Analysis of Modular Inverse $GF(p)$ Implementations

Gerald Lai

Abstract— **This paper examines several modular inverse algorithms in $GF(p)$ that have been proposed in literature. A survey of these algorithms attempts to study the evolution of modular inversion methods and trace key areas of improvement for hardware implementation efficiency.**

I. Introduction

The heavy use of modular inversions in $GF(p)$ for cryptographic computations today is indisputable. Modular inverse is essential for point operation computations on elliptic curves defined over a finite field $GF(p)$, for decipherment operations of the RSA algorithm, for the Diffie-Hellman key exchange method, for the acceleration of exponentiation operations using addition-subtraction chains and for verifications of the Digital Signature Standard [1], [2], [3], [4].

The classical definition of the modular inverse states that the inverse of an integer $a \in [1, p-1]$ modulo p, is defined as an integer $r \in [1, p-1]$ such that $a \cdot r \equiv 1 \pmod{p}$. This translates to the equation

$$r = a^{-1} \pmod{p} \tag{1}$$

Note that for an integer inverse to exist for a, a must be relatively prime to p. Hence, the greatest common divisor (GCD) of a and p must be equal to 1 or $gcd(a, p) = 1$. The modulus p is usually chosen as a prime number to fulfill this criterion.

Kaliski has modified the definition above to include a Montgomery inverse version of the classical modular inverse [1], [2], [4]. This new version is based on the Montgomery multiplication algorithm devised by P. Montgomery in 1985. The new definition has the Montgomery inverse of an integer $a \in [1, p-1]$ as b such that

$$b = a^{-1}2^n \pmod{p} \tag{2}$$

where p is relatively prime and $n = \lceil log_2\ p \rceil$.

The method of actually computing the modular inverse of an integer originated from one of the most basic algorithms, known as the Euclidean algorithm. This algorithm basically computes the GCD of two numbers.

Algorithm Euclidean
Input: integers a, b
Output: $f = gcd(a, b)$

1: while $(b \neq 0)$
2: $r = a \pmod{b}$
3: $a = b$
4: $b = r$
4: return $f = a$

It was shown that for any integers a and b, the greatest common divisor equation could be written in the form

$$gcd(a, b) = a \cdot u + b \cdot v \tag{3}$$

where u and v are both integers that always exist.

Based on equation (3), the extension to the Euclidean algorithm, known as the Extended Euclidean algorithm, also computed the GCD of the two integers a and b, plus an additional function to compute the integers u and v.

The special case of finding an inverse for the integer a holds true when

$$gcd(a, b) = 1 \tag{4}$$

and the whole equation is taken over (mod b). Hence, equations (3) and (4) can be rewritten as

$$a \cdot u = 1 - b \cdot v \equiv 1 \pmod{b} \tag{5}$$

The $(-bv)$ term drops due to the modulus, leaving integer u as the inverse of integer a (mod b).

The introduction of modular arithmetic to find the inverse of an integer allowed for realizable hardware implementation. As memory is finite in hardware, computations bounded by a finite field become more favorable. However, implementing the Extended Euclidean algorithm in hardware requires integer division, which is a serial operation that is very expensive in terms of hardware delay.

The integer division requirement problem can be solved by modifying the Extended Euclidean algorithm based on three observations [4]:

(i) If a and b are both even, $gcd(a, b) = 2 \cdot gcd(a/2, b/2)$
(ii) If a is even and b is odd, $gcd(a, b) = gcd(a/2, b)$
(iii) If both a and b are odd, $gcd(a, b) = gcd(|a - b|/2, b)$

A division-free approach is provided by the Binary Extended Euclidean algorithm by substituting the integer division portion of the Extended Euclidean algorithm with divisions by two, as shown from the observations above. Divisions by two represent right shift operations in hardware.

The Binary Extended Euclidean algorithm acts as the base algorithm for the design of modular inverse hardware. Further discussions on some of the work that have been proposed to improve the efficiency of this algorithm will be covered in the next few sections.

Author is a graduate student at the Department of Electrical Engineering & Computer Science, Oregon State University, Corvallis, Oregon 97331. E-mail: laige@ece.orst.edu

II. Classical Euclidean Algorithms

The first algorithm shown below is the basic Extended Euclidean algorithm (*ExtEucdInv*) that was derived with some modifications to allow for easy hardware implementation [1]. The basic structure of the modified algorithm is analogous to the Binary Extended Euclidean algorithm.

Algorithm ExtEucdInv (Extended Euclidean Inverse)
Input: $a \in [1, p-1]$ and p
Output: $r \in [1, p-1]$ and k, where $r = a^{-1} \pmod{p}$ and $n \leq k \leq 2n$

```
1:  u := p, v := a, r := 0, s := 1, k := 0
2:  while (v > 0)
3:      if (u is even) then
4:          if (r is even) then
5:              u := u/2, r := r/2, k := k + 1
6:          else
7:              u := u/2, r := (r + p)/2, k := k + 1
8:      else if (v is even) then
9:          if (s is even) then
10:             v := v/2, s := s/2, k := k + 1
11:         else
12:             v := v/2, s := (s + p)/2, k := k + 1
13:     else
14:         x := u − v
15:         if (x > 0) then
16:             u := x, r := r − s
17:             if (r < 0) then
18:                 r := r + p
19:         else
20:             v := −x, s := s − r
21:             if (s < 0) then
22:                 s := s + p
23: if (r > p) then
24:     r := r − p
25: if (r < 0) then
26:     r := r + p
27: return r and k
```

Note that the *while* conditional test for v is similar to the *while* conditional test for b in the original Euclidean Algorithm. The three *if* clauses refer to the algorithmic steps to handle the three GCD cases (i), (ii) and (iii) shown in the previous section that are required to remove integer division. The last two *if* statements act as a correction step to normalize the final result based on the modulus. Checking for evenness or oddness is done very easily in hardware by checking the least significant bit. In most algorithms, when an odd variable is detected, a right shift (division by two) cannot be done on that variable immediately without losing information. That variable has to be offset by the modulus (an odd number) to produce an even variable which could then be divided.

N. Takagi proposed a similar algorithm shown below. It is an extension of the ExtEucdInv algorithm that performs modular division. The Extended Binary GCD algorithm (*ExtBinGCDDiv*) could be made to do modular inversion by setting the numerator x to 1.

Algorithm ExtBinGCDDiv (Extended Binary GCD Modular Division)
Input: $m \in (2^{n-1}, 2^n)$, $x \in [0, m)$ and $y \in (0, m)$
Output: $z \equiv x/y \pmod{m}$

```
1:  a := y, b := m, u := x, v := 0, k := 0
2:  while (a > 0)
3:      while (a is even)
4:          a := a/2
5:          if (u is even) then
6:              u := u/2
7:          else
8:              u := (u + m)/2
9:          k := k − 1
10:     if (k < 0) then
11:         t := a, a := b, b := t
12:         t := u, u := v, v := t
13:         k := −k
14:     if (a + b is even) then
15:         a := (a + b)/2
16:         if (u + v is even) then
17:             u := (u + v)/2
18:         else
19:             u := (u + v − m)/2
20:     else
21:         a := (a − b)/2
22:         if (u − v is even) then
23:             u := (u − v)/2
24:         else
25:             u := (u − v + m)/2
26: if (b < 0) then
27:     z := m − v
28: else
29:     z := v
30: return z
```

The ExtBinGCDDiv algorithm performs modular division by intertwining a procedure for finding the modular quotient with that for calculating $gcd(y, m)$ [5], [6]. The counter k represents $\alpha - \beta$, where α and β are values such that 2^α and 2^β indicate the minimums of the upper bounds of $|a|$ and $|b|$ respectively for any $gcd(a, b)$ operation of the algorithm. Interestingly, this counter acts as an indicator to determine when to swap the (a, b) and (u, v) variables, which is part of the procedure of the original Euclidean algorithm. A 17-bit modular divider of the ExtBinGCDDiv algorithm has been implemented in VHDL [7]. The sequential design is approximately 1700 gates in AMI 0.5 micron technology.

III. Montgomery-Based Algorithms

The Montgomery-based algorithm for computing modular inverse, as proposed by Kaliski, performs the computation in two phases. The AlmMonInv algorithm shown below is a combination of both phases. The first phase performs the modular inverse computation by first taking

the equation terms into the Montgomery domain. It then performs what is known as an Almost Montgomery Inverse on those terms before converting those terms back to the integer domain using the Almost Montgomery Inverse Conversion algorithm (*AlmMonInvConv*). The Montgomery-based algorithm was shown to obtain a slight improvement over the classical algorithm [1].

Algorithm AlmMonInv (Almost Montgomery Inverse)
Phase I
Input: $a \in [1, p-1]$ and p
Output: $y \in [1, p-1]$ and k, where $r = a^{-1}2^k \pmod p$ and $n \leq k \leq 2n$

1: $u := p$, $v := a$, $r := 0$, $s := 1$, $k := 0$
2: while $(v > 0)$
3: if (u is even) then
4: $u := u/2$, $s := 2s$
5: else if (v is even) then
6: $v := v/2$, $r := 2r$
7: else if $(u > v)$ then
8: $u := (u - v)/2$, $r := r + s$, $s := 2s$
9: else
10: $v := (v - u)/2$, $s := s + r$, $r := 2r$
11: $k := k + 1$
12: if $(r \geq p)$ then
13: $r := r - p$
14: return $y := p - r$ and k

Algorithm AlmMonInvCor (Almost Montgomery Inverse Correction)
Phase II
Input: $y \in [1, p-1]$, p and k from Phase I
Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod p$ and $2k$ from Phase I

15: for ($i = 1$ to k) do
16: if (r is even) then
17: $r := r/2$
18: else
19: $r := (r + p)/2$
20: return r and $2k$

The key difference between the Montgomery and the classical algorithms is that once the Montgomery algorithm has the equation data in its domain, computations can be done very quickly. The only downside to this is that there is some cost in bringing the data back into the integer domain. The algorithm is much simpler for Phase I of the AlmMonInv compared to ExtEucdInv. There are less nested *if* statements, which could potentially be expensive in hardware. This issue will be discussed later in this paper.

A. A.-A. Gutub, A. F. Tenca and Ç. K. Koç proposed two VLSI implementations for the Montgomery modular inversion. The first design is a fixed fully-parallel hardware and the second design is a scalable hardware. Both designs were based on the Hardware Almost Montgomery Inverse algorithm (*HW-AlmMonInv*) shown below.

Algorithm HW-AlmMonInv (Hardware Almost Montgomery Inverse)
Input: $a \in [1, p-1]$ and p
Output: $result \in [1, p-1]$ and k, where $result = a^{-1}2^k \pmod p$

1: $u := p$, $v := a$, $r := 0$, $s := 1$, $x := 0$, $y := 0$, $z := 0$
2: $k := 0$
3: if $(u_0 = 0)$ then
4: $u := shiftR(u)$, $s := shiftL(s)$, goto 7
5: if $(v_0 = 0)$ then
6: $v := shiftR(v)$, $r := shiftL(r)$, goto 7
7: $x := u - v$; $y := v - u$; $z := r + s$
8: if $(x_{borrow} = 0)$ then
9: $u := shiftR(x)$, $r := z$, $s := shiftL(s)$, goto 7
10: $s := z$; $v := shiftR(y)$; $r := shiftL(r)$
11: $k := k + 1$
12: if $(v \neq 0)$ then
13: goto 2
14: $x := p - r$; $y := 2p - r$
15: if $(x_{borrow} = 0)$ then
16: $result := x$;
17: else
18: $result := y$;

The scalable architecture is the more attractive design of the two as it tends to have a shorter critical-path, compared to the fixed hardware. The scalable hardware is also designed to fit in a small area. The maximum number of bits the scalable hardware can handle depends only on the memory [2], [3], [4].

IV. Euclidean Algorithm Improvement

Two types of algorithms, which are the classical and the Montgomery-based algorithms, have been introduced. Certain modifications and improvements have been made to adapt these algorithms to hardware, as illustrated in the previous sections. The final algorithm (*ExtEucdInv-2*) shown is a step ahead of the ExtEucdInv algorithm in that it focuses on modifying certain parts of the basic algorithm, such as simplifying conditional checks and minimizing data dependencies, in order to reduce specific hardware costs. This approach is done at a tradeoff. In this case, extra hardware is used for auxiliary counters that are used to track the number of shifts within data registers [1]. This algorithm attempts to avoid the high delay operations such as the postponed halving in k iterations of Phase II of the AlmMonInv algorithm and miscellaneous conditional tests in *if* and *while* statements.

Algorithm ExtEucdInv-2
Input: $a \in [1, p-1]$ and p
Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod p$, c_u, c_v and $0 < c\_v + c\_u \leq 2n$

1: $u := p$, $v := a$, $r := 0$, $s := 1$, $c\_u := 0$, $c\_v := 0$
2: while $(u \neq \pm 2^{c\_u}$ & $v \neq \pm 2^{c\_v})$
3a: if (u_n, $u_{n-1} = 0$) or (u_n, $u_{n-1} = 1$ &

3b: OR(u_{n-2},...,u_0) = 1) then
4: if ($c\_u \geq c\_v$) then
5: $u := 2u$, $r := 2r$, $c\_u := c\_u + 1$
6: else
7: $u := 2u$, $s := s/2$, $c\_u := c\_u + 1$
8a: else if (v_n, $v_{n-1} = 0$) or (v_n, $v_{n-1} = 1$ &
8b: OR(v_{n-2},...,v_0) = 1) then
9: if ($c\_v \geq c\_u$) then
10: $v := 2v$, $s := 2s$, $c\_v := c\_v + 1$
11: else
12: $v := 2v$, $r := r/2$, $c\_v := c\_v + 1$
13: else
14: if ($v_n = u_n$) then
15: oper = "−"
16: else
17: oper = "+"
18: if ($c\_u \leq c\_v$) then
19: $u := u$ oper v, $r := r$ oper s
20: else
21: $v := v$ oper u, $s := s$ oper r
22: if ($v = \pm 2^{c\_v}$) then
23: $r := s$, $u_n := v_n$
24: if ($u_n = 1$) then
25: if ($r < 0$) then
26: $r := -r$
27: else
28: $r := p - r$
29: if ($r < 0$) then
30: $r := r + p$
31: return r, c_u and c_v

R. Lórencz simulated this algorithm with the ExtEucdInv and AlmMonInv algorithms to find out the number of additions/subtractions, shifts and tests that were executed when performing computations for 14,580,841 inverses [1]. According to Lórencz, the ExtEucdInv-2 algorithm outperforms the other two algorithms simply by augmenting algorithmic conditional tests.

It is important to note that benchmark results that show algorithmic performance do not necessarily reflect hardware implementation performance and costs for the same algorithm. For example, the number of addition and subtraction operations is indicative of how often the arithmetic units are active for the payload of computations. While this may be useful for certain power usage estimations, it does not provide a complete picture of the hardware costs in building the design. If delay is essential, the critical path of a particular hardware can be the limiting factor and hence, different implementations of the arithmetic units can be applied to offset the effects of carry propagation.

V. Hardware Efficiency Issues

Metrics of hardware implementations are often not readily apparent from the algorithms proposed using pseudocode. Making a pseudocoded algorithm realizable in hardware involves breaking up the existing algorithm into functional modular parts. Those parts have to be strung together in the datapath in an efficient way so as not to incur a longer critical path than need be. The datapath could also be rearranged in such a way that it allows for pipelining in order to increase result throughput or for scalability in order to allow for expandability [2], [3], [4].

The two main criteria in hardware implementations are speed and area. One other important criterion is power, which is well beyond the scope of this paper. The priorities of these criteria change depending on the platform of implementation. For example, on a cryptoserver, the speed of the hardware implementation is much more important as it has to service a large number of operations fairly quickly, whereas a smartcard may not need to compute as fast but is more area conscious. Therefore, it is important to notice the tradeoffs between the two criteria and how the pseudocoded algorithms affect them.

For instance, it is known that multiplexers take up a substantial amount of area in hardware due to its internal implementation. The number of *if* statements in the pseudocode of the algorithm determine the quantity of the multiplexers. This is due to the fact that *if-else* statements will be inferred as selection multiplexers in hardware. Nested *if* statements infer priority selection and can be realized as cascading multiplexers in hardware. Hence, excessive use of *if* statements within the pseudocode can introduce an incredible increase in hardware area and complicate the design of the hardware control unit.

Hardware costs of conditional tests in pseudocode vary depending on the kind of check that is performed. A check to see if a variable is negative, positive, even or odd is easily done without any cost by checking either the MSB or LSB of the variable. A check to see if a variable is equal to zero is also fairly cheap in hardware. This could be accomplished by either having a logical OR tree structure to sieve through the variable bits or a logical OR ripple structure that runs through every variable bit. The first approach has a critical path that is dependent on the number of bits and fan-in of the logical OR gates. The second approach has a longer critical path.

A check to see if a variable is equal to a particular value is slightly more expensive than the check for zero. It requires an XOR-OR tree or an XNOR-AND tree structure to accomplish this. Lastly, the check to see if a variable is greater than another variable involves a subtraction to acquire the difference between the variables and a comparison of the result to zero. The cost of this check depends on the implementation of the subtraction.

Additions or subtractions can vary in terms of hardware cost depending on how it is implemented. For example, a carry ripple adder (CRA) can incur a very long critical path for the carry propagation, so it is not often used for designs with a large number of bits. On the other hand, a carry save adder (CSA) requires additional area for registers to store their carry representations, even though it is roughly similar in terms of area to the CRA.

Division by two in hardware is simply the rewiring of bit signals. There is no area cost for this operation.

VI. Conclusion

As more and more crypto applications rely on the modular inverse operation, the need to implement the inversion algorithm in hardware efficiently becomes increasingly important. In this paper, several modular inverse algorithms in $GF(p)$ were analyzed to study their origins and to trace some of the evolutionary changes made to the basic algorithm. A few algorithm-to-hardware issues have been discussed. These issues suggest that a general methodology should be outlined to point out the effects of algorithm structure on how the hardware is implemented. Even now, based on the suggestion that GCD-type calculations may be too intricate to handle on cryptoprocessors, the newer GCD-free algorithms for computing modular inverse [8] is spawning interests and deserves further study.

References

[1] R. Lórencz, "New Algorithm for Classical Modular Inverse," in *Cryptographic Hardware and Embedded Systems – CHES 2002*, B. S. Kaliski Jr. et al., Ed. Aug. 2002, vol. 2523 of *Lecture Notes in Computer Science*, pp. 57–70, Springer-Verlag Heidelberg, Germany.

[2] A. A.-A. Gutub, A. F. Tenca, and Ç. K. Koç, "Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation," in *IEEE Computer Society Annual Symposium on VLSI*, Pittsburgh, Pennsylvania, Apr. 2002, pp. 53–58, IEEE Computer Society Press, Los Alamitos, California.

[3] A. A.-A. Gutub, A. F. Tenca, E. Savaş, and Ç. K. Koç, "Scalable and Unified Hardware to Compute Montgomery Inverse in GF(p) and GF(2^n)," in *Cryptographic Hardware and Embedded Systems – CHES 2002*, B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, Eds. Aug. 2002, vol. 2523 of *Lecture Notes in Computer Science*, pp. 484–499, Springer-Verlag Heidelberg, Germany.

[4] A. A.-A. Gutub, *New Hardware Algorithms and Designs for Montgomery Modular Inverse Computation in Galois Fields GF(p) and GF(2^n)*, Ph.D. thesis, Department of Electrical & Computer Engineering, Oregon State University, June 2002.

[5] N. Takagi, "A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm," *IEICE Trans. Fundamentals*, vol. E81-A, no. 5, pp. 724–728, May 1998.

[6] M. E. Kaihara and N. Takagi, "A VLSI Algorithm for Modular Multiplication/Division," *16th IEEE Symposium on Computer Arithmetic – ARITH-16'03*, pp. 220–227, June 2003.

[7] G. Lai, "Hardware Implementation of IDEA (International Data Encryption Algorithm)," *http://islab.oregonstate.edu/koc/ece575/04Project/Lai/idea.htm*, 2004.

[8] M. Joye and P. Paillier, "GCD-Free Algorithms for Computing Modular Inverses," in *Cryptographic Hardware and Embedded Systems – CHES 2003*, C. D. Walter et al., Ed. Oct. 2003, vol. 2779 of *Lecture Notes in Computer Science*, pp. 243–253, Springer-Verlag Heidelberg, Germany.