First Steps in ALPHA[1]

The Alpha team[2]

March 3, 2007

[1]Version of March 3, 2007
[2]Contact: alpha@irisa.fr. See also http://www.irisa.fr/api/ALPHA/welcome.html.

Contents

List of Figures

Foreword

MmAlpha is a free software, available under the Gnu Public License. It can be downloaded from the site:
`www.irisa.fr/api/...`
If you have any problem in understanding this document or trying the MmAlpha software, please send an e-mail to the following address: `alpha@irisa.fr`.

Although we cannot guarantee a strong support, we shall try our best to help solving your problem.

Many people contributed to MmAlpha.

- Pierrick Gachet designed an early version of Alpha.
- Christophe Mauras designed the first real version of Alpha, and implemented the ancestor of MmAlpha, called Alpha du Centaur.
- Herv Le Verge continued the work of Christophe Mauras, and worked on reduction operators. He also implemented the first version of the polyhedral library.
- Doran Wilde designed the first version of the parser, of the pretty-printers, of `WriteC`, and of many early packages of MmAlpha. He developped a new, more efficient version of the polyhedral library.
- Zbignew Chamsky contributed to the design of the first MmAlpha packages.
- Florent Dupont de Dinechin implemented the structuration of Alpha, and deeply improved the MmAlpha environment.
- Tanguy Risset is the conductor of the MmAlpha entreprise. He designed the scheduler, and many other packages.
- Patrice Quinton was the inspirer of Alpha, and implemented several minor packages of Alpha. He also designed numerous notebook.
- Sanjay Rajopadhye insuffled many ideas and designed the reduction expansion package.
- Patricia Le Moenner implemented the Vhdl generator.
- Fabien Quiller is currently implementing a new C generator.

We also thank the EEC, who sponsored the development of Alpha through several grants. The late C^3 research cooperation network also helped initial research on Alpha.

Chapter 1

Introduction

In this document, we introduce the basics of Alpha and we explain how to use MmAlpha.

1.1 What is Alpha?

Alpha is a language for the specification and the derivation of systolic algorithms and architectures, and more generally, for the specification and design of parallel algorithms and architectures based on the formalism called the *polyhedral model*.

Alpha is a functional data parallel language invented in the api research group in the Irisa laboratory in Rennes. The project was headed by Patrice Quinton and the first definition of the language was proposed by Mauras in [Mau89] in 1989. The original motivation was to provide a language for expressing algorithms in an extended version of the formalism of recurrence equations proposed by Karp, Miller and Winograd [KMW67]. The goal of this language was to provide a high level tool for the synthesis of parallel regular vlsi architectures.

Although Alpha stands for the language itself, it is often also associated with the environment in which it is currently developed: MmAlpha. MmAlpha is an interface based on the Mathematica software from which one can manipulate Alpha programs.

1.2 What is Alpha for?

Alpha is currently a research tool which provides research problems in different computer science areas fields like: functional language semantics, parallelization, code generation, optimization, polyhedral theory, vlsi synthesis, systolic arrays, etc.

One important long term goal is to promote the use of high level functional languages for the synthesis of parallel vlsi architectures.

1.3 What can you expect from Alpha?

From the short term point of view, Alpha can be useful for:

- Providing a correct recurrence equation specification for a particular algorithm.
- Simulating such a specification.
- Transforming and simplifying a recurrence equation specification.
- Computing on convex polyhedra.
- Scheduling and detecting parallelism.

If you are ready to invest a little more time on MmAlpha, you will probably be able to

- Describe a systolic architecture with AlpHard (see section 12.1).
- Generate vhdl from this description.
- Provide a path from high level functional specification of an algorithm to the layout description of a vlsi algorithm realizing it.

If you becomed hooked to Alpha, you will be able to take advantage of a *completely open framework* in which you will be able to design your own tools, or interface your own programs: the source code of all functions described in this manual is available.

1.4 How does MmAlpha work?

MmAlpha is written in C (for a small part) and in Mathematica (for a larger part).

The user should only see the Mathematica interface. Mathematica provides an interpreted language with high level built-in function for symbolic computations: MmAlpha uses these facilities for transforming Alpha programs. Mathematica has also a nice programming language in which one can do general computation as in any other programming language.

The basic principle of the MmAlpha environment is the following one: Mathematica stores an internal representation of an Alpha program (called Abstract Syntax Tree or AST) and perform computations on this internal representation via user's commands. These commands can be for example: viewing the Alpha program, checking its correctness, adding a variable, generating C code to simulate it, generate a systolic array, generate vhdl code, etc. All the computations happen on the internal AST which is stored in the Mathematica environment.

Specific modules in C are used for two purposes: various parsers and unparsers and computations on polyhedra. All the C functions are accessed via Mathematica, but they can also be called from a C program: this is particulary useful for the polyhedral library. The polyhedral library, written by H. Le Verge and D. Wilde (see appendix 2.4.4) is the computationnal kernel of the MmAlpha environment. Its use provides many transformations that cannot be done easily by hand.

1.5 What is in this document

The organization of this document is as follows. Chapter 2 is a brief overview of the MmAlpha environment.

In chapters 3 and 4, we introduce Alpha following an architecture-oriented approach. In other words, we describe Alpha programs whose interpretation is always an architecture.

Chapter 6 illustrates the derivation of systolic architectures from a high-level description.

In chapter 5, we introduce Alpha as a functional data parallel programming language. This point of view is complementary to that of the first two chapters, in that it show how to use Alpha as an algorithmic language.

Chapter 7 deals with structured and parameterised Alpha, which gives the full power of the language, both for the description of hardware and the specification of algorithms.

Chapters 8 and 9 are more "practical" and tool-oriented. In chapter 8, we describe the static analyzer of MmAlpha : it is a useful tool to check some properties of an Alpha program. Chapter 9 explaines how to generate code and simulate Alpha programs.

Chapter 10 is devoted to scheduling Alpha programs. Scheduling is one of the essential steps of the derivation of architectures as well as compiling Alpha to sequential and parallel machines.

Chapter 12 (still under heavy work...) is intended to explain how AlpHard is defined and how to generate vhdl code frome Alpha.

1.6 How to read this tutorial?

Depending on your interest in MmAlpha, you will be interested in particular chapters of this tutorial.

First, read chapter 2: it does not hurt... and does not take too much time. By the way, there is a separate, slightly extended version of this chapter available in document [**?**].

Chapter 6 will be worth reading, if you are interested in systolic array design.

You may skip chapters 3 and 4 if you are not immediately interested in deeply understanding the mechanisms of Alpha.

If you are interested in writing a "real" Alpha program, you will have to read chapters 7, 8, 9, and 10.

Finally, if you want to produce "real hardware", read chapter 12.

1.7 Other documents

This tutorial is not the only information available on Alpha and MmAlpha. There are papers, for the academic oriented people. There are also many demonstration notebooks. To access them, start Mathematica, type the command `start[]`, and enjoy[1].

[1]or dislike...

1.8 What we plan in the (near?) future

The current tutorial is not yet complete. Several aspects of Alpha and MmAlpha (already implemented) need to be described in detail. Among them, the generation of Vhdl, the subset of Alpha called AlpHard, and the functions in MmAlpha which allow one to manipulate convex domains and Alpha programs. Knowing how long it takes to write and check a documentation, we have chosen to release a MmAlpha version without a complete documentation, for those who are interested in working with us.

Chapter 2

A guided tour of Alpha

Version of March 3, 2007

This chapter briefly presents the main features of the Alpha language and the basic operations of the MmAlpha environment. Section 2.1 presents several examples of Alpha programs. In section 2.2, we introduce and illustrate a few basic manipulations of Alpha programs, whereas section 2.3 presents more advanced transformations. Structured programs are shown in 2.4.

2.1 Examples of Alpha programs

For a complete description of the Alpha language, refer to appendix A.1 and A.2 or to [**?**, Mau89, DQR95]. We introduce the basic features of the langage on the exemple of the matrix-vector multiplication.

An Alpha program is a *system*. Alpha variables are generalized arrays which can have any shape (not just rectangles). The set of indices of the array is called the *domain* of the variable. Example 1 below shows the declaration of a variable `a` whose domain is the set of points (i, j) in the triangle: $0 \leq i \leq j$; $j \leq 10$.

Example 1**:**

```
  a :  {i,j |  0<= i <= j; j <=10 }
```

An Alpha system has input and output variables, it may have local variables and also size parameters which allows parameterized programs to be defined. Variables are defined by a single equation (which usually has the form of a recurrence equation). The `case` construct allows one to define different values in different parts of the domain.

Example 2**:**

```
  a[i,j] =
  case
  {| j = 0 } :  0[];
  {| j > 0 } :  a[i,j-1]+1[];
  esac;
```

The ALPHA expression of example 2 defines the values of `a[i,0]` to be zero[1] and recursively defines `a` at all the other points in its domain. This equation defines a variable `a` such that `a[i,j]=j`. The above equation is printed in *array notation*. The real syntax of ALPHA is sligthly less readable but more consistent and logical from a semantic point of view. To illustrate this, example 3 shows the same definition of `a` in standard notation.

Example 3**:**

```
a =
case
{i,j | j = 0 } :  0.(i,j->);
{i,j | j > 0 } :  a.(i,j->i,j-1)+1.(i,j->);
esac;
```

Expression `expr = a.(i,j->i,j-1)` should be read as `a` *composed with the dependency function* `f(i,j)=(i,j-1)`. In other words, `expr[i,j]` has value `a[i,j-1]` at each point `(i,j)` such that `(i,j-1)` is in the domain of `a`. Similarly `expr2 = 0.(i,j->)` means that `expr2[i,j]` has value `0` for all `(i,j)`.

Note also that there is no sequential in the different computations: interchanging the two branches of the case in example 2 would define exactly the same value for `a`: ALPHA is therefore a declarative language. The evaluation order is implicit and there are tools for finding schedules for a given program. As ALPHA is a functional language, the only constraint that an evaluation order must follow is the data dependencies between variables. In example 2, obviously `a[i,j]` must be computed after `a[i,j-1]`.

More details on the langage can be found in appendix A.1. Figure 2.1 shows an ALPHA program to compute the multiplication of a matrix of size $n \times n$ and a vector of size n.

2.2 Basic manipulation of ALPHA programs

This section presents the first commands that you should learn in order to deal with ALPHA programs. Before reading this section, you should have set the different environment variables necessary to use the MMALPHA environment. This procedure is described in appendix C.

There are several ways of using MMALPHA:

[1]Syntactic note: constants are zero dimensionnal arrays hence the empty brackets in `0[]`. This notation is the source of many syntax errors, and we plan to modify it...

```
system prodVect: {N | N>1}
               (a : {i,j|1 <= i,j <= N} of integer;
                b : {i|1 <= i <= N} of integer)
       returns (c : {i|1 <= i <= N} of  integer);
  var
C :  {i,j|1 <= i <= N; 0<= j <=N} of  integer;
  let
    C[i,j] = case
              {|j=0} : 0[];
              {|j>=1} : C[i,j-1] + a[i,j] * b[j];
             esac;
    c[i]=C[i,N];
  tel;
```

Figure 2.1: Alpha program describing the matrix vector multiplication

1. Using the notebook interface. Type `mathematica` under `Unix`, or start Mathematica 3.0 in the Programs menu of Windows NT.
2. Using the Mathematica kernel directly. Type `math` under `Unix`, or start Mathematica 3.0 Kernel in the Programs menu of Windows NT.
3. Using the Mathematica kernel via `emacs` (`Unix` only).

The second method is not recommended. The first one is easy, but some people prefer using the third one. In the following, we shall assume that we interact with the kernel (using the second or the third method), but all commands we describe can also be used in a notebook.

Once MmAlpha is installed, start Mathematica and write an Alpha program (such as the one of figure 2.1 for instance) using your favorite text editor. Say you called this file `prodVect.alpha`. The commands described in this section allow you to **load** your program into Mathematica, **view** the program in Mathematica (array notation or standard notation), **save** the program in another file, perform **static analysis** and **schedule** the program. By the way, all these examples are alse available in the `Getting-started` notebook accessible by the `Master` notebook of MmAlpha.

2.2.1 Loading and viewing an Alpha program

When Mathematica is loaded, a few welcome messages about Alpha are printed out. Then one gets the usual Mathematica prompt:
`In[1]:=`
The name of the working directory can be printed out by typing:
`In[1]:=Directory[]`

If you see that this directory is not the one where you have put `prodVect.alpha`, change it by typing:
`In[2]:= SetDirectory[" the directory you want "]`

You can now **load** the ALPHA program into MATHEMATICA by typing:
`In[3]:= load["prodVect.alpha"];`
Note that most often, MMALPHA commands should end with a semi-colon. The reason is the following one: MMALPHA commands are MATHEMATICA functions, which most often return a transformed ALPHA program, expressed as its AST. If you forget the `;` symbol, MATHEMATICA just prints out the result of the function evaluation, which sometimes may take a few pages... The side effect of `load` is to assign the AST to the global MATHEMATICA variable `$result`.

You can view the program that has been stored in `$result` by typing:
`In[4]:= show[]`
By default, `show` pretty prints the program contained in `$result`, but more generally,
`show[ var ]`
would pretty print the program contained in MATHEMATICA variable `var`.

By the way, all MATHEMATICA functions have an on-line documentation: `?show` gives the help on `show`. Commands may have options. Type `Options[ command ]` to list the options of `command` together with their default value.

You may have noticed that the program printed on the screen looks different from the one of figure 2.1: this program is in standard notation. If you want to print it in array notation (which is much more readable) you should evaluate:
`In[6] := ashow[]`

To save the ALPHA program in another file, use the command `save` (or `asave` which writes in array notation). For instance:
`In[7]:= asave["myFile.alpha"]`
will write program of figure 2.1 in file `myFile.alpha`. This command is needed if one wants to save the content of `$result` after some transformations.

2.2.2 Analyzing and simulating an ALPHA program

Now that you have loaded an ALPHA program, you can start working on it. Your first action should be to check it for so-called *static errors* by using the command `analyze`:
`In[8]:= analyze[]`
Information about possible errors of the ALPHA program are printed out. If the analysis is successful, the result is `True`. The static analyzer of ALPHA does essentially two verifications: it checks the type of expressions – this is not a fantastic novelty, – but it also checks that variables have a definition in any point of their domain definition. This second verification is very powerful, and is much more original.

Another interesting analysis tool is the `scheduler`. It finds (whenever possible) a linear schedule for your ALPHA program that minimizes its execution time. The use of the scheduler is detailed in subsection 2.3.4.

Suppose now that you want to evaluate the ALPHA program you just loaded. There is no real compiler for ALPHA but we can generate `C` code that evaluates ALPHA in a demand driven way. The command for generating `C` code is `writeC`:

```
In[9]:= writeC["prodVect.c","-p 10"]
```

The `"-p 10"` argument indicates that the value of the parameter `N` will be set to 10 (the `C` code is not parameter independent). By default, this program, once compiled, reads its input from the standard input (`stdin`) and prints on the standard output (`stdout`).

2.3 Advanced manipulation of ALPHA programs

In this section we briefly review a few more advanced manipulations of ALPHA. Additional information is in the ALPHA tutorial and the ALPHA reference manual (these documents are part of the MMALPHA distribution).

2.3.1 Pipelining

Pipelining is a transformation widely used in systolic synthesis. It is also called *localization* or *uniformization*. It consists basically of replacing a broadcast by the pipeline of this value through all the computations that need it.

For instance, in the program of figure 2.1, we see (last term in the second branch of the `case`) that `b[j]` is used for the computation of `C[i,j]` for all i, $0 \leq i \leq N$.

To introduce a new variable `B1` which will pipeline the `b[j]` value from the computation of `C[j,0]` to `C[j,1]`, ... , `C[j,N]`, we use the following command:

```
In[10]:= pipall["C","b.(i,j->j)","B1.(i,j->i+1,j)"];
```

In this expression, the first argument is the variable whose equation is to be modified, the second argument is the expression to be pipelined (standard notation is mandatory here) and the last argument indicates the *direction* of the pipeline as well as the name of the new variable introduced. After the execution of this command, the program contained in `$result` is the one shown in figure 2.2.

Unlike previous transformations, pipelining changes the ALPHA program, but the resulting program is equivalent to the initial one. The modifications performed automatically by `pipeAll` are:

1. Determine the domain of `B1` and add a declaration for it.

2. Build the definition of `B1` based on the dependency (`(i,j->i+1,j)`) and the initialisation given (`b.(i,j->j)`).

3. Replace the original expression (`b.(i,j->j)`) by `B1`.

```
system prodVect :{N | 2<=N}
                (a : {i,j | 1<=i<=N; 1<=j<=N} of boolean;
                 b : {i | 1<=i<=N} of boolean)
       returns  (c : {i | 1<=i<=N} of boolean);
var
  B1 : {i,j | 1<=i<=N; 1<=j<=N; 2<=N} of boolean;
  C : {i,j | 1<=i<=N; 0<=j<=N} of boolean;
let
  B1[i,j] =
      case
        {| i=1; 1<=j<=N; 2<=N} : b[j];
        {| 2<=i<=N; 1<=j<=N} : B1[i-1,j];
      esac;
  C[i,j] =
      case
        {| j=0} : False[];
        {| 1<=j} : C[i,j-1] + a[i,j] * B1;
      esac;
  c[i] = C[i,N];
tel;
```

Figure 2.2: Alpha program of figure 2.1 after pipelining of `b` in the definition of `C`

2.3.2 Change of basis

The change of basis is another important transformation in systolic array design. It allows variables to be re-indexed, and is often used to map indices to time and space.

In the example of figure 2.2, suppose that we wish to express the computations in a new index basis `i',j'` such that `i'=i+j`, `j'=j`. We can perform the following change of basis:

```
In[11]:= changeOfBasis["C.(i,j->i+j,j)"];
```

This simply indicates that the transformation is to be applied to variable `C` and that the new coordinates in term of the old ones are given by the linear function `(i,j->i+1,j)`. Note that a change of basis is meaningful only if this linear function admits an integral left inverse: in this example, its left inverse is obviously `(i,j->i-1,j)`. The resulting program is shown in figure 2.3.

2.3.3 Normalization

The normalization transformation simplifies an Alpha program into a particular normal form called *case-restriction-dependency*. This function is very useful when one performs several automatic transformations that may render the program less and less readable. The command is simply:

```
system prodVect :{N | 2<=N}
                (a : {i,j | 1<=i<=N; 1<=j<=N} of boolean;
                 b : {i | 1<=i<=N} of boolean)
       returns  (c : {i | 1<=i<=N} of boolean);
var
  B1 : {i,j | 1<=i<=N; 1<=j<=N; 2<=N} of boolean;
  C : {i,j | j+1<=i<=j+N; 0<=j<=N} of boolean;
let
  B1[i,j] =
      case
        {| i=1; 1<=j<=N; 2<=N} : b[j];
        {| 2<=i<=N; 1<=j<=N} : B1[i-1,j];
      esac;
  C[i,j] =
      case
        {| j=0} : False[];
        {| 1<=j} : C[i-1,j-1] + a[i-j,j] * B1[i-j,j];
      esac;
  c[i] = C[i+N,N];
tel;
```

Figure 2.3: Alpha program of figure 2.2 after the change of basis on `C`

`In[12]:= normalize[];`
This transformation is illustrated in the Alpha tutorial.

2.3.4 Scheduling

The `schedule` command looks for a schedule for an Alpha program. The basic goal of the scheduler is to find a valid and good evaluation order. Here, the term *good* depends on the optimization criterion choosen: most often, it is the total evaluation time of the program, but one may also consider other criteria.

The time is considered as a discrete single rate clock. The overall idea of the scheduling process is to build a linear programming problem (LP) and to solve it with a software tool: this may be PiP[**?**], or LP-Solve[**?**], or even the Mathematica linear solver.

The Alpha scheduler provides several options to schedule a program. We consider here the simplest one (by default), called *monodimensional affine-by-variable schedule*. This esoteric name means that the evaluation date $T_A(i,j)$ of a given computation `A[i,j]` is given by an affine function of the indices and parameters:

$$T_A(i,j) = \tau_A^i i + \tau_A^j j + \tau_A^N N + \alpha_A$$

where N is a parameter of the Alpha program. The coefficients of this function are (in general) different for each variable in the system

The command to schedule a program is :
`schedule[]`
By default, it schedules `$result` and the resulting schedule is placed in a global variable named `$schedule`.

The `schedule` function has many options (type `Options[schedule]` for further information.) Some uses of the function `schedule` are:

- `schedule[]`
  find an affine by variable schedule for `$result` which minimizes the global execution time and assign it to `$schedule`.

- `schedule[sys]`
  find an affine by variable schedule for the Alpha system `sys` which minimizes the global execution time and assign it to `$schedule`.

- `schedule[{option1->value1,...,optionn->valuen}]`
  find a schedule for `$result` which respects the chosen options and assign it to `$schedule`.

- `schedule[sys,{option1->value1,...,optionn->valuen}]`
  find a schedule for the Alpha system `sys` which respects the chosen options and assign it to `$schedule`.

The schedule function is explained in more detail in the scheduler documentation given in file:

```
$MMALPHA/doc/user/docSched.dvi
```

2.4 Structured Alpha programs

Alpha programs can be structured: this section explains how this can be done.

2.4.1 Simple structures

Let us write an Alpha program for the addition of two integers (or fixed-point numbers) expressed as bit vectors. A binary adder is classically described as a sequence of *full adder* operations with the propagation of a carry bit from one full adder to the next one, as shown in Fig.2.4.

The following Alpha system describes a *full adder*:

```
system FullAdder (A,B,Cin : boolean)
         returns (X,Cout : boolean);
let
  X = A xor B xor Cin;
  Cout = (A and B) or (A and Cin) or (B and Cin);
tel;
```

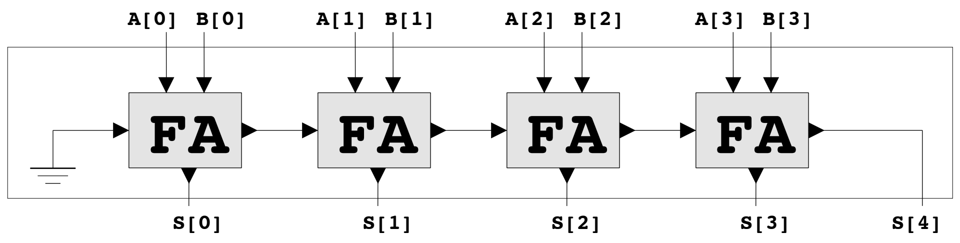


Figure 2.4: Addition of two integers (coded as bit vectors), using *full adders*

To build an adder using this program, we need to instanciate a collection of such system, as shown in Fig.2.4. The shape of this collection may be expressed as the ALPHA domain `{b|0<=b<W}` where `W` is a size parameter giving the number of bits of the adder.

The **use** construct of ALPHA allows precisely that: the following system describes in ALPHA the adder given in figure 2.4:

```
system Plus: {W|W>1} (A,B: {b| 0<=b<W} of boolean)               -- 1
             returns (S : {b| 0<=b<=W} of boolean);              -- 2
var                                                              -- 3
  Cin, Cout, X : {b| 0<=b<W} of boolean;                         -- 4
let                                                              -- 5
  Cin[b] =                                                       -- 6
    case                                                         -- 7
      {| b=0} : 0[];                                             -- 8
      {| b>0} : Cout[b-1];                                       -- 9
    esac;                                                        -- 10
  use {b| 0<=b<W} FullAdder[] (A,B,Cin) returns(X, Cout);        -- 11
  S[b] =                                                         -- 12
    case                                                         -- 13
      {| b<W} : X;                                               -- 14
      {| b=W} : Cout[W-1];                                       -- 15
    esac;                                                        -- 16
tel;                                                             -- 17
```

In this system, line 11 reads as follows:

> "Use (or instantiate) a collection of instances of the subsystem `FullAdder`. This collection has the shape of the extension domain `{b| 0<=b<W}` and is thus indexed by index `b`. Let the inputs of the `b`-th instance be the variables `A`, `B` and `Cin` at point `b`, and similarly let the outputs of this collection of instances be the variables `X` and `Cout`."

Lines 6-10 describe the carry propagation, and lines 12-16 define the output of this binary adder.

In other words, line 11 is a shortcut for the following equations, which are those of the system `FullAdder` whith the dimension of the variables extended from zero to one:

```
        X[b] = A[b] xor B[b] xor Cin[b];
        Cout[b] = (A[b] and B[b]) or (A[b] and Cin[b]) or (B[b] and Cin[b]);
```

2.4.2 Syntax of the *use* construct

The **use** construct appears at the syntactic level of an equation, since it is basically a shortcut for a set of equations. Here is the general syntax of an equation/use (see appendix A.1 for the meta syntax):

```
Equation ::=
      Identifier = Expression ;
   |  use [  ExtensionDomain ] Identifier
                 [ [ParamAssignment ] ]
                 ( ExpressionList )
        returns  ( IdentifierList ) ;
```

In this syntax we see that there is an optional *parameter assignment* which is discussed in the following. In the previous addition the subsystem `FullAdder` has no parameters, and the parameter assignment is therefore empty.

2.4.3 Manipulating structured programs

A structured program is stored in MmAlpha as a Mathematica list of systems called a *library*. The default library is stored in the global variable `$library`.

A structured program may be written in one single file or several distinct files. In the former case the `load[]` function returns a library composed of all the systems contained in the file, and stores this library in `$library`.

If the program is stored in several files, it is the responsability of the user to build a proper library, i.e. a MmAlpha list of all the systems needed by the hierarchical structure of the program. For this purpose, the user will typically use MmAlpha list manipulating functions such as `Join[]`, `Append[]`, etc.

In addition, two functions, `putSystem[]` and `getSystem[]`, may be used to extract a system from a library and to put back a modified system in a library. Typically a system is extracted from the library as the *current* system, modified by some program transformation, and then put back in the library.

2.4.4 Program transformations associated with structures

Most MmAlpha functions handle parameterized programs and *use* statements. There are, however, some major exceptions such as the `writeC[]` translator which generates code only for flat Alpha programs without subsystems. MmAlpha provides functions to transform a structured program into a “flat” equivalent one :

- `assignParameterValue[]` gives a value to a size parameter, i.e. it refines a generic system into a specialized one.
- `inlineSubSystem[]` expands a *use* statement, replacing it with the equations of the corresponding subsystem, properly modified to take the dimension extension into account.
- `inlineAll[]` recursively flattens a structured Alpha program.

For more information see the subsystem documentation

`$MMALPHA/doc/user/SubSystems.dvi`

Appendix: The polyhedral library

See [Wil94](`http://www.irisa.fr/EXTERNE/bibli/pi/pi785.html`

Chapter 3

Modelling synchronous architectures

1. Redraw figures (after reinstalling X11 and xfig)

In this chapter, we present the Alpha language through the description of very simple – and seemingly artificial – examples of synchronous circuits. Our goal is to show the basic constructors of the language: their syntax, their meaning, and how they can be used in practice.

3.1 Pointwise operators

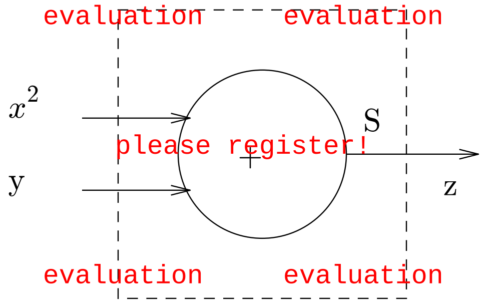


Figure 3.1: An adder

Consider the architecture shown in figure 3.1: there is a simple adder, which takes to input flows, respectively `x` and `y`, and returns `S`.

This architecture is described by the Alpha program of figure 3.2. A program is called a *system*. Line 1 is the head of the system, with name `exemple1`. This system has two input arguments named `x` and `y`. These arguments, called *variables*, are defined on the set {`t | 1 <= t`}. Line 3 defines a local variable `S` (for the moment, this variable is useless, but we shall soon find a use for it). Between the `let` and `tel` keywords of lines 4 and 7, we have two equations: line 5 says that `S` is the pointwise sum of `x` and `y`. In other words, $\forall t, S_t = x_t + y_t$. Line 6 says that `z` and `S` are identical.

```
system adder (x,y: {t | 1 <= t} of integer)          -- 1
returns (z: {t | 1 <= t} of integer);                 -- 2
var S: {t | 1 <= t} of integer;                       -- 3
let                                                   -- 4
  S = x + y;                                          -- 5
  z = S;                                              -- 6
tel;                                                  -- 7
```

Figure 3.2: An adder in ALPHA

Processing this example in MMALPHA

Here are the Mathematica expressions that allow one to try the previous example under MMAlpha.

```
load["adder.alpha"]     (* loads the alpha file *)
ashow[];                (* shows its content *)
```

3.2 Delays as dependence fonctions

We now modify this example, by adding a delay on the output, as shown in figure 3.3. The corresponding ALPHA program is shown in figure 3.4. In this example, the definition

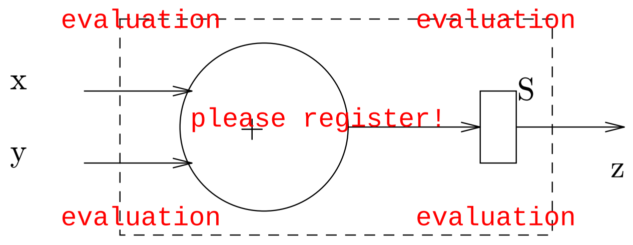


Figure 3.3: Adder delayed

domains of `z` and `S` have changed: indeed, the output of the adder is defined only one cycle after the input. Thus, `z` and `S` are declared on domain `{t|2<=t}`. Instruction 4 now defines `S` as being the sum of variables `x+y`, delayed by 1.

3.3 Retiming

ALPHA provides tools for retiming synchronous architectures. In the example of figure 3.3, one can shift the delay from the output of the adder to its inputs. This corresponds to replacing line 4 in program 3.4 by `S[t] = x[t-1] + y[t-1];`. The corresponding architecture is shown in figure 3.5.

```
system adderDelayed (x,y: {t|1<=t} of integer)    -- 1
returns (z: {t|2<=t} of integer);                 -- 2
var S: {t|2<=t} of integer;                       -- 3
let
  S[t] = (x + y)[t-1];                            -- 4
  z = S;                                          -- 5
tel;
```

Figure 3.4: Adder plus delay

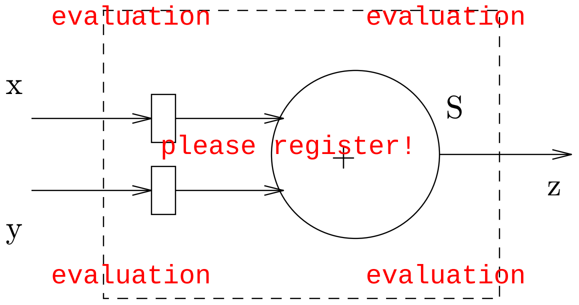


Figure 3.5: Retiming the adder

Retiming in MmAlpha

Do

```
load["adderDelayed.alpha"]; (* load the alpha program *)
ashow[];                      (* show it *)
normalize[];                  (* simplify it *)
ashow[];                      (* show the result *)
asave["file.alpha"];          (* save program *)
```

3.4 Case and restrictions

Two other constructs of the language are the case and the restriction. Consider the example of figure 3.6, which shows a new delayed and retimed version of the adder, where the output is taken through a multiplexer. In other words, depending on the value of index `t`, `S` is either 0 or the sum `x[t-1] + y[t-1]`. This is reflected in the definition of `S` in figure 3.7 (lines 2–5.) This definition uses a case expression. Each branch of the case is an Alpha expression. The first one is the constant 0, denoted as `0[]` in Alpha. It is considered only when `t` is equal to 1, which is reflected in the condition {`| t = 1`}. The second branch corresponds to the expression `x[t] + y[t]`, taken when the condition {`|t>1`} is true. Actually, we call an expression such as {`|t>1`} a *restriction*.

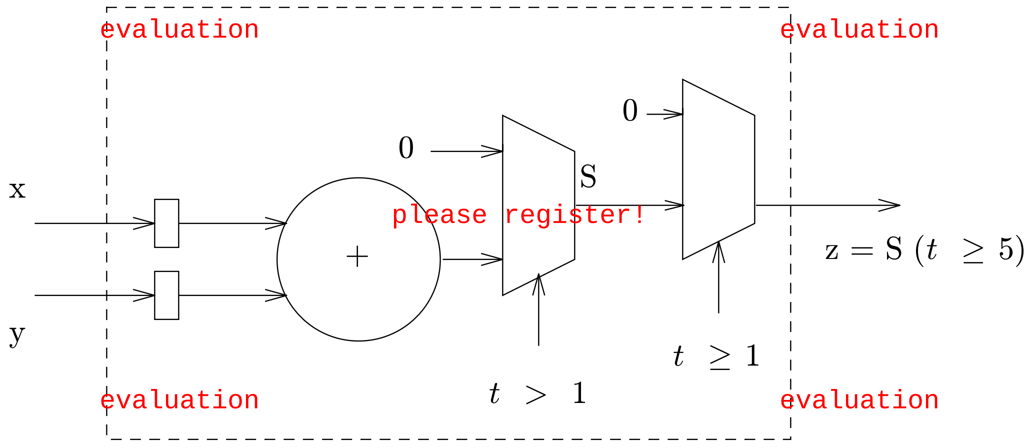


Figure 3.6: Adding a multiplexer

```
system adderMultiplexer (x,y: {t|1<=t} of integer)
returns (z: {t|5<=t} of integer);                    -- 1
var S: {t|1<=t} of integer;
let
 S[t] = case                                         -- 2
     {|t=1}: 0[];                                    -- 3
     {|t>1}: x[t-1] + y[t-1];                        -- 4
     esac;                                           -- 5
 z[t] = {|t>=5}: S[t];                               -- 6
tel;
```

Figure 3.7: Adder and multiplexer

The definition of `z` also uses a restriction. The definition of line 6 says that `z` is defined as `S`, but only for time instants `t` such that `{|t>=5}`.

Exercices in MATHEMATICA

Load and parse program adderMultiplexer.alpha.

```
load[adderMultiplexer.alpha];
ashow[adderMultiplexer];
```

3.5 Array and full form of ALPHA

In all the above examples, we have used the so called *array form* of ALPHA. The full form of the language is actually a little bit different, and a little bit more difficult to read initially. It is however needed in order to understand some of the properties of the language.

Let us consider again the definition of `S` in program 3.7, line 2 – 5:

```
S[t] = case
    {|t=1}: 0[];
    {|t>1}: x[t-1] + y[t-1];
    esac;
```

The full form of this expression is in fact:

```
S = case
     {t|t=1}: 0.(t->);
     {t|t>1}: x.(t->t-1) + y.(t->t-1);
 esac;
```

To explain this, we need to go more deeply in the structure of the language. In ALPHA, any expression is a function from a set of points – called the *domain* of the expression –, to some value set depending on the type of the expression. Expressions are either

- variables – e.g. `S`,
- scalar constants
- pointwise operations on expressions – e.g. `x + y`
- application of dependence functions to expressions, – e.g. `exp.f` where `f` is a dependence function explained later,
- restriction of expressions, – e.g. `dom:  exp`, where `dom` is a domain description,
- case expressions – e.g. `case e1 ; e2; esac`, where `e1` and `e2` are expressions.

Let us examine expressions in more details.

3.5.1 Domains

Domains of ALPHA are restricted to *polyhedra*, (more precisely, unions of finitely may polyhedra). A polyhedron is a set of integral points whose coordinate satisfy linear inequalities. For example: {`t | 1 <= t <= 10`} denotes such a set. So also does {`i,j | i>=10; 0 <= j <= i`} for example. {i — 0 ¡ i ¡ 10} —— {i — 15 ¡= i} is a union of polyhedra.

3.5.2 Variables

Variables are functions from a domain to a set of values. For example, in program 3.7, `S` is a function from the domain {`t|1<=t`} to the set of integers. Thus, `S` represents a collection of indexed values S_t, where t belongs to the domain of `S`.

ALPHA is a strongly-typed language, where variables are declared, together with their domain. This allows many compile time checks to be performed (see chapter 8).

Input variables are defined in the head of the system definition. Output variables are defined after the key-word `result`. Local variables are defined after the key-word `var` and the keyword `let`.

3.5.3 Constants

In ALPHA, constants are considered as functions from the singleton set Z^0 to a type. For example, 1 is the function from Z^0 to the integer value 1.

3.5.4 Pointwise operations

Pointwise operations generalize classical operations like +, -, * on integers, or `and`, `not`, `or` on booleans. So, `x + y` represents the pointwise operation on values x_t and y_t for all t. There is however, an important point to notice: if `op` is a pointwise operator and `e1` and `e2` are expressions, then the domain of the expression `e1 op e2` is the intersection of the domains of `e1` and `e2`.

3.5.5 Dependence functions

Dependence functions are affine functions on indices. They are denoted as

```
(idx, idx, ...  -> iexp, iexp, ...  )
```

where `idx` is an index name and `iexp` is an affine function of indices. For example: `(i,j -> i+2, j+3i-4)`} denotes such a function. So does also the function `f=(t -> t-1)`. Therefore, `x.(t -> t-1)` is an ALPHA expression, whose meaning is as follows. Remember that `x` is a function from the domain {`t | 1<=t`} to the set of integers. The composition of functions `x o f` is therefore defined, and this is just what `x.(t -> t-1)` represents. It is easy to see that this function is defined on the domain {`t | t>=2`}. More generally, the domain of any expression `e.f` is $\texttt{f}^{-1}(\text{dom}(\texttt{e}))$, that is to say, the pre-image of the domain of `e` by function `f`.

Note that a function `(->t)` denotes a mapping from the set Z^0 to Z. Similarly, `(t ->)` denotes a mapping from the set Z to Z^0. Such functions are used to extend scalar constants to higher dimensional domains.

3.5.6 Restriction expressions

As said before, restrictions allows one to restrict the definition of an expression to a sub-domain. To be consistent with the other language constructs, restrictions are defined as polyhedra, and therefore are denoted just as domains. For example, {`t | t>=5`}`: S` is a restriction of the expression `S` to domain {`t | t >= 5`}. The domain of a restricted expression `dom: e` is the intersection of the domain of the expression and the domain defined by the restriction, i.e., `dom` ∩ `e`.

3.5.7 Case expressions

As already seen, case expressions allow one to define a new function case by case. A case expression is defined on the union of the domains of its branch. Let `case exp1; exp2 esac` be a case expression, with branches `exp1` and `exp2`. Let `d1` and `d2` be respectively the domains of `exp1` and `exp2`. Assume that `d1` and `d2` have the same dimensionality and are disjoint (these properties are checked during static analysis). Assume also that `exp1` and `exp2` have the same type, say `T`. Then `case exp1; exp2 esac` is a function from `d1` ∪ `d2` to `T`, whose value is that of `exp1` on `d1`, and that of `exp2` on domain `d2`. For a detailed desciption of the domain computations and its use for the static analysis, see section 8.2.

Back to our example

We are now ready to consider again the example corresponding to figure 3.7. The definition of `S` is:

```
S = case
     {t|t=1}: 0.(t->);
     {t|t>1}: x.(t->t-1) + y.(t->t-1);
 esac;
```

Let us analyze each branch. `0` is a function from Z^0 to the integer value 0. Composing this function with the function `(t ->` extends the constant `0` to the set Z. In other words, `0.(t->)` denotes the function from Z to integers, whose value is `0` everywhere. {`t | t=1`}`: 0.(t->)` denotes the restriction of this function to the single point `t=1` of Z.

We can analyze the second branch in a similar fashion. We would conclude that it represents a function from the subset {`t | t>1`} of Z, whose value at point `t` is $x_{t-1} + y_{t-1}$.

Finally, the total case expression defines `S` as a function from {`t | t >=1`} to the integer set, whose value is `0` when `t = 1`, $x_{t-1} + y_{t-1}$ otherwise.

3.5.8 Array form

The array form is just a convenient way of presenting ALPHA programs, in a more "readable" form. The rule is simple to explain on the definition of `S`:

- Dependence functions of the form `.(indices -> exprs)` are rewritten as `[exprs]`,
- Restrictions of the form {`indices | inequalities`} are rewritten {`| inequalities`},
- Left-hand side variables are appended with the indices from the right hand side.

Thus, the definition of `S` in array form is

```
 S[t] = case
      {|t=1}: 0[];
      {|t>1}: x[t-1] + y[t-1];
  esac;
```

This notation can be used safely for any normalized expression, as we shall see in section 3.6.

Exercices in MMALPHA

Load a program, and see its full form using the `show[]` function, or its array form using `ashow[]`.

3.6 Substitution and normalization

One of the properties of ALPHA is called the referential transparency: it means that the meaning of an expression is independent of the context where it appears. For this reason, any symbol can be safely replaced by its definition. For example, by replacing the `S` symbol in the definition of `z` in program 3.7, we obtain the program shown in figure 3.7.

Let us consider the new definition of `z`:

```
z = {t | 5<=t} :
         case
           {t | t=1} : 0.(t->);
           {t | 2<=t} : x.(t->t-1) + y.(t->t-1);
         esac.(t->t);
```

It is rather intuitive that the first restriction {`t | 5<=t`} can be distributed in the branches of the case expression. Combining (i.e., intersecting) this restriction with the {`t | t=1`} leads obviously to an empty set, and consequently, to an empty branch. Similarly, combining {`t | 5<=t`} with {`t | 2<=t`} leads to {`t | 5<=t`}. In summary, we can imagine that the full definition of `z` can be simplified to something like `z = x.(t->t-1) + y.(t->t-1)`.

```
system adderMultiplexer (x : {t | 1<=t} of integer;
                         y : {t | 1<=t} of integer)
       returns          (z : {t | 5<=t} of integer);
var
  S : {t | 1<=t} of integer;
let
  S = case
        {t | t=1} : 0.(t->);
        {t | 2<=t} : x.(t->t-1) + y.(t->t-1);
      esac;
  z = {t | 5<=t} :
         case
           {t | t=1} : 0.(t->);
           {t | 2<=t} : x.(t->t-1) + y.(t->t-1);
         esac.(t->t);
tel;
```

Figure 3.8: Program of figure 3.7 after substitution of `S` in definition of `z`

```
system adderMultiplexer (x : {t | 1<=t} of integer;
                         y : {t | 1<=t} of integer)
       returns          (z : {t | 5<=t} of integer);
var
  S : {t | 1<=t} of integer;
let
  S[t] =
      case
        {| t=1} : 0[];
        {| 2<=t} : x[t-1] + y[t-1];
      esac;
  z[t] = x[t-1] + y[t-1];
tel;
```

Figure 3.9: Normalization of program 3.8 (in array form)

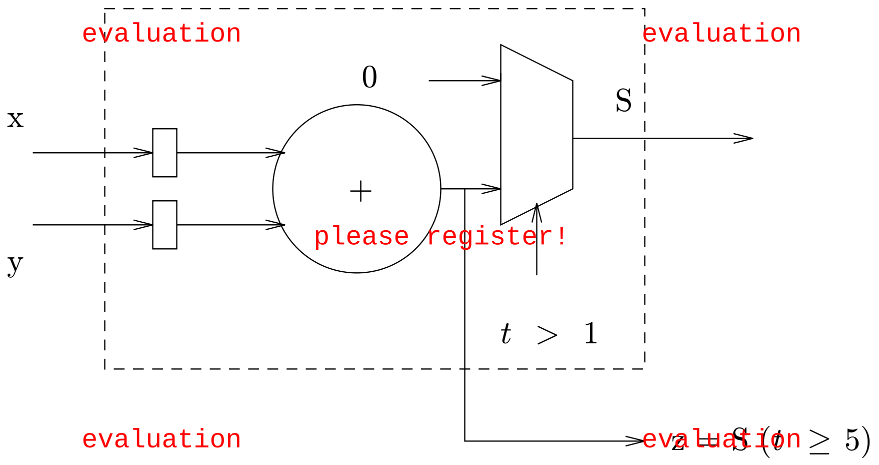


Figure 3.10: Architecture after substitution and normalization

These intuitive properties are in fact true, and such a simplification is done in a systematic way by means of the normalization transformation: one can prove that any expression can be simplified into a normalized one, with at most one level of outermost case, one level of restrictions, and one level of dependences.

After normalization, the program becomes that of figure 3.9.

We can interpret this program as representing the synchronous architecture of figure 3.10. We can see – and this is of course straightforward – that the multiplexer is indeed useless, since we are interested in the value of `z` only when `t>1`. This normalization property is one of the most important one of Alpha.

Exercices in Mathematica

Exercice 1

Write an Alpha program to model the following architecture:

Exercice 2

Write an Alpha program to model the circuit of figure 3.12.

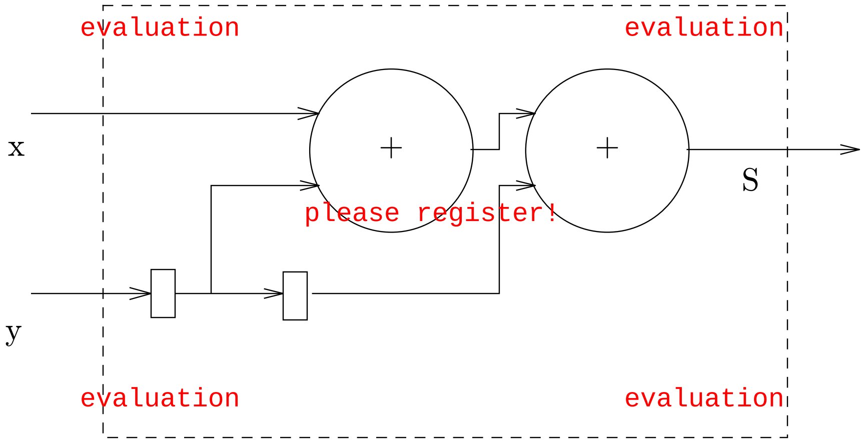


Figure 3.11: Exercice 1

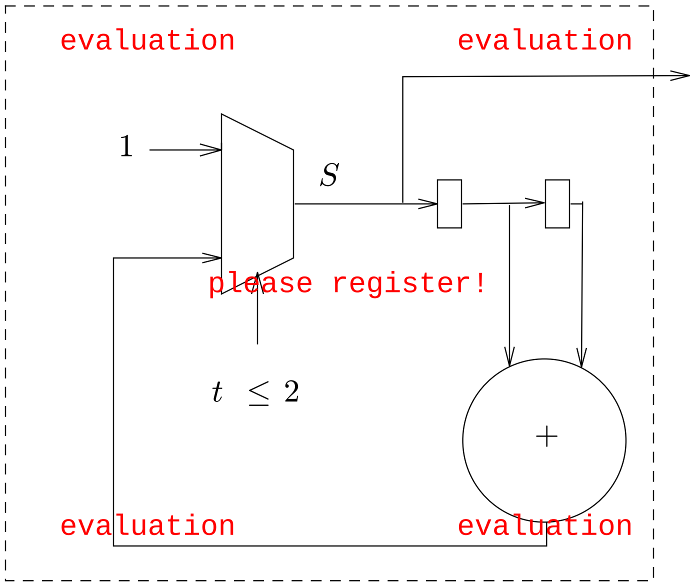


Figure 3.12: Exercice 2

Chapter 4

Arrays of synchronous operators

In this chapter, we show how arrays of hardware operators can be modelled in ALPHA. In chapter 3, the basic operators of the language were described. As we shall see here, a simple extension of these operators allows two-dimensional – and higher-dimensional – variables and expressions to be used.

4.1 Describing array of elements

Figure 4.1 shows an array of adders.

4.2 An array of adders

The corresponding ALPHA code is shown on figure 4.2. Blabla

4.3 Change of basis

Any ALPHA program can be seen as a black box which takes input variables, and produces output variables. Therefore, intermediate variables are unimportant, and, in particular, one can reindex them using one-to-one index mappings. Obviously,

```
A[i] = B[i];
B[i] = C[i];
```

is identical to

```
A[i] = B[i-1];
B[i] = C[i+1];
```

The change of basis operation does just this reindexing. In the context of ALPHA, the reindexing functions that we consider need to be restricted to affine *unimodular functions*, that is to say, affine mappings which admit an integral inverse.

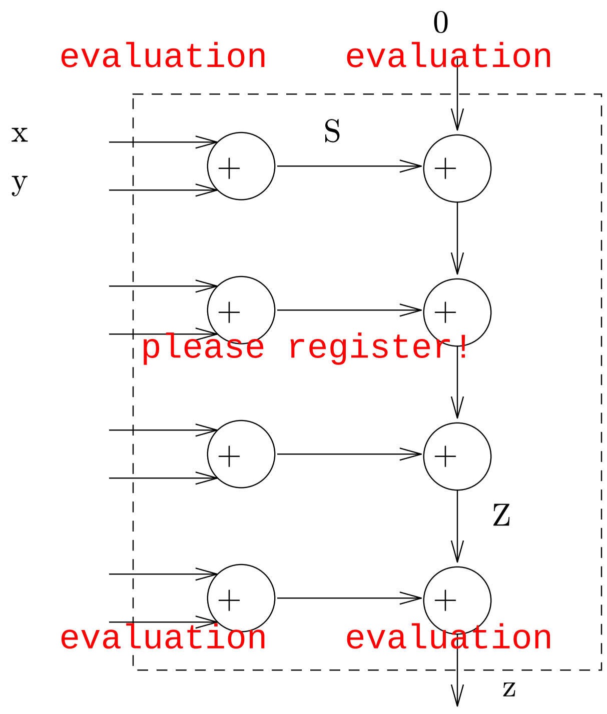


Figure 4.1: An array of adders

```
system adderarray (x,y : {t,p|t>=1;4>=p;p>=1} of integer)
       returns (z : {t|t>=2} of integer);
  var
    Z : {t,p|t>=1;4>=p;p>=0} of integer;
    S : {t,p|t>=1;4>=p;p>=1} of integer;
  let
    S[t,p] = x[t,p] + y[t,p];
    z[t] = Z[t,4];
    Z[t,p] = case
        {|t>=1;p=0} : 0[];
        {|t>=1;4>=p;p>=1} : Z[t,p-1] + S[t,p];
        esac;
  tel;
```

Figure 4.2: An array of adders in Alpha

Consider variable Z in program 4.2. Consider the reindexing function B from $\mathbf{Z^2}$ to itself, defined by $B(t,p) = (t+p,p)$. One can show that B is unimodular. Let us denote B using the ALPHA syntax, as `(t,p -> t+p,p)`. This function is one-to-one, and its inverse is B^{-1}, denoted `(t,p -> t-p,p)`.

To apply the change of basis defined by B on variable Z, we need to do the following operations:

1. Replace the domain of definition of Z by its image under the B mapping. As the domain of Z is {t,p|t>=1;4>=p;p>=0}, its image by B is {t,p | p+1<=t; 0<=p<=4}.
2. Replace any occurrence of Z in a right-hand side of an equation by `Z.(t,p -> t+p,p)`.
3. Replace the entire equation `Z = expr` defining Z by `Z = expr.(t,p -> t-p,p)`.

The new program is shown in figure 4.3. We can normalize it, which leads to program 4.4.

```
system adderarray (x : {t,p | 1<=t; 1<=p<=4} of integer;
                   y : {t,p | 1<=t; 1<=p<=4} of integer)
       returns    (z : {t | 2<=t} of integer);
var
  Z : {t,p | p+1<=t; 0<=p<=4} of integer;
  S : {t,p | 1<=t; 1<=p<=4} of integer;
let
  S = x.(t,p->t,p) + y.(t,p->t,p);
  z = (Z.(t,p->t+p,p)).(t->t,4);
  Z = (case
        {t,p | 1<=t; p=0} : 0.(t,p->);
        {t,p | 1<=t; 1<=p<=4} : (Z.(t,p->t+p,p)).(t,p->t,p-1) + S.(t,p->t,p);
      esac).(t,p->t-p,p);
tel;
```

Figure 4.3: Program adderarray, after change of basis `(t,p->t+p,p)` performed on Z

This new program can be interpreted as the architecture shown in figure 4.5.

Processing this example in MMALPHA

Load the array of adders:

```
load["adder-array.alpha"];    (* loads the alpha file *)
ashow[];                      (* shows its content *)
```

Apply a change of basis:

```
system adderarray (x : {t,p | 1<=t; 1<=p<=4} of integer;
                   y : {t,p | 1<=t; 1<=p<=4} of integer)
       returns    (z : {t | 2<=t} of integer);
var
  Z : {t,p | p+1<=t; 0<=p<=4} of integer;
  S : {t,p | 1<=t; 1<=p<=4} of integer;
let
  S[t,p] = x[t,p] + y[t,p];
  z[t] = Z[t+4,4];
  Z[t,p] =
      case
        {| 1<=t; p=0} : 0[];
        {| p+1<=t; 1<=p<=4} : Z[t-1,p-1] + S[t-p,p];
      esac;
tel;
```

Figure 4.4: Program adderarray, after change of basis and normalization

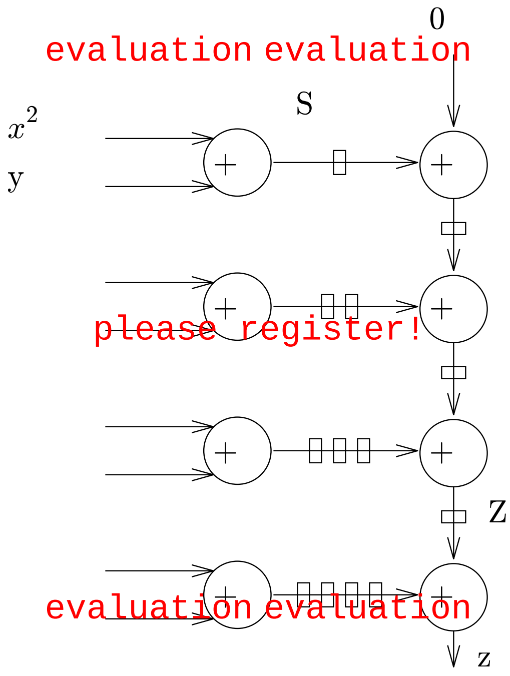


Figure 4.5: An array of adders

```
changeOfBasis["B.(t,p->t+p,p)"];  (* apply a change of basis *)
ashow[];                          (* shows the result *)
```

Notice that the result of a change of basis is automatically normalized.

4.4 Conclusion

In this chapter, we have seen how arrays of elements can be described using ALPHA. In order to present hardware examples, we have limited ourself to programs where the indexes can be interpreted as the time and the space (i.e., `t` and `p`). Actually, ALPHA programs are not restricted to such indexes. *Au contraire*, hardware-like ALPHA programs represent the ultimate goal of a synthesis process which starts from a behavioural specification. They are in fact a proper subset of all ALPHA programs.

In Chapter 5, we shall consider ALPHA from the data parallel perspective.

Chapter 5

The Alpha Language and System – Another point of view

5.1 Introduction

The last figure is garbage. The psfig inside has been commented out. See reference document in old pdf turorial. Alpha is a functional language, based on recurrence equations. All variables are type-declared at the beginning of a program, and represent multi-dimensional arrays, whose shapes can be arbitrary polyhedra. For example, the declaration `A: {i,j| 0<j<i<=N} of real`, specifies a (strictly) lower triangular, real matrix. To introduce the main features of the language, consider the problem of solving a system of linear inequalities, $Ax = b$, where A is a lower triangular, $n \times n$ matrix with unit diagonal. It is well known that this can be solved using forward substitution, as given by the following formula.

$$\text{for } i = 1 \ldots n, \quad x_i = b_i - \sum_{j=1}^{i-1} A_{i,j} x_j$$

The corresponding Alpha program (Fig. 5.1) is identical, except for syntactic sugar. The first line names the system and declares that it has a positive integer parameter, `N`. The next three lines are declarations of the input and output variables of the system. Note how each

```
system ForwardSubstitution : { N | N>1 }       -- comments are like this
                 ( A : { i,j | 0<j<i<=N } of real;    -- a 2D input variable
                   B : { i   | 0<i<=N } of real)      -- a 1D input variable
         returns ( X : { i   | 0<i<=N } of real );    -- a 1D output variable
  let
    X[i] = B[i] - reduce(+, (i,j -> i), A * X.(i,j->j))[i]
  tel;
```

Figure 5.1: Alpha program for the forward substitution algorithm

variable is defined over a certain range of indices, specified by a set of linear inequalities. This defines a polyhedron, called its *domain*. For example, the domain of `A` is triangular, while `B` and `X` are one dimensional vectors. Variables declared before the "`returns`" keyword are input variables, and those after it are outputs. Although our example does not illustrate it, a system may also have local variables, which are declared after the system header, using the keyword `var`. The body of the program is a number of equations delineated by the `let` and `tel` keywords. Here, we have a single equation corresponding to the above formula, which specifies that we compute `X[i]` by performing a reduction (any associative and commutative binary operator may be used; here we have `+`). The `(i,j->i)` specifies a projection of a two dimensional index space to one dimension. Intuitively, it states that within the body of the `reduce`, we have two index variables, `i` and `j`, but the latter is not visible outside the scope of the `reduce`. The body of the `reduce` is the expression, `A * X.(i,j->j)`, where `(i,j->j)`, is a *dependency*. It denotes the fact that to compute the body at index point, `[i,j]`, we need the value of `X` at index point `j` (the dependency of `A` is not explicitly written—it is the identity). Dependencies are an important aspect of Alpha. They describe "indexing functions" and have the syntax, `(idx, idx, ... -> i-expr, i-expr, ...)`, where each `idx` is an index name, and `i-expr` is an *affine* expression of the system parameters and the `idx`'s. The Alpha system uses this syntax for specifying a multidimensional affine function in many different contexts.

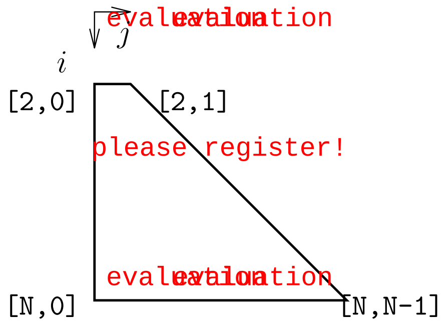
5.2 Polyhedra and affine functions

Alpha imposes the restriction that all variable domains are polyhedral, and all dependencies are *affine* functions. Although this may constrain our expressive power, it is a conscious choice that was made for many reasons. A large number of linear algebra and DSP and image processing algorithms fall within these limitations. It is also known that any nested loop program (in a conventional language) that satisfies certain constraints (loop bounds are affine functions of the parameters and outer indices, and the body contains only assignments to array variables which are accessed using only affine functions of the loop indices) can be modeled by a formalism that is similar to an Alpha program. Such assumptions are made in many techniques used in parallelizing compilers. By concentrating explicitly on this subset of programs, we can make use of powerful static analysis and optimization tools based on polyhedra and linear programming (implemented in our **polyhedral library**. Finally, it is precisely these restrictions that interact coherently and form the foundation of our transformational system. They also ensure that all Alpha programs can be put into a "normal" form, and in fact, all examples that we use here have been so normalized.

5.3 Transformations in Alpha

We now introduce some of the Alpha transformations used in converting our program into a single assignment C program.

```
            ⋮
var
  f : {i,j | (2,j+1)<=i<=N; 0<=j} of real;
let
  f[i,j] = case
        {| j=0} : 0[];
        {| 1<=j} : f[i,j-1] + A * X[j];
           esac;
  X[i] = case
        {| 2<=i} : B[i] - f[i,i-1];
        {| i=1} : B[i];
         esac;
tel;
```



The domain of `f`.

Figure 5.2: The forward substitution program after serialization

5.3.1 Serialization

The first one is used to **serialize** the reduction operation, and requires two parameters: the "direction" in which we accumulate the partial sums, and the name of the temporary variable. For our example, if we choose to accumulate in the *increasing* direction of `j` (specified as "`(i,j->i,j+1)`") and name the new variable, `f`, we obtain the Alpha program shown in Fig. 5.2. We also see two new syntactic constructs, the *case* (which has the usual meaning), and the *restrict*, which has the syntax, `<polyhedron>: <expr>`, and denotes the expression, `<expr>` but restricted to the subset of index points in `<polyhedron>`.

Observe that the domain of `f` is `{i,j| 0<=j<i; 2<=i<=N}` (a "nearly triangular" trapezium defined by the points `[2,0] [2,1] [N,N-1]` and `[N,0]`). Also note that the equation for `X` now has the expression `f[i,i-1]` replacing the reduction, and the new equation for `f` consists of an initialization at the boundary, and an accumulation using the *body of the original reduction*. All this is automatically determined by the system, using the polyhedral library.

5.3.2 Change of basis

Perhaps the most important transformation in the Alpha system is the **change of basis**. The intuition behind it is as follows. Since an Alpha variable can be viewed as a multidimensional array defined over a polyhedral domain, we should be able to "move" (or otherwise change the "shape" of) its domain and construct an equivalent program. For example, suppose we want to rotate the domain of `f` in Fig. 5.2 counterclockwise by 45°, i.e., we desire to map the point `[1,0]` to `[1,1]`, and `[1,1]` to `[0,1]` (this makes a vertical line diagonal, and a diagonal one horizontal). It is easy to verify that the mapping `(i,j -> i-j,i)` achieves this; the resulting program and transformed domain are shown in Fig. 5.3 (the changes are highlighted). In general, when variable `V` is transformed, the system must determine: (i) its new domain, (ii) the new case structure of its equation, (iii) the new dependencies for the uses of *all* variables in the equation for `V`, and (iv) the new dependencies for *uses* of `V`

```
            ⋮
  f : {i,j | 1<=i<=j; 2<=j<=N} of real;
let
  f[i,j] = case
        {| i=j}    : 0[];
        {| i<=j-1} : f[i+1,j] + A[j,-i+j] * X[-i+j];
           esac;
  X[i] = case
        {| 2<=i} : B[i] - f[1,i];
        {| i=1} : B[i] - 0[];
         esac;
tel;
```

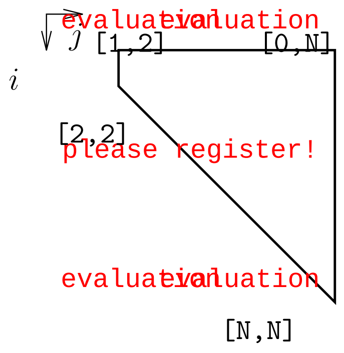


Figure 5.3: The transformed program and the new domain of f

(in all other equations). All of this is done automatically using the polyhedral library, and relies on the fact that the language allows a coherent interaction among the domains, the dependencies and the (normal form) structure of the programs.

The syntax for the affine mapping in a change of basis transformation is the same as for dependencies, but there is a subtle difference. We should be able to transform the variable in any (affine) manner, but we must always ensure that the *number* of index points remains invariant. For this, the affine change of basis must admit an integral inverse (i.e., be *unimodular*). We have also developed a *generalized change of basis*, which allows "non-square" mappings (non-unimodular transformations are encoded as a special case of this).

Chapter 6

Deriving systolic architectures using Alpha

Version of March 3, 2007

In this chapter, we describe a methodology for deriving systolic architectures using MmAlpha. This methodology is illustrated by means of an Alpha program for the matrix-vector multiplication. A companion notebook called `matvect.nb` allows the reader to execute interactively this demonstration. This notebook is situated in directory:
`$MMALPHA/demos/NOTEBOOKS/Matvect`
A link to this notebook is available in the master notebook (see **??**).

6.1 Introduction

The methodology presented here is inherited from the research of the "systolic community" [MC91, Kun82, QR89] and from its adaptation to Alpha [WS94, DL, LPR+96].

The successive steps of the derivation are the following ones:

1. High level specification in Alpha.
2. Uniformization.
3. Scheduling and mapping.
4. Control signal generation.
5. Generation of an AlpHard specification.

   These steps may be followed by three additional ones:

6. Translation to vhdl.
7. Logic synthesis.

8. Layout synthesis.

The five first transformations deal with producing a so-called AlpHard program. AlpHard is a subset of the Alpha language used for architectural description at the register transfer level. The description of AlpHard format can be found in [LPR+96], and is presented in chapter ??. In the present chapter, we only describe how to perform the first four above steps using MmAlpha. The result will be a single Alpha program which can be interpreted as a systolic array for executing the original Alpha specification. The main difference between this format – called Alpha0 – and AlpHard is their structuring: an AlpHard program is structured into several sub-systems while an Alpha0 program contains only one Alpha system.

6.2 Example of the matrix-vector product

We start by an Alpha description of the algorithm, as shown in figure 6.1.

Figure 6.1: Alpha program for the matrix-vector

The dependence graph of this program is shown in the left hand side of figure 6.2, for $N = M = 4$. In this figure, it can be seen that each component of the `b` vectors is broadcasted to several computations. For instance, b_1 is used in the computation of $a_{1,1}, a_{1,2}, a_{1,3}$ and $a_{1,4}$. The first step of the synthesis consists of "pipelining" these values in order to obtain a uniform dependence graph such as that shown in the right hand side of figure 6.2. This step is often called *uniformization* or *localization* in the litterature.

6.2.1 Uniformization

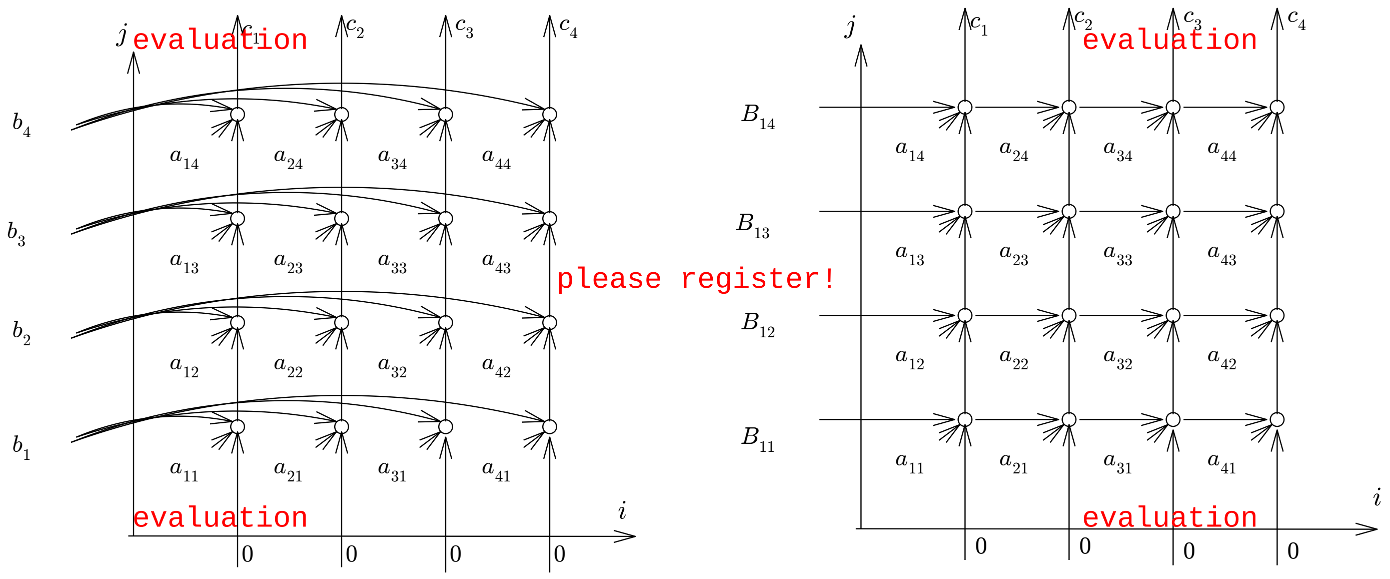
Methodologies for automatic uniformization were proposed by several researchers (see [QD88] for example). However, performing the uniformization process completely automatically is difficult as there are many possibilities for uniformizing a given program. Hence, this transformation requires some help from the user.

Before proceeding, we first perform a pre-processing transformation, which consists of adding a local variable `A` to replace the occurrence of `a` in the definition of `C`. The purpose of this modification is to anticipate a change of basis that will be done later on, and which cannot be applied on an input variable without altering the semantics of the program. The MmAlpha function which does this is:
```
addLocal["A = a"];
```

We are now ready to uniformize the occurence `b[i]` in the definition of `C`. The pipeline vector will be the direction of the $\vec{\imath}$ axis in our example, as shown in figure 6.2.

In our example, the MmAlpha function for uniformizing is:

```
pipeall["C","b.(i,j->j)","B1.(i,j->i+1,j)"];
```



j
c_1
c_2
c_3
c_4
b_4
b_3
b_2
b_1
a_{14} a_{24} a_{34} a_{44}
a_{13} a_{23} a_{33} a_{43}
a_{12} a_{22} a_{32} a_{42}
a_{11} a_{21} a_{31} a_{41}
i
0
B_{14}
B_{13}
B_{12}
B_{11}

Figure 6.2: Dependence graph for the program of figure 6.1, before and after uniformization.

The parameters given to `pipeall` can be read as follows:

> In the definition of `C`, pipeline the expression `b.(i,j->j)`. The pipeline vector is $(1, 0)$, the variable which will pipeline the value will be called `B1`.

Two remarks:

- The array notation `b[i]` cannot be used here, as it would be ambiguous.
- The pipeline vector is indicated as a translation in Alpha form.

The result of applying these commands is shown in figure 6.3 where only the equation of the resulting Alpha program are shown. Note that the domains of the branches of `B1` definition have been computed by `pipeall`. Note also that the dependence graph of this program – represented in the right hand side of figure 6.2 – is now uniform. For more information on the pipelining process refer to the documentation [?].

6.2.2 Scheduling and mapping

A schedule of this program is found by the `schedule` function. (see chapter 10 for more details.) Here we call this function with the option `scheduleType` in order to find out a schedule where all variables share the same linear part: this preserve the uniformity of the program. The MmAlpha command which does this is:

```
schedule[scheduleType->sameLinearPart];
```

```
B1[i,j] =
      case
        {| i=1; 1<=j<=M; 2<=N; 2<=M} : b[j];
        {| 2<=i<=N; 1<=j<=M; 2<=M} : B1[i-1,j];
      esac;
A[i,j] = a;
C[i,j] =
      case
        {| j=0} : 0[];
        {| 1<=j} : C[i,j-1] + A[i,j] * B1;
      esac;
```

Figure 6.3: Definitions of `B`, `A`, and `C` of program of figure 6.1 after uniformization

The result yields $T_C[i,j] = i + j$ and is illustrated in figure 6.4. The mapping must be given by the user. Following the standard systolic methodology, the allocation function a is a projection of the n-dimensional index space onto a $n-1$-dimensional space. Here, we assume that we want to project on the $\vec{\jmath}$ axis (`(i,j -> j)` (see figure 6.4). This results in a linear array of N processors.

6.2.3 Applying a spatio-temporal reindexing

We now have all the spatio-temporal information – timing, and spatial mapping – for obtaining a systolic array. We can make this information explicit in the Alpha program by performing a change of basis: in this way, in the new domains, the first index will represent the execution date and the second index will represent the processor number.

The function which does this transformation is:

```
applySchedule[];
renameIndices[{"t","p"}];
```

(The command `rename` allows one to change the names of the indices.) The resulting program is shown in figure 6.5. This program is scheduled and mapped: for example, `C[t,p]` will be computed at step `t` on processor `p`.

Note that `applySchedule` chooses the mapping of the program. It is possible to modify this mapping by applying another change of basis.[1]

6.2.4 Generation of an Alpha0 program

The generation of Alpha0 code consists of control signal generation, and simplification of the code. The control signal generation is necessary when the computation of a variable

[1]There is an incoherence here between the notebook and this chapter...

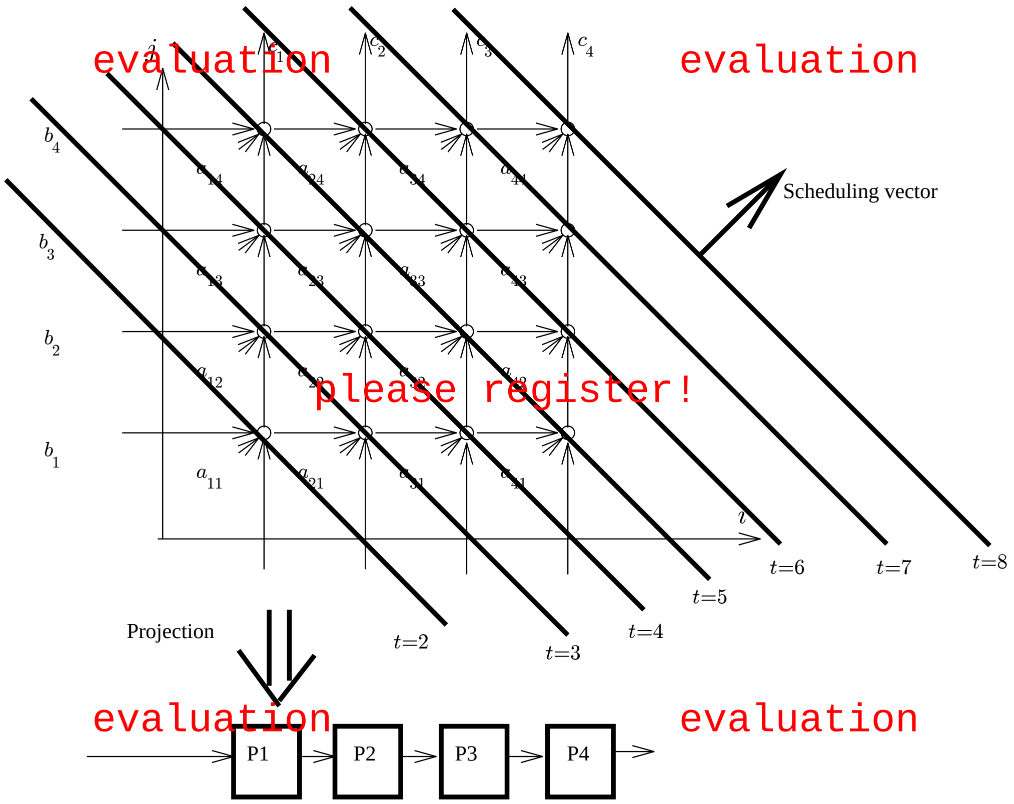


Figure 6.4: Scheduling and mapping of the uniform program

```
system MatVect :{N,M | 2<=N; 2<=M}
               (a : {i,j | 1<=i<=N; 1<=j<=M; 2<=N; 2<=M} of real;
                b : {i | 1<=i<=M; 2<=N; 2<=M} of real)
       returns (c : {i | 1<=i<=N; 2<=N; 2<=M} of real);
var
  B1 : {t,p | p+1<=t<=p+M; 1<=p<=N; 2<=N; 2<=M} of real;
  A : {t,p | p+1<=t<=p+M; 1<=p<=N; 2<=N; 2<=M} of real;
  C : {t,p | p<=t<=p+M; 1<=p<=N; 2<=N; 2<=M} of real;
let
  B1[t,p] =
      case
        {| 2<=t<=M+1; p=1; 2<=N; 2<=M} : b[t-p];
        {| p+1<=t<=p+M; 2<=p<=N; 2<=M} : B1[t-1,p-1];
      esac;
  A[t,p] = a[p,t-p];
  C[t,p] =
      case
        {| t=p; 2<=N; 2<=M} : 0[];
        {| p+1<=t; 2<=N; 2<=M} : C[t-1,p] + A[t,p] * B1[t,p];
      esac;
  c[i] = C[i+M,i];
tel;
```

Figure 6.5: Alpha program after change of basis

changes over execution time. For instance, consider variable C of the program of figure 6.5. When evaluating C[t,p], a processor p must know whether t=p of t>p, and this can only be done with the help of a control signal. In addition, this control signal should be inialized by a controller.

The situation is different for variable B1. There are two different definitions of B1[t,p] depending on whether p=1 or p>1: b[t-p] and B1[t-1,p-1]. But these conditions do not involve the time t. As a consequence, no control signal needs to be generated, as these conditions will be directly reflected in the structure of cell number 1, and cells number p, p>1.

```
C[t,p] =
    case
      {| p<=t<=p+M; 1<=p<=N; 2<=N; 2<=M} :
         case
           {| t=p; 1<=p<=N; 2<=N; 2<=M} : (if (loadC) then 0[] else True[]);
           {| p+1<=t<=p+M; 1<=p<=N; 2<=N; 2<=M} :
              (if (not loadC) then C[t-1,p] + A[t,p] * B1[t,p] else False[]);
         esac;
    esac;
loadC[t,p] =
    case
      {| p<=t<=p+M; 1<=p<=N; 2<=N; 2<=M} :
         case
           {| t=p; 1<=p<=N; 2<=N; 2<=M} : True[];
           {| p+1<=t<=p+M; 1<=p<=N; 2<=N; 2<=M} : False[];
         esac;
    esac;
```

Figure 6.6: Definition of C after space-time case separation, and control signal generation.

Control signal generation is performed by a single call to the Mathematica function toAlpha0v2:

```
toAlpha0v2[];
```

The Alpha0 program is shown in figure 6.7. It can directly be interpreted as the array of figure 6.8.

6.2.5 Obtaining an AlpHard specification

AlpHard is a subset of Alpha which represents hardware in a structured manner. AlpHard will be described precisely in section 12.1. From the Alpha0 specification, AlpHard code is obtained automatically by the command alpha0ToAlphard

```
system MatVect (a : {i,j | 1<=i<=4; 1<=j<=4} of real;
                b : {i | 1<=i<=4} of real)
       returns (c : {i | 1<=i<=4} of real);
var
  TSep5 : {t,p | p+1<=t<=p+4; 1<=p<=4} of real;
  TSep4 : {t,p | p+1<=t<=p+4; 1<=p<=4} of real;
  TSep3 : {t,p | p+1<=t<=p+4; 1<=p<=4} of real;
  TSep2 : {t,p | p+1<=t<=p+4; 1<=p<=4} of real;
  TSep1 : {t,p | p+1<=t<=p+4; 1<=p<=4} of real;
  B1 : {t,p | p+1<=t<=p+4; 1<=p<=4} of real;
  A : {t,p | p+1<=t<=p+4; 1<=p<=4} of real;
  C : {t,p | p<=t<=p+4; 1<=p<=4} of real;
  loadC : {t,p | p<=t<=p+4; 1<=p<=4} of boolean;
  B1reg1 : {t,p | p<=t<=p+3; 2<=p<=4} of real;
let
  B1reg1[t,p] = B1[t,p-1];
  B1[t,p] =
      case
        {| 2<=t<=5; p=1} : b[t-p];
        {| p+1<=t<=p+4; 2<=p<=4} : B1reg1[t-1,p];
      esac;
  A[t,p] = a[p,t-p];
  TSep1[t,p] = A[t,p];
  TSep2[t,p] = B1[t,p];
  TSep3[t,p] = C[t-1,p];
  TSep5[t,p] = TSep1 * TSep2;
  TSep4[t,p] = TSep3 + TSep5;
  C[t,p] =
      case
        {| p<=t<=p+4; 1<=p<=4} :
           case
             {| t=p; 1<=p<=4} : (if (loadC) then 0[] else True[]);
             {| p+1<=t<=p+4; 1<=p<=4} :
                (if (not loadC) then TSep4 else False[]);
           esac;
      esac;
  loadC[t,p] =
      case
        {| p<=t<=p+4; 1<=p<=4} :
           case
             {| t=p; 1<=p<=4} : True[];
             {| p+1<=t<=p+4; 1<=p<=4} : False[];
           esac;
      esac;
  c[i] = C[i+4,i];
tel;
```

Figure 6.7: Alpha0 program before translation in AlpHard, this program represent the array of figure 6.8

```
alpha0ToAlphard[controlVars[]];
```

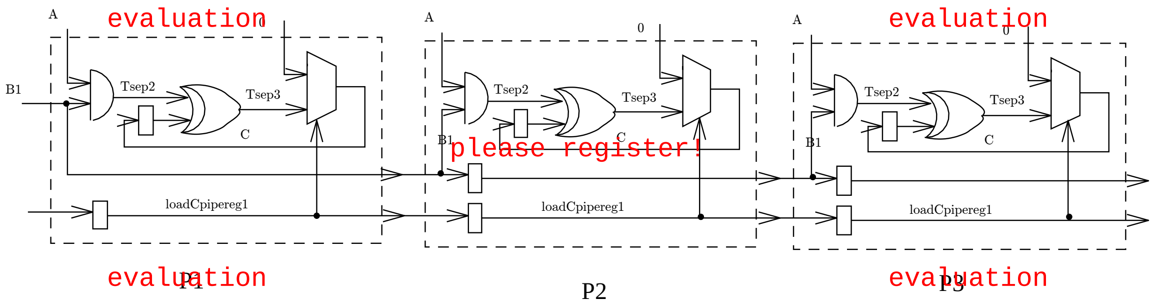


Figure 6.8: Array obtained after high level synthesis (here for $N = 3$)

6.3 Appendix: list of commands

```
addLocal["A = a"];
pipeall["C","b.(i,j->j)","B1.(i,j->i+1,j)"];
schedule[scheduleType->sameLinearPart];
applySchedule[];
renameIndices[{"t","p"}];
toAlpha0v2[];
alpha0ToAlphard[controlVars[]];
```

Chapter 7

Structured programming in Alpha

7.1 Introduction

This chapter presents generic and structured programming in Alpha.

It first introduces the notion of *size parameters* of an Alpha system: a size parameter is a special index, often noted in capital letters, used to give generic values to the size of the problem. For example, when talking of a NxN matrix product, "N" is a size parameter. In an Alpha system, such a parameter will appear in the domains and affine functions as an index which is global to all the domains of the system. The reader may already have noticed such size parameters in some of the previous examples, as their use is quite intuitive. This chapter will present them in greater detail.

The *program structures* in Alpha are then introduced: so far all the algorithms were simple enough to be described as a single Alpha system, but one may want to structure a complex problem into several smaller and simpler Alpha systems. The structure mechanisms presented here allow this.

7.2 Parameters

In most applications, it is useful to deal with *parameterized* algorithms. The polyhedral SARE model allows for a simple but powerful parameter scheme which is demonstrated in the following program, a simple matrix-vector product.

```
system matvect :{N | N>1}                                -- 1
               (M : {i,j | 1<=i,j<=N} of real;           -- 2
                V : {j | 1<=j<=N} of real)               -- 3
       returns (R : {i | 1<=i<=N} of real);              -- 4
var                                                      -- 5
  C : {i,j | 1<=i,j<=N} of real;                         -- 6
let                                                      -- 7
  C = case                                               -- 8
```

```
        {i,j | j=0} : 0.(i,j->);                      -- 9
        {i,j | j>0} : C.(i,j->i,j-1) + M * V.(i,j->j);  -- 10
      esac;                                           -- 11
  R = C.(i->i,N);                                     -- 12
tel;                                                  -- 13
```

In this system, N is a *size parameter* which is declared in the header of the system using a *parameter domain*, here {N |N>1}. This parameter domain may be any Alpha domain, in particular there is no limit on the number of parameters, and this domain may express any affine inequation between the parameters. For example, {M,N| M<2N} is a valid parameter domain.

Such a parameter may then appear anywhere in the system where indices are allowed, that is, in domains (see lines 2-4 of the previous program) and in affine functions (see line 12).

This parameterization retains all the properties of the language, for parameters are actually special indices which are shared by all the variables and subexpressions of a system. For example the parameterized domain of the variable M, defined line 2 as {i,j| 1<=i,j<=N}, actually denotes the domain {i,j,N| 1<=i,j<=N}. Similarly, the parameterized affine function (i->i,N) of line 12 actually represents the closed function (i,N->i,N,N). Thus a parameterized system is nothing but a usual SARE where all the objects share the parameter indices. Actually the system manipulated by MmAlpha is the following:

```
system matvect                                                  -- 1
              (M : {i,j,N | 1<=i,j<=N} of real;                 -- 2
               V : {j,N | 1<=j<=N} of real)                     -- 3
       returns (R : {i,N | 1<=i<=N} of real);                   -- 4
var                                                             -- 5
  C : {i,j,N | 1<=i,j<=N} of real;                              -- 6
let                                                             -- 7
  C = case                                                      -- 8
        {i,j,N | j=0} : 0.(i,j,N->);                            -- 9
        {i,j,N | j>0} : C.(i,j,N->i,j-1,N) + M * V.(i,j,N->j,N);  -- 10
      esac;                                                     -- 11
  R = C.(i,N->i,N,N);                                           -- 12
tel;                                                            -- 13
```

7.3 Structures

The issue of structuring a complex algorithm into a hierarchy of SAREs is more complex than it seems. Obviously, it is partly addressed by the decomposition of the problem into equations: it is always possible to break an equation into two simpler equations, using an auxilliary variable to hold a subexpression of the initial expression. The reverse operation, replacing a variable appearing in an expression with its definition, is also always possible.

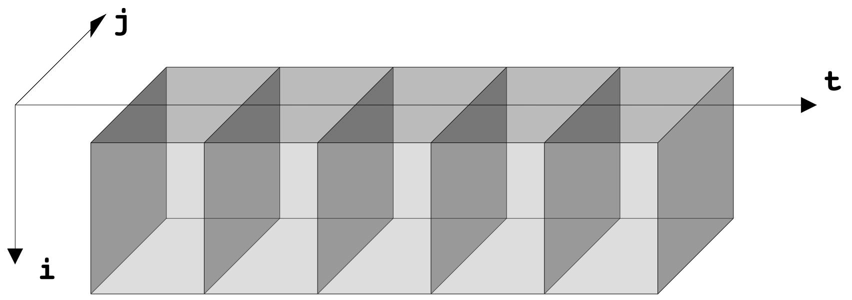


Figure 7.1: Linear collection of bidimensional matrices

The MmAlpha environment provides assistance for both these operations, thus ensuring that the resulting system is equivalent to the initial one[1].

Structuring using equations, however, basically remains first-order structuring (in the usual functional meaning), and is thus limited. These limits appear more clearly on the following example: suppose we want to write in Alpha an algorithm for an image processing application, which involves time-varying pixel matrices. Such matrices are represented in Alpha as three-dimensional data arrays (two matrix dimensions, and one – possibly unbounded – time dimension) as represented on Fig.7.1.

Now let us try to re-use the equations of the matrix-vector product to operate on these time-varying matrices. Obviously it is not possible in a straightforward manner, for the inputs don't have the proper dimension. For example we will need to rewrite the whole of the equation defining `C` to add one dimension to the domains and the affine functions, yielding the following equation:

```
C = case                                                      -- 8
      {i,j,t | j=0}: 0.(i,j,t->);                             -- 9
      {i,j,t | j>0}: C.(i,j,t->i,j-1,t) + M * V.(i,j,t->j,t); -- 10
    esac;                                                     -- 11
```

Structuring with variable will never be adequate when such a *dimension extension* is needed, which is the usual case. The remainder of this chapter presents the Alpha program structures addressing this problem.

7.4 Simple structures

Let us try and write an Alpha SARE for the addition of two integers (or fixed-point numbers) written in binary notation. Such a binary addition is classically described as a sequence of *full adder* operations with the propagation of a carry bit from one full adder to the following, as described by Fig.7.2.

The following system is a *full adder* function written in Alpha.

[1]These operations are similar to β-conversion in the lambda-calculus.

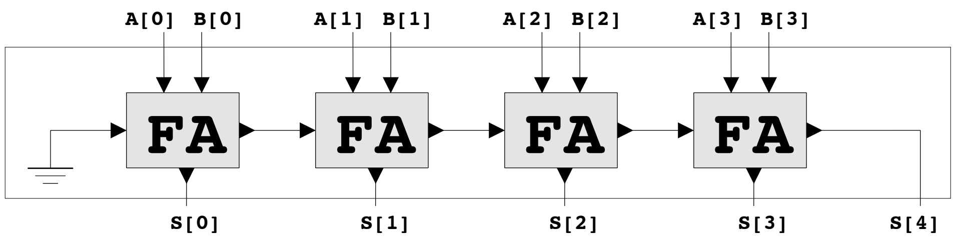


Figure 7.2: Addition of integers coded in boolean, using *full adders*

```
system FullAdder (A,B,Cin : boolean)
         returns (X,Cout : boolean);
let
  X = A xor B xor Cin;
  Cout = (A and B) or (A and Cin) or (B and Cin);
tel;
```

To build an adder system with this program, we need to be able to express that we have a linear collection of instances of this system, as shown in Fig.7.2. The shape of this linear collection may be expressed as an ALPHA domain, say `{b|0<=b<W}` where `W` is a size parameter giving the number of bits.

The structure construct **use** in ALPHA allows precisely that: the following system describes in ALPHA the adder given Fig.7.2:

```
system Plus: {W|W>1} (A,B: {b| 0<=b<W} of boolean)          -- 1
             returns (S : {b| 0<=b<=W} of boolean);          -- 2
var                                                          -- 3
  Cin, Cout, X : {b| 0<=b<W} of boolean;                     -- 4
let                                                          -- 5
  use {b| 0<=b<W} FullAdder[] (A,B,Cin) returns(X, Cout);   -- 6
  Cin[b] =                                                   -- 7
    case                                                     -- 8
      {| b=0} : 0[];                                         -- 9
      {| b>0} : Cout[b-1];                                   -- 10
    esac;                                                    -- 11
  S[b] =                                                     -- 12
    case                                                     -- 13
      {| b<W} : X;                                           -- 14
      {| b=W} : Cout[W-1];                                   -- 15
    esac;                                                    -- 16
tel;                                                         -- 17
```

In this system, the line 6 reads as follows: "Use a collection of instances of the subsystem `FullAdder`. This collection has the shape of the extension domain `{b|0<=b<W}` and is thus

indexed by index `b`. Let the inputs of the `b`-th instance be the variables `A`, `B` and `Cin` at point `b`, and similarly let the outputs of this collection of instances be the variables `X` and `Cout`." (The lines 7-11 describe the carry propagation, and lines 12-16 define the output of this binary adder.)

In other words, line 11 is a shortcut for the following equations, which are those of the system `FullAdder` whith the dimension of the variables extended from zero to one:

```
X[b] = A[b] xor B[b] xor Cin[b];
Cout[b] = (A[b] and B[b]) or (A[b] and Cin[b]) or (B[b] and Cin[b]);
```

7.5 Syntax of the *use* construct

The **use** construct appears at the syntactic level of an equation, since it is basically a shortcut for a set of equations. Here is the general syntax of an equation/use:

```
Equation ::
      Identifier = Expression ;
   |  use [  ExtensionDomain ] Identifier
                [ [ParamAssignment ] ]
                ( ExpressionList )
         returns  ( IdentifierList ) ;
```

In this syntax we see that there is an optional *parameter assignment* which is discussed in the following. In the previous addition, the subsystem `FullAdder` has no parameters, and the parameter assignment is therefore empty.

7.6 Binary multiplication in Alpha

When performed by hand, a multiplication is basically a collection of additions, as shown by Fig.7.3.

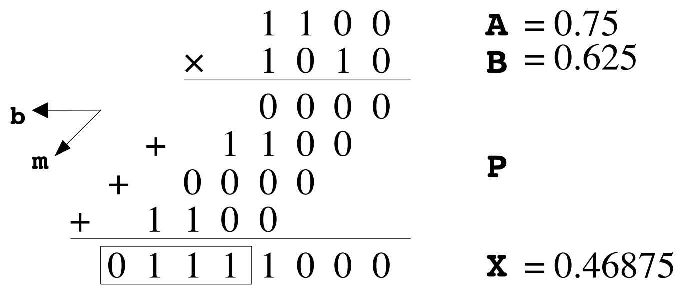


Figure 7.3: Product of two fixed point reals in binary representation

The following program is the Alpha incarnation of Fig.7.3. Line 8 performs the binary product of all the bits of the first operand by each bit of the second. Line 10 describes a linear collection of additions, indexed by `m` which is the row index in Fig.7.3. Lines 12-17 link the result of one additions to the input of the following.

```
system Times: {W|W>2} (A,B: {b| 0<=b<W} of boolean)   -- 1
              returns (X  : {b| 0<=b<W} of boolean);  -- 2
var                                                   -- 3
  P : {b,m| 0<=b,m<W } of boolean;                    -- 4
  Si : {b,m| 0<=b<W; 0<m<W } of boolean;              -- 5
  So : {b,m| 0<=b<=W; 0<m<W } of boolean;             -- 6
let                                                   -- 7
  P[b,m] = A[b] and B[m];                             -- 8
  use {m| 0<m<W} Plus[W] (Si,P) returns (So);         -- 9
                                                      -- 10
  Si[b,m] =                                           -- 12
                                                      -- 11
    case                                              -- 13
      {| m=1} : P[b,m-1];                             -- 14
      {| m>1} : So[b+1,m-1];                          -- 15
    esac;                                             -- 16
  X[b] = So[b+1,W-1];                                 -- 17
tel;                                                  -- 18
```

Here the parameter assignment `W` just equates the bit size parameter of the subsystem `Plus` and that of the multiplication. In the general case, however, this parameter assignment may be any affine function, which proves very powerful. Figure 7.4 describes the problem of accumulating binary numbers without overflow: the sum of two N-bits numbers may be a $N+1$ bits number, therefore we need to use additions of increasing sizes to avoid loss of bits. This is expressed in Alpha as:

```
use {i| 1<=i<N} Plus[W+i-1] (A1,A2) returns (Acc);
```

Another example where the natural structuring of an algorithm makes use of a parameter assignment depending on the extension indices is the Gaussian elimination given at the beginning of chapter 8. The reader is invited to try and understand this program.

Note that in all our examples, the extension domain is monodimensional, but this is not a rule: in the general case the extension domain may be any arbitrary Alpha domain. For example, in an image processing application, a 2-dimensional extension domain may be used to apply some function to all the points of an image.

7.7 Manipulating structured programs

A structured program is stored in MmAlpha as a Mathematica list of systems called a *library*. The default library is stored in the global variable `$library`.

A structured program may be written in one single file or several distinct files. In the latter case the `load[]` function returns a library composed of all the systems contained in the file, and stores this library in `$library`.

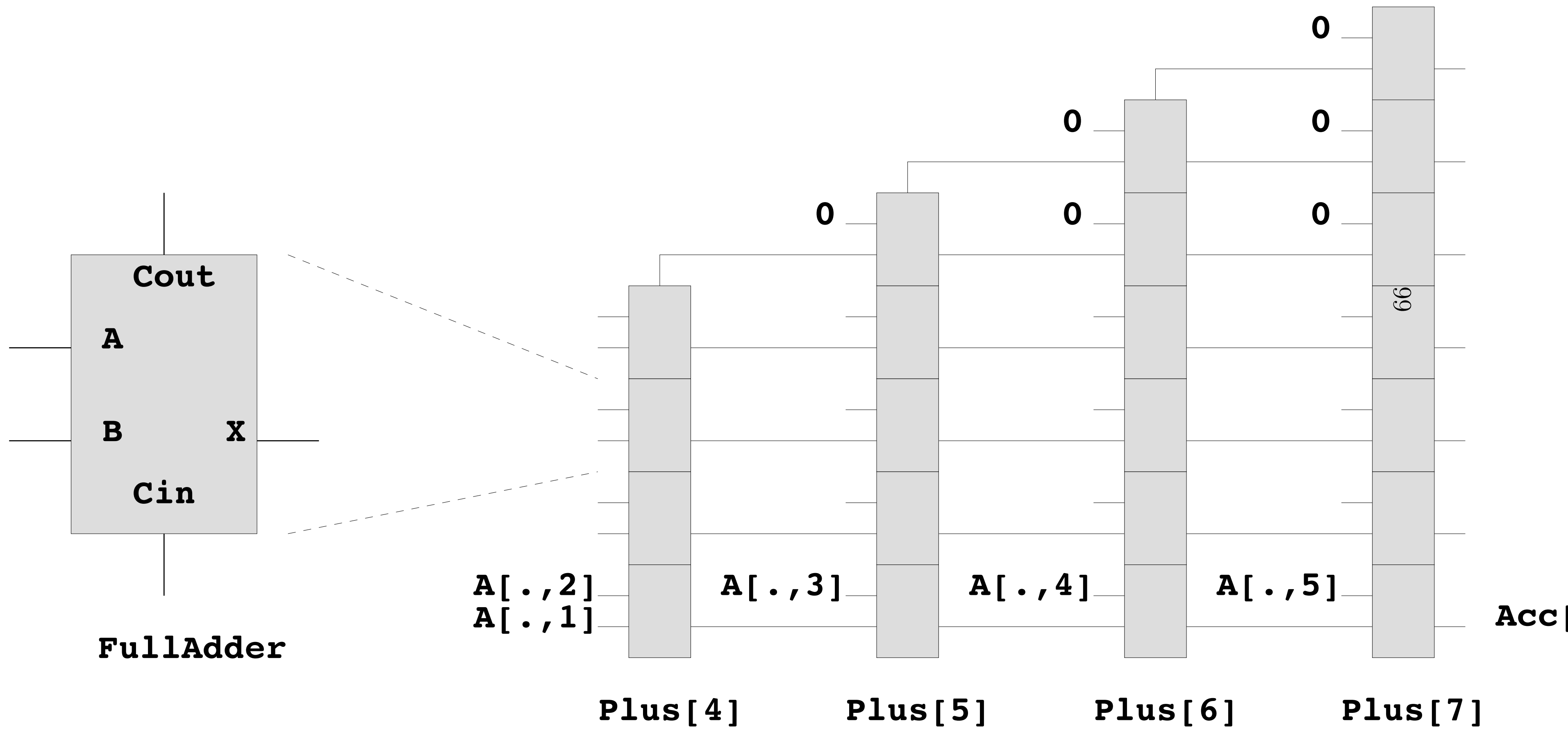
If the program is stored in several files, it is the responsibility of the user to build a proper library, i.e. a Mathematica list of all the systems needed by the hierarchical structure of the program. For this purpose, the user will typically use Mathematica list manipulating functions such as `Join[], Append[]`...

In addition, two functions, `putSystem[]` and `getSystem[]`, may be used to extract a system from a library and to put back a modified system in a library. Typically one of the system is extracted from the library, modified by some program transformation, and then put back in the library.

7.8 Program transformations associated with structures

Most MmAlpha functions handle parameterized programs and *use* statements. There are, however, some major exceptions such as the `writeC[]` translator. In this case, MmAlpha provides program transformations transforming a structured program into a simpler equivalent one :

- `assignParameterValue[]` gives a value to a size parameter, i.e. it refines a generic system into a specialized one.
- `inlineSubSystem[]` expands a *use* statement, replacing it with the equations of the corresponding subsystem, properly modified to take the dimension extension into account.
- `inlineAll[]` recursively flattens a structured Alpha program.


Cout
A
B
X
Cin
FullAdder
0
0
0
0
0
0
A[.,2]
A[.,1]
A[.,3]
A[.,4]
A[.,5]
Acc[
Plus[4]
Plus[5]
Plus[6]
Plus[7]

Chapter 8

Static analysis of Alpha programs

8.1 Introduction

This chapter describes the use of the static analysis tool `analyze[]`. It is divided in two parts: the first part deals with the static analysis of a single system, and the second describes the analysis of a structured program consisting of several systems. Any beginner in Alpha should read the first part, while the second part is left to more experienced users.

8.1.1 What is static analysis?

It is impossible to ensure that an Alpha system computes the expected result (to start with, the termination of such computation is well known to be indecidable). However there are a few necessary conditions for a system to be valid which may be verified *statically*, that is

- independently of any set of input values that may be fed to the system, and
- in a manner that is valid for any of these set of values.

8.1.2 What does the static analyzer do?

The Alpha static analyzer basically verifies the following rule:

For each point of the domain of a variable, there is one and only one computation defining the value of the variable at this point.

The static analyzer checks that this rule is ensured and outputs error messages giving the points where a variable is over- or under-defined.

8.1.3 When should the static analyzer be used?

This tool is very useful while writing and debugging Alpha code, and should be invoked systematically. Besides, *Alpha program transformations are only guaranteed to work on programs that pass the static analysis without error messages.*

This is in particularly true of the Alpha to C translator `writeC[]`: the C code generated by this command is likely to cause run-time errors if the initial Alpha program doesn't pass the static analysis. Therefore the static analyzer should be called before any simulation of the Alpha program.

8.1.4 An example

We will use in this chapter the following two-system example program which transforms a matrix into triangular form as first step of a Gaussian elimination[1].

The first system takes a square matrix and zeroes all the elements below the `K`-th diagonal of this matrix (one slice of Fig.8.1.4):

```
system ZeroColumn: {N,K| 1<=K<N} (A: {i,j| 1<=i,j<=N} of real)
                         returns (Ar: {i,j| 1<=i,j<=N} of real);
let
  Ar[i,j] = case
            {| i<=K}      : A[i,j];
            {| i>K; j<=K} : 0[];
            {| i>K; j>K}  : A[i,j] - A[K,j]*A[i,K]/A[K,K];
            esac;
tel;
```

The second system uses N instances of the first to compute the triangular matrix (see Fig.refgauss-slices):

```
system Gauss: {N | N>1} (A: {i,j | 1<=i,j<=N} of real)
                returns (T: {i,j | 1<=i,j<=N} of real);
var Ak : {i,j,k| 1<=i,j<=N; 1<=k<=N} of real;
    Ak1: {i,j,k| 1<=i,j<=N; 1<=k<N} of real;
let
  use {k| 1<=k<N} ZeroColumn[N,k] (Ak) returns (Ak1);
  Ak[i,j,k] = case
              {| k=1} : A[i,j];
              {| k>1} : Ak1[i,j,k-1];
              esac;
  T[i,j] = Ak[i,j,N];
tel;
```

What kind of information does the static analyzer give? Suppose a typo has replaced an `j` index with a `i` index, leading to the following `ZeroColumn` program:

[1]This program is neither optimal nor complete, it was written for the purpose of demonstrating the analyze[] function. Writing a complete Gaussian elimination is left as an exercise to the reader.

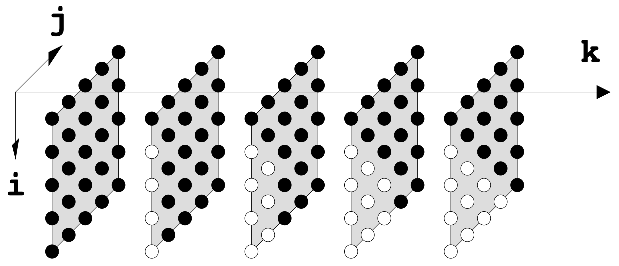


Figure 8.1: Triangularization in two systems

```
 system ZeroColumn: {N,K| 1<=K<N} (A: {i,j| 1<=i,j<=N} of real)
                          returns (Ar: {i,j| 1<=i,j<=N} of real);
 let
   Ar[i,j] = case
             {| i<=K}      : A[i,j];
             {| i>K; i<=K} : 0[];     -- The typo is hidden here
             {| i>K; j>K}  : A[i,j] - A[K,j]*A[i,K]/A[K,K];
             esac;
 tel;
```

This error can't be detected by a parser program – this program is syntactically correct. However, invoking the static analysis will yield the following messages:

```
In[5]:= analyze[];

WARNING: This expression has an empty domain :
{i,j | i=0; j=0; N=0; K=0; 1=0} : 0.(i,j->)

ERROR: Variable Ar not defined over the domain :
{i,j | K+1<=i<=N; 1<=j<=K}

*** Analysis failed ***
```

A warning and an error tell us that there is an error in the equation defining `Ar`. Several *error domains* may help us pinpoint precisely where the error is. The first warning message is enough to spot that the error is in the `case` subexpression containing the `0[]`. The `1=0` equation in the domain indicates that this domain is empty.

As this example shows, it may take a certain amount of experience to fully understand the error messages: the error domains are not always in the most readable form. However the indications given usually allow to spot the problem precisely: in our example the combination of both previous messages points exactly to the cause of the error.

The remainder of this paper describes the static analysis tool in more details.

8.2 Static analysis of an Alpha system

8.2.1 The domain of an expression

In Alpha every variable is declared with a polyhedral domain defining the set where it is expected to contain a value.

When an expression is built using such variables, this expression inherits the domain of these variables: for example if `A` and `B` are two expressions defined over some square domain `{i,j|0<i,j<10}`, then their sum `A+B` is defined everywhere both `A` and `B` are defined, that is on the same square domain.

Now if the domains of `A` and `B` are different, then the sum is still defined everywhere both `A` and `B` are defined, that is on the intersection of the domains of `A` and `B`.

It is possible to carry these ideas further, and thus to define the domain of any Alpha expression, knowing the domains of the subexpressions (see the discussion of section 3.5, page 31). The rules to apply, given below, are actually part of the definition of the semantics of the language. They rely only on operators preserving the set of Alpha domains and thus may be computed automatically: this is what the static analyzer does.

Constants	$\boxed{Dom(c) = \mathbf{Z}^0}$
Variables	$\boxed{Dom(V) \text{ is declared in the header}}$
Unary operators	$\boxed{Dom(-e) = Dom(e)}$
Binary operators	$\boxed{Dom(e_1 + e_2) = Dom(e_1) \cap Dom(e_2)}$ The sum of two variables is defined where both variables are defined
Ternary operators	$\boxed{Dom(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) = \bigcap_{i=1}^{3} Dom(e_i)}$ The `if then else` is considered as a ternary operator and is defined where the condition and both alternatives are defined
Restriction D is a domain	$\boxed{Dom(D : e) = D \cap Dom(e)}$ This operator restricts an expression to D
Affine dependency f is an affine function of the indices of the LHS	$\boxed{Dom(e[f]) = f^{-1}(Dom(e))}$ The value of $e[f]$ at point $\mathbf{z}$ is the value of e at point $f(\mathbf{z})$, hence the domain of $e[f]$
`case` operator	$\boxed{Dom(\texttt{case} e_1; ..; e_n; \texttt{esac}) = \bigcup_{i=1}^{n} Dom(e_i)}$ The `case` operator allows the piecewise definition of an expression by several subexpressions e_i with disjoint domains.

The function `expDomain[]` may be used to compute the domain of any Alpha expression, following these rules.

Now during the computation of the domain of an expression, the `case` operator introduces the possibility of having more than one subexpression define the value of the same point of the domain. Therefore the analysis tool has to check that the intersections of the domains of the case subexpressions are empty. It computes this intersection, and if it not empty it issues an error message as in the following example:

Example: in the equation defining `Ar`, if the second line of the `case` was wrongly written:

```
    {| i>=K; j<=K}: 0[];
```

The analysis tool will output the following message:

```
ERROR: in case statement: ...,
       domains of subexpressions overlap on:
       {i,j | i=K; 1<=j<=K}
```

There are other useful informations which the static analyzer provides. For example it is useful to detect an empty expression i.e. an expression with an empty domain: such an expression is in the best case pointless (in a case statement), and may be a source of errors. To avoid cascaded error messages, only the deepest empty subexpression is reported to the user.

Example: Still in the same equation, a mistake in the first `case` subexpression:

```
    {| i<=0}     : A[i,j];
```

will cause the following messages:

```
WARNING: This expression has an empty domain:
         {| i<=0} : A[i,j]
ERROR:   Variable Ar not defined over the domain:
         {i,j| 1<=i<=K; 1<=j<=N}
```

8.2.2 Equation analysis

Now we describe how the static analyzer works: it considers each equation $V[i,j\ldots] = e$, where e is an ALPHA expression. To ensure that there is at least one computation defining the value of V for each point $(i,j,\ldots)$ of its domain, the analysis tool computes $D' = Dom(V) \setminus Dom(e)$, where $\setminus$ denotes the set difference, $Dom(V)$ is the domain of the variable V (declared in the header of the system), and $Dom(e)$ is the domain of the expression as defined above.

If D' is non empty, an error is issued, stating that V is not defined over D'. This error domain will be useful to the user to spot the problem.

Example: In the equation of the system `ZeroColumn` defining `Ar`, suppose the first line of the `case` statement was mistyped:

```
    {| i<K}     : A[i,j];    -- instead of {| i<=K} ...
```

(notice the `<K` instead of `<=K`). The analysis tool will output the following message:

```
ERROR: Variable Ar not defined over the domain:
       {i,j| i=K; 1<=j<=N}
```

8.2.3 Parameter related analysis

If the system considered is parameterized, the static analysis process may be usefully refined by taking the parameters into account. Consider again the `Gauss` system:

```
system Gauss: {N | N>1} (A: {i,j | 1<=i,j<=N} of real)
                returns (T: {i,j | 1<=i,j<=N} of real);
var Ak : {i,j,k| 1<=i,j<=N; 1<=k<=N} of real;
    Ak1: {i,j,k| 1<=i,j<=N; 1<=k<N} of real;
let
  use {k| 1<=k<N} ZeroColumn[N,k] (Ak) returns (Ak1);
  Ak[i,j,k] = case
              {| k=1} : A[i,j];
              {| k>1} : Ak1[i,j,k-1];
              esac;
  T[i,j] = Ak[i,j,N];
tel;
```

Suppose there was no restriction on the parameter `N` of the system `Gauss`. Its header would be: `system Gauss: {N |}` . Now obviously for negative values of `N`, all the variables of this system have an empty domain, which should be pointed to the user as a possible source of errors. The static analyzer performs such parameter-related checks.

Example: We may restrict the parameter `N` of the system `Gauss` to be positive:

```
system Gauss: {N | N>0}
```

One may check that the system is still valid, even for `N=0`. However the analysis will issue the following message:

```
WARNING: for parameters {N| N=1}, the expression Ak1 has an empty domain
```

It is good programming practice to ensure that the system is valid for all the values of its parameter domain. It becomes mandatory if the program is composed of several systems, as shown in the following section.

8.3 Analysis of structured programs

8.3.1 Analysis of `use` statements

The analysis of a `use` statement is deduced from its *substitution semantics*: in short, a program containing a `use` statement is (by definition of the use) equivalent to one where this statement has been replaced with the body of the subsystem (properly modified to take into account the extra dimensions and the affine parameter assignment) and additional equations to perform the I/O passing: input equations relate the actual inputs and the formal ones :

```
      SubSystemInputVariable = ActualInputExpression ;
```

and output equations relate the actual outputs and the formal ones :

```
        ActualOutputVariable = SubSystemOutputVariable ;
```

Global checks The first validity conditions are that the subsystem has been declared somewhere in the program, that the correct number of actual inputs/outputs are given, and that their respective dimensions match the formal ones. The tool also checks, using the same techniques as previously, that the extension domain is non-empty for all the values of the parameters.

Then we consider the values given to the parameters of the subsystem by the caller. The parameters of the subsystem are an affine function of the caller parameter and, possibly, the extension indices. The analyzer checks that, for all the possible values of the caller parameters, and for all the points in the extension domain, the values assigned to the subsystem parameters fall within the range permitted, i.e. within the parameter domain of the subsystem. Otherwise an error message is issued, showing a domain which is the set of points where parameters are given but not expected.

Example: In the `Gauss` system, the following `use` statement:

```
    use {k| 0<=k<=N} ZeroColumn[N,k] (Ak) returns (Ak1);
```

will cause the following message:

```
 ERROR : in statement ‘‘use ... ZeroColumn...’’,
         parameter values in {N,K| K=0, N>=1} not allowed.
```

Obviously, the more restricted the parameter domain of the subsystem, the more acurate the checks performed here.

Input/Output checks The substitution semantics implies that the verifications to be performed on the I/Os of a subsystem `use` may be deduced from those of the equations described in 8.2.2. Error messages are given accordingly.

8.3.2 Global analysis of a library of systems

The validity of a `use` statement is then implied by the validity of the subsystem on its parameter domain, the condition that all the parameter values assigned by a `use` are allowed, and the validity of the virtual I/O passing equations.

We may thus describe the general down-top verification method for a complete program. Such a program is an acyclic graph of Alpha systems using each other (system `use` can not be mutually recursive). Systems without a `use` statement in their equations are called *leaves*. A system using a subsystem is called a *parent* of this subsystem.

- First, the leaves are analyzed, and their parameter domain is restricted as much as possible. No warning message should remain. For example, for the system `ZeroColumn`, we have to restrict `K` and `N` at least to the domain given in the correct version of this sytem (it is possible to constraint the parameters more than what the analysis suggests. For example we could put a bound on `N` depending on the application aimed at).
- Then the parents of the leaves are analyzed. If they are written to use a leaf with illegal values of its parameters, the tool will spot it and the programmer will be invited either to correct the error, or to restrict more the caller parameters. Meanwhile, the other equations of the caller are also analyzed, with the same effect.
- This process is repeated on the parents of the parents, and so on until the whole program passes the static analysis.

Note that one of the options to `analyze[]` decides whether the analysis is performed recursively on all the subsystems of the system currently being analyzed, or only to this system.

Chapter 9

Code generation and simulation of Alpha programs

In this chapter, we explain how to generate C code from an Alpha program. As seen in Chapter 5, Alpha programs are functional, and therefore, do not convey any execution ordering. However, the validation of Alpha programs, or even, the generation of code for a general-purpose or dedicated processor, requires the possibility to simulate the evaluation of Alpha expressions.

This can be done using the `writeC` MmAlpha command, as seen here.

An example

Consider the following Alpha program: To simulate this program, we first load this program,

```
system adder   (x: integer;
                y: integer)
       returns (z: integer);
let
  z = x + y;
tel;
```

Figure 9.1: A very simple example

then generate a C program in file "adder.c", using

```
load["adder-WriteC.alpha"];  (* load file *)
writeC["adder.c"];
```

Then we can compile this program:

```
cc -o adder adder.c
```

The execution of this program is shown here:

```
adder
Input x =2
x = 2
Input y =3
y = 3
z = 5
```

The C code prompts for the value of the input variables `x` and `y`.

9.1 Program with parameters

Programs with parameters cannot be simulated without giving a value to the parameters. This is obtained through an option of `writeC`.

```
system adder: {N|N>=1} (x: {t|1<=t<=N} of integer;
                y: {t|1<=t<=N} of integer)
       returns (z: {t|1<=t<=N} of integer);
let
  z = x + y;
tel;
```

Figure 9.2: A very simple example

```
load["adder-WriteC1.alpha"];  (* load program *)
writeC["adder1.c","-p 3"];
cc -o adder1 adder1.c
adder1Input x[1] =2
x[1]= 2
Input x[2] =3
x[2]= 3
Input x[3] =4
x[3]= 4
Input y[1] =-2
y[1]= -2
Input y[2] =-3
y[2]= -3
Input y[3] =-4
y[3]= -4
z[1]= 0
z[2]= 0
z[3]= 0
```

9.1.1 Additional examples

```
SetDirectory["/home/frodon/d01/api/quinton/alpha/sobel"];
load["sobel.a"];ashow[];
analyze[];
writeC["-p 5 35 3"];
schedule[{objFunction -> 1,addConstraints -> {"Tpixel[i,N,M,p]=i"},duration -> 2}]
```

Chapter 10

Scheduling Alpha programs

Version of March 3, 2007

This chapter explains how to use the scheduler for finding valid execution order of computations of Alpha programs.

10.1 Introduction

An Alpha program does not convey any sequential ordering: an execution order is semantically valid provided it respects the data dependencies of the program. Consider for instance the program of figure 10.1 which represents the computation of the product of a $M \times N$ matrix `a` by a M vector `b`. To execute such a program on a sequential machine, we must chose a computation order which respects data dependencies. Such an order is shown in figure 10.2-(a). Another possible order would be the demand-driven execution scheme used in the simulation, as shown in chapter 9 : to evaluate a variable, evaluate recursively the expressions that this variable depends on. Parallel executions are also possible. The program of figure 10.2-(b) shows a possible parallel order for computing the matrix-vector multiplication.

Figure 10.1: Alpha program for the matrix-vector product, no execution order is specified.

The basic goal of the scheduler is to find out valid execution orders, called *schedules* in what follows. A secondary goal is to find out *good* schedules. However, there is no best schedule: the quality of a schedule depends on a particular criterion.

In our model, the time is considered as a discrete single clock and we look for parallel ordering of the computations. The theoretical basis for the scheduling process is inherited from the research on systolic array and automatic parallelization. The scheduler implements two different procedures: one, called the Farkas method, is defined in [?]; the other one, called the vertex method, is presented in [?].

The scheduler of Alpha is an analysis tool: it does not modify the current program in `$result`. The result is the description of a possible parallel schedule. For instance, for the

```
For i=1 to N
   C[i,0]=0
EndFor
For i=1 to N
   For j=1 to M
       C[i,j]=C[i,j-1]+
                 a[i,j] * b[j]
   EndFor
EndFor
For i=1 to N
   c[i]=C[i,M]
EndFor

            (a)
```

```
ForAll i=1 to N
   C[i,0]=0
EndFor
For j=1 to M
   ForAll i=1 to N
       C[i,j]=C[i,j-1]+
                    a[i,j] * b[j]
   EndForAll
EndFor
ForAll i=1 to N
   c[i]=C[i,M]
EndForAll

            (b)
```

Figure 10.2: Two possible order of execution for the program of figure 10.1. The left one is parallel. (`ForAll` represents a parallel loop).

program of figure 10.1, if we assume that the inputs are available at time 0 and that each assignement takes 1 clock tick, we will obtain the information that, for all `i`, value `C[i,j]` can be computed at step `j`. Thus, for all `i`, the values `c[i]` can be computed at step `M+1`. This schedule can be summarized by:

$$\begin{array}{l} T_a[i,j] = 0 \\ T_b[i] = 0 \\ T_C[i,j] = j \\ T_c[i] = M+1 \end{array} \tag{10.1}$$

This particular schedule corresponds exactly to the parallel execution order of the program of figure 10.2-(b).

In the present chapter, we are interested in *affine by variable* schedules, i.e schedules in which, for each variable, the execution date is an affine function of the indices (and possibly of the structural parameters). The idea of the scheduling process is to gather all the constraints that the schedule must verify in a linear programming problem (LP) and to solve this LP using a LP-solver.

10.2 Basic Farkas scheduler

The scheduling process is quite complex, since many parameters can influence the result. In this section, we present the basics of the scheduler.

10.2.1 How to use the `schedule` function

The function to be called in order to schedule a program is `schedule`. Its effects is to find out a schedule for the ALPHA program contained in `$result`, and to put the schedule in the global MATHEMATICA variable `$schedule`. Options allow one to modify parameters, hence the possible forms of a call to the function `schedule` listed here:

- `schedule[]`
  finds an affine-by-variable schedule for `$result` and assigns it to `$schedule` (this schedule attempts to minimize the global execution time).

- `schedule[sys]`
  finds an affine-by-variable schedule for the ALPHA system `sys` and assigns it to `$schedule`

- `schedule[option_1->value_1,...,option_n->value_n]`
  finds a schedule for `$result` which respects the chosen options and assigns it to `$schedule`

- `schedule[sys,option_1->value_1,...,option_n->value_n]`
  finds a schedule for the ALPHA system `sys` which respects the chosen options and assigns it to `$schedule`

For example, calling the scheduler on the matrix-vector multiplication of figure 10.1 prints out the following informations:

```
In[30]:= schedule[];
Dependence analysis...
Duration : 1 for each equation
Building LP...
LP: 82 Constraints
Writing file for PIP....
Solving the LP...
Version C.3 MultiPrecision (mpip, rev. 1.3.0)
cross : 407646, alloc : 1
Max cross-product result: 4 (1 digits, base 10)
Max numerator term: 8 (2 digits, base 10)
Max denominator: 2 (1 digits, base 10)
n 1 u 130''' s 11'''
T_a{i, j, N, M} = 0
T_b{i, N, M} = 0
T_c{i, N, M} = 1 + M
T_C{i, j, N, M} = j

Out[30]= scheduleResult[1, {{a, {i, j, N, M}, sched[{0, 0, 0, 0}, 0]},
   {b, {i, N, M}, sched[{0, 0, 0}, 0]}, {c, {i, N, M}, sched[{0, 0, 1}, 1]},
   {C, {i, j, N, M}, sched[{0, 1, 0, 0}, 0]}}]
```

The first lines indicate which computations are performed. The Farkas scheduler first built a LP, then writes this LP in a file, then calls a LP-solver. Writing this file takes some time, thus informations are displayed for the user not to become anxious.

The lines starting by `Version` are output by the LP-solver PiP. The result of calling Pip is then shown. Here, for instance the schedule says that `B[i,j,k]` is computed at time `i`. The result (after `Out[30]`) is the corresponding MATHEMATICA expression representing `$schedule`.

For such a little program, only a few seconds are necessary to find out a schedule. But it may take quite a long time for larger ALPHA programs.

10.2.2 Format of the output of `schedule`

The output of the `schedule` function has a special form. It is enclosed in a structure whose head is `Alpha‘ScheduleResult`. The first argument of this structure is the type of the schedule (integer, coded as the option scheduleType, see section 10.3.1) and its second argument is the schedule itself. The syntax of this structure is:

```
<schedResult> ::= Alpha‘scheduleResult[scheduleType_Integer,{sched...}]
  <sched> ::=  { nameVar_String,
                 indices_List,
                 Alpha‘sched[tauVector_List,constCoef_Integer] }
```

The effect of the `schedule` function is to assign this form to te global variable `$schedule`.

Another example is given in section 10.4. A schedule can be displayed using the `show` command:

```
In[31]:=show[$schedule]

T_a{i, j, N, M} = 0
T_b{i, N, M} = 0
T_c{i, N, M} = 1 + M
T_C{i, j, N, M} = j

Out[31]:=
```

10.2.3 Using the result of the scheduler

We may use the result of the scheduler as an information on the program, but we can also tranform the ALPHA program such that this information become explicit in the program. For instance, on the matrix-vector example, if we perform the change of basis `(i,j -> j,i)` on variable `C` and if we rename (for instance) the first index `t` and the second one `j`, we obtain the program of figure 10.3.

Figure 10.3: Scheduled version of program of figure 10.1

This program is functionnaly equivalent to the one of figure 10.1 but it shows exactly at which clock tick each value of the local variable `C` is computed: `C[t,i]` is computed at step `t`. We call a program which has this property a *scheduled* ALPHA program. Of course, in a scheduled ALPHA program, the first index of the local variables defines a partial order which is compatible with data dependencies.

The scheduled program of figure 10.3 has been obtained by evaluating the expression `applySchedule[]`. This command applies a change of basis to each local variable of the program contained in `$result` in such a way that the first index in the new basis represent the time. The change of basis is not performed on the inputs and outputs, as otherwise the new program would not be equivalent to the previous one. Another example of the use of `applySchedule[]` is shown in section 10.4.

10.3 Advanced scheduling

In this section we introduce some of the parameters of the `schedule` command which allow more precise results to be obtained. A detailed description of this function can be found in the documentation file `$MMALPHA/doc/user/Scheduler_user_manual.ps`.

10.3.1 Options of schedule

The `schedule` function has many options. These options have default values indicated hereafter. To change these values, put one of the corresponding replacement rules as a parameter to the `schedule` function (see the example in section 10.4).

Option `scheduleType` This option gives the type of schedule. Its possible values are:

- `scheduleType -> affineByVar` (default) affine by variable scheduling;
- `scheduleType -> sameLinearPart` affine by variable scheduling with constant linear part;
- `scheduleType -> ?` affine by variable scheduling with constant linear part except for the parameters.

objFunction This option gives the objective function chosen. Its type is integer, the possible values are:

- `objFunction -> 0` (default) the total latency is minimized.
- `objFunction -> 1` no objective function (the coefficients of the scheduling vectors are minimized in a lexicographic order).

  It is mandatory not to minimize the total execution time when this time is not bounded (otherwise, the schedule will fail).

ratOrInt The resolution of the linear programming problem generated is done by the software MPPip which is an infinite precision version of the sofware PIP [?]. The resolution can be done with integer programming or rational linear programming. This option indicates whether the resolution of the LP by MPPip is done in integer mode (resulting scheduling vectors with integral coefficients) or in rationnal mode (resulting scheduling vectors may contains rationnal coefficients). Its type is integer, the possible values are:

- `ratOrInt -> 0` : rationnal solution.
- `ratOrInt -> 1` : (default) integer solution.

addConstraints This option allows to add some constraints to the LP generated. This option is very usefull to guide the scheduling process. Its type is a list of string, each string representing a constraint. The constraints authorized are affine constraints on scheduling vectors. There are two types of constraint added, one can force a scheduling vector value or simply set linear constraints on its components.

- forcing a variable A to be schedule at time i+2j+2 can be done with the constraints: `"TA[i,j]=i+2j+2"` .
- for more precise constraint, one can directly access to each components of the schedule functions of each variable. For instance `TAD2` will represent the variable coefficient of the second indice in the schedule of variable `A` (and `CA` will represent the constant part in the schedule of variable `A`). With these names, one can set linear constraints on these coefficients using operators `==` or `>=`. For instance, `{"TAD1 == 1","TAD2 == 2", "CA >= 2" }` is the same constraint as above except thar the constant is allowed to be greater than two.

Example of use:
`schedule[addConstraints->{"TA[i,j,N]=i+2j-2", "TBD2==2","TBD1+2TCD3>=1"}]`

duration indicates how to count the duration of each computation. The possible value for the duration option are:

- `duration -> {}` : (default) each equation takes 1 step (whatever complex the computation is).
- `duration -> {Integer...}` :

  The duration of each equation is given by the user. The list must contain as many integer as there are variables (do not forget the input variables). For instance on the program of figure 10.1, the command `schedule[]` is equivalent to the command: `schedule[duration->{1,1,1,1}]`.

10.4 Another example

Consider the program of figure 10.4, which represent a uniform program for the multiplication of matrices. in this section, we give the result of the schedule with different options.

Figure 10.4: Uniform matrix matrix product

Default use If you type the following command (after having loaded the program of figure 10.4):
`schedule[]`
the result should be:

```
T_a{i, j, N} = 0
T_b{i, j, N} = 0
T_c{i, j, N} = 1 + 2 N
T_B{i, j, k, N} = i
T_A{i, j, k, N} = j
T_C{i, j, k, N} = k + N

Out[14]= scheduleResult[1, {{a, {i, j, N}, sched[{0, 0, 0}, 0]},
   {b, {i, j, N}, sched[{0, 0, 0}, 0]}, {c, {i, j, N}, sched[{0, 0, 2}, 1]},
   {B, {i, j, k, N}, sched[{1, 0, 0, 0}, 0]},
   {A, {i, j, k, N}, sched[{0, 1, 0, 0}, 0]},
   {C, {i, j, k, N}, sched[{0, 0, 1, 1}, 0]}}]
```

The result (after `Out[14]`) is the Mathematica structure assigned to `$schedule`.

Adding a constraint Suppose that the inputs `a[i,j]` and `b[i,j]` arrive respectively at time `i+j` and `i+j+3`. Moreover, we want a schedule of unique linear part (because this allow a locally connected array to be obtained from a uniform program). We have to add the two constraints: `"Ta[i,j]=i+j"` and `"Tb[i,j]=i+j+3"`. Also, we need to set the option scheduleType to 2, hence the command is:

```
In[15]:= schedule[addConstraints->{"Ta[i,j]=i+j","Tb[i,j]=i+j+3"},
                  scheduleType->2]
```

The result is

```
T_a{i, j, N} = i + j
T_b{i, j, N} = 3 + i + j
T_c{i, j, N} = 5 + i + j + N
T_B{i, j, k, N} = 3 + i + j + k
T_A{i, j, k, N} = i + j + k
T_C{i, j, k, N} = 4 + i + j + k
```

Note that the unique linear part option is applied on local variables only (indeed, it cannot be enforced on inputs and outputs which do not have the same number of indices.) After scheduling, one can obtain a scheduled program by typing the command: `applySchedule[]`. The result is the program of figure 10.5.

Warning: the `applySchedule` function finds the change of basis by unimodular completion of the scheduling vector. The unimodular completion is not always possible (for instance, if the scheduling vector is null), in that case, applySchedule will perform a generalized change of basis and add a new dimension to the domain of the corresponding variable. A warning is printed out during this operation.

Figure 10.5: Scheduled matrix matrix product

10.4.1 What if no schedule can be found?

Due to the restriction to linear schedule, there may happen that the schedule function only answers: `No schedule was found, sorry...` There maybe several reasons for that:

- The program is not semantically correct, try `analyze[]`.
- There exists a schedule but the time is not bounded. In this case try with the option `objFunction` set to 1.
- No affine one dimensional schedule exists. You can try to find a multidimensional schedule with the `multiSched`.
- No schedule exists (there is no way of solving this problem).

10.5 To come soon

- Multi-dimensional schedule.
- Structured scheduling.
- Data-flow scheduling.

Chapter 11

The Alpha0 format

Version of March 3, 2007

In this chapter, we describe the Alpha0 format which is basically a non-structured version of the AlpHard language. We also describe the methodology to go from an Alpha program to an Alpha0 program and then how to translate it into AlpHard.

11.0.1 Definition of Alpha0

Alpha0 was introduced by C. Dezan in her Phd thesis ([?]). It was conceived as the lowest level of Alpha or equivalently as a subset of Alpha which can describe circuits at the register transfer level. the main weakness of this subset came from the fact that Alpha programs were not structured. Since, the structuring of Alpha was provided by F. Dupont [DQR95] and the structured version of Alpha0 was studied by P. Le Moënner [LPR+96]. AlpHard (described in chapter ??) is now intended to be this subset of Alpha with structural interpretation, from which we can translate Alpha into vhdl or other rtl description languages. However, during the transformation path from Alpha to AlpHard, the automatic structuring is a complicated process, hence it appeared that the Alpha0 format is still necessary as an intermediate format with all the register transfer level informations but in an unstructured form. We describe hereafter, the second version the Alpha0 format (Alpha0v2) which is slightly different from the version described in [?]

An Alpha program is in Alpha0 form if it contains no `use` statement and if it meets the following conditions:

1. there exits, for all declared domains (except for the domains of inputs and outputs variables) an interpretation function for the indices (see [?] for precise definition of *interpretation function*). Basically, each index is either a temporal index (indicating living dates of the signal) or a spatial index (indicating in which processor the signal lives).

2. The equations of the system define the output and local variables. Each operator involved in the equations has a structural interpretation. these equations are of four types: data equations, connection equations, control equations and mirror equations.

- data equations define the different signals of the program which are necessarily local variables. They are composed of the following operators:
  - pointwise operators represent the corresponding combinatorial operators.
  - restriction are used to restrict the spatial interpretation of indices.
  - dependencies are temporal dependencies representing registers or identity dependencies representing connections between two signal inside one cell.
  - case are spatial case allowing to gather several signals.
  - the `if` operator (with nested `case` in each branches) is interpreted as a multiplexer.
- connection equations are limited to the following form: a single spatial dependency between two signals (`A=B.(t,p->t,p-1)` for instance).
- control equations allow initialize control signals. they define signals which are only temporal (no spatial indices).
- Mirror equations: equation between input variables of the system and local variable of the system (only for interface). equation limited to an affine function applied to a variable (`A[t,p]=a[f(t,p)]` for input mirror equation and `b[i,j]=B[f(i,j)]` for output mirror equations)..

Examples of ALPHA0 programs are present in the directory `$MMALPHA/examples/Alpha0`, a small example is presented hereafter (figure 11.4).

11.1 From ALPHA to ALPHA0

The first attempt to go from High level specification in ALPHA to RTL specification in ALPHA0 has been done be Sie and Wilde [WS94]. This methodology has been extended and we discribe it briefly here, more informations can be obtained by executing the Notebooks demos: `Fir` and Matvect.

Consider the very simple program of figure 11.1

Translating the program in ALPHA0 consists in the following steps:

1. **Uniformize the program**. Here the program is already uniform, in general this procedure may be complicated MMALPHA provides tools for the automatic or designer-guided uniformization (see the documentation on the chapter **??**).

2. **Schedule the program**. It consists in finding an execution date (and a place also) for each computation. Usually, once uniformized, the program should be scheduled with a unique linear part (see the chapter 10 for detail on scheduling). The function `applySchedule` choose an allocation function. In our example, you can execute the following commands:

```
system NtimeNot :{N | 2<=N}
                (a : {i | 1<=i<=N} of boolean)
       returns  (b : {i | 1<=i<=N} of boolean);
var
  Acc : {i,j | 1<=i<=N; 1<=j<=N} of boolean;
let
  Acc[i,j] =
      case
        {| j=1} : a[i];
        {| 2<=j} : not Acc[i,j-1];
      esac;
  b[i] = Acc[i,N];
tel;
```

Figure 11.1: An Alpha program computing `N` times `Not` on an array `a`

```
schedule[]
applySchedule[]
renameIndices[{"t","p"}]
```

We obtain a linear array of processors, the resulting program is shown in figure 11.2

```
system NtimeNot :{N | 2<=N}
                (a : {i | 1<=i<=N} of boolean)
       returns  (b : {i | 1<=i<=N} of boolean);
var
  Acc : {t,p | 1<=t<=N; 1<=p<=N} of boolean;
let
  Acc[t,p] =
      case
        {| t=1} : a[p];
        {| 2<=t} : not Acc[t-1,p];
      esac;
  b[i] = Acc[N,i];
tel;
```

Figure 11.2: Program of figure 11.1 after applying schedule

3. **Pipeline Inputs and Output**. You can see in figure 11.2 that the `a` is input in every processors. If we want only the first processor to input data from the host, we could have pipelined the input with the following command:

`pipeIO["Acc","a.(t,p->p)","aIn.(t,p->t+1,p+1)","{t,p| p>=1}"]`. We can do the same treatment for the output. Here we chose not to pipeline the I/O for sake of simplicity.

4. **Go Down to Alpha0**. When the previous steps have been correctly executed, this step should be automatic with `toAlpha0v2[]` command. In Our case, if we apply this function to the program of figure 11.2, the printing on the screen are the following:

```
In[__]:= toAlpha0v2[];
Time index: {1}  space indices: {2}
Calling spaceTimeDecomposition[];
Calling makeAllMuxControl[];
  Equation of Acc...
   is in ST form
    Adding multiplexer.
  Equation of b...
Calling pipeAllControl[];
  Pipelining control for: Acc_ctl1
     From dimension 2 To dimension 1
        Warning, no pipe vector was found for control signal Acc_ctl1
        --> assuming broadcasted signal
     Control generated in cell: {p | 1<=p<=N; 2<=N}
Calling decomposeSTdeps[];
  In equation of Acc, adding a local variable: Acc_reg1
 Decomposing the space/time dependencies
Calling makeInputMirrorEqus[];
  Adding mirror equation for input a
Out[__]:=
```

The important treatment here is the generation and pipeline of a control signal. Indeed, the condition `t=1` of the program of figure 11.2 has to be realized in hardware. Hence a control signal was generated and a multiplexer added in each processors. Then MmAlpha tries to pipeline the control signal (it often happens in systolic architectures that the control can be pipelined). In our particular example, all processors start their computation simultaneously, hence the control signal is not pipelined but broadcasted. Other treatments are just syntactic rewriting. The result Alpha0 program is show on figure 11.3. note the precise information that appears on the life time of each signal (in this simple program, each signal lives from step 1 to step `N`, but in general it is much more complicated) and on the input date of each data (mirror equation contains the information of *how to use* the hardware – i.e. interface specification –).

11.2 From Alpha0 to AlpHard

The translation to AlpHard is automatic. the command to use is `alpha0toAlphard[]`. MmAlpha automatically detects all the spatial regions on which the computations are identical (the whole spatial region in our case: $\{p \| 1 \leq p \leq N\}$ and structures the program with one cell (i.e. one subsystem) for each region, one controller for initialization of control signal, one module for the complete circuit and one interface to keep the same I/O as the original program of figure 11.1. The interface system obtained in our example is shown on figure 11.4.

```
system NtimeNot :{N | 2<=N}
                (a : {i | 1<=i<=N} of boolean)
       returns  (b : {i | 1<=i<=N} of boolean);
var
  a_mirr1 : {t,p | t=1; 1<=p<=N} of boolean;
  Acc_reg1 : {t,p | 2<=t<=N; 1<=p<=N} of boolean;
  Acc_ctl1_In : {t,p | 1<=t<=N; 1<=p<=N} of boolean;
  Acc : {t,p | 1<=t<=N; 1<=p<=N} of boolean;
  Acc_ctl1 : {t | 1<=t<=N} of boolean;
let
  a_mirr1[t,p] = a[t+p-1];
  Acc_reg1[t,p] = Acc[t-1,p];
  Acc_ctl1_In[t,p] = Acc_ctl1[t];
  Acc_ctl1[t] =
      case
        case
          {| t=1} : True[];
          {| 2<=t} : False[];
        esac;
      esac;
  Acc[t,p] =
      case
        {| 1<=t} : if (Acc_ctl1_In) then
               case
                 {| t=1} : a_mirr1;
                 {| 2<=t} : False[];
               esac else
               case
                 {| t=1} : False[];
                 {| 2<=t} : not Acc_reg1;
               esac;
      esac;
  b[i] = Acc[N,i];
tel;
```

Figure 11.3: Program ALPHA0 derived from the program of figure 11.1. This ALPHA0 program describes a linear array of `N` processors. The equation defining `Acc` represent a multiplexer controlled by `Acc_ctl1` selecting `a_mirr1` or `Acc_reg1`. The equation defining `Acc_ctl1_In` is a control equation. The equations defining `a_mirr1` and `b` are mirror equations. The equation defining `Acc_reg1` is interpreted as a register. The equation defining `Acc_ctl1_In` is interpreted as a connection equation: broadcast of the control signal `Acc_ctl1` to all processors.

```
system NtimeNot :{N | 2<=N}
                (a : {i | 1<=i<=N} of boolean)
       returns  (b : {i | 1<=i<=N} of boolean);
var
  a_mirr1 : {t,p | t=1; 1<=p<=N} of boolean;
  Acc : {t,p | 1<=t<=N; 1<=p<=N} of boolean;
let
  a_mirr1[t,p] = a[t+p-1];
  b[i] = Acc[N,i];
  use  NtimeNotModule[N] (a_mirr1) returns  (Acc) ;
tel;
```

Figure 11.4: interface of the AlpHard Program obtain from the program of figure 11.3 by the `alpha0ToAlphard[]` command

Chapter 12

The AlpHard language

Version of March 3, 2007

In this chapter, we describe the AlpHard language. We also describe the vhdl generator.

12.1 AlpHard

The main goal of Alpha is to allow someone to produce a hardware implementation for an Alpha specification. To this end, a subset of Alpha, called AlpHard, is defined. We quickly introduce this language.

12.1.1 Basic concepts

AlpHard is a subset of the Alpha language that enables a structural definition of a regular architecture (systolic arrays, etc.) to be given.

AlpHard is intended to meet two important goals. First, AlpHard programs are obtained by automatic transformation of Alpha programs, and hence, AlpHard it must be a coherent subset of Alpha. Second, the architectural description given by AlpHard must:

- Provide *structuring* because complex design process must be hierarchical.
- Provide *genericity* in order to allow component reuse.
- Allow *regularity* to be described, in order to reuse hardware descriptions and simplify the design process.

Thus, AlpHard is hierarchical. At the lowest level of description one has *cells* consisting of combinational circuits, registers, multiplexors, etc. A cell may also contain other cells. At the same level in the hierarchy, we may also have *controllers* responsible for initialization[1].

[1]In systolic arrays, the control signals are themselves distributed in a regular fashion, and their data-path can also be described in terms of cells.

At this point we have a description of a piece of circuit that has no “spatial dimension”: it represents a single processing element which may later be instantiated at multiple spatial locations (like the full adder presented in section **??**).

The next level of the hierarchy is the *module* which allows the user to specify how different cells are assembled regularly in one or more spatial dimensions. This is achieved by *instantiating* previously declared cells. Typically, controllers are instantiated only once with no spatial replication. The separation of temporal and spatial aspects is also reflected by the fact that the equations describing the behavior of cells have only one local index variable (time), and in the equations for modules, the *dependencies* have only spatial indices, indicating pure (delay-less) interconnections (thus all registers are viewed as part of the cells).

12.1.2 A small example

These ideas are illustrated in figure 12.1 and 12.2, which show a three-cell module, where each cell is a simple register-inverter. The AlpHard code corresponding to the *cell* definition is in figure 12.2. The system `RegInvCell` has two parameters (lines 1) corresponding to its start time and its duration. Lines 3-5 are the declaration of input and output signals of the cell, the *domain* between the braces represents the lifetime of signals `A1` and `B1`. The equation defining `B1` (line 7) consists of an inverter combined with a delay `A1[t-1]` which represents a register. Notice that the only index appearing in the equations is the time, `t`.

The `RegInvModule` system is a *module* which uses the cell `RegInvCell` . Multiple copies are instantiated by means of the construct, `use {p|1<=p<=Size}`. Such a multiple instantiation also means that the variables `A` and `B` will have two indices, `t` (inherited from the `RegInvCell` definition) and `p` (from the multiple use in line 17). We also see that the value of the `Tinit` parameter is instantiated as a function of `p` so that the cells start at different instants of time. Lines 19-22 specify the connections between the input/output of the cells in the module. Notice that the constraints in the `case` involve only spatial indices (if it had involved temporal indices –i.e. `t`–, we would need some additional control hardware not described here). Also note that the time index, `t`, is the same on both sides of the equation (no register outside the cells).

For more information on AlpHard see [LPR+96] [**?**] and there are example of Alphard programs in the the directory `$MMALPHA/examples/Alphard`

To have information on the VHDL generator, see the notebooks:

- matvect demonstration,
- Fir demonstration,
- Vhdl.

They are all available as hyperlinks in the `master.nb` that you can start using the command `on[]` or `start[]`.

12.2 generating VHDL from AlpHard

The AlpHard format was conceived by P. Le Monner together with the translator to VHDL. The targeted VHDL is *synthetizable* VHDL, it has been tested on commercial tools like Compass, Synopsis,.... Currently, the VHDL translator allows to generate VHDL code from AlpHard programs which represent linear systolic arrays. The reason why we cannot generate code for 2-dimensional arrays is that the *synthesizable* VHDL*subset* [?] does not allow nested `generate` instructions[2].

12.2.1 Setting up the translation

the only operation that have to be done before translating AlpHard code is to affect the parameters value in the module and the controler (we cannot described arrays with parametric size in VHDL). As the current AlpHard program is a library, this process may be a little complicated (you have to ensure that the parameter instantiations between the caller and the called program is respected). Currently this can be done with `assignParameterValue` function. this function assign the parameter value in one system (`$result`), if you want to recursively assign the parameter value to the systems called by `$result`, you can use the `fixParameter` function which assigns a value to a given parameter for all systems in `$library`[3]. In the example of figure 12.1, you can use the following commands:

```
 assignParameterValue["Size",10];
assignParameterValue["Tinit",0];
assignParameterValue["Duration",100]
```

The resulting program is shown in figure 12.3. The command `alphardToVHDL[]` will generate one file per system of the library. Here, one file for the cell (named `regInvCell.vhd`) and one file for the module (named `regInvModule.vhd`). The VHDL code for the cell `RegInvCell` and the module `RegInvModule` are shown in figure **??** and 12.5

12.2.2 Description of the generated VHDL

`todo:` describe the translation of operators, registers, multiplexer, controler, cells, modules.

[2]The VHDL `generate` instruction is used for expressing the repetitive use of a cell in an array

[3]You have to ensure that this parameter has indeed exactly the same value in all systems of the library, this is not the case for `Tinit` in our example

```
system                                                        -- 1
RegInvCell:{Tinit,Duration| Tinit,Duration >= 0}              -- 2
           (A1 : {t | Tinit<=t<=Tinit+Duration} of boolean)    -- 3
    returns                                                   -- 4
           (B1 : {t | Tinit+1<=t<=Tinit+Duration+1} of boolean);-- 5
let                                                           -- 6
        B1[t] = not A1[t-1];                                  -- 7
tel;                                                          -- 8


system
RegInvModule:{Tinit,Duration,Size| Tinit,Duration, Size > 0}    -- 9
(a: {t|  Tinit<=t<=Tinit+Duration} of boolean)           -- 10
   returns                                                      -- 11
(b: {t|  Tinit+Size<=t<=Tinit+Duration+Size} of boolean);-- 12
var
A : {t,p | Tinit+p-1<=t<=Tinit+Duration+p-1; 1<= p <= Size}  of boolean;
B : {t,p |  Tinit+p<=t<=Tinit+Duration+p; 1<= p <= Size}  of boolean;
let
use {p | 1<= p <= Size} RegInvCell[Tinit+p-1,Duration]          -- 17
                        (A) returns (B);                        -- 18
A[t,p] = case                                                   -- 19
      {| p=1}: a[t];                                            -- 20
      {| p>1}: B[t,p-1];                                        -- 21
      esac;                                                     -- 22
b[t]=B[t,Size];                                                 -- 23
tel;
```

Figure 12.1: An AlpHard program describing a simple cell, (`RegInvCell`), and instantiation of `Size` copies of it in a module (`regInvModule`). Note how `p` is used to specify the value `Tinit` for each instance (the `p`-th cell starts `p-1` cycles after the first one).

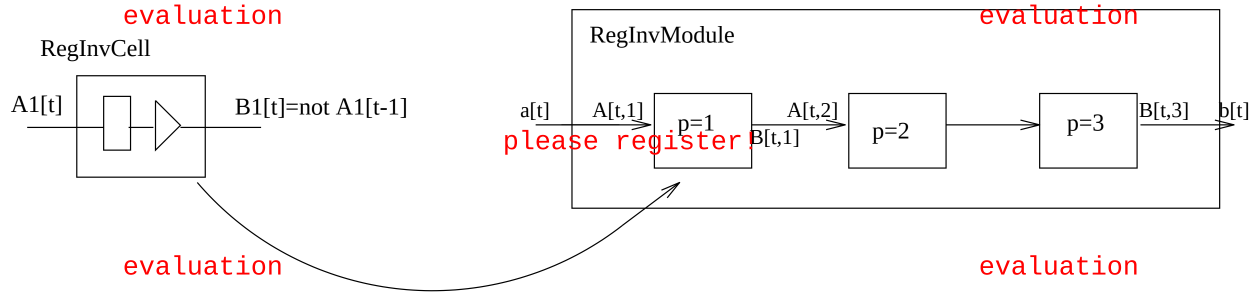


Figure 12.2: A simple architecture consisting of three identical cells

```
system RegInvCell :{Tinit,Duration | 0<=Tinit; 0<=Duration}
                  (A1 : {t | Tinit<=t<=Tinit+Duration} of boolean)
       returns    (B1 : {t | Tinit+1<=t<=Tinit+Duration+1} of boolean);
var
  A2 : {t | Tinit+1<=t<=Tinit+Duration+1; 0<=Tinit} of boolean;
let
  A2[t] = A1[t-1];
  B1[t] = not A2;
tel;

system RegInvModule (a : {t | 0<=t<=100} of boolean)
       returns      (b : {t | 10<=t<=110} of boolean);
var
  A : {t,p | p-1<=t<=p+99; 1<=p<=10} of boolean;
  B : {t,p | p<=t<=p+100; 1<=p<=10} of boolean;
let
  use {p | 1<=p<=10} RegInvCell[p-1,100] (A) returns  (B) ;
  A[t,p] =
      case
        {| p=1} : a[t];
        {| 2<=p} : B[t,p-1];
      esac;
  b[t] = B[t,10];
tel;
```

Figure 12.3: The AlpHard program of figure 12.1 with particular value for the parameters of the module. Note that the cell is still parameterized.

```
-- VHDL Model Created for "system RegInvCell"
 --  20/5/1999 11:27:36

library IEEE;
use IEEE.std_logic_1164.all;
library COMPASS_LIB;
 use COMPASS_LIB.STDCOMP.all;
library COMPASS_LIB;
 use COMPASS_LIB.COMPASS.all;

entity RegInvCell is
      Port ( Ck : In std_logic;
      A1 : In std_logic;
      B1 : Out std_logic );
end RegInvCell;


architecture Behavioral of RegInvCell is

  signal A2 : std_logic;

begin


  process(ck)
  begin
    if (ck='1' AND ck'event) then
        A2 <= A1;
    end if;
  end process;

  B1 <= ( not A2);


end Behavioral;
```

Figure 12.4: VHDL code generated from the `RegInvCell` cell of figure 12.3

```
todo
```

Figure 12.5: VHDL code generated from the `RegInvModule` cell of figure 12.3

Appendix A

The syntax of ALPHA

This appendix gives the syntax and the abstract syntax of the ALPHA language. We plan to illustrate this description with nice examples...

A.1 Definition of ALPHA

A.1.1 Meta Syntax

*phrase**	===	zero or more repetitions of *phrase*.
phrase1 | *phrase2*	===	alternation, either *phrase1* or *phrase2*.
[. . .]	===	optional phrase.
(. . .)	===	syntactic grouping.
bold	===	a terminal.
Italic	===	a non-terminal.

A.1.2 Systems

```
Program         ::   PDecl  PDecl *
PDecl           ::   SystemDecl

SystemDecl      ::   system Name [  :  ParamDecl  ] ( InputDeclList )
                       returns ( OutputDeclList ) ;
                       [  var LocalDeclList ; ]
                       Equationblock ;

Name            ::   Identifier

ParamDecl       ::   Domain

InputDeclList   ::   VarDeclList
```

```
OutputDeclList    ::   VarDeclList
LocalDeclList     ::   VarDeclList
```

A.1.3 Declarations of variables

```
VarDeclList       ::   VarDeclList *
VarDeclaration    ::   IdentifierList  :  [ Domain of ] ScalarType ;
ScalarType        ::   integer | real | boolean
```

A.1.4 Domains

```
Domain            ::   { IndexList | ConstraintList }
                     | Domain | Domain
                     | Domain & Domain
                     | Domain .AffineFunction
                     | ~ Domain
                     | Domain .convex
                     | ( Domain )

IndexList         ::   [ IndexList , ] Identifier

ConstraintList    ::   [ ConstraintList ; ] Constraint
Constraint        ::   IncreasingSeq | DecreasingSeq | EqualitySeq
IncreasingSeq     ::   ( IncreasingSeq | IndexExpList ) ( < | <= ) IndexExpList
DecreasingSeq     ::   ( DecreasingSeq | IndexExpList ) ( > | >= ) IndexExpList
EqualitySeq       ::   ( EqualitySeq | IndexExpList ) = IndexExpList
```

A.1.5 Equations

```
Equationblock     ::   let EquationList tel
EquationList      ::   [ EquationList ] Equation
Equation          ::   Identifier [ IndexList ] = Expression ;
                     | Identifier = Expression ;
                     | use [ ExtensionDomain ] Identifier [ .ParamAssignation ]
                         ( InputList )
                         returns ( IdentifierList ) ;

ParamAssignation  ::   AffineFunction

InputList         ::   [ InputList , ] Expression

ExtensionDomain   ::   Domain
```

A.1.6 Expressions

```
Expression        ::   case ExpressionList esac
                     | if Expression then Expression else Expression
                     | Domain :Expression
                     | Expression .AffineFunction
                     | Expression [ IndexExpList ]
                     | Expression BinaryOp Expression
                     | BinaryOp ( Expression , Expression )
                     | UnaryOp Expression
                     | reduce ( CommutativeOp , AffineFunction , Expression )
                     | ( Expression )
                     | Identifier
                     | Constant

ExpressionList    ::   [ ExpressionList ] Expression ;

BinaryOp          ::   CommutativeOp | RelativeOp | - | div | mod
CommutativeOp     ::   + | * | and | or | xor | min | max
RelativeOp        ::   = | <> | < | <= | > | >=
UnaryOp           ::   - | not | sqrt

Constant          ::   IntegerConstant | RealConstant | BooleanConstant
```

A.1.7 Dependance Functions and Index Expressions

```
AffineFunction    ::   ( IndexList -> IndexExpList )
IndexExpList      ::   [ IndexExpList , ] IndexExpression | IndexExpression
IndexExpression   ::   IndexExpression ( + | - ) IndexTerm | [ - ] IndexTerm
IndexTerm         ::   IntegerConstant Identifier | IntegerConstant | Identifier
```

A.1.8 Terminals

```
IntegerConstant   ::   [ - ] Number
RealConstant      ::   [ - ] Number .Number
BooleanConstant   ::   true | false |True | False
Number            ::   Digit Digit *
Digit             ::   0 | 1 |...| 9
Identifier        ::   Letter ( Letter | Digit ) *
Letter            ::   a |...| z | A |...| Z | _
```

A.2 ALPHA Abstract syntax

A.2.1 Meta Syntax

*phrase**	===	zero or more repetitions of *phrase*.
phrase1 | *phrase2*	===	alternation, either *phrase1* or *phrase2*.
[...]	===	optional phrase.
(...)	===	syntactic grouping.
`bold`	===	a terminal.
Italic	===	a non-terminal.

A.2.2 Systems

```
Program          ::   PDecl  PDecl *
PDecl            ::   SystemDecl

SystemDecl       ::   system Name [ : ParamDecl ] ( InputDeclList )
                        returns ( OutputDeclList ) ;
                        [ var LocalDeclList ; ]
                        Equationblock ;

Name             ::   Identifier

ParamDecl        ::   Domain

InputDeclList    ::   VarDeclList
OutputDeclList   ::   VarDeclList
LocalDeclList    ::   VarDeclList
```

A.2.3 Declarations of variables

```
VarDeclList      ::   VarDeclList *
VarDeclaration   ::   IdentifierList : [ Domain of ] ScalarType ;
ScalarType       ::   integer | real | boolean
```

A.2.4 Domains

```
Domain           ::   { IndexList | ConstraintList }
```

```
                   | Domain | Domain
                   | Domain & Domain
                   | Domain .AffineFunction
                   | ~ Domain
                   | Domain .convex
                   | ( Domain )

IndexList        ::  [ IndexList , ] Identifier

ConstraintList   ::  [ ConstraintList ; ] Constraint
Constraint       ::  IncreasingSeq | DecreasingSeq | EqualitySeq
IncreasingSeq    ::  ( IncreasingSeq | IndexExpList ) ( < | <= ) IndexExpList
DecreasingSeq    ::  ( DecreasingSeq | IndexExpList ) ( > | >= ) IndexExpList
EqualitySeq      ::  ( EqualitySeq | IndexExpList ) = IndexExpList
```

A.2.5 Equations

```
Equationblock    ::  let EquationList tel
EquationList     ::  [ EquationList ] Equation
Equation         ::  Identifier [ IndexList ] = Expression ;
                   | Identifier = Expression ;
                   | use [ ExtensionDomain ] Identifier [ .ParamAssignation ]
                       ( InputList )
                       returns ( IdentifierList ) ;

ParamAssignation ::  AffineFunction

InputList        ::  [ InputList , ] Expression

ExtensionDomain  ::  Domain
```

A.2.6 Expressions

```
Expression       ::  case ExpressionList esac
                   | if Expression then Expression else Expression
                   | Domain :Expression
                   | Expression .AffineFunction
                   | Expression [ IndexExpList ]
                   | Expression BinaryOp Expression
                   | BinaryOp ( Expression , Expression )
                   | UnaryOp Expression
                   | reduce ( CommutativeOp , AffineFunction , Expression )
```

```
                        | ( Expression )
                        | Identifier
                        | Constant

ExpressionList     ::   [ ExpressionList ] Expression ;

BinaryOp           ::   CommutativeOp | RelativeOp | - | div | mod
CommutativeOp      ::   + | * | and | or | xor | min | max
RelativeOp         ::   = | <> | < | <= | > | >=
UnaryOp            ::   - | not | sqrt

Constant           ::   IntegerConstant | RealConstant | BooleanConstant
```

A.2.7 Dependance Functions and Index Expressions

```
AffineFunction     ::   ( IndexList -> IndexExpList )
IndexExpList       ::   [ IndexExpList , ] IndexExpression | IndexExpression
IndexExpression    ::   IndexExpression ( + | - ) IndexTerm | [ - ] IndexTerm
IndexTerm          ::   IntegerConstant Identifier | IntegerConstant | Identifier
```

A.2.8 Terminals

```
IntegerConstant    ::   [ - ] Number
RealConstant       ::   [ - ] Number .Number
BooleanConstant    ::   true | false |True | False
Number             ::   Digit Digit *
Digit              ::   0 | 1 |...| 9
Identifier         ::   Letter ( Letter | Digit ) *
Letter             ::   a |...| z | A |...| Z | _
```

Appendix B

The demos of MmAlpha

Version of March 3, 2007

This chapter presents a few demos of MmAlpha. There two types of demos: some demos have a notebook interface, and the others should be executed in kernel mode. There are also various documentations available with the Alpha archive.

B.1 Notebook demos

All these demonstrations can be accessed directly by hyperlinks of the so-called *master* notebook, which is started by evaluating the command `start[]`.

Introduction. Introduction to Alpha

Domlib. Elementary computations on domains with Domlib.

Fir: Design of a FIR filter. The FIR demo shows the typical steps for the synthesis of a linear architecture for a Finite Impulse Response filter. The demo comprises the following steps: loading the initial program, pipelining, scheduling, assign parameter values, control signal generation, Alpha0 generation, AlpHard generation.

matvect: architecture for matrix vector multiplication. The MATVECT demo shows the typical steps for the synthesis of a linear architecture for a Matrix Vector Multiplication. The demo comprises the following steps: loading the initial program, pipelining, scheduling, assign parameter values, control signal generation, Alpha0 generation, AlpHard generation.

Tutorial_intro: very introductory tutorial. Corresponds to document [?].

mma-intro: quick introduction to Mathematica. Explains how Alpha ASTs are represented in Mathematica. Warning: the abstract syntax presentation may not be as reliable as the appendix of this manual (see chapter A).

Kalman: this example describes an application of MmAlpha to a Kalman filter. This application, quite complex, shows how to use structured scheduling. Reference [?] describes the background of this application.

Kalman-Sqrt: this example describes an application of MmAlpha to a Kalman filter, using a so-called square root covariance method. This application, quite complex, shows how to use structured scheduling. Reference [?] describes the background of this application.

Neural-Network: this notebook is not ready yet, but will be soon.

Fuzzy-Logic: an application of fuzzy logic to channel equalization.

Another serie of notebook allows one to become familiar with different packages of MmAlpha:

Alpha: the Alpha package.

Control: the Control package.

Matrix: the Matrix package.

ChangeOfBasis: the ChangeOfBasis package.

Normalization: the Normalization package.

Cut: the Cut package.

Decomposition: the Decomposition package.

Static: the Static package.

PipeControl: the PipeControl package.

Vhdl: the Vhdl package.

CheckAlpHard: the AlphHard package.

Meta: the Meta package.

Schedule: the various schedule packages.

dataFlowSchedule:

matlib:

Visual: the Visual package.

B.2 kernel demos

These demos are avaiblable at the following address:

`/udd/alpha/alpha\_beta/Mathematica/demos/demos.html`

INIT: An initiation to MMALPHA. The INIT demo shows a few basic functions of MMAlpha. The demo comprises the following steps: loading and showing a program, normalizing, showing domains etc...

FIR: Design of a FIR filter. The FIR demo shows the typical steps for the synthesis of a linear architecture for a Finite Impulse Response filter. The demo comprises the following steps: loading the initial program, pipelining, scheduling, assign parameter values, control signal generation, ALPHA0 generation, ALPHARD generation.

MATVECT: architecture for matrix vector multiplication. The MATVECT demo shows the typical steps for the synthesis of a linear architecture for a Matrix Vector Multiplication. The demo comprises the following steps: loading the initial program, pipelining, scheduling, assign parameter values, control signal generation, ALPHA0 generation, ALPHARD generation.

SVD: analysis of a Singular Value Decomposition algorithm. The SVD demo shows the analysis of a larger Alpha program. It shows the use of subsystems, the nlining subsystems, the CNF code generation, the C code generation and execution.

BINMULT: Synthesis of a multiplier. This demonstration shows the synthesis of a bit-serial multiplier, from an initial specification structured into several parameterized subsystems. The demo script contains the following steps: loading and displaying a multi-system program, conversion into C code for simulation, pipelining of the diffusions in the initial program, flattenning all the systems for synthesis, scheduling before space/time transformation, space/time transformation, pipelining of the inputs (so that all the data enters the first processor in bit-serial manner), simulation of the resulting array.

ESTIM: synthesis of motion estimation a two dimensionnal example, with simulation, on a real image, derivation of ALPHA0 code (warning, long demo).

Appendix C

Installing MmAlpha

C.1 Getting started

You may use either Mathematica under `emacs`, or directly on the Unix system.

- under `emacs`, type `esc-X Mathematica`.

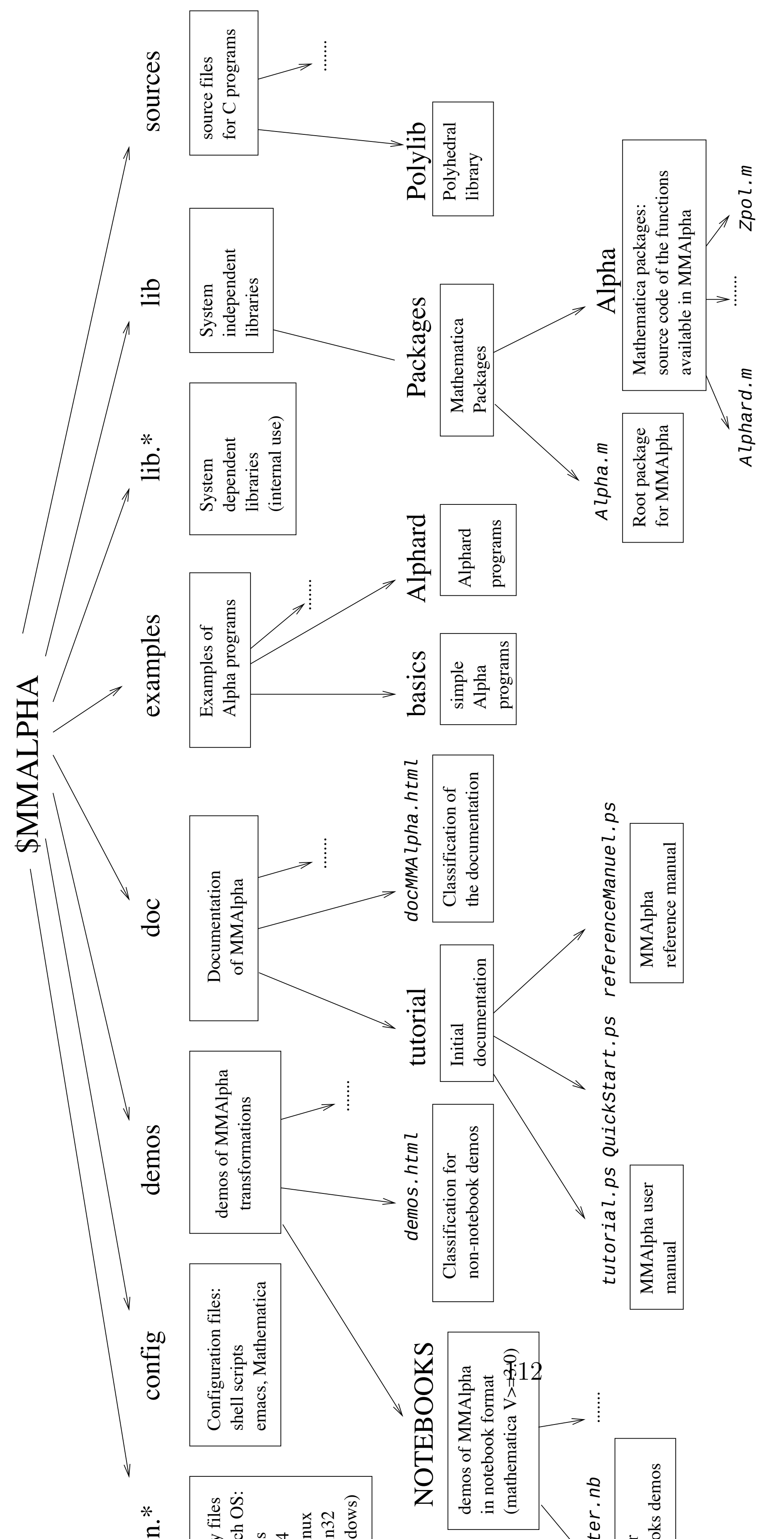
- type math (use the Mathematica textual interface).

- type mathematica (use the Mathematica textual interface).

When the Mathematica session is started, you can type `goDemo["INIT"]` for the initiation demonstration.

When the Mathematica session is started, you can type `on[]` or `start[]` for the initiation demonstration.

C.2 MmAlpha file hierarchy



Index

Bibliography

[AC97a] Api-Cosi. *MMAlpha Reference Manual*, 1997.

[AC97b] Api-Cosi. *MMAlpha tutorial*, 1997.

[DL] F. Dupont De Dinechin and P. Le Moenner. Zakopane, Poland.

[DQR95] F. D. De Dinechin, P. Quinton, and T. Risset. Structuration of the Alpha language. In W.K Giloi, S. Jahnichen, and B.D. Shriver, editors, *Massively Parallel Programming Models*, pages 18–24. IEEE Computer Society Press, 1995.

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

[Kun82] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[LPR$^+$96] 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.

[Mau89] C. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrone= s.* Thèse de doctorat, Ifsic, Université de Rennes 1, December 1989.

[MC91] G.M. Megson and D. Comish. Systolic algorithm design environments. In *Int. Specialist Seminar on Parallel Digital Processors*, pages 100–104, 1991.

[QD88] P. Quinton and V. Van Dongen. Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic arrays. In K. Bromley et al. eds., editor, *International Conference on Systolic Arrays*, pages 473–482. IEEE Computer Society Press, 1988.

[QR89] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures.* Prentice Hall and Masson, 1989.

[Wil94] D. Wilde. The Alpha language. Technical Report 827, Irisa, Rennes, France, Dec 1994.

[WS94] D. Wilde and O. Sie. Regular array synthesis using Alpha. Technical Report 829, Irisa, Rennes, France, May 1994.