Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation

Adnan Abdul-Aziz Gutub, Alexandre Ferreira Tenca, and Çetin Kaya Koç
Department of Electrical and Computer Engineering
Oregon State University, Corvallis, Oregon 97331, USA
{gutub,tenca,koc}@ece.orst.edu

Abstract

Modular inverse computation is needed in several public key cryptographic applications. In this work, we present two VLSI hardware implementations used in the calculation of Montgomery modular inverse operation. The implementations are based on the same inversion algorithm, however, one is fixed (fully parallel) and the other is scalable. The scalable design is the novel modification performed on the fixed hardware to make it occupy a small area and operate within better or similar speed. Both hardware designs are compared based on their speed and area. The area of the scalable design is on average 42% smaller than the fixed one. The delay of the designs, however, depends on the actual data size and the maximum numbers the hardware can handle. As the actual data size approach the hardware limit the scalable hardware speedup reduces in comparison to the fixed one, but still its delay is practical.

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 the RSA technique [4]. Modular inversion is a basic operation in the elliptic curve cryptography (ECC) [1,2,7,13]. This work is targeted mainly toward the ECC utilization because of its promise to replace several older cryptographic systems [7,13].

Inversion is well known to be the slowest computation among all other ECC arithmetic calculations [1,2,7,10-12]. To make modular inverse calculation faster is one of the two reasons to do inversion in hardware instead of software [10-12]. The other reason is security. For cryptographic applications, it is more secure to have all the computations handled in hardware, instead of mixing some computations in software with others in hardware. Software-based systems can be interrupted and trespassed by intruders much easier than hardware, which can jeopardize the whole application security.

Modular inversion is often performed by algorithms based on the Extended Euclidean algorithm [7]. Several inversion hardware attempts are described in the literature [10-13]. However, most of them [11-13] are for inversion in Galois Fields GF(2^k). The inversion in GF(2^k) is fast due to the elimination of the carry propagation delay in GF(2^k) calculations. Since we focus on GF(p), the designs proposed in [11-13] for GF(2^k) have no direct link to this work. Takagi in [10], proposed a hardware inversion algorithm with a redundant binary representation to avoid carry propagation delay. However, it requires more area and data transformation that is usually expensive.

The Montgomery modular inverse algorithm suitable for our research is portrayed in [1]. The algorithm requires two main operations: a Montgomery product and an *almost Montgomery inverse* (AlmMonInv) operation. This study is directed towards the implementation of the AlmMonInv. The Montgomery product is beyond the scope of this work and a scalable Montgomery multiplier, such as the one proposed in [6] can generate it.

Two AlmMonInv implementations are modeled, namely the fixed design and the scalable one. The fixed design is fully parallel and processes full precision numbers at every clock cycle. The scalable hardware, however, divides the numbers in words where each word is processed in a clock cycle. We show that the scalable hardware is more appropriate for cryptographic applications.

In the coming section, the reason behind choosing Montgomery modular method is described. Section 2 also presents the Montgomery inverse algorithm, used to derive the hardware algorithm proposed in this work. Section 3 explains the fixed (fully-parallel) hardware. Next, in section 4, the scalable hardware implementation is described in some detail. The comparison between the two hardware implementations is given in section 5.

2. Montgomery Inverse Algorithms

Cryptography is heavily based on modular multiplication, which involves the division by the modulus in its computations. Division, however, is a very expensive operation [8]. This fact made researchers seek

out methods to reduce the division impact and make modulo multiplication less time consuming.

In 1985, P. Montgomery invented his ingenious algorithm to perform modular multiplication without trial division [9]. 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 is paid in some extra computations to represent the numbers into Montgomery domain and vice-versa [1,2,6,9].

To use Montgomery's method for ECC, as an example, the integer numbers 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 ECC operations that need to be computed while computing in Montgomery domain, it made up the issue to have dedicated procedures to compute the modular inverse in the Montgomery domain [1,2]. They are named the Montgomery modular inverse algorithms.

Two Montgomery modular inverse studies are found in the literature [1,2]. Both modify a technique proposed by Kaliski, which is derived from the Extended Euclidean algorithm [3]. Kaliski algorithm [1,3] is divided in two phases. Phase one also called almost Montgomery inverse (AlmMonInv) in this work, takes the integer inputs a and p, and give outputs r and k; where $r = a^{-1}2^k \bmod p$, and $n<k<2n$ (n is the actual number of bits of the modulus p). Phase two takes the outputs of phase one as its inputs, and gives the final result of Kaliski algorithm: $x = a^{-1}2^m \bmod p$; where m is Montgomery constant [1-3]. Note that in both phases the integers: a and $x \in [1,p-1]$.

Kaliski method, basically takes integer a, and produces $x = a^{-1}2^m \bmod p$. If a, is an integer, the algorithm will calculate the inverse of a, but represented in Montgomery domain, as shown in Figure 1. In order to have fast ECC arithmetic, Montgomery multipliers are used and, as a consequence, numbers are represented into Montgomery domain and all modular operations should be performed in this domain. I.e., if the number a is already in Montgomery domain, the application of Kaliski's routine will not give the needed Montgomery inverse result. Some extra arithmetic operations are required to get it.

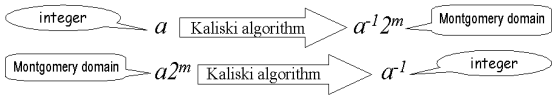

Figure 1 Kaliski algorithm

T. Kobayashi and H. Morita in 1999 [2], proposed techniques for modular inversion to make it suitable and faster than the original Kaliski routine. They modified the AlmMonInv algorithm by performing several matrix multiplications, instead of the simple multiplications by two. Their modification was targeted toward software implementation and for this reason was not so important to our work.

In July 2000, Savas and Koç [1] proposed to replace phase two of Kaliski's algorithm with a Montgomery multiplication, which resulted in a faster process. They also presented a complete Montgomery modular inverse algorithm by adding extra Montgomery multiplication operations. The main procedures used in the complete Montgomery inverse algorithm are the *Montgomery product* (MonPro) and the *almost Montgomery inverse* (AlmMonInv) [1]. Our effort, is directed towards the implementation of the AlmMonInv procedure in hardware. The MonPro is beyond the scope of this work. The AlmMonInv algorithm (Kaliski phase one [1,3]) is outlined below:

AlmMonInv (Almost Montgomery Inverse Algorithm)

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

3. The Fixed (fully-parallel) Design

This section discuses a fixed hardware design of the AlmMonInv algorithm. When observed from hardware point-of-view, the AlmMonInv algorithm contains operations that easily mopped to hardware features. For example, one-bit shifting of binary numbers to the right or left is equivalent to dividing or multiplying by two. Checking for a number to be even or odd is done observing its least significant bit (LSB). If it is found to be zero, the number is even. Comparison of two numbers is performed by subtracting them. If the subtraction result is positive (the subtractor output borrow bit is zero), then the first number is bigger. Such hardware mapping is shown in the hardware algorithm below:

Hardware AlmMonInv Algorithm (HW-Alg)

Input: $a \in [1, p-1]$, p = modulus.
Output: result $\in [1, p-1]$ and k; where result$=a^{-1}2^k \bmod p$
1. u = p, v = a, r = 0, s = 1, x = 0, y = 0, z = 0, k = 0
2. if ($u_0 = 0$) then {u = shift R(u); s = shift L(s)}; goto 7
3. if ($v_0 = 0$) then {v = shift R(v); r = shift L(r)}; goto 7
4. x = Subtract(u,v); y = Subtract(v,u); z = Add (r,s)
5. if($x_{borrow}=0$) then{u=shiftR(x);r=z;s=shiftL(s)};goto 7
6. s = z; v = shift R (y); r = shift L (r)
7. $k = k + 1$
8. if ($v \neq 0$) go to step 2
9. x = Subtract (p, r); y = Subtract ($2p$, r)
10. if($x_{borrow}=0$) then {result = x}; else {result = y}

Consider step 6 of AlmMonInv, if $u > v$ then the subtraction (u - v) takes place, otherwise, the subtraction (v - u) is calculated. In the worst case, two subtraction operations are performed, because the comparison of u and v is accomplished through subtraction of u and v. These two subtractions can be done in parallel (two subtraction modules) as shown in step 4 of HW-Alg. The same case applies to steps 9 and 10 of AlmMonInv, both subtractions may be performed in parallel.

All actual integers are represented by n-bit vectors, such as $u = (u_{n-1}, u_{n-2}, \ldots, u_2, u_1, u_0)$. The modulus is loaded into register u at step 1, then, register u is modified along with the algorithm. The modulus is essential at steps 9 and 10 of HW-Alg and for this reason, it is stored in a special register named p. The value of r cannot equal p except when a equals infinity. Thus the result of AlmMonInv equals either $2p$-r if r is greater than p, or p-r when r is less than p, as described in step 10 of HW-Alg.

3.1. The Fixed Hardware Design

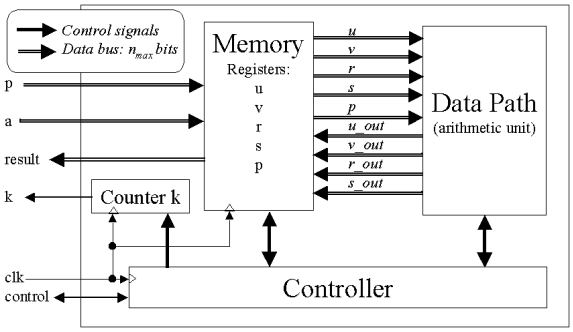

Figure 2 Fixed design hardware outline

The fixed design is made up of a memory unit, a controller, a k-counter, and a data path (arithmetic unit). The block diagram for the fixed design hardware is shown in Figure 2. All data buses are n_{max} bits wide (n_{max} is the maximum number of bits the hardware can handle). The memory unit is made of five registers *u, v, r, s* and *p* to hold n_{max} bits. The memory unit sends out all its content and loads new ones at every clock cycle, except register *p* that does not change during the computation. The data path (DP) takes the memory unit outputs and gives back the computed data to be stored through buses: *u_out*, *v_out*, *r_out*, and *s_out*. For example, in step 3 of HW-Alg, the changing is performed on *v* and *r* only. However, the DP provides the data to all four buses. Buses *v_out* and *r_out* are found to be modifications of *v* and *r*, while *u_out* and *s_out* are just the same *u* and *s* fed back. The DP performs the required computation depending on the LSBs of *u* and *v*, as clarified by HW-Alg. It contains several multiplexers to route and shift the data buses to perform steps 2, 3, 5, 6 and 10. It consists of an adder and two subtractors to perform steps 4 and 9. The counter unit performs step 7 of HW-Alg. All the components in the design are directed and synchronized by the controller.

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 is to be increased, even by one, the complete hardware needs to be replaced. In addition to that, if the design is implemented for a large number of bits, the hardware is huge and its' longest path is impractical. It will cause the hardware to run at a very low clock frequency. These issues motivated the search for a scalable hardware similar to what is proposed by Tenca and Koç in their *Scalable Architecture for Montgomery Multiplication* [6].

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 regions. 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 is replaced while the scalable computing unit is not changed. Finally, the number of clock cycles required for an operation to be computed must depend on the actual size of the numbers used, not on the maximum operand size.

Differently from what happens in the fixed precision hardware design, the scalable hardware has multi-precision operators for addition, subtraction and comparison. The subtraction used for comparison ($u > v$), is performed on a word-by-word basis until all the actual data words are processed, then, the subtractor 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*, need to remain as is until the subtractions processes complete, and the borrow-out bit appears. This forced the use of three more registers: *x*, *y* and *z*; where $x = u$-v, $y = v$-u and $z = r$+s. All operations (addition, subtraction, and shifting) of the scalable hardware algorithm are multi-precision computations. In other words, the numbers are utilized in each operation on a word-by-word basis until the entire number is processed.

4.1. The Scalable Hardware Design

The scalable hardware design is built of two main parts, a memory unit and a computing unit. The memory unit is not scalable because it has a limited storage defines the value 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.

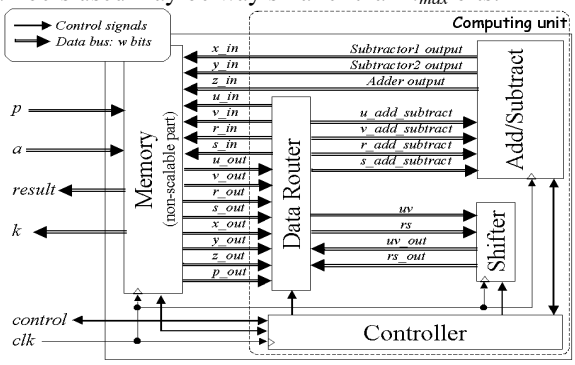

Figure 3 Scalable design hardware outline

The block diagram for the scalable hardware is shown in Figure 3. The memory unit is connected to the computing unit components. The computing unit is made of four hardware blocks, add/subtract block, shifter block, data router block, and the controller. All these computing unit blocks are briefly clarified after describing the non-scalable memory unit. The memory unit contains a counter to compute k (step 7 of HW-Alg) and eight first-in-first-out (FIFO) registers used to store the 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).

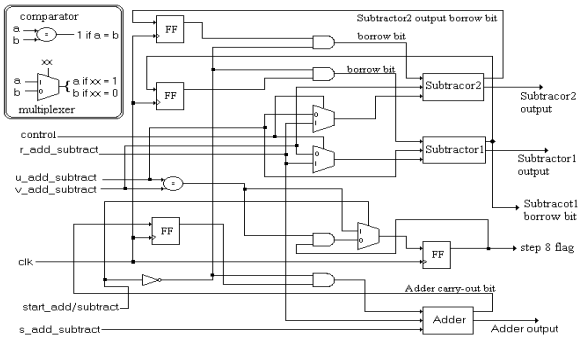

Figure 4 Scalable Add/Subtract unit

The add/subtract unit is built of an adder, two subtractors, four flip-flops, three multiplexers, a comparator, and logic gates, connected as shown in Figure 4. This unit performs one of two operations, either to calculate step 4 of HW-Alg: $x=u-v$, $y=v-u$, and $z=r+s$, or to calculate step 9: $x=p-r$ and $y=2p-r$. Three flip-flops are used to hold the intermediate carry-bit of the adder and borrow-bits of the two subtractors 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. This flag is used to perform step 8 of HW-Alg. The first subtractor borrow out bit is connected to the controller through a signal that is useful only at the end of the each multi-precision addition/subtraction operation. It (as x_{borrow} in HW-Alg) will affect the flow of the operation to choose either step 5 or 6 of HW-Alg. It is also essential in choosing the final result observed in step 10.

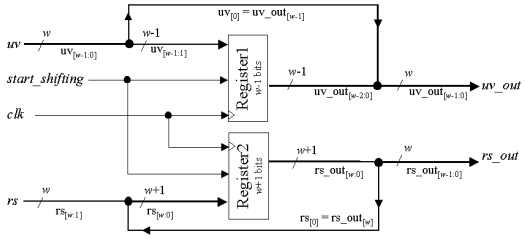

Figure 5 Shifter hardware

The shifter is made of two registers with special mapping of some data bits, as shown in Figure 5. Two types of shifting are needed in the hardware algorithm, shifting an operand (u or v) through the uv bus one bit to the right, and shifting another operand (r or s) through the rs bus one bit to the left. Shifting u or v is performed through Register1, which is of size $w-1$ bits. For each word, all the bits of uv are stored in Register1 except the LSB, it is read out immediately as the most significant bit (MSB) of the output bus uv_out. Shifting r or s to the left is performed via Register2, which is of size $w+1$ bits similar to shifting uv but to the other direction.

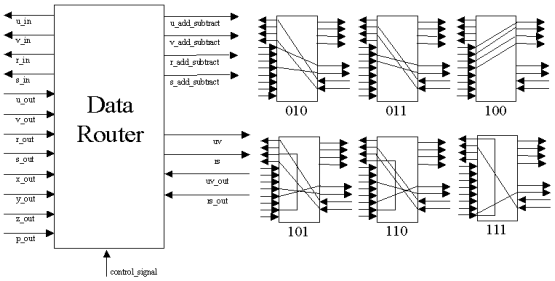

Figure 6 Data router configurations

The data router is made of ten multiplexers to connect the data going out of the memory unit to the inputs of the add/subtract unit or shifter. It also directs the shifted data values to go to their required locations in the memory unit. The possible configurations of the data router are shown in Figure 6. The controller is the unit that coordinates the flow of data to guide the hardware computation. Its made of a state machine easily derived from HW-Alg. The controller does not include counters to avoid any dependency on the number of bits that the system can handle.

5. Modeling and Analysis

Both designs were modeled and simulated in VHDL. The developed VHDL implementation of the scalable hardware has two main parameters, namely n_{max} and w. The fixed hardware, however, is parameterized by n_{max} only. Their area and speed are presented in this section. We didn't define a specific architecture for the adders and subtractors used in the design. Thus, the synthesis tool

chooses the best option from its library of standard cells. Since, both designs use the same type of adders and subtractors we can make a fair comparison.

5.1 Area Comparison

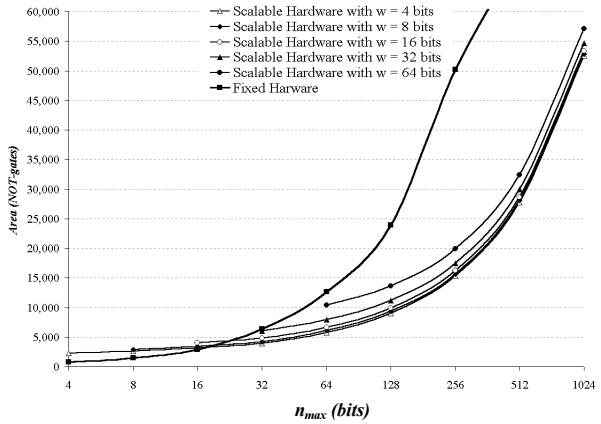

Figure 7 Area comparison

The exact area of any design depends on the technology and minimum feature size. For technology independence, we use the number of NOT-gates as an area measure [8]. A CAD tool from Mentor Graphics (Leonardo) was used. Leonardo takes the VHDL design code and provides a synthesized model with its area and longest path delay. The target technology is a *0.5μm* CMOS defined by the 'AMI0.5 fast' library provided in the ASIC Design Kit (ADK) from the same Mentor Graphics Company [14]. It has to be mentioned here that the ADK is developed for educational purposes and cannot be thoroughly compared to technologies adopted for marketable ASICs. It however, provides a framework to contrast the scalable hardware with the fixed one.

The sizes of the two designs, the scalable and the fixed one, are compared in Figure 7. Observe that the fixed design has a better area if the maximum number of bits used (n_{max}) is less than 32 what is not used in cryptography, small numbers are useless [7]. In fact, the advantage of the scalable hardware is found to make the size of the design as small as possible. For example, if $n_{max} = 512$ bits, the scalable hardware can be designed in less than half the area necessary for the fixed hardware.

5.2. Speed Comparison

The total computation time is a product of the number of clock cycles the algorithm takes and the clock period of the final VLSI implementation. This clock period changes with the value of w in the scalable hardware, and changes with the value of n_{max} in the fixed hardware. This is because $w = n_{max}$ in the fixed hardware. Table 1 lists the clock period for each design (data are generated by Leonardo).

The number of clock cycles depends completely on the data and its computation. For the fixed design, the number of clock cycles is $k+4$, where k is the number of iterations counted through the HW-Alg loop, step 2 to 7. The value of k (HW-Alg) is within the range $[n,2n]$ [1], which justify the use of its average of $3n/2$, for comparison purposes. This makes the total number of clock cycles required for the fixed design to complete a computation equal to $C_f = (3n/2) + 4$.

Table 1 All designs Clock cycle periods (nsec)

n_{max}	Scalable Hardware where w =					Fixed Design
	4	8	16	32	64	
4	9.62	12.39	19.48	30.66	54.93	11.41
8	9.62	12.39	19.48	30.66	54.93	15.96
16	9.62	12.39	19.48	30.66	54.93	26.5
32	9.62	12.39	19.48	30.66	54.93	48
64	9.62	12.39	19.48	30.66	54.93	92
128	9.62	12.39	19.48	30.66	54.93	178
256	9.62	12.39	19.48	30.66	54.93	350
512	9.62	12.39	19.48	30.66	54.93	694
1024	9.62	12.39	19.48	30.66	54.93	1382

The number of clock cycles in the scalable design is a function of three factors: k, w and n. The number of cycles to compute any scalable addition and/or subtraction is calculated as $\lceil n/w \rceil$, which makes the actual number of clock cycles depend on the real data used and its size. However, after several experiments, we concluded that approximately half the time step 2 or 3 of HW-Alg is needed and the other half step 4 is required. But the loop iteration time to execute step 2 or 3 is different than step 4. Step 4 needs extra cycles for the shifting operation after it. The number of cycles to perform each loop iteration (step 2 to 7 of HW-Alg) is calculated as $CPLI=[(\lceil n/w \rceil+1)/2]+\lceil n/w \rceil+3$, (CPLI stands for the clock cycles per loop iteration). The number of loop iterations of the algorithm is exactly equal to k. The overall number of cycles equals the CPLI × k (the number of loop iterations), plus the final operation of steps 9 and 10 (HW-Alg). The total number of cycles of the scalable hardware equals to $C_s=7+(7/2)k+[(4+(3/2)k)(\lceil n/w \rceil)]$, which was verified by VHDL simulation. If k is approximated to its average of $3n/2$ (similar to the fixed design), the function of the clock cycles would be $C_s=7+[(21/4)n]+[(4+(9/4)n)(\lceil n/w \rceil)]$.

The scalable hardware can have several designs for each n_{max} depending on w. For example, Figure 8 shows the delay of five designs of the scalable hardware compared to the fixed hardware, all modeled for $n_{max}=256$ bits. Observe how the actual data size (n) plays a big role on the speed of the designs. In other words, as n reduces for small w, the number of clock cycles decrease significantly, which considerably reduces the overall computing time of the scalable design. This is a major advantage of the scalable hardware over the fixed one.

The number of clock cycles of the fixed model depends on the actual size of the data used. However, its period always assume to have n_{max} bits to process. For example, if we are using n = 64 bits, and the design is

made for n_{max} = 256 bits, as of Figure 8, the fixed design will assume we are using all the 256 bits by placing zeros for the unused bits. All n_{max} bits are processed into the computation causing the fixed design to have more delay than all different scalable ones.

Another observation seen from Figure 8 is that the delay of all the scalable designs are better than the fixed one when $n \leq n_{max}/2$, except for $w=4$ bits that is better when $n \leq 3n_{max}/8$. The scalable designs with w = 8, 16, 32, and 64 bits are faster than the fixed one as long as $n \leq 128$ bits ($n \leq n_{max}/2$). However, for the scalable design with w = 4, it is faster than the fixed one while $n \leq 96$ bits ($n \leq 3n_{max}/8$). In fact, as w gets bigger the delay decreases, which is a normal speed area trade-off.

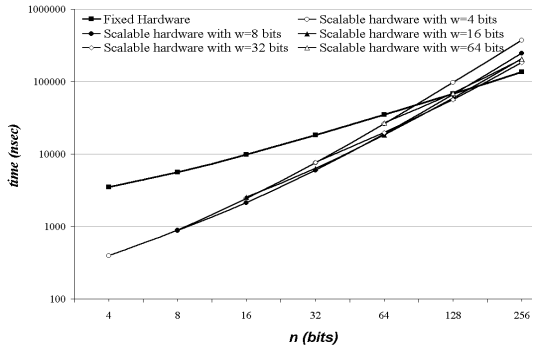

Figure 8 Delay comparison with n_{max}=256 bits

6. Conclusion

This paper presents two VLSI implementations for an algorithm used in the computation of Montgomery modular inverse arithmetic. The two designs are the fixed (fully parallel) hardware and the scalable hardware. The scalable architecture makes the design's longest-path shorter, compared to the fixed hardware. This affected the clock frequency of the scalable hardware to be higher. The scalable hardware is also designed to fit in a small area with the computation of numbers performed in a repetitive way. The maximum number of bits (n_{max}) the scalable hardware can handle depends only on the memory. If the number of bits exceeds the memory size, the memory unit is the only part that needs to be modified, while the scalable computing unit does not change. On the other hand, all the fixed hardware components need to be changed completely if any extra bit is to be added beyond the memory limit.

The scalable design shows area flexibility, depending on the number of bits used at each clock cycle (w). For example, if w = 4 bits and the design can handle up to 512 bits, the area of the scalable design is 60% less than the fixed hardware. The speed of this scalable hardware deviate depending on the actual number (n) of bits used; if $n \leq 192$, the scalable design is found to be faster than the fixed one. Therefore, the real time required to execute the algorithm loop iteration on the scalable hardware relies on the actual size of the operands, which is not the case for the fixed hardware. This made the scalable hardware speed more realistic than the fixed hardware speed.

The comparisons show that this scalable structure is very attractive for cryptographic systems, particularly for ECC because of its need for modular inversion of large numbers, which differ in size repetitively depending on the application usage.

7. Acknowledgments

We are thankful to the Information Security Laboratory researchers for their invaluable technical help and recommendations. This research received financial support from KFUPM-Saudi Arabia, NSF, and RTrust.

References

[1] Savas, and Koç, "The Montgomery Modular Inverse Revisited", *IEEE Transactions 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 Transactions 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 Transactions on Information Theory*, 22:644-654, Nov. 1976.

[6] Tenca, and Koç, "A Scalable Architecture for Montgomery Multiplication", *In Cryptographic Hardware and Embedded Systems*, number 1717 in Lecture notes in Computer Science. Springer, Berlin, Germany, 1999.

[7] Blake, Seroussi, and Smart, *Elliptic Curves in Cryptography*, Cambridge University Press: New York, 1999.

[8] Ercegovac, M. D., Lang, T., and Moreno, J. H., *Introduction to Digital System*, John Wiley & Sons, Inc., New York, 1999.

[9] Montgomery, P.L., "Modular Multiplication Without Trail Division", *Mathematics of Computation*, 44(170):519-521, April 1985.

[10] Naofumi Takagi, "Modular Inversion Hardware with a Redundant Binary Representation", *IEICE Transactions on Information and Systems*, E76-D(8): 863-869, Aug. 1993.

[11] 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.

[12] Choudhury, P. Pal., and Barua, R., "Cellular Automata Based VLSI Architecture for Computing Multiplication and Inverses in GF(2^m)", *Proceeding of the 7th IEEE International Conference on VLSI Design*, Calcutta, India, 5-8 January 1994.

[13] Hasan, M. A., "Efficient Computation of Multiplicative Inverse for Cryptographic Applications", *Proceeding of the 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, 11-13 June 2001.

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