

Design and Implementation of a Hardware Divider in Finite Field

Fareena Fiaz and Shahid Masud

fareena@ciitlahore.edu.pk and smasud@lums.edu.pk

COMSATS Institute of Information Technology and Lahore University of Management Sciences
Lahore, Pakistan

Abstract: An FPGA based implementation of divider in primary field i.e. $GF(p)$ is presented. The algorithm implemented is based on hardware adapted unified modular division for both integer and Montgomery domain. This algorithm can also determine the inverse of an element in $GF(p)$ when dividend is 1. The computational complexity has been reduced to using only additions, bit shifts and a simple control flow. The FPGA implementation of this design occupies only 15 registers and 6 adders with negligible overheads. The design can operate at over 80 MHz on Xilinx-XST™ Spartan device. It has been captured and synthesized using contemporary design flow and can be ported to any ASIC or PLD platform.

Keywords: Finite field (Galois field), unified modular division, primary field, greatest common divisor, Euclidean algorithm, Montgomery domain, unified Montgomery inverse

1. INTRODUCTION

Division in finite fields is an important arithmetic operation that is widely used in channel coding, cryptography, error correction and code construction applications. An algorithm that is suitable for hardware implementation should require few clock cycles and simple arithmetic operations. One such algorithm has been proposed for modular division and its inverse in $GF(p)$ is called the Unified Modular Division (UMD) algorithm [1]. This algorithmic approach can be used to determine modular inverse in both integer and Montgomery domains. This method is based on extended Euclidean algorithm [2] and binary GCD algorithm [3].

Modular division requires the modular inverse of an element in a finite field. Several architectures for implementing division in finite fields are in place. The most common among those are systolic architectures [4] solving a system of $2m-1$ linear equations over binary finite field also called binary Galois field, $GF(2)$. The complexity of these architectures is much higher than the one proposed in this paper. This paper proposes an FPGA based implementation of divider architecture in $GF(p)$ based on UMD.

The hardware implementation used in this paper reduces the complexity of all iterations making it less complex. Other modular division algorithms have integer and polynomial degree comparisons [5,6,7] as part of their control flow, which is obviated in this work.

The rest of this paper is organized as follows. The simulation and synthesis results are described in section 2. It is followed by the discussion of performance comparison in section 3. Paper summary and final conclusions are covered in section 4.

1.1 GCD and Extended Euclidean Algorithm

Multiplicative/modular inverse of an integer a (mod M) exists if and only if a and M are relatively prime. There are two methods to calculate this inverse. One is based on Fermat's Theorem which states that

$$a^{M-1} \pmod{M} = 1, \quad (1)$$

hence

$$a^{M-2} \pmod{M} = a^{-1} \pmod{M}. \quad (2)$$

Using this fact, the inverse may be calculated by modular exponentiation. However this is an expensive operation requiring on average of $1.5 \log_2 n$ multiplications for n bits.

Another method to calculate the inverse is through the knowledge that the greatest common divisor (gcd) of any two integers (a and M) may be expressed as a linear combination of two integers, say r and s . Since a and M are relatively prime, the following expression may be solved for r and s

$$a*r + M*s = 1 \pmod{M}. \quad (3)$$

This linearity implies that:

$$a*r \equiv 1 \pmod{M}. \quad (4)$$

Therefore, r is the inverse of $a \pmod{M}$. The values of r and s can be derived, by reversing the steps of the Euclidean algorithm. This procedure is known as the Extended Euclidean algorithm. The suggested algorithm is based on this extended Euclidean and binary gcd algorithm as mentioned in section 1. According to the extended Euclidean [2] algorithm, inverse of a modulo b is calculated as follows,

- Keep track of quotient
- Find integers x and y , such that $a*x + b*y = \text{gcd}(a,b)$. (5)

- If a & b relatively prime then $\text{gcd}(a,b) = \pm 1$. (6)

- using the fact in (6), (5) becomes $a*x + b*y = 1$,
 $a*x = 1 \pmod{b}$,
 $x = a^{-1} \pmod{b}$. (7)

Hence, x is multiplicative inverse of a modulo b .
Extended gcd is stated as follows:

Assuming $P(x)$ and $F(x)$ are 2 relatively prime polynomials, then according to (6) $\text{gcd}(P(x), F(x)) = 1$,

Where $F(x)$ = Irreducible polynomial
 $P(x) * S(x) + F(x) * R(x) = 1.$ (8)

Taking $\text{mod } F(x)$ of (8), it becomes
 $P(x) * S(x) = 1 \text{ mod } F(x),$ (9)

$S(x) = (P(x))^{-1} \text{ mod } F(x).$ (10)

Hence, $S(x)$ is multiplicative inverse of $P(x)$ modulo $F(x)$.

1.2 Hardware design of UMD divider

A prime field element Y in primary field i.e. $\text{GF}(p)$ is an integer in the set $\{0, 1, 2, \dots, p-1\}$, where p is an n -bit modulus in the range $2^n > p > 2^{n-1}$. A binary extension field element $Y(x)$ is a polynomial of degree less than n and greater than -1 . The pseudo code for UMD adapted to hardware implementation is depicted in Figure 1. This algorithm is based on the following conditions:

Given two numbers a and b . If a is even and b is odd then
 $\text{gcd}(a,b) = \text{gcd}(a/2,b).$ (11)

If both a and b are odd then 4 divides either $a+b$ or $a-b$.

In the first case
 $\text{gcd}(a,b) = \text{gcd}((a+b)/2,b) = \text{gcd}((a+b)/4,b),$ (12)

Inputs: $0 \leq X \leq p, 0 < Y < p, 2^{n-1} < p < 2^n$
Output: $Z = X/Y \text{ (mod } p)$
Algorithm: Divs_reg=Y, Divd_reg=X, Prime_reg=p,
 Partial_quotient = 0, $\delta = 0$

```

WHILE Divs_reg ≠ 0
  IF ( !Divs_reg[0] ) THEN
    Divs_reg <= Divs_reg /2 /*1-bit Right shift*/
    δ <= δ -1 /* 2's complement of δ*/
  ELSE
    IF (g[NBits-1]) THEN
      Swap (Divs_reg , Prime_reg)
      Swap (Divd_reg , Partial_quotient)
      δ <= -δ
    ENDIF
    k <= 1
    IF ((Divs_reg+Prime_reg) mod 4 ≠ 0) THEN
      k <= -1
    ELSE
      δ <= δ -1
    ENDIF
    Divs_reg <= (Divs_reg + k*Prime_reg)/2
    Divd_reg <= (Divd_reg + k*Partial_quotient)
  ENDIF
  Divd_reg <= (Divd_reg + Divd_reg[0]* p)/2
END WHILE
IF Prime_reg <= 1 THEN
  Quotient <= Partial_quotient
ELSE
  Quotient <= p -Partial_quotient
ENDIF

```

Figure 1: UMD Algorithm for hardware.

and

$$|(a+b)/4| \leq \max(|a/2|,|b/2|).$$

In the second case
 $\text{gcd}(a,b) = \text{gcd}((a-b)/2,b) = \text{gcd}((a-b)/4,d),$ (13)
 and
 $|(a-b)/4| \leq \min(|a/2|,|b/2|).$

The algorithm in Figure 1 performs these operations by $\text{Divs_reg} = (\text{Divs_reg} + k*\text{Prime_reg})/2$ in one iteration and $\text{Divs_reg} = \text{Divs_reg}/2$ in the following iteration. In any case since the result is stored back into Divs_reg , when $\text{Divs_reg} > \text{Prime_reg}$, the size of the bit vector Divs_reg is reduced by 1 bit by right shifting this register value. If $\text{Divs_reg} < \text{Prime_reg}$, the size of the bit vector may not be reduced and the counter variable δ is used to limit the number of iterations when the algorithm stays in this situation, forcing the swap of variables to the condition $\text{Divs_reg} > \text{Prime_reg}$, which is required for convergence. It can be shown that the two equivalences

$$\text{Partial_quotient} * Y \equiv \text{Prime_reg} * X \text{ (mod } p),$$
 (14)

and

$$\text{Divd_reg} * Y \equiv \text{Divs_reg} * X \text{ (mod } p),$$
 (15)

always hold. When the computation starts we have

$$\text{Partial_quotient} = 0,$$

$$\text{Prime_reg} = p,$$

(14) becomes

$$0 * Y \equiv X * p \text{ (mod } p) = 0.$$
 (16)

At the end of the computation, the values are

$$\text{Divs_reg} = 0,$$

$$\text{Partial_quotient} = \text{Quotient},$$

$$\text{Prime_reg} = 1 \text{ or } -1.$$

Since $\text{gcd}(\text{Divs_reg}, \text{Prime_reg}) = 1$, now (14) becomes

$$\text{Quotient} * Y \equiv X \text{ (mod } p) \Rightarrow \text{Quotient} \equiv (X/Y) \text{ mod } p.$$
 (17)

Elements shown in algorithm of Figure 1 have been represented as *register* in hardware terms. The addition/subtraction of elements in prime field is a conventional two's complement addition/ subtraction with prime modulo reduction, i.e. $\text{mod } p$. The register swapping in Verilog HDL has been achieved by generating two clock signals called *phase1* and *phase2* as can be seen from the timing diagram in Figure 2 below.

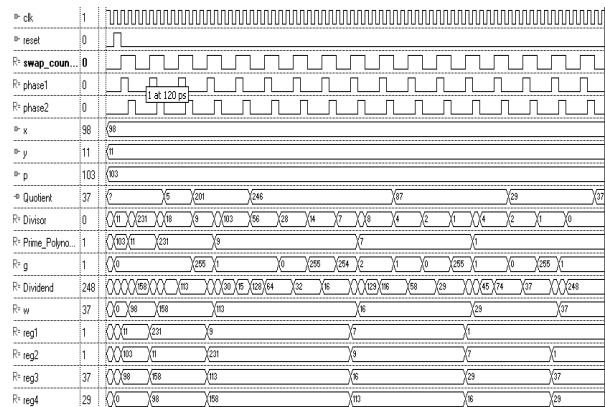


Figure 2: Simulation timing diagram of hardware UMD algorithm, for computing $(98/11) \text{ mod } 103$.

At positive edge of *phase1*, values of registers to be swapped are assigned to some temporary registers then at

the next clock cycle, which is the positive edge of *phase2*, values of those temporary register are assigned back to the original registers but in a swapped fashion, as shown in the *swapping* code below in Figure 3.

```

always @(posedge phase1 or posedge phase2)
if (Swap_counter) // Positive edge of Swap_counter
begin
if (g[NBits-1] == 1 || g == 8'b0)
begin
begin
if (phase1)
begin
reg1 <= Divisor;
reg3 <= Dividend;
reg2 <= Prime_Polynomial;
reg4 <= w;
end
else if (phase2)
begin
Prime_Polynomial <= reg1;
w <= reg3;
Divisor <= reg2;
Dividend <= reg4 ;
g <= -g;
end
end
end
end

```

Figure 3. Swapping module.

2. SIMULATION AND SYNTHESIS

Initial fixed-point verification of the UMD algorithm was developed in C-language. Hardware simulation was then developed in Verilog HDL and tested in Aldec™ environment. Finally, complete synthesis and implementation flow was done using Xilinx-XST™ tools.

The number of clock cycles required for binary division is very few and its complexity in terms of number of iterations is much less than other modular inversion techniques as already mentioned in section 1. Some performance results are described later in section 3.

An example of hardware division is shown in Figure 3 above, where the values chosen are 98 for dividend, 11 is the divisor and the prime polynomial has a value 103.

Results were verified both through C-code and Verilog HDL. The resultant quotient in this case comes out to be 37, which is also calculated below using (17) above.

$$\begin{aligned}
98/11 \text{ mod } (103) &= 37 \\
98 &= (11 * 37) \text{ (mod } 103) \\
&= 407 \text{ mod } 103 \\
&= 98
\end{aligned}$$

The C- code results are shown in Figure 4 and the Verilog HDL results are shown as timing diagram in Figure 2.

```

Dividend = U = 98
Divisor  = C = 11
Prime    = D = 103

Mod part: C: -46, D: 103, U: 98, W: 0, g: 0
u0: C: -46, D: 103, U: 49, W: 0, g: 0
u0: C: -23, D: 103, U: 76, W: 0, g: -1
Less than part: C: 103, D: -23, U: 0, W: 76, g: 1
Mod part: C: 40, D: -23, U: 76, W: 76, g: 1
u0: C: 40, D: -23, U: 38, W: 76, g: 1
u0: C: 20, D: -23, U: 19, W: 76, g: 0
u0: C: 10, D: -23, U: 61, W: 76, g: -1
u0: C: 5, D: -23, U: 82, W: 76, g: -2
Less than part: C: -23, D: 5, U: 76, W: 82, g: 2
Mod part: C: -14, D: 5, U: -6, W: 82, g: 2
u0: C: -14, D: 5, U: -3, W: 82, g: 2
u0: C: -7, D: 5, U: 50, W: 82, g: 1
Mod part: C: -6, D: 5, U: -32, W: 82, g: 1
u0: C: -6, D: 5, U: -16, W: 82, g: 1
u0: C: -3, D: 5, U: -8, W: 82, g: 0
Mod part: C: -4, D: 5, U: -90, W: 82, g: 0
u0: C: -4, D: 5, U: -45, W: 82, g: 0
u0: C: -2, D: 5, U: 29, W: 82, g: -1
u0: C: -1, D: 5, U: 66, W: 82, g: -2
Less than part: C: 5, D: -1, U: 82, W: 66, g: 2
Mod part: C: 2, D: -1, U: 148, W: 66, g: 2
u0: C: 2, D: -1, U: 74, W: 66, g: 2
u0: C: 1, D: -1, U: 37, W: 66, g: 1
Mod part: C: 0, D: -1, U: 103, W: 66, g: 1
u0: C: 0, D: -1, U: 103, W: 66, g: 1

Quotient: 37

```

Figure 4. C- Language results.

Implementation of the Verilog code on FPGA using Xilinx-XST™ Spartan II device gives the following results:

- Selected device: 2s30tq144-5
- a) Number of Registers: 15
- b) Number of Multiplexers: 3
- c) Number of Adders/Subtractors: 6
- d) Maximum operating clock frequency: 81.566MHz

3. RESULTS AND DISCUSSIONS

As already mentioned in Section 1, the complexity of this UMD hardware approach is much less than those of previous systolic architectures. Systolic architectures exist as parallel-in parallel-out, serial-in parallel-out, parallel-in serial-out and serial-in serial-out manner. The computation time of these architectures becomes impractical when the number of bits becomes large.

For testing our unified modular division algorithm, at least 100 random samples were used to verify the operations of this algorithm approach with Unified Montgomery inverse algorithm suggested in [5] and to obtain statistics.

For an element Y , this algorithm computes $Z = Y^{-1}2^k$, where $n \leq k \leq 2n$ is the number of iterations when Y has n bits. A correction step is needed to compute the inverse in the Montgomery domain ($Y^{-1}2^k$) or integer domain (Y^{-1}). Thus, the total number of iterations to compute the Montgomery and integer inverse are $2k-n$ and $2k$ respectively.

Table 1 shows the average number of iterations/ additions for the suggested algorithm (say *Alg1*) and the algorithm given in [5] (say *Alg2*) to compute modular inverse in Montgomery and integer domains for $GF(p)$.

Table 1: Average number of iterations/ additions for Montgomery and integer domains in GF(p).

GF(p) n-bits	Alg1	Alg2	Gain [%]	Alg2	Gain [%]
		Integer		Montgomery	
160	344/516	453/562	24/8.2	293/402	-17.4/ 28.3
192	410/620	544/682	24.6/9	352/490	-16.5/ 26.5
224	480/726	631/791	24/8.5	407/567	-17.9/-28
256	551/832	723/904	23.7/8	467/648	-18/-28.4
512	1105/1660	1442/1807	23.4/8	930/1295	-18.8/ 28.2

The Gain = $(Alg2 - Alg1) / Alg2$, (18)
is also calculated as pairs (iteration gain/addition gain). Note from table above that the suggested algorithm (Alg1) uses the same number of iterations and additions to compute the inverse in both domains.

Performance comparison shown in Table 1 has been shown pictorially in Figure 5 to give a clear picture for integer domain comparison. Similar graph can also be drawn for Montgomery domain.

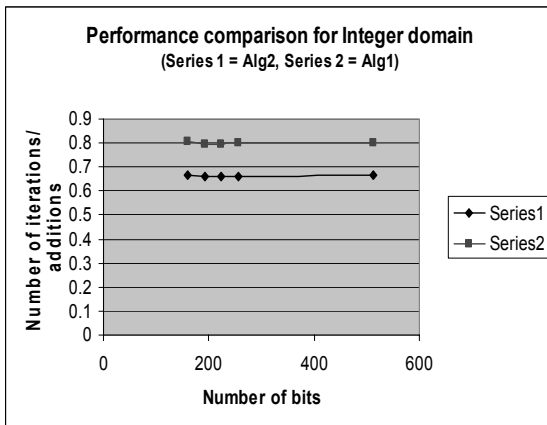


Figure 5. Performance comparison for integer domain.

It can be seen from above table and graph that Alg1 executes in about 25% less iterations than Alg2 when computing inverse in the integer domain. For computing the Montgomery inverse in GF(p), Alg1 has a maximum of 18.8% more iterations than Alg2. It can also be verified from above comparison graphs that UMD is much faster than other algorithms both for integer and Montgomery domains.

VHDL synthesis, and place and route were performed on Xilinx-XST™ ISE, as already mentioned in section 2. The results are post place and route with a top-level architecture to load the data in 16-bit words. The target FPGA device for this research is the Xilinx-XST™ Virtex Spartan II, which has 80 CLBs per column. Therefore the maximum unbroken chain length is 160 bits.

4. SUMMARY AND CONCLUSION

The unified modular hardware division implementation approach presented in this paper can be used efficiently to compute the inverse in primary field, GF(p) for both Montgomery and integer domains. The approach presented here reduces the complexity of all iteration making it faster than those of other algorithms that use comparison operations. The simplicity of the control (15 registers, 3 multiplexers, 6 adders) and the operations used in the design has made it suitable for decoding, code construction and cryptographic hardware.

To the best of our understanding, this is the first unified division/inversion algorithm to be implemented in hardware and requiring very few clock cycles. The design presented here is suitable for ASIC and PLD implementations up to 512 bits.

ACKNOWLEDGMENT

We would like to acknowledge useful email exchanges with Lo'ai Tawalbeh at Oregon State University, USA. Fareena Fiaz, is also grateful to Comsats Institute of Information Technology, Lahore for providing her financial assistance for participating in the conference.

REFERENCES

- [1] A.F. Tenca and L.A. Tawalbeh, "Algorithm for unified modular division in GF(p) and GF(2n) suitable for cryptographic hardware", Electronics Letters, Vol. 40, pp. 304-306, Dec. 2003.
- [2] Knuth, D.E.: "The art of computer programming, Volume 2, Seminumerical algorithms" (Addison-Wesley, 1998, 3rd edn).
- [3] Takagi, N.: "A VLSI algorithm for modular division based on the binary GCD algorithm", IEICE Trans. Fundam. Electron. Commun. Comput. Sci. (Japan), 1998, E81-A, (5), pp. 724 -728.
- [4] Chin-Liang Wang and Jung-Lung Lin, "A Systolic Architecture for Computing Inverses and Divisions in Finite Fields GF(2m)", IEEE Log Number 9207282, July 1, 1992.
- [5] Savas, E., and Koc, C.K.: "Architectures for unified inversion with applications in elliptic curve cryptography", 9th IEEE Int. conf. on Electronics, Circuits and Systems-ICECS'2002, Dubrovnik, Croatia, September 2002.
- [6] Goodman, J., and Chandrakasan, A.P." "An energy-efficient reconfigurable public-key cryptography processor", IEEE J. Solid State Circuits, 2001, 36, (11), pp. 1808-1820.
- [7] Wolkerstorfer, J.: P. "Dual field arithmetic unit for gf(p) and gf(2^n) in Kaliski, B.S. Jr. et al. (Ed.): "Cryptographic Hardware and Embedded Systems, CHES 2002" Lect. Notes in Comput. Sci., 2003, 2523, pp. 484-499.