Small FPGA-Based Multiplication-Inversion Unit for Normal Basis over $\mathrm{GF}(2^m)$

Métairie Jérémy, Tisserand Arnaud and Casseau Emmanuel

CAIRN - IRISA

July 9th, 2015

ISVLSI 2015

PAVOIS ANR 12 BS02 002 01

Summary

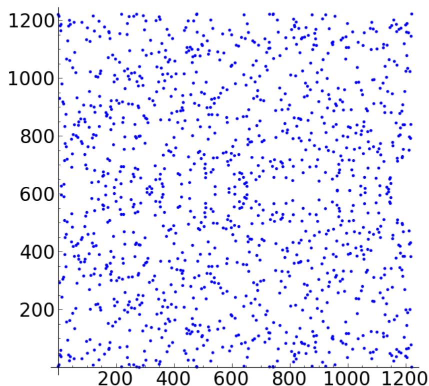
1. Elliptic Curves Background and State-of-the-Art
2. Proposed Solution
3. Architecture and Figures

Elliptic Curves

Elliptic Curves

$\mathbb{E} = \{(x, y) \in \mathrm{GF}(p)^2 \text{ such that } y^3 = x^3 + a \cdot x + b\}$

Equation: $y^2 = x^3 + 3x + 5$ in $\mathrm{GF}(1223)$



Point Operations

- $[k]P = \underbrace{P + P + \ldots + P}_{\text{k times}}$
- **ADD**: $R = P + Q$ with $P \neq Q$ and $R, P, Q \in \mathbb{E}$
- **DBL**: $R = P + P$ with $R, P \in \mathbb{E}$

Discrete Logarithm Problem

Knowing P and Q it is very hard to find $k \in \mathbb{Z}$ such $Q = [k]P$

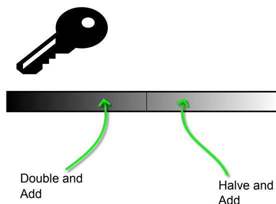
Double-And-Add vs. Halve-and-Add Algorithms

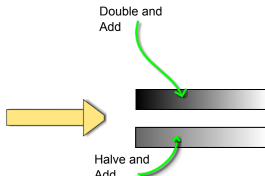
Inputs: $P \in \mathbb{E}$ and $k = (k_0, k_1, \ldots, k_{m-1}) \in \mathbb{N}$
Output: $Q = [k]P$

1: $Q \leftarrow \mathcal{O}$
2: **for** i **from** 0 to $m-1$ **do**
3:     **if** $k_i = 1$ **then**
4:         $Q \leftarrow Q + P$
5:     **end if**
6:     $P \leftarrow 2 \cdot P$
7: **end for**
8: **return** Q

Inputs: $P \in \mathbb{E}$ and $k = (\underline{k}_0, \underline{k}_1, \ldots, \underline{k}_{m-1}) \in \mathbb{N}$
Output: $Q = [k]P$

1: $Q \leftarrow \mathcal{O}$
2: **for** i **from** 0 to $m-1$ **do**
3:     **if** $\underline{k}_i = 1$ **then**
4:         $Q \leftarrow Q + P$
5:     **end if**
6:     $P \leftarrow P/2$
7: **end for**
8: **return** Q





- Faster Computation
- Protection against (some) Side Channel Attacks

Elliptic Curves over GF(2^m)

Definition

$\mathbb{E} = \{(x, y) \in \mathrm{GF}(2^m)^2$ such that $y^2 + x \cdot y = x^3 + a_2 \cdot x^2 + a_6\}$

Let $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ be two points in $\mathbb{E}$.
One can compute $R = P + Q$ as follows (affine coordinates):
$\lambda = \frac{x_p + x_q}{y_p + y_q}$ then
$x_r = \lambda^2 + \lambda + x_p + x_q + a$ and $y_r = \lambda \cdot (x_p + x_r) + x_r + p_y$

- Note that $\frac{1}{y_p + y_q}$ is costly to compute ($\approx$ 10 multiplications)

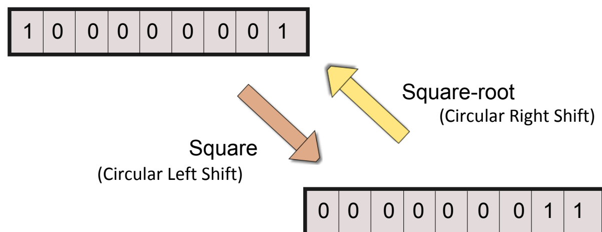
- Recommended m $\in \{163, 233, 283, 409, 571\}$

Normal Basis (NB)

Every element $A \in \mathbf{GF}(2^m)$ can then be written as follows:

$$A = \sum_{i=0}^{m-1} a_i \beta^{2^i} \text{ with } a_i \in \{0, 1\}$$

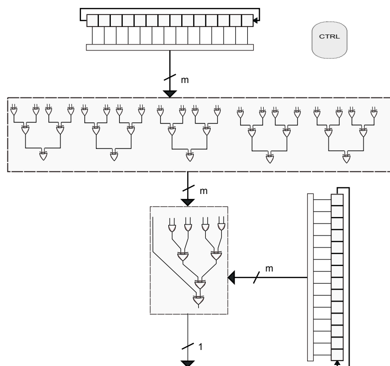
Note that element A can be stored as a vector $\underline{a} = [a_0, a_1, \ldots, a_{m-1}]$.



$\Rightarrow$ Easy squares but more complicated multiplications.

Massey-Omura Multiplication in Binary Finite Field. [4]

Inputs: $A \in \mathrm{GF}(2^m)$ (NB), $B \in \mathrm{GF}(2^m)$ (NB)
Output: $P = A \cdot B$ (NB)
1: $P \leftarrow 0$; $i \leftarrow 0$
2: **while** $i < m$ **do**
3: $P[0] \leftarrow A \cdot M_0 \cdot (B)^T$
4: $i \leftarrow i + 1$
5: $A \leftarrow \mathrm{LeftShift}(A, 1)$
6: $B \leftarrow \mathrm{LeftShift}(B, 1)$
7: $P \leftarrow \mathrm{LeftShift}(P, 1)$
8: **end while**
9: **return** P



Fermat's Little Theorem

Fermat's Little Theorem

For any $\alpha \in \mathrm{GF}(2^m)^*$

$$\alpha^{-1} = \alpha^{2^m-2}$$

If one wants to compute $\alpha^{2^{10}-2} = \alpha^{(1\ 111\ 111\ 110)_2}$, one can perform the following operations :

Itoh-Tsujii Sequence [3]

$P_0 = \alpha^{(1)_2}$
$P_1 = P_0^2 \cdot P_0 = \alpha^{(10)_2} \cdot \alpha^{(1)_2} = \alpha^{(11)_2}$
$P_2 = P_1^{2^2} \cdot P_1 = \alpha^{(1100)_2} \cdot \alpha^{(11)_2} = \alpha^{(1111)_2}$
$P_3 = P_2^{2^4} \cdot P_2 = \alpha^{(11110000)_2} \cdot \alpha^{(1111)_2} = \alpha^{(11111111)_2}$
$P_4 = P_3^2 \cdot P_0 = \alpha^{(111111110)_2} \cdot \alpha^{(1)_2} = \alpha^{(111111111)_2}$
$P_5 = P_4^2 = \alpha^{2^{10}-2} = \alpha^{(1\ 111\ 111\ 110)_2}$

Here, only 4 multiplications are necessary to perform the whole exponentiation (8 for square-and-multiply algorithm).

Using Symmetries in the Massey-Omura Algorithm [4]

For the special multiplication case $B = A^{2^j}$, a symmetry appears. Let us consider an example where $A = [a_0, a_1, a_2]$ and $B = [a_2, a_0, a_1]$.
The different steps of the regular Massey-Omura algorithm:

- Step 1 : $p_0 = \begin{bmatrix} a_0 & a_1 & a_2 \end{bmatrix} \cdot M_0 \cdot \begin{bmatrix} a_2 \\ a_0 \\ a_1 \end{bmatrix}$
- Step 2 : $p_1 = \begin{bmatrix} a_1 & a_2 & a_0 \end{bmatrix} \cdot M_0 \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$
- Step 3 : $p_2 = \begin{bmatrix} a_2 & a_0 & a_1 \end{bmatrix} \cdot M_0 \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_0 \end{bmatrix}$

Using Symmetries in the Massey-Omura Algorithm [4]

For the special multiplication case $B = A^{2^j}$, a symmetry appears. Let us consider an example where $A = [a_0, a_1, a_2]$ and $B = [a_2, a_0, a_1]$.
The different steps of the regular Massey-Omura algorithm:

- Step 1 : $p_0 = \begin{bmatrix} a_0 & a_1 & a_2 \end{bmatrix} \cdot M_0 \cdot \begin{bmatrix} a_2 \\ a_0 \\ a_1 \end{bmatrix}$
- Step 2 : $p_1 = \begin{bmatrix} a_1 & a_2 & a_0 \end{bmatrix} \cdot M_0 \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$
- Step 3 : $p_2 = \begin{bmatrix} a_2 & a_0 & a_1 \end{bmatrix} \cdot M_0 \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_0 \end{bmatrix}$

Proposed Multiplication Algorithm when $\gcd(j, m) = 1$

Inputs: $A \in \mathrm{GF}(2^m)$ (NB) , $B \in \mathrm{GF}(2^m)$ such that $B = A^{2^j}$ (NB) and $j \in \mathbb{N}$
Output: $P = A \cdot B$ in normal basis

1: $C \leftarrow \text{LeftShift}(B, m - j)$
2: $P \leftarrow 0$
3: $i \leftarrow 0$
4: **while** $i < \lceil m/2 \rceil$ **do**
5: $\quad g \leftarrow M_0 \cdot (A)^T$
6: $\quad P[j] \leftarrow g \cdot (C)^T$; $P[0] \leftarrow g \cdot (B)^T$
7: $\quad A \leftarrow \mathrm{LeftShift}(A, 2j)$; $B \leftarrow \mathrm{LeftShift}(B, 2j)$; $C \leftarrow \mathrm{LeftShift}(C, 2j)$
$\quad P \leftarrow \mathrm{LeftShift}(P, 2j)$
8: $\quad i \leftarrow i + 1$
9: **end while**
10: **return** P

Different j values may be used for the exponentiation process.
In hardware, variable shifters are area costly for large operands
$\Rightarrow$ We need to remove those $2j$ shifts.

Proposed Multiplication Algorithm with θ Constant

Inputs: $A \in \mathrm{GF}(2^m)$ (NB) , $B \in \mathrm{GF}(2^m)$ such that $B = A^{2^j}$ (NB) and $j \in \mathbb{N}$, $\theta \in \mathbb{N}$
Output: $P = A \cdot B$ in normal basis

1: $C \leftarrow \text{LeftShift}(B, m - j)$
2: $P \leftarrow 0$
3: $i \leftarrow 0$
4: **while** $i < \lceil N(j, \theta) \rceil$ **do**
5: $\quad g \leftarrow M_0 \cdot (A)^T$
6: $\quad P[j] \leftarrow Tmp \cdot (C)^T$; $P[0] \leftarrow Tmp \cdot (B)^T$
7: $\quad A \leftarrow LeftShift(A, \theta)$; $B \leftarrow LeftShift(B, \theta)$; $C \leftarrow LeftShift(C, \theta)$
$\quad P \leftarrow LeftShift(P, \theta)$
8: $\quad i \leftarrow i + 1$
9: **end while**
10: **return** P

$N(j, \theta)$ is the number of iterations to get all the bits of P.
Note that $N(j, \theta) \geq \lceil m/2 \rceil$.

A Wise Choice of the Constant Shift θ

The goal is now to find θ which minimizes $\mathcal{D} = \sum_{i \in \mathcal{I}} N(i, \theta)$ where $\mathcal{I}$ is the set of all the j implied in the computations of the $A^{2^j} \cdot A$ patterns used in the exponentiation (inversion).

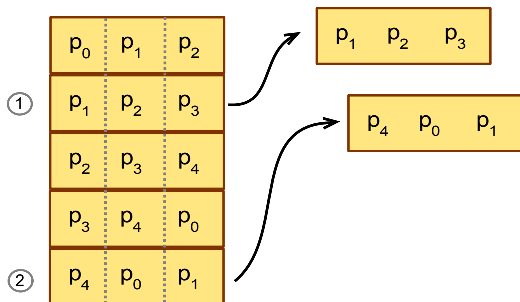
m	θ	$\mathcal{D}$
163	72	732
233	36	1046
283	28	1431
409	35	2263
571	171	3221

Definition

Permuted Normal Basis (PNB) representation where element $A = [a_0, a_1, a_2, \ldots, a_{m-1}]$ is represented by $A' = [\ a_0\ ,\ a_\theta\ ,\ a_{2\theta \bmod m}\ , \ldots ,\ a_{(m-1)\theta \bmod m}\ ]$.

Shifting Through BRAMs

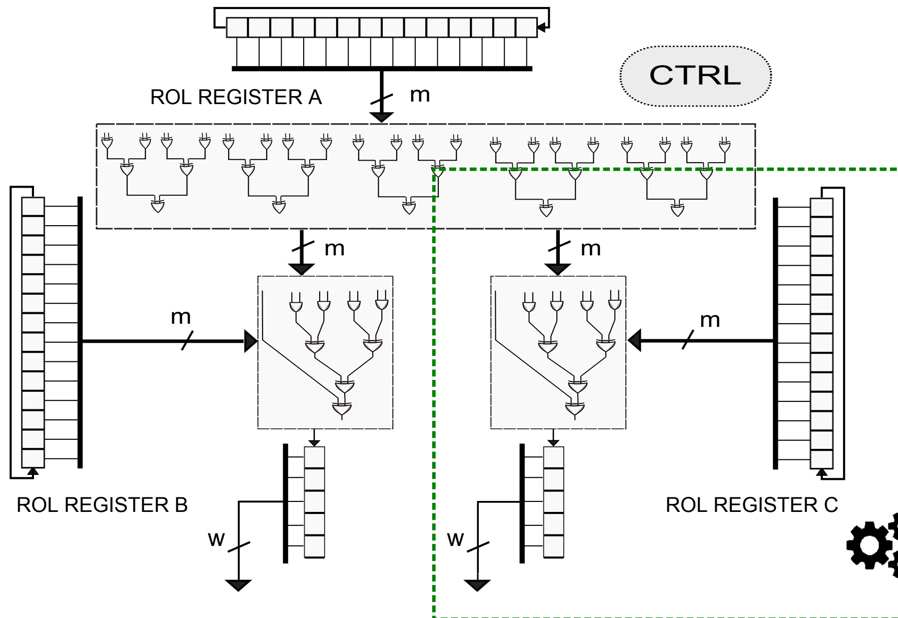
We duplicate w times the bits of $P = A \cdot B = [p_0, p_1, \ldots, p_{m-1}]$ in a BRAM using the following patterns:



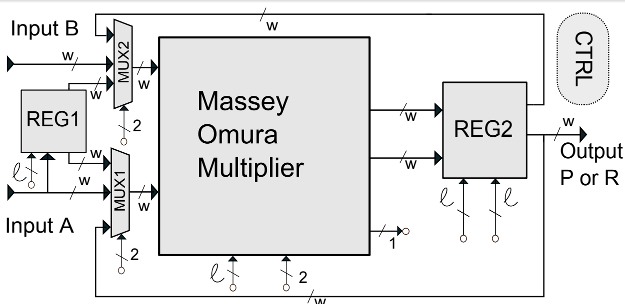
BRAM
$p_0, p_1, \ldots, p_{w-1}$
$p_1, p_2, \ldots, p_w$
$\vdots$
$p_{m-1}, p_3, \ldots, p_{m-w-2}$

BRAMs in recent FPGAs are large enough to support the $m \cdot w$ bits (18 Kb on a low-cost Spartan-6 and Virtex 4).

Architecture: Multiplier



Architecture: Multiplication-Inversion Unit (MIU)



Implementation of the Multiplication-Inversion Unit on Virtex-4 LX100 with $w = 32$ and $\ell = 10$.

m	Algo.	Area Slices (LUT, FF)	Freq. MHz	Inv. Time μs
571	MO1 [5]*	3378 (5615, 2016)	125	64.4
	RM2 [6]*	4976 (9445, 2090)	107	38.7
	our PNB	4308 (5928, 2650)	125	47.7
571	Hybrid ($d = 13$) [1]	#LUTs = 85268	74	4.98
	Parallel ($d = 13$) [2]	#LUTs = 56657	82	5.00

Implementation Results

Hardware implementation on on Virtex-4 LX100 and time estimation of a scalar multiplication ($m = 571$) only using the Halve-and-Add algorithm.

	Algorithm	halving ms	area #LUTs	ATP $\cdot 10^{-3}$
NAF	MO1 [5]*	17.3	5742	95
	RM2 [6]*	13.0	9572	122
	our PNB	**14.3**	6055	**82**
	Parallel IT (d=13) [2]	1.59	56784	90
	Hybrid IT (d=13) [1]	1.60	85395	136
3-NAF	MO1 [5]*	14.6	similar	79
	RM2 [6]*	8.95		76
	our PNB	**11.3**		**65**
	Parallel IT (d=13) [2]	1.34		74
	Hybrid IT (d=13) [1]	1.40		119

ATP: area-time product

Conclusion

We proposed a new Multiplication-Inversion Unit that

- Uses a new normal basis representation (PNB)
  $\Rightarrow$ replacement of large shifters by BRAMs
- Is $\approx$ 20% faster than classical MO approach for halving-based scalar multiplication

We still have to :

- Have a full implementation of a crypto-processor
- Study security aspects of our design

Thank you for your attention ! ☺

References

[1] R. Azarderakhsh, K. Jarvinen, and V. Dimitrov.
Fast inversion in GF(2^m) with normal basis using hybrid-double multipliers.
IEEE Trans. Comp., 63(4):1041–1047, April 2014.

[2] J. Hu, W. Guo J. Wei, and R.C.C. Cheung.
Fast and generic inversion architectures over GF(2^m) using modified Itoh-Tsujii algorithms.
IEEE Transactions on Circuits and Systems II: Express Briefs, 2015.
Accepted paper.

[3] Itoh and Tsujii.
A fast algorithm for computing multiplicative inverses in gf(2m) using normal bases.
Information and Computation, 1988.

[4] Omura Massey.
Computational method and apparatus for finite field arithmetic.
U.S. Patent Application, 1981.

[5] J. K. Omura and J. L. Massey.
Computational method and apparatus for finite field arithmetic.
US Patent US4587627 A, May 1986.

[6] A. Reyhani-Masoleh.
Efficient algorithms and architectures for field multiplication using Gaussian normal bases.
IEEE Trans. Comp., 55(1):34–47, 2006.

In the "modified" version of the algorithm, we proceed as:

- Step 1 :

$$g = M_0 \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

$$p_0 = \begin{bmatrix} a_2 & a_0 & a_1 \end{bmatrix} \cdot g^T \quad | \quad p_1 = \begin{bmatrix} a_1 & a_2 & a_0 \end{bmatrix} \cdot g^T$$

- Step 2 :

$$g = M_0 \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_0 \end{bmatrix}$$

$$p_2 = \begin{bmatrix} a_2 & a_0 & a_1 \end{bmatrix} \cdot g^T$$