UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATISTICA

Lucas Boppre Niehues

FAST MODULAR REDUCTION AND SQUARING IN $GF(2^m)$

Florianópolis

2017

Lucas Boppre Niehues

FAST MODULAR REDUCTION AND SQUARING IN $GF(2^m)$

Dissertação submetida ao curso de Pós-Graduação em Ciência da Computação para o Exame de Qualificação de Mestrado.
Orientador: Prof. Ricardo Felipe Custódio
Coorientador: Prof. Daniel Panario

Florianópolis

2017

Lucas Boppre Niehues

FAST MODULAR REDUCTION AND SQUARING IN $GF(2^m)$

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pelo curso de Pós-Graduação em Ciência da Computação.

Florianópolis, 01 de Janeiro 2017.

Prof. Chefe, Carina Dorneles
Coordenador do Curso

Prof. Ricardo Felipe Custódio
Orientador

Prof. Daniel Panario
Coorientador

Banca Examinadora:

Primeiro membro da banca
Presidente

Segundo membro da banca

RESUMO

Resumo em portugues
Palavras-chave: Finite field, number theory, polynomial, squaring, cryptography.

ABSTRACT

We present an efficient bit-parallel algorithm for squaring in $GF(2^m)$ using polynomial basis. This algorithm achieves competitive efficiency while being aimed at any choice of low-weight irreducible polynomial. For a large class of irreducible polynomials it is more efficient than the previously best general squarer. In contrast, other efficient squarers often require a change of basis or are suitable for only a small number of irreducible polynomials. Additionally, we present a simple algorithm for modular reduction with equivalent cost to the state of the art for general irreducible polynomials. This fast reduction is used in our squaring method.

LIST OF FIGURES

LIST OF TABLES

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE SÍMBOLOS

CONTENTS

1 INTRODUCTION

2 MATHEMATICAL BACKGROUND

3 REDUCTION AND SQUARING IN $GF(2^m)$

4 GENERALIZED REDUCTION AND SQUARING IN $GF(p^m)$

5 WORD PROCESSING

During our research we also studied $GF(2^m)$ arithmetic for CPUs, where the coefficient bits are grouped into words. Most of the results were about polynomial reduction modulo a trinomial, a common operation in $\mathbb{F}_2[x]/(x^m + x^a + 1)$. This type of field is interesting because it maximizes the number of zero coefficients in the the irreducible polynomial, enabling numerous simplifications. In turn, the reduction operation is used in multiplications and exponentiations, making speedups desirable.

Performing this operation in a CPU has a few differences from implementing a hardware circuit. For example, circuits often avoid irreducible polynomials with $a > m/2$. This is caused by the number of reduction steps in the standard technique, whcih represents interdependency of the calculations, and thus increases the maximum delay in a circuit. The maximum number k of reduction steps for a trinomial $x^m + x^a + 1$ in terms of the exponent a is given by Sunar and Koç (SUNAR; KOC, 1999)

$$k = \left\lfloor \frac{m-2}{m-a} \right\rfloor + 1. \tag{5.1}$$

However, the number of reduction steps does not affect software implementations. To illustrate this, we take two algorithms that perform modular reduction for the NIST polynomials $x^{233} + x^{74} + 1$ and $x^{409} + x^{87} + 1$ (HANKERSON; MENEZES; VANSTONE, 2006, p. 55). Then we manually adapt each of these algorithms to perform a reduction modulo its reciprocal $x^m + x^{a-m} + 1$. Notice the only change required is in the indexing and bitwise shifts. The total number of operations remains the same. Algorithms meant for circuit implementations, on the other hand, would have different delay characteristics when

To illustrate this, consider the following four CPU-targeted reduction algorithms: $x^{233} + x^{74} + 1$ and $x^{409} + x^{87} + 1$ (HANKERSON; MENEZES; VANSTONE, 2006, p. 55), and our adaptations. The second algorithm is our adaption to perform a polynomial reduction modulus $x^{233} + x^{159} + 1$, its reciprocal. Notice they perform exactly the same number of operations, but would have drastically different performance if implemented in hardware.

Algorithm 1 Hankerson's algorithm for reduction modulus $x^{233}+x^{74}+1$, a standardized NIST polynomial.

Input: $C[2m-2,0]$
Output: $C[m-1,0]$
1: **for** $i \leftarrow 15$ **downto** 8 **do**
2: $T \leftarrow C[i]$
3: $C[i-8] \leftarrow C[i-8] \oplus T << 23$
4: $C[i-7] \leftarrow C[i-7] \oplus T >> 9$
5: $C[i-5] \leftarrow C[i-5] \oplus T << 1$
6: $C[i-4] \leftarrow C[i-4] \oplus T >> 31$
7: **end for**
8: $T \leftarrow C[7] >> 9$
9: $C[0] \leftarrow C[0] \oplus T$
10: $C[2] \leftarrow C[2] \oplus T << 10$
11: $C[3] \leftarrow C[3] \oplus T >> 22$
12: $C[7] \leftarrow C[7]$&`0x1FF`
13: **return** C

Algorithm 2 Algorithm for reduction modulus $x^{233} + x^{159} + 1$, $(233, 74)$'s recriprocal.

Input: $C[2m-2,0]$
Output: $C[m-1,0]$
1: **for** $i \leftarrow 14$ **downto** 8 **do**
2: $T \leftarrow C[i]$
3: $C[i-8] \leftarrow C[i-8] \oplus T << 23$
4: $C[i-7] \leftarrow C[i-7] \oplus T >> 9$
5: $C[i-3] \leftarrow C[i-3] \oplus T << 22$
6: $C[i-2] \leftarrow C[i-2] \oplus T >> 10$
7: **end for**
8: $T \leftarrow C[7] >> 9$
9: $C[0] \leftarrow C[0] \oplus T$
10: $C[2] \leftarrow C[2] \oplus T << 31$
11: $C[3] \leftarrow C[3] \oplus T >> 1$
12: $C[7] \leftarrow C[7]$&`0x1FF`
13: **return** C

Algorithm 3 Hankerson's algorithm for reduction modulus $x^{409}+x^{87}+1$, a standardized NIST polynomial.

Input: $C[2m-2, 0]$
Output: $C[m-1, 0]$
1: **for** $i \leftarrow 25$ **downto** 13 **do**
2: $T \leftarrow C[i]$
3: $C[i-13] \leftarrow C[i-13] \oplus T << 7$
4: $C[i-12] \leftarrow C[i-12] \oplus T >> 25$
5: $C[i-11] \leftarrow C[i-11] \oplus T << 30$
6: $C[i-10] \leftarrow C[i-10] \oplus T >> 2$
7: **end for**
8: $T \leftarrow C[12] >> 25$
9: $C[0] \leftarrow C[0] \oplus T$
10: $C[2] \leftarrow C[2] \oplus T << 23$
11: $C[12] \leftarrow C[12]$&0x1FFFFFF
12: **return** C

Algorithm 4 Algorithm for reduction modulus $x^{409} + x^{322} + 1$, $(409, 87)$'s reciprocal.

Input: $C[2m-2, 0]$
Output: $C[m-1, 0]$
1: **for** $i \leftarrow 25$ **downto** 13 **do**
2: $T \leftarrow C[i]$
3: $C[i-13] \leftarrow C[i-13] \oplus T << 7$
4: $C[i-12] \leftarrow C[i-12] \oplus T >> 25$
5: $C[i-3] \leftarrow C[i-3] \oplus T << 9$
6: $C[i-2] \leftarrow C[i-2] \oplus T >> 23$
7: **end for**
8: $T \leftarrow C[12] >> 25$
9: $C[0] \leftarrow C[0] \oplus T$
10: $C[10] \leftarrow C[10] \oplus T << 2$
11: $C[12] \leftarrow C[12]$&0x1FFFFFF
12: **return** C

5.1 CHOICE OF A

The choice of the second coefficient a directly affects the performance characteristics of reduction algorithms. Several factors are at play:

- The number of reduction steps depends on the proximity of a and m. The number of steps increases with a, in discrete steps.
- The number of reduction steps affects the depth of the circuit when implemented in hardware, though usually in logarithm fashion.
- If $m - a < W$ software implementations will be less efficient. This is because there will be XOR operations inside words, and the operations can not be performed with full word-sized XORs.
- The alignment of a and $m - a$ with relation to W define if word-sized XORs will require bits of two words in the source or in the destination. Thus alignment can halve the number of operations involved.

5.2 EXISTING ALGORITHMS

Masoleh (??, p. 953) presents a recursive multiplier algorithm with $N_{\oplus} = m^2 - 1$ for $x^m + x^a + 1$ and $N_{\oplus} = m^2 - \frac{m}{2}$ if $a = \frac{m}{2}$. If we consider that the multiplication takes $(m-1)^2$ XORs, then, the number of XORs for the reduction is $2m - 2$ and $\frac{3}{2}m - 1$, respectively.

[NIST, Scott, etc. just dry descriptions]

5.3 REDUCTION EQUATION

[This is an attempt to formulate the reduction operation as an equation. This is useful for proving correctness of algorithms by showing equivalence to the equations. I'm not 100 they are correct. (I think they are not truncating the value to the modulus degree, but that should be easy to fix)]

General polynomial reduction equation with coefficients in $GF(2)$, with reduction steps in evidence:

$$\sum_{i=0}^{2m-2} c_i x^i \mod \sum_{i=0}^{m} k_i x^i = \tag{5.2}$$

$$\sum_{i=0}^{m-1} c_i x^i + \sum_{i=0}^{\lfloor \frac{m-2}{m-a} \rfloor} \sum_{l=0}^{m} \sum_{j=0}^{m-2-ia} k_l c_{j+m+ia} x^{j+l} \tag{5.3}$$

General trinomial reduction equation ($1 \leq a < m$), with reduction steps in evidence:

$$\sum_{i=0}^{2m-2} c_i x^i \mod x^m + x^a + 1 = \tag{5.4}$$

$$\sum_{i=0}^{m-1} c_i x^i + \sum_{i=0}^{\lfloor \frac{m-2}{m-a} \rfloor} \left(\sum_{j=0}^{m-2-ia} c_{j+m+ia} x^j + \sum_{j=0}^{m-2-ia} c_{j+m+ia} x^{j+a} \right) \tag{5.5}$$

Special case $a = 1$ (single step):

$$\sum_{i=0}^{m-1} c_i x^i + \sum_{j=0}^{m-2} c_{j+m} x^j + \sum_{j=0}^{m-2} c_{j+m} x^{j+a} \tag{5.6}$$

Special case $1 < a < \frac{m}{2}$ (two steps):

$$\sum_{i=0}^{m-1} c_i x^i + \sum_{j=0}^{m-2} c_{j+m} x^j + \sum_{j=0}^{m-2} c_{j+m} x^{j+a} + \sum_{j=0}^{m-2-a} c_{j+m+a} x^j + \sum_{j=0}^{m-2-a} c_{j+m+a} x^{j+a} \tag{5.7}$$

Special case $a = \frac{m}{2}$ (two steps with cancellation):

$$\sum_{i=0}^{m-1} c_i x^i + \sum_{j=0}^{\frac{m}{2}-2} c_{j+m} x^j + \sum_{j=0}^{m-2} c_{j+m} x^{j+\frac{m}{2}} + \sum_{j=0}^{\frac{m}{2}-2} c_{j+m+a} x^j \tag{5.8}$$

5.4 OPERATING ON WORDS

Bit manipulation operations are not efficiently supported in most programming languages. Programming languages, in general, are word oriented following a specific microprocessor architecture. For instance, numbers with 32- or 64-bit word size. The reduction algorithms themselves are strongly bit oriented. The codification of these algorithms to word oriented programming languages usually add some complexity in terms of new operations. Specific techniques that help us in this coding have been proposed in the literature(??).

Interoperability often means an irreducible polynomial must behave well in both hardware and software implementations. A straightforward approach to this problem is to develop a hardware implementation first, XOR'ing individual bits, and convert the algorithm to words. This can be thought of as parallelizing the operations, with SHIFTs and AND/OR masks for alignment. This is the approach used in this document. [more formalization? references of other similar approaches?]

The difficulty of the conversion process greatly varies depending on the access pattern of the algorithm and alignment of words. If all coefficients inside a range are XOR'ed once in a linear fashion, the XOR distance a multiple of the word size, and the start and end range lie in word boundaries, the converted algorithm gains a full speedup of up to `WORD_SIZE` times [$C_{word} = \lceil \frac{C_{bit}}{W} \rceil$].

5.5 BIT OPERATION REDUCTION ALGORITHMS

This section presents bit operation reduction algorithms for trinomials. The algorithms depend on the number of reduction steps given by Eq. 5.1. As k increases, more steps can be required to perform the reduction. The equation $x^{m+j} \equiv x^{a+j} + x^j mod(x^m + x^a + 1)$ is used to replace each coefficient at or over m with a smaller one, reducing the polynomial. The Algorithm 5 shows this reduction process.

However this algorithm introduces a number of repeated bit operations. Some of these operations could be canceled, and others could be grouped, so that they can be made once and reused when necessary.

We propose a general algorithm for all possible values of a, and one for the special case $m = 2a$. This is known in literature as Equally Spaced Polynomials. For comparison reasons we introduce instantiated algorithms for $a = 1$ and $a = m-1$, which are usually treated as special cases in the literature.

Algorithm 5 General reduction algorithm processed by bit, taken directly from equation 5 (update).

Input: a, $C[2m-2, 0]$
Output: $C[m-1, 0]$

```
1: k ← ⌊(m−2)/(m−a)⌋ + 1
2: r ← 0
3: for i ← 1 to k do
4:    r ← r + (m − a)
5:    for j ← 0 to m − 1 − r do
6:       C[j] ← C[j] ⊕ C[j + a + r]
7:    end for
8:    for j ← a to m − 1 do
9:       C[j] ← C[j] ⊕ C[j + r]
10:   end for
11: end for
12: return C
```

5.5.1 Case $(m-a) \mid a$

Situations where $m-a$ divides a result in cancelled items, coefficients that are XORed twice to the same bit. The most visible case is when $m-a=a$, the equally-spaced trinomial. In this case, the cancelled items translate directly to cancelled operations, and the reduction can be performed in less steps.

Other cases can be found by using the formula $a = \frac{\alpha}{\alpha+1}m$ with $a, \alpha \in \mathbb{Z}^+$ [I think it should be $\mathbb{N}$, Custodio disagrees]. The equally spaced case happens when $\alpha = 1$. As an example of other cases, $x^{20} + x^{15} + 1$, $x^{20} + x^{16} + 1$ and $x^{20} + x^{18} + 1$ all contain some cancellations. The number of XOR cancellations is $m - a - 1$, which explains the complexity of $\frac{3}{2}m - 3$ for the equally-spaced case. It's still an open problem if the non-equally spaced case can be optimized at all.

5.5.2 Case $a = 1$

When $a = 1$, the number of reduction steps given by the Eq. 5.1 is

$$k = \left\lfloor \frac{m-2}{m-a} \right\rfloor + 1 = \left\lfloor \frac{m-2}{m-1} \right\rfloor + 1 = 1.$$

This is the simplest case. The number of XOR operations is $N_{\oplus} = 2m - 2$.

Algorithm 6 Simple reduction algorithm for $x^m + x + 1$, $a = 1$

Input: m, $C[2m-2, 0]$
Output: $C[m-1, 0]$
1: **for** $i \leftarrow m-2$ **downto** 0 **do**
2: $\quad C[i+1] \leftarrow C[i+1] \oplus C[i+m]$
3: $\quad C[i] \leftarrow C[i] \oplus C[i+m]$
4: **end for**
5: **return** C

Algorithm 6 uses $2m - 2$ operations and has a depth of only 2. This is the best possible case for a bit-parallel circuit.

5.5.3 Case $1 < a < \frac{m}{2}$

The number of reduction steps given by the Eq. 5.1 is

$$k = \left\lfloor \frac{m-2}{m-a} \right\rfloor + 1 = 2.$$

The number of XOR operations is $N_{\oplus} = 2m - 2$.

5.5.4 Case $m = 2a$

The special case where $m = 2a$ operation algorithm is presented in Algorithm 8. This algorithm, known in literature as Equally Spaced Trinomial, is essentially the same stated by Wu(**??**, p. 753, Eq. 3). There, however, it used different vectors: one for the element to be reduced and other for the result itself. The number of reduction steps given by the Eq. 5.1 is

$$k = \left\lfloor \frac{m-2}{m-a} \right\rfloor + 1 = \left\lfloor \frac{2a-2}{2a-a} \right\rfloor + 1 = \left\lfloor \frac{2a-2}{a} \right\rfloor + 1 = 2.$$

7 is a $m = 2a$ version of algorithm 10 with cancellations. It is trivial to see that its complexity is $m + a - 1$, or, since $a = m/2$, the number of operations is exactly $\frac{3}{2}m - 1$.

Algorithm 7 Simple Reduction algorithm for $x^m + x^a + 1$, $a = \frac{m}{2}$.

Input: a, $C[0, 2m-2]$
Output: $C[m-1, 0]$
1: **for** $i \leftarrow 0$ **to** $m-2$ **do**
2: $\quad C[i+a] \leftarrow C[i+a] \oplus C[i+m]$
3: **end for**
4: **for** $i \leftarrow a-1$ **to** 0 **do**
5: $\quad C[i] \leftarrow C[i] \oplus C[i+m]$
6: **end for**
7: **return** C

5.5.4.1 Proof

Applying the equation of maximum number of reduction steps, by Sunar and Koç (**??**), to the equally spaced case we arrive at

$$k = \left\lfloor \frac{m-2}{m-\frac{m}{2}} \right\rfloor + 1 = 2 \tag{5.9}$$

Therefore the reduction equation is comprised of two steps. Since $(x^m)^n = (x^a+1)^n$, the full reduction formula for the equally spaced trinomial is:

$$\sum_{i=0}^{m-1} c_i x^i + \sum_{j=0}^{\frac{m}{2}-2} c_{j+m} x^j + \sum_{j=0}^{m-2} c_{j+m} x^{j+\frac{m}{2}} + \sum_{j=0}^{\frac{m}{2}-2} c_{j+m+a} x^j \tag{5.10}$$

Lines 1 and 2 of algorithm 7 correspond to the first reduction step for x^a (last summation). Lines 3 and 4 correspond truncated first step for x^0 (second summation). And because of the mutable nature of the data structure, lines 3 and 4 operate on bits that were already XOR'ed in the first loop. This overlap gives raises to the remaining a elements of the second reduction step for x^0 (third summation).

End of proof.

5.5.5 Case $a = m - 1$

The special case where $a = m - 1$ operation algorithm is presented in Algorithm 9. This algorithm has the same complexity as the

Algorithm 8 Reduction algorithm for $x^m + x^a + 1$, $m = 2a$.

Input: a, $C[4a-2, 0]$
Output: $C[2a-1, 0]$
1: **for** $i \leftarrow 0$ **to** $a-2$ **do**
2: $\quad C[i] \leftarrow C[i] \oplus C[i+2a] \oplus C[i+3a]$
3: **end for**
4: $C[a-1] \leftarrow C[a-1] \oplus C[3a-1]$
5: **for** $i \leftarrow a$ **to** $2a-1$ **do**
6: $\quad C[i] \leftarrow C[i] \oplus C[i+a]$
7: **end for**
8: **return** C

general algorithm, i.e, $N_{\oplus} = 2m - 2$. The number of reduction steps given by the Eq. 5.1 is

$$k = \left\lfloor \frac{m-2}{m-a} \right\rfloor + 1 = \left\lfloor \frac{m-2}{m-(m-1)} \right\rfloor + 1 = \lfloor m-2 \rfloor + 1 = m-1.$$

Algorithm 9 Reduction algorithm for $x^m + x^a + 1$, $a = m - 1$.

Input: a, $C[2m-2, 0]$
Output: $C[m-1, 0]$
1: **for** $i \leftarrow 2m-3$ **downto** m **do**
2: $\quad C[i] \leftarrow C[i] \oplus C[i+1]$
3: **end for**
4: $C[a] \leftarrow C[a] \oplus C[m]$
5: **for** $i \leftarrow 0$ **to** $m-2$ **do**
6: $\quad C[i] \leftarrow C[i] \oplus C[i+m]$
7: **end for**
8: **return** C

5.5.6 First General case for $1 \leq a < m$

The general bit operation algorithm is presented in Algorithm 10. This algorithm obviously uses $2(m-1) = 2m-2$ XORs.

Algorithm 10 Simple reduction algorithm for $x^m + x^a + 1$, $m \neq 2a$

Input: m, a, $C[2m-2, 0]$
Output: $C[m-1, 0]$
1: **for** $i \leftarrow m-2$ **downto** 0 **do**
2: $\quad C[i+a] \leftarrow C[i+a] \oplus C[i+m]$
3: $\quad C[i] \leftarrow C[i] \oplus C[i+m]$
4: **end for**
5: **return** C

5.6 WORD OPERATION REDUCTION ALGORITHMS

5.6.1 Notations

In this paper, we assume that one word has W bits where W is a multiple of 8. The bits of a word A are numbered from 0 to $W-1$, with the rightmost bit (LSB) of A designated as bit 0. The following standard notation is used to denote operations on words A and B:

$A \oplus B$	bitwise exclusive or.
$A \mid B$	bitwise or.
$A \,\&\, B$	bitwise AND.
$A \ll n$	left shift of A by n positions, $(n < W)$, with the bits fron
$A \gg n$	righ shift of A by n positions, $(n < W)$, with bits from W
C	Vector os bits.
D	Array de elements of W bits.
D_{Red}	Reduced element.

Figure 1 shows the representation of a element $d \in \mathbb{F}_{2^m}$ as an array D of t words of W bits, where $t = \lceil \frac{m}{W} \rceil$. The $s = tW - m$ highest order bits of $D[t-1]$ are not unused.

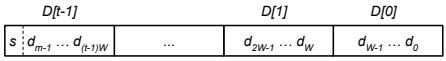

Figure 1: Representation of $d \in \mathbb{F}_{2^m}$ as an array D of W bit words. The $s = tW - m$ highest order bits of $D[t-1]$ are not unused.

For the case where $(m-1) \mod W > \frac{W}{2}$, the Figure 2 shows the representation of a element $d = ab$, $a, b \in \mathbb{F}_{2^m}$ as an array D of W-bit words. The $s = 2(tW - m) + 1$ highest order bits of $D[2t-1]$ are not unused.

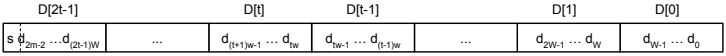

Figure 2: Representation of $d = ab$, $a, b \in \mathbb{F}_{2^m}$ as an array D of W-bit words. The $s = 2(tW - m) + 1$ highest order bits of $D[2t-1]$ are not unused.

When $(m-1) \mod W \leq \frac{W}{2}$, $D[2t-1]$ is not needed. Figure 3 shows the representation of a element $d = ab$, $a, b \in \mathbb{F}_{2^m}$ as an array D of W-bit words. The $s = 2(tW - m) + 1$ highest order bits of $D[2t-2]$ and $D[2t-1]$ are not unused.

Figure 3: Representation of $d = ab$, $a, b \in \mathbb{F}_{2^m}$ as an array D of W-bit words. The $s = 2(tW - m) + 1$ highest order bits of $D[2t-2]$ and $D[2t-1]$ are not unused.

In these cases a whole word is wasted, but length calculations become simpler: to get the length (in words) of a product of two elements, just add their lengths (in words).

5.6.2 Algorithms

The processing algorithms by word found in the literature for the irreducible polynomials are an implementation using words of the general algorithm 5 of bits. These algorithms, therefore, do not take into account the possible redundant operations inside. One explanation for this could be the fact that for the small middle exponents, this redundancy is small, almost negligible. However, for large exponents the redundancy increases considerably. If the algorithm benefit from these redundancies, you can get algorithms as effective as, or perhaps even better, for irreducibles with great exponents.

Custodio Teremos nessa seção:
a) Reducao por bit usando palavras
b) Redução usando palavras do algoritmo geral de bits;
c) Algoritmos do NITS por palavra;
d) Nossos algoritmos otimizados ($a = m/2$, $a \neq m/2$) processados por palavras;
e) Novos algoritmos para os trinomios do NITS mas usando feito usando nosso algoritmo otimizado para a primeira faixa de a;

5.6.3 General by word bit processing

For low weight irreducible polynomials with middle terms close to each other or, in particular, trinomials, the reduction may be performed efficiently by words as shown in Algorithm 11(**??**, p. 53). In this algorithm

$$r(x) = f(x) + x^m$$

Custodio Verificar o algorithm. Explicar (modificar) $C\{j\}$. Fazer uma figura semelhante a Fig. 2.9 do Hankerson para explicar de forma geral este algoritmo. Apresentar uma análise de complexidade para este algoritmo. Por exemplo, a soma re u_k pode ser feita por um *for* de W passos.

Algorithm 11 Reduction algorithm by word (one bit at a time) (Hankerson).

Input: $C[2m-2, 0]$, W
Output: $C[m-1, 0]$ *Precomputation.* Compute $u_k = x^k r(x)$, $0 \leq k \leq W-1$
1: **for** $i \leftarrow 2m-2$ **downto** m **do**
2: **if** $C[i] = 1$ **then**
3: $j \leftarrow \lfloor \frac{i-m}{W} \rfloor$
4: $k \leftarrow (i-m) - Wj$ Add $u_k(x)$ to $C\{j\}$
5: **end if**
6: **end for**
7: **return** C

5.6.4 Word operation of the case $a = 1$

Figure 4 is a representation of the case $a = 1$ and $(m-1) \bmod W > \frac{W}{2}$. In the figure, D is the element to be reduced, D_a are the bits reduced by the exponent a and D_0 are the bits reduced by 0. $D_{Red} = D \oplus D_a \oplus D_0$ is the reduced element. The hatched bits (shaded in figure) should be set to 0.

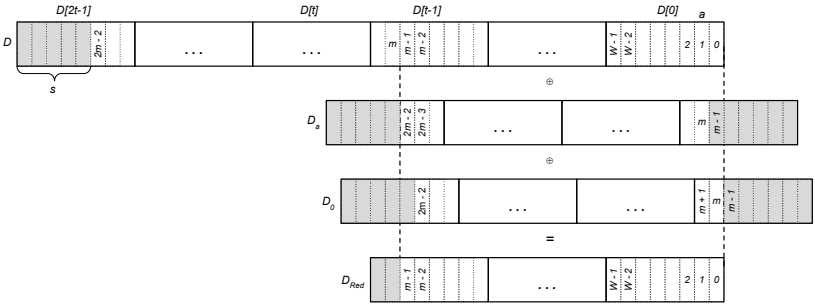

Figure 4: Word operation of the case $a = 1$.

Algorithm 12 Reduction algorithm by word for $x^m + x + 1$, from $D[2t-1]$.

Input: $D[2t-1, 0]$,t, m, W
Output: $D_{Red=D[t-1,0]}$

1: $u \leftarrow tW - m$
2: $h \leftarrow W - u$
3: $T \leftarrow D[t-1] \gg h$
4: $D[0] \leftarrow D[0] \oplus T \oplus (D[t] \ll u) \oplus (T \ll 1) \oplus (D[t] \ll (u+1))$
5: **for** $i \leftarrow 1$ **to** $t-2$ **do**
6: $\quad D[i] \leftarrow D[i] \oplus (D[i+t-1] \gg h) \oplus (D[i+t-1] \gg (h-1)) \oplus (D[i+t] \ll u) \oplus (D[i+t] \ll (u+1))$
7: **end for**
8: $T \leftarrow D[2t-1] \ll (2u+1)$ {This avoid using & to reset unused bits}
9: $D[t-1] \leftarrow D[t-1] \oplus (D[2t-2] \gg h) \oplus (D[2t-2] \gg (h-1)) \oplus (T \gg u) \oplus (T \gg (u+1))$
10: **return** D_{Red}

We have 4 XORs in line 4, $4(t-2)$ XORs in line 5, and 4 XORs

in line 8. The total is

$$N_{\oplus} = 4t.$$

We have

$$N_{Shifts} = 4t.$$

Now, we have the case where $(m-1) \mod W \leq \frac{W}{2}$. In this case, $D[2t-2]$ is not used as show in Figure 5. Thus, the algorithm procede to reduce the bits from $D[2t-1]$. The Algorithm is show in 13.

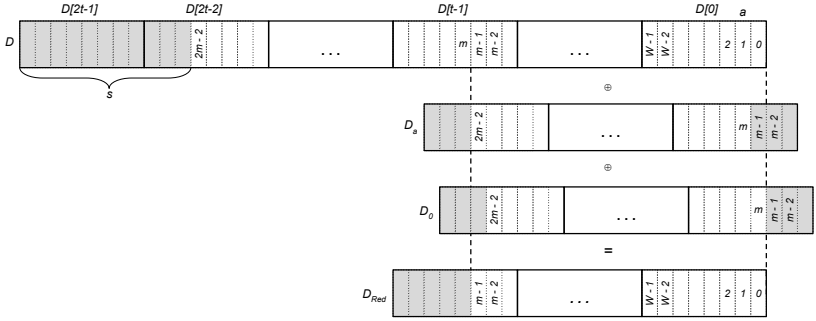

Figure 5: Word operation of the case $a = 1$ where $D[2t-1]$ is unused.

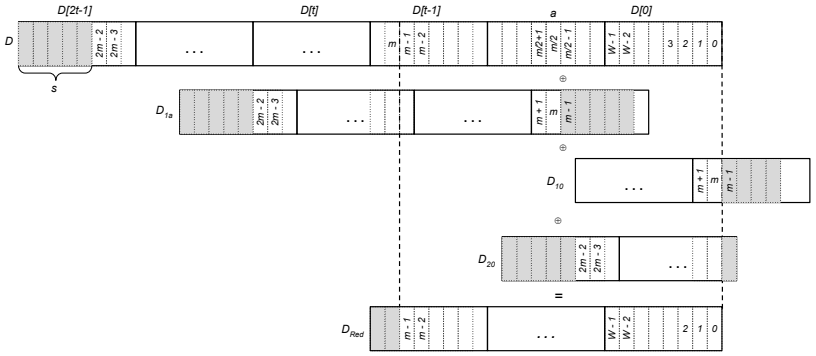

Figure 6: Word operation of the Algorithm 18 where $D[2t-1]$ is used.

We have 4 XORs in line 4, $4(t-3)$ XORs in line 5, and 4 XORs in line 8. The total is

$$N_{\oplus} = 4t - 4.$$

Algorithm 13 Reduction algorithm by word for $x^m + x + 1$, from $D[2t-2]$.

Input: $D[2t-2,0]$,t, m, W
Output: $D_{Red=D[t-1,0]}$
1: $u \leftarrow tW - m$
2: $h \leftarrow W - u$
3: $T \leftarrow D[t-1] \gg h$
4: $D[0] \leftarrow D[0] \oplus T \oplus (D[t] \ll u) \oplus (T \ll 1) \oplus (D[t] \ll (u+1)$
5: **for** $i \leftarrow 1$ **to** $t-3$ **do**
6: $\quad D[i] \leftarrow D[i] \oplus (D[i+t-1] \gg h) \oplus (D[i+t-1] \gg (h-1)) \oplus (D[i+t] \ll u) \oplus (D[i+t] \ll (u+1))$
7: **end for**
8: $T \leftarrow D[2t-2] \ll (2u+1-W)$ {This avoid using & to reset unused bits}
9: $D[t-2] \leftarrow D[t-2] \oplus (D[2t-3] \gg h) \oplus (D[2t-3] \gg (h-1)) \oplus (T \gg u) \oplus (T \gg (u+1))$
10: **return** D_{Red}

We have

$$N_{Shifts} = 4t - 4.$$

5.6.5 Word operation of the case $a < m/2$

Figure 7 is a representation of the case $a < m/2$. In the figure, D is the element to be reduced, D_a are the bits reduced by the exponent a and D_0 are the bits reduced by 0. $D_{Red} = D \oplus D_a \oplus D_0$ is the reduced element. The hatched bits (shaded in figure) should be set to 0.

5.6.6 Word operation of the Algorithm 8

Figure 8 is a representation of the algorithm 8. In the figure, D is the element to be reduced, D_a are the bits reduced by the exponent a and D_0 are the bits reduced by 0. $D_{Red} = D \oplus D_a \oplus D_0$ is the reduced element. The hatched bits (shaded in figure) should be set to 0.

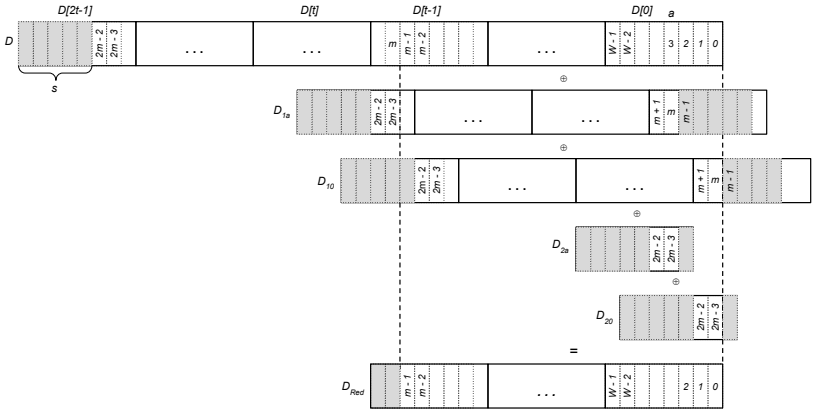

Figure 7: Word operation of the case $a < m/2$.

5.6.7 Word operation of the Algorithm 10

The word-oriented versions of the algorithm can be created by noticing the bit access pattern. The XOR operation pattern can be thought of XORing two ranges

Each of them perform a XOR operation between all bits in a certain range and counterparts a fixed distance away. This access pattern allows word-level parallelism. There are four issues that need to be paid attention to:

Misaligned start: The loop starts at $2m - 2$, which may not align with the word length ($W \nmid 2m - 2$). To avoid overwriting the upper bits of this incomplete word we must clear them from the `src` word.

Misaligned ends: The fix is identical to the misaligned start, where we clear the extra bits from the incoming word.

Misaligned source: The source/incoming word may not be aligned to word boundaries either. In this case we need to pull bits from the next word too, shifting both of them for correct placement. Unfortunately this issue affects the start and ends word too, further complicating their treatment.

Too small distances: Finally, if the distance between the XOR'ed words ($m - a$, m, respectively) is smaller than $W/2$, the same bit will be written and read in the same word-step. This is not possisble with a single operation, and requires handling each word in mulitple

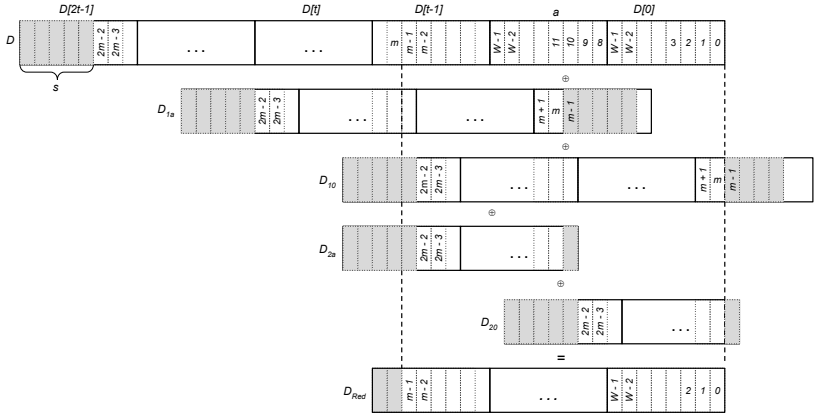

Figure 8: Word operation of the Algorithm 8.

parts. This is suboptimal and should be avoided by better choices of m, a and W.

Other bit-level operations are emulated with word masks and shifts.

To help transforming an algorithm in its word-parallel version we introduce the following helper functions. Notice the function calls can be inlined and all conditionals are open for static analysis (the final algorithm should contain no conditionals if optimized properly).

Algorithm 14 `XOR_BIT`: Single bit XOR inside word

Input: dst, src, W, $C[]$

1: $mask \leftarrow 1 << (src \mod W)$
2: $C[dst/W] \leftarrow C[dst/W] \oplus (C[src/W] \wedge mask) << (dst \mod W - src \mod W)$

Algorithm 15 `XOR_WORD`: Whole word XOR with possibly misaligned source

Input: dst, src, W, $C[]$

1: $left \leftarrow C[src/W] >> (src \mod W)$
2: $right \leftarrow C[src/W + 1] << (W - src \mod W)$
3: $C[dst/W] \leftarrow C[dst/W] \oplus (left \vee right)$

And the word-transformed algorithms are:

Algorithm 16 `XOR_PARTIAL_WORD`: XOR a range of bits inside a word, with possibly misaligned source

Input: dst, src, n_bits, W, $C[]$

1: $shift \leftarrow src \mod W$
2: $left\_n\_bits \leftarrow \min(n\_bits, W - shift)$
3: $leftmask \leftarrow ((1 << left\_n\_bits) - 1) << shift$
4: $left \leftarrow (C[src/W] \wedge leftmask) >> shift$
5: $right\_n\_bits \leftarrow n\_bits - left\_n\_bits$
6: **if** $right\_n\_bits > 0$ **then**
7: $\quad rightmask \leftarrow (1 << right\_n\_bits) - 1$
8: $\quad right \leftarrow (C[src/W + 1] \wedge rightmask) << left\_n\_bits$
9: **else**
10: $\quad right \leftarrow 0$
11: **end if**
12: $C[dst/W] = C[dst/W] \oplus ((left|right) << (dst\ W))$

Algorithm 17 `XOR_RANGE`: XOR a range of bits (across many words) with an equivalent range a certain distance away

Input: $start\_dst$, end_dst, $distance$, W, $C[]$

1: **if** $end\_dst/W = start\_dst/W$ **then**
2: $\quad$ `XOR_PARTIAL_WORD`($start\_dst$, $start\_dst + distance$, $end\_dst - start\_dst$, W, C)
3: **else**
4: $\quad$ **if** $start\_dst\ W \neq 0$ **then**
5: $\quad\quad remaining = W - (start\_dst\ W)$
6: $\quad\quad$ `XOR_PARTIAL_WORD`($start\_dst$, $start\_dst + distance$, $remaining$, W, C)
7: $\quad\quad start\_dst \leftarrow start\_dst + remaining$
8: $\quad$ **end if**
9: $\quad rounded\_end \leftarrow (end\_dst/W) * W$
10: $\quad dst \leftarrow start\_dst$
11: $\quad$ **while** $dst < rounded\_end$ **do**
12: $\quad\quad$ `XOR_WORD`(dst, $dst + distance$, W, C)
13: $\quad\quad dst \leftarrow dst + W$
14: $\quad$ **end while**
15: $\quad$ **if** $end\_dst\ W \neq 0$ **then**
16: $\quad\quad$ `XOR_PARTIAL_WORD`($rounded\_end$, $rounded\_end + distance$, $end\_dst\ W$, W, C)
17: $\quad$ **end if**
18: **end if**

Algorithm 18 Simple word-parallel reduction algorithm for $x^m + x^a + 1$, $a = \frac{m}{2}$

Input: m, a, W, $C\left[0, \left\lceil \frac{2m-2}{W} \right\rceil\right]$
Output: $C[0, m-1]$

```
1: XOR_RANGE(a, a + m, m, W, C)
2: XOR_RANGE(0, a − 1, m, W, C)
3: return C
```

Algorithm 19 Alternative reduction algorithm, uses more XORs but less calls to `XOR_RANGE`, may be useful for certain architectures.

Input: m, a, W, $C\left[0, \left\lceil \frac{2m-2}{W} \right\rceil\right]$
Output: $C[0, m-1]$

```
1: start_src ← m
2: step_size ← m − a
3: while start_src ≤ 2m − 2 do
4:     XOR_RANGE(a, a + 2m − 2 − start_src, start_src − a, W, C)
5:     start_src ← start_src + step_size
6:     step_size ← 2 × step_size
7: end while
8: XOR_RANGE(0, m, m, W, C)
9: return C
```

5.7 EXAMPLE ALGORITHMS

The algorithm 19 proposed can be applied to specific trinomials with fixed parameters. For example, for the commonly used field $x^{233}+x^{74}+1$, with 32-bit words, this results in algorithm 20. The total number of operations of this algorithm is 19 XORs, 16 ORs, 5 ANDs and 35 shifts, for a total of 75 word operations.

Algorithm 20 Reduction algorithm for $x^{233}+x^{74}+1$ with 32-bit words

Input: $C\left[0, \left\lceil \frac{2m-2}{W} \right\rceil\right]$
Output: $C[0, m-1]$

1: $C[2] \leftarrow C[2] \oplus ((C[7] \wedge \texttt{0x7ffffe00}) << 1)$
2: $C[3] \leftarrow C[3] \oplus ((C[7] >> 31) \vee (C[8] << 1))$
3: $C[4] \leftarrow C[4] \oplus ((C[8] >> 31) \vee (C[9] << 1))$
4: $C[5] \leftarrow C[5] \oplus ((C[9] >> 31) \vee (C[10] << 1))$
5: $C[6] \leftarrow C[6] \oplus ((C[10] >> 31) \vee (C[11] << 1))$
6: $C[7] \leftarrow C[7] \oplus ((C[11] >> 31) \vee (C[12] << 1))$
7: $C[8] \leftarrow C[8] \oplus ((C[12] >> 31) \vee (C[13] << 1))$
8: $C[9] \leftarrow C[9] \oplus ((C[13] >> 31) \vee ((C[14] \wedge \texttt{0x1ffff}) >> 1))$
9: $C[2] \leftarrow C[2] \oplus ((C[12] \wedge \texttt{0x3fffff00}) << 2)$
10: $C[3] \leftarrow C[3] \oplus ((C[12] >> 30) \vee (C[13] << 2))$
11: $C[4] \leftarrow C[4] \oplus ((C[13] >> 30) \vee ((C[14] \wedge \texttt{0x1ffff}) >> 2))$
12: $C[0] \leftarrow C[0] \oplus ((C[7] >> 9) \vee (C[8] << 23))$
13: $C[1] \leftarrow C[1] \oplus ((C[8] >> 9) \vee (C[9] << 23))$
14: $C[2] \leftarrow C[2] \oplus ((C[9] >> 9) \vee (C[10] << 23))$
15: $C[3] \leftarrow C[3] \oplus ((C[10] >> 9) \vee (C[11] << 23))$
16: $C[4] \leftarrow C[4] \oplus ((C[11] >> 9) \vee (C[12] << 23))$
17: $C[5] \leftarrow C[5] \oplus ((C[12] >> 9) \vee (C[13] << 23))$
18: $C[6] \leftarrow C[6] \oplus ((C[13] >> 9) \vee (C[14] << 23))$
19: $C[7] \leftarrow C[7] \oplus ((C[14] \wedge \texttt{0x3fe00}) >> 9)$
20: **return** C

5.8 NIST POLYNOMIALS AND THEIR COSTS

NIST Binary Polynomials and their costs (Guide to Elliptic Curve Crytography - Hankerson - page 76

- $x^{163}+x^7+x^6+x^3+1$ - $C[0..10]$, 41 XORs, 36 SHIFTs, 1 AND = 78 operations

- $x^{233} + x^{74} + 1$ - C[0..15!], 35 XORs, 35 SHIFTs, 1 AND = 71 operations
- $x^{283} + x^{12} + x^7 + x^5 + 1$ - C[0..17], 76 XORs, 76 SHIFTs, 1 AND = 153 operations
- $x^{409} + x^{87} + 1$ - C[0..25], 54 XORs, 54 SHIFTs, 1 AND = 109 operations
- $x^{571} + x^{10} + x^5 + x^2 + 1$ - C[0..35], 148 XORs, 148 SHIFTs, 1 AND = 297 operations

Pentanomials of same degree from the family $x^{b+2c} + x^{b+c} + x^b + x^c + 1$

- $x^{233} + x^{158} + x^{83} + x^{74} + 1$, score 3.717, $k_a = 4$
- $x^{163} + x^{117} + x^{71} + x^{46} + 1$, score 3.982, $k_a = 4$
- $x^{409} + x^{294} + x^{179} + x^{115} + 1$, score 4.015, $k_a = 4$
- $x^{409} + x^{337} + x^{265} + x^{72} + 1$, score 4.457, $k_a = 6$
- $x^{233} + x^{217} + x^{201} + x^{16} + 1$, score 7.240, $k_a = 15$
- ??571??

6 VISUAL DEBUGGER

7 EVALUTION?

8 FINAL CONSIDERATIONS

8.1 REFERENCES

REFERENCES

HANKERSON, D.; MENEZES, A. J.; VANSTONE, S. *Guide to Elliptic Curve Cryptography*. [S.l.]: Springer Science & Business Media, 2006.

SUNAR, B.; KOC, C. K. Mastrovito multiplier for all trinomials. *IEEE Transactions on Computers*, IEEE, v. 48, n. 5, p. 522–527, 1999.

Appendices

.1 EXAMPLE OF RESULTING ALGORITHM

Algorithm 21 Squaring for $\mathbb{F}_{2^{19}} \cong \mathbb{F}_2[x]/(x^{19}+x^5+x^2+x+1)$

Input: $A = [a_0, a_1, a_2, ..., a_{18}]$,
$f(x) = x^{19}+x^5+x^2+x+1$
Output: $A^2 \mod f$

1: $C = [a_0, 0, a_1, 0, a_2, 0, \ldots, 0, a_{18}]$
2: $C[22] \leftarrow C[22] \oplus C[36]$
3: $C[19] \leftarrow C[36]$
4: $C[18] \leftarrow C[18] \oplus C[36]$
5: $C[17] \leftarrow C[36]$
6: $C[20] \leftarrow C[20] \oplus C[34]$
7: $C[17] \leftarrow C[17] \oplus C[34]$
8: $C[16] \leftarrow C[16] \oplus C[34]$
9: $C[15] \leftarrow C[34]$
10: $C[18] \leftarrow C[18] \oplus C[32]$
11: $C[15] \leftarrow C[15] \oplus C[32]$
12: $C[14] \leftarrow C[14] \oplus C[32]$
13: $C[13] \leftarrow C[32]$
14: $C[16] \leftarrow C[16] \oplus C[30]$
15: $C[13] \leftarrow C[13] \oplus C[30]$
16: $C[12] \leftarrow C[12] \oplus C[30]$
17: $C[11] \leftarrow C[30]$
18: $C[14] \leftarrow C[14] \oplus C[28]$
19: $C[11] \leftarrow C[11] \oplus C[28]$
20: $C[10] \leftarrow C[10] \oplus C[28]$
21: $C[9] \leftarrow C[28]$
22: $C[12] \leftarrow C[12] \oplus C[26]$
23: $C[9] \leftarrow C[9] \oplus C[26]$
24: $C[8] \leftarrow C[8] \oplus C[26]$
25: $C[7] \leftarrow C[26]$
26: $C[10] \leftarrow C[10] \oplus C[24]$
27: $C[7] \leftarrow C[7] \oplus C[24]$
28: $C[6] \leftarrow C[6] \oplus C[24]$
29: $C[5] \leftarrow C[24]$
30: $C[8] \leftarrow C[8] \oplus C[22]$
31: $C[5] \leftarrow C[5] \oplus C[22]$
32: $C[4] \leftarrow C[4] \oplus C[22]$
33: $C[3] \leftarrow C[22]$
34: $C[6] \leftarrow C[6] \oplus C[20]$
35: $C[3] \leftarrow C[3] \oplus C[20]$
36: $C[2] \leftarrow C[2] \oplus C[20]$
37: $C[1] \leftarrow C[20]$
38: $C[5] \leftarrow C[5] \oplus C[19]$
39: $C[2] \leftarrow C[2] \oplus C[19]$
40: $C[1] \leftarrow C[1] \oplus C[19]$
41: $C[0] \leftarrow C[0] \oplus C[19]$
42: **return** $C[0], C[1], C[2], \ldots, C[18]$

Algorithm 21 is an instance of Algorithm **??**, wherein the irreducible polynomial is defined as $x^{19}+x^5+x^2+x+1$ by simply fixing f. We note the circuit has a delay of $3T_X$, due to the critical paths of the circuit lines $C[5]$ and $C[2]$. However, this value can be reduced with manual tweaks. By eliminating Lines 29, 31 and 38, and inserting $C[5] \leftarrow C[22] \oplus C[24]$ and $C[5] \leftarrow C[5] \oplus C[19]$ at the beginning, the delay of $C[5]$ is reduced to $2T_X$ (and one XOR is saved). By moving Line 39 to before Line 36 the critical path of $C[2]$ is also reduced to only $2T_X$, which becomes the total delay of the full circuit.