Applying Cryptographic Acceleration Techniques to Error Correction

Rémi Géraud[2,4], Diana-Ştefania Maimuţ[2], David Naccache[1,2], Rodrigo Portella do Canto[1], Emil Simion[3]

[1] Sorbonne Universités – Université Paris II
12 Place du Panthéon, F-75231, Paris, France
rodrigo.portella-do-canto@etudiants.u-paris2.fr
[2] École normale supérieure, Département d'informatique
45, rue d'Ulm, F-75230, Paris CEDEX 05, France
given_name.family_name@ens.fr
[3] University Politehnica of Bucharest
313 Splaiul Independenţei, Bucharest, Romania
esimion@fmi.unibuc.ro
[4] Ingenico Group
28-32 Boulevard de Grenelle, 75015 Paris, France
remi.geraud@ingenico.com

Abstract. Modular reduction is the basic building block of many public-key cryptosystems. BCH codes require repeated polynomial reductions modulo the same constant polynomial. This is conceptually very similar to the implementation of public-key cryptography where repeated modular reduction in $\mathbb{Z}_n$ or $\mathbb{Z}_p$ are required for some fixed n or p. It is hence natural to try and transfer the modular reduction expertise developed by cryptographers during the past decades to obtain new BCH speed-up strategies. Error correction codes (ECCs) are deployed in digital communication systems to enforce transmission accuracy. BCH codes are a particularly popular ECC family. This paper generalizes Barrett's modular reduction to polynomials to speed-up BCH ECCs. A BCH$(15, 7, 2)$ encoder was implemented in Verilog and synthesized. Results show substantial improvements when compared to traditional polynomial reduction implementations. We present two BCH code implementations (regular and pipelined) using Barrett polynomial reduction. These implementations, are respectively 4.3 and 6.7 faster than an improved BCH LFSR design. The regular Barrett design consumes around 53% less power than the BCH LFSR design, while the faster pipelined version consumes 2.3 times more power than the BCH LFSR design.

1 Introduction

Modular reduction (*e.g.* [3, 4, 8, 10]) is the basic building block of many public-key cryptosystems. We refer the reader to [3] for a detailed comparison of various modular reduction strategies.

BCH codes are widely used for error correction in digital systems, memory devices and computer networks. For example, the shortened BCH (48,36,5) was accepted by the U.S. Telecommunications Industry Association as a standard for the cellular Time Division Multiple Access protocol (TDMA) [11]. Another example is BCH(511, 493) which was adopted by International Telecommunication Union as a standard for video conferencing and video phone codecs (Rec. H.26) [5]. BCH codes require repeated polynomial reductions modulo the same constant polynomial. This is conceptually very similar to the implementation of public-key cryptography where repeated modular reduction in $\mathbb{Z}_n$ or $\mathbb{Z}_p$ are required for some fixed n or p [1].

It is hence natural to try and transfer the modular reduction expertise developed by cryptographers during the past decades to obtain new BCH speed-up strategies. This work focuses on the "polynomialization" of Barrett's modular reduction algorithm [1]. Barrett's method creates the operation $a \bmod b$ from bit shifts, multiplications and additions in $\mathbb{Z}$. This allows to build modular reduction at very marginal code or silicon costs by leveraging existing hardware or software multipliers.

Reduction modulo fixed multivariate polynomials is also very useful in other fields such as robotics and computer algebra (*e.g.* for computing Gröbner bases).

Structure of the paper: Section 2 recalls Barrett's algorithm. Section 3 presents our main theoretical results, *i.e.* a polynomial variant of [1]. Section 4 recalls the basics of BCH error correcting codes (ECC). Section 4.2 describes the integration of the Barrett polynomial variant in a BCH circuit and provides benchmark results.

2 Barrett's Reduction Algorithm

Notations. $\|x\|$ will denote the bit-length of x throughout this paper.

$y \gg z$ will denote binary shift-to-the-right of y by z bits *i.e.*:

$$y \gg z = \left\lfloor \frac{y}{2^z} \right\rfloor .$$

Barrett's algorithm (Algorithm 1) approximates the result $c = d \bmod n$ by a quasi-reduced integer $c+\epsilon n$, where $0 \leq \epsilon \leq 2$. Let $N = \|n\|, D = \|d\|$

and fix a maximal bit-length reduction capacity L such that $N \leq D \leq L$. The algorithm will work if $D \leq L$. In most implementations, $D = L = 2N$. The algorithm uses the pre-computed constant $\kappa = \lfloor 2^L/n \rfloor$ that depends only on n and L. The reader is referred to [1] for a proof and an analysis of Algorithm 1.

Algorithm 1: Barrett's Algorithm

Input: $n < 2^N, d < 2^D, \kappa = \left\lfloor \frac{2^L}{n} \right\rfloor$ where $N \leq D \leq L$
Output: $c = d \bmod n$

$c_1 \leftarrow d \gg (N-1)$
$c_2 \leftarrow c_1 \kappa$
$c_3 \leftarrow c_2 \gg (L-N+1)$
$c_4 \leftarrow d - nc_3$
while $c_4 \geq n$ **do**
  $c_4 \leftarrow c_4 - n$
end
return c_4

Example 1. Reduce $8619 \bmod 93 = 63$.

$n = 93 \Rightarrow N = 7$

$\kappa = \left\lfloor \frac{2^{32}}{n} \right\rfloor$ =10110000001011000000101100
$d = 8619$ =10000110101011
c_1 =1000011010101 1 = 10000110
c_2 =10111000011011100001101110000 1000
c_3 =1011100 00110111000011011100001000 =1011100
nc_3 =10000101101100
c_4 = 63

Work Factor: $\|c_1\| = D - N + 1 \simeq D - N$ and $\|\kappa\| = L - N$ hence their product requires $w = (D-N)(L-N)$ elementary operations. $\|c_3\| = (D-N) + (L-N) - (L-N+1) = D - N - 1 \simeq D - N$. The product nc_3 will therefore claim $w' = (D-N)N$ elementary operations. All in all, work amounts to $w + w' = (D-N)(L-N) + (D-N)N = (D-N)L$.

2.0.1 Dynamic Constant Scaling The constant κ can be adjusted on the fly thanks to Lemma 1.

Lemma 1. *If $U \leq L$, then $\bar{\kappa} = \kappa \gg U = \left\lfloor \frac{2^{L-U}}{n} \right\rfloor$.*

Proof. $\exists\, \alpha < 2^U$ and $\beta < n$ (integers) verifying:

$$\bar{\kappa} = \frac{\kappa}{2^U} - \frac{\alpha}{2^U} \text{ and } \kappa = \frac{2^L}{n} - \frac{\beta}{n}.$$

Therefore,

$$\min_{\alpha\,\beta} \left(\frac{2^{L-U}}{n} - \frac{\beta + \alpha n}{2^U n} \right) \leq \bar{\kappa} = \frac{2^{L-U}}{n} - \frac{\beta + \alpha n}{2^U n} \leq \max_{\alpha,\beta} \left(\frac{2^{L-U}}{n} - \frac{\beta + \alpha n}{2^U n} \right)$$

and finally,

$$\frac{2^{L-U}}{n} - 1 < \frac{2^{L-U}}{n} - 1 + \frac{1}{2^U n} \leq \bar{\kappa} \leq \frac{2^{L-U}}{n}.$$

□

Work factor: We know that $\bar{\kappa} = \kappa \gg L - D$. Let $c_5 = D - N + 1$. Replacing step 4 of Algorithm 1 with

$$c_6 \leftarrow d - n(\bar{\kappa} c_1 \gg c_5),$$

the multiplication of c_1 by $\bar{\kappa}$ (κ adjusted to $D - N$ bits, shifting by $L - D$ bits to the right), will be done in $O((D-N)^2)$.

Hence, the new work factor decreases to $(D-N)^2 + N(D-N) = (D - N)D$.

Example 2. Reconsidering example 1, *i.e.* computing 8619 mod 93 using the above technique, we obtain:

$D = \lceil \log_2 8619 \rceil = 14$
$\bar{\kappa}$ $=10110000\ \sout{00101100000101100}$
c_1 $=10000110\ \sout{101011}$ $=10000110$
$\bar{\kappa} c_1$ $=101110000100000$
$\bar{\kappa} c_1 \gg c_5$ $=1011100\ \sout{00100000}$
$n(\bar{\kappa} c_1 \gg c_5)$ $=10000101101100$
c_6 $= 63$

3 Barrett's Algorithm for Polynomials

3.1 Orders

Definition 1 (Monomial Order). *Let P, Q and R be three monomials in ν variables. $\rhd$ is a monomial order if the following conditions are fulfilled:*

- $P \rhd 1$
- $P \rhd Q \Rightarrow \forall R,\ PR \rhd QR$

Example 3. The lexicographic order on exponent vectors defined by

$$\prod_{i=1}^{\nu} x^{a_i} \succ \prod_{i=1}^{\nu} x^{b_i} \Leftrightarrow \exists i, a_j = b_j \text{ for } i < j \text{ and } a_i > b_i$$

is a monomial order. We denote the lexicographic order by $\succ$.

3.2 Terminology

In the following, capital letters will next denote polynomials and $\nu \in \mathbb{N}$.

Let $P = \sum_{i=0}^{\alpha} p_i \prod_{j=1}^{\nu} x_j^{y_{j,i}} \in \mathbb{Q}[\boldsymbol{x}] = \mathbb{Q}[x_1, x_2, ..., x_\nu]$.

The leading term of P according to $\rhd$, will be denoted by $\text{lt}(P) = p_0 \prod_{j=1}^{\nu} x_j^{y_{j,0}}$.

The leading coefficient of P according to $\rhd$ will be denoted by $\text{lc}(P) = p_0 \in \mathbb{Q}$.

The quotient $\text{lm}(P) = \dfrac{\text{lt}(P)}{\text{lc}(P)} = \prod_{j=1}^{\nu} x_j^{y_{j,0}}$ is the leading monomial of P according to $\rhd$.

The above notations generalize the notion of degree to exponent vectors:

$$\deg(P) = \deg(\text{lm}(P)) = \boldsymbol{y_0} = \langle y_{0,0}, \ldots, y_{\nu,0} \rangle.$$

Example 4. For $\succ$ and $P(x, y) = 2x_1^2x_2^2 + 11x_1 + 15$, we have:

$$\text{lt}(P) = 2x_1^2x_2^2,\ \text{lm}(P) = x_1^2x_2^2, \deg(P) = \langle 2, 2 \rangle \text{ and } \text{lc}(P) = 2.$$

Definition 2 (Reduction Step). *Let $P,\ Q \in \mathbb{Q}[\boldsymbol{x}]$. We denote by $Q \underset{P}{\to} Q_1$ the* **reduction step** *of Q (with respect to P and according to $\triangleright$) defined as the result given by the following operations:*

1. *Find a term t of Q such that monomial(t)=$\mathrm{lm}(P)m$*
2. *If such a t exists, return $Q_1 = Q - \dfrac{Pm}{\mathrm{lc}(P)}$. Else return $Q_1 = Q$.*

Example 5. Let $Q(x_1, x_2) = 3x_1^2x_2^2$ and $P(x_1, x_2) = 2x_1^2x_2 - 1$.

The reduction step of Q (with respect to P) is $Q \underset{\mathrm{P}}{\to} Q_1 = \dfrac{3x_2}{2}$.

Lemma 2. *Let $P,\ Q \in \mathbb{Q}[\boldsymbol{x}]$ and $\{Q_i\}$ such that $Q \underset{P}{\to} Q_1 \underset{P}{\to} Q_2 \underset{P}{\to} \dots$*

1. *$\exists i \in \mathbb{N}$ such that $j \geq i \Rightarrow Q_j = Q_i$*
2. *Q_i is unique*

We denote $Q \underset{P}{\overset{*}{\to}} Q_i = Q \bmod P$ and $\left\lfloor \dfrac{Q}{P} \right\rfloor = \dfrac{Q - Q \bmod P}{P} \in \mathbb{Q}[\boldsymbol{x}]$ and call Q_i the "residue of Q (with respect to P and according to $\triangleright$)".

Example 6. Euclidean division is a reduction in which $i = 1$.

3.3 Barrett's Algorithm for Multivariate Polynomials

We will now adapt Barrett's algorithm to $\mathbb{Q}[\boldsymbol{x}]$.

Barrett's algorithm and Lemma 1 can be generalised to $\mathbb{Q}[\boldsymbol{x}]$, by shifting polynomials instead of shifting integers.

Definition 3 (Polynomial Right Shift). *Let $P = \sum_{i=0}^{\alpha} p_i \prod_{j=1}^{\nu} x_j^{y_{j,i}} \in \mathbb{Q}[\boldsymbol{x}]$ and $\boldsymbol{a} = \langle a_1, a_2, ..., a_\nu \rangle \in \mathbb{N}^\nu$. We denote*

$$P \gg \boldsymbol{a} = \sum_{\varphi(\boldsymbol{a})} p_i \prod_{j=1}^{\nu} x_j^{y_{j,i} - a_i} \in \mathbb{Q}[\boldsymbol{x}],\ \textit{where } \varphi(\boldsymbol{a}) = \{i,\ \forall j,\ y_{i,j} \geq a_i\}.$$

Example 7.

If $P(x) = 17x^7 + 26x^6 + 37x^4 + 48x^3 + 11,$ then $P \gg \langle 5 \rangle = 17x^2 + 26x.$

Theorem 1 (Barrett's Algorithm for Polynomials). *Let:*

- $P = \sum_{i=0}^{\alpha} p_i \prod_{j=1}^{\nu} x_j^{y_{j,0}} \in \mathbb{Q}[\boldsymbol{x}]$ *and* $Q = \sum_{i=0}^{\beta} q_i \prod_{j=1}^{\nu} x_j^{w_{j,i}} \in \mathbb{Q}[\boldsymbol{x}]$ *s.t.* $\mathrm{lm}(Q) \rhd \mathrm{lm}(P)$
- $L \geq \max(w_{i,j}) \in \mathbb{N}, h(L) = \prod_{j=1}^{\nu} x_j^L$ *and* $K = \left\lfloor \frac{h(L)}{P} \right\rfloor$
- $\boldsymbol{y_0} = \langle y_{1,0}, y_{2,0}, ..., y_{\nu,0} \rangle \in \mathbb{N}^{\nu}$

Given the above notations, $(K(Q \gg \boldsymbol{y_0})) \gg (\langle L^{\nu} \rangle - \boldsymbol{y_0}) = \left\lfloor \frac{Q}{P} \right\rfloor$.

Proof. Let $G = h(L) \mod P$ and $B = (\mathrm{K}(Q \gg \boldsymbol{y_0})) = \frac{h(L) - G}{P} \left\lfloor \frac{Q}{\mathrm{lm}(P)} \right\rfloor$.

$$\Downarrow$$

$$B = \frac{\sum_{\varphi(\boldsymbol{y_0})} q_i \prod_{j=1}^{\nu} x_j^{L + w_{j,i} - y_{j,0}} - G \sum_{\varphi(\boldsymbol{y_0})} q_i \prod_{j=1}^{\nu} x_j^{w_{j,i} - y_{j,0}}}{P}$$

Applying the definition of "$\gg$", we obtain

$$B \gg (\langle L \rangle^{\nu} - \boldsymbol{y_0}) = \deg_{\geq \mathbf{0}} \frac{Q_{\varphi(\boldsymbol{y_0})} - G \sum_{\varphi(\boldsymbol{y_0})} q_i \prod_{j=1}^{\nu} x^{w_{j,i} - L}}{P}, \text{ where } \mathbf{0} = \langle 0 \rangle^{\nu}.$$

Thus,

$$B \gg (\langle L^{\nu} \rangle - \boldsymbol{y_0}) = \left\lfloor \frac{Q_{\varphi(\boldsymbol{y_0})}}{P} \right\rfloor - \deg_{\geq \mathbf{0}} \frac{G}{P} \sum_{\varphi(\boldsymbol{y_0})} q_i \prod_{j=1}^{\nu} x^{w_{j,i} - L} = \left\lfloor \frac{Q_{\varphi(\boldsymbol{y_0})}}{P} \right\rfloor.$$

We know that

$$P \rhd G \text{ and } L \geq \max(w_{i,j}), \text{ therefore } \deg_{\geq \mathbf{0}} \frac{G}{P} \sum_{\varphi(\boldsymbol{y_0})} q_i \prod_{j=1}^{\nu} x^{w_{j,i} - L} = 0.$$

Let $\bar{Q}$ be the irreducible polynomial with respect to P, obtained by removing from Q the terms that exceed $\mathrm{lm}(P)$.

$$\left\lfloor \frac{Q_{\varphi(\boldsymbol{y})}}{P} \right\rfloor = \frac{Q_{\varphi(\boldsymbol{y})} - (Q_{\varphi(\boldsymbol{y})} \bmod P)}{P} = \frac{(Q - \bar{Q})((Q - \bar{Q}) \bmod P)}{P}.$$

Hence,

$$B \gg (\langle L \rangle^{\nu} - \boldsymbol{y_0}) = \frac{(Q - \bar{Q})((Q - \bar{Q}) \bmod P)}{P}$$

$$\Downarrow$$

$$B \gg (\langle L \rangle^{\nu} - \boldsymbol{y_0}) = \left\lfloor \frac{Q}{P} \right\rfloor - \frac{\bar{Q} - \bar{Q} \bmod P}{P} = \left\lfloor \frac{Q}{P} \right\rfloor.$$

□

Algorithm 2: Polynomial Barrett Algorithm

Input: $P, Q \in \mathbb{Q}[\boldsymbol{x}]$ s.t. $P \rhd Q$

$h(L) = \boldsymbol{x}^L, \boldsymbol{y_0} = \deg P$ and $K = h(L) \bmod P$, where $\deg Q \leq \langle L, \ldots, L \rangle$

Output: $R = Q \bmod P$

1 $B \leftarrow (K(Q \gg \boldsymbol{y_0})) \gg (L - \boldsymbol{y_0})$

2 $R \leftarrow Q - BP$

3 **return** R

Remark. Let $Q = \sum_{i=0}^{\alpha} q_{i,j} \prod_{j=1}^{\nu} x_j^{w_{j,i}}$, $K = \sum_{i=0}^{\beta} k_{i,j} \prod_{j=1}^{\nu} x_j^{t_{j,i}}$, $\boldsymbol{y} = \langle y_1, ..., y_\nu \rangle$ and $\boldsymbol{z} = \langle z_1, ..., z_\nu \rangle$.

Let us have a closer look at the expression $B = (K(Q \gg \boldsymbol{y})) \gg \boldsymbol{z}$.

Given the final shifting by $\boldsymbol{z}$, the multiplication of K by $Q \gg \boldsymbol{y}$ can be optimised by being only partially accomplished. Indeed, during multiplication, we only have to form monomials whose exponent vectors $\boldsymbol{b} = \boldsymbol{w_i} + \boldsymbol{t_{i'}} - \boldsymbol{y} - \boldsymbol{z} = \langle b_1, ..., b_\nu \rangle$ are such that $b_j \geq 0$ for $1 \leq j \leq \nu$.

We implicitly apply the above in the following example.

Example 8. Let

$\rhd = \succ$

$P = x_1^2 x_2^2 + x_1^2 + 2x_1 x_2^2 + 2x_1 x_2 + x_1 + 1$

$Q = x_1^3 x_2^3 - 2x_1^3 + x_2^2 x_2^2 + 3.$

We let $L = 6$ and we observe that $\nu = 2$. We pre-compute K:

$$K = x_1^4x_2^4 - x_1^4x_2^2 + x_1^4 - 2x_1^3x_2^4 - 2x_1^3x_2^3 + 3x_1^3x_2^2 + 4x_1^3x_2 - 4x_1^3 +$$

$$4x_1^2x_2^4 + 8x_1^2x_2^3 - 5x_1^2x_2^2 - 20x_1^2x_2 + 3x_1^2 - 8x_1x_2^4 - 24x_1x_2^3 +$$

$$68x_1x_2 + 36x_1 + 16x_2^4 + 64x_2^3 + 36x_2^2 - 184x_2 - 239.$$

We first shift Q by $\boldsymbol{y_0} = \langle 2, 2\rangle$, which is the vector of exponents for $\mathrm{lm}(P)$.

$$Q \gg \boldsymbol{y_0} = (x_1^3x_2^3 - 2x_1^3 + x_2^2x_2^2 + 3) \gg \langle 2, 2\rangle = (x_1x_2 + 1)$$

Then, we compute $K(x_1x_2+1) = x_1^5x_2^5 - 2x_1^4x_2^5 - x^4y^4 + \{\text{terms} \prec x_1^4x_2^4\}$. This result shifted by $\langle L\rangle^\nu - \boldsymbol{y_0} = \langle 6, 6\rangle - \langle 2, 2\rangle = \langle 4, 4\rangle$ to the right gives:

$$A = x_1^5x_2^5 - 2x_1^4x_2^5 - x^4y^4 + \{\text{terms} \succ x_1^4x_2^4\} \gg \langle 4, 4\rangle = x_1x_2 - 2x_2 - 1.$$

It is easy to verify that:

$$Q - PA =$$

$$= (x_1^3x_2^3 - 2x_1^3 + x_1^2x_2^2 + 3) - (x_1^2x_2^2 + x_1^2 + 2x_1x_2^2 + 2x_1x_2 + x_1 + 1)(x_1x_2 - 2x_2 - 1)$$

$$\Downarrow$$

$$Q - PA = 4x_1x_2^3 + 6x_1x_2^2 - x_1^3x_2 + x_1^2x_2 + 3x_1x_2 + 2x_2 - 2x_1^3 + x_1^2 + x_1 + 4 \prec P.$$

Complexity: We refer the reader to Appendix A for a detailed computation of the complexity of Algorithm 2.

3.4 Dynamic Constant Scaling in $\mathbb{Q}[\boldsymbol{x}]$

Lemma 3. *If* $0 \leq u \leq L$, *then* $\bar{K} = K \gg \langle u\rangle^\nu = \left\lfloor \frac{h(L-u)}{P} \right\rfloor$.

Proof. $K = \left\lfloor \dfrac{h(L)}{P} \right\rfloor \Rightarrow\ K = \dfrac{h(L) - h(L) \bmod P}{P}$.

$$\text{Let } G = h(L) \bmod P \Rightarrow K = \frac{\prod_{j=1}^{\nu} {x_j}^L - G}{P}.$$

Since

$$\langle u\rangle^{\nu} \in \mathbb{N}^{\nu} \Rightarrow K \gg \langle u\rangle^{\nu} = \deg_{\geq \mathbf{0}} \frac{\prod_{j=1}^{\nu} {x_j}^{L-u} - G_{\varphi(\langle u\rangle^{\nu})}}{P}$$

$$\Downarrow$$

$$K \gg \langle u\rangle^{\nu} = \deg_{\geq \mathbf{0}} \frac{\prod_{j=1}^{\nu} {x_j}^{L-u}}{P} - \deg_{\geq \mathbf{0}} \frac{G_{\varphi(\langle u\rangle^{\nu})}}{P}.$$

We know that $P \rhd G,$ thus $P \rhd G_{\varphi(\langle u\rangle^{\nu})},$ thus $\deg_{\geq \mathbf{0}} \frac{G_{\varphi(\langle u\rangle^{\nu})}}{P} = 0.$

Finally,

$$K \gg \langle u\rangle^{\nu} = \left\lfloor \frac{\prod_{j=1}^{\nu} {x_j}^{L-u}}{P} \right\rfloor = \left\lfloor \frac{h(L-u)}{P} \right\rfloor.$$

□

Example 9. Let

$\rhd = \succ$

$P = x_1^2x_2^2 + x_1^2 + 2x_1x_2^2 + 2x_1x_2 + x_1 + 1$

$Q = x_1^3x_2^3 - 2x_1^3 + x_2^2x_2^2 + 3.$

We let $u = 4$ and we observe that $\nu = 2$. We pre-compute $\bar{K}$:

$\bar{K} = x_1^2x_2^2 - x_1^2 - 2x_1x_2^2 - 2x_1x_2 + 3x_1 + 4x_2^2 + 8x_2 - 5.$

We first shift Q by $\boldsymbol{y_0} = \langle 2, 2\rangle$, which is the vector of exponents for $\mathrm{lm}(P)$.

$$Q \gg \boldsymbol{y_0} = (x_1^3x_2^3 - 2x_1^3 + x_2^2x_2^2 + 3) \gg \langle 2, 2\rangle = (x_1x_2 + 1)$$

Then, we compute $\bar{K}(x_1x_2+1) = x_1^3x_2^3 - 2x_1^2x_2^3 - x_1^2x_2^2 + \{\text{terms} \prec x_1^2x_2^2\}$.

This result shifted by $\langle u\rangle^{\nu} - \boldsymbol{y_0} = \langle 4, 4\rangle - \langle 2, 2\rangle = \langle 2, 2\rangle$ to the right gives:

$$A = x_1^3x_2^3 - 2x_1^2x_2^3 - x_1^2x_2^2 + \{\text{terms} \succ x_1^2x_2^2\} \gg \langle 2, 2\rangle = x_1x_2 - 2x_2 - 1.$$

It is easy to verify that:

$$Q - PA =$$

$$= (x_1^3x_2^3-2x_1^3+x_1^2x_2^2+3)-(x_1^2x_2^2+x_1^2+2x_1x_2^2+2x_1x_2+x_1+1)(x_1x_2-2x_2-1)$$

$$\Downarrow$$

$$Q-PA = 4x_1x_2^3+6x_1x_2^2-x_1^3x_2+x_1^2x_2+3x_1x_2+2x_2-2x_1^3+x_1^2+x_1+4 \prec P.$$

4 Application to BCH Codes

4.1 General Remarks

BCH codes are cyclic codes that form a large class of multiple random error-correcting codes. Originally discovered as binary codes of length $2^m - 1$, BCH codes were subsequently extended to non-binary settings. Binary BCH codes are a generalization of Hamming codes, discovered by Hocquenghem, Bose and Chaudhuri [2, 4] featuring a better error correction capability. Gorestein and Zierler [6] generalised BCH codes to p^m symbols, for p prime. Two important BCH code sub-classes exist. Typical representatives of these sub-classes are Hamming codes (binary BCH) and Reed Solomon codes (non-binary BCH).

Terminology: We further refer to the vectors of an error correction code as *codewords*. The codewords' size is called the *length* of the code. The *distance* between two codewords is the number of coordinates at which they differ. The *minimum distance* of a code is the minimum distance between two codewords.

Recall that a *primitive element* of a finite field is a generator of the multiplicative group of the field.

4.1.1 BCH Preliminaries

Definition 4. *Let $m \geq 3$. For a length $n = 2^m-1$, a distance d and a primitive element $\alpha \in \mathbb{F}_{2^m}^*$, we define the binary BCH code:*

$$\mathrm{BCH}(n,d) = \{(c_0, c_1, ..., c_{n-1}) \in \mathbb{F}_2^n \mid c(x) = \sum_{i=0}^{n-1} c_i x^i \text{ satisfies}$$

$$c(\alpha) = c(\alpha^2) = ... = c(\alpha^{d-1})\}$$

Let $m \geq 3$ and $0 < t < 2^{m-1}$ be two integers. There exists a binary BCH code (called a $t-$error correcting BCH code) with parameters $n = 2^m - 1$ (the block length), $n - k \leq mt$ (the number of parity-check digits) and $d \geq 2t + 1$ (the minimum distance).

Definition 5. *Let α be a primitive element in $\mathbb{F}_{2^m}$. The* generator polynomial $g(x) \in \mathbb{F}_2[x]$ *of the $t-$error-correcting BCH code of length 2^{m-1} is the lowest-degree polynomial in $\mathbb{F}_2[x]$ having roots $\alpha, \alpha^2, ..., \alpha^{2t}$.*

Definition 6. *Let $\phi_i(x)$ be the minimal polynomial of α^i. Then,*

$$g(x) = \text{lcm}\{\phi_1(x), \phi_2(x), ..., \phi_{2t}(x)\}.$$

The degree of $g(x)$, which is the number of parity-check digits $n - k$, is at most mt.

Let $i \in \mathbb{N}$ and denote $i = 2^r j$ for odd j and $r \geq 1$. Then $\alpha^i = (\alpha^j)^{2^r}$ is a conjugate of α^j which implies that α^i and α^j have the same minimal polynomial, and therefore $\phi_i(x) = \phi_j(x)$. Consequently, the generator polynomial $g(x)$ of the t-error correcting BCH code can be written as follow:

$$g(x) = \text{lcm}\{\phi_1(x), \phi_3(x), \phi_3(x), ..., \phi_{2t-1}(x)\}.$$

Definition 7 (Codeword). *An $n-$tuple $c = (c_0, c_1, ..., c_{n-1}) \in \mathbb{F}_{2^n}$ is a codeword if the polynomial $c(x) = \sum c_i x^i$ has $\alpha, \alpha^2, ..., \alpha^{2t}$ as its roots.*

Definition 8 (Dual Code). *Given a linear code $C \subset \mathbb{F}_q^n$ of length n, the dual code of C (denoted by $C^\perp$) is defined to be the set of those vectors in $\mathbb{F}_q^n$ which are orthogonal*[5] *to every codeword of C,* i.e.*:*

$$C^\perp = \{v \in \mathbb{F}_q^n | v \cdot c = 0, \forall c \in C\}.$$

As α^i is a root of $c(x)$ for $1 \leq i \leq 2t$, then $c(\alpha^i) = \sum c_i \alpha^{ij}$. This equality can be written as a matrix product and results in the next property:

Property 1. If $c = (c_0, c_1, ..., c_{n-1})$ is a codeword, then the parity-check matrix H of this code satisfies $c \cdot H^T = 0$, where:

[5] The scalar product of the two vectors is equal to 0.

$$H = \begin{pmatrix} 1 & \alpha & \alpha^2 & \dots & \alpha^{n-1} \\ 1 & \alpha^2 & (\alpha^2)^2 & \dots & (\alpha^2)^{n-1} \\ 1 & \alpha^3 & (\alpha^3)^2 & \dots & (\alpha^3)^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \alpha^{2t} & (\alpha^{2t})^2 & \dots & (\alpha^{2t})^{n-1} \end{pmatrix}.$$

If $c \cdot H^T = 0$, then $c(\alpha^i) = 0$.

Remark 1. A parity check matrix of a linear block code is a generator matrix of the dual code. Therefore, c must be a codeword of the $t-$error correcting BCH code. If each entry of H is replaced by its corresponding $m-$tuple over $\mathbb{F}_2$ arranged in column form, we obtain a binary *parity-check matrix* for the code.

Definition 9 (Systematic Encoding). *In systematic encoding, information and check bits are concatenated to form the message transmitted over the noisy channel.*

The speed-up described in this paper applies to systematic BCH coding only.

Consider an (n, k) BCH code. Let $m(x)$ be the information polynomial to be coded and $m'x^{n-k} = m(x)$.

We can write $m'(x)$ as $m(x)g(x) + b(x)$.

The message $m(x)$ is coded as $c(x) = m'(x) - b(x)$[6].

BCH Decoding *Syndrome decoding* is a decoding process for linear codes using the parity-check matrix.

Definition 10 (Syndrome). *Let c be the emitted word and r the received one. We call the quantity $S(r) = r \cdot H^T$ the syndrome of r .*

If $r \cdot H^T = 0$ then no errors occurred, with overwhelming probability. If $r \cdot H^T \neq 0$, at least one error occurred and $r = c + e$, where e is an error vector. Note that $S(r) = S(e)$. The syndrome circuit consists of $2t$ components in $\mathbb{F}_{2^m}$. To correct t errors, the syndrome has to be a $2t$-tuple of the form $S = (S_1, S_2, \cdots, S_{2t})$.

[6] where $b(x)$ is the remainder of the division of $c(x)$ by $g(x)$

Syndrome In the polynomial setting, S_i is obtained by evaluating r at the roots of $g(x)$.

Indeed, letting $r(x) = c(x) + e(x)$, we have

$$S_i = r(\alpha^j) = c(\alpha^j) + e(\alpha^j) = e(\alpha^j) = \sum_{k=0}^{\nu-1} e_k \alpha^{ik}, \text{ for } i \leq 1 \leq 2t.$$

Suppose that r has ν errors denoted e_{j_i}. Then

$$S_i = \sum_{j=1}^{\nu} e_{j_i} (\alpha^i)^{j_\ell} = \sum_{j=1}^{\nu} e_{j_i} (\alpha^{j_\ell})^i.$$

Error Location Let $X_\ell = \alpha^{j_\ell}$. Then, for binary BCH codes, we have $S_i = \sum_{j=1}^{\nu} X_\ell^i$. The X_ℓs are called *error locators* and the *error locator polynomial* is defined as:

$$\Lambda(x) = \prod_{\ell=1}^{\nu} (1 - X_\ell) = 1 + \Lambda_1 x + ... + \Lambda_\nu x^\nu.$$

Note that the roots of $\Lambda(x)$ point out errors' places and the number of errors ν is unknown.

There are several ways to compute $\Lambda(x)$, *e.g.* Peterson's algorithm [7] or Berlekamp-Massey algorithm [8]. Chien's search method [9] is applied to determine the roots of $\Lambda(x)$.

Peterson's Algorithm Peterson's Algorithm 3 solves a set of linear equations to find the value of the coefficients $\sigma_1, \sigma_2, \ldots \sigma_t$.

$$\Lambda(x) = \prod_{\ell=1}^{\nu} (1 + \alpha^{jl}) = 1 + \sigma_1 x + \sigma_2 x^2 + \cdots + \sigma_t x^t$$

At the beginning of Algorithm 3, the number of errors is undefined. Hence the maximum number of errors to resolve the linear equations generated by the matrix S is assumed. Let this number be $i = \nu = t$.

Algorithm 3: Peterson's Algorithm

1 Initialization $\nu \leftarrow t$

2 Compute the determinant of S

$$\det(S) \leftarrow \det \begin{pmatrix} S_1 & S_2 & \cdots & S_t \\ S_2 & S_3 & \cdots & S_{t+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_t & S_{t+1} & \cdots & S_{2t-1} \end{pmatrix}$$

3 Find the correct value of ν

$$\begin{cases} \det(S) \neq 0 \quad \longrightarrow \quad \text{go to step 4} \\ \det(S) = 0 \quad \longrightarrow \quad \begin{cases} \textbf{if } \nu = 0 \textbf{ then} \\ \quad \text{The error locator polynomial is empty} \\ \quad \text{stop} \\ \textbf{else} \\ \quad \nu \longleftarrow \nu - 1, \text{ and then repeat step 2} \\ \textbf{end if} \end{cases} \end{cases}$$

4 Invert S and compute $\Lambda(x)$

$$\begin{bmatrix} \sigma_\nu \\ \sigma_{\nu-1} \\ \vdots \\ \sigma_1 \end{bmatrix} = S^{-1} \times \begin{bmatrix} -S_{\nu+1} \\ -S_{\nu+2} \\ \vdots \\ -S_{2\nu} \end{bmatrix}$$

Chien's Error Search Chien search finds the roots of $\Lambda(x)$ by brute force [4, 9]. The algorithm evaluates $\Lambda(\alpha^i)$ for $i = 1, 2, \ldots, 2^m - 1$. Whenever the result is zero, the algorithm assumes that an error occurred, thus the position of that error is located. A way to reduce the complexity of Chien search circuits stems from Equation 1 for $\Lambda(\alpha^{i+1})$.

$$\begin{aligned} \Lambda(\alpha^i) &= 1 + \sigma_1\, \alpha^i + \sigma_2\, (\alpha^i)^2 + \cdots + \sigma_t\, (\alpha^i)^t \\ &= 1 + \sigma_1\, \alpha^i + \sigma_2\, \alpha^{2i} + \cdots + \sigma_t\, \alpha^{it} \\ \Lambda(\alpha^{i+1}) &= 1 + \sigma_1\, \alpha^{i+1} + \sigma_2\, (\alpha^{i+1})^2 + \cdots + \sigma_t\, (\alpha^{i+1})^t \\ &= 1 + \alpha\, (\sigma_1\, \alpha^i) + \alpha^2\, (\sigma_2\, \alpha^{2i}) + \cdots + \alpha^t\, (\sigma_t\, \alpha^{it}) \end{aligned} \tag{1}$$

4.2 Implementation and Results

To evaluate the efficiency of Barrett's modular division in hardware, the BCH$(15, 7, 2)$ was chosen as a case study code. Five BCH encoder versions were designed and synthesized. Results are presented in detail in the coming sections.

4.2.1 Standard Architecture The `BCH-standard` architecture consists of applying the modular division using shifts and XORs. Initially, to determine the degree of the input polynomials, each bit[7] of the dividend and of the divisor are checked until the first bit one is found. Then, the two polynomials are left-aligned (*i.e.*, the two most significant ones are aligned) and XORed. The resulting polynomial is right shifted and again left-aligned with the dividend and XORed. This process is repeated until the dividend and the resulting polynomial are right-aligned. The final resulting polynomial represents the remainder of the division. Algorithm 4 provides the pseudocode for the standard architecture.

Algorithm 4: Standard modular division (`BCH-standard`)

Input: P, Q
Output: `remainder` $= Q \bmod P$

```
1  diff_degree ← deg(Q) − deg(P)
2  shift_counter ← diff_degree + 1
3  shift_divisor ← P ≪ diff_degree
4  remainder ← Q
5  while shift_counter ≠ 0 do
6      if remainder[p_degree + shift_counter − 1] = 1 then
7          shift_counter ← shift_counter − 1
           shift_divisor ← shift_divisor ≫ 1
8      end
9  end
10 return remainder
```

4.2.2 LFSR and Improved LFSR Architectures The `BCH-LFSR` design is composed of a control unit and a Linear-Feedback Shift Register (LFSR) submodule. The LFSR submodule receives the input data serially and shifts it to the internal registers, controlled by the enable signal. The LFSR's size (the number of parallel flip-flops) is defined by the BCH parameters n and k, *i.e.*, size(LFSR) = $n-k$, and the LFSR registers are called d_i, enumerated from 0 to $n-k-1$. The feedback value is defined by the XOR of the last LFSR register (d_{nk-1}) and the input data. The feedback connections are defined by the generator polynomial $g(x)$. In the case of BCH$(15, 7, 2)$, $g(x) = x^8 + x^7 + x^6 + x^4 + 1$, therefore the input of registers

[7] Considered in big endian order.

d_0, d_4, d_6 and d_7 are XORed with the feedback value. As shown in Fig. 1, the multiplexer that selects the bits to compose the final codeword is controlled by the counter. The LFSR is shifted k times with the feedback connections enabled. After that, the LFSR state contains the result of the modular division, therefore the bits can be serially shifted out from the LFSR register.

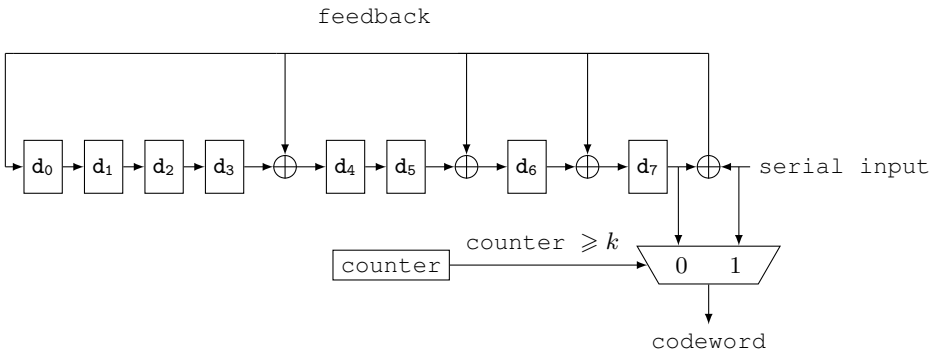

Fig. 1. Standard LFSR architecture block diagram. (Design `BCH-LFSR`)

To calculate the correct codeword, the LFSR must shift the input data during k clock cycles. After that, the output is serially composed by $n-k$ extra shifts. This means that the LFSR implementation's total latency is n clock cycles. Nevertheless, it is possible to save $n-k-1$ clock cycles by outputting the LFSR in parallel from the sub-module to the control unit after k iterations, while during the k first cycles the input data is shifted to the output register, as we perform systematic BCH encoding. This decreases the total latency to $k+1$ clock cycles. This method was applied to the `BCH-LFSR-improved` design depicted in Fig. 2.

4.2.3 Barrett Architecture (regular and pipelined) The LFSR submodule can be replaced by the Barrett submodule to evaluate its performance. Two Barrett implementations were designed: the first computes all the Barrett steps in one clock cycle, while the second approach, a pipelined block, works with the idea that Barrett operations can be broken down into up to $k+1$ pipeline stages, to match the LFSR's latency. The fact that Barrett operations can be easily pipelined drastically increases the final throughput, while both LFSR implementations do not allow for pipelining.

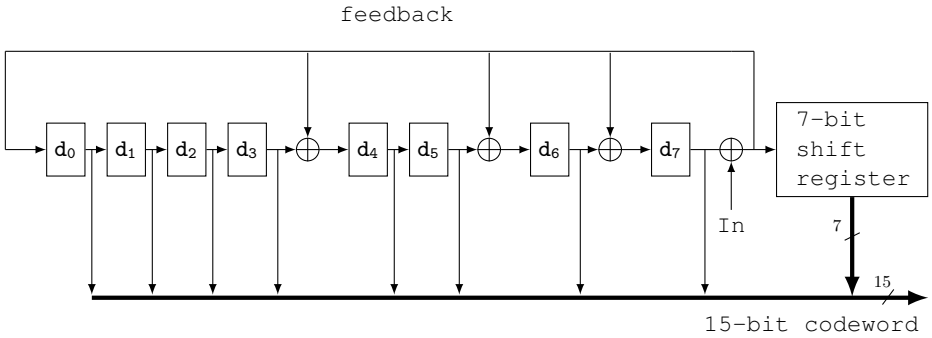

Fig. 2. Improved LFSR architecture block diagram. `In` denotes the module's serial input. (Design `BCH-LFSR-improved`)

In the Barrett submodule, the constants y_0, L, and K are pre-computed and are defined as parameters of the block. Since the Barrett parameter P is defined as the generator polynomial, P does not need to be defined as an input, which saves registers. As previously stated, Barrett operations were cut down to k iterations (in our example, $k = 7$). The first register in the pipeline stores the result of $Q \gg y_0$. The multiplication by K is the most costly operation, taking 5 clock cycles to complete. Each cycle operates on 3 bits, shifting and XORing at each one bit of K, according to the rules of multiplication. The last operation simply computes the intermediate result from the multiplication left-shifted by $L - y_0$.

4.2.4 Performance The gate equivalent (GE) metric is the ratio between the total cell area of a design and the size of the smallest NAND-2 cell of the digital library. This metric allows comparing circuit areas while abstracting away technology node sizes. *FreePDK45* (an open source 45nm Process Design Kit [12]) was used as a digital library to map the design into logic cells. Synthesis results were generated by Cadence Encounter RTL Compiler RC13.12 (v13.10-s021_1). `BCH-Barrett` presented an area comparable to the smallest design (`BCH-LFSR`). Although `BCH-Barrett` does not reach the maximum clock frequency, Table 1 shows that it actually reaches the best throughput among the non-pipelined designs, around 2.08Gbps. The `BCH-Barrett-pipelined` achieves the best throughput, but it reresents the biggest area and the more power consuming core. This is mainly due to the parallelizable nature of Barrett's operations, allowing the design to be easily pipelined and therefore further speed-up. The extra register barriers introduced in

`BCH-Barrett-pipelined` forces the design to present bigger area and a higher switching activity, which increases power consumption.

Design	Gate Instances	Gate Equivalent	Max Frequency (MHz)	Throughput (Mbps)	Power (μW)
`BCH-Standard`	310	447	741	690	978
`BCH-LFSR`	155	223	1043	972	920
`BCH-LFSR-improved`	160	236	1043	2080	952
`BCH-Barrett`	194	260	655	9150	512
`BCH-Barrett-pipelined`	426	591	995	13900	2208

Table 1. Synthesis results of the four BCH designs.

References

1. P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Proceedings on Advances in Cryptology - CRYPTO'86*, pages 311–323. Springer-Verlag, 1987.
2. R. C. Bose and D. K. Ray-Chaudhuri. On a Class of Error Correcting Binary Group Codes. *Information and Control*, 3(1):68–79, 1960.
3. A. Bosselaers, R. Govaerts, and Vandewalle J. Comparison of Three Modular Reduction Functions. In *Advances in Cryptology - EUROCRYPT93*, volume 773 of *Lecture Notes in Computer Science*, pages 175–186. Springer, 1994.
4. R. Chien. Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes. *IEEE Trans. Inf. Theor.*, 10(4):357–363, 2006.
5. G. Côté, B. Erol, M. Gallant, and F. Kossentini. H.263+: Video Coding at Low Bit Rates. *IEEE Transactions on Circuits and Systems for Video Technology*, 8:849–866, 1998.
6. D. Gorenstein and N. Zierler. A Class of Cyclic Linear Error-Correcting Codes in p^m Symbols. *J. Soc. Ind. Appl. Math.*, 9:207–214, 1961.
7. G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer, 2nd edition, 2007.
8. A. Hocquenghem. Codes correcteurs d'erreurs. *Chiffres*, 2:147–158, 1959.
9. F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. Northholland Publishing Company, 2nd edition, 1978.
10. D. Naccache and H. M'silti. A New Modulo Computation Algorithm. *Recherche Operationnelle - Operations Research (RAIRO-OR*, 24(3):307–313, 1990.
11. R. Steele. *Mobile Radio Communications*. IEEE Press, 1994.
12. J. E. Stine, I. D. Castellanos, M. Wood, J. Henson, F. Love, W. Rhett Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. FreePDK: An Open-Source Variation-Aware Design Kit. In *IEEE International Conference on Microelectronic Systems Education, MSE '07*, pages 173–174, 2007.
13. R. Tolimieri, M. An, and C. Lu. *Mathematics of Multidimensional Fourier Transform Algorithms*. Springer, 1993.

A Polynomial Barrett Complexity

We decompose the algorithm's analysis into steps and determine at each step the *cost* and the *size* of the result. Size is measured in the number of terms. In all the following we assume that polynomial multiplication is performed using traditional cross product. Faster (*e.g.* ν-dimensional FFT [13]) polynomial multiplication strategies may grandly improve the following complexities for asymptotically increasing $\boldsymbol{L}$ and ν.

Given our focus on on-line operations we do not count the effort required to compute K (that we assume given). We also do not account for the partial multiplication trick for the sake of clarity and conciseness.

Let $\boldsymbol{\omega} \in \mathbb{Z}^{\nu}$, in this appendix we denote by $||\boldsymbol{\omega}||$ the quantity

$$||\boldsymbol{\omega}|| = \prod_{j=1}^{\nu} \omega_j \in \mathbb{Z}.$$

1. $Q \gg \boldsymbol{y_0}$.

   1.1. **Cost**: $\mathrm{lm}(Q)$ is at most $\langle L, ..., L \rangle$ hence Q has at most L^{ν} monomials. Shifting discards all monomials having exponent vectors $\boldsymbol{\omega}$ for which $\exists j$ such that $\omega_j < y_{j,0}$. The number of such discarded monomials is $\mathrm{O}(||\boldsymbol{y_0}||)$, hence the overall complexity of this step is:

   $$\mathrm{cost}_1 = \mathrm{O}((L^{\nu} - ||\boldsymbol{y_0}||)\nu) = \mathrm{O}((L^{\nu} - \prod_{j=1}^{\nu} y_{j,0})\nu).$$

   1.2. **Size**: The number of monomials remaining after the shift is

   $$\mathrm{size}_1 = \mathrm{O}(L^{\nu} - ||\boldsymbol{y_0}||) = \mathrm{O}(L^{\nu} - \prod_{j=1}^{\nu} y_{j,0}).$$

2. $K(Q \gg \boldsymbol{y_0})$.

   Because K is the result of the division of $h(L) = \prod_{j=1}^{\nu} x_j^L$ by P, the leading term of K has an exponent vector equal to $\boldsymbol{L} - \boldsymbol{y_0}$. This means that K's second biggest term can be $x_1^{L-y_{1,0}} \prod_{j=2}^{\nu} x_j^L$. Hence, the size of K is

   $$\mathrm{size}_K = \mathrm{O}((L - y_{1,0})L^{\nu-1}).$$

2.1. **Cost**: The cost of computing $K(Q \gg \boldsymbol{y_0})$ is

$$\text{cost}_2 = \text{O}(\nu \times \text{size}_1 \times \text{size}_K).$$

2.2. **Size**: The size of $K(Q \gg \boldsymbol{y_0})$ is determined by $\text{lm}(K(Q \gg \boldsymbol{y_0})) = \text{lm}(K) \times \text{lm}(Q \gg \boldsymbol{y_0})$ which has the exponent vector $\boldsymbol{u} = (\boldsymbol{L} - \boldsymbol{y_0}) + \langle L - y_{1,0}, L, ..., L \rangle$.

$$\begin{aligned}\text{size}_2 = \text{O}(||\boldsymbol{u}||) &= \text{O}(2(L - y_{1,0}) \prod_{j=2}^{\nu} (2L - y_{j,0})) \\ &= \text{O}((L - y_{1,0}) \prod_{j=2}^{\nu} (2L - y_{j,0})).\end{aligned}$$

3. $B = (K(Q \gg \boldsymbol{y_0})) \gg (\boldsymbol{L} - \boldsymbol{y_0})$

3.1. **Cost**: The number of discarded monomials is $\text{O}(||\boldsymbol{L} - \boldsymbol{y_0}||)$, hence the cost of this step is

$$\text{cost}_3 = \text{O}((2(L - y_{1,0}) \prod_{j=2}^{\nu} (2L - y_{j,0}) - \prod_{j=1}^{\nu} (L - y_{j,0}))\nu).$$

3.2. **Size**: The leading monomial of B has the exponent vector $\boldsymbol{u} - \boldsymbol{L} - \boldsymbol{y_0}$ which is equal to $\langle L - y_{1,0}, L, ..., L \rangle$. We thus have $\text{size}_B = \text{size}_K$.

4. BP

The cost of this step is

$$\text{cost}_4 = \text{O}(\nu \times \text{size}_B \times \text{size}_P) = \text{O}(\nu \times \text{size}_B \times ||\boldsymbol{y_0}||).$$

5. Final subtraction $Q - BP$

The cost of polynomial subtraction is negligible with respect to cost_4.

6. **Overall complexity**

The algorithm's overall complexity is hence

$$\max(\text{cost}_1, \text{cost}_2, \text{cost}_3, \text{cost}_4) = \text{cost}_2.$$

A Polynomial Barrett: Scheme Code

$p_1(x) = \sum_{i=0}^{7}(10+i)x^i$ and $p_2(x) = x^3 + x^2 + 110$

```
(define p1 '((7 17) (6 16) (5 15) (4 14) (3 13) (2 12) (1 11) (0
10)))

(define p2 '((3 1) (2 1) (0 110)))
```

;shifting a polynomial to the right

```
(define shift (lambda (l q)

(if (or (null? l) (< (caar l) q)) '() (cons (cons (- (caar l) q)
(cdar l))

(shift (cdr l) q)))))
```

;adding polynomials

```
(define add (lambda (p q)

(degre (if (>= (caar p) (caar q)) (cons p (list q)) (add q
p)))))
```

;multiplying a term by a polynomial, without monomials $\prec x^{lim}$

```
(define txp (lambda (terme p lim)

(if (or (null? p) (> lim (+ (car terme) (caar p)))) '() (cons
(cons (+ (car terme)

(caar p)) (list (* (cadr terme) (cadar p)))) (txp terme (cdr p)
lim)))))
```

;multiplying a polynomial by a polynomial, without monomials $\prec x^{lim}$

```
(define mul (lambda (p1 p2 lim)

(if p1 (cons (txp (car p1) p2 lim) (mul (cdr p1) p2 lim))
'())))
```

;management of the exponents

```
(define sort (lambda (p n)

(if p (+ ((lambda(x) (if x (cadr x) 0)) (assoc n (car p))) (sort
(cdr p) n)) 0)))

(define order (lambda (p n)

(if(> 0 n) '() (let ((factor (sort p n))) (if (not (zero?
factor))
```

```
(cons (cons n (list factor)) (order p (-n 1))) (order p (-n
1)))))))
```

```
(define degre (lambda(p) (order p ((lambda(x)(if x x -1)) (caaar
p)))))
```

;Euclidean division

```
(define divide (lambda (q p r)
```

```
(if (and p (<= (caar p) (caar q))) (let ((tampon (cons (- (caar
q)(caar p))
```

```
(list (/ (cadar q) (cadar p)))))) (divide (add (map (lambda(x)
(cons (car x)
```

```
(list (-cadr x)))))(txp tampon p -1)) q) p (cons tampon r)))
(reverse r)))
```

```
(define division (lambda (q p) (divide q p '())))
```

;Barrett(k, L, last_P and Y representing K, L, P and ***y****)*

```
(define k)
```

```
(define y)
```

```
(define L 8)
```

```
(define last)
```

```
(define barrett (lambda (q p)
```

```
(if (eq ? last p) (letrec ((g (caar q)) (h (- (+ g 1) y)))
(shift (degre (mul
```

```
(shift k (-L g 1)) (shift q y) h)) h)) (begin (set! k (division
(list (cons L '(1) )) p)) (set! y (caar (set! last p))) (barrett
q p)))))
```