Getting started with ALPHA

Api, then Cosi, then R2D2 and Compsys*

June 2004

Abstract

This document is an introduction to the ALPHA language and to its use for regular program description, transformation, evaluation, and hardware implementation in the MMALPHA software. The ALPHA syntax is presented by means of examples. Basic manipulations of ALPHA programs are first shown. Then hardware generation and several advanced transformations – pipelining, change of basis, substitution, normalization, and scheduling, – are introduced and illustrated. Finally, the use of ALPHA subsystems and the ALPHARD hardware description language are presented.

1 Introduction

This document[1] should be read by all ALPHA beginners. It briefly presents the main features of the ALPHA language and the basic transformations of ALPHA programs which are available in the MMALPHA environment.

MMALPHA is a free software, available under the Gnu Public License. It can be downloaded from the site: `http://www.irisa.fr/cosi/ALPHA` . To be run, it requires Mathematica version 3.0 or later.

If you have any problem while reading this document or while trying the MMALPHA software, please send an e-mail to `alpha@irisa.fr`

What is ALPHA?

ALPHA is a functional data parallel language invented at Irisa in Rennes (France). The first definition of ALPHA was proposed by Mauras [Mau89] in 1989. The original motivation was to provide a language for expressing algorithms in an extended version of the formalism

*Api, Cosi and R2D2 are the names of the research groups that successively hosted research related on ALPHA and MMALPHA at Irisa, Rennes, France. Since 2001, Compsys in ENS Lyon also participates in teh development of MMALPHA.

[1] The source of this document is in:
`$MMALPHA/Mathematica/doc/sources/Quickstart/AlphaStart.tex`
and this file is in
`$MMALPHA/Mathematica/doc/tutorial/AlphaStart.pdf`

of recurrence equations proposed by Karp, Miller and Winograd [KMW67]. Other basic references are [Mol82, Qui84, RF86]. The goal of ALPHA is 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.

MMALPHA is still under development at Irisa and at ENS Lyon (France).

What is ALPHA for?

ALPHA is a research tool for various computer science fields such as functional language semantics, parallelization, code generation, optimization, polyhedral theory, VLSI synthesis, systolic arrays, etc.

One important long term goal of ALPHA is to promote the use of high-level functional languages for the synthesis of (parallel) VLSI architectures. From the short term, Alpha can be useful for:

- Providing a correct recurrence equation specification for a particular algorithm (see section 4 and 5).
- Simulating such a specification (see section 5.6).
- Transforming and simplifying a recurrence equation specification (see sections 5 and 7).
- Computing on convex polyhedra (see also `http://www.irisa.fr/polylib`).
- Scheduling programs and detecting parallelism (see section 5.7).

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

- Generate VHDL from this description.
- Provide a design path from the high-level functional specification of an algorithm to the layout description of a VLSI algorithm which implements it.

The remaining of this document is organized as follows. Section 2 presents the MMALPHA interface; section 3 explains the installation procedure; in section 4, the ALPHA language is briefly presented, while in section 5, basic operations on ALPHA programs are described; hardware generation is presented in section 6; in section 7, other transformations of ALPHA programs are explained, and in section 8, we explain how structured ALPHA programs can be structured and transformed. Finally, section 9 concludes and gives some additional references for further exploration of MMALPHA.

2 How does MMAlpha work?

MMAlpha is written in C and in Mathematica, but a user should only see it through its Mathematica interface.

Mathematica provides an interpreted language with high-level built-in functions for symbolic computations: MMAlpha uses these facilities for transforming Alpha programs. Mathematica embeds also a general-purpose 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 or functions. These commands can be for example: view an Alpha program, check its correctness, generate `C` code to simulate it, generate VHDL code, etc. All these transformations are done on the AST which is stored in a Mathematica variable named `$result`.

Specific `C` functions are used for two purposes: parse and unparse Alpha programs or expressions, and perform computations on polyhedra. All `C` functions are called via Mathematica. Most of them are available separately in the POLYLIB polyhedral library which constitutes the computationnal kernel of the MMAlpha environment.

3 Installing MMAlpha

Before going on, you should install the MMAlpha environment. The installation procedure is explained in the MMAlpha web site (`http://www.irisa.fr/cosi/ALPHA`).

The easiest way to use MMAlpha is to access it through its notebook interface. To do so, type `mathematica` under `Unix`, or start Mathematica 3.0 in the Programs menu of Windows NT[2]. [3]

Once MMAlpha is installed, open a new Mathematica notebook and evaluate the command `start[]` in this notebook[4]. This command opens a demonstration notebook which is part of the MMAlpha distribution. Then open the cell named "Introduction Notebooks"[5], and click on button "Getting-started". This opens a notebook where all examples of this document are shown.

4 The Alpha language

In this section, we present the syntax of the Alpha language.

[2] Any Mathematica version later than 3.0 will work.

[3] One can also access the Mathematica kernel directly. Type `math` under `Unix`, or start Mathematica 3.0 Kernel in the Programs menu of Windows NT. An alternative is to use the Mathematica kernel via `emacs`, but then the demonstration notebooks are not accessible.

[4] Type the command, then press simultaneously the Shift and the Enter keys.

[5] For the reader who is not yet familiar with Mathematica's notebooks, notebooks are organized as a tree of cells. Each cell is delimited by a brackets at its right. Selecting a cell (and all its subcells) is done by clicking once its bracket. To open a cell, double-click the bracket once the cell is selected.

4.1 ALPHA by examples

We introduce here the basic features of the langage on the matrix-vector multiplication example shown in Fig. 1.

```
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 1: ALPHA program describing the matrix vector product

Systems, variables, domains and parameters

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 4.1 below shows the declaration of a variable `a` whose domain is the set of points (i, j) in the triangle $\{0 \leq i \leq j; j \leq 10\}$.

Exemple 4.1

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

In the example of Fig. 1, domains are indexed by a parameter `N` which is declared, together with its domain, right after the system name. The values set of the parameters can be constrained by any kind of affine constraints. In general, parameters allow generic descriptions of algorithms to be specified.

Input, output and local variables

An ALPHA system has input and output variables. Here inputs are `a` and `b`, and output is `c`. It may have local variables, such as `C` here. Local variables are declared after the keyword `var`.

Equations and expressions

Each variable is defined by a unique 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.

Exemple 4.2

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

The ALPHA expression of example 4.2 defines the values of `a[i,0]` to be zero[6] and recursively defines `a` at all other points in its domain. This equation defines a variable `a` such that `a[i,j]=j`.

In Fig. 1, variable `C` is also defined as a recurrence equation, and its value at point (i, j) is $\sum_{j=1}^{j=i} a_{i,j} \times b_j$.

4.2 Array notation and standard notation

The above equations use the so-called *array notation* of ALPHA. The real syntax of ALPHA is sligthly less readable but more consistent and logical from a semantic point of view. To illustrate this, example 4.3 shows the same definition of `a` in standard notation.

Exemple 4.3

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

In such an expression, `a` denotes a variable, i.e., a function whose type is

$$\texttt{integer} \times \texttt{integer} \rightarrow \texttt{integer} \quad .$$

Expression `(i,j->i,j-1)` denotes a dependence function, i.e., the affine function

$$\texttt{integer} \times \texttt{integer} \rightarrow \texttt{integer} \times \texttt{integer}$$

which maps `(i,j)` to `(i,j-1)`. Expression `expr = a.(i,j->i,j-1)` denotes the composition of `a` and the dependency function `(i,j -> i,j-1)`. In other words, `expr[i,j]` has the value `a[i,j-1]` at each point `(i,j)` such that `(i,j-1)` is in the domain of `a`.

[6]Syntactic note: constants are zero dimensional arrays hence the empty brackets in `0[]`.

Similarly `expr2 = 0.(i,j->)` means that `expr2[i,j]` has value `0` for all `(i,j)`. This notation describes the extension (or broadcasting) of the constant `0` to all integral points `(i,j)` of the space.

An expression such as `{i,j | j > 0 } : a.(i,j->i,j-1) + 1.(i,j->)` represents the restriction of expression `a.(i,j->i,j-1) + 1.(i,j->)` to the domain `{i,j | j > 0}` (functions of ALPHA are partial).

Finally, `expr = case exp1; exp2; esac` denotes a case definition, where `exp1` and `exp2` are expressions on disjoint domains.

In summary, any ALPHA expression is either a variable, the composition of a variable and of a dependence function, a restriction, or a case statement[7].

Note that the order of the equations as well as the order of the expressions in a case statement is meaningless as ALPHA is declarative: interchanging the two branches of the case in example 4.2 would define exactly the same value for `a`.

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 any evaluation order must follow is that data dependencies between instances of variables must be respected. In example 4.2, obviously `a[i,j]` must be computed after `a[i,j-1]`.

4.3 More on the syntax of ALPHA

Section 8 described structured ALPHA programs. More details on the syntax of ALPHA can be found in appendix A.

5 Basic operations on ALPHA programs

This section presents the first commands that you should learn in order to deal with ALPHA programs.

Write an ALPHA program (such as the one of Fig. 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 in Mathematica, **view** the program in Mathematica (array notation or standard notation), **save** the program in another file, perform a **static analysis** and **simulate** the program.

All these examples are alse available in the `Getting-started` notebook accessible by the `Master` notebook of MMALPHA (after starting Mathematica, type the Mathematica command `start[]` in your notebook to access the `Master` notebook of MMALPHA.)

If you are not familiar with Mathematica, the Introduction notebooks section of the `Master` notebook points towards a brief introduction to Mathematica.

[7] There exists another construct, the `use` statement, which allows calls to subsystems to be written. This statement is explained in section 8.

5.1 Loading and viewing an ALPHA program

Once Mathematica has been started, commands can be send to the kernel through `Input` cells. The name of the working directory can be printed out by typing:

```
Directory[]
```

If you see that this directory is not the one where you have put `prodVect.alpha`, change it by typing:

```
SetDirectory[ "pathname" ]
```

where `pathname` is the directory that you want to become the current directory of MMALPHA (see the Mathematica Help or type `?Directory`).

You can now **load** the ALPHA program into Mathematica by typing:

```
load["prodVect.alpha"];
```

> Note that most often, it is useful to end MMALPHA commands with a semi-colon. Indeed, MMALPHA commands are Mathematica functions, which return the AST of a transformed ALPHA program. If you forget the `;` symbol, Mathematica just displays the result of the function evaluation, which sometimes may take a few pages...

As a side effect, the `load` command assigns the AST of the parsed program to the global Mathematica variable `$result`. In general, `$result` contains the result of the most recent transformation.

Troubleshooting Syntax errors in ALPHA programs are sometimes difficult to understand. Here is a list of frequent errors.

- In domains, inequalities are separated by semicolons, not colons. For example: `{i | i<10 ; i> 20 }`. Notice that the domain `{i,j | 1< i,j <20 }` represents the domain $\{i, j | 1 \leq i \leq 10 \wedge 1 \leq j \leq 10\}$, as the colon allows inequalities to be factorized.
- Input variables are separated by semicolons.
- All equations (even the last one) are ended by a semicolon.
- All branches of a case (even the last one) are ended by a semicolon.
- In the array notation, indexes of a right-hand side expression should appear in the right-hand side variable. For example, `a[i,j] = b[k]` is wrong, as `k` is not an index of `a`. Notice that indexes represent positions, not absolute names. It is perfectly possible to use different index names in the declaration of a variable, and in its definition, although this is a bad practice...
- Constants, such as `1[]` end up with brackets; boolean constants are `true[]` and `false[]`.
- Restrictions on branches of a case expression start with `{|`.

5.2 Viewing ALPHA programs

You can view the program that has been stored in `$result` by evaluating:

```
ashow[]
```

By default, `ashow` pretty prints the program contained in `$result` in array notation, but more generally, `ashow[ var ]` pretty prints the program contained in the Mathematica variable `var`.

To display a program in standard notation use the `show[]` command.

5.3 On-line documentation and options

All Mathematica functions have an on-line documentation: `?ashow` gives the help on `ashow`. Commands may also have options. Type `Options[ command ]` to list the options of `command` together with their default value.

The documentation of MMALPHA is far from being complete. Documentation notebooks are in directory

```
$MMALPHA/doc/packages/
```

and demos notebooks are available in

```
$MMALPHA/demos/NOTEBOOKS/
```

Documentation notebooks can be accessed using the command `docLink[ "topic" ]`, and demos can be open using `demoLink[ "demo" ]`: both commands paste an active button in the current notebook. Try for example

```
demoLink["Getting-started"]
```

5.4 Saving a program

To save the ALPHA program in a file, use the command `asave` (or `save` which saves in standard notation). For instance:

```
asave["myFile.alpha"]
```

writes the program of Fig. 1 in file `myFile.alpha` in the current directory. This command is useful in order to save the content of `$result` after some transformations.

5.5 Analyzing an ALPHA program

Now that you have loaded an ALPHA program, you can start working on it. Your first action should always be to check it for so-called *static errors* by using the `analyze` command:

```
analyze[]
```

Information about potential errors in the ALPHA program is printed out.

If the analysis is successful, the result is `True`. The static analyzer of ALPHA does essentially two kinds of verifications: it checks the type of all expressions – this is not a fantastic novelty, – but it also checks that variables are defined in any point of their domain definition. This second kind of verification is very powerful, and is much more original.

Notes.

- It is very important that your program passes successfully the `analyze` test. Indeed, some transformations assume that the program is correct.
- Correcting the mistakes of an ALPHA program may sometimes be difficult. Especially hard to fix are errors involving values of the parameters. Do not hesitate to send a program to `alpha@irisa.fr` if you have some trouble: experts will try to help you!
- The `analyze` command allows structured systems to be checked by means of the `recurse` option. See `?recurse`.

Troubleshooting If you use non conventional types, such as signed or unsigned short integer (denoted as `integer[S,5]` for example), `analyze` is unable to check the type correctness of expressions. Use the option `scalarTypeCheck -> False` option to avoid error messages.

More information A notebook giving more details on the static analyzer can be accessed using `demoLink["Static"]`.

5.6 Simulating a program

The second step of your design flow should be a simulation. To do so, you should first schedule the program, then generate a `C` program [8].

First, schedule the program by evaluating the command

```
schedule[]
```

This command should give you a schedule of the program, unless there is a dependence cycle in the program, or the program does not have a unidimensional schedule (see section 5.6 for more information on the scheduler.)

If the scheduler is successful, a `C` program can be generated using `cGen`:

```
cGen["prodVect.c", {"N"-> 10}]
```

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

On `Unix`, `gcc -o prodVect prodVect.c` followed by `./prodVect` will compile, then execute the simulated program in the current directory. By default, this program will prompt you for the values of the input instances.

Notes Options of `cGen` allow various forms of C programs to be generated. See the documentation notebook on code generation (`demoLink["cGen"]`).

[8] Several `C` code generators are available. As of the current revision of this document, the simplest one is `cGen`, and we do not present the other ones here.

Troubleshooting As already mentioned, the scheduled may fail for several reasons.

If your program describes an infinite calculation (for example, a filter), the scheduler will fail to find a schedule, as by default, it tries to minimize the total computation time. You can get a schedule by using the command

```
schedule[ optimizationType -> Null ]
```

Note that infinite calculations can also be represented by ALPHA program where the iteration domain is bounded by a parameter. This avoids the above problem.

Your program may not admit a unidimensional schedule. To check this, call it with the appropriate option

```
schedule[ multiDimensional -> True ];
```

You may then get a result, where variables are ordered with a multi-dimensional time. The `C`-code generator works also with multi-dimensional schedules.

If this is not successful, your program may contain a dependence cycle. Checking this property is undecidable, and therefore, nothing can be done to help you. In general, this happens because the program contains an error, and a careful check of the equations reveals it. (Removing some equations before scheduling may help localizing the fault(s).)

More on `C` program generation See the `demoLink["CodeGen"]` notebook.

5.7 Scheduling

Principles 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 program (LP) and to solve it with a software tool: this may be PiP[FT90], 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 $T_{\mathtt{A}}(i,j)$ of a given variable instance `A[i,j]` is given by an affine function of the indices and parameters:

$$T_{\mathtt{A}}(i,j) = \tau^i_{\mathtt{A}} i + \tau^j_{\mathtt{A}} j + \tau^N_{\mathtt{A}} N + \alpha_{\mathtt{A}}$$

where N is a parameter of the ALPHA program. The coefficients $\tau^j_{\mathtt{A}}$ of this function are (in general) different for each variable in the system.

As already seen, the Mathematica command to schedule a program is:

```
schedule[]
```

By default, it schedules `$result` and the resulting schedule is placed in a global variable named `$schedule`. To visualize the current content of the `$schedule` variable, use the command `showSchedResult[]`. The `schedule` function has many options. Type `Options[schedule]` for further information. Type `schedule[opt1->value1]` to change the default value of a particular option `opt1`.

Refining the schedule Once a monodimensional schedule is found, the scheduler may be used to refine the schedule, in order to obtain a "good" architecture. Two options can be used to do so.

The `addConstraints` option allows one to fix the values of the coefficients of a particular timing-function, or even to set the timing-function of a variable. Say for example, that you want the `a` variable to be available at time $\tau_{\mathtt{a}}(i,j) = i+j$, and the `b` variable to be available at time $\tau_{\mathtt{b}}(i) = i$, then use the command

```
schedule[ addConstraints -> {"Ta[i,j,N]=i+j", "Tb[i,N]=i"} ]
```

where `Ta[i,j,N]=i+j` sets the schedule of `a` and `Tb[i,N]=i` sets the schedule of `b`. Notice that the parameters – here `N` – must be included in the indexes of the timing-functions.

The `addConstraints` option allows also coefficients to be set. Coefficient $\tau^j_{\mathtt{A}}$ of the timing-function of a variable `A` is named `TADj`, and constant $\alpha_{\mathtt{A}}$ is named `CA`. To constrain the scheduler to set the first coefficient of the timing-function of `a` and the constant of the same timing-function to be 1, evaluate:

```
schedule[ addConstraints -> {"TaD1==1", "Ca==1"} ]
```

The `durations` option allows one to fix the number of cycles needed to evaluate a given equation of the ALPHA program. With the current version of the scheduler, the `durations` option allows you to specify how many time steps you allocate to the evaluation of one equation. Say for example that you would like to associate a duration of 7 to the evaluation of `C` and of 19 to the evaluation of `c`. Then you should use

```
schedule[ durations -> {0, 0, 19, 7} ]
```

The list of integer values corresponds to the input variables first, then the output variables (in their order of appearance in the header of the program), then the local variables (in their order of declaration), `{a, b, c, C}` in our example. (Notice that assigning a duration to input variables is meaningless.)

How to schedule Here is a typical approach to schedule a program.

- Use the plain `schedule[]` command first. It will immediately reveal if the program has a unidimensional schedule, in which case, *la vie est belle...*
- Remember that if your program describes an infinite calculation, you should set the `optimizationType` parameter to `Null`, or bound your iteration space in the ALPHA program.
- If this does not provide a schedule, use the multi-dimensional option to find a multi-dimensional schedule. But notice that MMALPHA cannot generate hardware for a multi-dimensional schedule... At least, you will be able to simulate the ALPHA program.
- Once a schedule has been found, refine it by adding constraints to the coefficients.

Troubleshooting The scheduler is probably the most important step of the design flow. As already seen, the scheduler is unable to find out a schedule if the program does not admit a mono-dimensional schedule, or, if the program contains dependence cycles.

Currently, MMALPHA does not generate hardware for multi-dimensional scheduled programs[9].

The scheduler is quite robust, and not too slow. For very big programs, it is better to use structered programs, and to use the structured scheduler (see section 8).

Be careful if you use additional constraints or durations: you may specify constraints that cannot be met by the scheduler...

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

```
$MMALPHA/doc/user/Scheduler_user_manual.ps
```

5.8 Mapping the program to a parallel array

Once a schedule (either mono or multi-dimensional) has been found, the program can be mapped to an architecture using the `appSched` function, for example:

```
appSched[]
```

The plain form rewrites the `$result` program by applying space-time transformations to all variables in such a way that the first component of the index is the time, and the other components represent the coordinates of a processor (also called, the allocation function).

6 Generating Hardware

Once a system is scheduled and allocated, generating hardware can be done in the following way.

First, transform the program into ALPHA0, a subset of the language:

```
toAlpha0v2[];
```

After execution of `toAlpha0v2`, `$result` contains one single program, whose expressions can be interpreted as hardware components.

> It is often a good idea to clean-up a little bit the program after exection of `toAlpha0v2`: first, one remove (some of) the idendity equations using
>
> ```
> removeIdEqus[];
> ```
>
> then, one simplifies the system by the command
>
> ```
> simplifySystem[ alphaFormat -> Alpha0 ]
> ```

[9] This is planned, but not yet realized...

Be careful not to forget the `alphaFormat` option, as otherwise, the simplification would destroy the case expressions which represent multiplexers.

Then rewrite it into AlpHard, using the command:

```
alpha0ToAlphard[];
```

After this command, the program becomes a set of structured systems contained in `$library`, and `$result` contains the main subsystem (see also section 8 for more information on structured systems).

Fix the value of the parameters:

```
fixParameter[ "N", 10 ];
```

Finally, generate VHDL code

```
a2v[]
```

Function `a2v` generates in the current directory one VHDL file for each subsystem of `$library`, except the main subsystem which cannot be translated into VHDL.

To see the VHDL code in the notebook, get as the current system the subsystem that you want to see (`getSystem[ name ]`), then evaluate `showVhdl[]`.

Troubleshooting Functions `toAlpha0v2` and `alpha0ToAlphard` require the program in `$result` to meet very specific constraints, and fail if these constraints are not met.

`toAlpha0v2` requires that `$result` has been scheduled and allocated.

`alpha0ToAlphard` has to be executed after `toAlpha0v2`.

`a2v` will almost allways fail on the main calling system which cannot be translated into VHDL: you can just ignore this error message. An alternative is to remove the calling subsystem of `$library`, before calling `a2v`.

The most common problem with `a2v` is that the parameter values are not fixed: then `a2v` fails to produce the controler and the modules. To see this, just display the current system using `ashow`.

To generate a VHDL file for only one subsystem of the library, get this subsystem as the current system using `getSystem`, then run `a2v[ $result ]`.

7 Other transformations of Alpha programs

In this section we briefly review a few more advanced manipulations of Alpha.

7.1 Pipelining

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

For instance, in the program of Fig. 1, we see (last term in the second branch of the `case` expression) that `b[j]` is used for the computation of `C[i,j]` for all i, $0 \leq i \leq N$. This means that `b[j]` will be broadcasted to all processors computing `C[i,j]`.

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:

```
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, and the second argument is the expression to be pipelined (standard notation is mandatory here) and the last argument indicates the *direction* of the pipeline `(i,j->i+1,j)` as well as the name `B1` of the new variable introduced. After the execution of this command, the program contained in `$result` is the one shown in Fig. 2. Note that the direction of pipeline gives the dependence vector of the pipeline, and not the flow direction of the pipelined variable (see the definition of `B1` in Fig. 2.)

```
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: ALPHA program of Fig. 1 after pipelining of `b` in the definition of `C`

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 given initialization equation (here, `b.(i,j->j)`).

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

Related functions `pipeline`, `pipeInfo`.

Troubleshooting The pipeline transformations are difficult to use, as they are not intuitive.

7.2 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 Fig. 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:

```
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 Fig. 3 (after its normalization).

Notes The `changeOfBasis` transformation can be used to rename the indexes of a variable, for example:

```
changeOfBasis[ "C.(t,p->t,p)" ]
```

It can also be used to place a local variable in a higher-dimensional space. For example

```
changeOfBasis[ "C.(i,j->i,j,1)", {"i","j","k"} ]
```

will add one dimension to the local `C` variable, and let its indexes become `i`, `j` and `k`.

Finally, note that `changeOfBasis` returns an non normalized program.

More information See the `ChangeOfBasis` notebook `docLink["ChangeOfBasis"]`.

7.3 Substitution

Substitution allows a variable occurence to be replaced by its definition. For example:

```
substituteInDef[ "c", "C" ]
```

substitutes the definition of `C` in equation defining `c`.

```
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 3: ALPHA program of Fig. 2 after the change of basis on `C`

Notes. The substitution does not normalize the program.

Troubleshooting. The substitution should normally be restricted by the context domain of the expression, but this is not done... Sometimes, the result is not correct.

More information ... available using `docLink[ "Substitution" ]`.

7.4 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:

```
normalize[];
```

Another useful command is

```
simplifySystem[];
```

which normalizes and simplifies a program.

More information See the `Normalization` notebook (`docLink["Normalization"]`.)

8 Structured ALPHA programs

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

8.1 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.4.

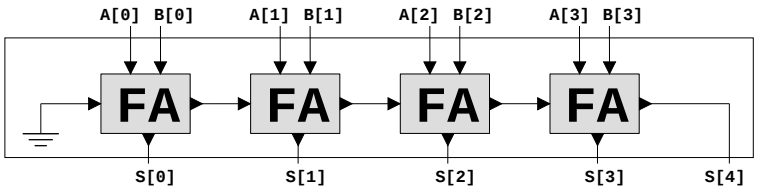

Figure 4: Addition of two integers (coded as bit vectors), using *full adders.*

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;
```

To build an adder using this program, we need to instanciate a collection of such systems, as shown in Fig.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 Fig. 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]);
```

Note. In general, the extension indexes are added *to the left* of the existing indexes. This cannot be seen in this example, since the full adder subsystem has no indexes.

8.2 Handling 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`.

In addition, two functions, `putSystem[]` and `getSystem[]`, may be used to get a system from a library as the current system `$result`, and conversely and to put back a modified system into 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.

The commands to be used are `getSystem[]` and `putSystem[]`.

8.3 Program transformations associated with structures

Most MMAlpha functions handle parameterized programs and *use* statements. There are, however, some major exceptions such as the `cGen` 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.

More on subsystems For more information see the subsystem documentation in file

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

If you are interested in scheduling structured subsystem, see an example in the `More` section of the `Master` notebook.

9 And now?

In this document, we have presented a few possibilities of MMALPHA. You should now know if you are interested in using the MMALPHA software.

If this is the case, you will find in the ALPHA distribution some additional examples. See appendix C.

References

[AC97] Api-Cosi. *MMAlpha Reference Manual*, 1997.

[BQRR98] S. Balev, P. Quinton, S. V. Rajopadhye, and T. Risset. Linear programming models for scheduling systems of affine recurrence equations – a comparative study –. In *10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998.

[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 Conmputer Society Press, 1995.

[dRR97a] F. de Dinechin, T. Risset, and S. Robert. Hierarchical static analysis for improving the complexity of linear algebra algorithms. In *Parallel computing*, 1997.

[DRR97b] Florent Dupont De Dinechin, Sophie Robert, and Tanguy Risset. Structured scheduling of recurrence equations. Technical Report 1140, IRISA, Rennes, France, 1997.

[FT90] Paul Feautrier and Nadia Tawbi. Résolution de systèmes d'inéquations linéaires; mode d'emploi du logiciel pip. Technical Report 90-2, Institut Blaise Pascal, UPMC, Laboratoire MASI, January 1990.

[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.

[Mau89] C. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes 1, IFSIC, December 1989.

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

[QRR96] P. Quinton, S. V. Rajopadhye, and T. Risset. Extension of the ALPHA language to recurrences on sparse periodic domains. In *Int. Conf. on Application Specific Array Processors*, 1996.

[QRR97] P. Quinton, S. V. Rajopadhye, and T. Risset. On manipulating z-polyhedra using a canonical representation. *Parallel Processing Letters*, 7(2):181–194, June 1997.

[Qui84] P. Quinton. Automatic synthesis of systolic arrays from recurrent uniform equations. In *11th Annual Int. Symp. Computer Arch., Ann Arbor*, pages 208–214, June 1984.

[RF86] S.V. Rajopadhye and R.M. Fujimoto. Systolic array synthesis by static analysis of program dependencies. Technical report, University of Oregon, 1986.

A Definition of ALPHA

A.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 Systems

Program stands for a library of ALPHA programs. *PDecl* (or *SystemDecl*) is a single system. *Name* is a system name. *ParamDecl* is the declaration of a parameter domain. *InputDeclList* is the list of input declarations. *OutputDeclList* is the list of output declarations. *LocalDeclList* is the list of local declarations.

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.3 Declarations of variables

VarDeclList stands for a variable declaration. Notice that scalar types have been extended to special hardware types, such as `integer[S,10]` – a signed 10 bit integer – or `integer[U,10]` – an unsigned 10 bit integer.

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

A.4 Domains

Domain stands for a domain declaration. *IndexList* is a list of indexes, and *ConstraintList* a list of constraints.

Notice that operations on domains such as intersection (`&`), union (`|`), complement ( ) and preimage by an affine function are allowed. The parser does the corresponding operation.

I am not sure that the `convex` syntax, nor the domain expressions fully work...

ConstraintList is a list of constraints, each one being an increasing constraint, a decreasing constraint, or an inequality.

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.5 Equations

Equationblock is the block of equation declarations. *Equation* represents an equation, which can be either in array notation (when the lhs has the form `var[...]`) or in standard notation. An equation can also be a `use` statement. [10]

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*

[10] I am not sure of the syntax for the paramAssignation with the dot...

ExtensionDomain :: *Domain*

A.6 Expressions

Expressions can be case expressions, if statements[11], restrictions, affine dependencies, binary operations, unary operations, and reductions[12]

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.7 Dependance Functions and Index Expressions

AffineFunction stands for affine functions.

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

[11] ALPHA conditional statements are strict, that is to say, both branches are evaluated, and moreover, the domain of the statement is the intersection of that of the condition and of the expressions.
[12] Reductions are allowed, but few transformations are currently available.

A.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` | `_`

B Description of the internal format of ASTs

This section describes the format of Abstract Syntax Trees (AST) of ALPHA programs, as handled by Mathematica. In other words, the AST of a program is the Mathematica expression that MMALPHA stores in variable `$result` when the `load` command is executed. It can be displayed just by having Mathematica evaluate the expression `$result`.

In the following description, non terminals are written inside angle brackets `<>`. For readability, keywords are written without the prefix `Alpha‘` which is implicit. For example, the keyword `system` is actually represented by the symbol `Alpha‘system`. The documentation is presented in five sections: Systems, Domains, Equations, Matrices and General.

B.1 Systems

```
<library>       ::= {<system> , ... , <system>}
<system>        ::= system [ <system_id>,
                             <param_space>,
                             <in_var>,
                             <out_var>,
                             <local_var>,
                             <equation_list> ]
<param_space>::= <domain>
<in_var>     ::= <declare_list>
<out_var>    ::= <declare_list>
<local_var>  ::= <declare_list>
<declare>    ::= decl [ <id>, <data_type>, <domain> ]
<data_type>  ::= integer | boolean | real | notype
```

B.2 Domains

The domain specification closely follows the internal format of the domain definition in the domain library. This was done to minimize the overhead of domain storage and of making library calls. In Mathematica, domains should only be changed by making calls to the domain library.

```
<domain>     ::= domain [ <dimension_number>,
                          <id_list>,
                          <polyhedron_list> ]
<polyhedron> ::= pol [ <constraints_number>,
                       <rays_number>,
                       <equations_number>,
                       <lines_number>,
                       <constraint_list>,
                       <ray_list> ]
<constraint> ::= { <const_type>,
                   <number>, ... , <number> }
<const_type> ::= 0 | 1
                 0 / 1 = constraint is equality / inequality
<ray>        ::= { <ray_type>,
                   <number>, ... , <number> }
<ray_type>   ::= 0 | 1
                 0 / 1 = ray is line / ray
```

B.3 Equations

```
<equation>   ::= equation [ <id>, <exp> ]
                 | use [ <id>,
                         <extension>,
                         <param_assign>,
                         <exp_list>,
                         <id_list> ]
<extension>  ::= <domain>
<param_assign> ::= <matrix>

<exp>        ::= var[<id>]
                 | const[<number>] | const[<boolean>] | const[<real>]
                 | binop [ <bop>, <exp>, <exp> ]
                 | unop  [ <uop>, <exp> ]
                 | if [ <exp>, <exp>, <exp> ]
                 | affine [ <exp>, <matrix> ]
                 | restrict [ <domain>, <exp> ]
                 | case [ <exp_list> ]
                 | call [ <id>, <exp_list> ]
                 | reduce [ <casop>, <matrix>, <exp> ]
<bop>        ::= add | sub | mul | div | idiv | mod | min | max
                 | eq | le | lt | gt | ge | ne | or | and | xor
<unop>       ::= neg | not | sqrt
<casop>      ::= add | mul | and | or | xor | min | max
```

B.4 Matrices

```
<matrix>     ::= matrix [ <rows_number>,
                          <cols_number>,
                          <id_list>,
                          { { <number>, <number>, ... , <number> },
                            { <number>, <number>, ... , <number> },
                            ...
                            { <number>, <number>, ... , <number> } } ]
```

B.5 General specifications

Numbers, ids, and lists, as used above, are defined generally (with `<*>` representing any nonterminal).

```
<*_number>   ::= <number>
<*_id>       ::= <id>
<*_list>     ::= { <*>, <*>, ... , <*> }

<number>     ::= [0-9][0-9]* | Infinity
<real>       :=  <number>.<number>
<boolean>    ::= True | False
<id>         ::= "a name"
<comment>    ::= (* blah blah blah *)
```

Reserved Keywords

Alpha‘add	Alpha‘affine	Alpha‘and	Alpha‘binop
Alpha‘boolean	Alpha‘call	Alpha‘case	Alpha‘const
Alpha‘decl	Alpha‘div	Alpha‘domain	Alpha‘eq
Alpha‘equation	Alpha‘ge	Alpha‘gt	Alpha‘idiv
Alpha‘if	Alpha‘integer	Alpha‘le	Alpha‘lt
Alpha‘matrix	Alpha‘max	Alpha‘min	Alpha‘mod
Alpha‘mul	Alpha‘ne	Alpha‘neg	Alpha‘not
Alpha‘notype	Alpha‘or	Alpha‘pol	Alpha‘real
Alpha‘reduce	Alpha‘restrict	Alpha‘sqrt	Alpha‘sub
Alpha‘system	Alpha‘unknown	Alpha‘unop	Alpha‘use
Alpha‘var	Alpha‘xor		

C A brief description of the MMALPHA distribution

C.1 The Master notebook

The master notebook is open by the `start[]` command. It contains 6 sections.

- The welcome section.

- The introduction notebooks: the `getting-started` notebook, and the `mma-intro` notebook.
- The simple examples section. It contains pointers to the matrix-vector, the Fir filter, the Fifo, and the delay line demonstrations.
- The advances examples section. Here are presented a demonstration of the Delayed Least-Mean Square filter, of the structured scheduler, and of the Samba architecture for DNA sequence alignment. [13]
- More contains an access to the Domlib notebook, and some suggestion about the organization of your own notebooks. It explained how you can set the variables `$myNotebooks`, `$myMasterNotebook`, then use the `myStart[]` and the `link` commands to access directly your own working space.
- Finally, the Tests section gives access to a test notebook.

C.2 Documentation

To explore the MMAlpha distribution, you can follow the html files which are in each subdirectory of the distribution. The structure of the distribution is as follows. The main directory is called MMAlpha. It contains another directory called Mathematica which is the MMAlpha distribution properly speaking. In the MMAlpha directory, the file `welcome.html` gives access to the documentation.

The Mathematica directory is organized as follows:

- `bin.cygwin` and `bin.solaris` directories contain the binary files for execution of MMAlpha on the Windows NT and Solaris system respectively.
- `config` contains the configuration files.
- `demos` contains the demonstrations notebooks.
- `doc` contains the documentation notebooks.
- `lib` contains the Mathematica packages which form MMAlpha.
- `sources` contains the source files of the `C` programs and of the latex documentation files.
- `tests` containes the test programs for MMAlpha.

[13] Do the DLMS... Check the structured scheduler...

Index

;, 7

a2v, 13
Abstract Syntax Tree, 3
addConstraints, 11
adder, 17
affine by variable, 10
Alpha0, 12
Alpha0, 13
alpha0ToAlphard, 13
alphaFormat, 13
AlpHard, 13
analyze, 8
appSched, 12
array notation, 5
asave, 8
ashow, 8
assignParameterValue[], 19
AST, 3

binary addition, 17

case expression, 5, 6
case, 5
cGen, 9, 18
change of Basis, 15
changeOfBasis, 15
checking Alpha programs, 8
CodeGen, 10
constant, 6

demoLink, 9
dependence cycle, 10
dependence function, 5
dimension extension, 18
Directory[], 7
domain, 4
durations, 11

equation, 5
expression, 5
extended change of basis, 15
extension domain, 18

filter, 10
fixParameter, 13
flattening a structured program, 19
full adder, 17

genericity, 17
getSystem[], 18
getSystem, 13
The Getting-started notebook, 27
Getting-started notebook, 3

infinite calculation, 10
inlineAll[], 19
inlineSubSystem[], 19
inlining a subsystem, 19
input variable, 4
Installing MMAlpha, 3
Introduction notebooks, 3

$library, 13
library, 18
$library, 18
load, 7, 18
local variable, 4
LP-Solve, 10

Mathematica, 3
The mma-intro notebook, 27
monodimensional, 10
multi-dimensional schedule, 10
multiDimensional, 10

normalization, 16
normalize, 16
notebook interface, 3

on-line documentation, 8
optimizationType, 10, 11
options, 8
output variable, 4

parameter, 9
PiP, 10
pipeAll, 14

Pipelining, 13
POLYLIB, 2
program structures, 17
putSystem[], 18

recurse, 9
removeEqus, 12
restriction, 6
$result, 3, 7, 8, 10, 12–14, 18

$schedule, 10
schedule, 9, 10
Scheduling, 10
SetDirectory, 7
show, 8
showSchedResult, 10
signed integers, 9
simplification, 16
simplifySystem, 13, 16
Simulation, 2
standard input, 9
standard notation, 5
standard output, 9
start[], 3
structured programming, 17
structures of programs, 17
substituteInDef, 15
substitution, 15
syntax of ALPHA, 3
system, 4

POLYLIB, 3
toAlpha0v2, 12

unidimensional schedule, 10
unsigned integers, 9
use statement, 17, 18
use, 17, 18

variable, 4
VHDL, 13

Contents

1 Introduction **1**

2 How does MMAlpha work? **3**

3 Installing MMAlpha **3**

4 The Alpha language **3**
4.1 Alpha by examples 4
4.2 Array notation and standard notation 5
4.3 More on the syntax of Alpha 6

5 Basic operations on Alpha programs **6**
5.1 Loading and viewing an Alpha program 7
5.2 Viewing Alpha programs 8
5.3 On-line documentation and options 8
5.4 Saving a program 8
5.5 Analyzing an Alpha program 8
5.6 Simulating a program 9
5.7 Scheduling 10
5.8 Mapping the program to a parallel array 12

6 Generating Hardware **12**

7 Other transformations of Alpha programs **13**
7.1 Pipelining 13
7.2 Change of basis 15
7.3 Substitution 15
7.4 Normalization 16

8 Structured Alpha programs **17**
8.1 Simple structures 17
8.2 Handling structured programs 18
8.3 Program transformations associated with structures 18

9 And now? **19**

A Definition of Alpha **21**
A.1 Meta Syntax 21
A.2 Systems 21
A.3 Declarations of variables 21
A.4 Domains 22
A.5 Equations 22
A.6 Expressions 23
A.7 Dependance Functions and Index Expressions 23

A.8 Terminals 24

B Description of the internal format of ASTs **24**
B.1 Systems 24
B.2 Domains 24
B.3 Equations 25
B.4 Matrices 26
B.5 General specifications 26

C A brief description of the MMALPHA distribution **26**
C.1 The `Master` notebook 26
C.2 Documentation 27