware with small *w* is smaller than the single bit shifting one with large *w*. Similarly, for the single bit shifting new hardware compared to the old hardware, as $n_{max}$ is low, the new hardware with small *w* is smaller than the old hardware with large *w*.

The percentage of area increases with relation to *w* for different scalable designs are shown in Fig. 6. All the percentages shown are for the new hardware designs compared to the old designs of [22]. Observe that the area increase goes low as *w* gets larger. In fact, the complete option is given to the application and its hardware capability. If area is available, the hardware chosen can be the biggest.

## *5.2 Delay Comparison*

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. 7 compares the delay of six scalable hardware designs of all types, the new single bit shifting hardware, the new multi-bit shifting hardware, and the old hardware of [22]. The study assumes all architectures are designed for maximum bits of $n_{max}=512$ bits, which is the practical number for future ECC applications [1]. Note that the difference in the number of bits of the actual data size ($n$) affects the number of cycles which changes the speed of the designs. In other words, as $n$ reduces and $w$ is small, the overall computing time of any scalable design reduces. This is a major advantage of the scalable hardware over all other non-scalable designs such that the computation time depends on the actual number of bits and not on the hardware capability $n_{max}$.

Fig. 7 shows that the computation time of all new designs are less than the old ones in all cases. Similarly, the new hardware with multi-bit shifting is always faster than the single bit shifting hardware. However, as the value $w$ goes large compared to the actual number of bits $n$, the computation time increase fast not benefiting from the scalability. In other words, as $w$ gets bigger the total time decreases fast, which is true in all different scalable designs as long as $n \geq w$.

Observe also in Fig. 7, as $n$ increases to the maximum, i.e., $n = n_{max} = 512$*-bits*, the fastest hardware is the new multi-bit shift scalable design with $w=128$ bits and $w=64$ *bits*, which are almost the same speed. This implies that even if you go to a bigger design you are not going to gain in speed anymore. Another interesting observation for the maximum $n$ is that the new multi-bit shift hardware with $w=16$ *bits* is slower than the single bit shift new hardware with $w\geq 32$ *bits*, which indicates the falseness of guessing that the bigger the designs always give higher speed.

The percentage speedup of the new hardware compared to the old one is shown in Fig. 8. The speedup percentage shown is for both types of new hardware designs, i.e. single bit shifting and multi bit shifting architectures. Interestingly, the multi-bit shifting new hardware is having a positive speedup percentage in most of the cases. On the other hand, the new hardware with single bit shifting is having negative speedup when $2n \geq w$. In other words, the single bit shifting new hardware is too slow compared to the old hardware whenever $w$ is larger or near the value of half $n$.

The multi bit shifting new hardware is faster than the single bit shifting one. The speedup of these two types of new hardware architectures are shown in Fig. 9. It can be observed that the multi bit shifting design is faster than the single bit shifting one within the range from 18% to 22%. Note that the percentage of speedup depends on the value of $n$. The $n$ values that give the best speedup percentage for all designs is summarized in Table 2.

**Table 2 New hardware n value for best speedup of multi-bit over single bit shift architectures.**

| ***w*** | **4** | **8** | **16** | **32** | **64** | **128** |
|---|---|---|---|---|---|---|
| ***n*** | 16 | 32 | 16-64 | 32 | 32-64 | 32-256 |

## *5.3 Area × Time of New Hardware*

Choosing the appropriate scalable design is depending on the importance of speed and area. In fact, as seen from the area study, Figure 5, and the delay one, Figure 7, as we increase in terms of area we gain in most of the cases in speed. However, is the speed gained worth the area paid?

To estimate an evaluation standard that relates between area and time, two *figure of merit* values are used depending on each factor importance. If area is assumed to have the same importance as time, AT (Area×Time) is used to decide the best design. On the other hand, if the time is the most important factor, $AT^2$ (Area×Time×Time) is considered. It is assumed that as the figure of merit values reduces as the design is better.

Figures 10 and 11 show the AT results of the scalable designs with respect to the number of bits *n* for single bit shifting and multi bit shifting architectures, respectively. Both AT figures show that our proposed designs with single bit shifting are giving the best designs at similar *w* values.

The best AT scalable architecture depends on the actual number of bits *n* required by an application. For example, if the number of bits is impractically low, i.e. *n=8 bits*, the best design would be with *w=n*. If the actual number of bits: $16 \leq n \leq 64$, the best hardware would be with *w* as the smallest *n* (*w=16 bits*). The design with *w=32 bits* is the appropriate for the actual number of bits: $128 \leq n \leq 256$. If *n>265 bits*, the suitable design would be with *w=64 bits*. Figures 10 and 11 confirm that there is no need to build scalable designs with *w≥128 bits*, as long as the hardware time and area both have the same importance. The AT best architectures *w* values related to *n* are summarized in Table 3.

**Table 3 AT best architectures depending on the actual number of bits (n).**

| ***n*** | ***8*** | ***16*** | ***32*** | ***64*** | ***128*** | ***256*** | ***512*** |
|---|---|---|---|---|---|---|---|
| ***w*** | 8 | 16 | 16 | 16 | 32 | 32 | 64 |

## *5.4 Area ×Time$^2$ of New Hardware*

$AT^2$ is the appropriate figure of merit to find the right and proper hardware assuming the time is much more important than the area. The best new hardware architecture with single bit shifting can be derived from Figure 12. Depending on the actual number of bits *n* the appropriate design word size *w* is chosen.

Recall that all designs are built to handle the maximum number of bits $n_{max}$ = *512 bits*. If $n$ = *8 bits*, the proper design to be selected is the one with $w$ = *8 bits*. When the actual number of bits: $16 \leq n \leq 32$, the suitable architecture is with $w$ = *16 bits*. As the actual number of bits goes practically large, i.e. $n \geq n_{max}/8$ ($n \geq 64$ *bits*), the best AT$^2$ single bit shifting design is always the one with $w = n_{max}/4$ ($w$=*128 bits*), as in Table 4.

**Table 4 AT$^2$ best single bit shifting architectures.**

| ***n*** | ***8*** | ***16*** | ***32*** | ***64*** | ***128*** | ***256*** | ***512*** |
|---|---|---|---|---|---|---|---|
| ***w*** | 8 | 16 | 16 | 128 | 128 | 128 | 128 |

The AT$^2$ of multi bit shifting architectures are shown in Figure 13. The appropriate multi bit shifting hardware for $n \geq 256$ *bits* is the one with $w$ = *64 bits*. If the actual number of bits: $64 \leq n \leq 128$, the suitable design is with $w$ = *32 bits*. Whenever $16 \leq n \leq 64$, the correct architecture to choose is with $w$ = *16 bits*. For $n$ = *8 bits* the design to be used should be with $w$ = *8 bits*. Note that the biggest hardware to be used is not to exceed $w$ = *64 bits*, according to this AT$^2$ study. Figure 13 is giving different suitable hardware designs than Figure 12 making a new summary table of best multi bit shifting hardware architectures related to $n$ shown as Table 5.

**Table 5 AT$^2$ best multi bit shifting architectures.**

| ***n*** | ***8*** | ***16*** | ***32*** | ***64*** | ***128*** | ***256*** | ***512*** |
|---|---|---|---|---|---|---|---|
| ***w*** | 8 | 16 | 16 | 32 | 32 | 64 | 64 |

## 6 Conclusion

This work modified a scalable VLSI architecture for GF(p) Montgomery modular inverse computation to gain in speed. The architecture is made of two parts, a non-scalable memory-shifting unit and a scalable module to handle operands of any precision. The word-size that the scalable 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 original old hardware had shifting operation performed within the computing unit. This shifting operation has been moved from the scalable computing unit to the non-scalable memory part. Two shifting strategies have been investigated, single bit shifting and multi bit shifting, which gave different speedup and hardware area results. In general, the new hardware with single bit shifting was double the area of the original old one gaining the speedup that can reach 28%. The multi bit shifting new hardware increased the original hardware area by a range from two times to four times, depending on the word size of the scalable unit *w*. It gained speedup that can reach 40% depending on the increase of the actual number of bits *n*. On the other hand, as *n* goes low, all new designs speedup reduce allowing for negative values. Negative speedup indicates that the features of the new proposed scalable hardware can be a burden instead of being a benefit; it increased the area and gave lower speed.

Depending on the actual number of bits *n* and the figure of merit AT or AT$^2$, different designs can be chosen. Table 6 below summarizes all best designs according to the actual number of bits *n* used. All designs are capable to handle up to 512 bits, but the appropriate one is selected depending on the actual number of bits *n* the application is expected to commonly have. The study 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.

**Table 6 Best new architectures according to *n*.**

| ***Range of actual number of bits n*** | ***AT Best architecture word w single& multi bit shift*** | ***AT$^2$Best architecture word size w*** | |
|---|---|---|---|
| | | ***Single bit shift*** | ***multi bit shift*** |
| ***8*** | 8 | 8 | 8 |
| ***16*** | 16 | 16 | 16 |
| ***32*** | 16 | 16 | 16 |
| ***64*** | 16 | 128 | 32 |
| ***128*** | 32 | 128 | 32 |
| ***256*** | 32 | 128 | 64 |
| ***512*** | 64 | 128 | 64 |

## 7 Acknowledgments

I would like to thank the British council in Saudi Arabia, for supporting this research through their postdoctoral program. Continuous support from KFUPM is also appreciated. I would like also to thank Dr. Tatiana Kalganova at the Electrical & Computer Engineering Department of Brunel University in Uxbridge, for hosting me during my visit to the UK and for all fruitful discussions and providing the facilities needed to finish this work.

## 8 References

[1] Blake, I., Seroussi, G., and Smart, N.: 'Elliptic Curves in Cryptography', (Cambridge University Press: New York, 1999)

[2] Savas, E., and Koc, C.: 'The Montgomery Modular Inverse – Revisited', *IEEE Trans. Computers*, 2000, 49, (7), pp. 763-766

[3] Kobayashi, T., and Morita, H.: 'Fast Modular Inversion Algorithm to Match Any Operation Unit', *IEICE Trans. Fundamentals*, 1999, E82-A, (5), pp. 733-740

[4] Hankerson, D., Menezes, J., and Hernandez, J.: 'Software Implementation of Elliptic Curve Cryptography Over Binary Fields', *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2000, Massachusetts

[5] Takagi, N.: 'Modular Inversion Hardware with a Redundant Binary Representation', *IEICE Transactions on Information and Systems*, 1993, E76-D, (8), pp. 863-869

[6] Takagi, N.: ‘A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm’, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 1998, E81-A, (5), pp.724-728

[7] Hasan, M.A.: ‘Efficient Computation of Multiplicative Inverse for Cryptographic Applications’, *Proceeding of the 15th IEEE Symposium on Computer Arithmetic*, 2001, Vail, Colorado

[8] Guo, J., and Wang, C.: ‘Systolic Array Implementation of Euclid’s Algorithm for Inversion and Division in GF($2^m$)’, *IEEE Trans. Computers*, 1998, 47, (10), pp. 1161-1167

[9] Fenn, S., Benaissa, M., and Taylor, D.: ‘GF($2^m$) Multiplication and Division Over the Dual Basis’, *IEEE Trans. Computers*, 1996, 45, (3), pp. 319-327

[10] Wang, C., Truong, T., Shao, H., Deutsch, L., Omura, J., and Reed, I.: ‘VLSI Architectures for Computing Multiplications and Inverses in GF($2^m$)’, *IEEE Trans. Computers*, 1985, C-34, (8), pp. 709-717

[11] Feng, G.: ‘A VLSI Architecture for Fast Inversion in GF($2^m$)’, *IEEE Trans. Computers*, 1989, 38, (10), pp. 1383-1386

[12] 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. VLSI*, 1993, 1, (1), pp. 22-30

[13] Daly, A., Marnane, W., and Popovici, E.: ‘Fast Modular Inversion in the Montgomery Domain on Reconfigurable Logic’, *Irish Signals and Systems Conference (ISSC)*, 2003, pp. 362-367

[14] Zhou, T., Wu, X., Bai, G., and Chen, H.: ‘New Algorithm and Fast VLSI Implementation for Modular Inversion in Galois Field GF(p)’, *IEEE International Conference on Communications, Circuits and Systems*, 2002, 2, pp. 1491-1495

[15] Zhou, T., Wu, X., Bai, G., and Chen, H.: ‘Fast GF(p) Modular Inversion Algorithm Suitable for VLSI Implementation’, *Electronics Letters*, 2002, 38, (14), pp. 706-707

[16] McIvor, C., McLoone, M., and McCanny, J.: ‘Improved Montgomery Modular Inverse Algorithm’, *Electronics Letters*, 2004, 40, (18), pp. 1110-1111

[17] Dormale, G., Bulens, P., and Quisquater, J.: ‘An Improved Montgomery Modular Inversion Targeted for Efficient Implementation on FPGA’, *International Conference on Field-Programmable Technology (FPT)*, 2004, pp. 441-444

[18] Fiaz, F., and Masud, S.: ‘Design and Implementation of a Hardware Divider in Finite Field’, *National Conference on Emerging Technologies*, 2004

[19] Feldhofer, M., Trathnigg, T., and Schnitzer, B.: ‘A Self-Timed Arithmetic Unit for Elliptic Curve Cryptography’, *Proceedings of the Euromicro Symposium on Digital System Design (DSD)*, 2002

[20] Tawalbeh, L., Tenca, A., Park, S., and Koc, C.: ‘A Dual-field Modular Division Algorithm and Architecture for Application Specific Hardware’, *Thirty-Eighth Asilomar Conference on Signals, Systems and Computers*, 2004, 1, pp. 483 – 487

[21] Gutub, A., Tenca, A., and Koc, C.: ‘Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation’, *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2002, Pittsburgh, Pennsylvania

[22] Gutub, A., and Tenca, A.: ‘Efficient Scalable VLSI Architecture for Montgomery Inversion in GF(p)’, *Integration, the VLSI Journal*, 2004, 37, (2), pp. 103-120

[23] Montgomery, P.: ‘Modular Multiplication Without Trail Division’, *Mathematics Computation*, 1985, 44, (170), pp. 519-521

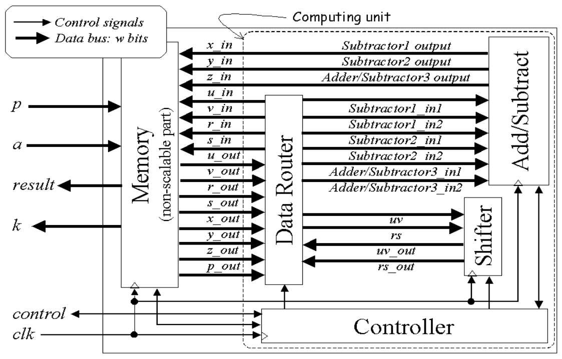
[24] Gutub, A., and Tenca, A.: ‘Efficient Scalable Hardware Architecture for Montgomery Inverse Computation in GF(P)’, *IEEE Workshop on Signal Processing Systems (SIPS)*, 2003, pp. 93-98

[25] Ercegovac, M., Lang, T., and Moreno, J.: ‘Introduction to Digital System’, (John Wiley & Sons, Inc., New York, 1999)

[26] http://www.mentor.com/partners/hep/AsicDesignKit/dsheet/ami05databook.html, Mentor Graphics Co., *ASIC Design Kit.*, accessed January 2006

Figure Captions:

Figure 1



Control signals
Data bus: w bits
Computing unit
p
a
result
k
control
clk
Memory
(non-scalable part)
Data Router
Add/Subtract
Shifter
Controller
x_in
y_in
z_in
u_in
v_in
r_in
s_in
u_out
v_out
r_out
s_out
x_out
y_out
z_out
p_out
Subtractor1_output
Subtractor2_output
Adder/Subtractor3_output
Subtractor1_in1
Subtractor1_in2
Subtractor2_in1
Subtractor2_in2
Adder/Subtractor3_in1
Adder/Subtractor3_in2
uv
rs
uv_out
rs_out

Figure 2

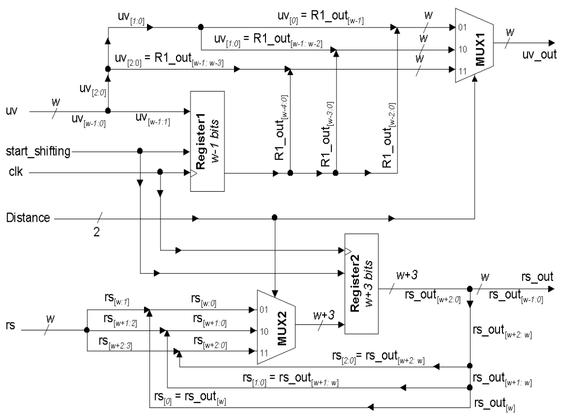

uv[1:0]
uv[0] = R1_out[w-1]
uv[1:0] = R1_out[w-1: w-2]
uv[2:0] = R1_out[w-1: w-3]
w
w
w
w
01
10
11
MUX1
uv_out
uv[2:0]
uv
w
uv[w-1:0]
uv[w-1:1]
Register1
w-1 bits
R1_out[w-4:0]
R1_out[w-3:0]
R1_out[w-2:0]
start_shifting
clk
Distance
2
Register2
w+3 bits
w+3
rs_out[w+2:0]
w
rs_out
rs_out[w-1:0]
rs[w:1]
rs[w+1:2]
rs[w+2:3]
rs
w
rs[w:0]
rs[w+1:0]
rs[w+2:0]
01
10
11
MUX2
w+3
rs_out[w+2: w]
rs[2:0]= rs_out[w+2: w]
rs[1:0] = rs_out[w+1: w]
rs_out[w+1: w]
rs[0] = rs_out[w]
rs_out[w]

Figure 3

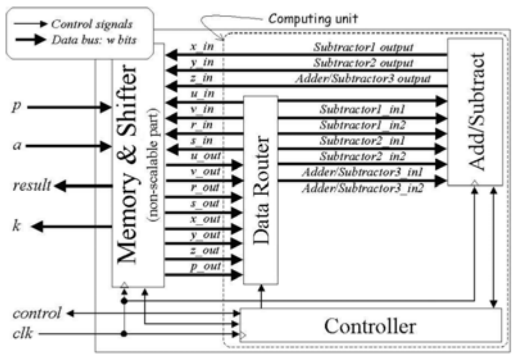

Control signals
Data bus: w bits
Computing unit
p
a
result
k
control
clk
Memory & Shifter
(non-scalable part)
x_in
y_in
z_in
u_in
v_in
r_in
s_in
u_out
v_out
r_out
s_out
x_out
y_out
z_out
p_out
Data Router
Subtractor1 output
Subtractor2 output
Adder/Subtractor3 output
Subtractor1_in1
Subtractor1_in2
Subtractor2_in1
Subtractor2_in2
Adder/Subtractor3_in1
Adder/Subtractor3_in2
Add/Subtract
Controller

Figure 4

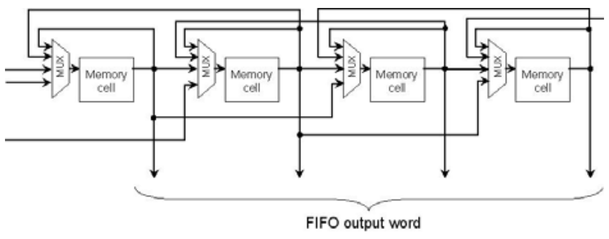

MUX
Memory cell
MUX
Memory cell
MUX
Memory cell
MUX
Memory cell
FIFO output word

Figure 5

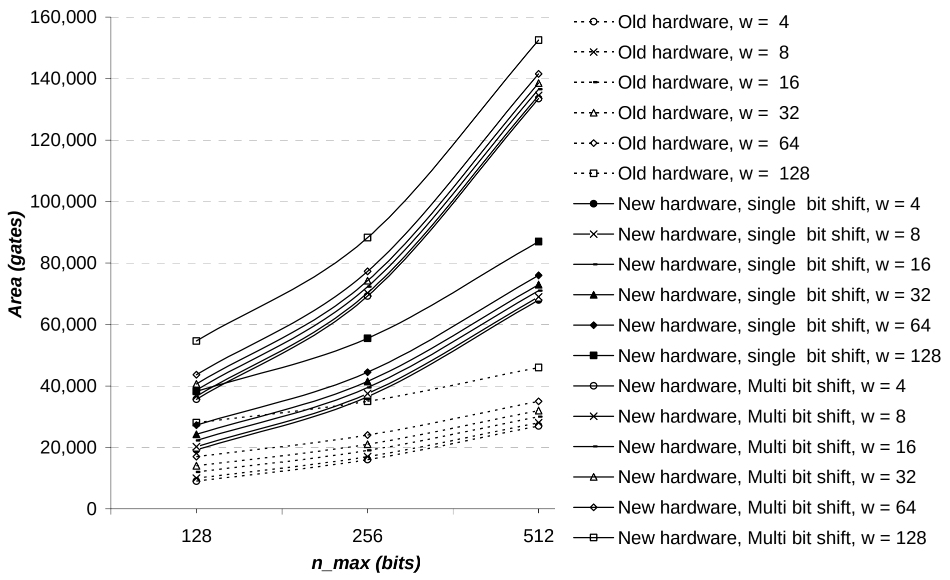
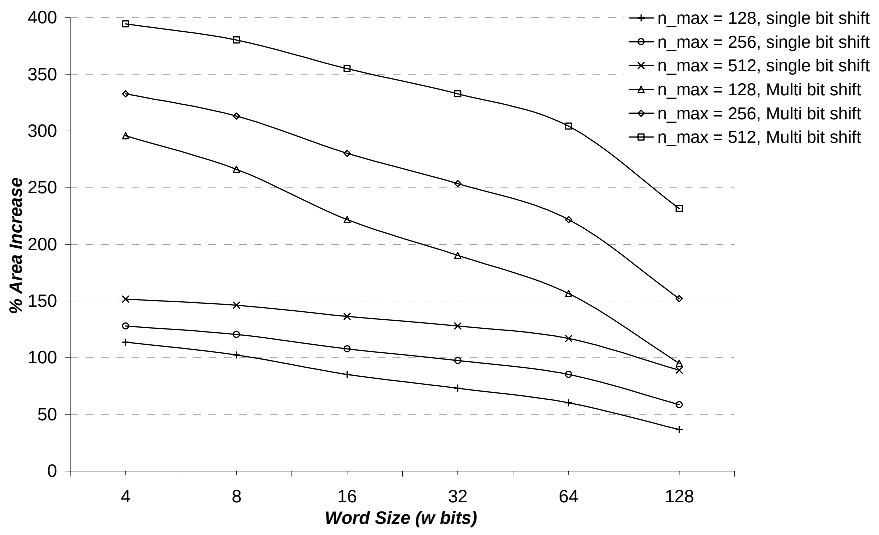
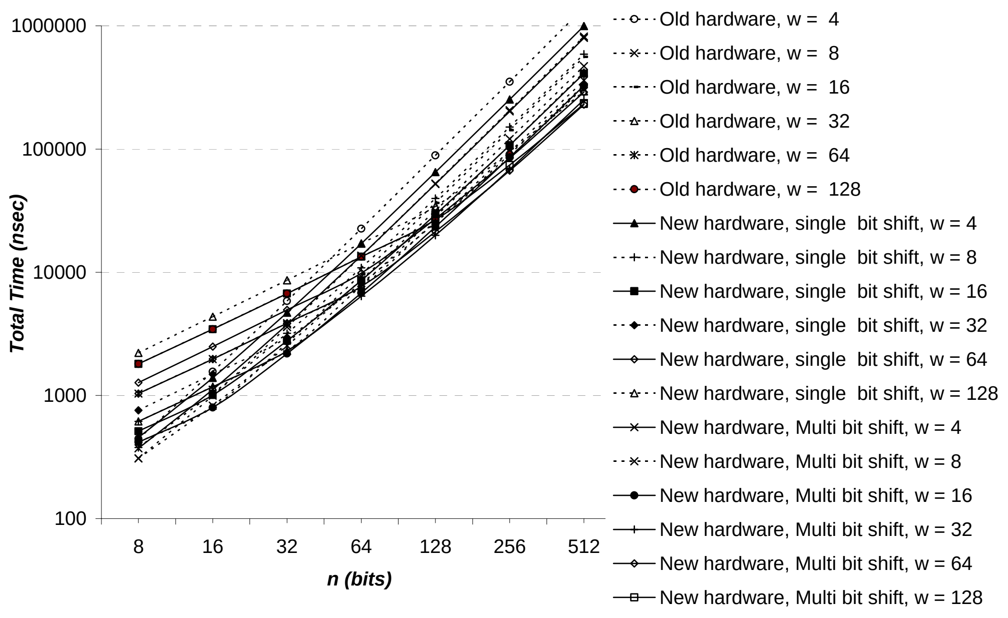

160,000
140,000
120,000
100,000
80,000
60,000
40,000
20,000
0
Area (gates)
128
256
512
n_max (bits)
Old hardware, w = 4
Old hardware, w = 8
Old hardware, w = 16
Old hardware, w = 32
Old hardware, w = 64
Old hardware, w = 128
New hardware, single bit shift, w = 4
New hardware, single bit shift, w = 8
New hardware, single bit shift, w = 16
New hardware, single bit shift, w = 32
New hardware, single bit shift, w = 64
New hardware, single bit shift, w = 128
New hardware, Multi bit shift, w = 4
New hardware, Multi bit shift, w = 8
New hardware, Multi bit shift, w = 16
New hardware, Multi bit shift, w = 32
New hardware, Multi bit shift, w = 64
New hardware, Multi bit shift, w = 128

Figure 6



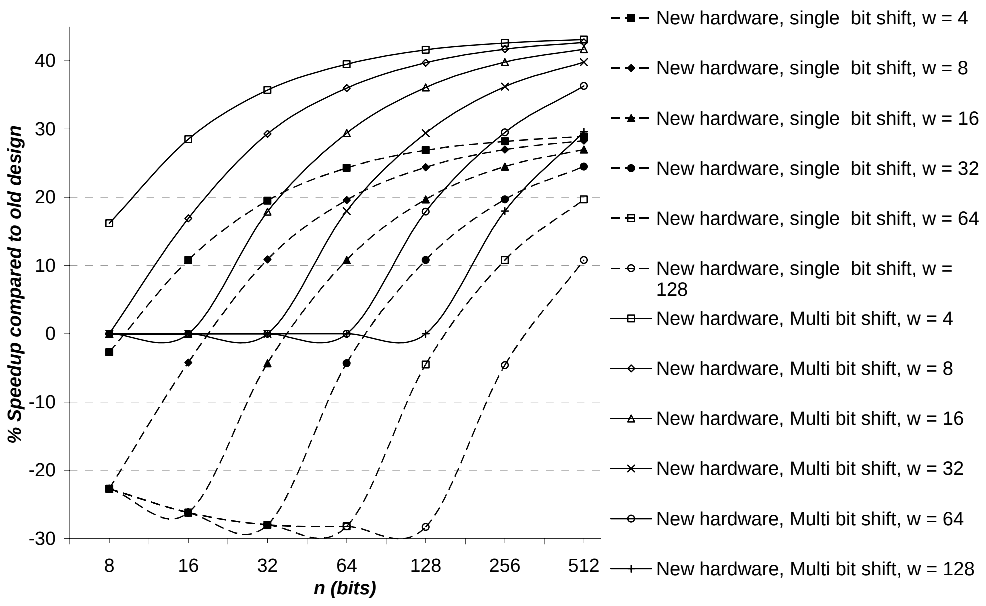
400
350
300
250
200
150
100
50
0
% Area Increase
4
8
16
32
64
128
Word Size (w bits)
n_max = 128, single bit shift
n_max = 256, single bit shift
n_max = 512, single bit shift
n_max = 128, Multi bit shift
n_max = 256, Multi bit shift
n_max = 512, Multi bit shift

Figure 7



1000000
100000
10000
1000
100
Total Time (nsec)
8
16
32
64
128
256
512
n (bits)
Old hardware, w = 4
Old hardware, w = 8
Old hardware, w = 16
Old hardware, w = 32
Old hardware, w = 64
Old hardware, w = 128
New hardware, single bit shift, w = 4
New hardware, single bit shift, w = 8
New hardware, single bit shift, w = 16
New hardware, single bit shift, w = 32
New hardware, single bit shift, w = 64
New hardware, single bit shift, w = 128
New hardware, Multi bit shift, w = 4
New hardware, Multi bit shift, w = 8
New hardware, Multi bit shift, w = 16
New hardware, Multi bit shift, w = 32
New hardware, Multi bit shift, w = 64
New hardware, Multi bit shift, w = 128

Figure 8



% Speedup compared to old design
40
30
20
10
0
-10
-20
-30
8
16
32
64
128
256
512
n (bits)
New hardware, single bit shift, w = 4
New hardware, single bit shift, w = 8
New hardware, single bit shift, w = 16
New hardware, single bit shift, w = 32
New hardware, single bit shift, w = 64
New hardware, single bit shift, w = 128
New hardware, Multi bit shift, w = 4
New hardware, Multi bit shift, w = 8
New hardware, Multi bit shift, w = 16
New hardware, Multi bit shift, w = 32
New hardware, Multi bit shift, w = 64
New hardware, Multi bit shift, w = 128

Figure 9

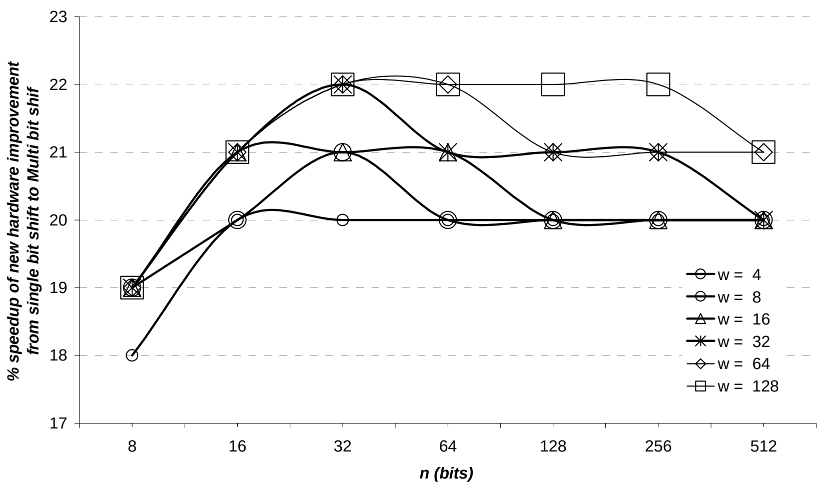
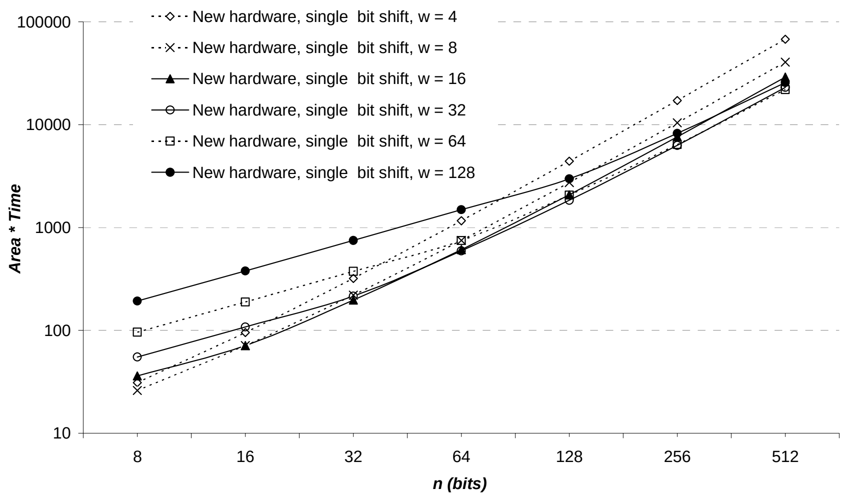

% speedup of new hardware improvement from single bit shift to Multi bit shif
n (bits)
w = 4
w = 8
w = 16
w = 32
w = 64
w = 128
23
22
21
20
19
18
17
8
16
32
64
128
256
512

Figure 10



New hardware, single bit shift, w = 4
New hardware, single bit shift, w = 8
New hardware, single bit shift, w = 16
New hardware, single bit shift, w = 32
New hardware, single bit shift, w = 64
New hardware, single bit shift, w = 128
100000
10000
1000
100
10
Area * Time
8
16
32
64
128
256
512
n (bits)

Figure 11

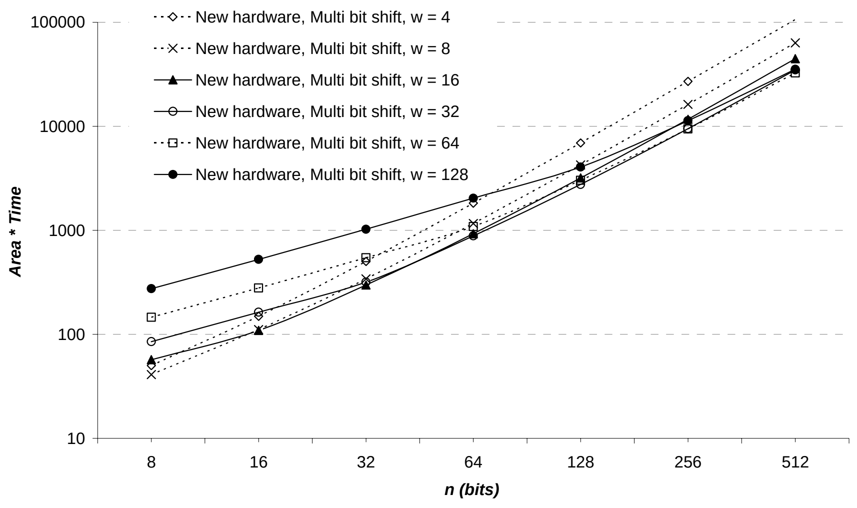
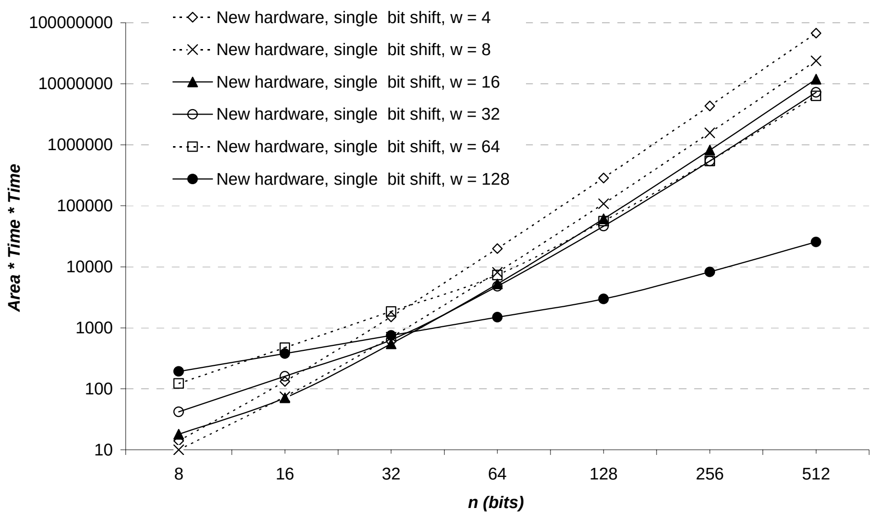
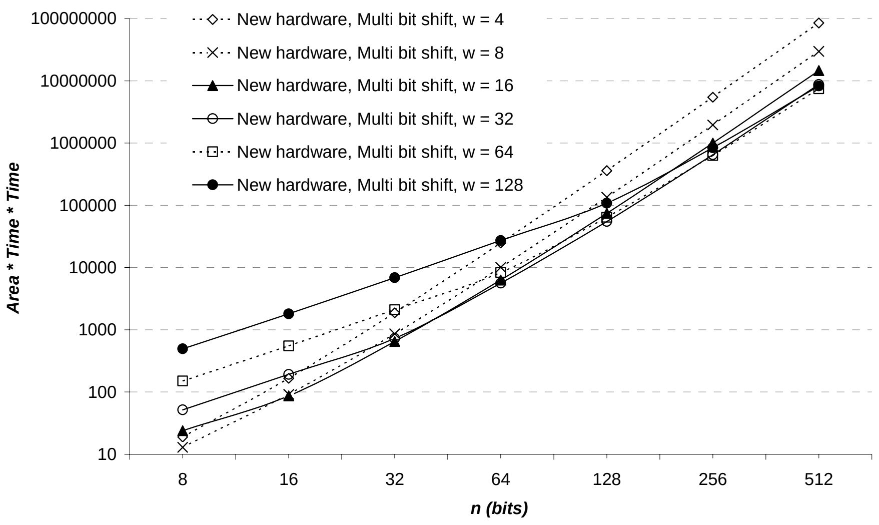

New hardware, Multi bit shift, w = 4
New hardware, Multi bit shift, w = 8
New hardware, Multi bit shift, w = 16
New hardware, Multi bit shift, w = 32
New hardware, Multi bit shift, w = 64
New hardware, Multi bit shift, w = 128
100000
10000
1000
100
10
Area * Time
8
16
32
64
128
256
512
n (bits)

Figure 12



New hardware, single bit shift, w = 4
New hardware, single bit shift, w = 8
New hardware, single bit shift, w = 16
New hardware, single bit shift, w = 32
New hardware, single bit shift, w = 64
New hardware, single bit shift, w = 128
100000000
10000000
1000000
100000
10000
1000
100
10
Area * Time * Time
8
16
32
64
128
256
512
n (bits)

Figure 13



100000000
10000000
1000000
100000
10000
1000
100
10
Area * Time * Time
8
16
32
64
128
256
512
n (bits)
New hardware, Multi bit shift, w = 4
New hardware, Multi bit shift, w = 8
New hardware, Multi bit shift, w = 16
New hardware, Multi bit shift, w = 32
New hardware, Multi bit shift, w = 64
New hardware, Multi bit shift, w = 128