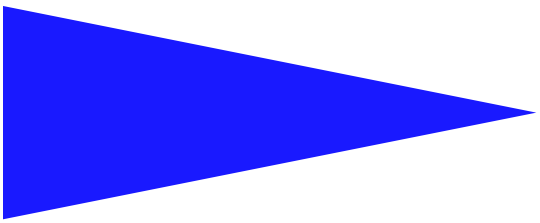
P U B L I C A T I O N
I N T E R N E
N° 829



REGULAR ARRAY SYNTHESIS USING ALPHA

DORAN K. WILDE AND OUMAROU SIÉ

I R I S A
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

Regular Array Synthesis using ALPHA

Doran K. Wilde* and Oumarou Sié**

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Publication interne n ° 829 — May 1994 — 13 pages

Abstract: We report our current research in a computer assisted methodology for synthesizing regular array processors using the ALPHA language and design environment. The design process starts from an algorithmic level description of the function and finishes with a netlist of an array processor which performs the specified function. To illustrate the proposed approach, we present the design of an array processor to do polynomial division.
Key-words: recurrence equations, systolic arrays, alignment, scheduling, pipelining, allocation, uniform dependencies, computer aided design, automatic synthesis, hardward design language, EDIF

(Résumé : tsvp)

*email: wilde@irisa.frThis work was partially supported by the Esprit Basic Research Action NANA 2, Number 6632 and by NSF Grant No. MIP-910852.
**email: sie@irisa.fr

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

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

Synthèse de réseaux réguliers avec ALPHA

Résumé : Ce papier présente une méthodologie de synthèse assistée par ordinateur dans l'environnement ALPHA. Cet environnement est dédié à la synthèse de réseaux réguliers. Partant d'une description comportementale de type algorithmique d'une fonction, le processus de conception génère la netlist du réseau de processeurs correpondant. L'approche est illustrée par la présentation de la synthèse d'un réseau de processeurs réalisant la division polynomiale.

Mots-clé : équations récurrentes, réseaux systoliques, alignement, ordonnancement, pipeline, allocation, dépendances uniformes, conception assistée par ordinateur, synthèse automatique, langage de description de matériel, EDIF

1 Introduction

We describe our experiences in using ALPHA as a tool to develop regular array processor circuits from a high level description of an algorithm. ALPHA is a functional language based on the formalism of systems of affine recurrence equations. Algorithms may be represented at a very high level in ALPHA, close to how one might specify them mathematically and are in an equational and fully parallel form. The input specification can be transformed into a form called ALPHA0 [2] (a subset ALPHA) which is suitable for generating a netlist for the regular array processor. Then finally, an EDIF netlist is generated for the circuit. ALPHA0 is obtained by performing a series of program transformations which can be independently proved, and thus the derived ALPHA0 program is correct by construction (assuming that the specification was correct). The ALPHA language is very restrictive in the class of algorithms that it can represent, but is useful for programming mathematical types of algorithms such as the kinds currently being proposed for signal and video processing. The current ALPHA environment is built on top of MATHEMATICA [13], a commercially available symbolic algebra system which supports imperative, functional, and rule-based programming paradigms built on top of a symbolic calculator. This environment provides the means to both create and run ALPHA transformation and analysis tools.

To demonstrate the proposed design process, we wanted to chose an example problem which was not too complex, but which at the same time tested a broad range of ALPHA capabilities in order to find the strengths and weaknesses of the ALPHA system. This project has thus helped us in development of the ALPHA environment. The example that we chose is polynomial division. This problem is interesting because it • involves a system of mutually recursive equations, • is specified in terms of two parameters which specify the size of the two inputs, • has more than one output, • has a systolic solution that requires more than one type of processor, and • is not always 100% efficient (depending on the chosen projection).

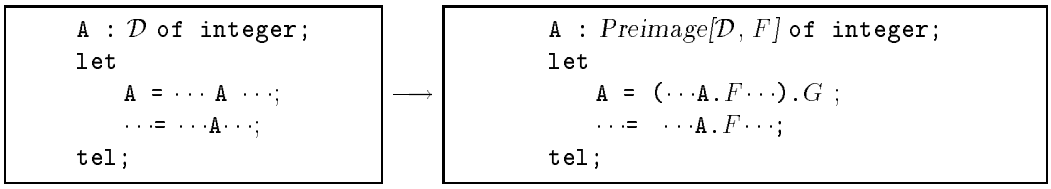
2 Introduction to ALPHA

ALPHA was originally designed in the context of systolic array synthesis research [4, 7] done at IRISA in France. A fundamental feature of ALPHA, which sets it apart from other functional languages, is that all variables in ALPHA are based on polyhedral index domains [12]. Variables are strongly typed and denote a mapping from an *index domain* (the set of all integral points within a specified union of polyhedra) to values in the *value domain* (integers, reals, or booleans). A polyhedron, $\{z \in Z^n | Az \geq b\}$, is the intersection of a finite set of halfplanes, each of which is defined by a linear inequality. The syntax of a polyhedron in ALPHA is (for example): `{ i,j | 0<i<N; 0<=j<i }`, and the syntax of the declaration of an integer variable A based on that index domain is: `A : { i,j | 0<i<N; 0<=j<i } of integer;`. A system may have input, output, and local variables. The equations that define the variables follow the declarations and are delimited by the pair of keywords `let` and `tel`. The language is equational, and each equation $\texttt{variable}_{LHS}\texttt{= expression}_{RHS}$ names a variable on the LHS and has an expression on the RHS. A *dependency* is an affine function of indices which maps the index domain of the LHS variable to an index domain on the RHS. Syntactically, a dependency is written as: `(index, index, ...` $\rightarrow$ `index-expr, index-expr, ...)`, where each `index-expr` is an affine expression of the indices to the left of the arrow. Examples of affine functions are `(i->i-2)` and `(Z,i,k->Z-1,2Z-k-1,k-1)`. In ALPHA , affine function operators can be written explicitly: `X = B.(i->i) - b.(i->i,i-1);` or equivalently, they can be written implicitly: `X[i] = B[i] - b[i,i-1];`

The syntax for an ALPHA expression is presented below. (Vertical bar is an alternate, square bracket is an optional construct and the asterisk is a Kleene star)

`data-variable	constant`	
`[expression] op expression`	`op` is a binary or unary operator
`expr . dep`	an expression composed with an affine dependency function
`domain : expr`	an expression restricted to a particular (sub)domain
`case expr* esac`	a union of expressions defined over disjoint subdomains

The denotational semantics of ALPHA [7] specify that every ALPHA expression denotes a function from indices to values. These semantics are fully compositional, and form the basis of a transformational system. The *change of basis* is the most common transformation and is similar to the reindexing of loop variables done in vectorizing and parallelizing compilers. A change of basis of variable `A` using the affine transformation function F and left inverse transformation function G, such that $(F \circ G)y = y,\ y \in \mathcal{D}_y$, is done by rewriting an ALPHA program as follows:



There is also an ALPHA transformation that "normalizes" any ALPHA program into a syntactic form where all case constructs occur at the outermost level in the RHS of an equation, and all compositions of `dep`'s are converted into a single `dep` function. Many other transformations are used and will be described in context of the example.

3 The polynomial division problem

Given two polynomials

$$\begin{aligned} A(x) &= a_N x^N + a_{N-1}x^{N-1} + \cdots + a_2 x^2 + a_1 x + a_0 \\ B(x) &= b_M x^M + b_{M-1}x^{M-1} + \cdots + b_2 x^2 + b_1 x + b_0 \end{aligned}$$

where N and M are parameters such that $N \geq M \geq 1$ and where $a_N \neq 0$ and $b_M \neq 0$. A computation produces the quotient Q and remainder R of the division of polynomial A by polynomial B, such that

$$\begin{aligned} Q(x) &= q_{N-M}x^{N-M} + q_{N-M-1}x^{N-M-1} + \cdots + q_2 x^2 + q_1 x + q_0 \\ R(x) &= r_{M-1}x^{M-1} + r_{M-2}x^{M-2} + \cdots + r_2 x^2 + r_1 x + r_0 \\ R(x) &= A(x) - B(x)Q(x) \end{aligned}$$

The problem is to produce a systolic array to do the above computation which uses the sequences $A_{N\cdots 0}$ and $B_{M\cdots 0}$ as inputs and produces $Q_{(N-M)\cdots 0}$ and $R_{(M-1)\cdots 0}$ as outputs.

Example: Consider the example of

$$\frac{A = 3x^7 + 4x^6 + 5x^5 + 6x^4 + 5x^3 + 4x^2 + 3x + 2}{B = x^3 + 2x^2 + 3x + 2} = Q = 3x^4 - 2x^3 + 6x - 3 \quad R = -8x^2 + 8$$

The computation can be done using standard hand methods as shown below.

$$
\begin{array}{llll|l|llll}
 & & & & & & & 3x^4 & -2x^3 & +0x^2 & +6x & -3 \\
x^3 & +2x^2 & +3x & +2 &) & 3x^7 & +4x^6 & +5x^5 & +6x^4 & +5x^3 & +4x^2 & +3x & +2 \\
 & & & & & 3x^7 & +6x^6 & +9x^5 & +6x^4 & & & & \\
\cline{6-9}
 & & & & & & -2x^6 & -4x^5 & +0x^4 & +5x^3 & & & \\
 & & & & & & -2x^6 & -4x^5 & -6x^4 & -4x^3 & & & \\
\cline{7-10}
 & & & & & & & 0x^5 & +6x^4 & +9x^3 & +4x^2 & & \\
 & & & & & & & 0x^5 & +0x^4 & +0x^3 & +0x^2 & & \\
\cline{8-11}
 & & & & & & & & +6x^4 & +9x^3 & +4x^2 & +3x & \\
 & & & & & & & & +6x^4 & +12x^3 & +18x^2 & +12x & \\
\cline{9-12}
 & & & & & & & & & -3x^3 & -14x^2 & -9x & +2 \\
 & & & & & & & & & -3x^3 & -6x^2 & -9x & -6 \\
\cline{10-12}
 & & & & & & & & & & -8x^2 & +0x & +8
\end{array}
$$

Description in ALPHA: The recurrence equations for polynomial division were derived following the above hand method and encoded into ALPHA. The $R = A - BQ$ pattern is clearly visible in the recurrence equations for *rr* below. The program was executed to validate the specification. This resulting ALPHA program is shown below.

```
-- Polynomial Division, parameterized

system dp      (a : { k,N,M | 0<=k<=N  ; N>=M>=1 } of integer;
                b : { k,N,M | 0<=k<=M  ; N>=M>=1 } of integer)
       returns (q : { j,N,M | 0<=j<=N-M; N>=M>=1 } of integer;
                r : { k,N,M | 0<=k<=M-1; N>=M>=1 } of integer);
var
  rr : { k,j,N,M | 0<=k<=M-1; 0<=j<=N-M; N>=M>=1 } of integer;
let
  q = case
      { j,N,M | j=N-M }    : a.(j,N,M->N,N,M)/b.(j,N,M->M,N,M);
      { j,N,M | j<=N-M-1 } : rr.(j,N,M->M-1,-j+N-M-1,N,M)/b.(j,N,M->M,N,M);
      esac;
  rr = case
       { k,j,N,M | j=0 } : a.(k,j,N,M->k+N-M,N,M)
                         - q.(k,j,N,M->-j+N-M,N,M) * b.(k,j,N,M->k,N,M);
       { k,j,N,M | j>=1; k=0 } : a.(k,j,N,M->-j+N-M,N,M)
                         - q.(k,j,N,M->-j+N-M,N,M) * b.(k,j,N,M->k,N,M);
       { k,j,N,M | j>=1; k>=1 } : rr.(k,j,N,M->k-1,j-1,N,M)
                         - q.(k,j,N,M->-j+N-M,N,M) * b.(k,j,N,M->k,N,M);
       esac;
  r = rr.(k,N,M->k,N-M,N,M);
tel;
```

As the recurrence equations were being derived, it was useful to create a graphical representation of the equations, in the form of a dependency graph. The dependency graph of figure 1 was helpful to us in the development of the recurrence equations.

This dependency graph is actually several transformations beyond the recurrence equations as specified above since communication has already been localized.

4 Design methodology

The following is a presentation of a design methodology for transforming an algorithmic specification into a netlist of systolic array processors. Each step may in turn require a number of low–level ALPHA transformations.

1. *Loading* Parse ALPHA into an abstract syntax tree (AST). The AST is the internal representation used by the system to do all transformations and operations.

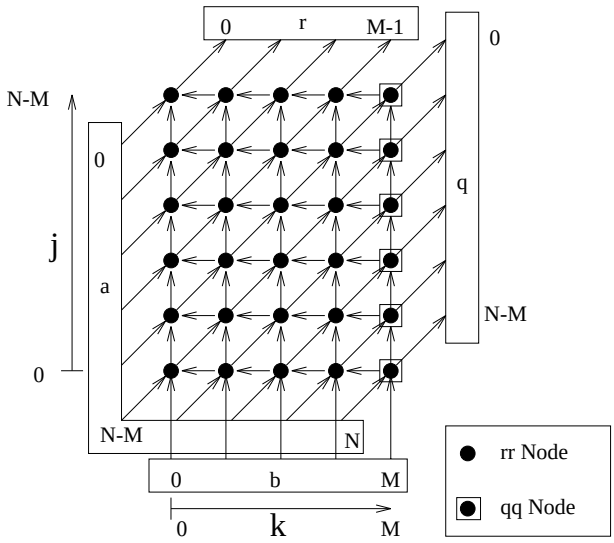


Figure 1: Dependency Graph for Polynomial Division Program

2. *Alignment of variables.* Place all local variables on a common dimensional grid in a manner which minimizes communication. This is also called placement, since variables are placed relative to each other. This gives a common reference space for all variables and facilitates later transformations and analysis.

3. *Localization of communication.* Pipeline broadcasted variables to where they are used in order to localize all communication between nodes on the dependency graph. Local communication is from a node (in the dependency graph) to neighboring nodes only. A localized program is called *uniform.*

4. *Fix parameters.* Parameters are fixed to specific values. Ideally, this should be done later in the design process. However, for the time being, the tool we have to find the schedule requires fixed sized, finite arrays.

5. *Choose a projection vector.* A projection vector (and thus an allocation function) is chosen. This is the direction that nodes in the dependency graph will be projected onto physical processors. All nodes falling on the same projection line will be mapped to the same physical processor.

6. *Pipeline inputs and outputs.* Inputs and outputs are pipelined so that all I/O to the array enters or exits at the ends of the array [8, 5]. In the case of a linear array, all I/O is to/from the processors at the two ends of the array.

7. *Find a timing function.* A function (change of basis) which maps one of the dimensions of the virtual domain to "time", is called a timing function. The program is called *causal* if for every dependency (def–use pair of variables) a value is used only after the time when it is computed.

8. *Apply timing and allocation functions.* Perform the change of bases to reflect the timing and allocation functions on all local variables. Indices are renamed "t" and "p" to represent time and processor (space).

9. *Separation of time and space.* In the definitions of local variables, separate case statements whose alternatives depend on both time and space into doubly nested case statements where the outer most nest (the *space case*) depends only on space and the inner most nest (the *time case*) separates the temporal behavior for each different range of processors. A single processor may do different computations at different times. After the separation of time and space, an ALPHA program shows how a variable is computed in each (range of) processor(s), as a function of time.

10. *Control Variable Generation.* The inner time cases generated above specify that, for some variables, different computations may need to be done at different times. In this step, a time dependent control variable, or set of control variables, is created to govern which computation to select. To chose among n different possibilities, requires $\lceil \log(n) \rceil$ control variables. A (set of) control variable(s) is created for each variable having a time case in the RHS of its definition. Later on, during netlist generation, these control variables will become control signals to multiplexors components [2, 9].

11. *Normalize to* ALPHA0*.* Separate statements which perform more than one level of arithmetic into multiple statements, introducing new local variables as needed. Control variables may be pipelined and simplifications and optimizations performed where possible. For example, two different, but compatible, control variables may be merged into a single variable.

12. *Generate netlist.* The array netlist generator creates a hierarchical netlist description in the EDIF format [11] from the array specification contained in an ALPHA0 file. The resulting netlist has the following parts:

    - *Basic cell library.* A technology dependent library of *basic components* such as registers, multiplexors, and basic arithmetic functional blocks.
    - *Structural descriptions of the processors.* A regular array is composed of one or more processor cell types. For each processor type in the array, the netlist contains a structural definition containing the *interface declaration*, the list of *instances* of basic components, and the *interconnection* between the components.
    - *Structural description of the array.* This description is the top level in the structural hierarchy and consists of *instances* of *processors* and the *interconnection* between the processors.

5 Transformation of the polynomial division problem

In this section, we present an annotated script file which gives each major low–level transformation needed to implement the steps described in section 4. This script file may be read and executed by the MATHEMATICA–ALPHA environment since the commands are in fact functions written in MATHEMATICA . Each command analyzes and/or transforms the current AST little by little toward a desired target format.

`load["dp.alpha"];`
: Loads the source specification file "dp.alpha". Parses the file and creates the AST inside MATHEMATICA with name `$result`.

`analyze[ ]`
: Performs static analysis on the program using the denotational semantics of ALPHA. Checks for domains which do not match their declarations and expressions which are defined over empty domains [10].

`addlocal["qq = q"];`
: Creates a local variable `qq` to represent `q` wherever it is used. This is done in preparation for aligning the computations associated with `q` with the other variables.

`addDimension["qq.(j,N,M->M,j,N,M)","k"];`

Places the one dimensional variable `qq` on a two dimensional `(k,j)`–grid by adding a dimension to it and placing it on the line `k=M`.

`changeOfBasis["qq.(k,j,N,M->k,N-M-j,N,M)"];`

Align the variable `qq` with the variable `rr` to make all communication between `qq` and `rr` local. See figure 1— the `qq` nodes are boxed and the `rr` nodes are not boxed.

`pipeline["rr","b.(k,j,N,M->k,N,M)","B1.(k,j,N,M->k,j+1,N,M)"];`

`pipeline["qq","b.(k,j,N,M->M,N,M)","B2.(k,j,N,M->k,j+1,N,M)"];`

`pipeline["rr","qq.(k,j,N,M->M,j,N,M)","Q.(k,j,N,M->k-1,j,N,M)"];`

The first command pipelines the expression `b.(k,j,N,M->k,N,M)` used in variable `rr`, creating a new local variable `B1` to do the job. `(k,j,N,M->k,j+1,N,M)` gives the direction to propagate the values inside `B1`. The other two pipeline commands perform similar tasks.

`merge["B1", "B2", "B"];`

`merge["Q","qq","Q"];`

Do a little simplification by combining compatible variables `B1` and `B2` and calling the combination `B`. Merge `Q` and `q` into `Q` as well. Having been pipelined and simplified, the program is now uniform.

`project["k,j,N,M| N=9;M=4 ", "B .( k,j,N,M->k,j )"];` (likewise for `Q` and `rr`)

`project["i,N,M| N=9;M=4 ", "a.( i,N,M->i )"];` (likewise for `b`, `r`, and `q`)

Fix the parameters `N` and `M` to the values 9 and 4, respectively. This is done for each of the 3 local variables, the 2 input variables, and the 2 outputs variables.

At this point the program is uniform and parameters are set to a specific problem size. The resulting ALPHA program is:

```
system dp (a : {i | i>=0;-i+9>=0} of integer;
           b : {k | k>=0;-k+4>=0} of integer)
       returns (q : {j | -j+5>=0;j>=0} of integer;
                r : {k | k>=0;-k+3>=0} of integer);
var
  Q : {k,j | -k+4>=0;j>=0;k>=0;-j+5>=0} of integer;
  B : {k,j | k>=0;-j+5>=0;-k+4>=0;j>=0} of integer;
  rr : {k,j | k>=0;j>=0;-k+3>=0;-j+5>=0} of integer;
let
  Q = case
        {k,j | -k+3>=0}: Q.(k,j->k+1,j);
        {k,j | k-4=0;j=0}: a.(k,j->9) / B;
        {k,j | k-4=0;j-1>=0}: rr.(k,j->3,j-1) / B;
      esac;
  B = case
        {k,j | j=0}: b.(k,j->k);
        {k,j | j-1>=0}: B.(k,j->k,j-1);
      esac;
  q = Q.(j->4,-j+5);
  rr =
      case
        {k,j | j=0}: a.(k,j->k+5) - Q * B;
        {k,j | k=0;j-1>=0}: a.(k,j->-j+5) - Q * B;
        {k,j | j-1>=0;k-1>=0}: rr.(k,j->k-1,j-1) - Q * B;
      esac;
  r = rr.(k->k,5);
tel;
```

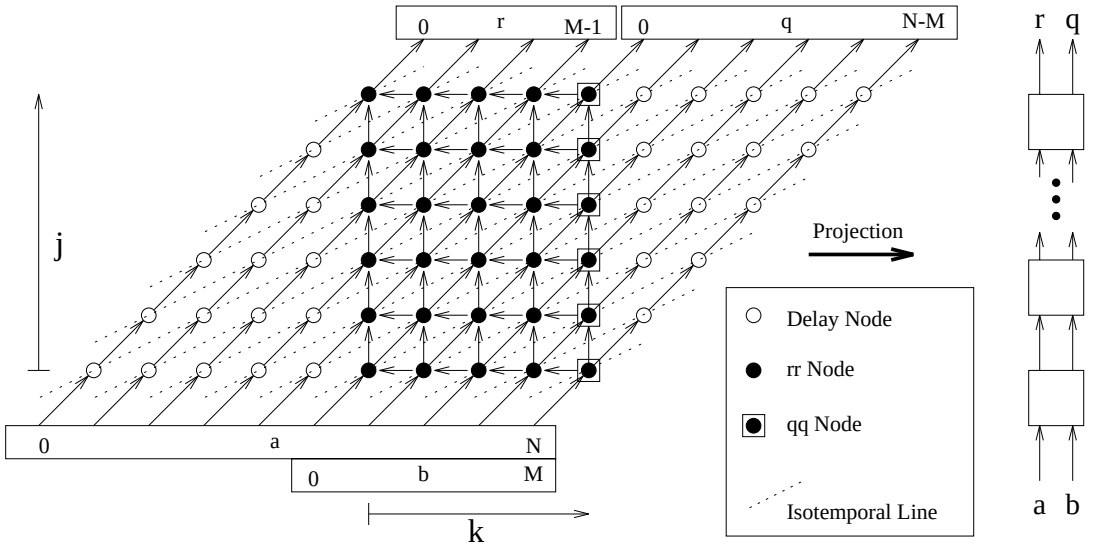


Figure 2: Dependency Graph for Polynomial Division Program

Now, we are ready to set the projection direction for the program. In this case, there are two good choices for projections: one along the X–axis (see figure 2) and a second along the Y–axis (see figure 3). Thus, the design sequence splits here into two different trajectories resulting in two entirely different designs, depending on which projection is chosen. We will follow the first choice—project along the X–axis.

`$space = {-1,0};`
: This is the (processor space) projection vector along the X–axis.

`pipeIO["a.(k,j->-j+5)","A.(k,j->k+1,j+1)","k,j | j>=0"];`
: The input variable `a` which enters on the left hand side of the dependency graph is pipelined such that `a` will only enter the processor array from the bottom (see figure 2).

`pipeIO["q","Q","Q2.(k,j->k+1,j+1)","k,j | j<=5"];`
: The output variable `q` derived from `Q` will be pipelined so that `q` exits from the top of the processor array (see figure 2).

`$time = minPeriodLinearOffsetSched[$space, "B", "Q", "rr","Q2","A"];`
: This is an analysis step which sets up and solves a linear programming problem to find an optimal schedule for uniformly dependent algorithms [1]. The result is a timing function. If the program is non–causal, or not uniform, this step will fail to find a schedule.

`ApplyChangeOfBasis[$time, $space, "B", "Q", "rr", "Q2", "A"]`
: The timing function and projection vector are used to compute change of basis functions for each of the specified variables. These change of basis transformations are then performed automatically.

Here, the program has been scheduled and projected. The indices representing time and processor space have been renamed "t" and "p". The program at this point is as follows:

```
system dp (a : {i|i>=0;-i+9>=0} of integer;
           b : {k|k>=0;-k+4>=0} of integer)
      returns (q : {j|-j+5>=0;j>=0} of integer;
               r : {k|k>=0;-k+3>=0} of integer);
```

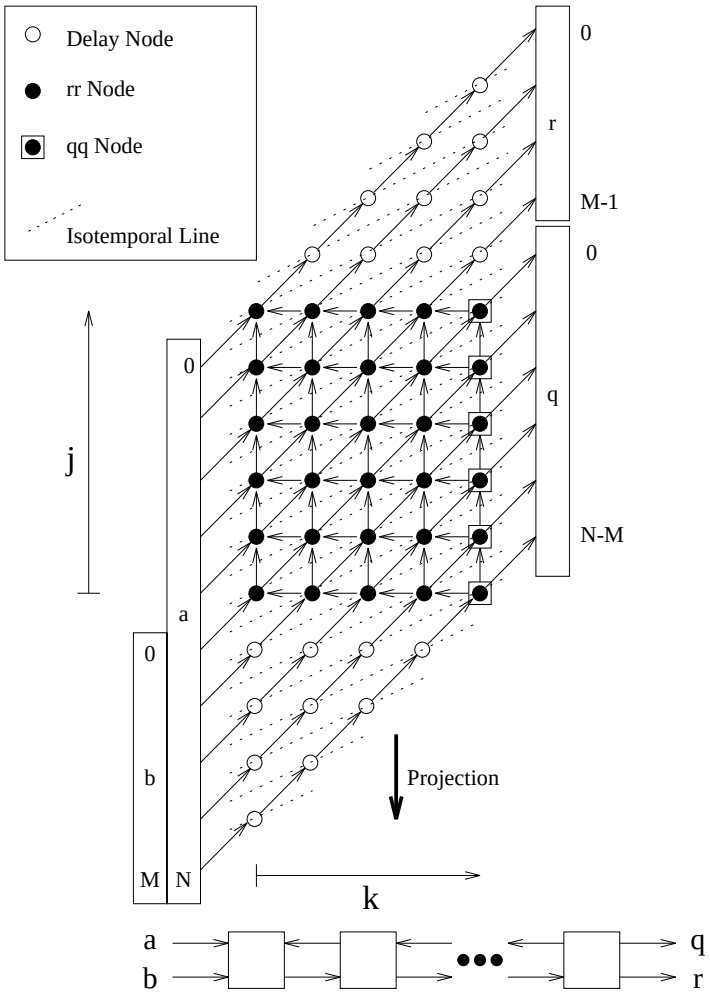


Figure 3: Dependency Graph for Polynomial Division Program

```
var
  Q2 : {t,p|-t+2p>=0;t-p>=0;-p+5>=0} of integer;
  A : {t,p|t-2p-4>=0;-t+p+9>=0;t-p-5>=0;p>=0} of integer;
  Q : {t,p|t-2p>=0;p>=0;-t+2p+4>=0;-p+5>=0} of integer;
  B : {t,p|-t+2p+4>=0;-p+5>=0;t-2p>=0;p>=0} of integer;
  rr : {t,p|-t+2p+4>=0;p>=0;t-2p-1>=0;-p+5>=0} of integer;
let
  Q2 = case
         {t,p|t-2p=0} : Q;
         {t,p|-t+2p-1>=0} : Q2.(t,p->t-1,p-1);
       esac;
  A = case
        {t,p|p=0} : a.(t,p->-t+9);
        {t,p|p-1>=0} : A.(t,p->t-1,p-1);
      esac;
  Q = case
        {t,p|t-2p-1>=0} : Q.(t,p->t-1,p);
```

```
        {t,p|t=0;p=0} : a.(t,p->9)/B;
        {t,p|t-2p=0;p-1>=0} : rr.(t,p->2p-1,p-1)/B;
      esac;
  B = case
        {t,p|p=0} : b.(t,p->-t+4);
        {t,p|p-1>=0} : B.(t,p->t-2,p-1);
      esac;
  q = Q2.(j->-j+10,5);
  rr = case
         {t,p|p=0} : a.(t,p->-t+9) - Q * B;
         {t,p|t-2p-4=0;p-1>=0} : A - Q * B;
         {t,p|p-1>=0;-t+2p+3>=0} : rr.(t,p->t-1,p-1) - Q * B;
       esac;
  r = rr.(k->-k+14,5);
tel;
```

We continue to evolve this program towards ALPHA0 which easily maps to circuit components such as registers, multiplexors, ALUs, and combinational logic.

`$timeIndexPos = {1}`

`$spaceIndexPos = {2}`

This tells the ALPHA environment where to find the time and space indices, respectively, within the list of indices.

`spaceTimeCase["rr", $timeIndexPos, $spaceIndexPos];` (likewise for `A`, `B`, `Q`, and `Q2`)

These transformations separate the cases dependent on both time and space indices into nested cases, where the outer cases depend only on processor space, and the inner cases specify a behavior for each range of processors. For example, the first command separates the space and time dependencies of the variable `rr` as follows:

```
rr = case
       {t,p | p=0}: a.(t,p->-t+9) - Q * B;
       {t,p | p-1>=0}:
          case
            {t,p | t-2p-4=0}: A - Q * B;
            {t,p | -t+2p+3>=0}: rr.(t,p->t-1,p-1) - Q * B;
          esac;
     esac;
```

Control signal generation

For the case statements whose alternatives depend on time, a control signal is generated which selects among the alternatives as a function of time. For example, for the variable `rr` (see above), a control variable `loadrr` is created and used in the modified definition of `rr` as follows:

```
    loadrr = case
               {t,p|p=0} :
                 case
                   {t,p|t-4=0} : True.(t,p->);
                   {t,p|-t+3>=0} : False.(t,p->);
                 esac;
               {t,p|p-1>=0} : loadrr.(t,p->t-2,p-1);
             esac;
    rr = case
           {t,p|p=0}    : a.(t,p->-t+9) - Q * B;
           {t,p|p-1>=0} : if (loadrr) then A - Q * B
                                      else rr.(t,p->t-1,p-1) - Q * B;
         esac;
```

Simplification and Optimization

In this step, the ALPHA program is finally transformed into ALPHA0. Control variables are combined when possible and intermediate variables are introduced where necessary in order to only have one single arithmetic operation per simple expression. Continuing the example above, new variables `QBout` and `rr1` have been added in the code below to bring it into ALPHA0 form.

```
QBout = Q * B;
rr1 = if (loadrr) then A else rr.(t,p->t-1,p-1);
rr = case
       {t,p|p=0}    : a.(t,p->-t+9) - QBout;
       {t,p|p-1>=0} : rr1 - QBout;
     esac;
```

The above is a part of the final ALPHA0 program.

`save["dp-final.alpha0"]`

The final ALPHA0 program is written out (the AST is translated back into ALPHA notation) to the file "dp-final.alpha0".

`alpha2edif <dp-final.alpha0 >dp.edif`

From the operating system, the final program is run through a netlist compiler which produces an EDIF netlist for the processor array. The translator makes use of a cell library which allows us to migrate between technologies and facilitates the interface with other tools such as a logic simulator or logic synthesizer. Presently, only one–bit processors are supported. For one–bit functions, the standard cell approach is better suited than the datapath approach [6]. In appendix A, we give the resulting schematics for the polynomial division problem. Figure 4 is a schematic of the linear array of processors composed of two processor types. Figures 5 and 6 are schematics of the first ($p = 0$) and subsequent ($1 \leq p \leq 5$) processors, respectively. Since these are single–bit processors, multiplication is done with an AND gate, subtraction is done with an XOR gate, and division is always by 1.

The EDIF netlist which is generated can be read by the MADMACS regular array layout generation system [3]. In this system, the physical topologies of the processor and array are specified and generated.

6 Conclusions

We have presented our current research in doing regular array synthesis using a program transformational approach in the ALPHA environment. This environment is a toolchest of mechanical transformation and analysis tools which can be applied to a program to move it along a trajectory in its design space from the original specification to a desired implementation. Transformations are designed and written to guarantee the semantic equivalence of a program, before and after being transformed. The choice of which transformation to apply at each step is made by the designer. We have demonstrated this approach using polynomial division, an example of medium complexity.

Ongoing research is being conducted to make the transformation process more and more automated, getting input from the designer when needed. This will make the ALPHA environment much closer to a compiler which automatically translates an algorithm into a netlist. There is also ongoing research in translating netlists of array processor architectures into geometry using the MADMACS tool. The ALPHA language is continuing to evolve and we are currently working to extend the ALPHA language to increase both its representational power and readability, all while retaining the strict denotational semantics needed to analyze and transform an ALPHA program. Optimization of an ALPHA program to minimize (or reduce) the number of registers, multiplexors, and ALUs needed is still an open problem.

7 Acknowledgments

Zbi Chamski programmed much of the current ALPHA system within the MATHEMATICA environment. Hervé Le Verge was helpful in giving insight and mathematical rigor for many of the transformations. Fernando Rosa Do Nascimento helped write the specification of polynomial division and validate it with simulation. Finally, we thank Patrice Quinton, Sanjay Rajopadhye, and Eric Gautrin for their valuable discussions and encouragement.

References

[1] A. Darte, L. Khachiyan, and Y. Robert. Linear scheduling is close to optimality. *International Conference on Application Specific Array Processors*, 37–46, 1992.

[2] Catherine Dezan. *Génération automatique de circuits avec ALPHA du CENTAUR.* PhD thesis, Université de Rennes 1, Rennes, France, Feb 1993.

[3] E. Gautrin and L. Perraudeau. *MADMACS: an environment for the layout of regular arrays*, pages 345–358. Elsevier Science Publishers B.V. (North-Holland), 1993.

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

[5] Hervé Le Verge. *Un environnement de tranformations de programmes pour la synthèse d'architectures régulières.* PhD thesis, Université de Rennes 1, Rennes, France, Oct 1992.

[6] R. Leveugle and C. Safinia. *Generation of optimized datapaths: bit-slice versus standard cells*, pages 153–166. Elsevier Science Publishers B.V. (North-Holland), 1993.

[7] Christophe Mauras. *Alpha, un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes 1, Rennes, France, Dec 1989.

[8] S. V. Rajopadhye. I/O behavior of systolic arrays. In *IEEE Workshop on VLSI Signal Processing*, pages 423–434, IEEE Press, November 1988.

[9] S. V. Rajopadhye. Synthesising systolic arrays with control signals from recurrence equations. *Distributed Computing*, 88–105, May 1989.

[10] Fernando Rosa Do Nascimento. *Méthodologie de Conception d'Architectures Spécialisées – une Etude de Cas.* PhD thesis, Université de Rennes 1, Rennes, France, Oct 1993.

[11] P. Stanford and P. Mancuso. *Electronic Design Interchange Format-Version 2 0 0, Recommended Standard EIA–548.* Electronic Industries Association, EDIF Steering Committee, Washington, D.C., 1989.

[12] D. Wilde. *A library for Doing Polyhedral Operations.* Technical Report Internal Publication 785, IRISA, Rennes, France, Dec 1993.

[13] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer, Second Edition.* Addison–Wesley Publishing Company, Inc., 1991.

A Schematics of Polynomial Division Circuit

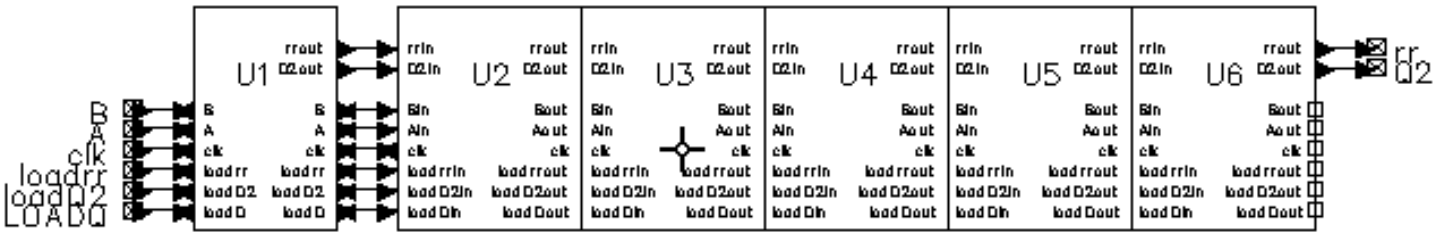


Figure 4: Schematic for processor array

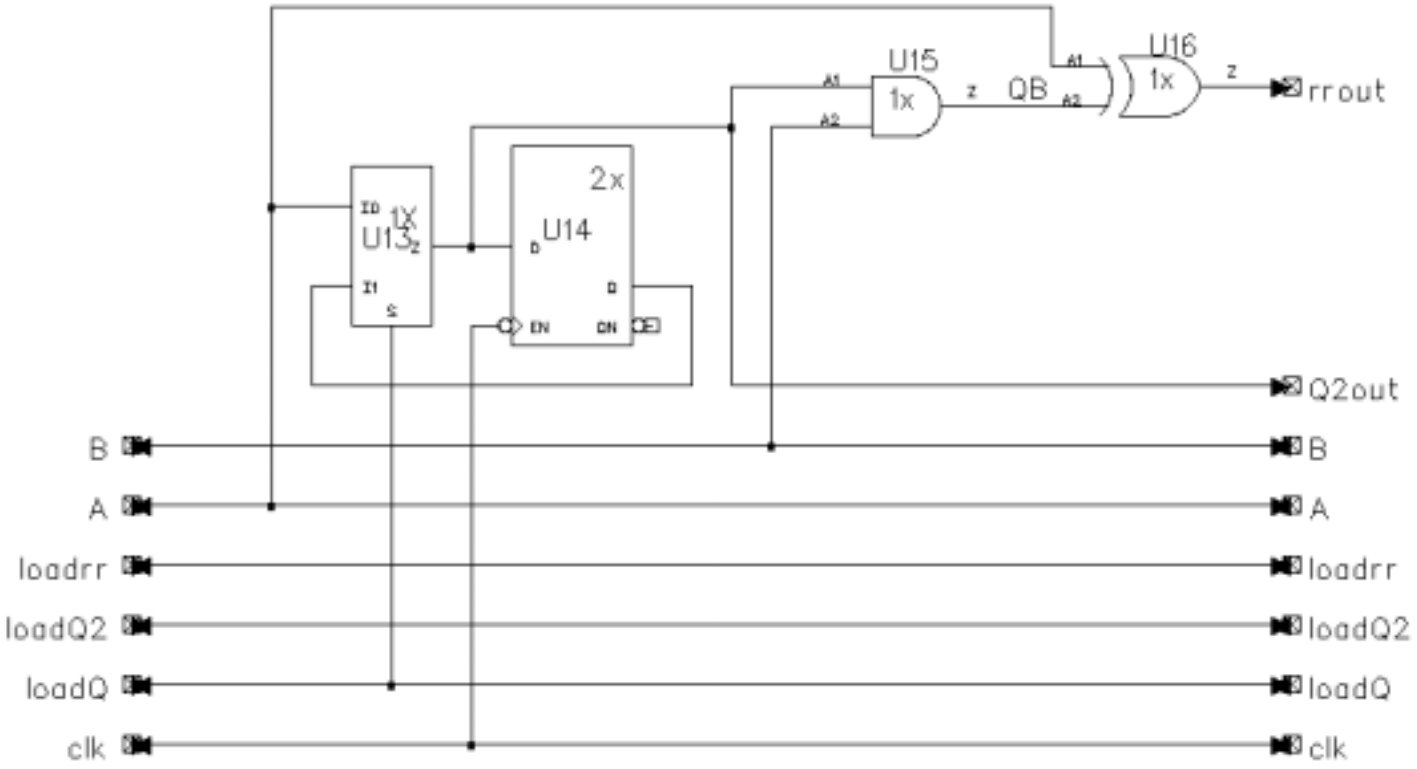


Figure 5: Schematic for Processor 2

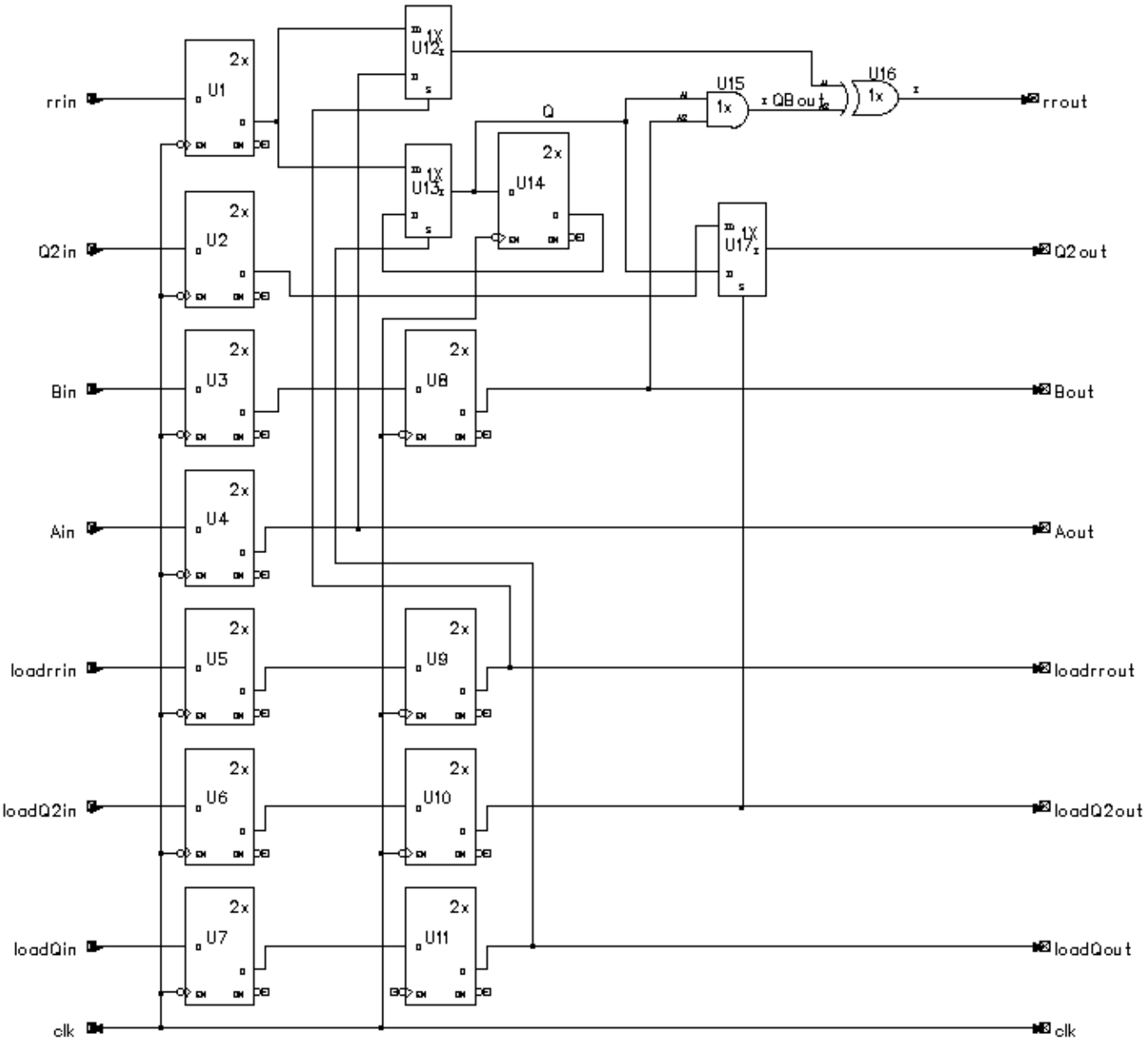


Figure 6: Schematic for Processor 1