Implementing reactive programs on circuits A hardware implementation of LUSTRE

Frédéric Rocheteau
DEC / Paris Research Laboratory

Nicolas Halbwachs
IMAG / LGI - Grenoble

Abstract

Synchronous languages constitute effective tools for programming real-time systems as far as they can be efficiently implemented. Implementing them by hardware is of course a good way for increasing their performances. Moreover, configurable hardware is now available which makes practical such an implementation. This paper describes an implementation of the synchronous declarative language LUSTRE on a "programmable active memory".

Keywords: Reactive systems, synchronous languages, silicon compilation

Contents

1 Introduction

Synchronous programming [BCG87] has been proposed as a paradigm for designing reactive systems. It is an abstract point of view about real-time, which consists of assuming that a program *instantly* reacts to external events (synchrony hypothesis). It allows providing programs with precise, deterministic and machine-independent semantics. Several programming languages have been designed according to this point of view, e.g., STATECHARTS [Har84], ESTEREL [BG85], SML [BC85], SIGNAL [BL90] and LUSTRE [CPHP87].

In practice, an implementation on a given machine satisfies the synchrony hypothesis if the reaction time is always shorter than the minimum delay separating two successive external events. So, the only real-time problem with a synchronous program is to minimize and measure its reaction time. A specific compiling technique has been proposed [BG85] for synchronous languages, which synthesizes the control structure of the program as a finite automaton. This technique has been applied to ESTEREL and LUSTRE and has been shown to produce very efficient sequential code.

In this paper, we consider an other, more radical way for minimizing the reaction time of a synchronous program, which consists of translating it directly into a circuit. Synchronous languages are especially good candidates for such a translation, because usual circuits behave synchronously, from some reasonable level of abstraction (SML was designed as a hardware description language). And among synchronous languages, LUSTRE is perhaps the one for which this translation is the most natural: LUSTRE is a data-flow language, and one goal we had when designing it, was to be able to describe hardware. As a matter of fact, one solution considered for translating ESTEREL into circuits [Ber91] was to translate ESTEREL into LUSTRE.

One can wonder whether the hardware implementation of reactive systems is of general and practical interest, considering the cost of circuit manufacturing. A first answer is that many reactive systems — for instance low level communication protocols — are actually implemented by special purpose circuits. An other answer is provided by *configurable hardware*. The prototype compiler described in this paper configures a *Programmable Active Memory* (PAM [BRV89]), designed in the Paris Research Laboratory of Digital Equipment. By loading a bitstream — an operation performed in about 20 milliseconds — the PAM can be configured into any digital circuit.

The paper is organized as follows: In section 2 we explain the notion of time in synchronous languages, in order to show the importance of minimizing program reaction times. Section 3 recalls the main aspects of LUSTRE and the PAM is briefly presented in Section 4. In Section 5, we show how a boolean LUSTRE program can be translated into a circuit description which is accepted as input by standard CAD tools. Then, we describe some extensions to LUSTRE which are needed for using it as a programming language for the PAM (Section 6). These extensions concern arrays and only affect the surface level of the language.

Throughout the paper, we shall consider a very simple example of real-time program implementing a watchdog.

2 Time in synchronous languages

Let us first recall how synchronous languages pretend to express real-time constraints without making reference to a global physical notion of time. In the synchronous world, the notion of physical metric time is replaced by a simple notion of order and simultaneity between events. The physical time (measured in *seconds*, e.g.) will be considered as an input event, among others, and will not play any privileged role. We say that time is *multiform*. For instance, consider the two following constraints:

"The train must stop within 10 seconds"
"The train must stop within 100 meters"

There is no conceptual difference between them, and there is no reason to express them by means of different primitives, as it would be the case in languages where the metric time has a special status. In a synchronous language, they will be expressed by analogous precedence constraints:

> "The event STOP must precede the 10th (100th) next occurrence of the event SECOND (METER)"

A synchronous program is supposed to receive external events, which can be either simultaneous or ordered. In response to these events, and *simultaneously* with them, it emits output events. When no input event occur, nothing happens in the program. We shall only consider *logical instants*, which are instants when one or several input events occur. Here is an example of behavior of a speed counter; it receives two kinds of events, SECOND and METER, and emit the value of the SPEED synchronously with any occurrence of METER:

Logical instant	1	2	3	4	5	6
Input events	METER	METER	METER	SECOND	METER	METER, SECOND
Output events				SPEED(3)		SPEED(2)

This simple and abstract point of view appeared to be very fruitful for programming real-time systems and for providing languages with clean, machine-independent semantics. Of course, it raises two practical problems:

- How does the interface of a synchronous program proceed, for deciding whether events are simultaneous or ordered?
- How can an actual machine instantly react to unpredictable input events?

Study of the former problem is ongoing; we consider it to be a bit apart from the main research about synchronous languages. In this paper, we shall focus on the later problem. The basic idea is that if the implementation on a given machine behaves "as if" the reactions be instantaneous, the synchrony hypothesis is a valuable and acceptable abstraction. It will be the case, in particular, if the system reacts to any input event before the next event occurs. Notice that, in that sense, the correctness of an implementation is "monotonic": if a program behaves well on a given machine, it will also behave well on a faster machine. Any other assumption than "zero time" about the reaction time of the machine would violate this property.

So, our problem is to minimize and measure the program reaction time, on a given machine. The first attempt to achieve this goal was to generate efficient, linear (i.e., without loop nor recursion) sequential code for program reactions. It has been achieved in the compilers of the languages ESTEREL [BG85] and LUSTRE [CPHP87,HRR91], by static synthesis of the control structure of the code as a finite automaton. A reaction of the program corresponds to a transition of the automaton. More recently [Ber91], a more radical solution has been investigated, which consists of implementing programs by hardware. This is the solution presented here for LUSTRE.

3 Overview of LUSTRE

We don't give here a detailed presentation of the language LUSTRE, which can be found elsewhere [CPHP87]. We only recall the elements which are necessary for understanding the paper.

A LUSTRE program specifies a relation between input and outputs variables. A variable is intended to be a function of time. Time is assimilated to the set of natural numbers. Variables are defined by means of equations: An equation `X=E` , where `E` is a LUSTRE expression, specifies that the variable `X` is always equal to `E`.

Expressions are made of variable identifiers, constants (considered as constant functions), usual arithmetic, boolean and conditional operators (considered as pointwisely applying to functions) and only two specific operators: the "previous" operator and the "followed-by" operator:

- If `E` is an expression denoting the function $\lambda n.e(n)$, then `pre(E)` is an expression denoting the function

  $$\lambda n. \begin{cases} nil & \text{if } n = 0 \\ e(n-1) & \text{if } n > 0 \end{cases}$$

  where *nil* is an undefined value.

- If `E` and `F` are two expressions of the same type, respectively denoting the functions $\lambda n.e(n)$ and $\lambda n.f(n)$, then `E -> F` is an expression denoting the function

  $$\lambda n. \begin{cases} e(n) & \text{if } n = 0 \\ f(n) & \text{if } n > 0 \end{cases}$$

A LUSTRE program is structured into *nodes*: a node is a subprogram specifying a relation between its input and output parameters. This relation is expressed by an unordered set of equations, possibly involving local variables. Once declared, a node may be functionally instantiated in any expression, as a basic operator.
As an illustration, Figure 1 shows a node describing a "watchdog": it receives

- two boolean inputs, `on` and `off`, which control its state: the watchdog is initially inactive, it becomes active whenever the input `on` is true, and becomes inactive whenever the input `off` is true.

- a boolean input `millisecond`, and an integer `delay`.

It returns a boolean output `alarm` which must be true whenever the watchdog remained active during `delay` milliseconds, i.e., during a delay in which `millisecond` has been `delay` times true. Notice that, while it is active, the watchdog can be set again with a new delay.

Figure 2 illustrates the behavior of the program: it shows the sequence of values of the expressions of the program, in response to particular sequences for input parameters. Vertical reading of this table gives the value of each expression at each execution cycle of the program.

```
node WATCHDOG (on, off, millisecond:  bool; delay:  int)
   returns (alarm:  bool);
var active:  bool; remaining:  int;
let
   alarm = active and (remaining = 0);
   active = if on then true
            else if off then false
            else (false -> pre(active));
   remaining = if on then delay
               else if active and millisecond
               then pre(remaining) - 1
               else pre(remaining);
tel;
```

Figure 1: Example of LUSTRE program: A watchdog

cycle nr.	0	1	2	3	4	5	6	7	8	9	10	11	12
`on`	*ff*	*tt*	*ff*	*ff*	*ff*	*tt*	*ff*	*tt*	*ff*	*ff*	*ff*	*ff*	*ff*
`off`	*ff*	*ff*	*ff*	*tt*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*tt*
`delay`	4	4	4	4	4	2	2	3	3	3	3	3	3
`millisecond`	*ff*	*ff*	*tt*	*ff*	*tt*	*ff*	*tt*	*ff*	*tt*	*ff*	*tt*	*tt*	*ff*
`active`	*ff*	*tt*	*tt*	*ff*	*ff*	*tt*	*tt*	*tt*	*tt*	*tt*	*tt*	*tt*	*ff*
`pre(active)`	*nil*	*ff*	*tt*	*tt*	*ff*	*ff*	*tt*	*tt*	*tt*	*tt*	*tt*	*tt*	*tt*
`remaining`	*nil*	4	3	3	3	2	1	3	2	2	1	0	0
`pre(remaining)-1`	*nil*	*nil*	3	2	2	2	1	0	2	1	1	0	-1
`alarm`	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*tt*	*ff*

Figure 2: Behavior of the Watchdog program

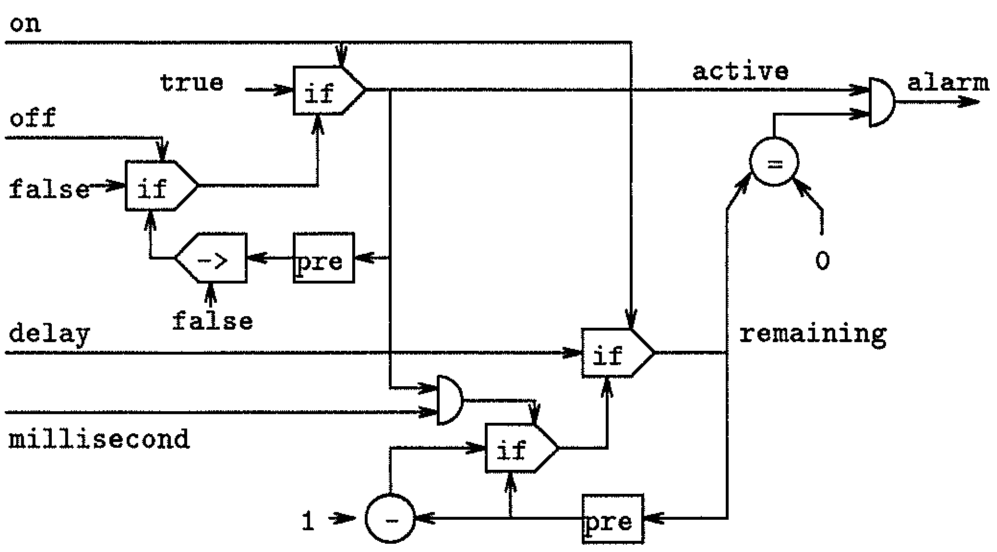


Figure 3: The operator net of the watchdog

LUSTRE programs can be viewed as data-flow operator nets: each variable is a "wire" in the net, and nodes are compound operators. Figure 3 shows the net associated with the program WATCHDOG. This point of view will be the basis of the translation to hardware.

4 Programmable Active Memories

Let us recall the concept of Programmable Active Memory, as defined in [BRV89]:

> *A* PAM *is a uniform array of identical cells all connected in the same repetitive fashion. Each cell, called a* PAB *for programmable active bit, must be general enough so that the following holds true: Any synchronous digital circuit can be realized (through suitable programming) on a large enough* PAM *for a slow enough clock.*

To support intuition, we shall consider a particular architecture, represented in Fig. 4. This particular PAM is a matrix of identical PABs, each of which having (see Fig. 4.a):

- 4 bits of input $< i_0, i_1, i_2, i_3 >$
- 1 bit of output O
- A 1-bit register (flip-flop) with input R and output r, synchronized on the PAM's global clock
- A universal combinatorial gate, with inputs $< i_0, i_1, i_2, i_3, r >$ and outputs $< O, R >$. This gate can be configured into any boolean function with 5 inputs and 2 outputs, by means of $2 \times 2^5 = 64$ control bits, which specify the truth table of the function.

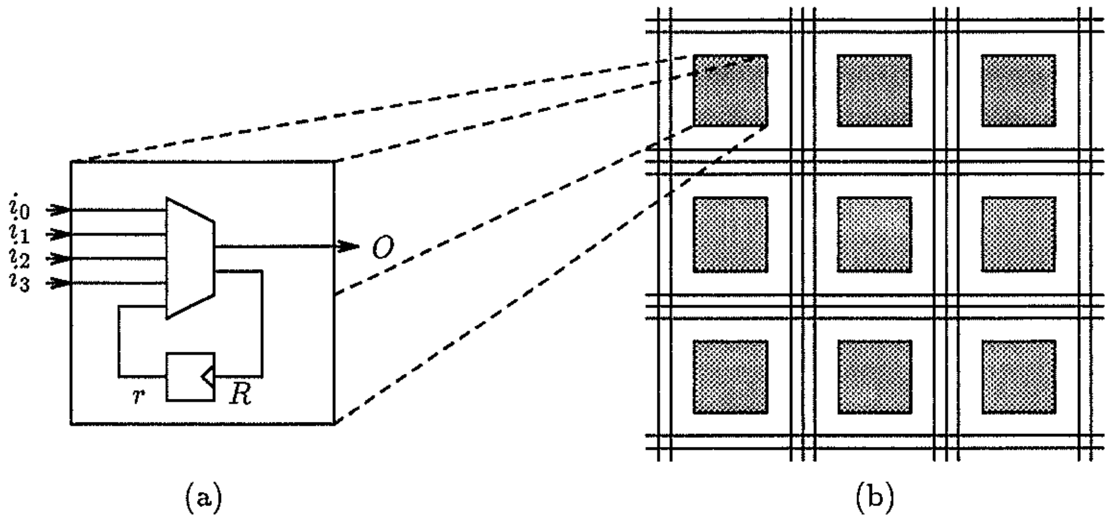


Figure 4: A simple Programmable Active Memory

Between the rows and the columns of cells, there are communication lines (see Fig 4.b) to which the pins of the cells can be connected. These connections and the connections between horizontal and vertical lines can also be configured by means of additional control bits.

Such a PAM, with n active bits, can be configured by downloading a sequence of control bits for configuring the PABs and their connections.

We shall keep this simple model as intuitive support, although the actual target machine of our prototype compiler is slightly more complicated. The target machine is the *Perle* family, studied and built in DEC-PRL, and based on Logic Cell Arrays designed by Xilinx Inc. [Xil88]. The presently available *Perle-0* version is a matrix of 40×80 (double) PABs, and the next version will be about 4 times larger.

Building the control bitstream corresponding to a given circuit configuration is of course a non trivial problem, in spite of available tools. In the case of *Perle*, the standard tools provided by Xilinx, together with the tools developed in DEC-PRL, take as input a logical description of each PAB, together with optional placement indications. They finish the placement, perform automatic routing, and produce the bitstream. Our goal is to translate a LUSTRE program into a description being usable as input of these tools.

5 Implementing boolean LUSTRE on the PAM

We briefly describe the translation of a boolean LUSTRE program into a layout for the PAM (see [Roc89] for more details). It requires

- translating LUSTRE operators in terms of hardware operators (gates, flip-flops)
- implementing the resulting operator net by means of connected PABs

Translation of LUSTRE operators

The first step of the compilation of a boolean program consists of translating its corresponding operator net into a net of gates and flip-flops.

The operator net corresponding to a boolean LUSTRE program contains boolean operators (`or`, `and`, `not`, `=`), conditional (`if_then_else`), and temporal (`pre`, `->`) operators.

Notice that what we call "boolean operators" in LUSTRE are not strictly boolean because of the undefined value *nil*. However, although most of the LUSTRE operators are strict with respect to *nil*, in a legal LUSTRE program, the apparition of a *nil* value may not influence the outputs of the program. This property is checked by the compiler. So, in a legal program we can replace the undefined value by any boolean value without changing the outputs of the program. As a consequence, LUSTRE boolean operators can be straightforwardly translated into gates. The conditional operator can also be translated into a set of gates, using the boolean identity:

```
if A then B else C = (B and A) or (C and not A)
```

The "previous" operator will be obviously implemented by means of a flip-flop. In the technology used, the initial value of flip-flops is 0, so *nil* is considered to be 0. The "followed-by" operator is implemented by means of the *reset* input of the circuit:

```
A -> B = if RESET then A else B = (A and RESET) or (B and not RESET)
```

Example: The definition of the variable `active` of the watchdog

```
active = false -> if on then true
                  else if off then false
                  else pre(active)
```

will be translated into

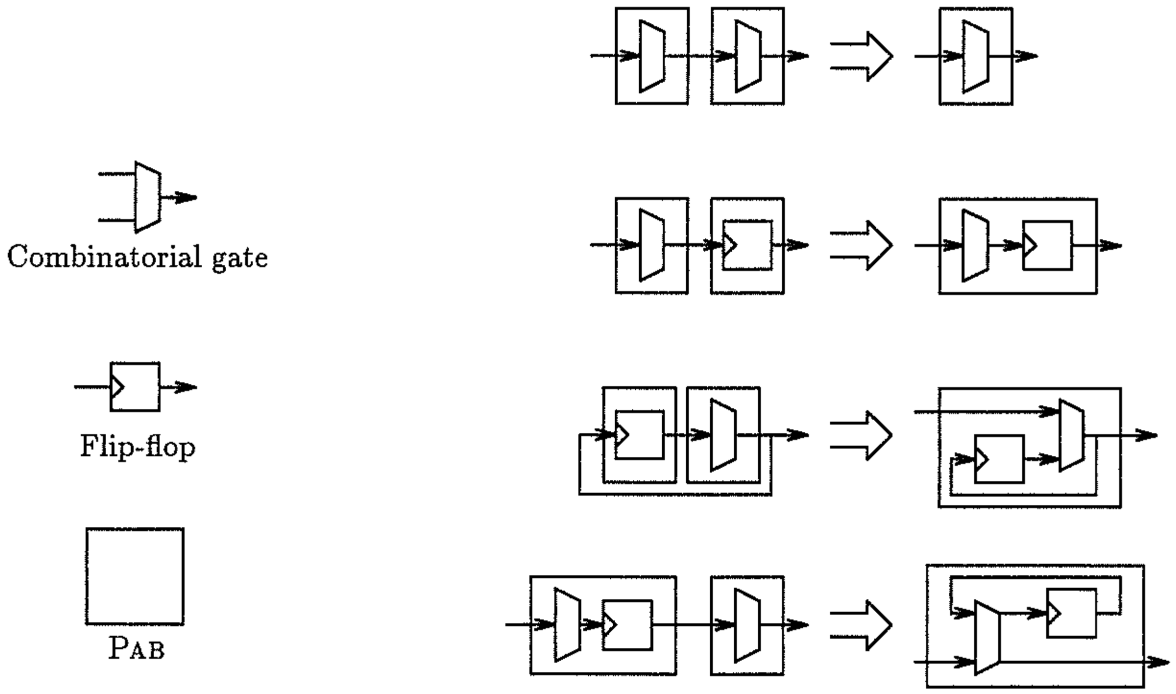
```
active = (false and RESET) or
            (not RESET and ((true and on) or (not on and
            ((false and off) or (not off and FLIP_FLOP(active))))))
```

which, of course, can be simplified into

```
active = not RESET and (on or (not off and FLIP_FLOP(active)))
```

"Packing" operators into PABs

The next task concerns the expression of the resulting net of gates and flip-flops by means of PABs. The simplest way for performing this task consists of using one PAB for each operator in the net. Of course this solution is very unefficient, but we shall use it as a starting point. It is then improved by applying a set of packing rules. Fig. 5.b shows some of these rules, using the notations of Fig. 5.a. The rules are applied according to some



(a) Notations (b) Rules

Figure 5: Some rules for packing operators into PABs

simple heuristics. For instance, the net computing the variable `active` (see Fig. 6) may be packed into one PAB.

6 Extending LUSTRE for programming the PAM

We have shown that the implementation of boolean LUSTRE on the PAM is quite straightforward. If we want to deal with a greatest subset of the language, we have to implement integer variables by vectors of bits. On the other hand, LUSTRE is a good candidate as a high level language for programming the PAM, but lacks some features, concerning regular structures (arrays) and net geometry. In this section, we propose some extensions to the language, which permit

- to deal with a greatest subset of LUSTRE than the purely boolean part. In particular, integers will be considered as vectors of bits.
- to make easier its use for describing circuits. Arrays will be available for describing regular structures. They will also carry placement informations.

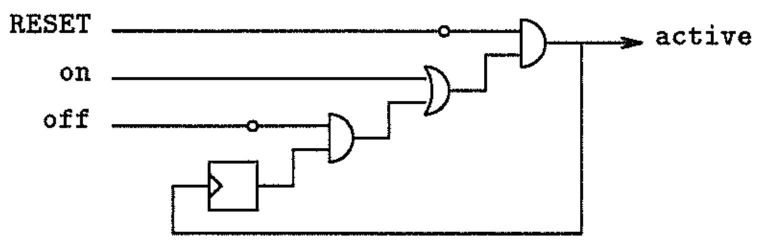


Figure 6: The net computing the variable "`active`"

6.1 Arrays in LUSTRE

Although they were considered in the very first design of the language, arrays have not yet been introduced in LUSTRE, since their translation to sequential code raises difficult problems, concerning the order of computations. These problems disappear when a fully parallel implementation is considered. We propose here a notion of array, compatible with the principles of the language. Introducing arrays will allow integer values to be considered as boolean arrays, with arithmetic operators operating on arrays. Considering a number as e.g., a 32-bit array instead of 32 unrelated boolean variables, is also interesting for placement on the PAM: it strongly suggests to implement it as a register.

LUSTRE contains three predefined data types: boolean, integer and real, and allows abstract data types to be imported from an host language. There is only one way for building compound types, by tupling: if τ_0, τ_1, ..., τ_n are types, so is $[\tau_0, \tau_1, \ldots, \tau_n]$, which is the type of tuples $[X_0, X_1, \ldots, X_n]$ of LUSTRE variables, where X_i is of type τ_i. If `X` is an expression of type tuple and `i` is an integer constant, `X[i]` denotes the (`i` + 1)-th component of `X` (tuple components are numbered from 0).

The notion of array we propose is a special case of tuple: Let us define an *index* to be a non negative integer constant, known at compile time. If τ is a type, and `n` is an index, then τ`^n` is the type of arrays of `n` elements of type τ, numbered from 0 to `n-1` (this notation refers to Cartesian power of τ). An array is a tuple, all components of which have the same type. As a consequence, if `X` is an array of type τ`^n` and `i` is an index, `X[i]` denotes the `i`-th component of `X` (provided `0`$\leq$`i<n`). One can also access a slice of an array: If `X` is as above and `i` and `j` are indexes smaller than `n`, then `X[i..j]` is the array

- `[X[i],X[i+1],...,X[j]]` of type τ`^(j-i+1)` , if `i`$\leq$`j`
- `[X[i],X[i-1],...,X[j]]` of type τ`^(i-j+1)` , otherwise.

If E_1, E_2, ..., E_n are expressions of the same type τ, $[E_1, E_2, \ldots, E_n]$ denotes the array whose i-th component is E_i. By extension, `E^n` denotes the array `[E,E,...,E]`.

Of course, polymorphic LUSTRE operators can be applied to arrays. We introduce also the following notion of polymorphism: any operator op of sort

$$\tau_1 \times \tau_2 \times \ldots \tau_i \rightarrow \tau_1' \times \tau_2' \times \ldots \tau_j'$$

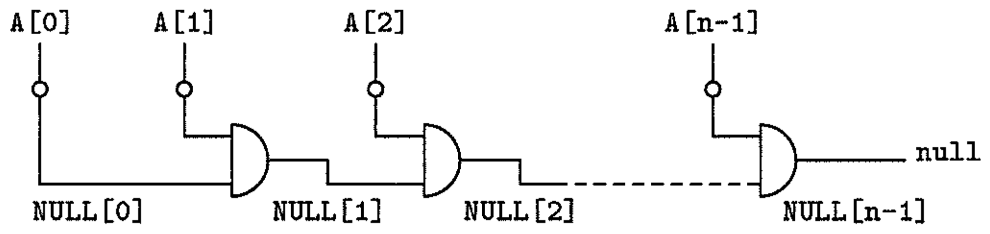


Figure 7: The net of the Zero comparator

(i.e., taking i parameters of respective types $\tau_1, \tau_2, \ldots, \tau_i$ and returning j results of respective types $\tau'_1, \tau'_2, \ldots, \tau'_j$) is implicitly overloaded to have the sort

$$\tau_1\text{\^{}n} \times \tau_2\text{\^{}n} \times \ldots \tau_i\text{\^{}n} \;\rightarrow\; \tau'_1\text{\^{}n} \times \tau'_2\text{\^{}n} \times \ldots \tau'_j\text{\^{}n}$$

for any index n. For instance, the operator and, of sort bool × bool → bool may be applied to two arrays A and B of type bool^n, returning the array C such that C[i] = (A[i] and B[i]), for any i=0...n-1.

6.2 Examples

We shall translate our watchdog program into a boolean program. First, we have to express arithmetic operators as operating on boolean vectors. Let us give a comparator to zero and a combinatorial decrementer:

Zero comparator : It takes a vector of booleans, representing an integer, together with its size, and returns *true* if and only if the represented integer is zero (see the resulting net on Fig. 7):

```
node NULL(const n:  int; A: bool^n) returns(null:  bool);
var NULL: bool^n;
let
   null = NULL[n-1];
   NULL[1..n-1] = NULL[0..n-2] and not A[1..n-1];
   NULL[0] = not A[0];
tel;
```

Combinatorial decrementer: It is made of a general adder:

```
node DECR(const n:  int; A: bool^n) returns (D: bool^n);
var carry_out:  bool;
let
   (S,carry_out) = ADD(n,A,true^n);
tel;
```

The n-bits adder is standard; it is made of n 1-bit adders:

```
node ADD(const n:int;A,B:bool^n) returns (S:bool^n; carry_out:bool);
var CARRY: bool^n+1;
let
   CARRY[0] = false;
   (S,CARRY[1..n]) = AD1(A,B,CARRY[0..n-1]);
   carry_out = CARRY[n];
tel;

node AD1(a,b,carry_in:  bool) returns (s, carry_out:  bool);
let
   s = XOR(a, XOR(b,carry_in));
   carry_out = (a and b) or (b and carry_in) or (carry_in and a);
tel;
```

Full watchdog: Using these boolean implementations of arithmetic operators, the watchdog program can be translated into a boolean program. Here we choose a 8-bits representation of integers:

```
const size = 8;
type Int = bool^size;
node WATCHDOG (on, off, millisecond:  bool; delay:  Int)
   returns (alarm:  bool);
var active:  bool; remaining:  Int;
let
   alarm = active and NULL(size,remaining);
   active = if on then true
            else if off then false else (false->pre(active));
   remaining = if on^size then delay
               else if (active and millisecond)^size
               then DECR(size, pre(remaining))
               else pre(remaining);
tel;
```

The automatic translation of the initial program into this one is not yet implemented. However, our prototype compiler, called POLLUX, translates the above program into the layout (for *Perle-0*) shown in Fig. 8, described in a format that can be provided to standard CAD tools. This layout must be interpreted as follows:

- Cell (a) computes the 4th bit of `remaining-1`, according to the equation

  ```
  D[3] = PR[3] xor 1 xor C[2]
  ```

- Cells (b) and (c) respectively compute the 3rd and 2nd bits of `remaining-1` and the

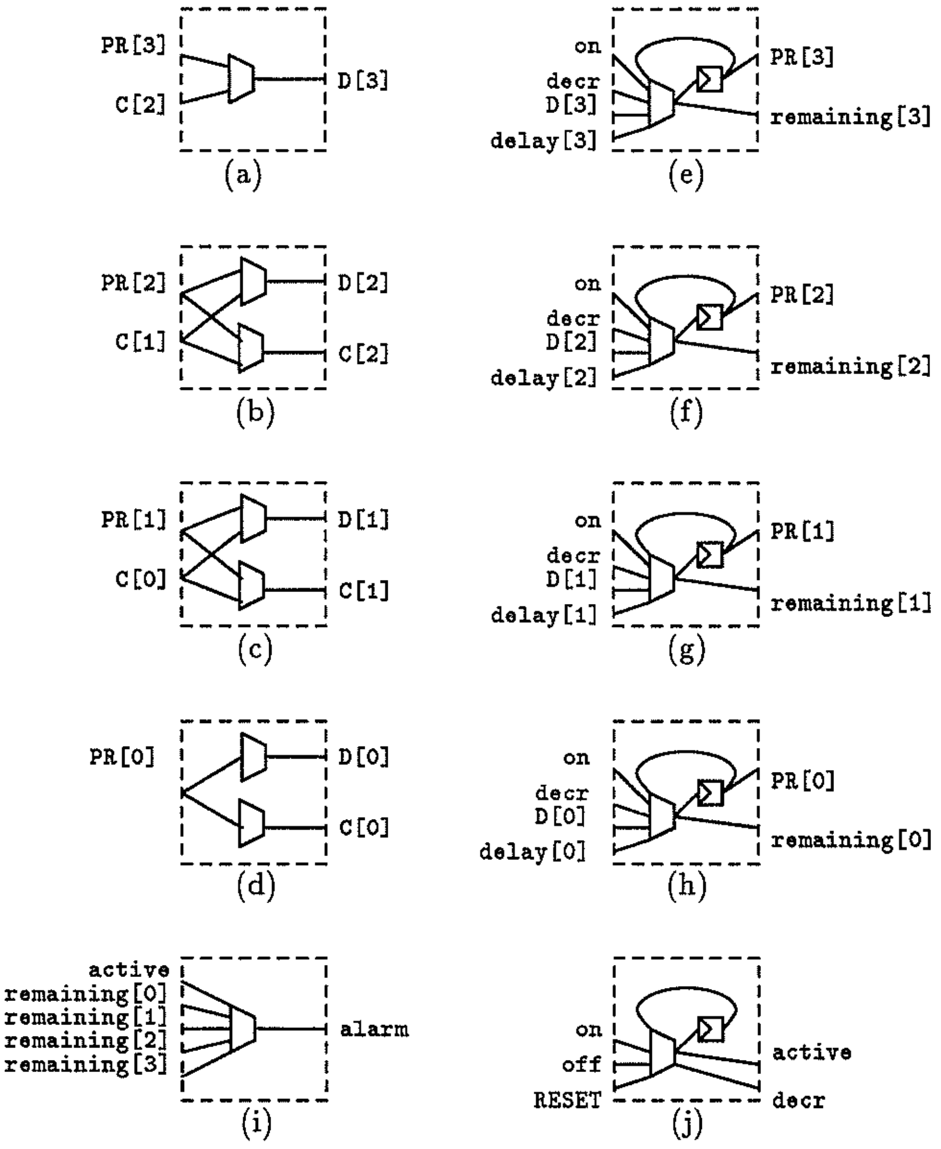


Figure 8: Layout of the Watchdog on Perle-0

corresponding carry, according to the equations

```
D[2] = PR[2] xor 1 xor C[1]
C[2] = PR[2] or C[1]
D[1] = PR[1] xor 1 xor C[0]
C[1] = PR[1] or C[0]
```

- Cell (d) computes its first bit and the first carry

```
D[0] = not PR[0]
C[0] = PR[0]
```

- Cells (e), (f), (g), (h) compute the 4 bits of `remaining` and `pre(remaining)`, according to the equations:

```
PR[i] = Flop(remaining[i])
remaining[i] = (on and delay[i]) or (decr and D[i]) or PR[i]
```

- Cell (i) computes

```
alarm = active and not(remaining[0] or
            remaining[1] or remaining[2] or remaining[3])
```

- Cell (j) computes

```
active = on or (not off and not RESET and Flop(active))
decr = active and millisecond
```

Its critical path is of about 60ns (much less than the time needed by a MC-68000 to perform a "load register" statement!).

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.

[BCG87] G. Berry, P. Couronné, and G. Gonthier. *Synchronous programming of reactive systems, an introduction to* ESTEREL. Technical Report 647, INRIA, 1987.

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

[BG85] G. Berry and G. Gonthier. *The synchronous programming language* ESTEREL, *design, semantics, implementation.* Technical Report 327, INRIA, 1985. to appear in Science of Computer Programming.

[BL90] A. Benveniste and P. LeGuernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, may 1990.

[BRV89] P. Bertin, D. Roncin, and J. Vuillemin. *Introduction to programmable active memories.* Technical Report, Digital Paris Research Laboratory, june 1989.

[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*, january 1987.

[Har84] D. Harel. Statecharts: a visual approach to complex systems. In *Advanced NATO Institute on Logics and Models for Verification and Specification of Concurrent Systems*, 1984.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from dataflow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming,* Passau, august 1991.

[Roc89] F. Rocheteau. *Programmation d'un circuit massivement parallèle à l'aide d'un langage déclaratif synchrone.* Technical Report SPECTRE L10, IMAG, Grenoble, june 1989.

[Xil88] Xilinx, Inc. The programmable gate array data book. 1988. Product Specification.