Polynomial Inversion Over GF(2^m) Based on Extended Euclidean Algorithm

University of Victoria

FPGA Design and Implementation

Prepared by: Ibrahim Hazmi

Supervised by: Dr. Fayez Gebali

Contents

Abstract

The objective of the project was to design a "Polynomial Inverter" over binary field based on the extended Euclidean algorithm (EEA). The main focus in the project was on the functionality of the design, which has been verified in ISE simulation and implemented successfully on Xilinx Spartan 3E FPGA.

The project has been divided into three stages. First, designing the binary division algorithm (BDA), which was the main building block of the inversion circuit and the bottle nick of the design as well. The ultimate goal in this stage was to produce the remainder and the quotient in a minimum number of cycles. Computing the quotient was the most challenging part of the BDA design as the idea was to utilize degree difference of the polynomials in order to obtain the right number of required shifts each cycle. There were different attempts to design the BDA, starting with a spreadsheet, which works similarly to the manual procedure and was used as a reference. Then a C code was written to realize the idea in software before implementing the circuit in hardware using VHDL.

The second stage was plugging the BDA into the GCD circuit, which consists primarily of two multiplexers and a register in addition to the BDA component. Finally, the stage of finalizing the EEA circuitry by connecting the GCD module with the inversion extension, which uses a polynomial multiplier and adder to reverses each step in the Euclidean algorithm at the same time to compute the inverse of the input polynomial.

Project Overview

Objective

The objective of this project was to design a Digital Circuit that computes the Modular Inverse of Polynomial function over Binary Field. The Polynomial binary field Inverter is based on Extended Euclidean Algorithm (EEA). The design entry method is VHDL and the target implementation platform is Xilinx Spartan3E FPGA.

Introduction

Finite field arithmetic is considerably important for the field of information security. For instance, performing the scalar point multiplication in elliptic curve cryptography (ECC) requires executing algorithms based on finite field arithmetic. Implementing field arithmetic, e.g., polynomial inversion over binary field $GF(2^m)$, on hardware platforms is quite more challenging than performing them in a software environment. However, hardware implementation allows for high performance as there are great opportunities to occupy less area, perform faster operations, consume less power, or to obtain a reasonable combination of all of these. In general, inversion over finite field is the most expensive operation to be implemented in hardware [1].

The finite field $GF(2^m)$ can be represented by a polynomial function with respect to a variable **x**, as follows: $\boldsymbol{GF(2^m) = x^{m-1} + x^{m-2} + ... + x^2 + x^1 + x^0}$. In polynomial arithmetic , an irreducible polynomial of degree m is used to reduce the result of any operation if its degree is greater than *m-1*. In many standard implementations of ECC operation, trinomial (3 terms polynomial), or pentanomial (5 terms polynomial), are used for more efficiency [1]. The multiplicative inverse of a polynomial $a(x)$ in $GF(2^m)$ is defined as the computation process to find a polynomial $a^{-1}(x)$ in $GF(2^m)$, so $a(x) \cdot a^{-1}(x)$ **modulo** $f(x) = 1$. Inversion algorithms can be classified into two main categories; the Extended Euclidean Algorithm based (**EEA**-based) algorithms, and the Fermat's Little Theorem based (FLT-based) ones [2].

Background on the extended Euclidean algorithm (EEA)

The **EEA** generally computes the multiplicative inverse of $a(x)$ **modulo** $f(x)$, if the degree of $a(x)$ is less than the degree of $f(x)$ and the ***gcd*** **[**$f(x)$, $a(x)$**]** equals one. **EEA** is an extension of euclidean algorithm that computes the greatest common divider (**GCD**), which is based on the fact that the **GCD** of two numbers divides the remainder of the division between them [3]:

$$gcd\,[f(x), a(x)] = gcd\,[a(x), r(x)], \quad gcd\,[f(x), a(x)] = gcd\,[a(x), r_0(x)]$$

$$\text{where } f(x) = [a(x) \times q(x)] \oplus q(x) \quad gcd\,[a(x), r_0(x)] = gcd\,[r_0(x), r_1(x)]$$

Binary Polynomial Divisions (BPDs) are done iteratively until $\boldsymbol{r_n(x) = 0}$ is obtained, then the **GCD** equals $\boldsymbol{r_{n-1}(x)}$. If $f(x)$ is an irreducible polynomial and ($f(x) \neq x$), then $\boldsymbol{gcd}\,[f(x), a(x)] = \boldsymbol{1}$. In addition to computing the **GCD** of the polynomials $a(x)$ and $f(x)$, the **EEA** produces the coefficients of Bézout's identity; $\boldsymbol{r(x)}$ and $\boldsymbol{s(x)}$ such that:

$$\boldsymbol{1} = gcd\,[a(x), f(x)] = [\boldsymbol{r(x)} \cdot a(x) + \boldsymbol{s(x)} \cdot f(x)]\ mod\ f(x),$$

Since $[s(x) \cdot \boldsymbol{f(x)}]\ mod\ \boldsymbol{f(x)} = 0$, we have:

$$\boldsymbol{r(x)} \cdot a(x) = \boldsymbol{1}\ mod\ f(x), \text{ or } [a(x) \cdot \boldsymbol{r(x)}]\ mod\ f(x) = \boldsymbol{1}.$$

Therefore, $\boldsymbol{r(x)}$ is the multiplicative inverse of a polynomial $a(x)$ **modulo** $f(x)$, i.e., $a^{-1}(x)$, and $\boldsymbol{s(x)}$ is a multiplicative inverse of $f(x)$ **modulo** $a(x)$ [4][5].

Project Description and Project Milestones

It has been argued that the performance of the multiplication based (**FLT-based**) inversions makes them more utilized in field arithmetic, although their hardware design complexity is not attractive. For instance, a study in [2] shows that over 75% of designers prefers **FLT-based** inverters, e.g., Itoh-Tsujii, due to their high speed. On the other hand, **EEA-based** ones can be preferable alternatives only as a companion to the affine coordinate system in case of ECC scalar multiplier. Otherwise, their performance is still considered uncompromising in many other cases.

In this project, there was an attempt to design a binary polynomial EEA-based inverter that performs the inversion process in parallel with the GCD computation. Therefore, it could be a promising step towards the realization of a competitive EEA-based inverters in terms of performance/area tradeoff. The design of this inverter has started starting from its bottleneck, the division algorithm, which was hypothesized to produce the quotient and the remainder simultaneously in a minimum number of cycles. Since the data-path was constructed accordingly, a 4-state FSM controller was enough to execute the EEA inversion. These four states include the initialization and the output states.

The design specifications are listed in (**TABLE 1**), while (**TABLE 2**) is an outline of the project milestones, and the achieved percentage of them during the journey of the ELEC543 course.

TABLE 1: DESIGN SPECIFICATIONS

I/O Specification	**Input**: A nonzero binary polynomial *a(x)* of degree at most *m –1* **Output**: binary polynomial $a(x)^{-1}$ *mod f*
Circuit Function	The design is simply an EEA data-path unit and FSM controller (**Fig.1**).
Algorithm(s)	The algorithms to be explored are the Extended Euclidean Algorithm (**EEA**) (**Algorithm 1**), and the Binary Polynomial Division (**BPD**) (**Algorithm 2**).
Performance	To be determined by functionality and latency at this stage, then it would be part of the future work to reduce the latency-area product.

Algorithm 1: EEA	**Algorithm 2: BPD**
INPUT: Binary polynomials **a(x)** of max degree m –1 & **f(x)** of m degree **OUTPUT**: a^{-1} mod f. 1. **a**←a(x), **f**← f(x) . 2. **g1**←0, **g2**←1. 3. **WHILE** r≠0 (gcd ≠1) do 3.1 perform **BPD** to find **r** & **q** (f=a×q+r). 3.2 **g1**←g2, **g2**←(g2×q) ⊕ g1. 3.3 **f**←a, **a**←r. 4. **RETURN** (**g2**).	**INPUT**: **a(x)** & **f(x), OUTPUT**: **r** and **q**. 1. **a**←a(x), **f**← f(x), **r**←1, **q**←1. 2. **WHILE** (f_deg > a_deg) **DO** 2.1 **dfa**← f_deg - a_deg. 2.2 **a**← a <<dfa. 2.3 **r**← a xor f. 2.4 **dfr**← f_deg – r_deg. 2.5 **IF** (r_deg>a_deg) **THEN q**←(q<<dfr)+1. 2.6 **ELSE q**←q<<dfa. 3. **RETURN** (**r, q**).

TABLE 2: PROJECT MILESTONES

#	Description	Done
1	Define the field, and the algorithm, and suggest a functional block diagram for the design	100
2	Realize the BPD algorithm in a spreadsheet and a C-code as design references	100
3	Design the BPD circuit in VHDL and examine its functionality (DataPath + FSM)	100
4	Construct the simple GCD circuit using BPD module as the main building block	100
5	Build EEA module that satisfies the stated specifications and perform a practical simulation	100
6	Implement the Design on Xilinx Spartan3E FPGA and test it using NEXYS2 board	100

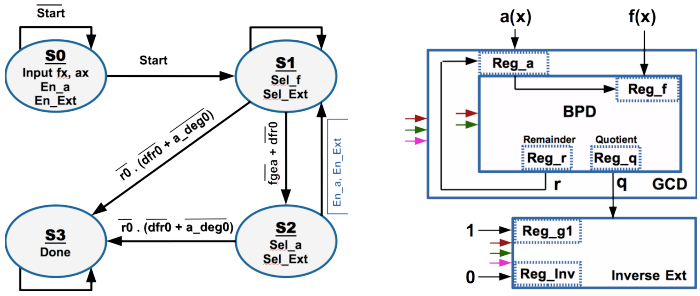


Fig.1 EEA data-path unit and FSM controller

Design methodology and alternatives

This implementation of the EEA inversion was performed as a direct interpretation of the algorithm. The main focus in the project was on the **functionality** of the design, which has been satisfied in ISE simulation and implemented successfully in Xilinx Spartan 3E **FPGA**. Different design strategies have been utilized within the design, i.e., hierarchy, abstraction, modularity, and locality [8]. The design process has started from the bottom to the top level of the hierarchy, i.e., from the BPD to the EEA, while for abstraction, there was utilization of the behavioural and structural strategies. Furthermore, the design has several modules that could be reused within and outside the project involving leading-zeros counters (CLZs), Barrel shifters and absolute difference units (ADUs). The CLZ searches for the most significant set bit in a word of 32-bit to decide the degree of the polynomial, and the ADU, which is created using a carry-chain adder and carry generator, determines the degree difference. Whereas, the Barrel shifters are used to obtain the accurate shift required for the algorithm operations. Multiplexers and registers were also modularized and generalized with variable operand sizes using ***generic*** statement. Finally, some modules, such as ADUs, were localized to reduce the interconnection jumps. For example, the primitives *MUXCY* and *XORCY* were connected as a carry-chain to perform 4-bit ripple carry addition within one slice inside the FPGA. (**TABLE 3**) presents the hardware requirements for the design in general manner.

TABLE 3: HARDWARE REQUIREMENTS

Level	Components				
EEA (Top Level)	BPD	Extension	2 Multiplexers	1 Register	FSM Controller
Extension	Multiplier	XOR	2 Multiplexers	2 Registers	-
Binar Polynomial Divider (BPD)	3 Count-Leading Zeros (CLZs)	2 Subtractors and 1 XOR	5 Multiplexers	3 Registers	3 Barrel Shifters

Alternatively, linear-feedback shift register (LFSR) can be used for the binary polynomial division instead of our BPD. However, the latency of LFSR is quite higher [8]. In the top level. It is also possible to use regular integer subtractors instead of out ADUs and use the carry-out signal as a comparison signal between the two inputs. Finally, CLZ circuit has been designed using simple search algorithm. An alternative to such circuit would be a binary tree search mechanism, which is faster and area saving choice. There are many other multipliers than Karatsuba's to try in the future as a potential alternatives. **In the next section, the design will be discussed in details.**

The Design in Details

As, the design has started from the bottom level of the hierarchy, i.e., the BPD, this report will start with the BPD module, which accepts two inputs, *dividend (f), divisor (a),* and produces two outputs, *quotient (q)* and *remainder (r)*. Another output signal in "Finish" signal, which indicates that the binary polynomial division (BPD) is done and *(q & r)* are ready for the next round in the Euclidean algorithm (GCD). Then the extension, i.e., the inversion module, will be discussed focusing on the polynomial multiplication process and the time allocation for the control signals. Finally, the top-level, i.e., EEA module, will be presented highlighting the main tricks that has been applied to allow for parallel processing for GCD and EGCD at the same time.

Binary Polynomial Divider (BPD) Design

TABLE 4: BPD SPECIFICATIONS

Fig.2 BPD Block Diagram			
a(x)	in	std_logic_vector	divisor
f(x)	in	std_logic_vector	dividend
reset	in	std_logic	asynchronous
clock	in	std_logic	clock input
Sel_f	in	std_logic	from FSM
fgea	out	std_logic	to FSM
a_d0	out	std_logic	to FSM
fr0	out	std_logic	to FSM
r	out	std_logic_vector	remainder
q	out	std_logic_vector	quotient

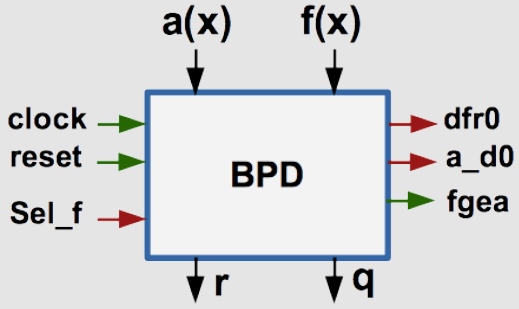


Fig.3 BPD Flowchart

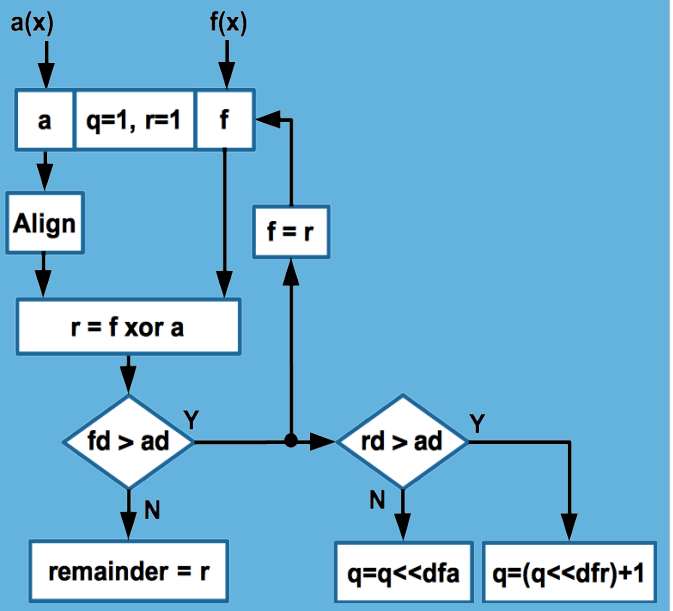
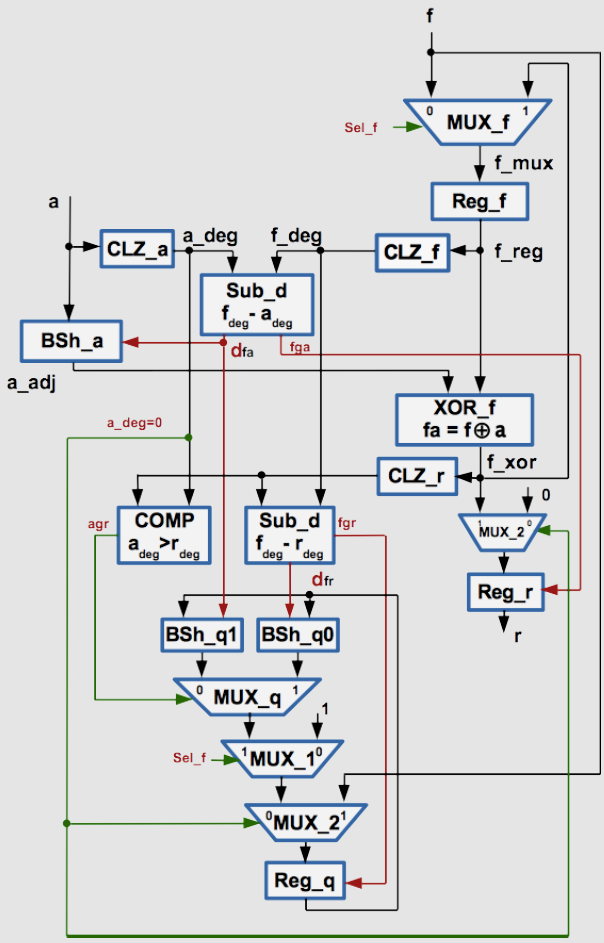


TABLE 5: BPD CIRCUIT DIAGRAM AND COMPONENTS

Fig.4 BPD Circuit



Multiplexer Entity

```
entity MUX is
  Port ( in0, in1 : in  STD_LOGIC_VECTOR (N-1 downto 0);
         Sel : in  STD_LOGIC;
         result : out  STD_LOGIC_VECTOR (N-1 downto 0));
end MUX;
```

Register Entity

```
entity RegN is
  Port ( D : in  STD_LOGIC_VECTOR (N-1 downto 0);
         clock, reset, En : in  STD_LOGIC;
         Q : out  STD_LOGIC_VECTOR (N-1 downto 0));
end RegN;
```

Comparator Entity

```
entity Cmp is
  Port (   A,R : in  STD_LOGIC_VECTOR (N-1 downto 0);
           AGR : out  STD_LOGIC;);
end Cmp;
```

Barrel Shifter Entity

```
entity bshift is -- barrel shifter
  Port ( shift: in STD_LOGIC_VECTOR (7 downto 0);-- #shift
         input : in STD_LOGIC_VECTOR (31 downto 0);
         output : out STD_LOGIC_VECTOR (31 downto 0));
end ADU;
```

ADU/Subtractor Entity

```
entity ADU is
  Port (   I1, I2 : in  STD_LOGIC_VECTOR (N-1 downto 0);
           GN : inout  STD_LOGIC;
           SUB : out  STD_LOGIC_VECTOR (N-1 downto 0));
end ADU;
```

CARRY4: 4-bit Ripple Adder (Carry-Chain)

```
entity Carry4 is
  Port ( S, DI : in  STD_LOGIC_VECTOR (3 downto 0);
         CI : in  STD_LOGIC;   CO : inout  STD_LOGIC;
         O : out  STD_LOGIC_VECTOR (7 downto 0));
end Carry4;
architecture Structural of Carry4 is
begin
MUXCY_0: MUXCY port map (O => CO(0), CI => CI, DI =>
DI(0), S => S(0));
XORCY_0: XORCY port map (O => O(0), CI => CI, LI =>
S(0)); -- O(0) <= S(0) xor CI;
MUXCY_1: MUXCY port map (O => CO(1), CI => CO(0), DI
=> DI(1), S => S(1));
XORCY_1: XORCY port map (O => O(1), CI => CO(0), LI =>
S(1)); -- O(1) <= S(1) xor CO(0);
MUXCY_2: MUXCY port map (O => CO(2), CI => CO(1), DI
=> DI(2), S => S(2));
XORCY_2: XORCY port map (O => O(2), CI => CO(1), LI =>
S(2)); -- O(2) <= S(2) xor CO(1);
MUXCY_3: MUXCY port map (O => CO(3), CI => CO(2), DI
=> DI(3), S => S(3));
XORCY_3: XORCY port map (O => O(3), CI => CO(2), LI =>
S(3)); -- O(3) <= S(3) xor CO(2);
end Structural;
```

Leading-Zeros Counter (CLZ) VHDL

```
entity CLZ is
   generic (N : integer);
   Port ( Input : in  STD_LOGIC_VECTOR (N-1 downto 0);
          Degree : out  STD_LOGIC_VECTOR (7 downto 0));
end CLZ;
architecture Behavioural of CLZ is
begin -- Simple Search, can be replaced by binary search
process (Input)
begin
for i in Input'range loop
 if (Input(i) = '1') then
   Degree <= STD_LOGIC_VECTOR(to_unsigned(i,8)); exit;
 end if;
end loop;
end process;
end Behavioural;
```

TABLE 6: BPD VHDL

```
entity PolyDiv is
   Generic (n: integer:= 32);
   Port ( a, f : in  STD_LOGIC_VECTOR (n-1 downto 0);
        clock, reset, Sel_f : in  STD_LOGIC;
        remainder, quotient : out  STD_LOGIC_VECTOR (n-1 downto 0);
        fgea, a_deg0, dfr0 : inout  STD_LOGIC);
end PolyDiv;
architecture Structural of PolyDiv is
Signal Sel_q, En_q, fga, agr, fgr, dfa0, xor0 : STD_LOGIC;
Signal a_deg, f_deg, r_deg, dfa, dfr, dar, dar2: STD_LOGIC_VECTOR(7 downto 0);
Signal f_xor, f_mux, f_reg, a_adj, q_mux0, q_mux, r, q_reg, q00, q0, q1: STD_LOGIC_VECTOR(n-1 downto 0);
begin
-- Remainder Computation Part
MUX_f: entity work.MUX2 Generic Map (32) Port Map (f, f_xor, Sel_f,f_mux);
Reg_f: entity work.RegN Generic Map (32) Port Map (clock, reset, '1', f_mux, f_reg); -- Always enabled
CLZ_a: entity work.clz Generic Map (32) Port Map (a, a_deg); -- Leading Zero/MSB Set for (f, a, and f xor a)
CLZ_f: entity work.clz Generic Map (32) Port Map (f_reg, f_deg);
CLZ_x: entity work.clz Generic Map (32) Port Map (f_xor, r_deg);
Sub_d: entity work.ADU Port Map (f_deg, a_deg, fga, dfa); -- If_deg - a_degl Computing the degree differences for adjusting
Sub_q: entity work.ADU Port Map (f_deg, r_deg, fgr, dfr); -- If_deg - r_degl  the divisor (a) and computing the quotient (q)
Cmp_a: entity work.cmp Generic Map (8) Port Map (a_deg, r_deg, agr); -- a_deg < r_deg
Adj_a: entity work.bshift Port Map (dfa, a, a_adj); -- [a_adj = a << dfa] Align the divisor (a) with the dividend (f) for the xor
f_xor <= a_adj xor f_reg; -- dividend XOR aligned divisor
Reg_r: entity work.RegN Generic Map (32) Port Map (clock, reset, fgea, f_xor, r); -- Remainder Register
-- Quotient Computation Part
dfr0 <= dfr(7) or dfr(6) or dfr(5) or dfr(4) or dfr(3) or dfr(2) or dfr(1) or dfr(0); -- an output signal
dfa0 <= dfa(7) or dfa(6) or dfa(5) or dfa(4) or dfa(3) or dfa(2) or dfa(1) or dfa(0); -- Generating q Select and Enable Signals
xor0 <= f_xor(7) or f_xor(6) or f_xor(5) or f_xor(4) or f_xor(3) or f_xor(2) or f_xor(1) or f_xor(0);
fgea <= fga or not(dfa0); -- an output signal
Sel_q <= agr or not(xor0); -- When dar=0, we need Sel_q=0 Sel_q=1 when a>r
En_q <= not (Sel_f) or fga; -- Enable reg_q at S0 and whenever dfa>=1
adq_0: entity work.bshift Port Map (dfr, q_reg, q00); -- q0 = (q << dfr) + 1 IF dfr>=1 and a_deg <= r_deg
q0 <= q00(n-1 downto 1) & '1';
adq_1: entity work.bshift Port Map (dfa, q_reg, q1); -- q1 = q << dfa IF dfr>=1 and a_deg > r_deg
MUX_q: entity work.MUX2 Generic Map (32) Port Map (q0, q1, Sel_q, q_mux0); -- clz(r)>clz(a) is the Sel signal
MUX_1: entity work.MUX2 Generic Map (32) Port Map (x"00000001", q_mux0, Sel_f, q_mux); -- q=1 at S0, the initialization
Reg_q: entity work.RegN Generic Map (32) Port Map (clock, reset, En_q, q_mux, q_reg); -- Quotient Register
a_deg0 <= a_deg(7) or a_deg(6) or a_deg(5) or a_deg(4) or a_deg(3) or a_deg(2) or a_deg(1) or a_deg(0); -- Special Case:
MUX_ra1: entity work.MUX2 Generic Map (32) Port Map (x"00000000", r, a_deg0, remainder); -- If a = 1 then q = f and r = 0
MUX_qa1: entity work.MUX2 Generic Map (32) Port Map (f, q_reg, a_deg0, quotient); -- When a_deg = 0, i.e. a=1
end Structural;
```

The Extension Module Design

TABLE 7: EXTENSION SPECIFICATIONS

Fig.5 Extension Block Diagram			
a(x)	in	std_logic_vector	divisor
q	in	std_logic_vector	quotient
r	in	std_logic_vector	remainder
reset	in	std_logic	asynchronous
clock	in	std_logic	clock input
Sel	in	std_logic	MUXs select
En	in	std_logic	Registers En
$a^{-1}(x)$	out	std_logic_vector	inverse

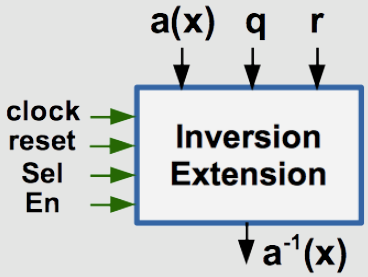
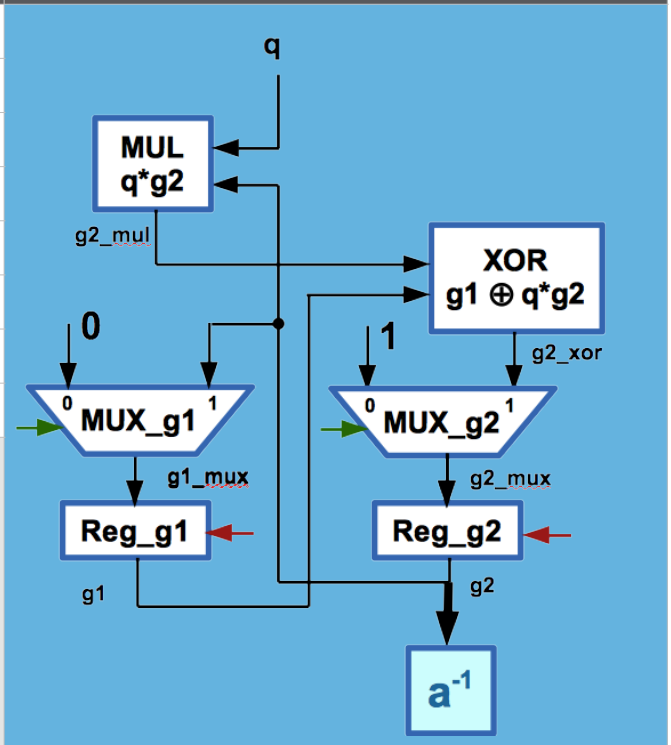


Fig.6 Extension Circuit Diagram



Extension Module VHDL

```
entity Ext_Poly is
  generic (N : integer);
  Port (a, r, q: in STD_LOGIC_VECTOR (N-1 downto 0);
        clock, reset, Sel, En : in  STD_LOGIC;
     a_inv : out  STD_LOGIC_VECTOR (N-1 downto 0));
end Ext_Poly;

architecture Structural of Ext_Poly is
Signal  g1,g1_mux,g2,g2_mul,g2_xor,g2_mux:
STD_LOGIC_VECTOR(n-1 downto 0);
begin
MUX_g1: entity work.MUX2 Generic Map (32)
   Port Map (x"00000000", g2, Sel, g1_mux); -- MUXs
MUX_g2: entity work.MUX2 Generic Map (32)
   Port Map (x"00000001", g2_xor, Sel, g2_mux);
Reg_g1: entity work.RegN Generic Map (32)
   Port Map (clock, reset, En, g1_mux, g1); -- Registers
Reg_g2: entity work.RegN Generic Map (32)
   Port Map (clock, reset, En, g2_mux, g2);
Mul_g: entity work.PolyMul Generic Map (32)
   Port Map (q, g2, g2_mul); --Multiplier
g2_xor <= g1 xor g2_mul; -- XOR
a_inv <= g2;
end Structural;
```

Polynomial Multiplier (Karatsuba) VHDL

```
-- Book: Hardware Implementation of Finite-Field Arithmetic
entity PolyMul is -- Modified Code: 32-bit output instead of 64
  generic (N : integer);
  Port (a, b: in STD_LOGIC_VECTOR (N-1 downto 0);
        d : out  STD_LOGIC_VECTOR (N-1 downto 0));
end PolyMul;
architecture simple of PolyMul is
type matrixand is array (0 to N-1) of STD_LOGIC_VECTOR(M-1
downto 0);
Signal a_by_b: matrixand;
begin
gen_ands: for k in 0 to M-1 generate l1: for i in 0 to k generate
   a_by_b(k)(i) <= A(i) and B(k-i); end generate; end generate;
d(0) <= a_by_b(0)(0);
gen_xors: for k in 1 to M-1 generate
l3: process(a_by_b(k))
  variable aux: std_logic;
  begin
   if (k < M) then  aux := a_by_b(k)(0);
    for i in 1 to k loop aux := a_by_b(k)(i) xor aux; end loop;
   else aux := a_by_b(k)(k);
    for i in k+1 to M-1 loop aux:= a_by_b(k)(i) xor aux; end loop;
   end if;
    d(k) <= aux;
 end process; end generate;
end simple;
```

Top-Level Design: The Extended Euclidean Algorithm (EEA)

TABLE 8: TOP LEVEL SPECIFICATIONS

Fig.7 EEA Block Diagram			
a(x)	in	std_logic_vector	input poly
f(x)	in	std_logic_vector	irreducible
reset	in	std_logic	asynchronous
clock	in	std_logic	clock input
Start	in	std_logic_vector	Controller
$a^{-1}(x)$	out	std_logic_vector	inverse

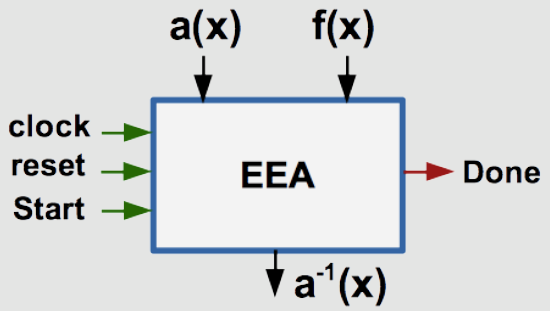
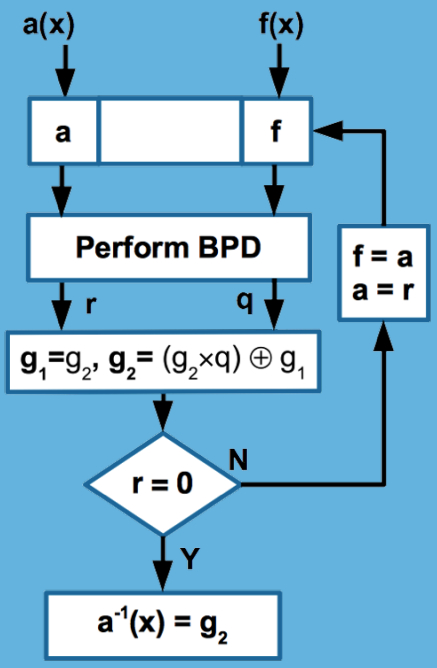


Fig.8 EEA Flowchart



EEA Module VHDL

```
entity PolyEEA is
  generic (N : integer:= 32);
  Port (a, f: in STD_LOGIC_VECTOR (N-1 downto 0);
        clock, reset, Start : in  STD_LOGIC;
        a_inv : out  STD_LOGIC_VECTOR (N-1 downto 0);
        Done : out  STD_LOGIC;);
end PolyEEA;

architecture Structural of PolyEEA is
Signal Sel_f,  Sel_a, En_a, fgea, a_deg0, dfr0, r0, Sel_Ext, En_Ext : STD_LOGIC;
Signal ax_mux, a, f, r, q, GCD: STD_LOGIC_VECTOR(n-1 downto 0);
begin
-- GCD Block:
-- MUXs
MUX_fx: entity work.MUX2 Generic Map (32) Port Map (fx, a, Sel_a, f);
MUX_ax: entity work.MUX2 Generic Map (32) Port Map (ax, r, Sel_a, ax_mux);
-- reg_a
Reg_a: entity work.RegN Generic Map (32) Port Map (clock, reset, En_a, ax_mux, a);
--PolyDiv
Div_P: entity work.PolyDiv Generic Map (32) Port Map (a, f, clock, reset, Start, Sel_f, r, q, fgea, a_deg0, dfr0);
-- remainder = 0
r0 <= r(7) or r (6) or r(5) or r (4) or r(3) or r (2) or r(1) or r (0);
--Poly Extension Block:(Extended Euclidean)
Ext_P: entity work.Ext_Poly Generic Map (32) Port Map (ax, r, q,  Sel_Ext, En_Ext, clock, reset, Start, a_inv); -- En_a/fga
-- FSM Controller
FSM_0: entity work.Poly_FSM Port Map (clock,reset,Start,fgea,a_deg0,dfr0,r0,En_a,Sel_a,Sel_f,En_Ext,Sel_Ext,Done);
GCD <= a;
end Structural;
```

TABLE 9: EEA CIRCUIT DIAGRAM AND FSM CONTROLLER

Fig.9 EEA Circuit Diagram

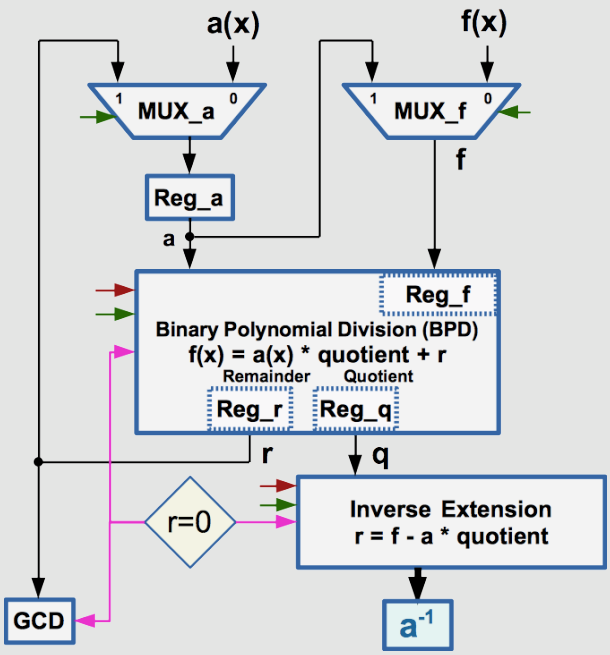


Fig.10 FSM-Controller State Machine

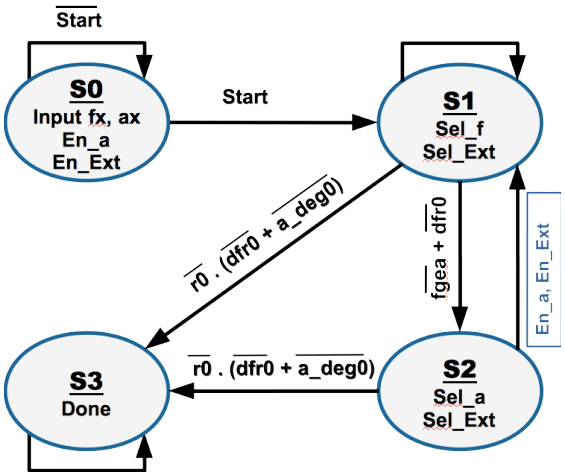
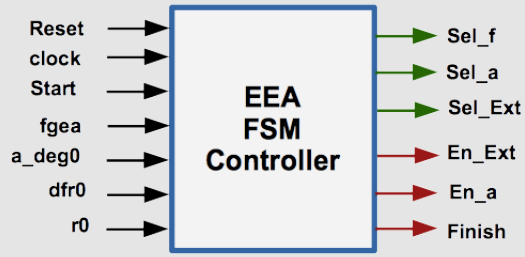


Fig.11 FSM-Controller Block Diagram



FSM-Controller VHDL

```
entity Poly_FSM is
Port( clock, reset, Start, fgea, a_d0, dfr0, r0: in Std_Logic;
En_a, Sel_a, Sel_f, En_Ext, Sel_Ext, Finish: out Std_Logic);
end Poly_FSM;
architecture behave of Poly_FSM is
 type StateType is (S0, S1, S2, S3);
 Signal CurrentState : StateType; -- C_S
 Signal NextState : StateType; -- N_S
begin
 C_S: process; (clk, Reset)
 begin
  if (reset = '1') then
      CurrentState <= S0;
  elsif (rising_edge(clock)) then
      CurrentState <= NextState;
  end if;
 end process C_S;
 N_S: process (CurrentState, Start, fgea, a_deg0, dfr0, r0)
 begin
    case CurrentState is
    WHEN S0 =>
          Finish <= '0';
          Sel_a <= '0'; Sel_f <= '0'; Sel_Ext <= '0';
          En_a <= '1'; En_Ext <= '1';
          if (Start = '1') then
             NextState <= S1;
          else
             NextState <= S0
          end if;
   WHEN S1 =>
          Sel_f <= '1'; Sel_Ext <= '1';
          En_a <= '0'; En_Ext <= '0';
          if (r0 = '0' AND (dfr0 = '0'  OR a_deg0 = '0')) then
              Finish <= '1';
              NextState <= S3;
          else
              Finish <= '0';
              if (fgea = '0' OR dfr0 = '0') then
                    NextState <= S2;
              else
                    NextState <= S1;
              end if;
          end if;
    WHEN S2 => -- a=r, f=a
          Sel_a <= '1'; Sel_f <= '0'; Sel_Ext <= '1';
          Finish <= '0';
          if (r0 = '0' AND (dfr0 = '0'  OR a_deg0 = '0')) then
              En_a <= '0'; En_Ext <= '0';
              NextState <= S3;  -- to the output state
          else
              En_a <= '1'; En_Ext <= '1';
              NextState <= S1;
          end if ls;
     WHEN S3 => -- output state
          Finish <= '1';
          NextState <= S3;
   end case;
 end process N_S;
end behave;
```

Simulation

Here is an example of how the EEA algorithm works in this design showing the four important registers for the whole operation of finding the inverse of *a(x) mod f(x)*.

TABLE 10: EEA SIMULATION EXAMPLE

f(x) = $x^{20} + x^5 + 1$ = x"0100021" a(x) = $x^{19} + x^{18} + x^{17} + x^{16}\ x^7 + x^2$ = x"00F0084"

Itr	r	q	g1	g2
0	00000001	00000001	00000000	00000001
1	000101AD	00000003	00000001	00000003
2	000009AF	0000000F	00000003	00000010
3	000001AF	00000026	00000010	00000263
4	0000009A	0000000F	00000263	00001C21
5	**00000001**	00000003	00001C21	**00002600**

The example above shows that the multiplicative inverse of (x**00F0084** *mod* x**0100021**) is (x**0002600**) as well as the **Simulation** in (**Fig.12**) and the **Hardware demonstration** in (**Fig.17**)

Fig.12 Simulation Results for EEA Design (Numbers in Hex)

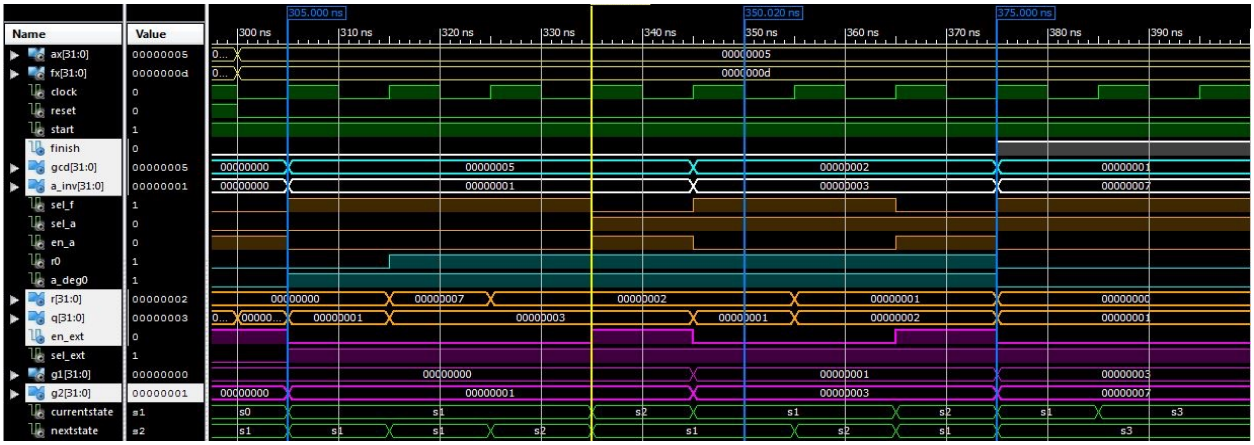
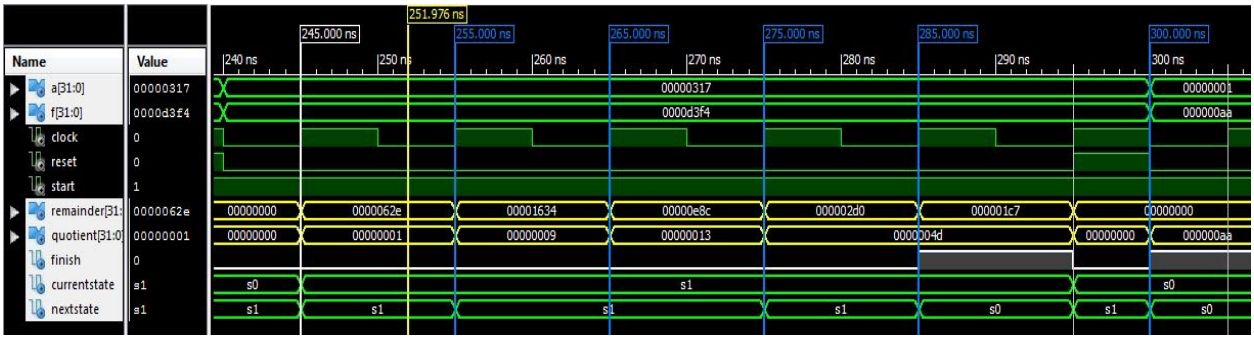


Fig.13 Simulation Details for another Example of EEA Design

Here is another example for the BPD algorithm and its simulation results independently, whose **Simulation** is shown in (**Fig.14**) and (**Fig.15**).

TABLE 11: BPD SIMULATION EXAMPLE

f(x) = x"000d3f4" a(x) = x"0000317"



ltr	a	f	r	q
0	00000317	0000D3F4	XXXXXXX	00000001
1	00000317	00001633	00001634	00000009
2	00000317	00000E8C	00000E8C	00000013
3	00000317	000002D0	000002D0	0000004D
4	XXXXXXXX	XXXXXXXX	**000001C7**	**0000004D**

Fig.14 Simulation Example for BPD Design

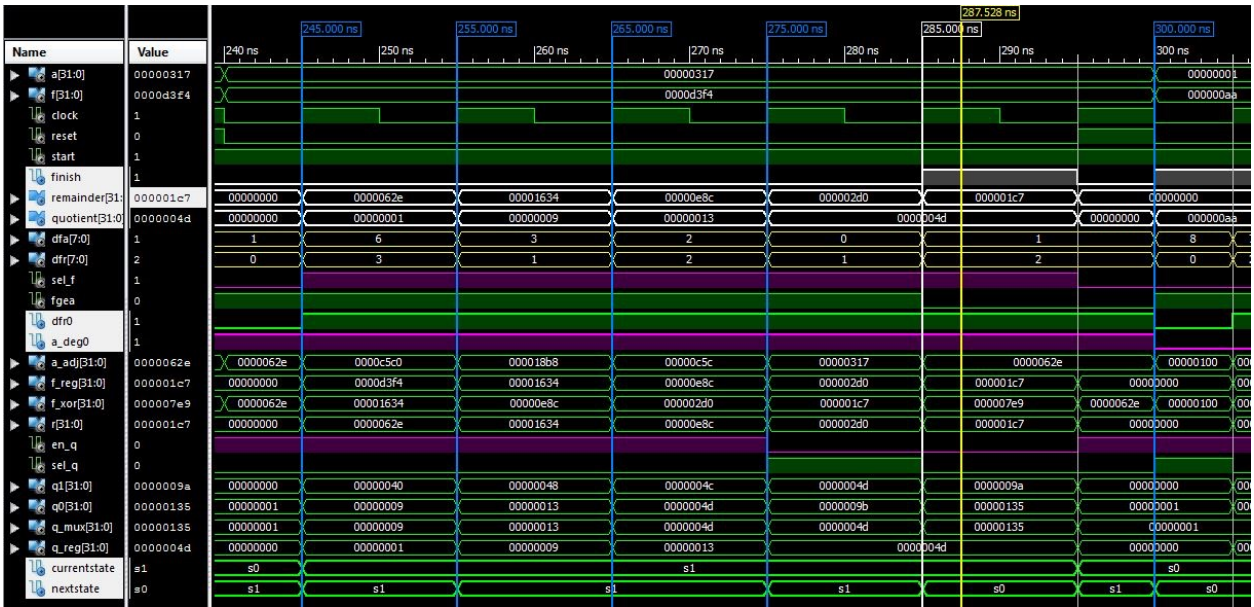


Fig.15 Simulation Details for BPD Design

Hardware Implementation

The design has been implemented successfully on Xilinx Spartan3E FPGA and was tested using NEXYS2 board.

There was a need to add two extra modules for the sake of demonstration, an input selector and a display controller. The content of EEA.ucf file is shown bellow:

NET "Start" LOC=H13 NET "reset" LOC=B18 NET "clock" LOC=U9	NET "an[0]" LOC=F17 NET "an[1]" LOC=H17	NET "sseg[0]" LOC=L18 NET "sseg[1]" LOC=F18	NET "sseg[4]" LOC=G14 NET "sseg[5]" LOC=J17
NET "s[0]" LOC=G18 NET "s[1]" LOC=H18	NET "an[2]" LOC=C18 NET "an[3]" LOC=F15	NET "sseg[2]" LOC=D17 NET “sseg[3]” LOC=D16	NET "sseg[6]" LOC=H14 NET “Done" LOC=P4

And here are some snapshots for the design working on **NEXYS2 board.**

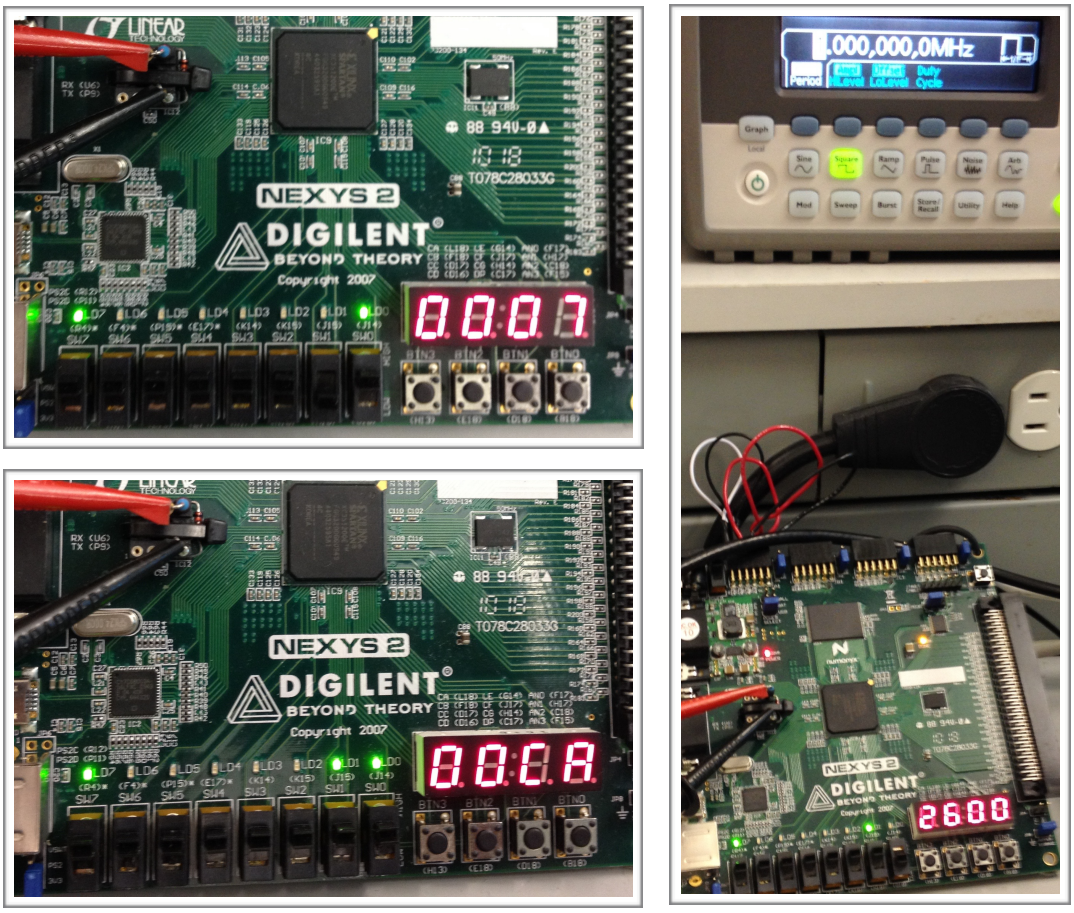
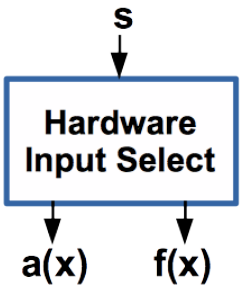
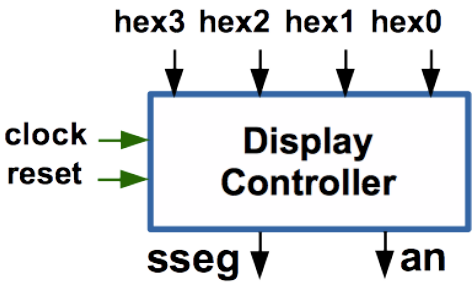


Fig.16 EEA hardware Demonstration

TABLE 11: HARDWARE EXTENSIONS

Fig.17 Hardware Input				Fig.18 Hardware Output			
s	in	std_logic_vector	input selector	**hex16**	in	std_logic_vector	4 hex inputs
reset	in	std_logic	asynchronous	**reset**	in	std_logic	asynchronous
clock	in	std_logic	clock input	**clock**	in	std_logic	clock input
a(x)	out	std_logic_vector	Controller	**an**	out	std_logic_vector	7-seg selector
f(x)	out	std_logic_vector	inverse	**sseg**	out	std_logic_vector	7-seg inputs





Input Controller VHDL

```
entity Mux4x1 is
Port ( s : in STD_LOGIC_VECTOR (1 downto 0);
       ax, fx : out STD_LOGIC_VECTOR (n-1 downto 0));
end Mux4x1;
architecture Behavioural of Mux4x1 is
begin process (s) begin case s is
when "00" => ax <= x"00000002"; fx <= x"00000007";
when "01" => ax <= x"00000005"; fx <= x"0000000d";
when "10" => ax <= x"000f0084"; fx <= x"00100021";
when "11" => ax <= x"00000053"; fx <= x"0000011b";
end case; end process;
end Behavioural;
```

EEA Module VHDL

```
entity PolyEEA is
  generic (N : integer:= 32);
  Port (s: in STD_LOGIC_VECTOR (1 downto 0);
        clock, reset, Start : in STD_LOGIC;
        an : out STD_LOGIC_VECTOR (3 downto 0);
        sag : out STD_LOGIC_VECTOR (6 downto 0);
        Done : out STD_LOGIC;);
end PolyEEA;
architecture Structural of PolyEEA is
Signal ax, fx, a_inv: STD_LOGIC_VECTOR(n-1 downto 0);
begin -- SAME AS EEAPoly.vhd Except:
MUX_In: entity work.Mux4x1 Generic Map (32) Port Map
(s, ax, fx);
DISP_7: entity work.display_controller port map(clock,
reset,a_inv(15 downto 12), a_inv(11 downto 8), a_inv(7
downto 4), a_inv(3 downto 0),an, sseg);
end Structural;
```

Output Controller VHDL

```
entity display_controller is -- Craig Kief, Engineering Faculty
port( clk, reset: in std_logic; -- at University of New Mexico
     hex3, hex2, hex1, hex0: in std_logic_vector(3 downto 0);
     an: out std_logic_vector(3 downto 0);
     sseg: out std_logic_vector(6 downto 0));
end display_controller;
architecture arch of display_ controller is
signal q_reg, q_next: unsigned(1 downto 0);
signal sel: std_logic_vector(1 downto 0);
signal hex: std_logic_vector(3 downto 0); begin
 process(clk, reset) begin
   if reset='1' then q_reg <= (others=>'0');
   elsif (rising_edge(clk)) then  q_reg <= q_next; end if;
 end process;
 q_next <= q_reg + 1; sel <= std_logic_vector(q_reg);
 process(sel, hex0, hex1, hex2, hex3) begin
  case sel is
      when "00" => an <= "1110"; hex <= hex0;
      when "01" => an <= "1101"; hex <= hex1;
      when "10" => an <= "1011"; hex <= hex2;
      when others => an <= "0111"; hex <= hex3;
end case;
end process;
with hex select  sseg(6 downto 0) <=
"1000000" when "0000", --0  "1111001" when "0001", -- 1
"0100100" when "0010", --2  "0110000" when "0011", -- 3
"0011001" when "0100", --4  "0010010" when "0101", -- 5
"0000010" when "0110", --6  "1111000" when "0111", -- 7
"0000000" when "1000", --8  "0010000" when "1001", -- 9
"0001000" when "1010", --A, "0000011" when "1011", -- B
"1000110" when "1100", --C  "0100001" when "1101", -- D
"0000110" when "1110", --E, "0001110" when others;  -- F
end arch;
```

Conclusion

The “Polynomial Inverter” over binary field based on the extended Euclidean algorithm has beed designed using VHDL, simulated using ISE simulator, implemented on Xilinx Spartan3E FPGA, and demonstrated using NEXYS2. Functionality of the design, which has been satisfied in ISE simulation and hardware implementation, was main focus in the project was the target methodology. Different design strategies have been utilized within the design, e.g., hierarchy, abstraction, modularity, and locality.

The project was build in three levels, the binary division level, the GCD computation level, and the EEA, in which the GCD is computed at the same time the inverse of the input polynomial is generated.

Binary Polynomial Divider (BPD) was the bottleneck of the inversion circuit design and the challenge was to produce the remainder and the quotient in a minimum number of cycles. Computing the quotient was the most challenging part of the BDA design as the idea was to utilize degree difference of the polynomials in order to obtain the right number of required shifts each cycle. The GCD circuit was just two multiplexers and a register connected to the BDA module to update the divisor and dividend values. The EEA module was simply the GCD module with the inversion extension, which uses a polynomial multiplier and adder to compute the inverse of the input polynomial synchronously with GCD calculation.

The promising results obtained in this project has raised the enthusiasm towards exploring more regarding field arithmetic in general, and continuing the research that has been established during this journey. For the future work, the optimization of this design in terms of the utilization of the best alternatives and the FPGA primitives, might be a reasonable starting points. Also, by studying the recent proposed architectures in the literature and comparing the results, there would be high potential for designing an EEA inverter, which provides reasonable delay-area product.

Appendix

Numerator Polynomial			0	0	1	0	1	0	1	0	0	1	1	0	0	1	1	0	0
			0	5				4				C				C			
Denominator polynomial			0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1
			0	0				2				0				B			
										x^									
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ReOrdered Numerator			0	0	1	1	0	0	1	1	0	0	1	0	1	0	1	0	0
ReOrdered Denominator			1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
15		2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
10		7	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
Division	q	d																	
Numerator			0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	0	1
	1	5	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	1
			0	0	0	0	1	1	0	1	0	1	1	0	1	0	1	0	0
	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
			0	0	0	0	1	1	0	1	0	1	1	0	1	0	1	0	
	1	3	0	0	0	0	0	1	1	0	1	0	0	0	0	0	1		
			0	0	0	0	1	0	1	1	1	1	1	0	1	0	0		
	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
			0	0	0	0	1	0	1	1	1	1	1	0	1	0			
	1	1	0	0	0	1	1	0	1	0	0	0	0	0	1				
			0	0	0	1	0	0	0	1	1	1	1	0	0				
	0	0	0	0	0	0	0	0	0	0	0	0	0						
			0	0	0	1	0	0	0	1	1	1	1	0					
	1	0	0	1	1	0	1	0	0	0	0	0	1						
			0	1	1	1	1	0	0	1	1	1	0						
	1	0	1	1	0	1	0	0	0	0	0	1							
			1	0	1	0	1	0	0	1	1	0							
	1	0	1	0	1	0	0	0	0	0	1								
			0	0	0	0	1	0	0	1	0								
	1	0	0	1	0	0	0	0	0	1									
			0	1	0	0	1	0	0	0									
	0	0	0	0	0	0	0	0	0										
			0	1	0	0	1	0	0										
	0	0	0	0	0	0	0	0											
			0	1	0	0	1	0											
	1	0	0	0	0	0	1												
			0	1	0	0	0												
	0	0	0	0	0	0													
			0	1	0	0													
	0	0	0	0	0														
			0	1	0														
	1	0	0	1															
			0	0															
	0	0	0																
			0																

remainder (r)			0	0	0	0	0	0	1	1	1	1	0	0	0	1	0	0	0
Quotient (q)			1	0	1	0	1	0	1	1	1	1	0	0	1	0	0	1	0

Fig.19 BPD in Excel Spreadsheet as a start reference

```c
int main() { // Polynomial Binary Division
    unsigned int a, f, a_adj, q=1, r=0; // first q should be 1
    int d=1;
    printf("a = "); scanf ("%x",&a);
    printf("f = "); scanf ("%x",&f);
    while (d>0){
        d= clz(a)- clz(f);
        if(d<0)
            a_adj = 0;
        else
            a_adj = a << d;
        r= f ^ a_adj;
        if((clz(r)-clz(f))>=1 & clz(r)<=clz(a)){
            q= q<< (clz(r)-clz(f));
            q=q+1;}
        else if((clz(r)-clz(f))>=1 & clz(r)>clz(a))
            q= q<< (clz(a)-clz(f));
        f=r;}
    printf("r = %x\n",r); // Formating the output r and q
    for (int n=0; n<32; n++){
        printf("%x",!!((r << n)& 0x80000000));
        if (n==3 | n==7 | n==11 | n==15 | n==19 | n==23 | n==27)
            printf(" ");} /* insert a space between nybbles */
    printf("\nq = %x\n",q);
    for (int i=0; i<32; i++){
        printf("%d",!!((q << i)& 0x80000000));
    if (i==3 | i==7 | i==11 | i==15 | i==19 | i==23 | i==27)
        printf(" ");} /* insert a space between nybbles */
    return 0;}
int clz(unsigned int x) // parallel computation of CLZ
{  static const unsigned int bval[] =
    {0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4};
    unsigned int r = 0;
    if (x & 0xFFFF0000) { r += 16/1; x >>= 16/1; } //16-1 works
    if (x & 0x0000FF00) { r += 16/2; x >>= 16/2; }
    if (x & 0x000000F0) { r += 16/4; x >>= 16/4; }
    return 32-(r + bval[(x & 0x0000F000)]+ bval[(x & 0x00000F00)]+ bval[(x & 0x000000F0)]+ bval[(x & 0x0000000F)]);}
```

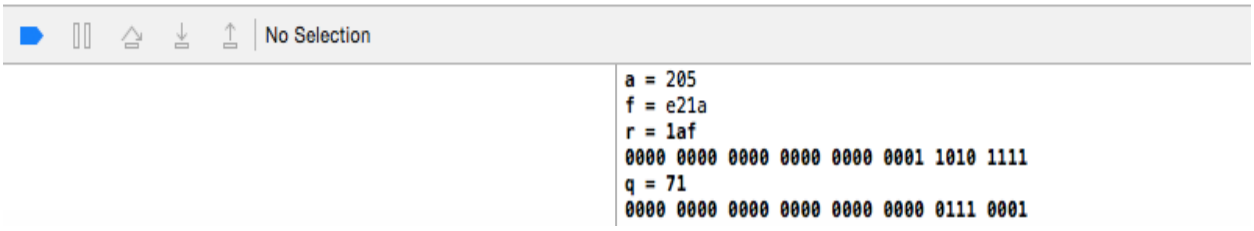


Fig.20 BPD in C as a programming model

Bibliography

1. Hankerson, D. *et al.* Guide to elliptic curve cryptography. New York, NY [u.a.]: Springer, 2004
2. Hazmi, I. *et al.* “Review of Elliptic Curve Processor Architectures”, PACRIM, University of Victoria, 2015.
3. Hazmi, I. "GCD FPGA-Based Design." ELEC569A Project Report, University of Victoria, 2014.
4. Paar, C., & Pelzl, J. Understanding cryptography: A textbook for students and practitioners. (Understanding Cryptography.) Berlin: Springer, 2010
5. Rodríguez-Henríquez, Francisco, et al. “Cryptographic algorithms on reconfigurable hardware”. Springer Science & Business Media, 2007.
6. Koç, C. (2009). Cryptographic Engineering . Boston, MA: Springer US.
7. Beard, J. “Binary Polynomial Division,” Lecture notes: EE521: Analog and Digital Communications, Temple University, 2006, [Online], Available at http://jameskbeard.com/Temple/Data/Binary_Polynomial_Division.pdf.
8. Gebali, F. (2015). ELEC543 ‘Digital VLSI Design. -[Lecture Notes]
9. Kang, Min-Sup, and Byong-Chan Jeon. "Design of Iterative Divider in GF (2^{163}) Based on Improved Binary Extended GCD Algorithm." The KIPS Transactions: PartC 17.2, 2010.
10. Kobayashi, Katsuki, and Naofumi Takagi. "Fast Hardware Algorithm for Division in Based on the Extended Euclid's Algorithm With Parallelization of Modular Reductions." Circuits and Systems II: Express Briefs, IEEE Transactions on 56.8 (2009): 644-648.
11. Zadeh, Abdulah Abdulah. "Division and Inversion Over Finite Fields." CRYPTOGRAPHY AND SECURITY IN COMPUTING (2012): 117.
12. Bajard, Jean Claude, Nicolas Meloni, and Thomas Plantard. "Study of modular inversion in RNS." Proc. SPIE. Vol. 5910. 2005.
13. C.P, N. and M. Ravi Kumar, K. “Efficient Comparator based Sum of Absolute Differences Architecture for Digital Image Processing Applications.” International Journal of Computer Applications, 96(4), pp. 17-24, 2014.
14. R. Afreen and S. Mehrotra, 'A Review on Elliptic Curve Cryptography for Embedded Systems', International Journal of Computer Science and Information Technology , vol. 3, pp. 84103, 2011.
15. M. Sandoval, 'A reconfigurable and interoperable hardware architecture for ECC', Ph.D, National Institute for Astrophysics, Optics and Electronics (INAOE), Puebla, 2008.
16. C. Rebeiro, 'Architecture Explorations for Elliptic Curve Cryptography on FPGAs', Master of Science, Indian Institute of Technology, Madras, 2009.

The project

The objective of the project was to design a “Polynomial Inverter” over binary field based on the extended Euclidean algorithm (EEA). The main focus in the project was on the functionality of the design, which has been satisfied in ISE simulation and implemented successfully in Xilinx Spartan 3E FPGA.

The project was build in three levels, the binary division level, the GCD computation level, and the EEA, in which the GCD is computed at the same time the inverse of the input polynomial is generated. The Binary Polynomial Divider (BPD) was the bottleneck of the inversion circuit design and the challenge was to produce the remainder and the quotient in a minimum number of cycles. Computing the quotient was the most challenging part of the BDA design as the idea was to utilize degree difference of the polynomials in order to obtain the right number of required shifts each cycle. The GCD circuit was just two multiplexers and a register connected to the BDA module to update the divisor and dividend values. The EEA module was simply the GCD module with the inversion extension, which uses a polynomial multiplier and adder to compute the inverse of the input polynomial synchronously with GCD calculation.