POLLUX : A LUSTRE based hardware design environment

Frédéric ROCHETEAU
IMAG-LGI, Grenoble
and DEC-PRL, Paris

Nicolas HALBWACHS
IMAG-LGI, Grenoble

April 13, 1994

Abstract

This paper presents a high-level hardware design environment called POLLUX. A design description is written with the LUSTRE data-flow language, and used by the different POLLUX tools to produce the corresponding synchronous circuit or, for example, a simulation program, that can be compiled and executed on a sequential machine.

LUSTRE has been chosen because it is well-suited to hardware description, but it has also been extended to become a full synchronous hardware description language. The first extension allows the description of a floorplan to be included in a LUSTRE program. Other extensions (arrays, recursion or parameterized subprograms) have been introduced in order to make simpler the description of large designs, without loosing the primitive advantages of the language, especially precision and clarity.

The circuits generated by POLLUX are currently used to configure a Programmable Active Memory, where they can be executed.

1 Introduction

The range of applications that can be implemented by hardware have been extended by the generalization of the use of ASICs and the apparition of new kinds of components like programmable gate arrays, which reduce both design time and cost. These solutions, which result from the increasing VLSI chip density, give alternative solutions to custom hardware limited to specific applications.

An other way to reduce the design time is the development of powerful automatic tools (for logic minimization, placement, routing, ...) and very high level CAD systems. These tools have many other advantages and become absolutely necessary since designs are more and more complex; they simplify design descriptions, make them as independent as possible of the technology or limit design errors.

The key element of such a CAD system is the description language, which must be adapted to this use. For POLLUX, we choose the synchronous declarative language LUSTRE [CPHP87, HCRP91]. Its main advantage is its very simple and well defined semantics which allows a very precise, formal and clear hierarchical design description. This point is essential since one of the main reasons to develop hardware is to speed-up applications to obtain better performances. The designer must be able to have a full control on all the implantation parameters (like logic or register generation, floorplan) to obtain a fast enough result.

Other advantages of LUSTRE are:

- The possibility to make many static verifications to detect design errors (e.g., combinatorial loops). This is another consequence of its simple semantics.
- Its pleasant syntax which allows readable programs to be written, in a formalism close to the format of the design specifications.
- The other tools previously developed. LUSTRE was first designed to describe real time applications and an environment for executing LUSTRE programs on sequential machines or proving their correctness was developed. It gives interesting tools for simulation and verification of hardware designs.

Section 2 gives a brief overview of the language LUSTRE, in its basic version. In section 3, we present the extensions added to LUSTRE, in order to make easier circuit description, and section 4 describes the main tools available in POLLUX.

2 Basic LUSTRE

A LUSTRE program describes an operator net controlled by a global clock. When executing, this net receives, at each clock tick, a set of inputs and calculates a set of outputs. The language is based on the perfect synchrony hypothesis which can be found in other languages like ESTEREL [BS91], SML [BC85] or SIGNAL [?]. This means that all the computations or communications take no time and that the net is supposed to react instantaneously and to produce its outputs at the same time it receives its inputs.

This net is described by a system of equations. Its nodes are LUSTRE operators and identifiers associated with its wires are variable names. There is exactly one equation for each local or output variable and equations can be written in any order.

For instance, let us give the LUSTRE program describing a 1-bit adder and the corresponding net (see Fig. 1):

```
node Add1 (a,b,c:bool) returns (s,r:bool);
let
   s = a xor b xor c;
   r = (a and b) or (a and c) or (b and c);
tel;
```

The intuitive semantics of LUSTRE is based on the notion of *stream* which is an infinite sequence of values. Any LUSTRE expression `E` denotes a stream; the nth element E_n of the stream represents the value taken by `E` at the nth clock tick.

An equation "`X=E`" where `X` is a variable and `E` an expression specifies that $\forall i \ X_i = E_i$. Expressions are made of constants, variables and operators. The stream associated with a constant is a constant sequence: for instance `true` represents the stream $(true, true, true, \ldots)$.
There are two kinds of operators:

- The data operators which are the usual operators extended to operate pointwisely on streams. For instance, the following table illustrates the result of the operator "`or`":

`X :`	(	x_0	,	x_1	,	x_2	,	x_3	,	x_4	,	$\ldots$	)
`Y :`	(	y_0	,	y_1	,	y_2	,	y_3	,	y_4	,	$\ldots$	)
`X or Y :`	(	$x_0 \vee y_0$	,	$x_1 \vee y_1$	,	$x_2 \vee y_2$	,	$x_3 \vee y_3$	,	$x_4 \vee y_4$	,	$\ldots$	)

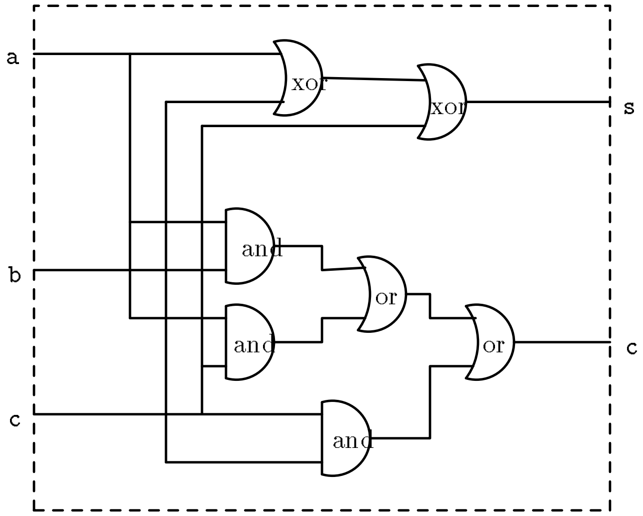

Figure 1: Full adder cell

- The sequence operators are specific. We only present those which will be used in this paper, the initialization operator `->` and the "previous" operator `pre` (see the table below):
  - the stream associated with the expression `E -> F` is the stream associated with `F` where the first element has been replaced by the first element of the stream of `E`. In other words, `E -> F` is always equal to `F`, but at the initial instant, when it is equal to `E`.
  - the nth value of the stream associated with `pre E` is the $(n-1)$th value of the stream of `E`. Its first value is undefined.

`X` :	(	x_0	,	x_1	,	x_2	,	x_3	,	x_4	,	...	)
`Y` :	(	y_0	,	y_1	,	y_2	,	y_3	,	y_4	,	...	)
`X -> Y` :	(	x_0	,	y_1	,	y_2	,	y_3	,	y_4	,	...	)
`pre X` :	(	*nil*	,	x_0	,	x_1	,	x_2	,	x_3	,	...	)

The user can define also its own operators, through the declaration of nodes which are LUSTRE subprograms. For instance, this is a 1-bit incrementer, which calls the one 1-adder node, described above:

```
node Incr1 (incr:  bool) returns (s, c:  bool);
var ps:bool;
let
   (s,c) = Add1(0, ps, incr);
   ps = false -> pre s;
tel;
```

The following table shows an example run of this program:

cycle	0	1	2	3	4	5	6	7	8
incr	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*	*true*
s	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*true*
ps	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*
r	*false*	*false*	*false*	*false*	*false*	*true*	*false*	*false*	*false*

3 Extensions

Using this simple language, one can describe any synchronous circuit. Experiments on small ones show that the language is well-suited to synchronous hardware description. But for describing large circuits, with wide data paths (typically 32 or 64 bits), LUSTRE needs some extensions:

- Structured types and recursion, particularly for describing regular parts.
- Placement equations for expressing circuit floorplanning.
- Parametrized nodes for describing generic circuits, like adders or counters which are widely used, with different sizes or shapes.

These extensions only concern the surface level of the language, since any program written with this extented language can be translated in basic LUSTRE. So, all the standard LUSTRE tools remain available.

3.1 Structured data

There is only one predefined type in the subset of LUSTRE we consider, the type boolean. All the other types must be user-defined, and are derived from the boolean type by tupling.

A tuple `T` with n elements is defined by an expression $\texttt{[E}_0\texttt{, E}_1\texttt{, ..., E}_{n-1}\texttt{]}$, from n expressions $\texttt{E}_0$, $\texttt{E}_1$, ..., $\texttt{E}_{n-1}$ of respective types τ_0, τ_1 ..., τ_{n-1}. Its type is noted $[\tau_0, \tau_1 \ldots, \tau_{n-1}]$. Its elements are indexed from 0 to n-1, the $(i+1)$th is selected by the expression `T[i]`, where `i` is an *index*, i.e., an integer constant known at compile-time. Since elements of a tuple can be themselves tuples, it possible to define hierarchical tuples. In this case, an element at depth k can be selected either directly by an expression $\texttt{T[i}_0\texttt{,i}_1\texttt{,...,i}_{k-1}\texttt{]}$, or by means of k successive selections by the expression $\texttt{T[i}_0\texttt{][i}_1\texttt{]} \ldots \texttt{[i}_{k-1}\texttt{]}$.

For instance, if `T` is the tuple $\texttt{[[e}_0\texttt{, e}_1\texttt{, e}_2\texttt{], [e}_3\texttt{, e}_4\texttt{, e}_5\texttt{, e}_6\texttt{]]}$,

- `T[0]` selects the tuple $\texttt{[e}_0\texttt{, e}_1\texttt{, e}_2\texttt{]}$.
- both `T[1,2]` and `T[1][2]` select the element $\texttt{e}_5$.

A selection can concern a single tuple element, but also a sub-tuple. The expression `T[i..j step s]` defines the tuple `[T[i], T[i+s], ...,T[j]]` of size $\mid j-i \mid /s+1$, where i is the position of the first selected element, j is the last one and s is the distance between two selected elements (if the step s is equal to 1 or -1, it can be omitted). In the case of a hierarchical tuple, the general selection expression is $\texttt{T[i}_0\texttt{..j}_0\texttt{ step s}_0\texttt{,i}_1\texttt{..j}_1\texttt{ step s}_1\texttt{,..., i}_k\texttt{..j}_k\texttt{ step s}_k\texttt{]}$, where $\texttt{i}_p\texttt{..j}_p\texttt{ step s}_p$ specifies the elements selected at depth p. For instance, considering the tuple `T` defined in the previous example, the result of the expression `T[0..1,0..2 step 2]` is the tuple $\texttt{[[}e_0\texttt{, }e_2\texttt{], [}e_3\texttt{, }e_5\texttt{]]}$.

Tuples have been introduced in the language to make simpler the description of regular parts, to group variables computed in the same way but with different data, and to describe them with a single equation. To go further, we consider any operator or node to be polymorphic: it can be applied on all the elements of a tuple. For instance, the `or` operator has the basic signature `bool`$\times$`bool`$\rightarrow$`bool`, but can also be applied to tuple operands; the expression

$[[e_0,\ e_1,\ e_2],\ [e_3,\ e_4,\ e_5,\ e_6]]$ `or` $[[f_0,\ f_1,\ f_2],\ [f_3,\ f_4,\ f_5,\ f_6]]$

stands for

$[[e_0$ `or` $f_0,\ e_1$ `or` $f_1,\ e_2$ `or` $f_2],\ [e_3$ `or` $f_3,\ e_4$ `or` $f_4,\ e_5$ `or` $f_5,\ e_6$ `or` $f_6]]$

A tuple with n elements of same type τ is called an array, its type can be noted $\tau\hat{}n$ instead of $[\tau, \tau \ldots, \tau]$ (This notation refers to cartesian power). All the previous operators defined on tuples can be applied to arrays, together with two specific ones:

- The first one builds an array compound of n equal elements. If `E` is an expression of type τ, the expression `E^n` is the n elements array `[E,E,...,E]`, of type $\tau\hat{}n$.
- The last operator performs a permutation of the indexes of a multi-dimensional array. If `T` is an array of type $\tau\hat{}n_0\hat{}n_1 \ldots \hat{}n_{k-1}$, the expression $\mathtt{T}\{\mathtt{p}_0,\ \ldots,\ \mathtt{p}_{k-1}\}$, where $\{\mathtt{p}_0, \ldots, \mathtt{p}_{k-1}\}$ is a permutation of $\{0, \ldots, k-1\}$, denotes an array of type $\tau\hat{}n_{p_0}\hat{}n_{p_1} \ldots \hat{}n_{p_{k-1}}$, whose element of indexes $[i_{p_0}, i_{p_1},\ \ldots,\ i_{p_{k-1}}]$ is $\mathtt{T}[i_0, i_1,\ \ldots,\ i_{k-1}]$. For instance, if `T` is a two dimensional array considered as a matrix, the transposed matrix is obtained by the expression `T{1,0}`.

As an illustration of the use of tuples and polymorphism, let us describe a 32 bit wide carry ripple adder.

```
node Add32 (A,B:bool^32; ci:bool) returns (S:bool^32; co:bool);
var C:bool^33;
let
   C[0] = ci;
   (S,C[1..32]) = Add1(A,B,C[0..31]);
   co = C[32];
tel;
```

This adder is made of 32 one bit adders. The first and third equations specify respectively that the input carry `ci` is connected to the input of the first full adder cell and the output carry `co` is the output carry of the last one. The second equation describes the connections between the differents cells: the ith cell receives the ith bit of the operands `A` and `B`, and the output carry of the $(i-1)$th cell, and produces the ith bit of the sum and its own carry.

3.2 Floorplanning

Hardware designs are often developed to obtain a fast implementation of an algorithm. A part of the design process consists in reducing as much as possible the critical path, by reducing the number of consecutive combinatorial gates and the length of the connections. This last work can only be done if it is possible, not only to specify the circuit behavior, but also to control the design floorplan. A total or partial floorplan definition can also be useful with an automatic placer, to simplify its task and help it to find the best result.

In Lustre, the floorplan description consists in placing some (or all) variables on an infinite 2D grid. Their position is given by a second set of equations which can have two forms:

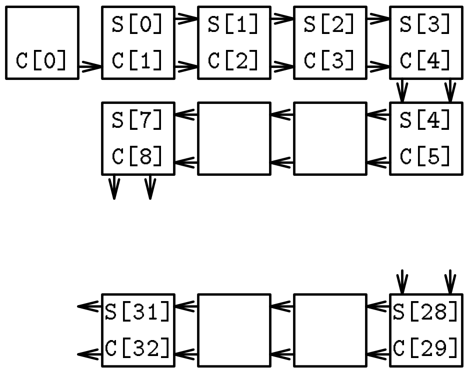

Figure 2: 32-bits adder floorplan

- The equation "`X at (a,b)`" defines (a, b) to be the absolute position of `X` on the grid.
- The equation "`X at (a,b) from Y`" defines the position of `X` relatively to that of `Y`. If the position of `Y` is (c, d), then the position of `X` is $(a + c, b + d)$.

This second equation system is completely independent from the first one. In the example below, the 32-bits adder is given the floorplan shown on Fig. 2:

```
C[0] at (0,0);
C[1..4] at (1,0) from C[0..3];
C[5..8] at (0,1) from C[4..1];
C[9,32] at (0,2) from C[1..24];
S at (0,0) from C[1..32];
```

3.3 Node extensions

In Lustre, a node can be seen as a new operator added to the language, but also as a full program which can be compiled alone. This property is very useful for debugging, but it forbids to write general nodes which are independent of the size of their operands, their placement ....

So, we add the possibility to define parameterized nodes. A node parameter is a particular input which can be evaluated at compile time for all instanciations of the node. For this reason, a parameterized node cannot be compiled alone, since its definition depends on the calling environment. In counterpart, program writting is easier and its possible to build general libraries, integer operators for instance, which is not available in the compiler.

We give below a node describing a n-bits adder, as a generalization of the 32-bits adder:

```
node Add(const n:int; A,B:bool^n; ci:bool) returns (S:bool^n; co:bool);
var C:bool^(n+1);
let
   C[0] = ci;
   (S,C[1..n]) = Add1(A,B,C[0..n-1]);
   co = C[n];
tel;
```

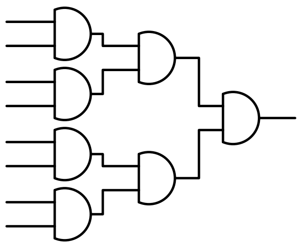

Figure 3: Recursive node instanciation

Here is an instanciation of this node as a 32-bits adder:

```
node AddN32 (A,B:bool^32; ci:bool) returns (S:bool^32; co:bool);
let
   (S,co) = Add(32,A,B,ci);
tel;
```

An other interesting feature of compile-time parameters is the definition of recursive nodes. This was impossible before because one step of the LUSTRE compilation process consists in expanding all the calls, which cannot be done in case of a recursive call. Recursion can be now used through the "`with`": in the expression "`with` e_1 `then` e_2 `else` e_3", the boolean expression e_1 must be evaluable at compile time; if its result is true, the whole expression is replaced by e_2, and otherwise by e_3. This operator can also be used for conditional compilation.

The following recursive node describes a operator performing the conjunction of all the elements of an array:

```
node recand (const n:int;a:bool^n) returns (c:bool);
let
   c = with (n =1) then a
      else recand(n div 2, a[0..(n div 2)-1]) and
            recand((n+1) div 2,a[(n div 2)..(n-1)]);
tel;
```

and the call "`recand(8,A)`", where `A` is of type `bool^8`, defines the net shown in Fig. 3.

4 Design process

We present now the available design tools in POLLUX, through a small example of design. The Bresenham's raster line drawing algorithm is implemented on software by the following program (written in Pascal-like syntax):

```
proc Bresenham (x,y,dx,dy:int)
var Error,i:int;
   Error := 2*dy -dx; i := dx;
   Plot (x,y);
   while (i>0)
      if (Error>=0) then
         Error := Error - 2*dx; y := y+1;
      endif;
      x := x+1; Error := Error + 2*dy;
      Plot (x,y); i := i-1;
   endwhile;
endproc;
```

This algorithm finds a sequence of points that is a good discrete representation of a line, between the points (x, y) and $(x + dx, y + dy)$. This is only the basic algorithm so the parameters must verify that $dx >= dy >= 0$. Each step of the iteration finds of one of these points, by looking at the position of the preceding one relatively to the ideal line. If the i^{th} point (x+i-1,y+k) is above the ideal line (Error<0) then the $i + 1^{th}$ is (x+i,y+k) and (x+i,y+k+1) else (Error>=0). the first point is (x,y).

Translated in LUSTRE, this program becomes:

```
node bresenham (Reset:bool;x,y,dx,dy:int)
returns (Drawing:bool; xc,yc:int);
var Error, i :int;
let
   Error = if Reset then 2*dy - dx
         else if pre(Error)>=0 then pre(Error) - 2*dx
         else pre(e) + 2*dy;
   i = if Reset then dx else pre(i) - 1;
   xc = if Reset then x else pre(xc)+1;
   yc = if Reset then y else if pre(Error)>=0 then pre(y)+1 else pre(y);
   Drawing = (i>0);
tel;
```

Its behavior is the following: Every time the `Reset` input is true, the parameters `x1,y1,dx,dy` are loaded and the program begins to draw a line between the points (x, y) and $(x + dx, y + dy)$. The `Drawing` output becomes true from this instant until the end of the line drawing. The other outputs `xc` and `yc` give at these instants one of the points composing the line. If *Reset* is set before the end of a line drawing, it is stopped and a new one begins.

To develop an efficient hardware implementation, we translate this program in boolean LUSTRE, and modify the algorithm to reduce the critical path and the number of adders. The new description is (henceforth, we use 0 and 1 to denote the boolean values *false* and *true*):

```
const size = 8; xsize = size+2;
type tni = bool^size; xtni = bool^xsize;

node bresenham (Reset:bool;x1,y1,dx,dy:tni)
returns (Drawing:bool; xc,yc:tni);
var   Error,OldError:xtni;
      Val,ErrorInc,ErrorDec,Left,Right,xdx,xdy:xtni;
      Foo1,Foo2:bool;
let
-- Initializations
   xdx = Register(xsize,Reset,Extend(size, xsize, dx));
   xdy = Register(xsize,Reset,Extend(size, xsize, dy));
   ErrorDec = RegisterB (xsize,Reset,Val);
   (Val,Foo2) = Sub(xsize, Mult2(xsize, xdy), Mult2(xsize, xdx), 0);
   ErrorInc = RegisterB(xsize,Reset,Mult2(xsize, xdy));
-- Error
   (Error, Foo1) = Add (xsize, Left, Right, Reset);
   Left = if Reset^xsize then Mult2(xsize, xdy) else OldError;
   Right = if Reset^xsize then not xdx
            else if (OldError[xsize-1])^xsize
               then ErrorInc else ErrorDec;
   OldError = Previous (Error);
-- Outputs
   xc = Counter(size,Reset,1,x1);
   yc = Counter(size,Reset,not (OldError[xsize-1]),y1);
   Drawing = Delay(size, Reset, dx);
tel;
```

The other node are described at the end of the paper.

The constant size is the numbers of bits necessary to represent the points. To avoid overflows, Error needs size+2 bits to be calculated. We are now going to present the role of the POLLUX main tools in the design process.

4.1 Simulation

The first one produces a simulation program that can be used to make a functional debugging of the description. This program is composed of a single loop, the ith iteration corresponding to the ith cycle of the LUSTRE program:

- it reads the new value of the inputs.
- it computes the current value of all the variables.
- it writes the value of the outputs.

For instance, we want to observe the 7 first cycles of the algorithm, when it has to draw a line between the points $(1, 0)$ and $(6, 2)$ (We take `size=3`). The corresponding input data are:

cycle	0	1	2	3	4	5	6
Reset	1	0	0	0	0	0	0
`x1[0]`	1	0	0	0	0	0	0
`x1[1]`	0	0	0	0	0	0	0
`x1[2]`	0	0	0	0	0	0	0
`y1[0]`	0	0	0	0	0	0	0
`y1[1]`	0	0	0	0	0	0	0
`y1[2]`	0	0	0	0	0	0	0
`dx[0]`	1	0	0	0	0	0	0
`dx[1]`	0	0	0	0	0	0	0
`dx[2]`	1	0	0	0	0	0	0
`dy[0]`	0	0	0	0	0	0	0
`dy[1]`	1	0	0	0	0	0	0
`dy[2]`	0	0	0	0	0	0	0

and the simulation produces the following results:

cycle	0	1	2	3	4	5	6
`Drawing`	1	1	1	1	1	1	0
`xc[0]`	1	0	1	0	1	0	1
`xc[1]`	0	1	1	0	0	1	1
`xc[2]`	0	0	0	1	1	1	1
`yc[0]`	0	0	1	1	0	0	0
`yc[1]`	0	0	0	0	1	1	1
`yc[2]`	0	0	0	0	0	0	0

To detect design errors during simulation, it is also possible to add assertions to the equation system, which are constraints on the program or on its environment. An assertion is a Lustre boolean expression, whose value must be true at each cycle of the program. This is dynamically checked by the simulator. For instance, the algorithm we use can draw a line only in the first octant, this means that xdx must be always greater than xdy. This can be specified by the assertion "`assert GT(xsize,xdx,xdy)`", where `GT` is the following Lustre node, receiving two n-bits boolean arrays `a` and `b`, considered as integers, and returning true if and only if `a` is greater than `b`.

```
node GT (const n:int; a,b:bool^n) returns (gt:bool);
var s:bool^n;
let
   (s,gt) = Add(n, a, not b, 1);
tel;
```

Lustre is a highly specialized language, not adapted to describe everything. However, it can be interfaced with more general languages like C. A Lustre program can import types (considered as new atomic types), constants and functions (considered as new data operators). This is particularly useful in simulation, for instance to simulate the environment and to interpret the results. Coming back to our example, the following node effectively draws the lines on a screen:

```
node Env (reset:bool;x1,y1,dx,dy:tni) returns (ok:bool);
var drawing:bool; xc,yc:tni;
let
   (drawing, xc, yc) = Bresenham(reset, x1, y1, dx, dy);
   (win,ok) = if drawing then Plot (OpenWin()-> pre win, xc, yc)
                 else OpenWin()->pre win;
tel;
```

4.2 Circuit generation

The Pollux main tool is devoted to the generation of synchronous circuits. This is realized in two steps. The first step transforms the input Lustre program into a purely boolean Lustre program which contains only boolean variables, `pre` and `->` operators. This is done by expanding nodes and arrays. The second step is then straightforward: the synchronous circuit description — a set of combinatorial gates and registers, and optionally their placement — is obtained by replacing all the Lustre operators by their implementation.

First, an expression containing only boolean operators is implemented by a combinatorial gate. An expression `pre E` is implemented by a synchronous flip-flop and an expression `E->F` is replaced by `(first and E) or (not first and F)`. To implement this second operator we have introduced a special signal, `first`, true at the first instant and always false after. It is typically a reset signal.

The circuits produced by Pollux are implemented on the Pam [BRV90] a machine developed in Dec-prl for fast hardware prototyping. The Pam is a matrix of one bit processors together with an interconnection network, both being configurable by a bitstream loaded at the power-up. This machine is an ideal support for the test and temporal debug of Pollux generated circuits. It is very flexible, any modification or correction can be performed in few minutes by generating a new bitstream and resetting the Pam to load it. It gives also a fast implementation of the designs, their clock frequency is typically between 50 and 200 ns.

The actual implementation of the Pam is the Perle family. The Perle0 prototype is a 5×5 matrix of Xilinx's programmable gate arrays, called Logic Cell Arrays (Lca), which provides 3000 1-bit processors. A circuit produced by Polluxis used by the tools developed at Dec-prl and then by the Xilinx tools which finish placement and routing before producing the configuration bitstream (in fact one bitstream for each Lca).

If we compile our example with the operators described in the appendix we obtain a hardware design that have a clock cycle of about 120ns. This result can be reduced by several means at the Lustre level to obtain a clock cycle of about 50ns by

- using better operators like carry look ahead adder.
- using retiming techniques to rewrite the description and reduce the critical path.
- specifying placement to reduce the length of critical connections like carries.
- taking into account in the Lustre description the characteristics of the Pam. For instance, a Pam cell can realize any function that have less than five inputs, so we try, in our Lustre description, to write logical functions taking into account this constraint.

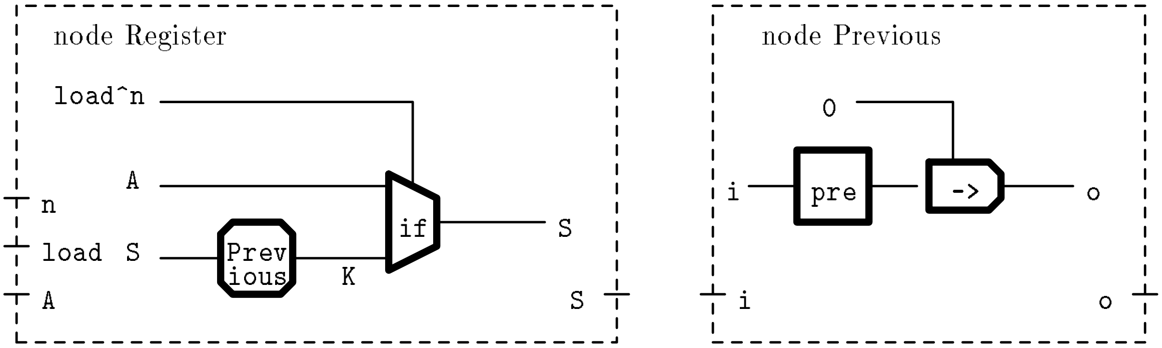

Figure 4: Schema for the node `Register`

- Using low level libraries, for instance making use of a fast routing channel or of registers with enable. At the LUSTRE level, these elements are considerd as external nodes, althougth they can be specified in LUSTRE for simulation.

The hardware implementation of ESTEREL on the PAM is also being considered [Ber91]. Experiments show that, like in software, LUSTRE and ESTEREL are complementary: ESTEREL is particularly well-suited for describing control parts, while LUSTRE is adapted to data-paths.

4.3 Schemas

If text permits a concise description of complex circuits, graphic is a better representation for small ones, particularly when placement is considered. LUSTRE being a data-flow language, every LUSTRE program is also an operator net, which provides a simple and natural graphic syntax. The graphic representation of a LUSTRE program is a set of schemas which are interpreted as follow:

- Each node is enclosed by a dashed line and its name is a text inside which begins by the keyword `node`.
- The data go from left to right, inputs wires to an operator come from its left, while those on its right are its outputs.
- The input or output order is given by their position (the first is the higher).

For instance, Fig. 4 is the result drawn by POLLUX from the node Register. One can edit and modify a schema. For instance, the schema of Fig. 4 can be modified into the one shown by Fig. 5.

These schemas can be then interpreted to produce the modified textual program:

```
node Previous (i:bool)
returns (o:bool);
let
   o = 0 -> pre i;
tel;
```

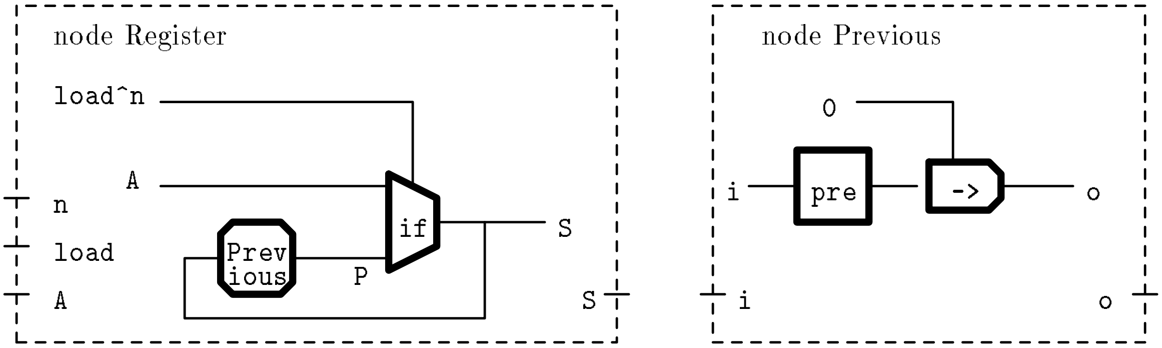

Figure 5: Modified schema for the node `Register`

```
node Register (const n:int;load:bool;A:bool^n)
returns (S:bool^n);
var P:bool^n;
let
   S = if load^n
         then A
         else P;
   P = Previous(S);
   P at (0,0) from S;
tel;
```

5 Conclusion

We have developed an extension of Lustre and a set of tools to provide an environment to simplify and speed-up synchronous hardware design.

A prototype of Pollux have been achieved and integrated in the Pam CAD system, where it is used to developp real applications. We envisage now to enrich it with other post-processors, to interface it with other tools, for logic minimization for instance, or CAD system, like model [GBR84].

References

[BC85] M. C. Browne and E. M. Clarke. Sml — a high-level language for the design and verification of finite state machines. Research Report CMU-CS-85-179, Carnegie Mellon University, 1985.

[Ber91] G. Berry. A hardware implementation of pure Esterel. In *ACM Workshop on Formal Methods in VLSI Design, Miami*, January 1991.

[BRV90] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J. McWhirter, and E. Swartzlander, editors, *Systolic Array Processors*. Prentice-Hall, 1990.

[BS91] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, Munchen*, January 1987.

[GBR84] J.P. Gray, I. Buchanan, and P.S. Robertson. Desining gate arrays using a silicon compiler. In *19th Design Automation Conference*, number 24.3, pages 377–383, 1984.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

Appendix: some nodes of general usage

```
node Add1 (a,b,c:bool)
returns (s,r:bool);
let
   s = a xor b xor c;
   r = (a and b) or (a and c) or (b and c);
   r at (0,0) from s;
tel;


node Add (const n :  int; A,B:bool^n; ci:bool)
returns (S:bool^n; co:bool);
var C:bool^(n+1);
let
   C[0] = ci;
   (S,C[1..n]) = Add1(A,B,C[0..n-1]);
   co = C[n];
tel;


node Sub (const n :  int; A,B:bool^n; ci:bool)
returns (S:bool^n; co:bool);
let
   (S,co) = Add(n, A, not B, not ci);
tel;


node Mult2 (const n:  int; A: bool^n)
returns (S:bool^n);
let
   S[0] = 0;
   S[1..n-1] = A[0..n-2];
tel;
```

```
node Extend (const n,m:int;A:bool^n)
returns (S:bool^m);
-- m > n
let
   S[0..n-1] = A;
   S[n..m-1] = 0^(m-n);
tel;

node Previous (i:bool)
returns (o:bool);
let
   o = 0 -> pre i;
tel;

node Register (const n:int;load:bool;A:bool^n)
returns (S:bool^n);
var K:bool^n;
let
   S = if load^n then A else K;
   K = Previous(S);
   K at (0,0) from S;
tel;

node RegisterB (const n:int;load:bool;A:bool^n)
returns (S:bool^n);
let
   S = Previous(if load^n then A else S);
tel;

node Counter (const n:int;load,incr:bool;A:bool^n)
returns (S:bool^n);
var c:bool; K,P:bool^n;
let
   (P,c) = Add(n,0^n,K,incr);
   S = if (load^n) then A else P;
   K = Previous (S);
   K at (0,0) from P;
   P at (MUN,0) from S;
tel;
```

```
node Delay (const n:int;load:bool;A:bool^n)
returns (counting:bool);
var K,P,S:bool^n; c:bool;
let
   (P,c) = Add(n,S, 1^n, 0);
   S = if load^n then A else K;
   counting = load or Previous(c and (counting or load));
   K = Previous (P);
   K at (0,0) from P;
   P at (MUN,0) from S;
   S[n-1] at (0,0) from counting;
   S[0..n-2] at (0,MUN) from S[1..n-1];
tel;
```