INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE
N° 1069

A LINEAR SYSTOLIC ARRAY FOR THE COMPUTATION OF GRÖBNER BASIS

TANGUY RISSET
AND
YANNICK SAOUTER

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

A linear systolic array for the computation of Gröbner basis

Tanguy Risset
and
Yannick Saouter

Thème 1 — Réseaux et systèmes
Projet API

Publication interne n° 1069 — Décembre 1996 — 40 pages

Abstract: Gröbner basis are a powerful tool with many applications in symbolic computation. In this article, we describe a linear systolic array which can be used as coprocessor for efficient computation of Gröbner basis. It was designed by synthesis from the high-level specification language Alpha.

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Un réseau systolique linéaire pour le calcul de bases de Gröbner

Résumé : Les bases de Gröbner sont un outil puissant avec de nombreuses applications dans le domaine du calcul symbolique. Dans cet article, nous décrivons un réseau systolique linéaire qui peut être utilisé comme coprocesseur pour le calcul efficace des bases de Gröbner. Il a été produit par synthèse à partir du langage de spécification de haut niveau Alpha.

1 Introduction

Gröbner basis were introduced by Bruno Buchberger [1] in 1965 in the context of his work on performing algorithmic computations in residue classes of polynomial rings. This concept has many applications in symbolic computations [2, 3, 4, 5, 6], but also in computational non-linear geometry [7], in automatic geometry proving [8, 9] and in modelling theory [10, 11]. However those fields of application do not require real-time application and thus dedicated hardware is generally not necessary. Gröbner basis have also applications in robotics [12, 13], but the power of computation needed to fit real-time constraints is far away from what is possible in current technology. In these recent years, several applications of Gröbner basis have appeared in the field of error-correcting codes [14, 15, 16], for the decoding of some block codes. We present in this article a linear systolic array for implementation of Gröbner basis algorithms at real-time speed. In section 2 and 3, we introduce the context of Gröbner basis. In section 4, we present the Buchberger's algorithm which is used to compute these basis. In section 5, we precise the choices for the implementation. In section 6, we describe our linear systolic array. Section 7 is devoted to summarize our implementation results. At last, in section 8, we describe possible future improvements.

2 Definitions and theorems

In this section we briefly present the notions about ideals in rings of multivariate polynomials. We refer the reader to [17, 18] for a full course on the subject.

2.1 Ideals

Definition 2.1 (Ideal) *Let $(A, +, .)$ be a commutative ring. A subset I of A will be said to be an* ideal *if the two following conditions are true:*

- *$(I, +)$ is a group,*
- *for all $x \in I$ and for all $a \in A$, we have $a.x \in I$.*

We can note that ideals always exist for a ring A: for instance A is an ideal of itself.

Definition 2.2 (Noetherian ring) *Let A be a commutative ring. The ring A will be said to be* noetherian *if for any denumerable set of ideals I_n, $n \in \mathbb{N}$, such that $I_n \subset I_{n+1}$ for all $n \in \mathbb{N}$, there exists an integer n_0 such that $I_{n_0} = I_{n_0+k}$ for all $k \in \mathbb{N}$.*

Definition 2.3 (System of generators) *Let I be an ideal of a ring A. A family (f_j), $j \in J$ of elements of I, where J is a set denumerable or not will be said to be a* system of generators *of I if for any $x \in I$, there exists a family (a_j), $j \in J$, of elements of A, with only a finite number of non-null a_j, such that $x = \sum_{j \in J} a_j.f_j$.*

We have then:

Theorem 2.1 *A ring A is noetherian if and only if any ideal I of A is finitely generated, i.e. has a system of generators, whose cardinal is finite.*

In fact a Gröbner basis will consist in a particular system of generators in ideal of polynomials.

2.2 Multivariate polynomials

Let A be a commutative ring and $A[\mathrm{X}_1, \mathrm{X}_2, ..., \mathrm{X}_\mathrm{n}]$ be the ring of polynomials with n indeterminates and coefficients over A. We have then:

Theorem 2.2 *The ring $A[\mathrm{X}_1, \mathrm{X}_2, ..., \mathrm{X}_\mathrm{n}]$ is noetherian.*

We define also:

Definition 2.4 (Terms and monomials) *Let $T^n = \{{\mathrm{X}_1}^{a_1}...{\mathrm{X}_\mathrm{n}}^{a_n} \mid a_i \in I\!N\}$. The elements of T^n are called* terms. *We call* monomials *the elements of $A[\mathrm{X}_1, \mathrm{X}_2, ..., \mathrm{X}_\mathrm{n}]$ of type aT, $a \in A$, $T \in T^n$.*

2.3 Term ordering

In this paragraph we define orders on the set of the terms.

Definition 2.5 (term-ordering) *Let σ be a strict total ordering on the set of terms T_n. This order will be called a* term-ordering *if the two following conditions are also true:*

- $1 <_\sigma t$ *for all term* $t \in T_n - \{1\}$,
- *If $t_1 <_\sigma t_2$, for t_1 and t_2 in T_n, then $t.t_1 <_\sigma t.t_2$, for all t in T_n.*

The reader may easily verify that the lexicographical order and the total degree order are term-orderings. The reader may also verify that for one variable polynomials the only existing term-ordering is the common degree of polynomials.

3 Gröbner basis

In this section we introduce the concept of Gröbner basis. From now on, our multivariate polynomial will not have coefficient in a simple ring but in a field. If we take the example of a ring of one variable polynomial such as $\mathbb{Q}[X]$, for instance, it is well-known that this ring is principal (i.e. any ideal is generated by one polynomial). As a consequence, to decide whether a polynomial q belongs to an ideal I, in this case, it suffices to divide the polynomial q by the generating polynomial q_0 of I and the remainder by Euclidean division is null if and only if $q \in I$. Gröbner basis are the equivalent in multivariate polynomials of the generating polynomial q_0, and the term-orderings are the equivalent of the natural degree for one variable polynomials which is used to order the monomials when one performs an euclidean division. Since rings of multivariate polynomials are noetherian, a Gröbner basis will contain several polynomials in general and since there exists many different term-orderings, the Gröbner basis of an ideal also depend on the chosen term-orderings. Let us formalize these concepts.

Definition 3.1 (leading term - monomial - coefficient) *Let K be a field, $A = K[\mathrm{X}_1, \mathrm{X}_2, ..., \mathrm{X}_\mathrm{n}]$ and σ a term ordering. Then every non zero polynomial $f = f(\mathrm{X}_1, \mathrm{X}_2, ..., \mathrm{X}_\mathrm{n})$ can be written in a unique way as $f = \sum_{i=1}^{r} c_i T_i$, with $c_i \in K^*$ and $T_1 >_\sigma T_2 >_\sigma ... >_\sigma T_r$. We denote $c(f, T_i) = c_i$ and:*

$Lt_\sigma(f)$ *the term T_1 and we call it the* leading term *of f with respect to σ,*

$Lc_\sigma(f)$ *the coefficient c_1 and we call it the* leading coefficient *of f with respect to σ,*

$Lm_\sigma(f)$ *the monomial $c_1.T_1$ and we call it the* leading monomial *of f with respect to σ.*

Definition 3.2 (reduction) *Let f and g two polynomials of A and $F = \{f_1, ..., f_t\} \subset A - \{0\}$. We say that f* reduces *in one step as g along F if there exists $f_i \in F$ and $T \in T^n$ such that:*

$$g \;=\; f - \frac{c}{lc_\sigma(f_i)} T.f_i$$

where $c = c(f, T.Lt_\sigma(f_i))$. This notion extend immediately by transitive closure and we denote it $\xrightarrow{F}$.

In fact the definition of one step of reduction is the equivalent of one step of euclidean division in the case of one variable polynomial. One may note that if we strictly follow the definition, we do not necessarily eliminate the leading term of g.

It is not immediately clear that the reduction process ends, that is to say that after a finite number of reduction we are led to a polynomial which cannot be reduced any more in a different polynomial. A relation with such property is called a noetherian relation. We refer the reader to [17, p. B17] for a proof that the relation $\xrightarrow{\mathrm{F}}$ is noetherian.

Definition 3.3 (confluence) *Let E be a set and $\mathcal{R}$ a reflexive and transitive relation on E. The relation $\mathcal{R}$ is said to be* confluent, *if whenever $x\mathcal{R}y_1$ and $x\mathcal{R}y_2$, for x, y_1 and y_2 in $\mathcal{R}$, then there exists z in E such that $y_1\mathcal{R}z$ and $y_2\mathcal{R}z$.*

In general, when F is an arbitrary set of polynomials, the relation $\xrightarrow{\mathrm{F}}$ is not confluent.

Definition 3.4 (Gröbner basis) *Let $F = \{f_1, ..., f_t\} \subset A - \{0\}$, and (F) the ideal generated by F. If the relation $\xrightarrow{F}$ is confluent then F is said to be a* Gröbner basis *of (F).*

The reader will be referred again to [17, p. B23] to have a proof of the fact that any ideal of multivariate polynomial has Gröbner basis. The interest of such basis appears then clearly: whatever is the considered chain of reductions, one converges always to the same result. An important property in relation is the following one:

Theorem 3.1 *Let $F = \{f_1, ..., f_t\} \subset A - \{0\}$ be a Gröbner basis of the ideal (F). Let $f \in A$, then we have:*

$$f \in (F) \iff f \xrightarrow{F} 0$$

4 The Buchberger's algorithm

In the previous section we have seen that Gröbner basis have a great role with respect to the ideal they generate. In this section, we will see how such basis are effectively built. This algorithm can be found in [1]. Some improvements are possible [19], but we will not take them into account here since this would not affect the architecture of our systolic coprocessor but only the monitoring program of its host.

4.1 S-polynomial

Definition 4.1 (S-polynomial) *Let f_1 and f_2 be two polynomials of $A = K[\mathrm{X_1}, \mathrm{X_2}, ..., \mathrm{X_n}]$, and σ a term-ordering. Then the* S-polynomial *of f_1 and f_2 will be the polynomial:*

$$f \quad = \quad Lc_\sigma(f_2).u_1.f_1 - Lc_\sigma(f_1).u_2.f_2$$

where $u_1 = \frac{lcm(Lt_\sigma(f_1),Lt_\sigma(f_2))}{Lt_\sigma(f_1)}$ and $u_2 = \frac{lcm(Lt_\sigma(f_1),Lt_\sigma(f_2))}{Lt_\sigma(f_2)}$ (lcm denotes the least commom multiple of two polynomials).

4.2 The algorithm

In this paragraph we suppose that we are given a set $F = \{f_1, ..., f_t\}$ of multivariate polynomials and we wish to compute a Gröbner basis of the ideal (F) with respect to a term ordering σ. The Buchberger's algorithm is the following:

Algorithm 1
Input*: F, σ.*

- **Initialization***: Set $G = F$ and $B = \{\{f_1, f_2\} \mid f_1, f_2 \in F, f_1 \neq f_2\}$.*
- **Step 1***: If $B = \emptyset$, then* **Stop***.*
- **Step 2***: Choose $\{f_1, f_2\} \in B$ such that $lcm(Lt_\sigma(f_1), Lt_\sigma(f_2))$ is minimal with respect to σ. Set $B = B - \{f_1, f_2\}$.*
- **Step 3***: Compute $h = S(f_1, f_2)$ and compute h' such that $h \xrightarrow{G} h'$.*
- **Step 4***: If $h' \neq 0$, then set $B = B \cup \{\{g, h'\} \mid g \in G\}$ and $G = G \cup \{h'\}$. Goto* **Step 1***.*

Output*: G.*

This algorithm ends and gives in G a Gröbner basis for the ideal (F) with respect to the term-ordering σ. In this algorithm the iteration set G always grows. In fact, this set could be reduced after or during the computation by checking for any polynomial $f \in G$, whether it reduces into 0 along the set $G - \{f\}$. In the next sections we will show how to implement the basic operations of reduction and of computation of the S-polynomials in a linear systolic array.

5 Using Gröbner basis for error correcting codes

As we have seen in the introduction, Gröbner basis are widely used in many scientific fields. However, their use in the field of error correcting code is restricted to special cases. Gröbner basis are used in multi-variable codes (semi simple codes, Abelian codes, see [20] for a survey). These codes manipulates elements of $K[X_1, \ldots, X_n]/I$ where K is a Galois field, and I is an ideal of $K[X_1, \ldots, X_n]$ generated by $(P_1, \ldots, P_k)$. The Gröbner basis is generated from $(P_1, \ldots, P_k)$ by a series of reductions (definition 3.2) and computations of S-polynomials (definition 4.1) as explained in section 4. It is used, for instance, for reducing a polynomial to its class in $K[X_1, \ldots, X_n]/I$ by a series of reduction. Thus the reduction operator is very often used when computing in $K[X_1, \ldots, X_n]/I$. It is easy to see that the computation of the S-polynomial is very similar to the reduction operation. In the following we will propose a systolic array for computing either the S-polynomial of two polynomials or the reduction of one polynomial on another.

In multi-variable codes, the field K is a finite field (or Galois field) with p^n elements (GF(p^n) see [21]) where p is a prime number. This imply that all the operations on coefficients of the polynomials will need special operators for computations in Galois fields. In this paper, we will not address the problem of deriving efficient operators, we have chosen one possible implementation, however the problem has already been studied and some regular [20, 22] and irregular [23, 24] architectures have been proposed.

It is also important to know the number of variables of the ring of polynomials. If there is only one variable, the field is principal and there is only one generator to find. Most of multi-variable codes use two or three indeterminates with lexicographic order on the degree. For matter of simplicity, we will describe an architecture for GF(p^n)$[X, Y]$. The overall principles of the architecture can easily be extended to a greater number of indeterminates, the only things that will change are the operators related to the term ordering (which, in this paper, is chosen to be the lexicographic ordering).

5.1 Implementation choices

The target architectural model will be a systolic linear unidirectional array. This model is synchronous, communications with the host are known statically [25]. The array is unidirectional (all communication channels go in the same direction), thus the algorithm can be partitioned and completed in several passes if the array is not big enough (LPGS partitioning [26]). Finally, this model can be implemented either in ASIC or FPGA technology.

Now, we have to choose the representation for the polynomials. There are mainly two possibilities. One is to use a static data type (like a matrix) with one memory location for every monomial of the polynomial, the coefficient of X^xY^y will be stored in memory case (x, y). The other one (sparse representation) is to use a dynamic linked list keeping only non null coefficients. This list will be composed of triplets: (λ, x, y) where λX^xY^y is a monomial of the polynomials. In this last representation, we impose the list to be ordered with respect to the term ordering.

A first natural idea when implementing algorithms on systolic arrays is to choose the first representation (like in [27] for instance), because the static representation enable a regular processing. However, we have two reasons for choosing the linked list representation. Firstly, we make the assumption that the average number of monomials in our polynomials if very low compared to the maximum number of possible monomials. Under this assumption, this representation will use much less memory (as most of the possible coefficients of the polynomials will be null). Secondly, the multiplication of a polynomial by a monomial is much more complicated in first representation (a global shift of all the coefficients of the array instead of two additions for each coefficients of the list).

Now, we have to choose a representation for the coefficients of $\mathrm{GF}(p^n)$ (in this paper, we will work on $\mathrm{GF}(2^8)$). As $\mathrm{GF}(2^8)$ is a Galois field, there always exists an element of the field α such that the whole field is generated by this element i.e: $\mathrm{GF}(2^8)$ is exactly the set $0, 1, \alpha, \alpha^2, \ldots, \alpha^{254}$. Thus we can represent one element λ by an integer between 0 and 255, 0 for $\lambda = 0$ and $i \in [1, 255]$ for $\lambda = \alpha^{i-1}$. We have to perform multiplications and additions on these coefficients, multiplications can be performed by a simple addition (modulo 255) and a test on the coefficient. The addition is a little bit more complicated. In this paper, we have chosen to realize it with a look-up table (see section 6.3).

Thus a polynomial will be available in the form of a linked list of triplets (λ_i, x_i, y_i) of integer with $i \in I$ (in section 7, we have chosen, for the implementation, λ, x and y to be between 0 and 255). This series will be supposed to have their terms ordered in lexicographical decreasing order on (x_i, y_i), i.e. the greatest term with respect to the order σ is (λ_0, x_0, y_0) and the following monomials have a lexicographic-degree decreasing while i is increasing. We also suppose that the coefficient λ_0 is non null.

5.2 Algorithms

In this section we rewrite the algorithms of reduction and of computation of the S-polynomials with the notation of linked lists. Thus we suppose that we are given two series $(\lambda_{i1}, x_{i1}, y_{i1})$, and $(\lambda_{i2}, x_{i2}, y_{i2})$ with $i \in I$, which represents the monomials of the two polynomials arguments of the operation with the previous conventions. At last the result of the operation will be a serie (μ_i, x'_i, y'_i) with $i \in J$. With these notations, we can write the algorithms.

Algorithm 2 (algorithms of the reduction and S-polynomial)
Input: $(\lambda_{i1}, x_{i1}, y_{i1})$, $(\lambda_{i2}, x_{i2}, y_{i2})$.

- **Step 1 (Reduction)**: *Set* $\Lambda_1 = \lambda_{02}$, $\Lambda_2 = \lambda_{01}$, $u_{x1} = 0$, $u_{y1} = 0$. $mx = x_{01} - x_{02}$ *and* $my = y_{01} - y_{02}$. *If* $mx < 0$ *or* $my < 0$ *then set* $J = I$ *and* $\{(\mu_i, x'_i, y'_i)\} = \{(\lambda_{i1}, x_{i1}, y_{i1})\}$. **Stop** *else set* $u_{x2} = mx$, $u_{y2} = my$.
- **Step 1' (S-polynomial)**: *Set* $MX = Max(x_{01}, x_{02})$, $MY = Max(y_{01}, y_{02})$, $u_{x1} = MX - x_{01}$, $u_{x2} = MX - x_{02}$, $u_{y1} = MY - y_{01}$, $u_{y2} = MY - y_{02}$, $\Lambda_1 = \lambda_{02}$, $\Lambda_2 = \lambda_{01}$.
- **Step 2**: *Set* $i1 = i2 = 1$ *and* $j = 0$. *While* $i1 \neq Card(I) + 1$:
  - *If* $x_{i1.1} > x_{i1.2}$ *or* $x_{i1.1} = x_{i2.2}$ *and* $y_{i1.1} > y_{i2.2}$ *then set* $(\mu_j, x'_j, y'_j) = (\lambda_{i1.1} \times \Lambda_1, x_{i1.1} + u_{x1}, y_{i1.1} + u_{y1})$, $i1 = i1 + 1$, $j = j + 1$.
  - *If* $x_{i1.1} = x_{i1.2}$ *and* $y_{i1.1} = y_{i2.2}$ *then set* $(\mu_j, x'_j, y'_j) = (\lambda_{i1.1} \times \Lambda_1 - \lambda_{i2.2} \times \Lambda_2, x_{i1.1} + u_{x1}, y_{i1.1} + u_{y1})$. *If* $\lambda_{i1.1} \times \Lambda_1 - \lambda_{i2.2} \times \Lambda_2 \neq 0$, *set* $j = j + 1$. *Set* $i1 = i1 + 1$, $i2 = i2 + 1$.
  - *If* $x_{i1.1} < x_{i1.2}$ *or* $x_{i1.1} = x_{i2.2}$ *and* $y_{i1.1} < y_{i2.2}$ *then set* $(\mu_j, x'_j, y'_j) = (-\lambda_{i2.2} \times \Lambda_2, x_{i2.2} + u_{x2}, y_{i2.2} + u_{y2})$, $i2 = i2 + 1$, $j = j + 1$.
- **Step 3**: *Set* $J = [1..j - 1]$.

Output: (μ_j, x'_j, y'_j).

In fact the algorithm of reduction described above is not the one of the definition 3.2: the reduced polynomial is multiplied by $lc_\sigma(f_2)$. This relation is still noetherian and the Buchberger's algorithm leads to polynomials proportionnal with constants to the ones obtained by the original algorithm. From an architectural point of view this new expression of the reduction avoid the operation of division and involves two multiplications which can be performed in a parallel fashion.

As they are written, these algorithms could be directly implemented by FIFO register files. There would be two FIFO files, one for each polynomial and a cell of a FIFO would have three sub-cells (one for the coefficient of the monomial, one for the degree in X and the last one for the degree in Y). Every incrementation in the variables $i1$ and $i2$ would respectively correspond to a shift of the first or the second FIFO file and the computations of the algorithm would be performed in a single arithmetical unit. It has to be noted that FIFO files imply an asynchronous exchange, typically in handshake scheme, with the host which is a great factor of loss of speed.

6 The systolic array

The systolic array is composed of three types of processors (or cells). The first and the last cell are special ones. All the other ones are identical, they compose an array which is almost a systolic sort array (with one little difference as we will see later). We want to be able to execute the following operation:

$$P = lc_\sigma(P_1)u_2P_2 - lc_\sigma(P_2)u_1P_1 \tag{1}$$

Where $u_1 = 1$, $u_2 = T$ in the case of a reduction (see definition 3.2) and $u_1 = \frac{lcm(Lt_\sigma(f_1), Lt_\sigma(f_2))}{Lt_\sigma(f_1)}$, $u_2 = \frac{lcm(Lt_\sigma(f_1), Lt_\sigma(f_2))}{Lt_\sigma(f_2)}$ in the case of the S-polynomial. As indicated in section 5.1, the coefficients of the polynomials arrive in decreasing degree order.

The first cell determines the value of the monomials u_1 and u_2 to be multiplied by the monomials of P_1 and P_2, executes the multiplications and interleaves the coefficients of the two resulting polynomials. Thus at the output of the first cell, we have only one stream of polynomial coefficients, but these coefficients are unordered and there remains to perform the additions of coefficients having the same degree. This work is what is done by the remaining of the array.

The middle cells sort the coefficients in decreasing degree order. This is very close to a classical systolic sort: each cell stores the first coefficient it receives, after that, for each coefficient received, if it has a lower degree than the one stored, it is transmitted to the next cell, if the degree is higher, the internal coefficient is transmitted and the current coefficient is stored and if the degrees are equal, the current coefficient is stored in an additional register (this case can appear only once

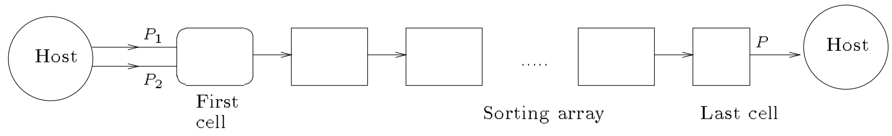

Figure 1: The systolic array

for each degree, when the two modified input polynomials have a coefficient of this degree).

The sorted coefficients of the polynomial are stored in the array. To empty the array, a control signal indicating the end of the computation is sent in the array, and the coefficients are output toward the end of the array. As there may be some degree for which two coefficients have been stored, they are sent in parallel on two channels. These two coefficients are added in the last cell of the array which is connected to the host (see figure 1).

In the following, we describe precisely the internal architecture of each type of cell. We give a description of the behavior, schematics at the register transfer level. The algorithms performed by each cell can be found in the appendix.

6.1 The first cell

In this section (as in section 6.2, and 6.3) We use the notations of algorithm 2. We recall that the representation introduced in section 5.1 impose that each coefficient of the polynomial is represented by a triplet of integer (λ, x, y) (for $\lambda X^x Y^y$) this means that we need three channels for each polynomial.

This processor (represented in figure 2) receives the two polynomials from the host. At the first step (indicated by the *init* signal), it computes and stores the coefficients $\Lambda_1, u_1, \Lambda_2, u_2$, this is done by the *coefficient computation* of figure 3. The values of u_1 and u_2 are set accordingly to a signal (*Redu*) indicating if the computation to perform is a reduction or the computation of a S-polynomial. It also output a signal $Valid$ which indicates if the reduction is possible.

During the other steps, the cell receives monomials m_1 and m_2 from P_1 and P_2, computes $\Lambda_1 m_1 u_1$ and $\Lambda_2 m_2 u_2$ (*Addition* in figure 4). At this step, we have two monomials arriving in parallel. The *Multiplex* cell in figure 4 merges the two channels. For this we have to suppose the original coefficients of P_1 and P_2 are sent every two clock ticks (or that their input is controlled by a clock which is twice slower). This multiplexer is controlled by signal $ck2$ which is true every even

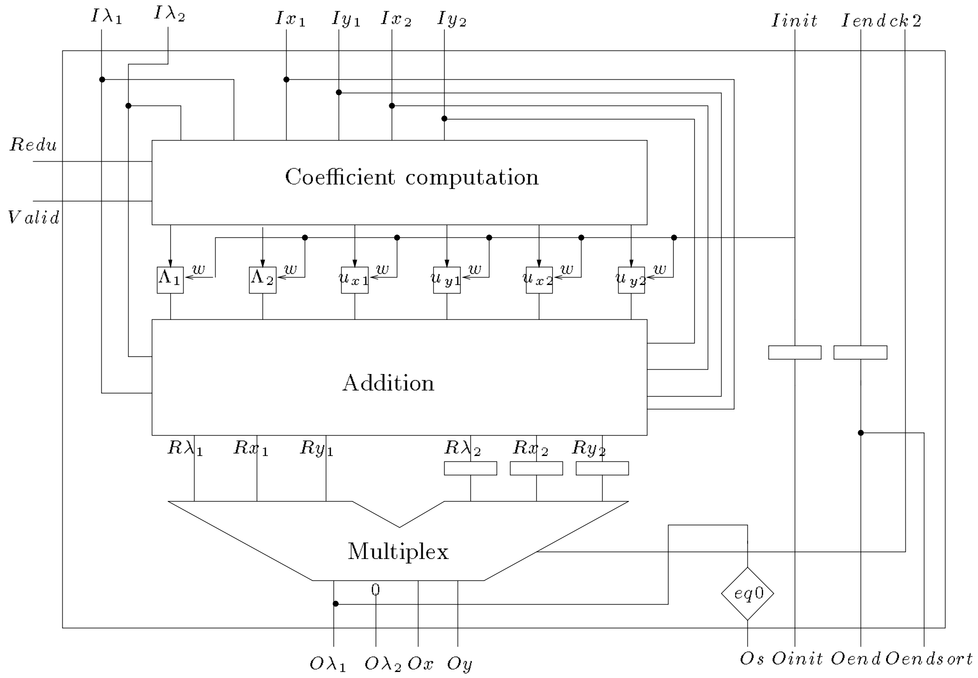

Figure 2: First cell of the array. The squares are registers, w is the write signal, rectangle are latches, the three sub-cells are detailled in figure 3 and 4

clock ticks (such signal can be implemented efficiently from the clock signal, without adding a special input port). The *end* signal is not used in the first cell, it is just transmitted to the next cell. Note that the values stored in the registers are not output at the end, they are only used for internal computations.

Coefficient computation: This part of the cell is only used at the first step of the algorithm, it is controlled by the signal $Redu$ which indicates if we compute a reduction or a S-polynomial. At the first step Λ_1 is exactly λ_{02} and Λ_2 is exactly λ_{01} thus computing Λ_1 and Λ_2 resumes in interchanging λ_1 and λ_2. For the case of a S-polynomial (on the right of the al cell on figure 3), we compute the maximum of the powers on X and Y and then we subtract each coefficient to the corresponding maximum. In the case of a reduction, we have a control signal which indicates if the reduction is possible (i.e if $Ix_2 \geq Ix_1$ and $Iy_2 \geq Iy_1$). If the reduction is not possible, a signal is output ($Valid$) which tells the host that the computation stops. If the reduction is possible, u_{x1}, u_{x2}, u_{y1} and u_{y2} are set like in algorithm 2.

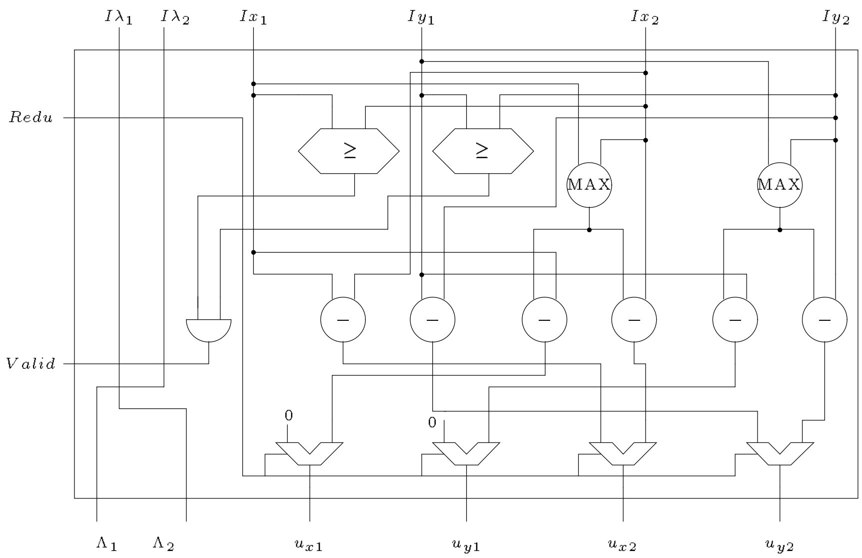

Figure 3: Coefficient computation

Addition and Multiplex The *Addition* cell (left one on figure 4) performs the computations $\Lambda_1 u_1 m_1$ and $\Lambda_2 u_2 m_2$. It receives Λ_1, Λ_2, u_1 and u_2 which where stored in the registers after the computations explained in previous paragraph, and the coefficient of the current monomials m_1 and m_2 of P_1 and P_2 directly from the host. This computation consists in the addition upon the degree and the multiplication (in $\mathrm{GF}(p^n)$) of the coefficients of the monomials. The multiplexing of the two resulting coefficient streams takes place in the multiplex cell (figure 4). This multiplexer chooses the left coefficients or the right coefficients depending if $ck2$ is set to 0 or 1. An additional channel is initialized to 0 for further use in the sorting cells.

6.2 The sorting array

This array is essentially devoted to sort the monomials leaving the first cell. There is many account in the area of systolic sorting. We define here the latency of a systolic array as the time necessary to wait for between two successive input data streams. The article [28] describes a linear bidirectionnal array which sorts data with a LIFO pile behavior and has a nearly null latency if both entries of the array are used. A similar sorting cell is described in [29]. A bidirectionnal VLSI sorter using only $N/2$ comparators for N items to sort is described in [30]: the drawback of this array is

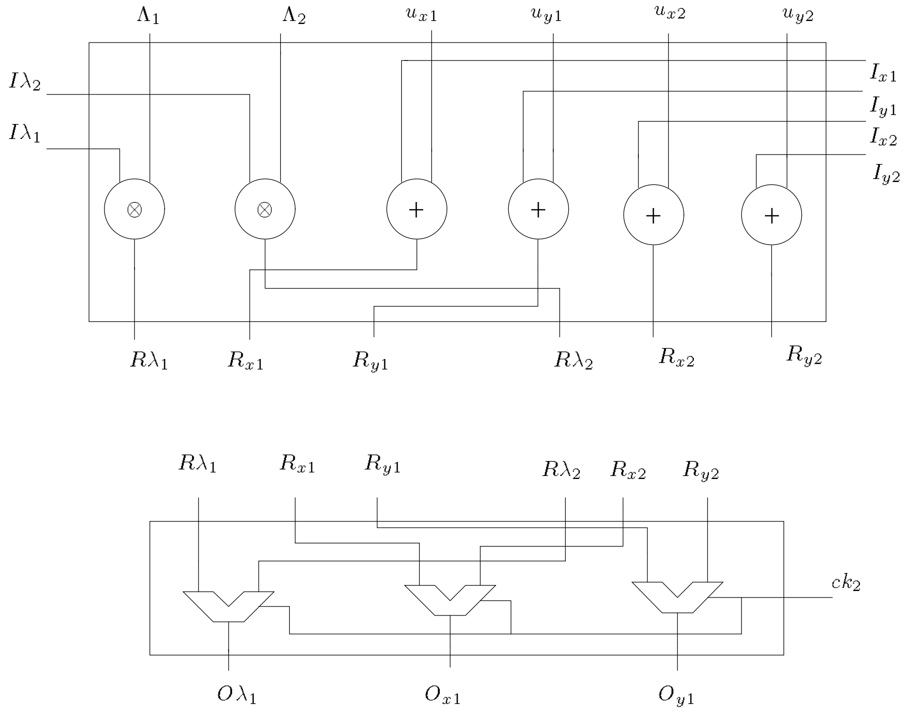

Figure 4: Addition and Multiplex cell, ($\otimes$ denotes multiplication in Galois fields)

a latency of N clock ticks. The two dimensionnal sorting has also been studied in systolic design [31] or toric design [32]. The array we use has one lexicographical comparator (so counting as two integer ones) for each cell and has a null latency. Since we aim real-time applications, this latter quality is crucial.

Behavior of the cell In fact each cell will have 8 input signals and 8 corresponding output signals. These signals will be:

- λ_1 and λ_2 coefficients in GF(p^n),
- x and y which will be positive integers,
- *end*, *endsort*, *init* and s all boolean signals.

Internally the cell will contain 6 registers to store data from its input ports: λ_1, λ_2, x, y, s and *end*. The bit register s will contain the information whether the coefficients stored in the cell correspond effectively to a monomial of the final result or not (s stands for significative). Its value will be respectively 1 or 0.

The signal *endsort* when active will specify to the cell that the sorting of monomials is finished and that the coefficients, λ_1, λ_2, x, y and s stored in the cell have to be transmitted to the corresponding output ports.

The signal *end* when active will specify that the sorting procedure is finished for the cell, i.e. the coefficients λ_1, λ_2, x, y and s received on the input ports do not have to be taken into account so have to be copied directly to the output port. This signal will set the register *end* to 1 so that the cell will keep the information that the sorting procedure is finished. The signal *endsort* will be prioritary with respect to *end*.

The signal *init* is the signal of initialization. Its effect are to set the internal booleans s and *end* of the cell to 0 and the value of λ_2 to 0. For this signal, there is no problem of priority since the actions of the other signals *end* and *endsort* are independent of its own actions. As a consequence, a cell can receive at the same time the signal *init* and any of the two other ones (especially *endsort*).

When none of these three signals is active, we are in steady state of the sorting array. If the input port of the boolean s is equal to 0, the cell just transmits this value to the corresponding output port. On the other case, the cell checks its internal value s. If this latter value is equal to 0, then the cell stores the values λ_1, λ_2, x and y of the input ports in the corresponding internal register and sets its internal boolean s to 1. In the last case (both the values s are equal to 1), a lexicographical comparison have to be done with the stored monomials.

There is then three cases. If the internal monomial is strictly greater than the input monomial, the input ports are copied onto the output ones. If the internal monomial is strictly inferior, then the stored value λ_1, λ_2, x, y are copied to the output ports and the output port of s is set to 1, the input values λ_1, λ_2, x, y are stored in replacement and the register s is set to 1. At last, if both monomial have the same degree, then the register λ_2 is set with the value read on the input port of λ_1 and the output port of s receives the value 0.

The behavior of the cell is described by the functional algorithm 4.

In dynamic behavior, the first cell of the sorting array will receive first a *init* signal, then the stream of data λ_1, λ_2, x and y, beginning one clock tick later. One tick after the end of the data stream, the first cell receives both signal *end* and *endsort*. As we have seen previously those signals can overlap with the *init* signal of the next data stream. In fact, *init* and *endsort* are delayed of two ticks for each sorting cell and *end* is delayed of one tick. This scheduling has been validated by functional simulation under the Alpha environment tool.

It is possible to easily explain the delays of each signals by taking into account the modularity of the array. For instance, when the first cell receives a *init* signal, it erases its internal registers. The next tick, since it has no significant data, it just stores incoming data and produce no output. Then it begins to ouput a stream of data. So, by modularity, since the *init* signal needs to be present one clock tick before the stream of input data, this signal needs to be delayed of two ticks in order to be synchronized for the second cell. The same hypothesis obviously apply by induction on the following cells. Likewise the signal *endsort* must be delayed of two clock ticks by cell. Indeed this signal indicates to a cell that it has to flush its internal registers to the ouput. The next cell is supposed to be in idle state: it just copies input data to output ports. Thus at the following tick, it will have to drive the data flushed by the previous cell, i.e. it will have to remain in idle state. Then at the next tick, this cell will be allowed to flush its own data. Thus the signal *endsort* is delayed of two ticks. At last the signal *end* indicates to a cell that sorting is finished and that it has to become idle. When a cell receive this signal, this means that from now on, it will not transmit again data which needs to be sorted. Thus the last data which may need to be sorted is transmitted exactly at this tick. This means that the next cell will not transmit data which need to be sorted from next tick on. This imply that the signal *end* needs to be delayed of one tick.

Architecture of the cell With the preceding functional explanation of a cell one can draw exactly the internal layout of a cell. This layout is detailled in the figures

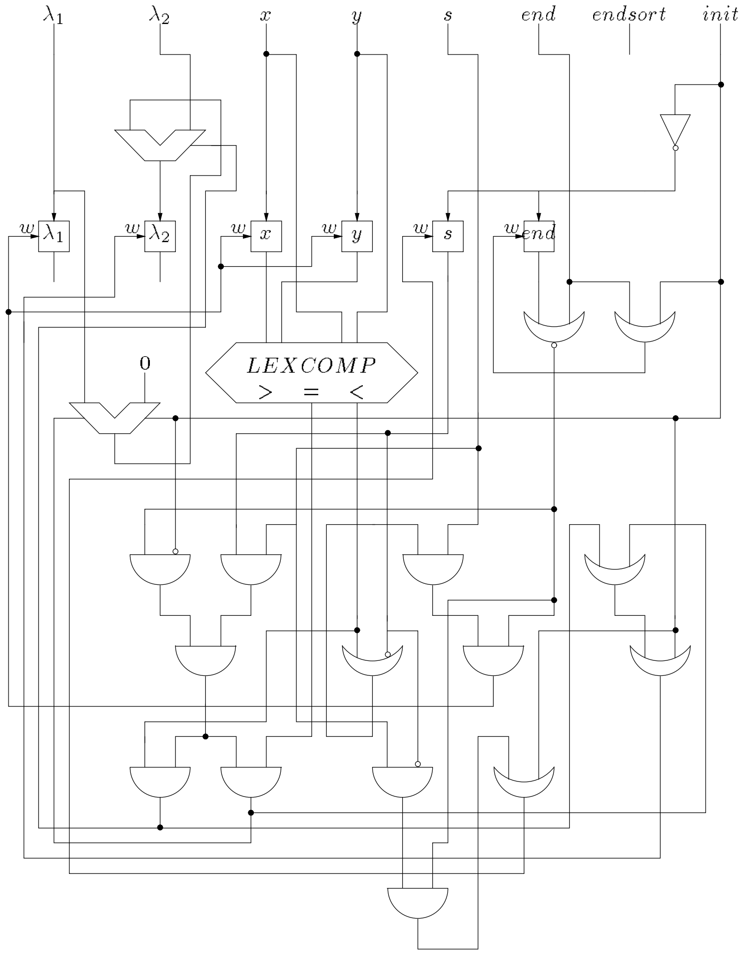

Figure 5: Schematics of the registers

5 and 6. The figure 5 describes how the internal registers are loaded and which are their loading command, while the figure 6 describes the computation of the output from the input ports and the values stored in the registers. The figure 7 describes how the lexicographical comparator is implemented with two integer comparators and several logical gates.

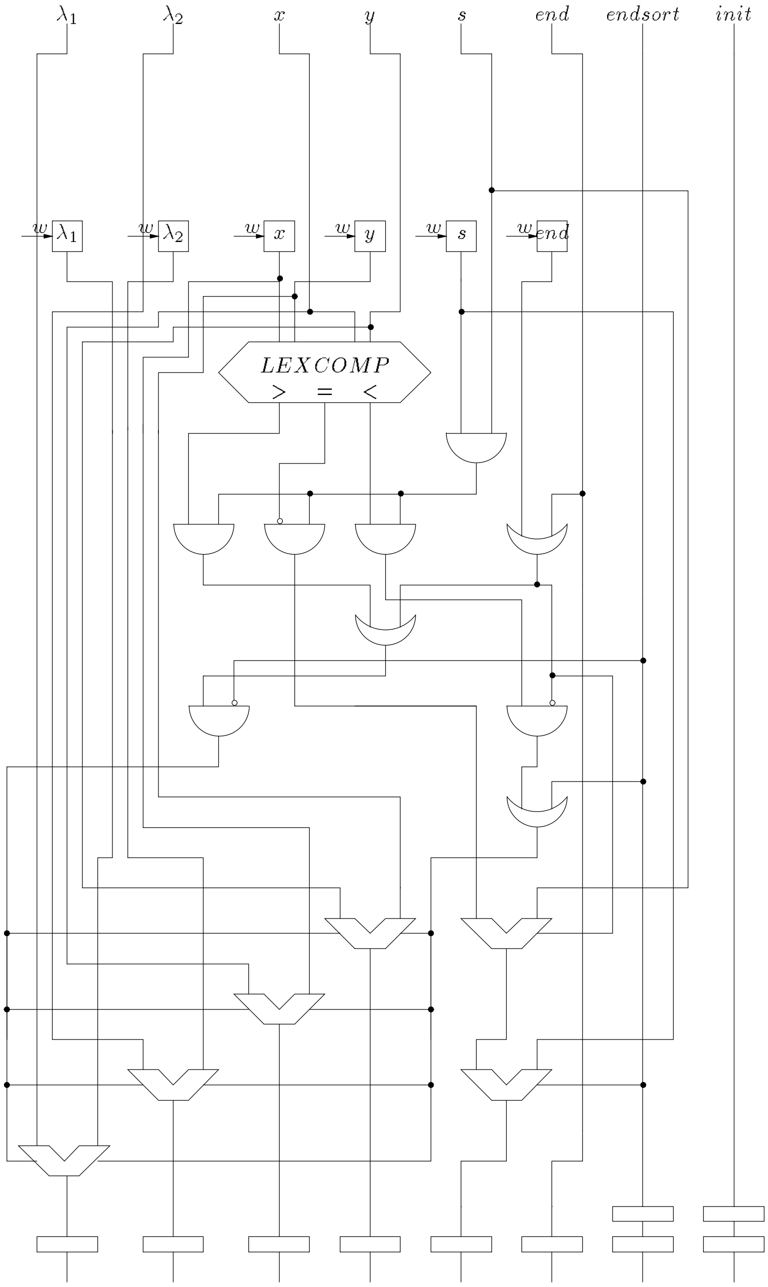

Figure 6: Schematics of the outputs

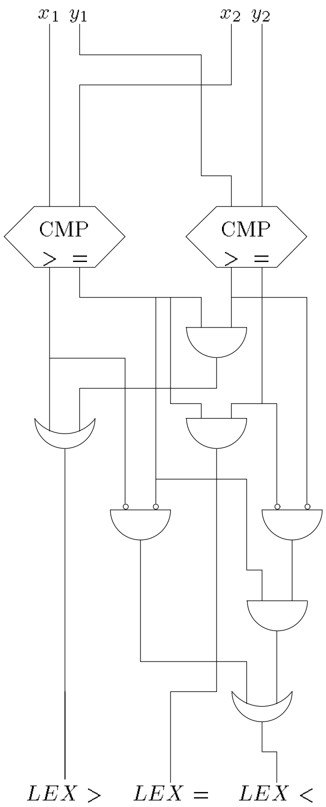

Figure 7: Schematics of the lexicographical comparator

6.3 The tail of the array

This paragraph is devoted to the last cell of the circuit. Its behavior is simple and combinatorial: it just adds the values λ_1 and λ_2 output of the sort array and transmits the value λ of the sum and the values of the monomial x and y. Since the circuit is synchronous, the other signals are useless for the host: the host has an exact timing of the date when it has to collect the result. The internal structure of this cell is then just an adder for Galois fields coefficients. Indeed since we operate on GF(256), the primary field is GF(2), whose characteristics is just 2. As a consequence, in this field any element x verifies the identity $x + x = 0$ and thus the operations of substractions and additions just coincide. Suppose that we have to add two constants λ_1 and λ_2 of GF(256). Then if at least one of them is equal to 0 the result of the addition is the other value. On the contrary, λ_1 and λ_2 can be expressed as α^i and α^j where α is a generator of $GF(256) - \{0\}$ and $0 \leq i, j \leq 254$. If i and j are equal, since the primary subfield is GF(2) the result of the addition is 0. At last, on the default case, we have $\alpha^i + \alpha^j = \alpha^{min(i,j)}(1 + \alpha^{max(i,j)-min(i,j)}) = \alpha^{min(i,j)+Z(\alpha^{max(i,j)-min(i,j)})}$ where Z denotes the Zech's logarithm (see [20, p. 460] for details). In our implementation, this addition is performed with using a ROM device containing the Zech's logarithm table for GF(256).

This algorithm easily leads to the architecture illustrated by figure 8. Signals x and y are not reported since they are just driven through the cell without modification nor delay. In this architecture, the ROM device contains the values of $Z(\alpha^i)$ in terms of i. Note that since the operations involving the exponents of α in what precedes are substractions, minima and maxima, we do not need to correct the shift of 1, introduced by our conventions on the values λ (see section 5.1).

7 Complexity and Implementation

7.1 Complexity of the array

The number of cell of the array is not fixed, suppose that you are given an array of with m cells and polynomials with n coefficients (ideally, we would like to have $m = 2n$ as the result polynomial may have up to $2n$ coefficients). We suppose first that $m \geq 2n$. In that case, the time elapsed between the input of the first coefficient in the array and the output of the last coefficient at the end is: $4n + m$ ($2n$ for the input of all the coefficients as they arrive in the array every two clock ticks, m for the first coefficient to be output at the end and $2n$ for the other coefficient to be output too). However, a new computation can be started as soon as all the coefficients have

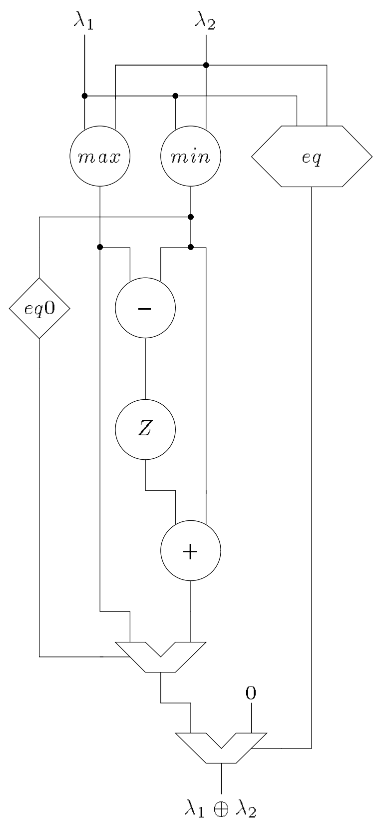

Figure 8: Functionnal description of the GF(256) adder, (Z denotes the Zech's logarithm)

been input, thus the delay between two successive computation is $2n$. The array complexity is mainly due to latches, registers and operators on integers, the number of integral latches is approximately $8(m+1)$, we need approximately 6 adders (in the first cell), $2m$ comparators and one programmable ROM with 256×8 bits stored (in the last cell). We will see in section 8 that, if m is smaller than $2n$, the computation will be performed with partitioning.

To give a more precise idea of the resulting complexity, we have synthesized the corresponding layout.

7.2 Methodology

We have synthesized the layout corresponding to the array described in section 6. The methodology we have used is the following : first we describe the array with the Alpha language [33, 34] which is a functional language that allow to describe a systolic array with a high level formalism. At this point, we were able to simulate the array at the functional level. Then we translated this specification into VHDL code which can be synthesize by commercial tools (this translator is part of the environment implementing Alpha). The advantages of choosing Alpha for expressing our array are the following one:

- the simulation of the array is easy with a high level language like Alpha. Also, Alpha has a very powerful static analyzer which permits, in particular, to check some timing properties (like the latency). These two aspects provide a specification in which it is easier to correct bugs than directly in VHDL.
- Alpha programs can be parameterized, thus we are able to describe the systolic array without knowing exactly what will be the size of the final array. This genericity is very important to reuse the design: if we want to build another array which is bigger, we can start from the Alpha code we already use (and already tested as well).
- the hierarchy (together with the above genericity) provided in Alpha allow to have a hierarchical design. In figure 9, it is shown that the hierarchical design gives better result. This is due to the fact that the placement procedure during the design is not efficient if it is done automatically by the synthesis tool, as this tool cannot recognize regularity and repetitive patterns (like the regular use of the sorting cells). One of the basic feature of Alpha is to provide an efficient way to express regularity. This is especially useful for systolic arrays.

Then we synthesized the VHDL specification with the Compass software and we obtained reports on the size and clock cycle of the layout. The synthesis was done without manual editing (except for the placement procedure, see section 7.3). We asked for the best clock cycle without any resource constraints. The optimization of the layout is out of the scope of the paper, we present these result because they give a idea of the real complexity and speed of the array.

7.3 Architectural synthesis results

The results of the synthesis are presented in figure 9. These synthesis where done with the Compass tool. The technology used is $\lambda = 0.4\mu$. The complete design process consist of the synthesis (generation of a net-list), the placement of the blocks and the routing of the wires. The last two phases can be done hierarchically (i.e. the design is done once for each type of cell and is re-used when this cell is used) or not (flat design, default in Compass). Tests have been made on both hierarchical and flat design. In both cases, we have tested a complete automatic design and a design where the placement procedure was performed by hand (which is very quick in practice). The clock cycle found (timing report) was 42 ns (approximately 160 Mbits for the array). The first 5 tests were done without the last cell. The ROM was integrated in the last three designs which finally gives a total area of (approximately) 36 squared millimeter for an array with 20 cells.

The area reports are quite realistic as they really represent the material used. The timing reports are just evaluation (done by Compass), the timing report do not change from one design to the other as the evaluation depend only on the net-list which is the same in each design. Of course, the clock cycle does not depend on the number of cell of the array. When one integrate the ROM which is designed by a special tool for memory synthesis, the timing report is not given by Compass, but the clock cycle given here can easily be reached by the ROM we use. The clock cycle can be optimized by adding latches on the design of the systolic array (for instance, if we add a latch between the *Addition* cell and the *Multiplex* cell of figure 4, the clock cycle can be lowered from 42 ns to 32 ns). These timing optimizations are not tackled in this paper, just note that such optimizations could be easily performed in the Alpha environment as this environment provides transformations that are valid (keep the semantics of the programs unchanged).

The test clearly show that the best array is obtain with a structured synthesis and manual placement of the blocks. With the clock cycle, and in the hypothesis where $m = 2n = 20$, the complete computation is performed in $2.52\mu s$ and 1000 computations can be done in $0.84s$. The total array of such chip would be around

		area report ($\lambda = 0.4\mu$)	surface
without ROM	5 cells, flat automatic placement	$6633 \times 12163\ \lambda$	12.9 mm^2
	5 cells structured automatic placement	$4757 \times 15635\ \lambda$	11.9 mm^2
	5 cells, structured manual placement	$4222 \times 15468\ \lambda$	10.4 mm^2
	20 cells, structured automatic placement	$10406 \times 37684\ \lambda$	62.7 mm^2
	20 cells, structured manual placement	$5871 \times 36108\ \lambda$	33.9 mm^2
with ROM	5 cells, flat manual placement	$6580 \times 10763\ \lambda$	11.3 mm^2
	20 cells, structured automatic placement	$9606 \times 38225\ \lambda$	58.7 mm^2
	20 cells structured manual placement	$9666 \times 23371\ \lambda$	36.144 mm^2

Figure 9: Experimental results. The synthesis have been done for arrays of 5 and 20 cells, tests have been made with a flat (unstructured) VHDL description or with the structuration given in Alpha, and both automatically and with manual improvement of the placement of the blocks

$36mm^2$. If one wants a bigger number of cell (for bigger polynomials), it is easy to spread cells on several chips as connection between successive cells are simple.

8 Improvements

In order to approach a real implementation several improvements have to be considered. The first point is the reuse of the sorting array. Indeed if the number of coefficients driven in the chip is superior than the number of cells of the sorting array, the last monomials of the result will not be sorted. To be more precise if we are given again two polynomials of n monomial values and if we have m cells with $m < 2n - 2$, the host will receive as result $2n - 2$ coefficients. The first $2n - 2 - m$ coefficients received will not be sorted and the last m coefficients will be all greater than the previous ones and effectively sorted in decreasing order. In order to finish the computations the first $2n - 2 - m$ coefficients will have to be stored in the host and reintroduced in the sorting array (thus avoiding the first cell) and again until all coefficients are sorted. The process will then take $\lceil \frac{2n-2}{m} \rceil$ passes. The $k - th$ phase will last $2n - 2 - (k - 1).m + 1$ clock ticks for the sorting of m items (the sorting array has to receive the *init* signal one clock tick before the effective data and from a phase to the following one the signals *endsort* and *init* can overlap), and the last phase will last $((2n - 2)\ mod\ m) + 1$ clock ticks if there is optimal pipeline with the preceding phase. Thus the global time computation for a single instance with two polynomials with n coefficient is, in terms of clock ticks:

$$T_{comp} = \begin{cases} 2n & \textbf{if} \quad 2n \leq m \\ a(2n-2) - \frac{a(a-1)}{2}.m + ((2n-2)\ mod\ m) + 1 & \textbf{if} \quad 2n > m \end{cases}$$

where $a = \lfloor \frac{2n-2}{m} \rfloor$.

If we have continuous use of the circuit, these durations have again to be lowered by 1 since the *endsort* signal of a computation and the *init* signal of the following one can overlap. At last, an overhead of $2m$ have to be added to the initial time of computation since the first instance of monomials take $2m$ clock ticks to go through the entire array.

The second improvement which might be considered is to represent our Galois field coefficients in linear basis [20, p.112] instead of considering them as powers of a given generator. This point of view enable to achieve bit-pipeline architectures. As a consequence we can reduce the number of pads of the chip and routing wires inside the chip: every galoisian coefficient is then serially represented. Systolic architectures are again especially suited for the galoisian operators and there is

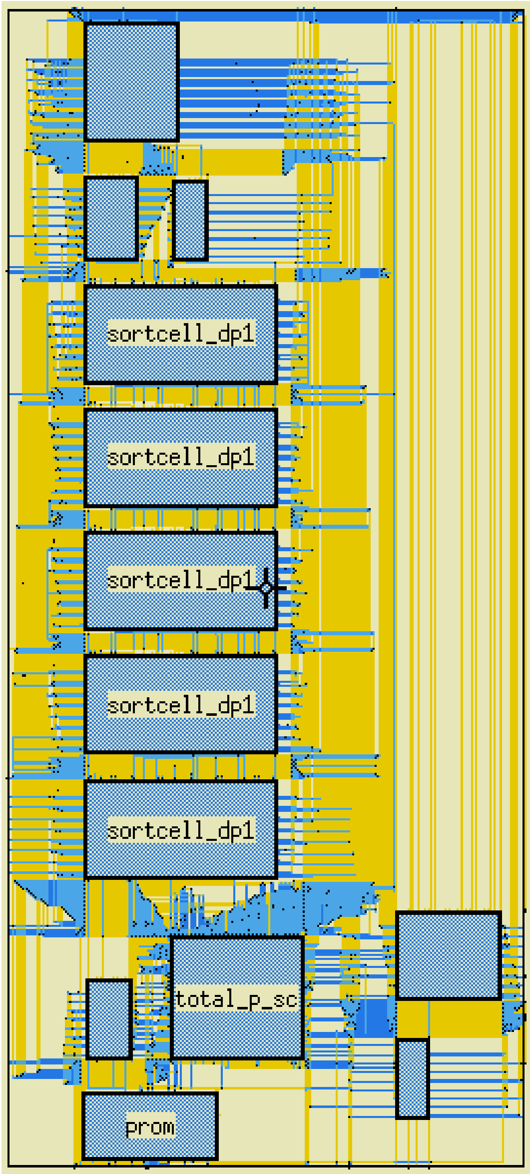

Figure 10: Final design, flat synthesis with five cells and the ROM, automatic placement of the blocs (surface: 11 mm^2).

many works describing systolic multipliers [22, 35, 36]. The integer coefficients can also be serialized (although it may be seen independently from the galoisian coefficients) but there is here an additionnal problem: the addition operators need a serialization from the least significant bit to the most one while the natural way to serialize minima and maxima operators is the opposite one. One solution would be to adopt the convention from least to most significant bits, to convert the serial stream into parallel in front of the comparators, to make a parallel comparison and to convert back the parallel stream into a serial one. This solution has the great drawback to add an overhead at each comparator and thus to arbitrarily divide the clock frequency. Another solution might be to have redundant representations, such as Avezianis'one [37]. The test of equality with 0 needed in the first cell is known to be a dead-end with this representation but maybe it might be considered as an input of the host or preprocessed in another well-suited respresentation.

Some layout improvements are also possible. For instance, the cell *addition* described in figure 4 can be optimized. Its entries u_{x1}, u_{y1}, I_{x1}, I_{y1} on one hand and u_{x2}, u_{y2}, I_{x2}, I_{y2} on the other one are simultaneously active at one tick and simultaneously unactive at the following tick. The results R_{x2} and R_{y2} are then delayed of one clock tick and multiplexed with the output R_{x1} and R_{y1} (see figure 1). According to Leiserson's synchronization theorem, the latches that delayes R_{x2} and R_{y2} can be displaced before the entries u_{x2}, u_{y2}, I_{x2}, I_{y2} without altering the functionality of the cell. But in this case, the four integral adders of the cell `addition` instead of being all active together are active alternatively by pair at each tick. As a consequence we can multiplex them and make savings on the layout of two adders. This optimization comes from a property of the input signals and so cannot be known statically. In our application, this waste has no real importance since the sorting array is occupying most of the layout design and thus this multiplexing would be a minor space optimization. However this emphasizes the fact that if obtaining a functional layout from a high level specification can be made semi-automatically, it is extremely difficult to obtain an optimal one.

At last, it is possible to improve the efficiency and the speed of convergence of the Buchberger's algorithm by using several such chips in parallel and monitoring them with independent computations of S-polynomials and reductions. Such an algorithm can be found for instance in [38].

9 Conclusion

In this article, we described a systolic array dedicated to computation relative to Gröbner basis for two variables polynomials with coefficient in Galois field. The results obtained show that integration into a chip is effectively possible for polynomials of at least 50 variables. The timing reports exhibit performance of approximatively 160 Mbits/s. It has to be noted that this value is indicative: the version of technology available with the version of Compass do not allow much better implementations in general. However, these performances are effective (i.e. the layouts provided in this article would work at such frequency) and allows to address at least many applications with not too high data transmission rate, as e.g. magnetic tapes recording. It is clear, that optimized versions might allow better performances and address intensive communication applications.

It has to be noted that this circuit has been obtained nearly entirely automatically (only optimizations of block placing with Compass have been manually made) from the AlpHard specification.

References

[1] B. Buchberger. *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ideal.* PhD thesis, Math. Inst. University of Innsbruck, Austria, 1965.

[2] M. Sweedler. Using Gröbner bases to determine the algebraic and transcendental nature of field extensions: return of the killer tag variables. In *Proc. 10th International Symposium, AAECC-10*, pages 66–75, 1993.

[3] W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating Gröbner bases. *Journal of Symbolic Computation*, 2(1):83–98, March 1986.

[4] D. Armbruster and H. Kredel. Constructing universal unfoldings using Gröbner bases. *Journal of Symbolic Computation*, 2(4):383–388, December 1986.

[5] L. Robbiano. Term ordering on the polynomial ring. In *EUROCAL 85, Lecture Notes in Computer Science 204*, pages 513–517, April 1985.

[6] B. Buchberger. A survey on the method of Gröbner bases for solving problems in connection with systems of multivariate polynomials. In *Proc. 2nd Int. Symp. on Symbolic and Algebraic Computation by Computers*, pages 7–1–7–15, 1985.

[7] B. Buchberger. *Trends in Computer Algebra, Lecture Notes in Computer Science 296*, chapter Applications of Gröbner bases in non-Linear Computational Geometry, pages 52–80. Springer-Verlag, 1987.

[8] D. Kapur. Using Gröbner bases to reason about geometry. *Journal of Symbolic Computation*, 2(4):399–408, December 1986.

[9] B. Kutzler and S. Stifter. On the application of Buchberger's algorithm to automated geometry theorem proving. *Journal of Symbolic Computation*, 2(4):389–397, December 1986.

[10] F. Pauer and S. Zampieri. Gröbner bases with respect to generalized term orders and their application to the modelling problem. *Journal of Symbolic Computation*, 21(2):155–168, February 1996.

[11] M. P. Saccomani, S. Audoly, L. D'Angio, and C. Cobelli. The Buchberger algorithm to study the a priori identifiability of biological system models. Technical Report of University of Padova, Italy, 1996.

[12] D. Lazard. On the representation of rigid-body motions and its application to generalized platform manipulators. In G. Hommel J. Angeles and P. Kovacs, editors, *Computational Kinematics*, pages 175–182. Kluwer, 1993.

[13] S. Lösch. Parallel redundant manipulator based on open and closed normal Assur chains. In J-P. Merlet and B. Ravani, editors, *Computational Kinematics*, pages 251–260. Kluwer, 1995.

[14] S.M. Jennings. A Gröbner basis view of the Welch-Berlekamp algorithm for Reed-Solomon codes. *IEE Proceedings Communications*, 142(6):349–351, December 1995.

[15] X. Chen, I.S. Reed, T. Helleseth, and T.K. Truong. Use of Gröbner bases to decode binary cyclic codes up to the true minimum distance. *IEEE Transactions on Information Theory*, 40(5):1654–1661, September 1994.

[16] H. Chabanne. Permutation decoding of abelian codes. *IEEE Transactions on Information Theory*, 38(6):1826–1829, November 1992.

[17] L. Robbiano. *The Curves Seminar at Queens's - Volume V*. Queen's University, 1988.

[18] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer-Verlag, 1992.

[19] B. Buchberger. A criterion for detecting unnecessary reductions in the construction of Gröbner bases. In *EUROSAM 79, Lecture Notes in Computer Science 72*, pages 3–21, 1979.

[20] A. Poli and L. Huguet. *Error correcting codes: Theory and applications.* Prentice-Hall, 1992.

[21] R.J. Mac Eliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer, 1987.

[22] C.S. Yeh, I.S. Reed, and T.K. Truong. Systolic multipliers for finite fields GF(2^m). *IEEE Trans. Comp.*, C-33(4):357–360, 1984.

[23] W. W. Peterson. *Error Correcting Codes.* MIT Press, Cambridge, 1972.

[24] H. Brunner, A. Curiger, and M. Hofstetter. On computing multiplicative inverses in GF(2^m). *IEEE Trans. Comp.*, 48(8):1010–1015, 1993.

[25] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[26] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Transactions on Computers*, 35(1):1–12, jan 1986.

[27] D. Wilde and O. Sie. Regular array synthesis using Alpha. Technical Report Internal Publication 829, IRISA, Rennes, France, May 1994.

[28] A. Mukhopadhyay. Weavesort — A new sorting algorithm for VLSI. Technical Report TR-53-81, University of Central Florida, 1981.

[29] M.J. Carey, P.M. Hansen, and C.D. Thompson. RESST: A VLSI implementation of a record-sorting stack. Technical Report CS-248, University of California at Berkeley, 1982.

[30] Y.-H. Choi and M. Malek. A fault-tolerant VLSI sorter. In *IEEE Int. Conf. on Computer Design: VLSI in Computers*, pages 510–513, 1985.

[31] S. Akl and H. Schmeck. Systolic sorting in a sequential input/output environment. *Parallel Computing*, 3:11–23, 1986.

[32] F.S. Wong and M.R. Ito. Parallel sorting on a re-circulating systolic sorter. *The Computer Journal*, 27(3):260–269, 1984.

[33] D. K. Wilde. The Alpha language. Research Report 999, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, Januar 1994.

[34] P. Le Moenner, L. Perraudeau, S. Rajopadhye, T. Risset, and P. Quinton. Generating regular arithmetic circuits with AlpHard. In *Massively Parallel Computing Systems (MPCS'96)*, May 1996.

[35] M. Diab and A. Poli. A new bit-serial systolic multipliers for GF(2^m) using irreducible trinomials. *Electronic Letters*, 27(13):1183–1184, 1991.

[36] M. Kovac, N. Ranganathan, and M. Varasani. SIGMA: A VLSI systolic array implementation of a Galois field GF(2^m) based multiplication and division algorithm. *IEEE Trans. VLSI Systems*, 1:22–30, 1993.

[37] A. Avezianis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400, 1961.

[38] J.C. Faugère. Parallelization of Gröbner basis. In *Parallel Symbolic Computation: 1st Int. Symp.*, pages 124–132, September 1994.

A algorithm for the first cell

The functionnement of the cell can be described by the following algorithm:

Algorithm 3 Input signals at time *t:* $I\lambda_1(t)$, $I\lambda_2(t)$, $Ix_1(t)$, $Iy_1(t)$, $Ix_2(t)$, $Iy_2(t)$, $Redu(t)$ $Iend(t)$, $Iinit(t)$
Internal signals at time *t:* $\Lambda_1(t)$, $\Lambda_2(t)$, $u_{1x}(t)$, $u_{1y}(t)$, $u_{2x}(t)$, $u_{2y}(t)$
$R\lambda_1(t)$, $R\lambda_2(t)$, $Rx_1(t)$, $Ry_1(t)$, $Rx_2(t)$, $Ry_2(t)$,$Valid(t)$
Output signals at time *t:* $O\lambda(t)$, $O\lambda_2(t)$, $Ox(t)$, $Oy(t)$, *Os(t)*, $Oend(t)$, $Oendsort(t)$, $Oinit(t)$

$$Valid(t) \;:=\; \begin{cases} 1 & \textbf{if} \quad Ix_2(t) \geq Ix_1(t) \; and \; Iy_2(t) \geq Iy_1(t) \\ 0 & \textbf{if} \quad Ix_2(t) < Ix_1(t) \; or \; Iy_2(t) < Iy_1(t) \end{cases}$$

$$\Lambda_1(t) \;:=\; \begin{cases} \lambda_2(t) & \textbf{if} \quad Iinit(t) = 1 \\ \Lambda_1(t-1) & \textbf{if} \quad Iinit(t) = 0 \end{cases}$$

$$
\begin{aligned}
\Lambda_2(t) &:= \begin{cases} \lambda_1(t) & \textbf{if} \quad Iinit(t) = 1 \\ \Lambda_2(t-1) & \textbf{if} \quad Iinit(t) = 0 \end{cases} \\
u_{1x}(t) &:= \begin{cases} max(Ix_1(t), Ix_2(t)) - Ix_1(t) & \textbf{if} \quad Iinit(t) = 1 \ and \ Redu(t) = 0 \\ 0 & \textbf{if} \quad Iinit(t) = 1 \ and \ Redu(t) = 1 \\ u_{1x}(t-1) & \textbf{if} \quad Iinit(t) = 0 \end{cases} \\
u_{1y}(t) &:= \begin{cases} max(Iy_1(t), Iy_2(t)) - Iy_1(t) & \textbf{if} \quad Iinit(t) = 1 \ and \ Redu(t) = 0 \\ 0 & \textbf{if} \quad Iinit(t) = 1 \ and \ Redu(t) = 1 \\ u_{1y}(t-1) & \textbf{if} \quad Iinit(t) = 0 \end{cases} \\
u_{2x}(t) &:= \begin{cases} max(Ix_1(t), Ix_2(t)) - Ix_2(t) & \textbf{if} \quad Iinit(t) = 1 \ and \ Redu(t) = 0 \\ Ix_2(t) - Ix_(t)1 & \textbf{if} \quad Iinit(t) = 0 \ and \ Redu(t) = 1 \\ u_{2x}(t-1) & \textbf{if} \quad Iinit(t) = 1 \end{cases} \\
u_{2y}(t) &:= \begin{cases} max(Iy_1(t), Iy_2(t)) - Iy_2(t) & \textbf{if} \quad Iinit(t) = 1 \ and \ Redu(t) = 0 \\ Iy_2(t) - Iy_1(t) & \textbf{if} \quad Iinit(t) = 1 \ and \ Redu(t) = 1 \\ u_{2y}(t-1) & \textbf{if} \quad Iinit(t) = 1 \end{cases} \\
R\lambda_1(t) &:= \Lambda_1(t) \otimes \lambda_1(t) \\
R\lambda_2(t) &:= \Lambda_2(t) \otimes \lambda_2(t) \\
Rx_1(t) &:= Ix_1(t) + u_{1x}(t) \\
Ry_1(t) &:= Iy_1(t) + u_{1y}(t) \\
Rx_2(t) &:= Ix_2(t) + u_{2x}(t) \\
Ry_2(t) &:= Iy_2(t) + u_{2y}(t) \\
O\lambda_2(t) &:= 0 \\
O\lambda_1(t) &:= \begin{cases} R\lambda_1(t) & \textbf{if} \quad ck2(t) = 1 \\ R\lambda_2(t-1) & \textbf{if} \quad ck2(t) = 0 \end{cases} \\
Ox(t) &:= \begin{cases} Rx_1(t) & \textbf{if} \quad ck2(t) = 1 \\ Rx_2(t-1) & \textbf{if} \quad ck2(t) = 0 \end{cases} \\
Oy(t) &:= \begin{cases} Ry_1(t) & \textbf{if} \quad ck2(t) = 1 \\ Ry_2(t-1) & \textbf{if} \quad ck2(t) = 0 \end{cases} \\
Oend(t) &:= Iend(t-1); \\
Oendsort(t) &:= Iend(t-1); \\
Oinit(t) &:= Iinit(t-1); \\
Os(t) &:= \begin{cases} 1 & \textbf{if} \quad O\lambda_1(t) \neq 0 \\ 0 & \textbf{if} \quad O\lambda_1(t) = 0 \end{cases}
\end{aligned}
$$

B Algorithm for the sorting cells

Algorithm 4
Input signals at time t: $I\lambda_1(t)$, $I\lambda_2(t)$, $Ix(t)$, $Iy(t)$, $Is(t)$, $Iend(t)$, $Iendsort(t)$, $Iinit(t)$
Internal registers at time t: $R\lambda_1(t)$, $R\lambda_2(t)$, $Rx(t)$, $Ry(t)$, $Rs(t)$, $Rend(t)$
Output signals at time t: $O\lambda_1(t)$, $O\lambda_2(t)$, $Ox(t)$, $Oy(t)$, $Os(t)$, $Oend(t)$, $Oendsort(t)$, $Oinit(t)$

$$O\lambda_1(t) := \begin{cases} R\lambda_1(t-1) & \textbf{if} \quad Iendsort(t-1)=1 \\ I\lambda_1(t-1) & \textbf{if} \quad Iendsort(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=1 \textbf{ or } Rend(t-1)=1) \\ I\lambda_1(t-1) & \textbf{if} \quad Iendsort(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=1 \textbf{ and } Rs(t-1)=1 \textbf{ and} \\ & \qquad [cmp >](t-1)=1 \\ R\lambda_1(t-1) & \textbf{if} \quad Iendsort(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=1 \textbf{ and } Rs(t-1)=1 \textbf{ and} \\ & \qquad [cmp <](t-1)=1 \end{cases}$$

$$O\lambda_2(t) := \begin{cases} R\lambda_2(t-1) & \textbf{if} \quad Iendsort(t-1)=1 \\ I\lambda_2(t-1) & \textbf{if} \quad Iendsort(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=1 \textbf{ or } Rend(t-1)=1) \\ I\lambda_2(t-1) & \textbf{if} \quad Iendsort(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=1 \textbf{ and } Rs(t-1)=1 \textbf{ and} \\ & \qquad [cmp >](t-1)=1 \\ R\lambda_2(t-1) & \textbf{if} \quad Iendsort(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=1 \textbf{ and } Rs(t-1)=1 \textbf{ and} \\ & \qquad [cmp <](t-1)=1 \end{cases}$$

$$Ox(t) \quad := \quad \begin{cases} Rx(t-1) & \textbf{if} \quad Iendsort(t-1) = 1 \\ Ix(t-1) & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 1 \textbf{ or } Rend(t-1) = 1) \\ Ix(t-1) & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp >](t-1) = 1 \\ Rx(t-1) & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp <](t-1) = 1 \end{cases}$$

$$Oy(t) \quad := \quad \begin{cases} Ry(t-1) & \textbf{if} \quad Iendsort(t-1) = 1 \\ Iy(t-1) & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 1 \textbf{ or } Rend(t-1) = 1) \\ Iy(t-1) & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp >](t-1) = 1 \\ Ry(t-1) & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp <](t-1) = 1 \end{cases}$$

$$Os(t) := \begin{cases} Rs(t-1) & \textbf{if} \quad Iendsort(t-1) = 1 \\ Is(t-1) & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 1 \textbf{ or } Rend(t-1) = 1) \\ 0 & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 0 \\ 0 & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 0 \\ 1 & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp =](t-1) = 0 \\ 0 & \textbf{if} \quad Iendsort(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp =](t-1) = 1 \end{cases}$$

$$Oend(t) := Iend(t-1)$$

$$Oendsort(t) := Iendsort(t-2)$$

$$Oinit(t) := Iinit(t-2)$$

$$R\lambda_1(t) := \begin{cases} R\lambda_1(t-1) & \textbf{if} \quad (Iend(t-1) = 1 \textbf{ or } Rend(t-1) = 1) \\ R\lambda_1(t-1) & \textbf{if} \quad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 0 \\ I\lambda_1(t-1) & \textbf{if} \quad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 0 \\ I\lambda_1(t-1) & \textbf{if} \quad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp <](t) = 1 \\ R\lambda_1(t-1) & \textbf{if} \quad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp <](t) = 0 \end{cases}$$

$$R\lambda_2(t) \quad := \quad \begin{cases} 0 & \textbf{if} \quad Iinit(t-1) = 1 \\ R\lambda_2(t-1) & \textbf{if} \quad Iinit(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 1 \textbf{ or } Rend(t-1) = 1) \\ R\lambda_2(t-1) & \textbf{if} \quad Iinit(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 0 \\ R\lambda_2(t-1) & \textbf{if} \quad Iinit(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 0 \\ I\lambda_1(t-1) & \textbf{if} \quad Iinit(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp =](t-1) = 1 \\ I\lambda_2(t-1) & \textbf{if} \quad Iinit(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp <](t-1) = 1 \\ R\lambda_2(t-1) & \textbf{if} \quad Iinit(t-1) = 0 \textbf{ and} \\ & \qquad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp >](t-1) = 1 \end{cases}$$

$$Rx(t) \quad := \quad \begin{cases} Rx(t-1) & \textbf{if} \quad (Iend(t-1) = 1 \textbf{ or } Rend(t-1) = 1) \\ Rx(t-1) & \textbf{if} \quad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 0 \\ Ix(t-1) & \textbf{if} \quad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 0 \\ Ix(t-1) & \textbf{if} \quad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp <](t-1) = 1 \\ Rx(t-1) & \textbf{if} \quad (Iend(t-1) = 0 \textbf{ and } Rend(t-1) = 0) \textbf{ and} \\ & \qquad Is(t-1) = 1 \textbf{ and } Rs(t-1) = 1 \textbf{ and} \\ & \qquad [cmp <](t-1) = 0 \end{cases}$$

$$Ry(t) := \begin{cases} Ry(t-1) & \textbf{if} \quad (Iend(t-1)=1 \textbf{ or } Rend(t-1)=1) \\ Ry(t-1) & \textbf{if} \quad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=0 \\ Iy(t-1) & \textbf{if} \quad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=1 \textbf{ and } Rs(t-1)=0 \\ Iy(t-1) & \textbf{if} \quad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=1 \textbf{ and } Rs(t-1)=1 \textbf{ and} \\ & \qquad [cmp<](t-1)=1 \\ Ry(t-1) & \textbf{if} \quad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=1 \textbf{ and } Rs(t-1)=1 \textbf{ and} \\ & \qquad [cmp<](t-1)=0 \end{cases}$$

$$Rend(t) := \begin{cases} 0 & \textbf{if} \quad Iinit(t-1)=1 \\ 1 & \textbf{if} \quad Iinit(t-1)=0 \textbf{ and } Iend(t-1)=1 \\ Rend(t-1) & \textbf{if} \quad Iinit(t-1)=0 \textbf{ and } Iend(t-1)=0 \end{cases}$$

$$Rs(t) := \begin{cases} 0 & \textbf{if} \quad Iinit(t-1)=1 \\ Rs(t-1) & \textbf{if} \quad Iinit(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=1 \textbf{ or } Rend(t-1)=1) \\ 1 & \textbf{if} \quad Iinit(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad Is(t-1)=0 \textbf{ and } Rs(t-1)=1 \\ Rs(t-1) & \textbf{if} \quad Iinit(t-1)=0 \textbf{ and} \\ & \qquad (Iend(t-1)=0 \textbf{ and } Rend(t-1)=0) \textbf{ and} \\ & \qquad (Is(t-1)=1 \textbf{ or } Rs(t-1)=0) \end{cases}$$

$$[cmp=](t) := \begin{cases} 1 & \textbf{if} \quad (Rx(t)=Ix(t)) \textbf{ and } (Ry(t)=Iy(t)) \\ 0 & \textbf{if} \quad (Rx(t)\neq Ix(t)) \textbf{ or } (Ry(t)\neq Iy(t)) \end{cases}$$

$$[cmp<](t) := \begin{cases} 1 & \textbf{if} \quad (Rx(t)<Ix(t)) \\ 1 & \textbf{if} \quad (Rx(t)=Ix(t)) \textbf{ and } (Ry(t)<Iy(t)) \\ 0 & \textbf{if} \quad (Rx(t)>Ix(t)) \\ 0 & \textbf{if} \quad (Rx(t)=Ix(t)) \textbf{ and } (Ry(t)\geq Iy(t)) \end{cases}$$

$$[cmp>](t) := \begin{cases} 1 & \textbf{if} \quad (Rx(t)>Ix(t)) \\ 1 & \textbf{if} \quad (Rx(t)=Ix(t)) \textbf{ and } (Ry(t)>Iy(t)) \\ 0 & \textbf{if} \quad (Rx(t)<Ix(t)) \\ 0 & \textbf{if} \quad (Rx(t)=Ix(t)) \textbf{ and } (Ry(t)\leq Iy(t)) \end{cases}$$