



UNIVERSITÀ DEGLI STUDI DI PISA
DIPARTIMENTO DI MATEMATICA
CORSO DI LAUREA MAGISTRALE IN MATEMATICA

Algebraic Techniques for Circuit Verification

Tesi di Laurea Magistrale

17 ottobre 2014

Candidato
Oscar Papini

Relatori
Prof.ssa **Patrizia Gianni**
Dott. **Barry Trager**

ANNO ACCADEMICO 2013/2014

Contents

Introduction	xi
1 Reed-Solomon Encoding and Decoding	1
1.1 How to Encode a Word	1
1.2 How to Decode a Word	3
1.3 The Combined RS Encoder/Syndrome Generator	5
2 The Commutative Algebra Approach	9
2.1 The Nullstellensatz for Finite Fields	9
2.2 Verification of Combinational Circuits	11
2.3 Dealing with Time: the Polynomial Method	14
2.4 Dealing with Time: the Module Method	16
2.5 A First Result: Arbitrariness of the Coefficient Field	19
3 The Difference Equations Approach	21
3.1 Difference Equations	21
3.2 The Z-Transform	23
3.3 Transfer Functions	25
3.4 Circuit Equivalence	26
4 Correctness of the Combined RS Encoder/Syndrome Generator	27
4.1 Particular Instances of the Circuit	27
4.1.1 2-Error-Correcting RS over \mathbb{F}_8	27
4.1.2 2-Error-Correcting RS over \mathbb{F}_8 with a Bug	31
4.1.3 4-Error-Correcting RS over \mathbb{F}_{32}	32
4.2 General Case	35
4.2.1 Applying the Z-Transform	37
4.2.2 Equivalence to the Standard Encoder	39
Conclusions and Further Developments	47

A Error-Correcting Codes	49
A.1 Generality of Error-Correcting Codes	49
A.2 Some Examples of Error-Correcting Codes	50
A.2.1 Linear Codes	50
A.2.2 Cyclic Codes	51
A.2.3 BCH Codes	52
A.3 Reed-Solomon Codes	54
B Logic Circuits	57
B.1 Combinational Circuits	57
B.2 Sequential Circuits	58
Bibliography	59

List of Figures

1.1	A standard RS encoder	3
1.2	Computation of the syndrome.	4
1.3	A set of circuit units for syndrome computation.	4
1.4	The modified RS encoder with a linear feedback loop.	5
1.5	The final version of the RS encoder.	6
1.6	The combined encoder/syndrome generator.	6
2.1	A two-bit multiplier over \mathbb{F}_4	12
2.2	A recursive multiplier over \mathbb{F}_q	15
2.3	A recursive adder over \mathbb{F}_q	18
3.1	The same of Figure 1.2.	25
4.1	The encoder of Figure 1.5 with $t = 2$	28
4.2	A bug in the circuit of Figure 4.1.	32
4.3	The encoder with $t = 4$	33
4.4	The same of Figure 1.5.	36
B.1	A combinational circuit that computes $(a \wedge b) \vee (\neg c)$	58
B.2	A typical memory register.	58

List of Tables

2.1	Translation between Boolean expression and ring operation in a Boolean algebra.	12
2.2	Polynomial translation of the gates of Figure 2.1.	13
3.1	Some common sequences with their Z-transforms.	24
4.1	List of the polynomials associated to the circuit in Figure 4.1. . . .	29

List of Symbols

Spaces and Algebraic Structures

\mathbb{K}, \mathbb{F}_q	Field, finite field with q elements
$\overline{\mathbb{K}}$	Algebraic closure of the field \mathbb{K}
$\mathbb{K}^{\mathbb{N}}$	Set of sequences $(a_n)_{n \in \mathbb{N}}$ with $a_n \in \mathbb{K}$ for all $n \in \mathbb{N}$
$\mathbb{K}[[Z]]$	Ring of the formal power series in Z over \mathbb{K}
$\mathcal{M}_{m \times n}(\mathbb{K})$	Space of $m \times n$ matrices with coefficients in \mathbb{K}
(G)	Ideal generated by the set G



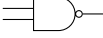


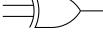
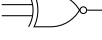
Notation of Special Functions

$\#(A)$	Cardinality of A
$\text{char}(\mathbb{K})$	Characteristic of \mathbb{K}
$\lfloor x \rfloor$	Floor function
$\mathcal{V}_{\mathbb{L}}(I)$	Affine variety of the ideal $I \subseteq \mathbb{K}[\mathbf{X}]$ over $\mathbb{L} \supseteq \mathbb{K}$; $\mathcal{V}(I) := \mathcal{V}_{\overline{\mathbb{K}}}(I)$
\sqrt{I}	Radical of the ideal I
$\mathcal{J}(V)$	Vanishing ideal of the set V
$\text{lc}(f)$	Leading coefficient of f
$\text{lm}(f)$	Leading monomial of f
$\text{lt}(f)$	Leading term of f ; $\text{lt}(f) = \text{lc}(f) \cdot \text{lm}(f)$
$S(f, g)$	S-polynomial between f and g
A^T	Transpose of the matrix A

Logic Operators

$\neg a$	NOT a
$a \wedge b$	a AND b
$a \vee b$	a OR b
$a \dot{\vee} b$	a XOR b

Logic Gates

	NOT gate
	AND gate
	NAND gate
	OR gate
	NOR gate
	XOR gate
	XNOR gate

Notational Remark. In this work, **boldface** symbols (such as \mathbf{a}) represent multi-dimensional objects, like vectors or n -tuples, whereas roman symbols (such as a) usually denote one-dimensional values. Variables or unknown quantities are written with lower case letters, while upper case is used for indeterminates in polynomial rings.

Introduction

A circuit is a set of (logic or arithmetic) gates connected by wires. Its purpose is to implement a function in a way that can be easily realized in the physical world (by means of electronic components like transistors and capacitors).

Suppose that we have a specific function in mind and we want to build a circuit which is able to compute it. We ask a logic designer who gives us the description of the logic gates and their connections. How can we be sure that the circuit does actually do what we want it to do? That is the verification problem, that is to say, the problem of finding a formal proof of the correctness of a circuit.

In this thesis we want to describe some of the algebraic techniques used to solve the verification problem. In particular, we will focus on two main types of methods:

- the *Commutative Algebra* approach, which reduces the problem to the well-known Ideal Membership Problem;
- the *Difference Equations* approach, which obtains a system of difference equations from the circuit, thus allowing a systematic study from that point of view.

Our starting point is a concrete circuit: a combined Reed-Solomon encoder/syndrome generator proposed by Fettweis and Hassner in [2]. An encoder is a device capable of converting a message into a codeword. Coding theory is a branch of mathematics which develops particular algebraic structures (the codes) which are suitable for transmitting or storing information, because they are able to detect and/or correct possible errors.

In the first chapter we describe briefly how the encoding and the decoding of an RS code work, as well as the circuits to perform those operations. We also introduce the combined encoder/syndrome generator of which we want to prove the correctness.

The second chapter deals with the verification techniques based on commutative algebra. In particular, we follow an article by Lv, Kalla and Enescu [7] which uses the Nullstellensatz for Finite Fields to reduce the verification of a circuit to

an Ideal Membership Problem. They developed this method for combinational circuits only; we generalize it to include also sequential ones. Moreover, we show another method, based on the vector space structures in play, which is less expensive but works only with linear circuits (whose output depends linearly on the input).

The difference equations are introduced in the third chapter. After a brief description of them, we explain how to translate a circuit into a system of difference equations and illustrate some of the main tools used to study these equations.

In the fourth chapter we employ all these techniques to verify the encoder proposed by Fettweis and Hassner. We begin with particular instances of the circuit, for which we may use the commutative algebra approach. Then we will prove the correctness of the circuit structure, translating it into a system of difference equations, which are more suitable for this kind of proof. We will actually prove that a generalized version of the circuit works.

We conclude hinting at possible further developments of these topics, which may take different paths: we may look for some improvements of the efficiency of the methods shown, or we could analyse the combined circuit more in depth in order to have a better understanding of its functioning, in case that the ideas behind it could be applied in other situations.

Chapter 1

Reed-Solomon Encoding and Decoding

In this chapter we are going to describe how a Reed-Solomon (RS) code works. In particular, we will see how to encode a word, turning it into a codeword, and how to perform the actual error correction.

RS codes are usually implemented in technologies which rely on fast information retrieval, such as CD/DVD units and digital television broadcasting. Because of this, researchers and engineers have focused their attention on a *hardware* implementation rather than a software one. We will present the classical circuits used in encoding/decoding of a RS code, as well as one combined encoder/syndrome calculator presented by Fettweis and Hassner in [2], which reduces the global amount of hardware complexity. A proof of the correctness of this circuit is the main task of this thesis work.

We will assume some basic knowledge of error-correcting codes and logic circuits. A brief summary of these topics can be found respectively in Appendix A and B.

Notational Remark. The expression *n-word* will denote (not so surprisingly) a word of length n .

1.1 How to Encode a Word

Let \mathbb{F}_q be a finite field with characteristic 2 and let n , k and t be integers such that $n = q - 1$ and $k = n - 2t$. Let C be an RS code defined on \mathbb{F}_q of length n , with an error-correction capacity of t symbols. Thus the encoder takes a word $\mathbf{w} \in (\mathbb{F}_q)^k$ and gives a codeword $\mathbf{c} \in C \subseteq (\mathbb{F}_q)^n$. We suppose that C is generated

by the polynomial

$$g(X) := \prod_{i=1}^{2t} (X - \alpha^i)$$

where α is a fixed primitive element of \mathbb{F}_q .

The Mattson-Solomon polynomial (see Appendix A) defines a method to encode a k -word into an n -codeword, but unfortunately this method isn't systematic.

Definition 1.1. An encoding method is *systematic* if the n -codeword is obtained by concatenating the k -word to be encoded with $n - k$ parity-check symbols.

Obviously a systematic method is preferable to a non-systematic one, because it is possible to retrieve the message by truncating the $n - k$ parity-check symbols after the error-correction step, thus avoiding a decoding passage which may be both time- and resources-costing.

In order to systematically encode a k -word $w(X) = w_0 + w_1X + \dots + w_{k-1}X^{k-1}$, we just need a well-known algorithm: the Euclidean division of polynomials. In fact, the $2t$ parity-check symbols are just the coefficient of the remainder $r(X) = r_0 + r_1X + \dots + r_{2t-1}X^{2t-1}$ obtained from dividing $X^{2t}w(X)$ by the generator polynomial $g(X)$, and the codeword is $c(X) = X^{2t}w(X) + r(X)$. To see that c belongs to the RS code, notice that $X^{2t}w(X) = g(X)h(X) + r(X)$ for some h and that $c(X) = g(X)h(X)$ because $\text{char}(\mathbb{F}_q) = 2$.

Example 1.1. Let us define a 2-error-correcting RS code over \mathbb{F}_8 , which we obtain as $\mathbb{F}_2[\alpha]$ with $\alpha^3 + \alpha + 1 = 0$. It is a code of length 7 with 3 message symbols and 4 parity-check digits, with generator polynomial

$$g(X) := (X - \alpha)(X - \alpha^2)(X - \alpha^3)(X - \alpha^4)$$

A word $(\beta_2, \beta_1, \beta_0)$ is seen as $m(X) := \beta_2X^2 + \beta_1X + \beta_0$ and its encoding is

$$m(X)X^4 + r(X) = \beta_2X^6 + \beta_1X^5 + \beta_0X^4 + \gamma_3X^3 + \gamma_2X^2 + \gamma_1X + \gamma_0,$$

where $r(X) := \gamma_3X^3 + \gamma_2X^2 + \gamma_1X + \gamma_0$ is the remainder of the Euclidean division of $m(X)X^4$ by $g(X)$. For example, the polynomial $X^2 + (\alpha + 1)X + (\alpha^2 + \alpha)$ is encoded as

$$X^6 + (\alpha + 1)X^5 + (\alpha^2 + \alpha)X^4 + (\alpha^2)X^2 + (\alpha^2 + 1)X + \alpha. \quad (1.1)$$

At hardware level, an RS systematic encoder is a division circuit, as shown in

Figure 1.1. In that figure, the symbols $\rightarrow \oplus \rightarrow$ and $\beta \circlearrowleft$ represent respectively

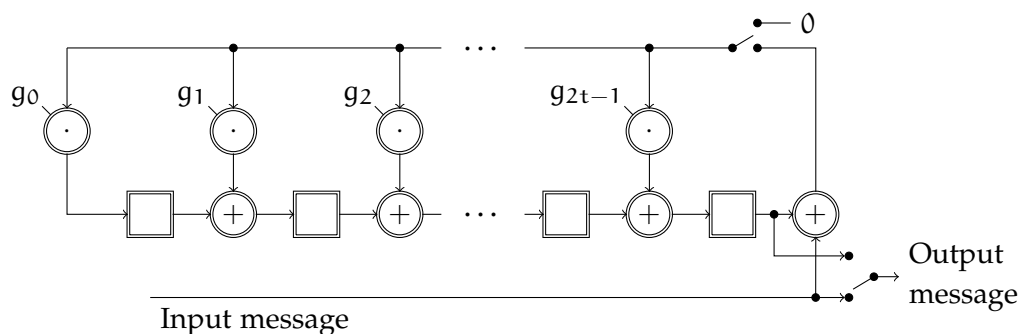


Figure 1.1: A standard RS encoder. The g_i 's are the coefficients of the generator polynomial.

an adder and a multiplier for a fixed β , which operate on elements of \mathbb{F}_q . The symbol $\boxed{}$ is a register capable of storing an element of \mathbb{F}_q , initialized to 0.

At first, and for k clock cycles, the two switches are both in the lower position: the first k symbols of the output word are just the input symbols. After k clock cycles, the registers contain the parity-check symbols, and the switches are activated in order to cut off the input wire and concatenate the parity-check symbols to the output.

1.2 How to Decode a Word

After having received an n -word $v(X)$, we have to decode it, i.e. we have to find the most likely codeword that has been transmitted. This requires a number of steps.

1. First of all, we have to compute the syndrome of the word. The $2t$ symbols of the syndrome are in fact the values $s_i := v(\alpha^i)$, for $i = 1, \dots, 2t$.
2. Then we solve the key equation and retrieve the error locator polynomial $\sigma(Z)$ and the error evaluator polynomial $\omega(Z)$.
3. We use $\sigma(Z)$ and $\omega(Z)$ to reconstruct the error pattern $e(X)$.
4. Finally, we compute the codeword $c(X) = v(X) + e(X)$.

For further information about steps 2. and 3., see Appendix A; we are going to focus our attention just on the first step. Computing syndromes is in fact quite simple; it is a polynomial evaluation, which means that it can be done

by Horner's Algorithm, that is to say, by iterated multiplication and addition, having written

$$v_0 + v_1X + \cdots + v_{n-1}X^{n-1} = v_0 + X(v_1 + X(v_2 + \cdots + X(v_{n-2} + v_{n-1}X) \dots)).$$

The circuit shown in Figure 1.2 perform this task—the coefficients of $v(X)$, starting

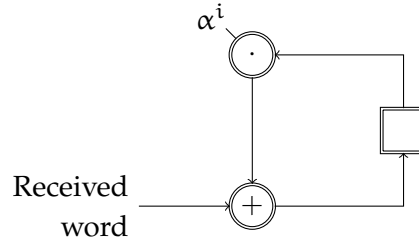


Figure 1.2: Computation of the syndrome $s_i = v(\alpha^i)$.

from the leading coefficient, are entered one by one at each clock cycle; when the last one is input, the register stores the value s_i . A set of $2t$ units of this circuit allows to compute the values s_i for $i = 1, \dots, 2t$.

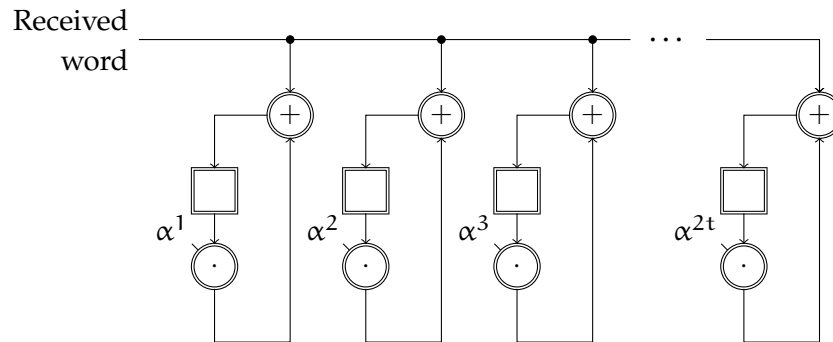


Figure 1.3: A set of circuit units for syndrome computation.

Example 1.2. Continuing from Example 1.1, let's compute the syndrome for the codeword (1.1), which we call $w(X)$. That is not difficult and the result is

$$(w(\alpha), w(\alpha^2), w(\alpha^3), w(\alpha^4)) = (0, 0, 0, 0).$$

Suppose that an error occurred and we received the word

$$\tilde{w}(X) = X^6 + (\alpha + 1)X^5 + (\alpha^2 + 1)X^4 + (\alpha^2)X^2 + (\alpha^2 + 1)X + \alpha$$

(where the only difference is the coefficient of X^4 : in $w(X)$ it was $\alpha^2 + \alpha$). The syndrome is

$$(\tilde{w}(\alpha), \tilde{w}(\alpha^2), \tilde{w}(\alpha^3), \tilde{w}(\alpha^4)) = (1, \alpha^2 + \alpha, \alpha, \alpha^2 + \alpha + 1).$$

From this information we solve the key equation, obtaining

$$\begin{aligned}\sigma(Z) &= 1 + (\alpha^2 + \alpha)Z; \\ \omega(Z) &= Z.\end{aligned}$$

Now, $\alpha^2 + \alpha = \alpha^4$, i.e. there have been a single error in the fourth coefficient; from Equation (A.1) we obtain the error pattern

$$e(X) = (\alpha + 1)X^4$$

and finally $\tilde{w}(X) + e(X) = w(X)$, successfully correcting the error.

1.3 The Combined RS Encoder/Syndrome Generator

One of the disadvantages of the circuit described in Section 1.1 is that it requires the coefficients of the generator polynomial. Suppose that a change of the code specification is needed, for example because we have to increase the error-correcting capacity. It suffices to add some powers of α , but this means more multiplier/adder loops in the circuit of Figure 1.1 and, above all, a new computation of the g_i 's. In other words, we have to rebuild the circuit.

In their article [2], Fettweis and Hassner notice that the encoder and the decoder are rarely active at the same time, thus it is possible to imagine a circuit in which hardware resources are shared between them. By manipulating the standard circuit of Figure 1.1, they managed to describe a combined encoder and syndrome generator for an RS code.

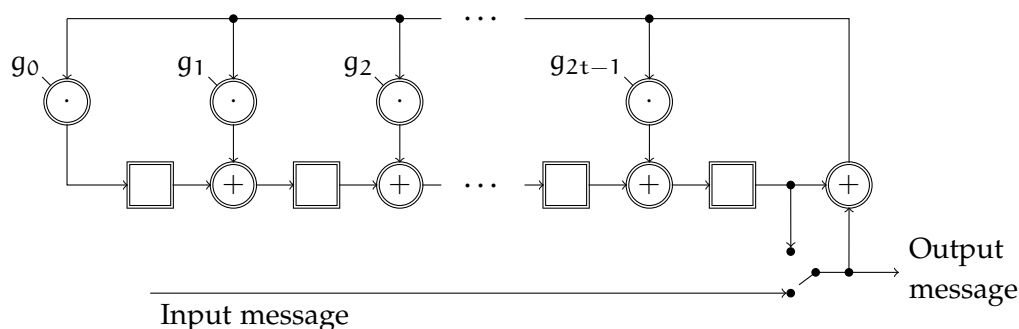


Figure 1.4: The modified RS encoder with a linear feedback loop.

Their main problem was the switch in the feedback loop, which is a non-linearity that prevented them from applying linear system transformation of the

loop itself. However, they realized that, since the characteristic of the coefficient field is 2, the same result could be achieved by assuring that the two inputs of the last adder in the chain were the same, therefore forcing the output of the adder to be 0, as shown in Figure 1.4. The feedback loop is now linear and can be studied by means of linear transformations.

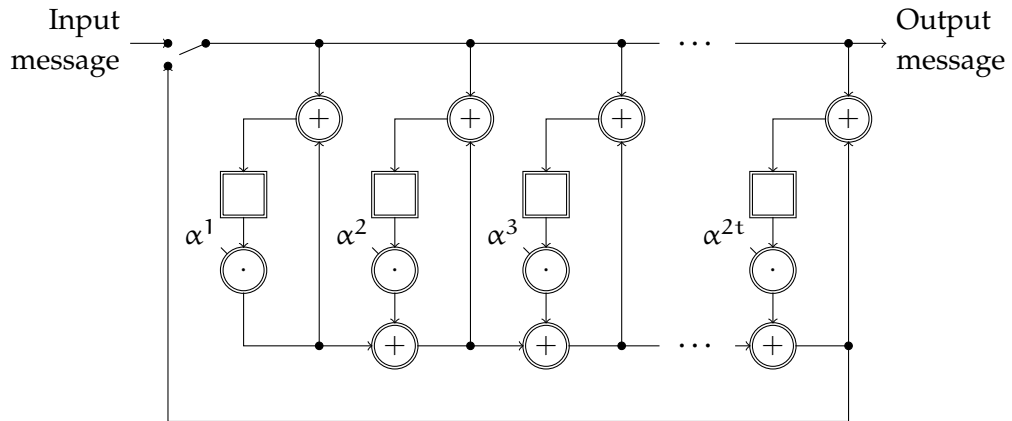


Figure 1.5: The final version of the RS encoder.

Figure 1.5 represents the resulting RS encoder. Notice that the multipliers use the roots of the generator polynomial (i.e. the α^i 's), and not the coefficients. In fact, there is a strong resemblance between this circuit and the one of Figure 1.3: the only difference is the lower adder chain.

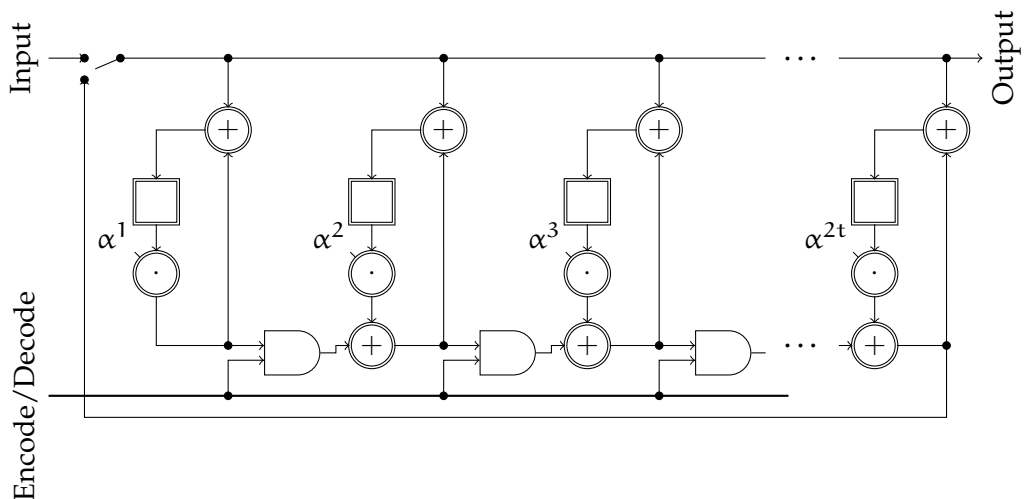


Figure 1.6: The combined encoder/syndrome generator.

The encoder of Figure 1.5 and the syndrome generator of Figure 1.3 can now be combined as shown in Figure 1.6. The “Encode/Decode” wire carries a stream of 1’s (for encoding) or 0’s (for decoding) of the suitable bandwidth, so that the AND gates let the information pass through the lower adder chain (in case of encoding) or break the chain (in case of decoding). When computing the syndrome, the switch is constantly in the upper position and the output end can be ignored, as the syndrome values will appear in the registers.

The particular structure of this encoder, which uses the α^i ’s instead of the g_i ’s, ensures an easy way to achieve different error-correcting capabilities with the same circuit: it suffices to cut off some of the loops in order to remove the relative α^i from the generator polynomial, effectively obtaining an encoder for the appropriate RS code.

In the next chapter we will develop a computational algebra method in order to perform the verification of sequential circuits. In particular, we will employ it to confirm that instances of the combined encoder/syndrome generator do actually work.

Chapter 2

The Commutative Algebra Approach

One of the most important results in commutative algebra is the so-called *Nullstellensatz*, which relates ideals in the polynomial ring with varieties in the affine space. In this chapter we are going to see how that theorem can be used for the verification of logic circuit.

In particular, we will introduce a verification technique developed by Lv, Kalla and Enescu in [7] for combinational circuits. Starting from there, we will define a generalization of that method in order to be able to deal with sequential circuits. We will call it the *polynomial method*.

We will also show another approach (the *module method*) that is more efficient than the polynomial one, but can be applied only to a specific class of circuits.

2.1 The Nullstellensatz for Finite Fields

Let us recall the basic facts of algebraic geometry. Let \mathbb{K} be a field and $\mathbb{K}[\mathbf{X}] := \mathbb{K}[X_1, \dots, X_n]$. For any subset $\mathcal{P} \subseteq \mathbb{K}[\mathbf{X}]$ and any field extension $\mathbb{L} \supseteq \mathbb{K}$, we denote by $\mathcal{V}_{\mathbb{L}}(\mathcal{P})$ the (*affine*) *variety* of \mathcal{P} over \mathbb{L} , that is

$$\mathcal{V}_{\mathbb{L}}(\mathcal{P}) := \{\mathbf{a} \in \mathbb{L}^n \mid \forall p \in \mathcal{P}, p(\mathbf{a}) = 0\}.$$

We will omit the subscript for $\mathbb{L} = \overline{\mathbb{K}}$. It is easy to show that $\mathcal{V}_{\mathbb{L}}(\mathcal{P}) = \mathcal{V}_{\mathbb{L}}(\langle \mathcal{P} \rangle)$, so we can consider just ideals instead of generic sets of polynomial.

For any subset $V \subset \mathbb{K}^n$, the set of polynomials that vanish on all the points of V forms an ideal, called the *vanishing ideal* of V :

$$\mathcal{I}(V) := \{f \in \mathbb{K}[\mathbf{X}] \mid \forall \mathbf{a} \in V, f(\mathbf{a}) = 0\}.$$

Theorem 2.1 (Weak Nullstellensatz). *Let \mathbb{K} be an algebraically closed field and $I \subseteq \mathbb{K}[\mathbf{X}]$ an ideal. Then $\mathcal{V}(I) = \emptyset$ if and only if $I = \mathbb{K}[\mathbf{X}]$.*

Theorem 2.2 (Strong Nullstellensatz). *Let \mathbb{K} be an algebraically closed field and $I \subseteq \mathbb{K}[\mathbf{X}]$ an ideal. Then $\mathcal{J}(\mathcal{V}(I)) = \sqrt{I}$.*

There is a little problem: we are going to use finite fields, which are never algebraically closed. For, suppose $\mathbb{K} = \{q_1, \dots, q_\ell\}$, the polynomial

$$(X - q_1) \cdot \dots \cdot (X - q_\ell) + 1$$

belongs to $\mathbb{K}[X]$ but has no root in \mathbb{K} . This prevents us from applying the Nullstellensatz directly.

What can we do? Actually, there is an elegant solution to our problem. All we have to do is to identify which part of the variety in $(\overline{\mathbb{F}_q})^n$ lies in fact in $(\mathbb{F}_q)^n$, and there is a special polynomial that encodes the elements of \mathbb{F}_q .

Proposition 2.3. *All and only the fixed points of the Frobenius automorphism*

$$\begin{array}{ccc} \varphi_q: \overline{\mathbb{F}_q} & \longrightarrow & \overline{\mathbb{F}_q} \\ & & x \longmapsto x^q \end{array}$$

are the elements of \mathbb{F}_q . In other words, the roots of the polynomial $X^q - X$ are the elements of \mathbb{F}_q .

As a consequence, we may hope that adding relations of the form $X_i^q - X_i$ (which we will call *field equations*) would help us solve the problem. In fact, that is what happens, as stated in the following theorems.

Theorem 2.4 (Weak Nullstellensatz over Finite Fields). *Let $I \subseteq \mathbb{F}_q[\mathbf{X}]$ be an ideal, and $I_0 := (X_1^q - X_1, \dots, X_n^q - X_n)$. Then $\mathcal{V}_{\mathbb{F}_q}(I) = \emptyset$ if and only if $(I, I_0) = \mathbb{F}_q[\mathbf{X}]$.*

Proof. Let $\overline{\mathbb{F}_q}$ denote the algebraic closure of \mathbb{F}_q and let $\mathcal{V}(I)$ be the variety of I over $\overline{\mathbb{F}_q}$. We have $\mathcal{V}_{\mathbb{F}_q}(I) = \mathcal{V}(I) \cap (\mathbb{F}_q)^n$. Now, $(\mathbb{F}_q)^n$ is a finite set of points, so it is $\mathcal{V}(I_0)$ for some I_0 . It follows that

$$\mathcal{V}_{\mathbb{F}_q}(I) = \mathcal{V}(I) \cap (\mathbb{F}_q)^n = \mathcal{V}(I) \cap \mathcal{V}(I_0) = \mathcal{V}(I, I_0). \quad (2.1)$$

We have only to prove that $I_0 = (X_1^q - X_1, \dots, X_n^q - X_n)$, but this is a straightforward consequence of Proposition 2.3. \square

Lemma 2.5. *For any ideal $I \subseteq \mathbb{F}_q[\mathbf{X}]$, the ideal (I, I_0) is radical.*

Proof. Since any ideal is contained in its radical, we only have to prove that

$$\sqrt{(I, I_0)} \subseteq (I, I_0).$$

Let $f \in \sqrt{(I, I_0)}$. This means there is an $s \in \mathbb{N}$ such that $f^s \in (I, I_0)$. Now let $[f]$ and $[I]$ be the classes of f and I in $\mathbb{F}_q[\mathbf{X}]/I_0$. We have $[f]^s \in [I]$, and it suffices to prove that $[f] \in [I]$.

Firstly we show that, for any $[g] \in \mathbb{F}_q[\mathbf{X}]/I_0$, $[g]^q = [g]$. This can be done by induction on the number of terms in g .

1. If $[g] = [c\mathbf{X}^\alpha]$ is a monomial, then $[g]^q = [c^q\mathbf{X}^{q\alpha}] = [c\mathbf{X}^\alpha] = [g]$.
2. Suppose $[g] = [h_1] + [h_2]$. By inductive hypothesis $[h_1]^q = [h_1]$ and $[h_2]^q = [h_2]$. Then

$$\begin{aligned} [g]^q &= ([h_1] + [h_2])^q = \sum_{i=0}^q \binom{q}{i} [h_1]^i [h_2]^{q-i} = \\ &= [h_1]^q + [h_2]^q = [h_1] + [h_2] = [g]. \end{aligned}$$

Since $[f]^q = [f]$, without loss of generality we may assume $s < q$. But now $[f]^s \in [I]$ and also

$$[f]^s [f]^{q-s} = [f]^q = [f] \in [I].$$

This ends the proof. □

Theorem 2.6 (Strong Nullstellensatz over Finite Fields). *Let $I \subseteq \mathbb{F}_q[\mathbf{X}]$ be an ideal. Then $\mathcal{J}(\mathcal{V}_{\mathbb{F}_q}(I)) = (I, I_0)$.*

Proof. By the Strong Nullstellensatz applied to the ideal (I, I_0) and Lemma 2.5, we have

$$\mathcal{J}(\mathcal{V}(I, I_0)) = (I, I_0).$$

The thesis follows by Relation (2.1). □

2.2 Verification of Combinational Circuits

In this section we will describe the verification technique developed by Lv, Kalla and Enescu in [7]. They use the results of the previous section to translate the verification problem in an ideal membership problem, which can be effectively solved using standard algebraic methods. They focus on *combinational* circuits only, where the output is a polynomial function of the input.

We begin with what we want the circuit to do. For example, we may want to build a multiplier over \mathbb{F}_4 , i.e. a circuit that takes $a, b \in \mathbb{F}_4$ and gives $ab \in \mathbb{F}_4$.

We encode the relation between input and output in a multivariate polynomial (*specification polynomial*) $f(Z, A_1, \dots, A_k)$, where Z and A_1, \dots, A_k are variables representing the output and the inputs respectively. In our example, we have $f(Z, A, B) = Z - AB$.

Then we have the *implementation* of the circuit, which is the information about the number and the type of logic gates needed, and their disposition in the circuit. The gates are modelled with the standard correspondence between Boolean algebras and Boolean rings, which can be found in Table 2.1. In addition

Boolean expression	Ring operation
$\neg a$	$1 + a$
$a \wedge b$	$a \cdot b$
$a \vee b$	$a \cdot b + a + b$
$a \underline{\vee} b$	$a + b$

Table 2.1: Translation between Boolean expression and ring operation in a Boolean algebra.

to these, we have to include any other relation between the variables appearing in the specification polynomial and the variables which arise from the description of the circuit.

An example of a multiplier over \mathbb{F}_4 is given in Figure 2.1.

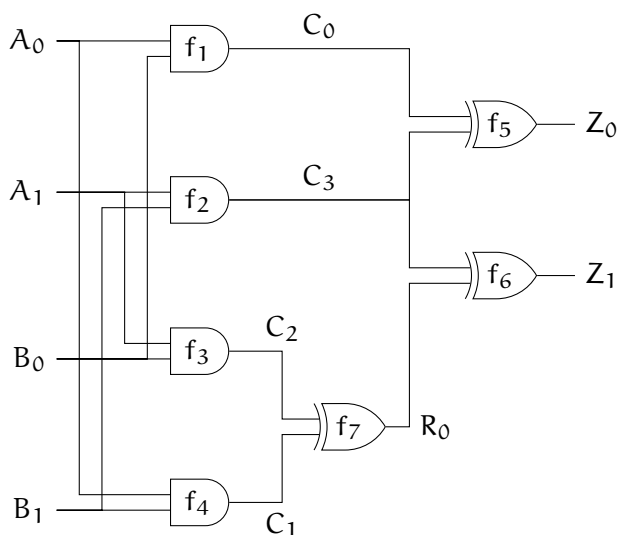


Figure 2.1: A two-bit multiplier over \mathbb{F}_4 .

All the variables in Figure 2.1 may assume a binary value. The result of the

polynomial translation is shown in Table 2.2.

Gate	Polynomial Expression
f_1	$C_0 - A_0B_0$
f_2	$C_3 - A_1B_1$
f_3	$C_2 - A_1B_0$
f_4	$C_1 - A_0B_1$
f_5	$Z_0 - (C_0 + C_3)$
f_6	$Z_1 - (C_3 + R_0)$
f_7	$R_0 - (C_2 + C_1)$

Table 2.2: Polynomial translation of the gates of Figure 2.1.

We also have to link the specification variables and the input/output of the circuit. In this example, we assume that \mathbb{F}_4 is given as $\mathbb{F}_2[\alpha]$ where $\alpha^2 + \alpha + 1 = 0$. The relations between Z, A, B and the variables appearing in the circuit are

$$f_8: A - (A_0 + A_1\alpha), \quad f_9: B - (B_0 + B_1\alpha), \quad f_{10}: Z - (Z_0 + Z_1\alpha).$$

Now, what does “the circuit works” mean? All the polynomial obtained from the implementation generate an ideal I in a suitable polynomial ring. A point of $\mathcal{V}_{\mathbb{F}_q}(I)$ describes a way to assign values to the polynomial variables which comply with the circuit constraints. To verify the correctness of the circuit, we have to test whether these values also agree with the specification polynomial f , that is to say, whether f vanishes on all points of $\mathcal{V}_{\mathbb{F}_q}(I)$. By Theorem 2.6, this can be done by checking if $f \in (I, I_0)$. Notice also that if there is a point $\mathbf{p} \in \mathcal{V}_{\mathbb{F}_q}(I)$ such that $f(\mathbf{p}) \neq 0$, we are sure that there is a bug in the design.

The steps of the verification algorithm are now clear:

1. deduce the set of polynomials I corresponding to the circuit instance;
2. append to these the suitable field equations I_0 ;
3. compute a Gröbner basis \mathcal{G} of $I \cup I_0$;
4. reduce the specification polynomial f with respect to \mathcal{G} ;
5. if the output is 0, the circuit is correct; otherwise, there is a bug in the design.

In the rest of their article, Lv, Kalla and Enescu explain some techniques that improve the efficiency of the algorithm. In particular,

- they describe how to choose a monomial ordering such that the set of polynomial obtained in step 1. above is already a Gröbner basis, thus eliminating the need of computing one with Buchberger’s Algorithm;
- they use an improved reduction approach based on Faugère’s F_4 Algorithm, which has overall better performances.

2.3 Dealing with Time: the Polynomial Method

In the last section we focused our attention on combinational circuits, following [7]. What we want to do now is try to take time into account, that is to say, we want to generalize that method in order to obtain a way to verify sequential circuits.

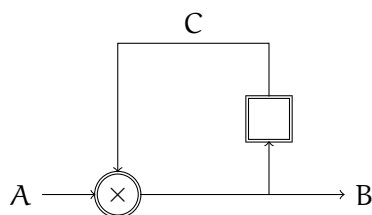
Let us consider a memory register. From Appendix B we know that it is a device capable of storing data. It changes its state at every clock cycle, updating its content according to its input terminal.

How can we represent a situation like this? It is not hard to imagine that we may label each polynomial variable with an integer index, which states the clock cycle at which we are considering the variable. In other words, we encode the information in the i -th wire of the circuit at time j in a variable X_{ij} of the polynomial ring. Obviously this brings to polynomial rings with a huge number of variables.

In this setting, every (combinational) logic gate $f(X_1, \dots, X_k)$ has to be repeated for each time: the ideal describing the circuit contains $f(X_{1j}, \dots, X_{kj})$ for all j . (Notice that we are assuming that the number of clock cycles required by the circuit is known in advance.) A memory register with input D and output Q is represented by the set of polynomials $Q_{j+1} - D_j$, meaning “the output at time $j + 1$ is equal to the input at time j ”. The polynomial $Q_1 - \beta$ is added to the set, representing the initial condition of the register (at time 1 the value β is stored).

A point in the variety $\mathcal{V}_{\mathbb{F}_q}(I)$ now encodes the possible values for the variables according not only to the circuit constraints but also to the evolution in time. Correctness of the design is achieved by assuring that the specification polynomial (also modelled taking time into account) vanishes on all the points of $\mathcal{V}_{\mathbb{F}_q}(I)$.

Example 2.1. We want to build a circuit that takes one value $a_i \in \mathbb{F}_q$ at each clock cycle and, after n cycles, returns $a_1 \cdot \dots \cdot a_n$. The circuit shown in Figure 2.2 is a good candidate. The gate \otimes multiplies two elements of \mathbb{F}_q . The inputs are given in the wire A and we expect the output from wire B . The register is set to 1 in the beginning.

Figure 2.2: A recursive multiplier over \mathbb{F}_q .

To have an instance of the circuit, let us set $n = 5$. The polynomial ring in which we are working is

$$R := \mathbb{F}_q[A_1, A_2, A_3, A_4, A_5, B_1, B_2, B_3, B_4, B_5, C_1, C_2, C_3, C_4, C_5].$$

The specification polynomial is $f: B_5 - A_1 A_2 A_3 A_4 A_5$. Let us model the gates.

The multiplier. The polynomials are $B_i - A_i C_i$ for each $i = 1, \dots, 5$.

The register. At first the register contains 1, so we have $C_1 = 1$; then, for $i = 2, \dots, 5$, we derive $C_i = B_{i-1}$.

To these we have to add the field equations $\ell_i^q - \ell_i$ for each $\ell \in \{A, B, C\}$ and $i = 1, \dots, 5$.

We choose \mathbb{F}_8 for a test with the software Sage. At first we define \mathbb{F}_8 and R with

```
K.<a>=FiniteField(8);
R=PolynomialRing(K, "A1, A2, A3, A4, A5, B1, B2, B3, B4, B5, C1, C2,
  C3, C4, C5");
R.inject_variables();
```

Then we define the ideal

```
I=Ideal (B1-A1*C1, B2-A2*C2, B3-A3*C3, B4-A4*C4, B5-A5*C5, C1-1,
  C2-B1, C3-B2, C4-B3, C5-B4, A1^8-A1, A2^8-A2, A3^8-A3, A4^8-A4,
  A5^8-A5, B1^8-B1, B2^8-B2, B3^8-B3, B4^8-B4, B5^8-B5, C1^8-C1,
  C2^8-C2, C3^8-C3, C4^8-C4, C5^8-C5);
```

and the specification polynomial

```
f=B5-A1*A2*A3*A4*A5;
```

The result of $f.reduce(I.groebner_basis())$ is 0, thus we may conclude that the circuit works.

2.4 Dealing with Time: the Module Method

The polynomial method has a major drawback in the fact that it requires rings with a lot of variables, which usually means high computational cost. In this section we try to overcome this problem by choosing another mathematical structure to represent the evolution of the circuit in time. Unfortunately, the price we have to pay is a loss of generality—this other method can be used only for a special class of circuits (which includes the combined RS encoder/syndrome generator of Chapter 1).

At first we thought to employ *modules* instead of rings. The environment is the module $\mathbb{K}[\mathbf{X}]^T = \mathbb{K}[X_1, \dots, X_n]^T$, where X_1, \dots, X_n represent wires and T is the number of clock cycles. The j -th component represents the j -th cycle. In other words, if $(\mathbf{e}_1, \dots, \mathbf{e}_T)$ is the canonical basis of $\mathbb{K}[\mathbf{X}]^T$ (as a $\mathbb{K}[\mathbf{X}]$ -module), the term $X_i \mathbf{e}_j$ stores the information of the i -th wire at time j .

Now it is easy to understand why only particular circuits can be verified with this method. For example, the recursive multiplier of Figure 2.2 *can't* be tested: the specification polynomial would contain expressions like $(A\mathbf{e}_1) \cdot \dots \cdot (A\mathbf{e}_k)$ which have no meaning in this context. The suitable circuits for this method are the *linear* ones, that is to say, their output is a \mathbb{K} -linear combination of their inputs.

This brought us to redefine the method: the underlying sets remain the same, but now they are regarded as \mathbb{K} -vector spaces instead of $\mathbb{K}[\mathbf{X}]$ -modules.

The circuit is modelled as a \mathbb{K} -vector subspace of $\mathbb{K}[X_1, \dots, X_n]^T$ generated by its constraints, which are derived with the following rules.

- Every combinatorial logic gate has to be repeated for each component (i.e. we have $f(X_1, \dots, X_n)\mathbf{e}_j$ for all $j = 1, \dots, T$). This means that for every logic gate $f(X_1, \dots, X_n)$ the columns of $f(X_1, \dots, X_n)I_T$, where I_T is the $T \times T$ identity matrix, belong to the subspace defining the circuit.
- A memory register with input D and output Q is described by the $T - 1$ vectors $Q\mathbf{e}_{j+1} - D\mathbf{e}_j$ (for $j = 1, \dots, T - 1$) and the initialization vector $(Q - \beta)\mathbf{e}_1$ if the register stores β as its initial value.

The specification polynomial now becomes a *specification vector*, built in the same way as in the polynomial method (with the obvious substitutions), and we have to test its membership to the subspace defined by the circuit.

Theorem 2.7. *Let $M \subseteq \mathbb{K}[\mathbf{X}]^T$ be the \mathbb{K} -subspace generated by the circuit units (as well as any other relation that is used by the polynomial method, such as the field equations), and let $\mathbf{f} \in \mathbb{K}[\mathbf{X}]^T$ be the specification vector. If $\mathbf{f} \in M$, the circuit works.*

Proof. In this proof, we will use the following notation:

- $\mathbf{X} = \{X_1, \dots, X_n\}$ will denote the set of the wire variables, without any reference to time;
- $\mathbf{X}_j := \{X_{ij} \mid i = 1, \dots, n\}$ is the set of the wire variables with a (fixed) time index j , which ranges from 1 to the number T of clock cycles required by the circuit;
- $\mathbf{X}^* := \mathbf{X}_1 \cup \dots \cup \mathbf{X}_T$.

We define the \mathbb{K} -linear map $\Psi: \mathbb{K}[\mathbf{X}]^T \rightarrow \mathbb{K}[\mathbf{X}^*]$ as

$$\begin{array}{ccc} \mathbb{K}[\mathbf{X}]^T & \longrightarrow & \bigoplus_{j=1}^T \mathbb{K}[\mathbf{X}_j] \longrightarrow \mathbb{K}[\mathbf{X}^*] \\ \begin{pmatrix} f_1(\mathbf{X}) \\ \vdots \\ f_T(\mathbf{X}) \end{pmatrix} & \longmapsto & \begin{pmatrix} f_1(\mathbf{X}_1) \\ \vdots \\ f_T(\mathbf{X}_T) \end{pmatrix} \longmapsto \sum_{j=1}^T f_j(\mathbf{X}_j). \end{array}$$

In other words, the first map simply renames the variables of f_j adding the time index j , and the second map sums all the components.

Suppose that M is generated by the (finite) circuit relations $\mathbf{v}_1, \dots, \mathbf{v}_m$. Our claims are that

1. $\Psi(f)$ and $\Psi(\mathbf{v}_1), \dots, \Psi(\mathbf{v}_m)$ are respectively the specification polynomial and the polynomials representing the gates and registers for the polynomial method;
2. if $f \in M$, then $\Psi(f)$ belongs to the ideal generated by $\Psi(\mathbf{v}_1), \dots, \Psi(\mathbf{v}_m)$, thus proving the correctness of the circuit by the polynomial method of Section 2.3.

For $j = 1, \dots, T$ let \mathbf{e}_j be the vector with 1 in the j -th component and 0 elsewhere. The first claim follows directly by definition of Ψ . Here are some examples:

- if $g(\mathbf{X})$ represents a combinational logic gate, we have $g(\mathbf{X})\mathbf{e}_j$ for each $j = 1, \dots, T$, whose image under Ψ is $g(\mathbf{X}_j)$;
- a register with input D and output Q is modelled by $Q\mathbf{e}_{j+1} - D\mathbf{e}_j$, which is sent to $Q_{j+1} - D_j$ by Ψ .

The second claim is clear too, because of the linearity of Ψ : suppose that $f \in M$ and write

$$\mathbf{f} = \sum_{i=1}^m \beta_i \mathbf{v}_i$$

for $\beta_1, \dots, \beta_m \in \mathbb{K}$. Then

$$\Psi(f) = \sum_{i=1}^m \beta_i \Psi(\mathbf{v}_i) \in (\Psi(\mathbf{v}_1), \dots, \Psi(\mathbf{v}_m)).$$

This completes the proof. \square

Example 2.2. We slightly modify the circuit of Figure 2.2 so that now it recursively *adds* instead of multiplies.

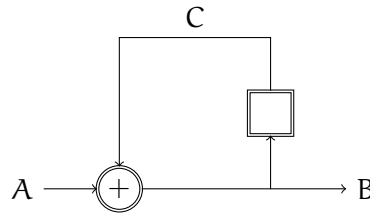


Figure 2.3: A recursive adder over \mathbb{F}_q .

This time the register is set to 0 at first. We choose for example $n = 5$ and obtain the relations

$$\mathbf{A} := \begin{pmatrix} B - (A + C) & 0 & 0 & 0 & 0 \\ 0 & B - (A + C) & 0 & 0 & 0 \\ 0 & 0 & B - (A + C) & 0 & 0 \\ 0 & 0 & 0 & B - (A + C) & 0 \\ 0 & 0 & 0 & 0 & B - (A + C) \end{pmatrix}$$

for the adder, and

$$\mathbf{R} := \begin{pmatrix} C & -B & 0 & 0 & 0 \\ 0 & C & -B & 0 & 0 \\ 0 & 0 & C & -B & 0 \\ 0 & 0 & 0 & C & -B \\ 0 & 0 & 0 & 0 & C \end{pmatrix}$$

for the register. (We should add also the field equations, but in this simple example they are irrelevant.) The specification vector is

$$\mathbf{f} := \begin{pmatrix} -A \\ -A \\ -A \\ -A \\ B - A \end{pmatrix}.$$

We have to check whether f belongs to the subspace of the \mathbb{K} -linear combinations of the columns of \mathbf{A} and \mathbf{R} . A quick computation, in fact, tells us that

$$f = \mathbf{a}_1 + \mathbf{a}_2 + \mathbf{a}_3 + \mathbf{a}_4 + \mathbf{a}_5 + \mathbf{r}_1 + \mathbf{r}_2 + \mathbf{r}_3 + \mathbf{r}_4 + \mathbf{r}_5$$

(where the \mathbf{a}_i 's and the \mathbf{r}_i 's are the columns of \mathbf{A} and \mathbf{R}), thus establishing the correctness of the circuit.

2.5 A First Result: Arbitrariness of the Coefficient Field

In this section we are going to show that, if the circuit specification and the constraints are modelled with linear polynomials, a correctness proof based on the polynomial method will be valid independently of the underlying field of coefficients—under the obvious assumption that the adders and the scalar multipliers of a particular instance of the circuit are designed to operate with elements of the suitable field.

Remember that the information about our working field is given by the field equations, which are polynomials like $X_i^q - X_i$. The field equations have to be appended to the ideal generated by the circuit constraints in order to trigger the Nullstellensatz over Finite Fields. The question is, do we really need them? Generally speaking, the answer is yes: there are examples in which their absence brings to wrong results. The situation is different when we are dealing with linear polynomials.

Recall that there are some useful criteria that avoid some computation when looking for a Gröbner basis. We will report here just one of them; its proof can be found, for example, in [1].

Proposition 2.8 (Product Criterion). *Let $f, g \in \mathbb{K}[X_1, \dots, X_n]$ be polynomials such that $\text{lt}(f)$ and $\text{lt}(g)$ are coprime. Then their S -polynomial $S(f, g)$ reduces to 0.*

Lemma 2.9. *The S -polynomial between two linear polynomials either reduces to 0 or is linear. Moreover, reducing a linear polynomial with other linear polynomials results in a linear polynomial.*

Proof. A linear polynomial $f \in \mathbb{K}[X_1, \dots, X_n]$ may be written as

$$f(\mathbf{X}) = \alpha_1 X_1 + \dots + \alpha_n X_n + \alpha_0.$$

Let f and g be linear polynomials, with $\text{lt}(f) = \alpha_i X_i$ and $\text{lt}(g) = \beta_j X_j$.

- If $i \neq j$, the two leading terms are coprime, hence $S(f, g)$ reduces to 0 by the Product Criterion.

- If $i = j$, suppose $\text{lt}(f) = \alpha_i X_i$ and $\text{lt}(g) = \beta_i X_i$, then $S(f, g) = \beta_i f - \alpha_i g$ is a linear polynomial.

As far as reduction is concerned, a linear polynomial f can be reduced by another linear polynomial g only if $\text{lm}(g)$ appears among the monomials of f ; in that case, supposing that $\text{lt}(g) = \beta_i X_i$ and that there is $\alpha_i X_i$ in f , reduction of f by g results in $f - (\alpha_i/\beta_i)g$, which is a linear polynomial. \square

Now, the polynomial method requires an ideal membership test in order to prove the correctness of the circuit. In particular, for linear circuits we have to check whether a linear specification polynomial belongs to an ideal generated by other linear polynomials and the field equations. But the field equations are not linear, which means they will never participate during the reduction step of the algorithm, because of their high degree. Thus, it is possible to change them with other field equations without affecting the passages of the membership test.

This proves that changing the field of the coefficients (i.e. changing the field equations) doesn't modify the outcome of the test. We conclude that the polynomial method can be used for testing correctness of circuits whose constraints are given by linear polynomials, without having to fix a field for the coefficients.

However, we have to be very careful when applying the principle stated above. The field equations, in fact, limit the values that the variables may take; they don't affect the coefficient field directly. This means that, if the circuit is designed to work with specific constraints over the coefficients (e.g. only in characteristic 2), the field has to fulfil these requirements.

Chapter 3

The Difference Equations Approach

The commutative algebra techniques analysed in Chapter 2 can be applied when the number of clock cycles is known in advance. In this chapter we try to change our point of view: we translate the behaviour of the circuit into a system of difference equations. These equations relate the state and the output at time k with the state and the input at time $k - 1$; as a consequence, it is not necessary to know the number of steps in advance, because we can proceed one step at a time.

We will consider only linear difference equations, because the RS encoder we are going to study is a linear circuit. Thus, the methods that we are going to show are not suitable for other types of circuits.

3.1 Difference Equations

A difference equation may be seen as a sort of “discrete differential equation”, that is to say, it is a relation that expresses the evolution in time of a discrete function (i.e. a sequence).

Definition 3.1. A (linear) difference equation of order k with coefficients in \mathbb{K} is a relation of the form

$$\sum_{j=0}^k h_j a_{i+j} = 0, \quad (3.1)$$

where $i \in \mathbb{N}$, $h_j \in \mathbb{K}$, $h_0 \neq 0$, $h_k = 1$ and the a_i 's are unknowns. A solution of the difference equation is a sequence $(a_i)_{i \in \mathbb{N}} \in \mathbb{K}^{\mathbb{N}}$ whose terms satisfy (3.1).

It is not difficult to show that the set of all solutions of a difference equation of order k is a k -dimensional \mathbb{K} -linear subspace of $\mathbb{K}^{\mathbb{N}}$. In fact, for each

$(b_0, \dots, b_{k-1}) \in \mathbb{K}^k$ there exists a unique solution $(a_i)_{i \in \mathbb{N}}$ such that $a_0 = b_0, \dots, a_{k-1} = b_{k-1}$.

Definition 3.2. Let $\sum h_j a_{i+j} = 0$ be a difference equation of order k . The polynomial

$$h(T) := \sum_{j=0}^k h_j T^j$$

is called *characteristic polynomial* of the equation.

The roots of the characteristic polynomial play an important role in determining a so-called *closed form* of a solution, that is an expression of the n -th term a_n that depends on n only, and not on the previous terms. Let α be a root of $h(T)$ in a suitable algebraic extension of \mathbb{K} . Then the sequence $(\alpha^n)_{n \in \mathbb{N}}$ obviously satisfies the difference equation. Assuming that the roots of $h(T)$ are distinct (let them be $\alpha_1, \dots, \alpha_k$), linearity implies that for any \mathbb{K} -linear combination

$$s_n := y_1 \alpha_1^n + \dots + y_k \alpha_k^n, \quad y_1, \dots, y_k \in \mathbb{K}$$

the sequence $(s_n)_{n \in \mathbb{N}}$ is a solution of the equation.

Example 3.1. The Fibonacci numbers are generated by the second order difference equation (with coefficients in \mathbb{R})

$$a_{i+2} = a_{i+1} + a_i \tag{3.2}$$

along with the initial conditions $a_0 = 0, a_1 = 1$. The characteristic polynomial is $h(T) = T^2 - T - 1$, whose roots are

$$\frac{1 + \sqrt{5}}{2} \quad \text{and} \quad \frac{1 - \sqrt{5}}{2}.$$

The set of the solutions of Equation (3.2) is

$$\left\{ \left(y_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + y_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)_{n \in \mathbb{N}} \mid y_1, y_2 \in \mathbb{R} \right\}.$$

By imposing the initial condition

$$\begin{cases} y_1 + y_2 = 0 \\ y_1 \left(\frac{1 + \sqrt{5}}{2} \right) + y_2 \left(\frac{1 - \sqrt{5}}{2} \right) = 1 \end{cases}$$

and solving the system, we obtain the well-known formula for the Fibonacci numbers

$$a_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n.$$

So, how can a sequential circuit be translated into a difference equation? There are three important sequences related to a generic sequential circuit:

- the *input* sequence $(u(t))_{t \in \mathbb{N}}$;
- the *state* sequence $(x(t))_{t \in \mathbb{N}}$;
- the *output* sequence $(y(t))_{t \in \mathbb{N}}$.

Each sequence contains information about the input, state and output of the circuit at every clock cycle t . The behaviour of a linear circuit is encoded in two difference equations:

$$\begin{cases} x(t+1) = Ax(t) + Bu(t) & (3.3a) \\ y(t) = Cx(t) + Du(t) & (3.3b) \end{cases}$$

where A, B, C, D are constants determined by the circuit constraints. Equation (3.3a) tells us how the state changes in function of the current state and input; Equation (3.3b) express the output in terms of the state and the input. Notice that both equations are linear.

All the objects in Equations (3.3) may be scalars, vectors or matrices, depending on the particular circuit, if those representations are more suitable. The dimensions of the constants are chosen in order to preserve the equations' consistency.

Once translated into a system of difference equations, a circuit can be studied from that point of view. For example, we may know *a priori* how the output $y(t)$ is related to the input $u(t)$, so that we just have to check if these sequences satisfy Equations (3.3). In the following sections we will describe some tools used to analyse difference equations.

3.2 The Z-Transform

A difference equation talks about sequences; we may enter the realm of Algebra turning those sequences into algebraic objects.

Definition 3.3. Let $\mathbf{a} = (a_n)_{n \in \mathbb{N}} \in \mathbb{K}^{\mathbb{N}}$ be a sequence. The *Z-transform* of \mathbf{a} , denoted by $\mathcal{Z}[\mathbf{a}]$, is the formal power series

$$\mathcal{Z}[\mathbf{a}] := \sum_{n=0}^{\infty} a_n Z^{-n} \in \mathbb{K}[[Z^{-1}]].$$

Table 3.1 shows some Z-transforms for some of the basic sequences. Besides the definition, there are some tools that allow to compute Z-transforms from other known ones.

Name	Definition	Z-transform
Discrete impulse	$\delta_0(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}$	$\mathcal{Z}[\delta_0] = 1$
Discrete step	$h_n = 1 \forall n \in \mathbb{N}$	$\mathcal{Z}[h] = \frac{Z}{Z-1}$
Geometric sequence	$g_n = \alpha^n$	$\mathcal{Z}[g] = \frac{Z}{Z-\alpha}$

Table 3.1: Some common sequences with their Z-transforms.

Proposition 3.4 (Linearity of Z-Transform). *For any $\mathbf{a}, \mathbf{b} \in \mathbb{K}^{\mathbb{N}}$ and $\alpha, \beta \in \mathbb{K}$, we have*

$$\mathcal{Z}[\alpha \mathbf{a} + \beta \mathbf{b}] = \alpha \mathcal{Z}[\mathbf{a}] + \beta \mathcal{Z}[\mathbf{b}].$$

Proof. It follows immediately from the \mathbb{K} -vector space structure of $\mathbb{K}^{\mathbb{N}}$ and $\mathbb{K}[[X^{-1}]]$. \square

Proposition 3.5 (Forward Shift Operator). *Let $\mathbf{a} = (a_n)_{n \in \mathbb{N}}$ be a sequence. Define the forward shift operator $\mathcal{F}: \mathbb{K}^{\mathbb{N}} \rightarrow \mathbb{K}^{\mathbb{N}}$ as*

$$(\mathcal{F}(\mathbf{a}))_n := a_{n+1} \quad \forall n \in \mathbb{N}.$$

Then

$$\mathcal{Z}[\mathcal{F}(\mathbf{a})] = Z \mathcal{Z}[\mathbf{a}] - Z a_0.$$

Proof. This is a straightforward computation, isolating the first term of the series $Z \mathcal{Z}[\mathbf{a}]$. \square

Proposition 3.6 (Backward Shift Operator). *Let $\mathbf{a} = (a_n)_{n \in \mathbb{N}}$ be a sequence. Define the backward shift operator $\mathcal{B}: \mathbb{K}^{\mathbb{N}} \rightarrow \mathbb{K}^{\mathbb{N}}$ as*

$$(\mathcal{B}(\mathbf{a}))_n := \begin{cases} a_{n-1} & \text{for } n \neq 0, \\ 0 & \text{for } n = 0. \end{cases}$$

Then

$$\mathcal{Z}[\mathcal{B}(\mathbf{a})] = Z^{-1} \mathcal{Z}[\mathbf{a}].$$

Proof. It suffices to collect a Z^{-1} from $\mathcal{Z}[\mathcal{B}(\mathbf{a})]$. \square

3.3 Transfer Functions

Let us consider a system of difference equations associated to a circuit

$$\begin{cases} x(t+1) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases}$$

with $x(0) = x_0$. We compute the Z-transforms of all the sequences appearing in those equations. Let $X(Z) := \mathcal{Z}[x(t)]$, $Y(Z) := \mathcal{Z}[y(t)]$ and $U(Z) := \mathcal{Z}[u(t)]$. Linearity and forward shift rule lead to

$$\begin{cases} ZX(Z) - Zx_0 = AX(Z) + BU(Z) \\ Y(Z) = CX(Z) + DU(Z) \end{cases}$$

and a simple manipulation gives

$$\begin{cases} X(Z) = Z(ZI - A)^{-1}x_0 + (ZI - A)^{-1}BU(Z) \\ Y(Z) = ZC(ZI - A)^{-1}x_0 + (C(ZI - A)^{-1}B + D)U(Z) \end{cases}$$

where I is an identity matrix of appropriate dimension.

Definition 3.7. The *transfer function* of a circuit is the ratio

$$G(Z) := C(ZI - A)^{-1}B + D$$

between the Z-transform of the output and the Z-transform of the input for the initial state $x_0 = 0$.

Notice that the transfer function does not depend on the particular input, but only on the linear system that we are studying.

Example 3.2. Let's compute the transfer function of a single unit of the syndrome generator, like the one of Figure 1.2. In this case, input and output symbols are

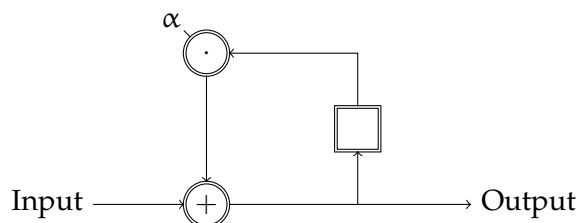


Figure 3.1: The same of Figure 1.2.

managed one at a time, and there is just one register, so the sequences $x(t)$, $y(t)$ and $u(t)$ are all 1-dimensional.

Now, let's follow the wires. The content of the register is multiplied by α , then added to the input, and the result is given as output; the difference equation that models this behaviour is

$$y(t) = \alpha x(t) + u(t).$$

The same output wire is also the one which updates the register, so we have

$$x(t+1) = \alpha x(t) + u(t).$$

In other words, this circuit has $A = C = \alpha$ and $B = D = 1$. The transfer function is then

$$G(Z) = 1 + \frac{\alpha}{Z - \alpha} = \frac{Z}{Z - \alpha}.$$

3.4 Circuit Equivalence

Besides verifying a circuit correctness, the difference equations may also test the equivalence of two linear circuits.

Definition 3.8. Two circuits described by the systems

$$\begin{cases} x(t+1) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases} \quad \text{and} \quad \begin{cases} w(t+1) = \tilde{A}w(t) + \tilde{B}u(t) \\ y(t) = \tilde{C}w(t) + \tilde{D}u(t) \end{cases}$$

are *algebraically equivalent* if there exists an invertible linear transformation T such that $x(t) = Tw(t)$ for all t and

$$\begin{aligned} \tilde{A} &= T^{-1}AT, & \tilde{B} &= T^{-1}B, \\ \tilde{C} &= CT, & \tilde{D} &= D. \end{aligned}$$

It is easy to show that two algebraically equivalent circuits are "equivalent" in the sense that they produce the same outputs if their input is the same. In fact, from the second circuit we have

$$\begin{cases} w(t+1) = T^{-1}ATw(t) + T^{-1}Bu(t) \\ y(t) = CTw(t) + Du(t) \end{cases}$$

and, multiplying by T on the left the first equation and writing $x(t) = Tw(t)$, we obtain the equations of the first circuit.

As we may expect, since the transfer function depends on the behaviour of a circuit, two algebraically equivalent circuits have also the same transfer function:

$$\begin{aligned} \tilde{G}(Z) &= \tilde{C}(ZI - \tilde{A})^{-1}\tilde{B} + \tilde{D} = \\ &= CT(ZT^{-1}IT - T^{-1}AT)T^{-1}B + D = \\ &= CTT^{-1}(ZI - A)^{-1}TT^{-1}B + D = \\ &= C(ZI - A)^{-1}B + D = G(Z). \end{aligned}$$

Chapter 4

Correctness of the Combined RS Encoder/Syndrome Generator

In this chapter we will apply the methods described in the previous chapters in order to prove that the combined RS encoder/syndrome generator of Chapter 1 is correct. In the “decode” mode, the circuit reduces to the standard syndrome generator of Figure 1.3, so we will consider the “encode” mode only.

At first we will fix a length and an error capability of the RS code and we will verify an instance of the circuit with those parameters using the polynomial method. Then we will translate the general circuit into a system of difference equations and will work on them, in order to prove that the encoder is correct for any choice of the parameters.

4.1 Particular Instances of the Circuit

The polynomial method requires all the circuit parameters to be known in advance. It is useful when we have a particular instance of the circuit and we want to test its correctness. Unfortunately, it relies on actual computations, which means we can’t leave a literal parameter (such as $2t$, the number of check symbols). In other words, this method is unable to prove the correctness of the circuit *structure*—it verifies single instances.

4.1.1 2-Error-Correcting RS over \mathbb{F}_8

Let us begin with the 2-error-correcting RS code over \mathbb{F}_8 of Example 1.1. Recall that the encoder takes a word of $(\mathbb{F}_8)^3$ and ensures a codeword of $(\mathbb{F}_8)^7$, adding 4 parity-check symbols of \mathbb{F}_8 .

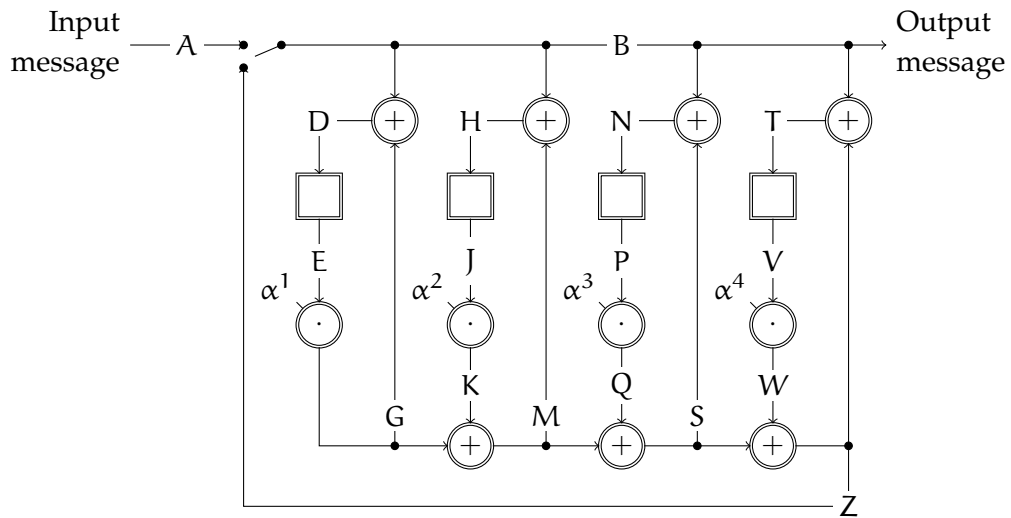


Figure 4.1: The encoder of Figure 1.5 with $t = 2$. Each wire is labelled with a polynomial indeterminate.

The actual circuit is shown in Figure 4.1. The seventeen wires are each labelled with a letter from

$$\mathcal{L} := \{A, B, D, E, G, H, J, K, M, N, P, Q, S, T, V, W, Z\}.$$

The circuit requires seven clock cycles, so there is a total of 119 polynomial variables

$$\mathcal{L}^* := \{\ell_i \mid \ell \in \mathcal{L}, i = 1, \dots, 7\},$$

that is to say, our environment ring will be $\mathbb{F}_8[\mathcal{L}^*]$.

We have now to translate the design of the circuit into an ideal. Table 4.1 lists the polynomials which arise from the various gates, as well as the field equations.

After that we define the specification polynomials. This encoding method is systematic, so the first three outputs have to be equal to the inputs:

$$\begin{aligned} f_1 &:= B_1 - A_1 \\ f_2 &:= B_2 - A_2 \\ f_3 &:= B_3 - A_3. \end{aligned}$$

Recall that, for a message $m(X)$, the parity-check symbols are obtained by dividing $m(X) \cdot X^{2t}$ by $g(X)$ and taking the coefficients of the remainder, where t is the error-correcting capability of the code and g is the generator polynomial. In

Type of gate	Polynomials	Ranges	#
First row of adders	$B_i + G_i - D_i, B_i + M_i - H_i,$ $B_i + S_i - N_i, B_i + Z_i - T_i$	$i = 1, \dots, 7$	28
Registers initialization	E_1, J_1, P_1, V_1		4
Registers	$E_{i+1} - D_i, J_{i+1} - H_i,$ $P_{i+1} - N_i, V_{i+1} - T_i$	$i = 1, \dots, 6$	24
Multipliers	$G_i - \alpha E_i, K_i - \alpha^2 H_i,$ $Q_i - \alpha^3 P_i, W_i - \alpha^4 V_i$	$i = 1, \dots, 7$	28
Second row of adders	$G_i + K_i - M_i, M_i + Q_i - S_i,$ $S_i + W_i - Z_i$	$i = 1, \dots, 7$	21
Switch	$B_i - A_i$ $B_i - Z_i$	$i = 1, \dots, 3$ $i = 4, \dots, 7$	7
Field equations	$\ell_i^8 - \ell_i$	$\ell_i \in \mathcal{L}^*$	119
Total number of polynomials			231

Table 4.1: List of the polynomials associated to the circuit in Figure 4.1.

our case ($t = 2, m(X) = (A_1)X^2 + (A_2)X + (A_3)$)

$$\begin{aligned} m(X) \cdot X^{2t} &= (A_1)X^6 + (A_2)X^5 + (A_3)X^4, \\ g(X) &= X^4 + (\alpha + 1)X^3 + X^2 + \alpha X + (\alpha + 1), \end{aligned}$$

and a quick computation gives us

$$\begin{aligned} f_4 &:= B_4 - ((\alpha^2 + \alpha)A_1 + \alpha^2 A_2 + (\alpha + 1)A_3) \\ f_5 &:= B_5 - (A_1 + A_2 + A_3) \\ f_6 &:= B_6 - ((\alpha^2 + \alpha)A_1 + (\alpha^2 + 1)A_2 + \alpha A_3) \\ f_7 &:= B_7 - ((\alpha^2 + \alpha + 1)A_1 + (\alpha^2 + 1)A_2 + (\alpha + 1)A_3). \end{aligned}$$

Let $I \subseteq \mathbb{F}_8[\mathcal{L}^*]$ the ideal generated by the 231 polynomials of Table 4.1. Correctness of the circuit is established if $f_i \in I$ for $i = 1, \dots, 7$.

We tested our method with the Sage software on a Intel[®] Atom[™] N550 processor @1.50 GHz with 2 GB RAM @667 MHz. First of all we have to set up the environment.

```
K.<a>=FiniteField(8);
```

```
R=PolynomialRing(K, "A1,A2,A3,A4,A5,A6,A7,B1,B2,B3,B4,B5,B6,
  B7,D1,D2,D3,D4,D5,D6,D7,E1,E2,E3,E4,E5,E6,E7,G1,G2,G3,
  G4,G5,G6,G7,H1,H2,H3,H4,H5,H6,H7,J1,J2,J3,J4,J5,J6,J7,
  K1,K2,K3,K4,K5,K6,K7,M1,M2,M3,M4,M5,M6,M7,N1,N2,N3,N4,
  N5,N6,N7,P1,P2,P3,P4,P5,P6,P7,Q1,Q2,Q3,Q4,Q5,Q6,Q7,S1,
  S2,S3,S4,S5,S6,S7,T1,T2,T3,T4,T5,T6,T7,V1,V2,V3,V4,V5,
  V6,V7,W1,W2,W3,W4,W5,W6,W7,Z1,Z2,Z3,Z4,Z5,Z6,Z7");
```

```
R.inject_variables();
```

The next step is putting together all the equations into an ideal I:

```
I=Ideal (B1+G1-D1, B2+G2-D2, B3+G3-D3, B4+G4-D4, B5+G5-D5,
  B6+G6-D6, B7+G7-D7, B1+M1-H1, B2+M2-H2, B3+M3-H3, B4+M4-H4,
  B5+M5-H5, B6+M6-H6, B7+M7-H7, B1+S1-N1, B2+S2-N2, B3+S3-N3,
  B4+S4-N4, B5+S5-N5, B6+S6-N6, B7+S7-N7, B1+Z1-T1, B2+Z2-T2,
  B3+Z3-T3, B4+Z4-T4, B5+Z5-T5, B6+Z6-T6, B7+Z7-T7, E1, J1, P1,
  V1, E2-D1, E3-D2, E4-D3, E5-D4, E6-D5, E7-D6, J2-H1, J3-H2,
  J4-H3, J5-H4, J6-H5, J7-H6, P2-N1, P3-N2, P4-N3, P5-N4, P6-N5,
  P7-N6, V2-T1, V3-T2, V4-T3, V5-T4, V6-T5, V7-T6, G1-a*E1,
  G2-a*E2, G3-a*E3, G4-a*E4, G5-a*E5, G6-a*E6, G7-a*E7,
  K1-(a^2)*J1, K2-(a^2)*J2, K3-(a^2)*J3, K4-(a^2)*J4,
  K5-(a^2)*J5, K6-(a^2)*J6, K7-(a^2)*J7, Q1-(a^3)*P1,
  Q2-(a^3)*P2, Q3-(a^3)*P3, Q4-(a^3)*P4, Q5-(a^3)*P5,
  Q6-(a^3)*P6, Q7-(a^3)*P7, W1-(a^4)*V1, W2-(a^4)*V2,
  W3-(a^4)*V3, W4-(a^4)*V4, W5-(a^4)*V5, W6-(a^4)*V6,
  W7-(a^4)*V7, G1+K1-M1, G2+K2-M2, G3+K3-M3, G4+K4-M4,
  G5+K5-M5, G6+K6-M6, G7+K7-M7, M1+Q1-S1, M2+Q2-S2, M3+Q3-S3,
  M4+Q4-S4, M5+Q5-S5, M6+Q6-S6, M7+Q7-S7, S1+W1-Z1, S2+W2-Z2,
  S3+W3-Z3, S4+W4-Z4, S5+W5-Z5, S6+W6-Z6, S7+W7-Z7, B1-A1,
  B2-A2, B3-A3, B4-Z4, B5-Z5, B6-Z6, B7-Z7, A1^8-A1, A2^8-A2,
  A3^8-A3, A4^8-A4, A5^8-A5, A6^8-A6, A7^8-A7, B1^8-B1, B2^8-B2,
  B3^8-B3, B4^8-B4, B5^8-B5, B6^8-B6, B7^8-B7, D1^8-D1, D2^8-D2,
  D3^8-D3, D4^8-D4, D5^8-D5, D6^8-D6, D7^8-D7, E1^8-E1, E2^8-E2,
  E3^8-E3, E4^8-E4, E5^8-E5, E6^8-E6, E7^8-E7, G1^8-G1, G2^8-G2,
  G3^8-G3, G4^8-G4, G5^8-G5, G6^8-G6, G7^8-G7, H1^8-H1, H2^8-H2,
  H3^8-H3, H4^8-H4, H5^8-H5, H6^8-H6, H7^8-H7, J1^8-J1, J2^8-J2,
  J3^8-J3, J4^8-J4, J5^8-J5, J6^8-J6, J7^8-J7, K1^8-K1, K2^8-K2,
  K3^8-K3, K4^8-K4, K5^8-K5, K6^8-K6, K7^8-K7, M1^8-M1, M2^8-M2,
  M3^8-M3, M4^8-M4, M5^8-M5, M6^8-M6, M7^8-M7, N1^8-N1, N2^8-N2,
```

```

N3^8-N3, N4^8-N4, N5^8-N5, N6^8-N6, N7^8-N7, P1^8-P1, P2^8-P2,
P3^8-P3, P4^8-P4, P5^8-P5, P6^8-P6, P7^8-P7, Q1^8-Q1, Q2^8-Q2,
Q3^8-Q3, Q4^8-Q4, Q5^8-Q5, Q6^8-Q6, Q7^8-Q7, S1^8-S1, S2^8-S2,
S3^8-S3, S4^8-S4, S5^8-S5, S6^8-S6, S7^8-S7, T1^8-T1, T2^8-T2,
T3^8-T3, T4^8-T4, T5^8-T5, T6^8-T6, T7^8-T7, V1^8-V1, V2^8-V2,
V3^8-V3, V4^8-V4, V5^8-V5, V6^8-V6, V7^8-V7, W1^8-W1, W2^8-W2,
W3^8-W3, W4^8-W4, W5^8-W5, W6^8-W6, W7^8-W7, Z1^8-Z1, Z2^8-Z2,
Z3^8-Z3, Z4^8-Z4, Z5^8-Z5, Z6^8-Z6, Z7^8-Z7);

```

```
G=I.groebner_basis();
```

The Gröbner basis^[1] of I has 119 polynomials and has been computed in 1.86 s. Now we define the specification polynomials.

```

f1=B1-A1;
f2=B2-A2;
f3=B3-A3;
f4=B4-(a^2+a)*A1-(a^2)*A2-(a+1)*A3;
f5=B5-A1-A2-A3;
f6=B6-(a^2+a)*A1-(a^2+1)*A2-(a)*A3;
f7=B7-(a^2+a+1)*A1-(a^2+1)*A2-(a+1)*A3;

```

The correctness of the circuit is established once we have verified that

$$f_1, \dots, f_7 \in I,$$

for which we can ask Sage:

```
[f1.reduce(G), f2.reduce(G), f3.reduce(G), f4.reduce(G),
f5.reduce(G), f6.reduce(G), f7.reduce(G)];
```

The answer is

```
[0, 0, 0, 0, 0, 0, 0]
```

and we can conclude that the circuit works correctly.

4.1.2 2-Error-Correcting RS over \mathbb{F}_8 with a Bug

Now suppose that there is a bug in the circuit implementation. In this example, we change one of the adders of the second row with a multiplier (see Figure 4.2). The only difference in I is that the polynomials $M_i + Q_i - S_i$ become $M_i Q_i - S_i$.

We tried to perform the computation, but after more than 6 hours both the 2 GB RAM and the 1 GB SWAP memories got saturated, and we didn't manage to obtain a Gröbner basis.

^[1]Sage uses the DEGREVLEX monomial ordering by default.

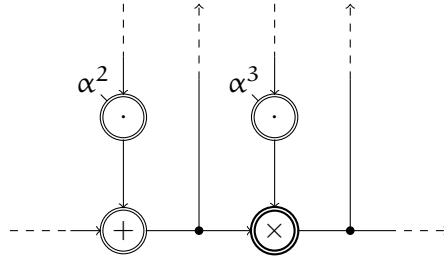


Figure 4.2: A bug in the circuit of Figure 4.1.

4.1.3 4-Error-Correcting RS over \mathbb{F}_{32}

For the last example we choose the field \mathbb{F}_{32} , implemented as $\mathbb{F}_2[\alpha]$ with minimal polynomial $\alpha^5 + \alpha^2 + 1$, and $t = 4$, so that we have 23 message symbols and 8 check symbols. The generator polynomial is

$$g(X) := (X - \alpha) \cdot \dots \cdot (X - \alpha^8).$$

There are 33 wires and 31 clock cycles are needed, for a total of 1023 polynomial indeterminates. In particular, we will work in $\mathbb{F}_{32}[\mathcal{L}^*]$ with

$$\mathcal{L} := \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, \\ R, S, T, U, V, W, X, Y, Z, \Gamma, \Theta, \Lambda, \Xi, \Phi, \Psi, \Omega\}$$

and $\mathcal{L}^* := \{\ell_i \mid \ell \in \mathcal{L}, i = 1, \dots, 31\}$. It is not difficult to write the polynomial constraint of this circuit: they are similar to the ones seen before. We will not report them here, mainly because they take too much space—there are 992 polynomials which arise from the gates and 1023 field equations, which means a total of 2015 polynomials!

The first 23 specification polynomials are $f_i := B_i - A_i$ (for $i = 1, \dots, 23$), because the encoding is systematic. These already belong to the ideal defined by the circuit (they appear in the description of the switch), so we may ignore them. A (not so) quick computation gives the formulas for the other outputs.

$$f_{24} := B_{24} + (\alpha^4 + \alpha^2 + \alpha + 1)A_1 + (\alpha^3 + \alpha^2)A_2 + (\alpha^3 + \alpha^2 + \alpha + 1)A_3 + \\ + (\alpha^4 + \alpha^3)A_4 + (\alpha^4 + \alpha^2 + 1)A_5 + (\alpha^4)A_6 + (\alpha^4 + \alpha^3 + \alpha^2 + 1)A_7 + \\ + (\alpha^4 + \alpha^2 + \alpha)A_8 + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha)A_9 + (\alpha^4 + \alpha^3 + \alpha)A_{10} + \\ + (\alpha^3 + \alpha + 1)A_{11} + (\alpha)A_{12} + (\alpha^3 + \alpha^2 + \alpha)A_{13} + (\alpha^3 + 1)A_{14} + \\ + (\alpha^2)A_{15} + (\alpha^3 + \alpha^2 + 1)A_{16} + (\alpha^3)A_{17} + (\alpha^2 + \alpha + 1)A_{18} + \\ + (\alpha^3 + \alpha^2 + \alpha + 1)A_{19} + (\alpha)A_{20} + (\alpha^4 + \alpha^2 + 1)A_{21} +$$

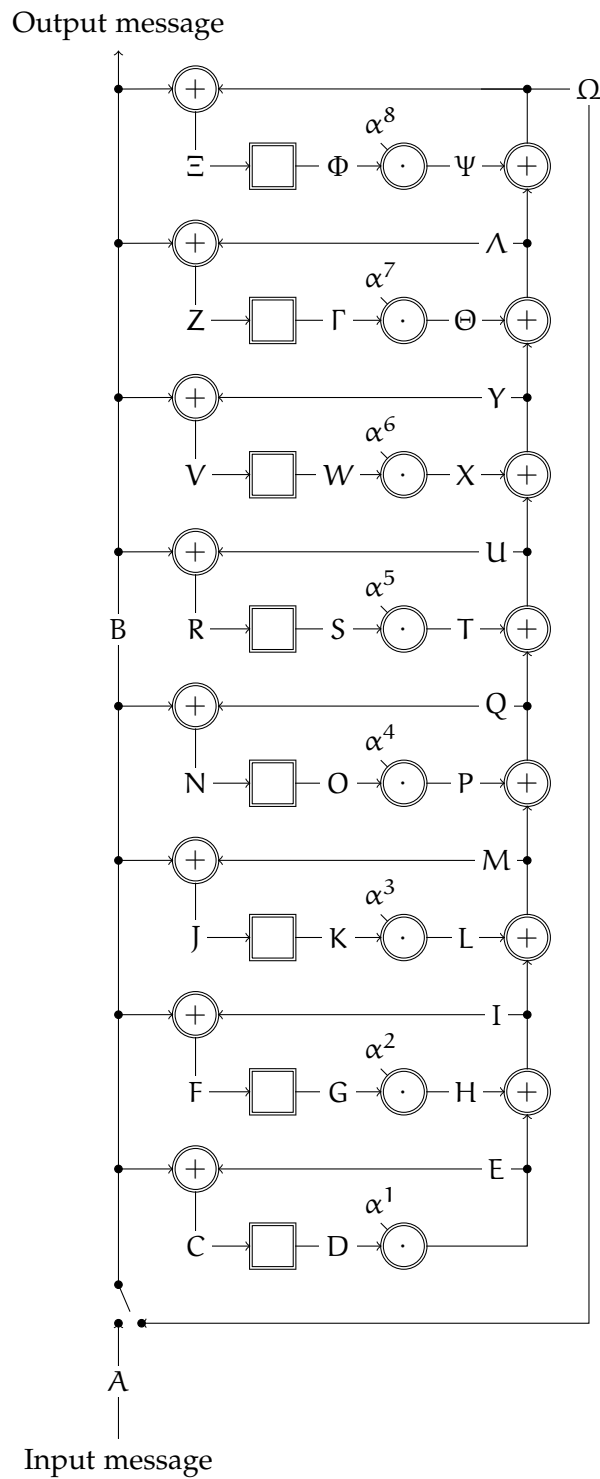


Figure 4.3: The encoder with $t = 4$.

$$\begin{aligned}
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1)A_{22} + (\alpha^3)A_{23} \\
f_{25} := & B_{25} + (\alpha^3 + 1)A_1 + (\alpha^4 + \alpha^3)A_2 + (\alpha^4 + \alpha^3 + \alpha + 1)A_3 + (\alpha^4 + 1)A_4 + \\
& + A_5 + A_6 + (\alpha + 1)A_7 + (\alpha^4 + \alpha^3 + \alpha^2)A_8 + (\alpha^4 + \alpha^3 + \alpha^2 + 1)A_9 + \\
& + (\alpha^4)A_{10} + (\alpha^3)A_{11} + (\alpha^4 + \alpha^3 + \alpha + 1)A_{12} + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + 1)A_{13} + (\alpha^3 + \alpha^2)A_{14} + (\alpha^3 + \alpha^2)A_{15} + \\
& + (\alpha + 1)A_{16} + (\alpha^2 + \alpha + 1)A_{17} + (\alpha^4 + \alpha^2 + 1)A_{18} + (\alpha^4)A_{19} + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1)A_{20} + (\alpha^4 + \alpha^3 + \alpha + 1)A_{21} + \\
& + (\alpha^4 + \alpha^2 + \alpha)A_{22} + (\alpha^4 + \alpha^2 + 1)A_{23} \\
f_{26} := & B_{26} + (\alpha^4 + \alpha + 1)A_1 + (\alpha^3 + \alpha^2 + \alpha)A_2 + (\alpha^2 + 1)A_3 + \\
& + (\alpha^4 + \alpha^2 + 1)A_4 + (\alpha^3 + \alpha^2 + 1)A_5 + (\alpha^4 + \alpha^2 + \alpha)A_6 + (\alpha^2)A_7 + \\
& + (\alpha^2 + 1)A_8 + (\alpha + 1)A_9 + (\alpha^4 + \alpha^3 + \alpha^2)A_{10} + (\alpha^4 + \alpha + 1)A_{11} + \\
& + (\alpha^2 + \alpha + 1)A_{12} + (\alpha^4 + \alpha + 1)A_{13} + (\alpha^4 + 1)A_{14} + (\alpha^4 + \alpha)A_{15} + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha)A_{16} + (\alpha^4 + \alpha^3 + \alpha)A_{17} + (\alpha + 1)A_{18} + \\
& + (\alpha^3)A_{19} + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1)A_{20} + (\alpha + 1)A_{21} + (\alpha^4 + 1)A_{22} + \\
& + (\alpha^3 + \alpha^2 + \alpha + 1)A_{23} \\
f_{27} := & B_{27} + (\alpha + 1)A_1 + (\alpha^4 + \alpha^3 + \alpha^2 + 1)A_2 + (\alpha^4 + 1)A_3 + \\
& + (\alpha^4 + \alpha^3 + 1)A_4 + (\alpha^3)A_5 + (\alpha^2 + \alpha)A_6 + (\alpha^4 + \alpha^3 + \alpha^2)A_7 + \\
& + (\alpha^3)A_8 + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha)A_9 + A_{10} + (\alpha^4 + \alpha^3 + \alpha)A_{11} + \\
& + (\alpha^3 + \alpha^2 + 1)A_{12} + (\alpha^4 + \alpha^2 + \alpha + 1)A_{13} + (\alpha^3 + \alpha + 1)A_{14} + \\
& + (\alpha^3)A_{15} + (\alpha^4 + \alpha + 1)A_{16} + (\alpha^3 + 1)A_{17} + (\alpha^4 + \alpha)A_{18} + \\
& + (\alpha^4 + \alpha^3 + \alpha^2)A_{19} + (\alpha^4 + \alpha^2 + \alpha)A_{20} + (\alpha)A_{21} + \\
& + (\alpha^4 + \alpha^2 + \alpha + 1)A_{22} + (\alpha^2 + \alpha)A_{23} \\
f_{28} := & B_{28} + (\alpha^4 + \alpha^3 + \alpha^2 + 1)A_1 + (\alpha^3 + \alpha^2 + \alpha)A_2 + (\alpha^4 + \alpha^3 + \alpha)A_3 + \\
& + (\alpha^3 + \alpha + 1)A_4 + (\alpha^3)A_5 + (\alpha^2 + \alpha + 1)A_6 + (\alpha)A_7 + \\
& + (\alpha^2 + \alpha + 1)A_8 + (\alpha^2 + \alpha)A_9 + (\alpha^3)A_{10} + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha)A_{11} + \\
& + (\alpha^4 + \alpha^2 + \alpha)A_{12} + (\alpha^3 + \alpha^2)A_{13} + (\alpha^2)A_{14} + (\alpha^4 + \alpha + 1)A_{15} + \\
& + (\alpha + 1)A_{16} + (\alpha^2 + \alpha)A_{17} + (\alpha^4 + \alpha^3 + \alpha + 1)A_{18} + \\
& + (\alpha^4 + \alpha^2 + 1)A_{19} + (\alpha^4)A_{20} + (\alpha^2 + \alpha + 1)A_{21} + \\
& + (\alpha^3 + \alpha)A_{22} + (\alpha)A_{23} \\
f_{29} := & B_{29} + (\alpha^3 + \alpha + 1)A_1 + (\alpha^2 + 1)A_2 + (\alpha^4)A_3 + (\alpha^3 + \alpha^2 + \alpha + 1)A_4 + \\
& + (\alpha^2)A_5 + (\alpha^3 + \alpha^2 + 1)A_6 + (\alpha^4 + \alpha^3)A_7 + (\alpha^3 + \alpha + 1)A_8 + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha)A_9 + (\alpha^4 + \alpha^2 + \alpha + 1)A_{10} + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha)A_{11} + (\alpha^4 + \alpha^3 + \alpha)A_{12} + (\alpha^3 + \alpha)A_{13} +
\end{aligned}$$

$$\begin{aligned}
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha)A_{14} + (\alpha^3 + \alpha^2)A_{15} + (\alpha^3 + 1)A_{16} + \\
& + (\alpha^4 + \alpha + 1)A_{17} + (\alpha^3)A_{18} + (\alpha^2 + 1)A_{19} + (\alpha^4 + 1)A_{20} + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1)A_{21} + (\alpha^4 + \alpha^3 + \alpha^2)A_{22} + (\alpha^4 + \alpha^3 + \alpha)A_{23} \\
f_{30} := & B_{30} + (\alpha^4)A_1 + (\alpha)A_2 + (\alpha^2 + \alpha + 1)A_3 + (\alpha)A_4 + (\alpha^3 + \alpha^2 + \alpha)A_5 + \\
& + (\alpha^4 + \alpha^3)A_6 + (\alpha)A_7 + (\alpha^4 + \alpha)A_8 + (\alpha^3 + \alpha^2 + \alpha + 1)A_9 + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + 1)A_{10} + (\alpha^4 + \alpha)A_{11} + (\alpha^3 + \alpha^2 + \alpha + 1)A_{12} + \\
& + (\alpha)A_{13} + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha)A_{14} + (\alpha^4 + \alpha^3 + 1)A_{15} + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1)A_{16} + (\alpha^2 + \alpha + 1)A_{17} + \\
& + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1)A_{18} + (\alpha^3 + \alpha)A_{19} + (\alpha^4 + \alpha^2)A_{20} + \\
& + (\alpha^4)A_{21} + A_{22} + (\alpha^4 + \alpha)A_{23} \\
f_{31} := & B_{31} + (\alpha^4 + \alpha^3 + 1)A_1 + (\alpha^4 + \alpha^2 + \alpha)A_2 + (\alpha^4 + \alpha^2 + \alpha + 1)A_3 + \\
& + (\alpha^3 + \alpha + 1)A_4 + (\alpha^4 + \alpha^3 + \alpha)A_5 + (\alpha^2 + \alpha)A_6 + (\alpha^2)A_7 + \\
& + (\alpha^3 + 1)A_8 + (\alpha^4 + \alpha^3 + \alpha^2 + 1)A_9 + (\alpha)A_{10} + (\alpha^3 + \alpha)A_{11} + \\
& + (\alpha^4 + \alpha + 1)A_{12} + (\alpha^3)A_{13} + (\alpha^4 + \alpha^2)A_{14} + (\alpha^4 + \alpha^3 + \alpha^2)A_{15} + \\
& + (\alpha^3 + \alpha^2 + 1)A_{16} + (\alpha^4 + \alpha^3 + \alpha + 1)A_{17} + (\alpha^4 + \alpha^2 + \alpha)A_{18} + \\
& + (\alpha^3 + \alpha)A_{19} + (\alpha^3 + \alpha + 1)A_{20} + (\alpha^3 + \alpha^2)A_{21} + \\
& + (\alpha^3 + \alpha^2 + 1)A_{22} + (\alpha^2 + 1)A_{23}
\end{aligned}$$

Having in mind the failure in computing the Gröbner basis of the previous example, we tried to bypass the problem. Remember that the circuit is linear, and the biggest part of the computation is the reduction of the field equations. Fortunately, we may apply the results of Section 2.5, so only the 992 polynomials from the circuit are needed. Our PC took 30.1 s to compute the 992 polynomials of the Gröbner basis \mathcal{G} of this smaller ideal. All the f_i 's reduce to zero with respect to \mathcal{G} , from which we may conclude that this instance of the circuit works.

4.2 General Case

In this section we describe the circuit with a system of difference equations and try to deduce the correctness from these. Actually, there are *two* sets of equations that control the behaviour of the circuit: before turning the switch and after.

Notational Remark. In order to not carry the 2 in the formulas, in this section we will suppose that there are t check symbols (instead of $2t$). Moreover, we suppose that the switch is activated after k symbols, so that the time indexes from 0 to $k - 1$ refer to the circuit before turning down the switch. The total number of cycles required is $n = k + t$ (time indexes from 0 to $n - 1$).

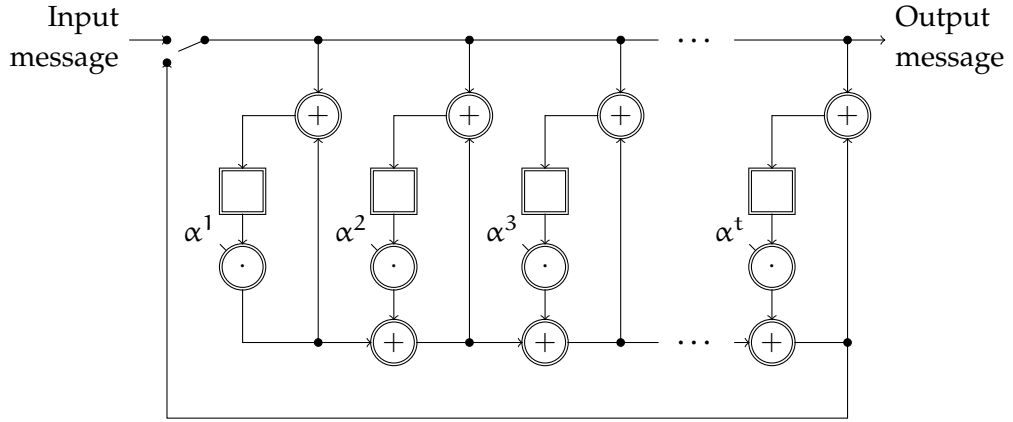


Figure 4.4: The same of Figure 1.5.

Figure 4.4 is the same of Figure 1.5, which we recall here for convenience. Let $u(s)$ ($s \in \mathbb{N} \setminus \{0\}$) be the sequence of the inputs, $y(s)$ the one of the outputs, and $\mathbf{x}(s) = (x_1(s), \dots, x_t(s))$ the content of the t registers, which will represent the state of the circuit.

Let us begin from the first phase of the circuit, i.e. when the switch is in the upper position. The equation for the output is easy:

$$y(s) = u(s)$$

because the encoding is systematic. Let us focus on the state; the i -th register at time $s + 1$ receives the result of the adder in the first row, which adds the input and the result of

$$\alpha x_1(s) + \alpha^2 x_2(s) + \dots + \alpha^i x_i(s). \quad (4.1)$$

So we have

$$\mathbf{x}(s + 1) = A\mathbf{x}(s) + B u(s),$$

where

$$A := \begin{pmatrix} \alpha & 0 & \dots & \dots & 0 \\ \alpha & \alpha^2 & 0 & & \vdots \\ \vdots & \vdots & \alpha^3 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ \alpha & \alpha^2 & \alpha^3 & \dots & \alpha^t \end{pmatrix} \quad \text{and} \quad B := \begin{pmatrix} 1 \\ 1 \\ \vdots \\ \vdots \\ 1 \end{pmatrix}.$$

Then there is the second phase, when the switch is turned down. The input $u(s)$ is no more relevant; the output is computed from the content of the registers with

$$y(s) = \alpha x_1(s) + \alpha^2 x_2(s) + \dots + \alpha^t x_t(s)$$

which we may write as

$$\mathbf{y}(s) = \mathbf{C}^T \mathbf{x}(s), \quad \text{with } \mathbf{C} := \begin{pmatrix} \alpha \\ \alpha^2 \\ \vdots \\ \vdots \\ \alpha^t \end{pmatrix}.$$

The i -th register is updated with the sum of this output and again the result of Equation (4.1), thus we have

$$\mathbf{x}(s+1) = \tilde{\mathbf{A}}\mathbf{x}(s),$$

where (remember that we are working in fields with characteristic 2)

$$\tilde{\mathbf{A}} := \mathbf{A} + \mathbf{B}\mathbf{C}^T = \begin{pmatrix} 0 & \alpha^2 & \alpha^3 & \dots & \alpha^t \\ 0 & 0 & \alpha^3 & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \alpha^t \\ 0 & 0 & \dots & \dots & 0 \end{pmatrix}.$$

In conclusion, the systems of difference equations that model our encoder are

$$\begin{cases} \mathbf{x}(s+1) = \mathbf{A}\mathbf{x}(s) + \mathbf{B}\mathbf{u}(s) \\ \mathbf{y}(s) = \mathbf{u}(s) \end{cases} \quad \text{and} \quad \begin{cases} \mathbf{x}(s+1) = \tilde{\mathbf{A}}\mathbf{x}(s) \\ \mathbf{y}(s) = \mathbf{C}^T \mathbf{x}(s) \end{cases} \quad (4.2)$$

where \mathbf{A} , \mathbf{B} , \mathbf{C} and $\tilde{\mathbf{A}}$ are defined above. The first system is valid for $s = 0, \dots, k-1$ and the second from $s = k$ onward.

4.2.1 Applying the Z-Transform

We will now try to manipulate the Z-transforms of Equations (4.2) in order to reduce the circuit verification to an algebraic relation.

Remember that, if $m(Z) = m_0 + \dots + m_{k-1}Z^{k-1}$ is the message polynomial and $g(Z)$ is the generator polynomial of the code, the encoder computes $m(Z) \cdot Z^t \pmod{g(Z)}$. How does $m(Z) \cdot Z^t$ relate to the input sequence $u(s)$? If

$$\mathcal{U}(Z) = u(0) + u(1)\frac{1}{Z} + u(2)\frac{1}{Z^2} + \dots$$

is the Z-transform of $u(s)$, in which we may suppose $u(\ell) = 0$ for $\ell \geq k$ (because we are dealing with polynomials), then the polynomial $U(Z) := Z^{n-1}\mathcal{U}(Z)$ is the input polynomial, that is to say, the circuit computes $U(Z) \pmod{g(Z)}$ if correct.

Now, linearity allows us to verify only inputs of the form

$$\delta_h(s) = \begin{cases} 1 & \text{if } s = h \\ 0 & \text{otherwise} \end{cases}$$

for $h \leq k-1$; their Z-transform is $1/Z^h$ and they represent the polynomial Z^{n-1-h} . A quick computation from Equations (4.2) tells us that, when the switch is activated, the registers $\mathbf{x}(k-1)$ contain

$$\mathbf{x}(k-1) = A^{k-1-h} \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} =: \mathbf{x}_0.$$

This is the starting point for the second system. The Z-transform of this system (as seen in Section 3.3) is

$$\mathbf{y}(Z) = ZC^T(ZI - \tilde{A})^{-1}\mathbf{x}_0.$$

The first t terms of the series $\mathbf{y}(Z)$ represent the coefficients of the output polynomial; we expect it to have degree $t-1$, so we may say that the polynomial $Y(Z) := Z^{t-1}\mathbf{y}(Z)$ (where the series $\mathbf{y}(Z)$ is truncated at degree $t-1$, but this won't be a problem, as we will see in a moment) is the output polynomial, that is to say,

$$Y(Z) \equiv U(Z) \pmod{g(Z)} \quad (4.3)$$

or, in other words, $Y(\alpha^i) + U(\alpha^i) = 0$ for all $i = 1, \dots, t$.

Let's write Equation (4.3) more explicitly. First of all, $\det(ZI - \tilde{A}) = Z^t$, so we may write

$$(ZI - \tilde{A})^{-1} = \frac{1}{Z^t} \tilde{M}(Z)$$

where $\tilde{M}(Z) \in \mathcal{M}_t(\mathbb{K}[Z])$ is a suitable matrix with *polynomial* entries. Thus

$$\begin{aligned} Y(Z) &= Z^{t-1}\mathbf{y}(Z) = \\ &= Z^{t-1}ZC^T \frac{1}{Z^t} \tilde{M}(Z) A^{k-1-h} \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \\ &= C^T \tilde{M}(Z) A^{k-1-h} \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}. \end{aligned}$$

Notice that the last expression is polynomial—it doesn't involve series. In conclusion, the verification test reduces to check whether the polynomial

$$Z^{n-1-h} + C^T \tilde{M}(Z) A^{k-1-h} \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

vanishes on all α^i , for $i = 1, \dots, t$.

We performed some tests with the Sage software for fixed values of n , h and t and they all gave the desired result. However, a general proof involves manipulation of symbolic expressions, which is currently unavailable on the computer algebra systems (as far as we know).

4.2.2 Equivalence to the Standard Encoder

One of the possible ways to verify a circuit is to check whether it is equivalent to another circuit, whose correctness is established. In this case, we have the standard encoder (see Figure 1.1). Let us write the difference equations for it. We will use $u(s)$ and $y(s)$ as the input and output sequences respectively, and $\mathbf{w}(s) = (w_1(s), \dots, w_t(s))$ as the registers. Even in this circuit we distinguish two phases: before and after activating the switch.

Before the switch, the output is linked to the input, so we have $y(s) = u(s)$. In the meantime, the registers are loaded. If the generator polynomial is $g(X) = g_0 + g_1X + \dots + g_{t-1}X^{t-1} + X^t$, then it is easy to show that

$$\mathbf{w}(s+1) = M_g \mathbf{w}(s) + G u(s),$$

where

$$M_g := \begin{pmatrix} 0 & \dots & \dots & 0 & g_0 \\ 1 & 0 & & \vdots & g_1 \\ 0 & 1 & \ddots & \vdots & g_2 \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 1 & g_{t-1} \end{pmatrix} \quad (4.4)$$

is the companion matrix of g , and

$$G := \begin{pmatrix} g_0 \\ \vdots \\ g_{t-1} \end{pmatrix}.$$

After the switch, the input is cut off, and the output is just a concatenation of the content of the registers. In particular, we have

$$y(s) = w_t(s) = D^T \mathbf{w}(s) \quad \text{and} \quad \mathbf{w}(s+1) = N \mathbf{w}(s)$$

where

$$\mathbf{N} := \begin{pmatrix} 0 & \cdots & \cdots & \cdots & 0 \\ 1 & 0 & & & \vdots \\ 0 & 1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{D} := \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{pmatrix}.$$

We want to prove that the combined encoder is equivalent to the standard one. Notice that we can't apply directly the results of Section 3.4, because of the switch—our circuits are not purely linear. However, let us look at the behaviour of the two encoders a little more in depth.

When the switch is activated, the standard encoder contains its output in its registers (in fact, its phase two just concatenates the values in the registers). Instead, the combined encoder requires its phase two in order to fully compute its output.

One possible interpretation is that the combined encoder does its computations during phase one using the “wrong” basis and it needs the phase two in order to change basis, whereas the standard encoder uses the “right” basis from the beginning.

Translating the previous remarks in terms of the sequences of inputs, states and outputs of the circuit, we have

$$w_i(k-1) = y(n-i), \quad \text{for } i = 1, \dots, t$$

(there is an $n-i$ because the standard encoder actually outputs the content of w_t as the first symbols and continues until w_1). We are going to use the system that describes phase two of the combined encoder in order to write the outputs $y(n-i)$ as functions of the states $x_j(k-1)$, thus obtaining a candidate matrix \mathbf{T} for the equivalence test. From now on, we will write simply $\mathbf{w} = (w_i)_{i=1, \dots, t}$ and $\mathbf{x} = (x_j)_{j=1, \dots, t}$ instead of $\mathbf{w}(k-1)$ and $\mathbf{x}(k-1)$ respectively.

By inspection, it is easy to show that

$$\mathbf{y}(k+\ell) = \mathbf{C}^T \tilde{\mathbf{A}}^\ell \mathbf{x}$$

for all $\ell = 0, \dots, t-1$. So, defining \mathbf{S} as the matrix whose i -th row is $\mathbf{C}^T \tilde{\mathbf{A}}^{t-i}$, we have that $\mathbf{w} = \mathbf{S}\mathbf{x}$, thus our candidate change of coordinates is $\mathbf{T} := \mathbf{S}^{-1}$. What we have to do now is to verify that

$$\mathbf{M}_g = \mathbf{T}^{-1} \mathbf{A} \mathbf{T} \quad \text{and} \quad \mathbf{G} = \mathbf{T}^{-1} \mathbf{B}.$$

Actually, the structure of \mathbf{S} is more easily exploited if we raise to a more general point of view, that is to say, we will now suppose that the roots of $g(Z)$

are r_1, \dots, r_t , not necessarily consecutive powers of α . For $\ell = 1, \dots, t$ let us call S_ℓ the $\ell \times \ell$ matrix obtained using only the roots r_1, \dots, r_ℓ . We want to find a relation between S_t and S_{t-1} in order to try an inductive proof. It is not difficult to recognize the pattern:

$$S_t = \left(\begin{array}{c|c} \mathbf{0}_{t-1}^T & \\ \hline S_{t-1} & \begin{array}{c} r_t S_{t-1} \mathbf{1}_{t-1} \\ \hline r_t \end{array} \end{array} \right)$$

where $\mathbf{0}_t$ and $\mathbf{1}_t$ are the column vectors of t zeros and t ones respectively.

Example 4.1. For $t = 4$, the matrix S_4 is

$$\begin{pmatrix} 0 & 0 & 0 & r_1 r_2 r_3 r_4 \\ 0 & 0 & r_1 r_2 r_3 & r_1 r_2 r_4 + r_1 r_3 r_4 + r_2 r_3 r_4 \\ 0 & r_1 r_2 & r_1 r_3 + r_2 r_3 & r_1 r_4 + r_2 r_4 + r_3 r_4 \\ r_1 & r_2 & r_3 & r_4 \end{pmatrix}.$$

In fact, a way to build S_t is: for any non-empty subset $J = \{j_1, \dots, j_i\}$ of $\{1, \dots, t\}$ (with $\#(J) = i$ and $j_1 < \dots < j_i$) consider the term $\prod_{j \in J} r_j$. Add this term to the i -th row from the bottom in the j_i -th column, i.e. the i -th row from the bottom contains all the subsets of cardinality i , and the j -th column contains all the subsets whose maximum element is j .

Proposition 4.1. *Let*

$$g_{(t)}(Z) := (Z - r_1) \cdot \dots \cdot (Z - r_t) = Z^t + g_{t-1}^{(t)} Z^{t-1} + \dots + g_0^{(t)}$$

(we assume, as always, that we are working in a characteristic 2 field, so that we don't care about minus signs) and let

$$G_t = \begin{pmatrix} g_0^{(t)} \\ \vdots \\ g_{t-1}^{(t)} \end{pmatrix}.$$

Then, for any t , $G_t = S_t \mathbf{1}_t$.

Proof. We prove the proposition by induction on t .

t = 1. In this case $S_1 = r_1$ and $G_1 = r_1$, so we have obviously

$$r_1 = r_1 \cdot 1.$$

$t-1 \Rightarrow t$. A little computation with the elementary symmetric functions give the following relations between the coefficients of $g_{(t)}$ and the ones of $g_{(t-1)}$:

$$\begin{aligned} g_{t-1}^{(t)} &= g_{t-2}^{(t-1)} + r_t \\ g_{t-2}^{(t)} &= g_{t-3}^{(t-1)} + r_t g_{t-2}^{(t-1)} \\ g_{t-3}^{(t)} &= g_{t-4}^{(t-1)} + r_t g_{t-3}^{(t-1)} \\ &\vdots \\ g_0^{(t)} &= r_t g_0^{(t-1)}. \end{aligned} \quad (4.5)$$

Now, our inductive hypothesis allows us to write

$$S_t = \left(\begin{array}{c|c} \mathbf{0}_{t-1}^T & r_t \mathbf{G}_{t-1} \\ \hline S_{t-1} & \\ \hline & r_t \end{array} \right).$$

We will denote by $G[i]$ the i -th component of the vector G (because there are already too many sub- and superscripts). Let us compute $S_t \mathbf{1}_t$ one component at a time. The first row gives us

$$r_t \mathbf{G}_{t-1}[1] = r_t g_0^{(t-1)} = g_0^{(t)} = G_t[1].$$

Then, for $i = 2, \dots, t-1$, we have

$$\begin{aligned} (S_t \mathbf{1}_t)[i] &= (S_{t-1} \mathbf{1}_{t-1})[i-1] + r_t \mathbf{G}_{t-1}[i] = \\ &= G_{t-1}[i-1] + r_t \mathbf{G}_{t-1}[i] = \\ &= g_{i-2}^{(t-1)} + r_t g_{i-1}^{(t-1)} = g_{i-1}^{(t)} = G_t[i]. \end{aligned}$$

The last element is just

$$G_{t-1}[t-1] + r_t = g_{t-2}^{(t-1)} + r_t = g_{t-1}^{(t)} = G_t[t]$$

and the proof is complete. \square

Proposition 4.2. *Following the notation of the previous proposition, let $(M_g)_t$ be the companion matrix of $g_{(t)}$, written as in (4.4). Let also*

$$C_t := \begin{pmatrix} r_1 \\ \vdots \\ r_t \end{pmatrix} = \begin{pmatrix} C_{t-1} \\ r_t \end{pmatrix}, \quad A_t := \begin{pmatrix} r_1 & 0 & \cdots & 0 \\ r_1 & r_2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ r_1 & r_2 & \cdots & r_t \end{pmatrix} = \begin{pmatrix} A_{t-1} & | & \mathbf{0}_{t-1} \\ \hline & & C_t^T \end{pmatrix}.$$

Then, for any t , $S_t A_t = (M_g)_t S_t$.

The first term is a little more tricky. First of all, M_1 can be rewritten as

$$\left(\begin{array}{c|c} \mathbf{0}_t^T & \\ \hline I_{t-1} & G_{t-1} \end{array} \right).$$

With that in mind, we get

$$\begin{aligned} M_1 S_t &= \left(\begin{array}{c|c} \mathbf{0}_t^T & \\ \hline I_{t-1} & G_{t-1} \end{array} \right) \left(\begin{array}{c|c} \mathbf{0}_{t-1}^T & \\ \hline S_{t-1} & \begin{array}{c} r_t G_{t-1} \\ \hline r_t \end{array} \end{array} \right) = \\ &= \left(\begin{array}{c|c} \mathbf{0}_t^T & \\ \hline (M_g)_{t-1} S_{t-1} & (\star) \end{array} \right). \end{aligned}$$

By inductive hypothesis $(M_g)_{t-1} S_{t-1} = S_{t-1} A_{t-1}$, so we're done if we prove that the column (\star) is $\mathbf{0}_{t-1}$. Who is the element (i, t) of $M_1 S_t$, for $i = 2, \dots, t$? With the same notation of Proposition 4.1 we have

$$(M_1 S_t)[i, t] = r_t G_{t-1}[i-1] + g_{i-2}^{(t-1)} r_t.$$

But $G_{t-1}[i-1] = g_{i-2}^{(t-1)}$, and we are working in characteristic 2, so

$$(M_1 S_t)[i, t] = 0$$

and that concludes the proof. \square

The previous propositions tell us that the circuit works not only for RS codes, but with any polynomials with roots r_1, \dots, r_t —that is to say, both the standard and the combined encoders compute the coefficients of the remainder modulo the polynomial $(Z - r_1) \cdot \dots \cdot (Z - r_t)$. Moreover, we never used the fact that the roots r_1, \dots, r_t are distinct.

The only problem rises when at least one of the roots is 0: in that case the matrix S is no longer invertible, and the proof of correctness doesn't work. We could try to find another way to prove that the circuit works also in this case; however, there is an easier workaround. We can write the polynomial $g(Z)$ as

$$g(Z) = Z^s h(Z)$$

where s is the multiplicity of 0 as a root of g and $h(0) \neq 0$. Then

$$m(Z) \cdot Z^t \pmod{g(Z)} = Z^s (m(Z) \cdot Z^{t-s} \pmod{h(Z)})$$

and we can use the circuit to compute $m(Z) \cdot Z^{t-s} \pmod{h(Z)}$. The Z^s term just shifts the output adding s zeros as the last symbols.

For our work, we are satisfied with a proof of correctness when none of the r_i 's is zero—after all, this is the case of the RS encoder, where $r_i = \alpha^i$. In the future we hope to extend our proof so that it will include the circuits whose g 's could have 0 as one of their roots.

Conclusions and Further Developments

In this work we showed several algebraic techniques that can be used to perform the formal verification of a circuit. We have seen different structures (polynomials, difference equations) into which the information on the behaviour of the circuit can be encoded in order to make it easier to manipulate.

The polynomial method, unlike the others described in this thesis, does not need the circuit to be linear, thus allowing to prove correctness for general circuits. Its major drawback is the high computational cost, because it requires a huge number of polynomials in a lot of variables for complex circuits.

The module method is less expensive than the polynomial method, but it works only for linear circuits. Moreover, in this thesis we proved that the module method gives us a sufficient condition to verify the correctness of a circuit. It would be interesting to investigate whether this condition is also necessary.

As we have already mentioned, Lv, Kalla and Enescu in [7] suggest some improvements that cut the high cost of their method. They choose a suitable term ordering in order to avoid the computation of a Gröbner basis and perform an efficient F_4 -style polynomial reduction. Can these improvements be adapted for our generalization of their method? We would like to answer to this question in the future.

As far as the difference equations are concerned, we just showed the main techniques that involve this mathematical tool. Actually, we never used the fact that the equations arise from a circuit—these techniques may be used to analyse problems which can be described by means of recursive relations. In particular, we have seen that this kind of relations may be encoded into matrices, and manipulating these matrices may lead to a formal proof of correctness. Unfortunately, there's no much we can do from a computational point of view, because at the moment there is no tool that manages symbolic expression. Our hope is that in the future new and more powerful computer algebra systems will handle these expressions.

Throughout this work we focused on a specific circuit, which is the combined RS encoder/syndrome generator proposed by Fettweis and Hassner in [2]. We used it as a starting point for the development of the verification methods. We showed that it still works if we use generic elements r_1, \dots, r_t instead of the roots α, \dots, α^t (of course it won't encode an RS word anymore) and we even allowed multiple roots. There is a lot more that can be studied of this circuit. For example, can we slightly modify it in order to make it work for fields of characteristic different from 2? We leave this as an open question for further analyses.

Appendix A

Error-Correcting Codes

In this appendix we recall some basic facts about the theory of error-correcting codes. Our short exposition is not meant to be complete; there are plenty of books for the interested reader.

In the last decades the importance of storing and transmitting information has grown more and more, especially in digital form. Soon a problem arose: environmental noises may corrupt data, even if sent by a most shielded channel; damages of the storage device (such as scratches on a CD) may cause data loss. The solution was to encode information in a way that allows to retrieve as much as possible of the original data: this led to the idea of error-correcting codes.

A.1 Generality of Error-Correcting Codes

Let's begin with the basic definition.

Definition A.1. An *alphabet* is a finite set whose elements are called *symbols*. A *word* is a finite sequence of symbols. A *code* is a (not necessarily finite) set of words.

Our alphabet will always be a (finite) field \mathbb{K} , unless otherwise stated. Moreover, we will deal only with *block codes*, that is to say, our words will always have a finite and fixed length, say n . Thus, our codes will be subsets of \mathbb{K}^n . We will always assume that n and $\text{char}(\mathbb{K})$ are coprime.

Definition A.2. The *Hamming distance* between two words $\mathbf{v} = (v_1, \dots, v_n)$ and $\mathbf{w} = (w_1, \dots, w_n)$ is defined as

$$d(\mathbf{v}, \mathbf{w}) := \#\{i = 1, \dots, n \mid v_i \neq w_i\}.$$

The *weight* of a word \mathbf{w} is its distance from the zero word, i.e. $\text{wt}(\mathbf{w}) := d(\mathbf{w}, \mathbf{0})$. Given a code C , its *distance* is the minimum of the distances between its words:

$$d := \min_{\substack{\mathbf{v}, \mathbf{w} \in C \\ \mathbf{v} \neq \mathbf{w}}} d(\mathbf{v}, \mathbf{w}).$$

Definition A.3. Let C be a code, \mathbf{v} the sent word and \mathbf{w} the received word. The *error* between \mathbf{v} and \mathbf{w} is defined as $\mathbf{e} = \mathbf{w} - \mathbf{v}$. The code *detects* the error if $\mathbf{v} + \mathbf{e} \notin C$ for all $\mathbf{v} \in C$. The code *corrects* the error if $d(\mathbf{w}, \mathbf{v} + \mathbf{e}) > d(\mathbf{v}, \mathbf{v} + \mathbf{e})$ for all $\mathbf{v}, \mathbf{w} \in C$ with $\mathbf{v} \neq \mathbf{w}$, in other words, if \mathbf{v} is the nearest word to $\mathbf{v} + \mathbf{e}$ among all words in C .

Proposition A.4. Let C be a code with distance d .

- The code can detect at most $d - 1$ errors in a word. In other words, the code detects all errors \mathbf{e} with $\text{wt}(\mathbf{e}) \leq d - 1$ and there exists an error $\bar{\mathbf{e}}$ with $\text{wt}(\bar{\mathbf{e}}) = d$ which is not detected by the code.
- The code can correct $\lfloor \frac{d-1}{2} \rfloor$ errors in a word. In other words, the code corrects all errors \mathbf{e} with $\text{wt}(\mathbf{e}) \leq \lfloor \frac{d-1}{2} \rfloor$.

A.2 Some Examples of Error-Correcting Codes

How can we actually perform the operations of error detection and error correction? The code would have some structure which allows us to decode properly the received message. *Algebraic* structures, such as vector spaces or ideals in suitable rings, have good properties in order to define codes with great capability of error correction.

A.2.1 Linear Codes

The simplest structure that we are going to describe is that of vector space.

Definition A.5. A *linear code* of dimension k is a k -dimensional vector subspace of \mathbb{K}^n .

Let $\mathcal{B} := (\mathbf{v}_1, \dots, \mathbf{v}_m)$ be a basis of the linear code C . The matrix

$$G := \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_m \end{pmatrix} \in \mathcal{M}_{m \times n}(\mathbb{K}),$$

whose rows are the elements of \mathcal{B} , is called *generator matrix* of the code. In fact, G defines an isomorphism between \mathbb{K}^m and C , sending $\mathbf{u} \in \mathbb{K}^m$ to $\mathbf{w} := \mathbf{u} \cdot G \in C$.

Definition A.6. A *parity-check matrix* for a k -dimensional code $C \subseteq \mathbb{K}^n$ is a matrix $P \in \mathcal{M}_{n \times (n-k)}(\mathbb{K})$ such that $\mathbf{u} \in \mathbb{K}^n$ belongs to C if and only if $\mathbf{u} \cdot P = \mathbf{0}$.

We can always find a parity-check matrix for C : it suffices to consider a basis $(\mathbf{w}_1, \dots, \mathbf{w}_{n-k})$ of C^\perp , the orthogonal complement of C , and let

$$P := \left(\begin{array}{c|c|c|c} \mathbf{w}_1 & \mathbf{w}_2 & \dots & \mathbf{w}_{n-k} \end{array} \right).$$

Definition A.7. Let P be a parity-check matrix for a code C . The *syndrome* of a word $\mathbf{u} \in \mathbb{K}^n$ is the vector $\mathbf{s} := \mathbf{u} \cdot P \in \mathbb{K}^{n-k}$.

So, a word $\mathbf{v} \in \mathbb{K}^n$ belongs to the code if and only if its syndrome is $\mathbf{0}$. Actually, syndromes are useful also to correct errors. Fix a syndrome \mathbf{s} and consider the word $\mathbf{v}_s \in \mathbb{K}^n$ which has the smallest weight among the ones whose syndrome is \mathbf{s} . Now suppose that the received word \mathbf{w} has \mathbf{s} as syndrome. Linearity implies that $\mathbf{w} - \mathbf{v}_s \in C$ and this word is the closest to \mathbf{w} among all words of C , by definition of \mathbf{v}_s . Thus $\mathbf{w} - \mathbf{v}_s$ is the most likely word that has been sent.

A.2.2 Cyclic Codes

In this subsection we add some extra algebraic structure to a linear code.

Definition A.8. A linear code C is called *cyclic* if

$$(c_1, \dots, c_n) \in C \Leftrightarrow (c_n, c_1, \dots, c_{n-1}) \in C$$

for all $\mathbf{c} = (c_1, \dots, c_n) \in \mathbb{K}^n$.

Where does the extra structure come from? Consider the \mathbb{K} -vector space isomorphism

$$\begin{array}{ccc} \mathbb{K}^n & \longrightarrow & \mathbb{K}[X]/(X^n - 1) \\ (c_0, \dots, c_{n-1}) & \longmapsto & [c_0 + c_1X + \dots + c_{n-1}X^{n-1}], \end{array}$$

where as usual $[p(X)]$ denotes the equivalence class modulo $(X^n - 1)$. Notice that $(c_0, \dots, c_{n-1}) \mapsto (c_{n-1}, c_0, \dots, c_{n-2})$ translates into multiplication by $[X]$ when read in $\mathbb{K}[X]/(X^n - 1)$.

Lemma A.9. A code $C \subseteq \mathbb{K}^n$ is cyclic if and only if it is an ideal of $\mathbb{K}[X]/(X^n - 1)$.

Definition A.10. Let C be a cyclic code. Notice that $\mathbb{K}[X]/(X^n - 1)$ is a principal ideal ring, so there exists $g(X) \in \mathbb{K}[X]$ such that $C = ([g(X)])$ and we can choose g such that $\deg(g) < n$. The polynomial $g(X)$ is called *generator polynomial* of C .

Proposition A.11. *The factors of $X^n - 1$ are exactly the possible generator polynomials for a cyclic code in \mathbb{K}^n .*

It is easy to show that, for a cyclic code C generated by $g(X)$, the matrix whose rows are the coefficients of $X^i \pmod{g(X)}$ (for $i = 0, \dots, n-1$) is a parity-check matrix for C . The syndrome for a word $w(X)$ with respect to this matrix is simply $w(X) \pmod{g(X)}$.

Actually, there is another tool which we can use to detect errors. If $g(X)$ is the generator polynomial of a cyclic code C , there exists $h(X)$ such that $g(X)h(X) = X^n - 1$. In particular, $c(X) \in C$ if and only if $c(X)h(X) = 0$ in $\mathbb{K}[X]/(X^n - 1)$. The polynomial $h(X)$ is called *check polynomial* for the code C .

The following proposition gives us another description of a cyclic code C . Recall that if $n = q^r - 1$ for some r (with q power of a prime), the roots of $X^n - 1$ are exactly the elements of \mathbb{F}_{q^r} (and are, therefore, distinct).

Proposition A.12. *Let $g(X) \in \mathbb{F}_q[X]$ be the generator polynomial of a cyclic code C such that the length of the codewords is $n = q^r - 1$ for some r and let $\beta_1, \dots, \beta_t \in \mathbb{F}_{q^r}$ be its roots. Then*

$$C = \{c(X) \in \mathbb{F}_q[X] \mid c(\beta_1) = \dots = c(\beta_t) = 0 \text{ in } \mathbb{F}_{q^r}\}.$$

It follows that syndromes may be computed by evaluation of the received word.

A.2.3 BCH Codes

The last proposition leads us to the definition of a particular class of codes.

Definition A.13. A (*primitive*) *BCH code*^[1] of *designed distance* δ is a cyclic code of length $n = q^r - 1$ over \mathbb{F}_q (for some r) whose generator polynomial is the least common multiple of the minimal polynomials of $\beta^\ell, \beta^{\ell+1}, \dots, \beta^{\ell+\delta-2}$ for some ℓ , where β is a primitive n -th root of unity in \mathbb{F}_{q^r} .

It is easy to show that a BCH code of designed distance δ has distance $d \geq \delta$ (hence the name of *designed distance*).

Decoding of a BCH code requires the computation of two specific polynomials, one that finds errors and the other that tells how to correct them. Consider a BCH code of length $n = q^r - 1$ over \mathbb{F}_q with $\delta = 2t + 1$ and let β be a primitive element of \mathbb{F}_{q^r} . Let $c(X)$ be the transmitted codeword and $w(X)$ the received one. The error between them is

$$e(X) := w(X) - c(X) = e_0 + e_1X + \dots + e_{n-1}X^{n-1}.$$

^[1]BCH stands for Bose, Chaudhuri and Hocquenghem who defined this class of codes in the late '50.

Let $M := \{i \mid e_i \neq 0\}$, i.e. the positions where an error occurred, and let $e := \#(M)$ be the number of errors (we are assuming $e \leq t$, otherwise we can't correct the error).

Definition A.14. The *error locator polynomial* is

$$\sigma(Z) := \prod_{i \in M} (1 - \beta^i Z).$$

The *error evaluator polynomial* is

$$\omega(Z) := \sum_{i \in M} \left(e_i \beta^i Z \prod_{j \in M \setminus \{i\}} (1 - \beta^j Z) \right).$$

If we know the polynomials $\sigma(Z)$ and $\omega(Z)$, we can detect and correct the error. As a matter of fact, an error occurred at i if and only if $\sigma(\beta^{-i}) = 0$ and, in that case, we can compute

$$e_i = -\frac{\omega(\beta^{-i})\beta^i}{\sigma'(\beta^{-i})}. \quad (\text{A.1})$$

There seems to be a problem: $\sigma(Z)$ and $\omega(Z)$ are defined in terms of the e_i 's, that is exactly what we are looking for. Notice that

$$\begin{aligned} \frac{\omega(Z)}{\sigma(Z)} &= \sum_{i \in M} \frac{e_i \beta^i Z}{1 - \beta^i Z} = \sum_{i \in M} e_i \sum_{k=1}^{\infty} (\beta^i Z)^k = \\ &= \sum_{k=1}^{\infty} Z^k \sum_{i \in M} e_i \beta^{ki} = \sum_{k=1}^{\infty} Z^k e(\beta^k) \end{aligned}$$

(all passages are done in the ring of formal power series over \mathbb{F}_{q^r}). Now, for $i = 1, \dots, 2t$ we have $e(\beta^i) = w(\beta^i)$, because $c(X)$ is a codeword (thus $c(\beta^i) = 0$). We know the first $2t$ terms of the power series expansion of $\omega(Z)/\sigma(Z)$, that is to say, we know that

$$\frac{\omega(Z)}{\sigma(Z)} \equiv \sum_{k=1}^{2t} w(\beta^k) Z^k \pmod{Z^{2t+1}}. \quad (\text{A.2})$$

If $s(Z) := \sum w(\beta^k) Z^k$ is the *syndrome polynomial*, we may rewrite Equation (A.2) as

$$s(Z)\sigma(Z) \equiv \omega(Z) \pmod{Z^{2t+1}}.$$

This is called *key equation* for BCH codes. This can be solved by brute force (writing the solution with unknown coefficients leads to a linear system), but more efficient methods are preferred, in particular the *Berlekamp-Massey Algorithm* or one based on the Extended Euclidean Algorithm.

A.3 Reed-Solomon Codes

It turns out that the simplest BCH codes, i.e. the ones with $n = q - 1$, have actually many useful applications.

Definition A.15. A *Reed-Solomon code* (or RS code) is a primitive BCH code of length $n = q - 1$ over \mathbb{F}_q .

The generator polynomial of an RS code with designed distance δ is

$$g(X) = \prod_{i=\ell}^{\ell+\delta-2} (X - \beta^i)$$

where β is a primitive element of \mathbb{F}_q .

Proposition A.16. An RS code with designed distance δ has distance $d = \delta$.

One of the key features of RS codes is that they can be viewed in two different ways, linked by the *discrete Fourier transform*.

Definition A.17. Let R be a commutative ring with unity. An element $\alpha \in R$ is a *principal n -th root of unity* if

1. $\alpha^n = 1$;
2. $\sum_{j=0}^{n-1} \alpha^{jk} = 0$ for all $k = 1, \dots, n - 1$.

If R is a domain, it suffices to consider a *primitive n -th root of unity*, replacing 2. with

- 2'. $\alpha^k \neq 1$ for all $k = 1, \dots, n - 1$.

Definition A.18. Let R be a commutative ring with unity and let $\alpha \in R$ be a principal n -root of unity. Consider an element $\mathbf{v} \in R^n$, whose components are labelled as (v_0, \dots, v_{n-1}) for simplicity of notation. The map

$$\begin{aligned} \mathcal{F}: R^n &\longrightarrow R^n \\ \mathbf{v} &\longmapsto \mathbf{f} \end{aligned}$$

such that, for $k = 0, \dots, n - 1$,

$$f_k = \sum_{j=0}^{n-1} v_j \alpha^{jk}$$

is called *discrete Fourier transform*.

Definition A.19. Let $v(X) = v_0 + v_1X + \cdots + v_{n-1}X^{n-1} \in \mathbb{K}[X]$ and let α be a primitive n -th root of unity in \mathbb{K} . The *Mattson-Solomon polynomial* of $v(X)$ is the polynomial

$$\hat{v}(X) := \sum_{j=0}^{n-1} v(\alpha^{-j})X^j = \sum_{j=0}^{n-1} v(\alpha^j)X^{n-j} \pmod{X^n - 1}.$$

The discrete Fourier transform and the Mattson-Solomon polynomial are obviously related, in fact, under the usual bijection between $\mathbf{v} = (v_0, \dots, v_{n-1})$ and $v(X) = v_0 + v_1X + \cdots + v_{n-1}X^{n-1}$, we may see that the n -tuple $\mathcal{F}(\mathbf{v})$ is mapped to the polynomial $\hat{v}(X^{-1})$ (remember that X is invertible modulo $X^n - 1$).

The usual properties of the Fourier transform hold, as we can see in the next propositions.

Proposition A.20. For $a(X), b(X) \in \mathbb{K}[X]/(X^n - 1)$, let ab be the standard polynomial multiplication and $a \cdot b$ the component-wise multiplication, i.e.

$$(a \cdot b)(X) := \sum_{j=0}^{n-1} (a_j b_j)X^j.$$

Then $\widehat{ab} = \hat{a} \cdot \hat{b}$.

Proposition A.21 (Inversion Formula). For all $a(X) \in \mathbb{K}[X]/(X^n - 1)$ we have

$$a(X) = \frac{1}{n} \hat{a}(X^{-1})$$

where $n = 1 + \cdots + 1$ (n times) in \mathbb{K} (remember that n and $\text{char}(\mathbb{K})$ are coprime).

The Mattson-Solomon polynomial exploits the double nature of RS codes. Let $n = k + 2t$ and consider the set

$$D := \{g(X) \in \mathbb{K}[X] \mid \deg(g) < k\}.$$

Let $g(X) = g_0 + g_1X + \cdots + g_{k-1}X^{k-1} \in D$ viewed as a polynomial of $\mathbb{K}[X]/(X^n - 1)$ and let $\hat{g}(X)$ its Mattson-Solomon polynomial. The Inversion Formula tells us that $\hat{g}(\alpha^{-i}) = ng_{n-i} = 0$ for all $i = 1, \dots, 2t$. In other words, the set

$$C := \{\hat{g}(X) \mid g \in D\}$$

is an RS code (it fits Definition A.13 of a BCH code with $\ell = 1$ and $\beta = \alpha^{-1}$). This description gives an efficient encoding procedure for an RS code, identifying a codeword with a polynomial of degree less than k .

Appendix B

Logic Circuits

There are many ways to represent a function, or more in general a complex system, which is something that “has inputs and outputs and exhibits explicit behaviour, characterized by functions that translate the inputs into new outputs” ([4, p. 1]). One of them is with a circuit.

Circuits are useful also because they have a double interpretation as abstract objects which implement functions and actual devices which do the same in the “real world”—there are physical components like transistors, capacitors, resistors, that reproduce the behaviour of a circuit, thus allowing a concrete computation of the function it implements.

Digital systems (i.e. system whose inputs and outputs are represented by discrete values) have a rigorous formulation founded on the rules of mathematical logic and Boolean algebra. In particular, it is possible to represent the basic logic operations (AND, OR, . . .) by means of single units (*gates*). A circuit can be viewed as a collection of these gates linked by wires which carry the information.

B.1 Combinational Circuits

Combinational circuits are characterized by the fact that there is no feedback—that is to say, there is no wire which serves both as an output and as an input. They are the so-called “circuits without memory”: the outputs depend only on the current inputs.

Arithmetic operators, such as adders, are examples of this kind of circuit. The value of the output is clearly completely determined by the values of the inputs.

The basic building blocks for combinational circuits are the *gates*, which represent single Boolean operations. More complex Boolean expression are obtained by connecting gates in a suitable way; an example can be seen in Figure B.1. The function implemented by the circuit is at first translated into

a Boolean expression, then converted into an appropriate set of interconnected gates.

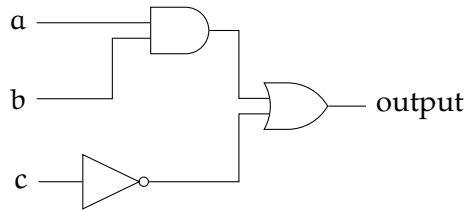


Figure B.1: A combinational circuit that computes $(a \wedge b) \vee (\neg c)$.

B.2 Sequential Circuits

In contrast with the combinational case, sequential circuits “have memory”—their outputs depends not only on the current inputs but also on the history of all previous inputs. In practice, such circuits have a small number of unique configurations (*states*) such that an input can modify the state and the output is based on both the current input and the current state.

For our purposes, the configuration changes in response to a special reference signal, the *clock*. Such circuits are called *synchronous*.

Besides the logic gates, sequential circuits use particular devices called *memory registers*, which are used to store information. A typical register is shown in Figure B.2.

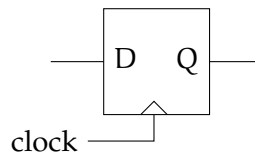


Figure B.2: A typical memory register.

It outputs constantly its content on terminal Q. When the clock ticks, it changes its stored value memorizing the one which is on the input data terminal D.

Bibliography

- [1] David A. Cox, John Little, and Donal O’Shea. *Ideals, Varieties, and Algorithms*. Springer-Verlag, third edition, 2007.
- [2] Gerhard Fettweis and Martin Hassner. A Combined Reed-Solomon Encoder and Syndrome Generator with Small Hardware Complexity. In *Proceedings of ISCAS ’92 — IEEE International Symposium on Circuits and Systems*. Volume 4, 1992, pages 1871–1874.
- [3] Sicun Gao. Counting Zeros over Finite Fields with Gröbner Bases. Master Thesis. Carnegie Mellon University, 2009.
- [4] Randy H. Katz. *Contemporary Logic Design*. Benjamin-Cummings Publishing Co., Inc., 1993.
- [5] Shu Lin and Daniel J. Costello. *Error Control Coding*. Prentice-Hall, second edition, 2004.
- [6] J. H. Van Lint. *Introduction to Coding Theory*. Volume 86 of *Graduate Texts in Mathematics*. Springer-Verlag, third edition, 1998.
- [7] Jinpeng Lv, Priyank Kalla, and Florian Enescu. Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(9):1409–1420, 2013.
- [8] Jinpeng Lv, Priyank Kalla, and Florian Enescu. Verification of Composite Galois Field Multipliers over $\text{GF}((2^m)^n)$ Using Computer Algebra Techniques. In *2011 IEEE International High Level Design Validation and Test Workshop*, 2011, pages 136–143.
- [9] William Wesley Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. MIT Press, second edition, 1972.
- [10] William A. Stein et al. *Sage Mathematics Software (Version 6.3)*. Website: <http://www.sagemath.org>. The Sage Development Team, 2014.

Special Thanks To...

Ed eccoci arrivati alla pagina dei ringraziamenti, che probabilmente sarà la più letta dell'intera tesi. Sono stato indeciso fino all'ultimo momento: scrivo questa pagina oppure no? Da una parte, avrei voluto dedicare un po' di spazio ai ringraziamenti, anche perché nella tesi triennale non l'avevo fatto; dall'altra, vedevo due grossi ostacoli:

1. avevo paura di dimenticarmi qualcuno (sapete, con l'età, la memoria inizia a vacillare...);
2. non volevo fare l'elenco dei nomi, tipo lista della spesa.

Alla fine sono giunto a un compromesso: per non fare un torto a nessuno, ho deciso di non mettere nomi e cognomi esplicitamente (con poche, notevoli eccezioni), in modo da aggirare entrambi i problemi.

Quindi, i miei ringraziamenti vanno semplicemente a tutti coloro che ho incontrato in questi anni. Un grande grazie per il sostegno reciproco durante le lezioni, quando ci guardavamo sconsolati cercando di capire qualcosa da spiegazioni caotiche. Grazie per le cene trascorse in compagnia (e qui mi rivolgo soprattutto agli eccellenti cuochi) e anche grazie per i dopo cena trascorsi in compagnia. Grazie a chiunque abbia reso più piacevole la vita in dipartimento. Una menzione speciale spetta ai membri del PHC (so che sto violando la regola di non citare esplicitamente nessuno, ma quando ho accennato al fatto che forse non avrei scritto i ringraziamenti, mi hanno fissato in cagnesco...).

Nonostante le regole che mi sono imposto, alcuni ringraziamenti sono troppo importanti per non essere inclusi. Innanzitutto, vorrei ringraziare i miei relatori, prof.ssa Patrizia Gianni e dott. Barry Trager, per il costante impegno e la pazienza che hanno avuto nel seguirmi. Il percorso che ha portato a questa tesi magari non è stato dei più lineari, ma con il loro aiuto sono riuscito a non farmi sopraffare dalla mole di lavoro.

Grazie anche ai miei genitori Floriana e Maurizio, per aver sempre creduto in me e per avermi dato la possibilità di frequentare un'università e un corso

di laurea a cui tenevo particolarmente. Chiedo loro scusa per le sporadiche incomprensioni e spero in futuro di renderli orgogliosi di me.

Infine, *last but not least*, un ringraziamento speciale va a Luigia. Con te ho condiviso momenti indimenticabili (uno su tutti, il mondiale di ciclismo a Firenze: un pomeriggio intero sotto il diluvio!). Mi hai fatto riscoprire la passione per la musica classica. Grazie per quel pizzico di pazzia che porti nelle mie giornate. Grazie per un sacco di piccole altre cose che non posso elencare qui, perché altrimenti i ringraziamenti diventerebbero lunghi quanto la tesi stessa. Grazie soprattutto per essere sempre al mio fianco.