HAL
archives-ouvertes.fr

The Palindrome systolic array revisited

Hervé Le Verge, Patrice Quinton

▶ **To cite this version:**

Hervé Le Verge, Patrice Quinton. The Palindrome systolic array revisited. [Research Report] RR-1578, INRIA. 1992. <inria-00074982>

HAL Id: inria-00074982
https://hal.inria.fr/inria-00074982

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1578

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

THE PALINDROME SYSTOLIC ARRAY REVISITED

Hervé LE VERGE
Patrice QUINTON

Janvier 1992

RR-1578

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Retour sur le réseau systolique du palindrome
The palindrome systolic array revisited*

Hervé Le Verge et Patrice Quinton
IRISA-CNRS, Campus de Beaulieu
35042 Rennes Cedex, France
e-mail : quinton@irisa.fr

A paraître dans les actes du colloque international
Research Direction in High-Level Parallel Programming Languages
Springer-Verlag.

Novembre 1991
Publication Interne n° 618 - 14 pages - Programme 1

Résumé: Le but de cet article est de présenter le langage ALPHA, et son utilisation pour la synthèse de réseaux réguliers. Pour ce faire, on considère le fameux exemple du reconnaisseur de chaînes palindromes, qui a servi de support à l'illustration de plusieurs méthodes de synthèse. On commence par résumer les principes de ALPHA. Puis, on explique en détail la synthèse du reconnaisseur de palindromes temps-réel, illustrant ainsi les possibilités de ALPHA.

Abstract: The purpose of this article is to present a language, ALPHA, and its use for the synthesis of regular arrays. To this end, we consider the famous example of the palindrome recognizer, which has served as a support for the illustration of various design methodologies. We first summarize the characteristics of the ALPHA language. Then we explains in detail the synthesis of a real-time palindrome recognizer, thus illustrating the potential of ALPHA.

*Ces recherches sont partiellement soutenues par le programme de recherches coordonnées du MRT C^3, et par le projet ESPRIT BRA NANA, numéro 3280

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (URA 227) UNIVERSITÉ DE RENNES I INSA DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITÉ DE RECHERCHE DE RENNES)

The palindrome systolic array revisited*

Hervé Le Verge and Patrice Quinton

IRISA-CNRS, Campus de Beaulieu, 35042 Rennes Cedex, France
e-mail : quinton@irisa.fr

1 Introduction

As density of integrated circuit increases, parallel arrays – i.e. systolic arrays[Kun82], wavefront arrays[KAGR82], regular iterative arrays[Rao85], to name a few types of such architectures – become one of the favorite architectural style of special-purpose system designers. The reasons are well known. Parallel arrays have high performances and are modular, and these properties make them well suited for implementing systems commonly found in many application areas. On the other hand, the main concern of system designers is to produce as quickly as possible a special-purpose system – i.e. a combination of hardware and software – without default. Parallel regular arrays are complex algorithms, thus difficult to master. This often leads to errors in the design process.

The purpose of this article is to present a language, ALPHA, and its use for the synthesis of regular arrays. To this end, we consider the famous example of the palindrome recognizer, which has served as a support for the illustration of various design methodologies. This algorithm is considered by Cole[Col69], who describes a systolic array. Leiserson and Saxe[LS81] consider this example to illustrate their Systolic Conversion Theorem. More recently, Van de Snepscheut and Swenker[dSS89] also investigate the derivation of parallel algorithms for the palindrome recognition, by means of stepwise refinement method.

In Sect. 2, we first summarize the characteristics of the ALPHA language. Then, Sect. 3 explains in detail the synthesis of a real-time palindrome recognizer, thus illustrating the potential of ALPHA.

2 The ALPHA language

The principles of ALPHA have been presented in detail elsewhere[DVQS91, DGL*91, LMQ91]. The purpose of this section is just to introduce the non familiar reader with the notations and main ideas of the language.

The ALPHA language is based on the recurrence equation formalism. It is therefore an *equational language*, whose constructs are well-suited to the expression of regular

* This work was partially funded by the French Coordinated Research Program C^3 and by the Esprit BRA project NANA.

algorithms. The ALPHA language can also be used to describe *synchronous systems*, and therefore, provides a natural framework for the transformation of *algorithm specifications* into *architectures*. Interactive transformations of ALPHA programs can be done using the ALPHA DU CENTAUR environment, implemented with the language design system CENTAUR[BCD*87]. ALPHA DU CENTAUR includes a library of mathematical routines that are used to search efficient transformations of programs.

2.1 The basics

An ALPHA program is a collection of single assignment equations. ALPHA follows the classical principles of a *structured, strongly typed functional language*.

To explain the language, let us consider the ALPHA program, also called a *system of equations*, presented in Fig. 1 which represents an iterative version of the calculation

```
system example ( X : {i|1 ≤ i ≤ 3} of integer )
returns ( s : integer );
var
sum : {i|0 ≤ i ≤ 3} of integer ;
let
sum = case
        {i|i = 0} : 0.(i →);
        {i|1 ≤ i ≤ 3} : X + sum.(i → i − 1);
        esac;
s   = sum.(→ 3);
tel ;
```

Fig. 1. Example of ALPHA program

$s = \sum_{i=1}^{3} X_i$. This program takes an input variable X, indexed on the set $\{i|1 \leq i \leq 3\}$ of integer, and returns an integer s. Moreover, there is local variable *sum*, defined on the set $\{i|0 \leq i \leq 3\}$. Between the keywords **let** and **tel**, we find the definition of *sum* and s. Each definition provides a synonymy between a variable and an ALPHA expression. ALPHA variables and expressions are in fact functions from an index domain of $\mathbb{Z}^n$ - the *spatial domain* of the variable or the expression- to a set of values of a given type (boolean, integer, real, in the current version.) Spatial domains of ALPHA variables are restricted to integral points of convex polyhedral domains (see [Sch86] for notions on convex polyhedra).

2.2 Motionless and spatial operators

ALPHA expressions are obtained by combining variables (or recursively, expressions) together with two sorts of operators : *motionless operators* and *spatial operators*.

Motionless operators are the generalization of classical operators to ALPHA expressions. Operators defined in such a way are usual unary and binary operators on basic types, and the conditional operator **if..then..else**. As an example, given one-dimensional

variables X and Y, the expression $X + Y$ represents a function defined on the intersection of the domains of X and Y, and whose value at index i is $X_i + Y_i$.

Spatial operators are the only operators which operate explicitly on spatial domains. The *dependence operator* combines *dependence functions* and expressions. Dependence functions are affine mapping between spatial domains, and are denoted $(i, j, ... \rightarrow f(i, j, ..))$ where f is an affine mapping. Given an expression E and a dependence function *dep*, $E.dep$ denotes the composition of functions E and *dep*. As an example, the expression $sum.(i \rightarrow i - 1)$ denotes the expression whose i-th element is sum_{i-1}. Note that constants are defined on $\mathbb{Z}^0$, and $(\rightarrow i)$ denotes the mapping from $\mathbb{Z}^0$ to $\mathbb{Z}$: the definition $s = sum.(\rightarrow 3)$ in Fig. 1 says that s is sum_3 (the value of *sum* at index 3.) The *restriction* operator restricts the domain of an expression, by means of linear constraints. In Fig. 1, the expression $\{i|1 \leq i \leq 3\} : X + sum.(i \rightarrow i - 1)$ restricts the domain of $X + sum.(i \rightarrow i - 1)$ to the segment $[1, 3]$. The **case** operator combines expressions defined on disjoint domains into a new expression, as the variable *sum* of program shown in Fig. 1.

The spatial operators allow recurrence equations to be expressed. In Fig. 1, the value of the variable *sum* is the sequence of partial sums of the elements of X and is defined by means of a **case**, whose first branch specifies the initialization part and the second one the recurrence itself.

2.3 Basic transformations

As any functional language, ALPHA follows the substitution principle : any variable can be substituted by its definition, without changing the meaning of the program. Substituting *sum* in the definition of s in program of Fig. 1, gives the program shown in Fig. 2. One

```
system example ( X : {i|1 ≤ i ≤ 3} of integer )
returns ( s : integer );
var
sum : {i|0 ≤ i ≤ 3} of integer ;
let
sum = case
        {i|i = 0} : 0.(i →);
        {i|i > 0} : X + sum.(i → i − 1);
        esac;
s   = (case
         {i|i = 0} : 0.(i →);
         {i|i > 0} : X + sum.(i → i − 1);
         esac).(→ 3);
tel ;
```

Fig. 2. Program 1 after substituting sum by its definition

can show that any ALPHA expression can be rewritten in an equivalent expression, called its *normal form*, whose structure is composed of a unique **case**, and all dependencies are directly associated with variables and constants. This normal form is also called *Case-Restriction-Dependence* form. The normalization process often simplifies an expression,

and can be used, together with the substitution, to do a symbolic simulation of an ALPHA program. For example, the definition of s in program (2) becomes after normalization :

$$s = X.(\rightarrow 3) + sum.(\rightarrow 2);$$

and by repeating this process :

$$s = X.(\rightarrow 3) + (X.(\rightarrow 2) + (X.(\rightarrow 1) + 0.(\rightarrow)));$$

which is just the definition of s.

A *change of basis* can be applied to the index space of any local variable, using a straightforward syntactic transformation of the equations [LMQ90]: in order to apply the change of basis defined by a unimodular dependence function *dep* to a variable X, one needs to replace the definition domain of X by its image by *dep*, to replace right-hand side occurrences of X by $X.dep$, and finally to replace the equation $X = exp$ by $X = exp.dep^{-1}$. The case when *dep* is not unimodular can be dealt with by first embedding the variable X in a higher dimensional index space. The change of basis transformation is the core of *space-time reindexing*, as will be shown below.

3 Synthesis of a real-time palindrome recognizer

The goal of the following ALPHA exercise is to show how the classical systolic palindrome array described in [Col69] can be synthesized from "as high-level a specification as possible". After describing this initial specification, we outline each one of the transformations needed to reach an ALPHA program "reasonably close" to the hardware description of the solution. All transformations, but a few ones that we will mention, were performed using ALPHA DU CENTAUR, that is to say, fully automatically. However, the choice of the transformations to be applied and the order of their applications was manual. We should emphasize that the goal was *not* to find out a new palindrome recognizer, but rather to prove by construction the correctness of an implementation of the classical solution.

3.1 The problem and its initial specification

Let $a = a_0.....a_{n-1}$ be a string of n characters. It is said to be a *palindrome* if, for all i, $0 \leq i \leq n-1$, a_i is equal to a_{n-i-1}. The problem we want to solve is to find out a *real-time palindrome recognizer*, that is to say, a device which reads a_i in increasing order of i, and answers immediately after reading a_{n-1} whether $a_0....a_{n-1}$ is a palindrome or not. From this informal description of the algorithm, we get the first ALPHA program of Fig. 3. The program takes the string a as input, and returns a boolean function *pal*, defined as

$$pal_n = \bigwedge_{0 \leq i < (n-1)/2} (a_i = a_{n-i-1}). \qquad (1)$$

Without loss of generality, the actual program presents the derivation of a *bounded* palindrome array, able only to handle strings of at most 8 symbols. The definition of *pal* uses the reduction operator **red** od ALPHA, whose precise description and operation are beyond the scope of this paper[Lev91].

```
system palindrome ( a : {i|7 ≥ i ≥ 0} of integer )
returns ( pal : {n|n ≥ 1} of integer );

let
pal = red( ∧ , (i, n → n) , {i, n|8 ≥ n ≥ 2i + 2} : a.(i, n → i) = a. (i, n → −i + n − 1) );
tel ;
```

Fig. 3. Initial specification of the palindrome algorithm

3.2 Serialization of the reduction operator

Figure 4 shows the program, after replacing the reduction operator by a recurrence. To do so, we need to introduce a new variable, p, defined over a domain of dimension 2. This variable is defined by a recurrence, initialized with the null element of $\wedge$, i.e., **true**. The recurrence is done by decreasing value of the index i in the reduction of equation (1). The result pal of the program is now defined by $pal_n = p_{0,n}$.

```
system palindrome ( a : {i|i ≥ 0; 7 ≥ i} of integer )
returns ( pal : {n|n ≥ 1} of integer );
var
p : {i, n|i ≥ 0; 8 ≥ n; n ≥ 2i + 2} ,
    {i, n|i ≥ 1; 2i + 1 ≥ n; n ≥ 2i; 8 ≥ n} of boolean ;
let
pal = {i|8 ≥ i; i ≥ 2} : p.(i → 0, i);
p   = case
      {i, n|i ≥ 1; 2i + 1 ≥ n; n ≥ 2i; 8 ≥ n} : true.(i, n →);
      {i, n|i ≥ 0; 8 ≥ n; n ≥ 2i + 2} : p.(i, n → i + 1, n) ∧
      a.(i, n → i) = a. (i, n → −i + n − 1);
      esac;
tel ;
```

Fig. 4. Version after serialization of $\bigwedge$

3.3 Uniformization

The next transformation, referred to as *uniformization*, *pipelining*, or *localization* in the literature, is rather complex. The definition of p in the program of Fig. 4 contains two instances of the variable a which are not two-dimensional. The uniformization transformation aims to replace these instances by new variables $A1$ and $A2$, which are defined by induction on the domain of the variable p, in such a way that the arguments of the equation be defined on the same domain, thus leading to *uniform recurrence equations*[KMW67]. The interested reader will find in [QD89] details on this transformation.

```
system palindrome ( a : {i|i ≥ 0; 7 ≥ i} of integer )
returns ( pal : {n|n ≥ 1} of integer );
var
A2 : {i, n|i ≥ 0; 8 ≥ n; n ≥ 2i + 1} of integer;
A1 : {i, n|i ≥ 0; 8 ≥ n; n ≥ 2i + 2} of integer;
p  : {i, n|i ≥ 0; 8 ≥ n; n ≥ 2i + 2} ,
     {i, n|n ≥ 2i; i ≥ 1; 8 ≥ n; 2i + 1 ≥ n} of boolean ;
let
pal = {i|8 ≥ i; i ≥ 2} : p.(i — 0, i);
p   = case
      {i, n|n ≥ 2i; i ≥ 1; 8 ≥ n; 2i + 1 ≥ n} : true.(i, n →);
      {i, n|i ≥ 0; 8 ≥ n; n ≥ 2i + 2} : p.(i, n → i + 1, n) ∧ A1 = A2;
      esac;
A1  = case
      {i, n|i ≥ 0; 3 ≥ i; n = 2i + 2} : a.(i, n → i);
      {i, n|i ≥ 0; 8 ≥ n; n ≥ 2i + 3} : A1.(i, n → i, n − 1);
      esac;
A2  = case
      {i, n|i = 0; 8 ≥ n; n ≥ 1} : a.(i, n → −i + n − 1);
      {i, n|8 ≥ n; i ≥ 1; n ≥ 2i + 1} : A2.(i, n → i − 1, n − 1);
      esac;
tel ;
```

Fig. 5. Version after uniformization

3.4 Connecting *A1* and *A2*

The uniformization of the occurrences of a in the definition of p done in Sect. 3.3, has the effect of introducing two different flows of a data: one for the variable *A1*, and one for *A2*. This situation, although perfectly correct, is undesirable from the point of view of the architecture design, as it will result in two flows of data carrying the same values. To avoid this problem, one can "connect" these flows, by noticing that the initial values of *A1*, defined in the branch

$$\{i, n|i \geq 0; 3 \geq i; n = 2i + 2\} : \text{a}.(i, n \rightarrow i);$$

of the case expression, are in fact equal to $A2.(i, n \rightarrow i, n-1)$. The result is shown in Fig. 6. This rather heuristic transformation cannot be done automatically. However, one can

```
A1 = case
     {i, n|i ≥ 0; 3 ≥ i; n = 2i + 2} : A2.(i, n → i, n − 1);
     {i, n|i ≥ 0; 8 ≥ n; n ≥ 2i + 3} : A1.(i, n → i, n − 1);
     esac;
```

Fig. 6. Connection *A1* and *A2*

prove that the resulting program is equivalent by repeated substitution and normalization of *A2* in the new equation.

3.5 Initialization of p

A similar transformation has to be applied to the initialization part of the equation which defines p. The idea behind this transformation is, by anticipating the final shape of the architecture, to avoid broadcasting an initialization control signal to all the cells of the architecture. To this end, the initialization part of the definition of p is split in two sub-equations, each one defined on one segment domain (see Fig. 7). Then, uniformization is applied on each new equation.

```
p = case
    {i,n|7 ≥ 2i; i ≥ 1; 2i + 1 = n} : true.(i,n →);
    {i,n|i ≥ 1; 4 ≥ i; n = 2i} : true.(i,n →);
    {i,n|i ≥ 0; 8 ≥ n; n ≥ 2i + 2} : p.(i,n → i + 1,n) ∧ A1 = A2;
    esac;
```

Fig. 7. Splitting the initialization part of the definition of p

```
system palindrome ( a : {i|i ≥ 0; 7 ≥ i} of integer )
returns ( pal : {n|n ≥ 1} of integer );
var
init2 :   {i,n|i ≥ 1; 4 ≥ i; n = 2i} of boolean;
init1 :   {i,n|7 ≥ 2i; i ≥ 1; 2i + 1 = n} of boolean;
    ...
let
p    =  case
        {i,n|7 ≥ 2i; i ≥ 1; 2i + 1 = n} : init1;
        {i,n|i ≥ 1; 4 ≥ i; n = 2i} : init2;
        {i,n|i ≥ 0; 8 ≥ n; n ≥ 2i + 2} : p.(i,n → i + 1,n) ∧ A1 = A2;
        esac;
    ...
init1 =  case
        {i,n|3 = n; 1 = i} : true.(i,n →);
        {i,n|7 ≥ 2i; i ≥ 2; 2i + 1 = n} : init1.(i,n → i − 1,n − 2);
        esac;
init2 =  case
        {i,n|2 = n; 1 = i} : true.(i,n →);
        {i,n|i ≥ 2; 4 ≥ i; n = 2i} : init2.(i,n → i − 1,n − 2);
        esac;
tel ;
```

Fig. 8. Uniformization of initialization signals

3.6 Embedding and change of basis

The last transformation of the program is known in the literature as *space-time reindexing*. It corresponds to finding a time axis and a processor axis in the index space, in such a way that the resulting system of equations represent the operation of a synchronous architecture. Techniques to do this are well-known (see [Mol82], among many others). In the present case, the time component, i.e, the time at which calculation (i, n) is done, is $t(i, n) = 2n - i$, and the space component, i.e. the number of the processor calculating (i, n) is simply i. In term of ALPHA program transformation, space-time reindexing amounts to operate a change of basis, as described in Subsect. 2.3. However, a careful analysis of the dependencies reveals that the period of the cells of the desired architecture is 2, i.e., each cell operates only every other tick of the clock. As a consequence, one cannot use directly a *unimodular* change of basis. To circumvent this problem, one uses the following trick : first, local variables are *embedded* in a three-dimensional space, simply by adding a new index (k for example), arbitrarily set to 0. The effect of this embedding is illustrated in Fig. 9 for variable *A2*. Then, one performs a unimodular change

```
system palindrome ( a : {i|i ≥ 0; 7 ≥ i} of integer )
returns ( pal : {n|n ≥ 1} of integer );
var
   ...
 A2 :   {i, n, k|i ≥ 0; 8 ≥ n; n ≥ 2i + 1; k = 0} of integer;
   ...
let
   ...
 A2 =  case
       {i, n, k|i = 0; 8 ≥ n; n ≥ 1; k = 0} : a. (i, n, k → −i + n − 1);
       {i, n, k|8 ≥ n; i ≥ 1; n ≥ 2i + 1; k = 0} : A2.(i, n, k → i − 1, n − 1, 0);
       esac;
   ...
tel ;
```

Fig. 9. After embedding

of basis, chosen in such a way that its *projection* on the first two indexes correspond to the desired, non unimodular, space-time transformation. In our example, the change of basis we are looking for is $(t, p, k) = (2n - i + k, i, n + k)$. The same change of basis is performed on all variables, except *init1* and *init2* which are additionally translated by $(t, p, k) = (-3, -1, 0)$ in such a way that the initialization signal enter cell number 0 of the array. The result is shown in Fig. 10. It can readily be interpreted as the systolic architecture depicted in Fig. 11. The array has *period* 2, and uses $n/2$ cells. Notice that the initialization of the cells is *fully systolic*, as the operation of the array makes no assumption on the initial state of the registers of the cells: all data and control signals enter the array in cell 0, and results are obtained in real-time in cell 0 as well.

```
system palindrome ( a : {i|i ≥ 0; 7 ≥ i} of integer )
returns ( pal : {n|n ≥ 1} of integer );
var
init2 : {t, p, k|8 ≥ k; 2p + 2 = k; k ≥ 2; 2t + 6 = 3k} of boolean;
init1 : {t, p, k|19 ≥ 2t; 3p + 2 = t; t ≥ 2; 2t + 5 = 3k} of boolean;
A2   : {t, p, k|8 ≥ k; 2t ≥ 3k + 1; 2k = t + p; 2k ≥ t} of integer;
A1   : {t, p, k|p ≥ 0; t ≥ 3p + 4; 16 ≥ t + p; 2k = t + p} of integer;
p    : {t, p, k|2k ≥ t + 1; 2t ≥ 3k; 2k = t + p; 3k + 1 ≥ 2t; 8 ≥ k} ,
       {t, p, k|p ≥ 0; t ≥ 3p + 4; 16 ≥ t + p; 2k = t + p} of boolean ;
let
pal  = {i|8 ≥ i; i ≥ 2} : p.(i → 2i, 0, i);
p    = case
       {t, p, k|25 ≥ 2t; t = 3p + 2; t ≥ 5; 2t = 3k + 1} : init1.(t, p, k → t − 3, p − 1, k);
       {t, p, k|t ≥ 3; 3p = t; 12 ≥ t; 3k = 2t} : init2.(t, p, k → t − 3, p − 1, k);
       {t, p, k|p ≥ 0; t ≥ 3p + 4; 16 ≥ t + p; 2k = t + p} :
              p.(t, p, k → t − 1, p + 1, k) ∧ A1 = A2;
       esac;
A1   = case
       {t, p, k|13 ≥ t; 3p + 4 = t; t ≥ 4; 2t = 3k + 2} : A2.(t, p, k → t − 2, p, t + p − k − 1);
       {t, p, k|p ≥ 0; t ≥ 3p + 6; 16 ≥ t + p; 2k = t + p} : A1.(t, p, k → t − 2, p, k − 1);
       esac;
A2   = case
       {t, p, k|t ≥ 2; p = 0; 16 ≥ t; 2k = t} : a.(t, p, k → t − k − 1);
       {t, p, k|p ≥ 1; t ≥ 3p + 2; 16 ≥ t + p; t + p = 2k} : A2.(t, p, k → t − 1, p − 1, k − 1);
       esac;
init1 = case
       {t, p, k|3 = k; p = 0; 2 = t} : true.(t, p, k →);
       {t, p, k|t ≥ 5; 3p + 2 = t; 19 ≥ 2t; 3k = 2t + 5} :
              init1.(t, p, k → t − 3, p − 1, t + p − k + 2);
       esac;
init2 = case
       {t, p, k|t = 0; p = 0; 2 = k} : true.(t, p, k →);
       {t, p, k|9 ≥ t; t = 3p; t ≥ 3; 2t + 6 = 3k} : init2.(t, p, k → t − 3, p − 1, t + p − k + 2);
       esac;
tel ;
```

Fig. 10. The final ALPHA program

4 Conclusion

We have described the ALPHA language, and illustrated its use for the derivation of a palindrome real-time recognizer. The ALPHA DU CENTAUR environment includes a translator to the input language of a standard cell VLSI generator, which accepts as input a subset of ALPHA very similar to the final version of the palindrome. Its use for the automatic synthesis of a systolic correlator is reported in [DGL*91]. Our experience with ALPHA has shown us that it is a very concise means of describing regular algorithms, and of deriving correct parallel arrays for these algorithms. In particular, the possibility of expressing all steps of the algorithm transformations using a unique language is very convenient.

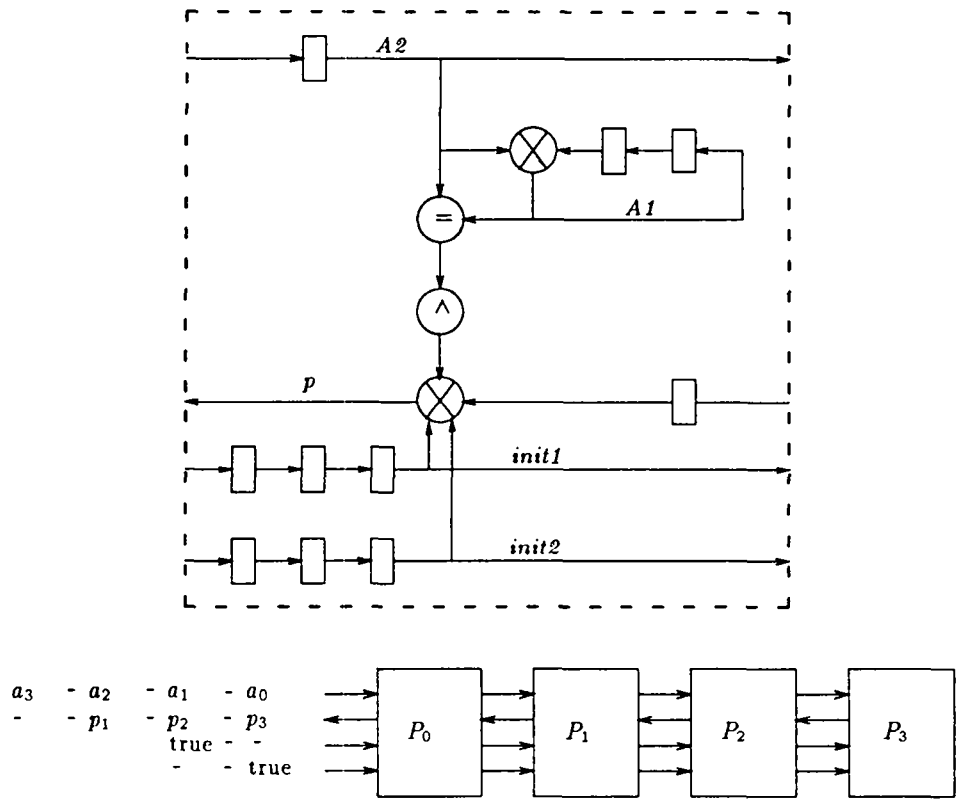

Fig. 11. Cell structure and architecture

References

[BCD*87] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. *CENTAUR: the System.* Technical Report 777, INRIA, 1987.

[Col69] S.N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Tr. on Computer*, 18(4):349–365, 1969.

[DGL*91] C. Dezan, E. Gautrin, H. Leverge, P. Quinton, and Y. Saouter. Synthesis of systolic arrays by equation transformations. In *ASAP'91*, IEEE, Barcelona, Spain, September 1991.

[dSS89] J.L.A. Van de Snepscheut and J.B. Swenker. On the design of some systolic algorithms. *JACM*, 36:826–840, 1989.

[DVQS91] C. Dezan, H. Le Verge, P. Quinton, and Y. Saouter. The ALPHA DU CENTAUR environment. In P. Quinton and Y. Robert, editors, *International Workshop Algorithms and Parallel VLSI Architectures II*, North-Holland, Bonas, France, June 1991.

[KAGR82] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, and D.V.B. Rao. Wavefront array processor: language, architecture, and applications. *IEEE Trans. on Computers*, C-31(11):1054–1066, Nov 1982.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.

[Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46 multiprocessors, parallel processing, systolic arrays, VLSI,, January 1982.

[Lev91] H. Leverge. *Reduction operators in* ALPHA. Research Report, IRISA, adressirisa, November 1991. to appear.

[LMQ90] H. Leverge, C. Mauras, and P. Quinton. A language-oriented approach to the design of systolic chips. In *International Workshop on Algorithms and Parallel VLSI Architectures*, Pont-à-Mousson, June 1990. To appear in the Journal of VLSI Signal Processing, 1991.

[LMQ91] H. Leverge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[LS81] C.E. Leiserson and J.B. Saxe. Optimizing synchronous systems. In *22th Annual Symp. on Foundations of Computer Science*, pages 23–36, IEEE Press, Oct 1981.

[Mol82] D.I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Transactions on Computers*, C-31(11), November 1982.

[QD89] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *The Journal of VLSI Signal Processing*, 1:95–113, 1989. Quinton89c.

[Rao85] S.K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Standford University, U.S.A., October 1985.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming. Wiley-Interscience series in Discrete Mathematics*, John Wiley and Sons, 1986.

This article was processed using the LaTeX macro package with LMAMULT style

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 609 INTEGRATION D'UN CORRECTEUR ORTHOGRAPHIQUE DANS L'EDITEUR STRUCTURE GRIF
Patrice FRISON, Eric PICHERAL, Hélène RICHY
Octobre 1991, 22 pages.

PI 610 SYNCHRONIZATION AND CONCURRENCY MEASURES FOR DISTRIBUTED COMPUTATIONS
Michel RAYNAL
Octobre 1991, 20 pages.

PI 611 MALI v06 - TUTORIAL AND REFERENCE MANUAL
Olivier RIDOUX
Octobre 1991, 86 pages.

PI 612 SENSITIVITY COMPUTATION IN NETWORK RELIABILITY ANALYSIS
Gerardo RUBINO
Octobre 1991, 38 pages.

PI 613 OPAC : A FLOATING-POINT COPROCESSOR DEDICATED TO COMPUTE-BOUND KERNELS
André SEZNEC, Karl COURTEL
Octobre 1991, 28 pages.

PI 614 CONTROLLING AND SEQUENCING AN HEAVILY PIPELINED FLOATING-POINT OPERATOR
André SEZNEC, Karl COURTEL
Octobre 1991, 28 pages.

PI 615 ON FAULT-TOLERANT SYMBOLIC COMPUTATIONS
Bernard DELYON, Oded MALER
Novembre 1991, 18 pages.

PI 616 USING COHERENCE TO ACCELERATE RADIOSITY
Pierre TELLIER, Eric MAISEL, Kadi BOUATOUCH, Eric LANGUENOU
Novembre 1991, 16 pages.

PI 617 INTERVAL APPROXIMATIONS OF MESSAGE CAUSALITY IN DISTRIBUTED EXECUTION
Claire DIEHL, Claude JARD
Novembre 1991, 44 pages.

PI 618 RETOUR SUR LE RESEAU SYSTOLIQUE DU PALINDROME
Hervé LE VERGE, Patrice QUINTON
Novembre 1991, 14 pages.

ISSN 0249 - 6399