Report

Lola system notes

Author(s):
Wirth, Niklaus

Publication Date:
1995

Permanent Link:
https://doi.org/10.3929/ethz-a-006651386

Rights / License:
In Copyright - Non-Commercial Use Permitted
Lola System Notes

N. Wirth

Abstract

These notes describe the Lola System, a collection of tools supporting the design of digital circuits. Rather than a manual for the user, they are an explanation of the system’s structure and algorithms, intended as a guide to the implementor of additional tools. Such tools are integrated into the Lola System by their use of the common data structure representing digital circuits.

Contents

0. Introduction
1. The basic data structure
2. Basic operations
3. Generating the data structure by program
4. Generating the data structure through a compiler
5. The compiler’s object code
6. Interpretation of the compiled data structure
7. Circuit simulation
8. Circuit implementation
9. Fuse-map generation for a PLD
10. The GAL circuit board for Ceres and loading the fuse map
11. Converting expressions into normal form
12. References
13. Program Listings

0. Introduction

The Lola System is a toolbox consisting of various modules whose commands serve to specify, implement, and test digital circuits. These notes explain its structure to the user of Lola and to the implementer of additional tools. The system’s base is a module containing the definition of the central data structure used to describe digital circuits. Its name is LSB (Lola System Base). Typically, such a data structure is generated from a Lola text by the compiler, and thereafter used as argument for further processing steps, such as simplification, analysis, comparison, simulation, and layout generation. Fig. 1 gives an overview of the described components and their interdependence.

![Diagram](image)

Fig. 1. Module structure of the Lola system

First we describe the basic data structure representing circuits and serving as the common ground for the various tools. Then follows an explanation of some basic operations on the structure, operations provided in module LSB. A brief chapter shows how the structure can be generated by programs formulated in a programming language such as Oberon. The programs are supported by an appropriate auxiliary module,
A much preferable way is to specify circuits in a suitable hardware description language, such as Lola [1, 2]. This approach requires the availability of a compiler, and has the advantage that the compiler can perform various consistency checks. We refrain from describing the compiler in any details here, and instead concentrate on explaining its output. This is again a data structure representing the source program in a form suitable for processing, namely for expanding it, according to given parameters, into the data structure mentioned above.

The description of the basic data structure is followed by two examples of applications that are based on that data structure. The first application tool is a program to simulate synchronous, digital circuits. We note that this Simulator imports LSB only, as do also the subsequently described tools.

Circuit implementation for a field-programmable gate array (FPGA) is performed "by hand" using a layout editor described in [3, 4]. The connection between the circuit's specification in Lola and the layout is established by a tool called Checker [5]. This program checks the consistency of the presented layout and the Lola specification by comparing the data structure compiled from the Lola program with the data structure extracted from the layout.

An example of a tool for the fully automatic generation of an implementation is module GAL. Its target is the PLD GAL22V10B, i.e. its output is the fuse map for that PLD. The module also contains a command for downloading the generated data. The circuit, in order to be implementable on this device, must obey certain restrictions. In particular, the combinational parts of the circuit must be denoted by expressions in disjunctive normal form. Another tool called DNF serves to convert an arbitrary expression into normal form.

1. The Basic Data Structure

Systems are said to be integrated, if various programs operate on a common data structure. In Oberon [6], the role of programs is taken over by commands, i.e. by procedures declared in various modules. Hence, these modules import data types and the data structure from a common base module. These types define the elements of the shared data structure. Here the base module is called LSB (Lola System Base). Apart from the types, the module also exports a few basic procedures generating elements of the structure or operating on it.

A digital circuit consists of gates and connecting wires. Abstractly it is represented by a graph consisting of nodes (gates) and edges (wires, signals). The pertinent data types are defined as

\[
\text{Signal} = \text{POINTERTO SignalDesc}; \\
\text{SignalDesc} = \text{RECORD} x, y: \text{Signal}; \\
\quad \text{fct, val, u, v: SHORTINT} \\
\quad \text{END}
\]

Attribute \textit{fct} denotes the kind of gate (operator) represented by a node, such as \textit{not}, \textit{and}, or \textit{or}. The attributes \textit{val}, \textit{u}, and \textit{v} are used for various purposes by various commands, and no common meaning is attributed to them by the base module. The following constants are defined in module LSB as values for \textit{fct}. These values mirror the operators available in the language Lola. \textit{sr} stands for set-reset latch, and \textit{tsg} for tri-state gate. The significance of \textit{mux1}, \textit{reg1}, and \textit{link} is explained further below.

\[
\begin{align*}
\text{buf} &= 7 & \text{not} &= 8 & \text{and} &= 9 & \text{or} &= 10 & \text{xor} &= 11 \\
\text{mux} &= 12 & \text{mux1} &= 13 & \text{reg} &= 14 & \text{reg1} &= 15 \\
\text{latch} &= 16 & \text{sr} &= 17 & \text{tsg} &= 18 & \text{link} &= 19
\end{align*}
\]
The leaves of a circuit graph are special nodes carrying a name representing an input signal or input variable:

\[
\begin{align*}
\text{Variable} &= \text{POINTER TO VarDesc;} \\
\text{Name} &= \text{ARRAY 7 OF CHAR;} \\
\text{VarDesc} &= \text{RECORD (SignalDesc)} \\
&\quad \text{name: Name; } \\
&\quad \text{class: SHORTINT; } \\
&\quad \text{next, dsc: Variable}
\end{align*}
\]

An example of a simple circuit is shown in Fig. 2. In addition to its representation by a schematic diagram and by a LSB-data structure its formulation as an expression is also shown. For variables, the (inherited) field \textit{fct} indicates the variable's data type. \textit{BIT}, \textit{TS}, and \textit{OC} stand for binary, tri-state, and open collector respectively. The following constants are defined in module LSB:

\[\begin{align*}
\text{bit} &= 0 \\
\text{ts} &= 1 \\
\text{oc} &= 2 \\
\text{array} &= 4 \\
\text{record} &= 5
\end{align*}\]

Nodes contain fields \textit{x} and \textit{y} for operands, hence represent binary operators. Multiplexers and registers are operators with 3 operands. They are represented by double nodes as shown in Fig. 3.

The field's \textit{next} and \textit{dsc} of the type \textit{VarDesc} serve to represent the set of variables occurring in a circuit. \textit{next} simply is the link in the linear list of variables. If a variable is structured, the field \textit{dsc} is the root of the list of its components. The field \textit{name} of a component specifies the element's index in case of an array, and the component's name in case of a user-declared type (record). The data structure representing the following variables declared in Lola is shown in Fig. 4.
TYPE T;
  IN x: BIT; OUT y: BIT;
BEGIN ...
END T;

VAR u: BIT; v: TS;
a: [4] BIT; t0, t1: T;

The (inherited) field y refers back to the ancestor of a component variable, i.e. \( v.dsc.y = v \). The field is used to reconstruct the variable's full name. The field x is the root of the structure which defines the variable's "value", and the field class specifies the kind of variable: input, output, inout, or local. The latter, and the values of x and y are not shown in Fig. 4.

\[
\begin{align*}
\text{var} & = 0 & \text{in} & = 1 & \text{out} & = 2 & \text{io} & = 3 \\
\end{align*}
\]

The representation of the following assignments is shown in Fig. 5.

VAR x, y, z: BIT; t: TS; u: OC;

\[
\begin{align*}
x & := y + z; \\
t & := x | y; t := z | u; \\
u & := y; u & := z
\end{align*}
\]

The data structure corresponding to the following specification of the element of a binary adder is given in Fig. 6.
IN x, y, ci: BIT;
VAR d, s, co: BIT;

\[
d := x - y; \quad s := d - ci; \quad co := x \times y + d \times ci
\]

Fig. 6. Adder element – Schematic and data structure

A second example is the element of a binary counter shown in Fig. 7.

IN e: BIT; VAR q, co: BIT;

\[
q := \text{REG(clk: '1, q - e); co := q \times e}
\]

Fig. 7. Binary counter element – Schematic and data structure
Module LSB exports four variables: \texttt{org}, \texttt{zero}, \texttt{one}, \texttt{clk}. The first serves as anchor for the data structure to be accessed by various application commands. By convention \texttt{org} points to a composite (record) variable representing the entire circuit. \texttt{org.dsc} then is the root of the list of variables occurring in the circuit. \texttt{zero} and \texttt{one} are "variables" denoting the constants 0 and 1. \texttt{clk} denotes the global clock signal of synchronous circuits.

And finally, there is a global variable \texttt{Log}, a text displayed in a viewer opened when LSB is initialized. \texttt{Log} serves as visible output for the various operators in the Lola system. Two commands operate on the \texttt{Log}-text and its viewer respectively. Text and viewer are opened when the system is started. Command \texttt{OpenLog} serves to open a viewer containing the text, and \texttt{ClearLog} deletes the viewer's contents.

2. Basic Operations

In addition to the data types and global variables, module LSB exports a few procedures. \texttt{New} and \texttt{NewVar} serve to generate elements of the data structure. \texttt{Show} generates a textual expression representing a structure by recursively traversing the data structure. \texttt{WriteName} yields the possibly composite name of a given variable. It is the inverse of \texttt{This(org, name)}, whose value is the component variable of variable \texttt{org} with component name (or index) \texttt{name}. \texttt{Assign} assigns the given pointer to the root variable \texttt{org}.

```pascal
PROCEDURE New(f: SHORTINT; x, y: Signal): Signal;
VAR z: Signal;
BEGIN NEW(z); z.fct := f; z.x := x; z.y := y; RETURN z
END New;

PROCEDURE NewVar(f, val: SHORTINT; x, y: Signal;
    next: Variable; VAR name: ARRAY OF CHAR): Variable;
VAR v: Variable;
BEGIN NEW(v); v.fct := f; v.val := val; v.x := x; v.y := y;
    v.next := next; COPY(name, v.name); RETURN v
END NewVar;

PROCEDURE WriteName(VAR W: Texts.Writer; v: Variable);
BEGIN
    IF v.y # NIL THEN WriteName(W, v.y(Variable)); Texts.WriteString(W, v.name)
    END WriteName;

PROCEDURE This(org: Variable; VAR name: ARRAY OF CHAR): Variable;
VAR v: Variable; i, j: INTEGER;
    id: ARRAY 16 OF CHAR;
BEGIN v := org.dsc; i := 0;
    LOOP j := 0;
        WHILE (name[i] > ' ') & (name[i] # '.') DO id[j] := name[i]; INC(j); INC(i) END ;
        id[j] := 0X;
    WHILE (v # NIL) & (v.name # id) DO v := v.next END ;
    IF name[i] = '.' THEN
        IF (v # NIL) & (v.fct IN {array, record}) THEN v := v.dsc; INC(i)
        ELSE v := NIL; EXIT
    END
    ELSE EXIT
    END;
RETURN v
END This;

PROCEDURE Assign(v: Variable);
BEGIN org := v
END Assign;
```
PROCEDURE ShowTree(x: Signal);
BEGIN
  IF x # NIL THEN
    IF x IS Variable THEN WriteName(W, x(Variable))
    ELSE Texts.Write(W, '(
    ShowTree(x.x); Texts.Write(W, code[x.fct]); ShowTree(x.y); Texts.Write(W, ')')
  END
END
END ShowTree;

PROCEDURE Show(x: Variable; lev: INTEGER);
VAR typ: SHORTINT;
BEGIN typ := x.fct;
  IF typ = record THEN
    x := x.dsc;
    WHILE x # NIL DO Show(x, lev+1); x := x.next END ;
    Texts.Append(Log, W.buf)
  ELSIF typ = array THEN
    x := x.dsc;
    WHILE x # NIL DO Show(x, lev); x := x.next END
  ELSIF typ # integer THEN
    WriteName(W, x);
    IF (lev = 0) & (x.class # var) THEN Texts.Write(W, '*') END ;
    IF x.x # NIL THEN Texts.WriteString(W, ' := '); ShowTree(x.x) END ;
    Texts.WriteLn(W); Texts.Append(Log, W.buf)
  END
END Show;

Apart from these procedures of a rather auxiliary nature, LSB provides two basic operations on circuits, namely that of simplification and of checking for cycles within combinational circuits. Such loops constitute unwanted race conditions. The procedures are listed in the Appendix.

Procedure Simplify(v) traverses the data structure assigned to v trying to apply any of the simplification rules listed below. If v is a structured variable, then all components are traversed. This process is repeated until no more simplifications are applicable. Repetition is necessary, because a simplification resulting in some variable obtaining the constant value 0 or 1 may propagate to other variables. The simplification rules are:

\[
\begin{align*}
\neg\neg x &= x \\
\neg 0 &= 1 \\
\neg 1 &= 0 \\
x + 1 &= 1 \\
1 + x &= 1 \\
x \times 0 &= 0 \\
0 \times x &= 0 \\
x + 0 &= x \\
0 + x &= x \\
x \times 1 &= x \\
1 \times x &= x \\
x - 0 &= x \\
0 - x &= x \\
x - 1 &= \neg x \\
1 - x &= \neg x \\
\text{MUX}(0; x, y) &= x \\
\text{MUX}(1; x, y) &= y \\
\neg x - 1 &= x \\
1 - \neg x &= x
\end{align*}
\]

Procedure Loops(v) also traverses the structure assigned to v, and if v is structured, it traverses the structures of all of its components. We assume that at the outset all nodes are colored black. During the traversal, we may think of all visited black nodes being colored gray. If an encountered node is already gray, then a (combinational) loop is present. When returning to a node from the traversal of its descendants, the node is colored white, and if any white node is encountered during a further traversal, then the node’s descendants are not further followed because they evidently had already been visited. For coloring the node field val is being used.

3. Generating the Data Structure by Program
The most obvious way to generate a data structure is by writing a program and executing it. In order to simplify the task of deriving the program corresponding to a circuit, and to make the program directly mirror the signal equations as closely as possible, a set of auxiliary procedures is introduced, typically contained in a library module. We propose the following set, perhaps to be augmented:

```
PROCEDURE Var(type, class: INTEGER; next: LSB.Variable; name: ARRAY OF CHAR); LSB.Variable;
    VAR v: LSB.Variable;
    BEGIN v := LSB.NewVar(type, 0, NIL, NIL, ancestor, name);
        v.class := class; RETURN v
    END Var;

PROCEDURE And(x, y: LSB.Signal): LSB.Signal;
    BEGIN RETURN LSB.New(LSB.and, x, y)
    END And;

PROCEDURE Or(x, y: LSB.Signal): LSB.Signal;
    BEGIN RETURN LSB.New(LSB.or, x, y)
    END Or;

PROCEDURE Xor(x, y: LSB.Signal): LSB.Signal;
    BEGIN RETURN LSB.New(LSB.xor, x, y)
    END Xor;

PROCEDURE Not(x: LSB.Signal): LSB.Signal;
    BEGIN RETURN LSB.New(LSB.not, NIL, x)
    END Not;

PROCEDURE Reg(en, d: LSB.Signal): LSB.Signal;
    BEGIN RETURN LSB.New(LSB.reg, LSB.clk, LSB.New(LSB.reg1, en, d))
    END Reg;
```

Consider now the previously presented example of an adder element with the definitions

\[ d := x - y; \quad s := d - c_i; \quad c_o := x \cdot y + d \cdot c_i \]

For each signal variable, a program variable with the same name is declared. Then a record is generated by calling `LSB.NewVar`, appropriately linking the generated records together. At last, a single assignment is made to each variable's `x-field defining the circuit for which the variable stands:

```plaintext
VAR x, y, d, s, co: LSB.Variable;
    x := Var(LSB.bit, LSB.in, NIL, 'x');
    y := Var(LSB.bit, LSB.in, x, 'y ');
    ci := Var(LSB.bit, LSB.in, y, 'ci');
    d := Var(LSB.bit, LSB.var, ci, 'd ');
    s := Var(LSB.bit, LSB.var, d, 's ');
    co:= Var(LSB.bit, LSB.var, s, 'co');
    dx := Xor(x, y);
    sx := Xor(d, ci);
    co.x := Or(And(x, y), And(d, ci))
```

The second example is that of the binary counter element, also encountered above:

```plaintext
VAR q, e, co: LSB.Variable;
    q := LSB.NewVar(LSB.bit, LSB.var, NIL, NIL, NIL, 'q ');
    e := LSB.NewVar(LSB.bit, LSB.in, NIL, NIL, q, 'e ');
    co := LSB.NewVar(LSB.bit, LSB.var, NIL, NIL, e, 'co');
    qx := Reg(LSB.one, Xor(q, e));
    co.x := And(q, e)
```

Note that the assignments representing the allocation of variables must precede the assignments of the expressions, which directly appear in prefix form, and thereby make the program intuitively rather obvious. Generating data structures by such programs more or less directly follows a fixed recipe, and it
has the great advantage that all facilities offered by a programming language are at our disposal. We particularly refer to repetitive and conditional statements to be used for generating replications of a circuit pattern. The similarities between circuit and program structures become quite evident.

4. Generating the Data Structure through a Compiler

Apart from the mentioned advantages, the method of generating the data structure from a program also has severe shortcomings. This is so even if the programming language allows to express the signal definitions in infix form through the use of operator overloading. The primary shortcoming is the lack of any sort of consistency checking.

1. No check whether all variables (except inputs) have been assigned an expression.
2. No check against redefinitions (multiple assignments).
3. No check for signal types, such as bus types allowing multiple assignments.
4. No check for correct structuring (arrays, records).

The remedy lies in the use of a notation tailored to the description of circuits, containing sufficient redundancy to allow these checks to be made mechanically by a compiler. Typically, such a compiler generates some code. The interpretation of this code, called the execution of the (compiled) program, then generates the desired data structure. We have postulated a simple notation for this purpose, called Lola for Logic Language [1, 2]. Its most noteworthy characteristic is that it allows to specify circuit types. It is possible to generate (declare) instances (variables) of such (structured) types. Furthermore, the types themselves can be parametrized.

A detailed description of the Lola compiler is beyond the scope of this text. Suffice it to say that our compiler follows the simple principle of recursive descent. We rather concentrate our attention to a specification of the generated code. The interface definition is the following, where procedure Module compiles a Lola text.

```
DEFINITION LSC;
  IMPORT Texts;
  (*for constant and type declarations, see below*)
  VAR guard, globalScope, localScope: Object; body: Item;
  PROCEDURE Module(T: Texts.Text; pos: LONGINT);
END LSC.
```

An early version of our Lola compiler generated directly executable code, i.e. an binary object file whose execution generated the data structure. A more flexible, portable, and simpler solution lies in choosing an interpretable code in the form of a binary tree. This tree closely mirrors the syntactic structure of the source program. We emphasize that this tree structure is different from the previously presented data structure. To be specific, interpretation of the former generates the latter. We will explain this process further below.

5. The Compiler's Object Code

The object code is a tree with two types of nodes called Item and Object. The latter is an extension of Item and carries a name (identifier). Objects represent entities declared in the Lola program, i.e. constants, variables, and types. The field next serves to link objects together, i.e. to form a symbol table. These types are declared as follows:

```
Item = POINTER TO ItemDesc;
Object = POINTER TO ObjDesc;
ItemDesc = RECORD
  tag, val: INTEGER;
a, b: Item
END;
```
The structure generated by the compiler from a given program can be derived from the translation rules given below for the various constructs of the language Lola. The translation rules consist of pairs, the left side specifying the source construct, the right side the corresponding piece of the data structure. The latter is specified by a text rather than a picture, the notation \([\text{fct}, x, y]\) standing for an element (of type \text{Item}, or \text{Object} if named). The form \(\{\text{op}, x_1, x_2, \ldots, x_n\}\) stands for the list \(\{\text{op}, x_1, \{\text{op}, x_2, \ldots \{\text{op}, x_n, \text{NIL} \} \ldots \}\}\). The list of possible tag values is:

<table>
<thead>
<tr>
<th>Language construct</th>
<th>Structure element</th>
</tr>
</thead>
<tbody>
<tr>
<td>(~ x)</td>
<td>[not, NIL, x]</td>
</tr>
<tr>
<td>MUX(s: x, y)</td>
<td>[mux, s, [mux1, x, y]]</td>
</tr>
<tr>
<td>REG(ck: en, d)</td>
<td>[reg, ck, [reg1, en, d]]</td>
</tr>
<tr>
<td>LATCH(g, d)</td>
<td>[latch, g, d]</td>
</tr>
<tr>
<td>SR(s', r')</td>
<td>[sr, s', r']</td>
</tr>
<tr>
<td>x*y</td>
<td>[and, x, y]</td>
</tr>
<tr>
<td>x DIV y</td>
<td>[div, x, y]</td>
</tr>
<tr>
<td>x MOD y</td>
<td>[mod, x, y]</td>
</tr>
<tr>
<td>−x</td>
<td>[neg, NIL, x]</td>
</tr>
<tr>
<td>x + y</td>
<td>[or, x, y]</td>
</tr>
<tr>
<td>x − y</td>
<td>[xor, x, y]</td>
</tr>
<tr>
<td>x#y</td>
<td>[eql, x, y]</td>
</tr>
<tr>
<td>x &lt; y</td>
<td>[lss, x, y]</td>
</tr>
<tr>
<td>x &lt;= y</td>
<td>[leq, x, y]</td>
</tr>
<tr>
<td>x &gt; y</td>
<td>[gtr, x, y]</td>
</tr>
<tr>
<td>x =&gt; y</td>
<td>[geq, x, y]</td>
</tr>
<tr>
<td>x := y</td>
<td>[assign, x, y]</td>
</tr>
<tr>
<td>x := y 1z</td>
<td>[tssass, x, [tsg, y, z]]</td>
</tr>
<tr>
<td>x(p1, p2, \ldots, pn)</td>
<td>[call, x, {list, p1, p2, \ldots, pn}]</td>
</tr>
<tr>
<td>IF x THEN S0 ELSE S1 END</td>
<td>[if, x, [if1, S0, S1]]</td>
</tr>
<tr>
<td>FOR x := y \ldots z DO S END</td>
<td>[for, x, {for1, y, [for2, z, S]]}</td>
</tr>
<tr>
<td>S1; S2; \ldots; Sn</td>
<td>{list, S1, S2, \ldots, Sn}</td>
</tr>
<tr>
<td>[n] T</td>
<td>[array, n, T]</td>
</tr>
<tr>
<td>IN x1, x2, \ldots, xn;</td>
<td>{op, x1, x2, \ldots, vn}</td>
</tr>
<tr>
<td>OUT y1, y2, \ldots, yn</td>
<td></td>
</tr>
<tr>
<td>INOUT z1, z2, \ldots, zn;</td>
<td></td>
</tr>
<tr>
<td>VAR v1, v2, \ldots, vn</td>
<td></td>
</tr>
</tbody>
</table>

(1) The tag field in each node indicates the kind of object represented (in, out, io, var, par).
(2) The b-field of the last parameter in the list points to the called variable's type, i.e. is not NIL.

As an example, the structure resulting from the compilation of the following Lola program is shown in Fig. 8. It includes a declared type \(T\), of which two instances \(G\) and \(H\) are created.
\textbf{MODULE M1;}

\textbf{TYPE T(N);}  
\textbf{IN x: BIT;}  
\textbf{OUT y: BIT;}  
\textbf{VAR a: [N] BIT;}  
\textbf{BEGIN} \text{y := a.1 + x} \text{END T;}  

\textbf{VAR u, v, w: BIT;}  
\textbf{G: T(2); H: T(3);}  
\textbf{BEGIN G(v); H(v); u := H.y}  
\textbf{END M1.}
The list A of declared types (see Fig. 8) is rooted in the global variable \texttt{LSC.globalScope.next}. List B of declared global objects (except types) is rooted in variable \texttt{LSC.localScope.next}. List C of statement structures of the module body is rooted in \texttt{LSC.body}. Lists A and B end with the sentinel element \texttt{LSC.guard} instead of with \texttt{NIL}. Items with \texttt{tag = lit} carry a literal value in field \texttt{val}.

The compiler consists of the two modules LSS (Lola System Scanner, 2400 bytes) and LSC (Parser and generator, 6900 bytes), and it is activated by the command

\begin{itemize}
  \item \texttt{Lola.Compile @} compiling the text beginning with the most recent selection
  \item \texttt{Lola.Compile *} compiling the text in the marked viewer
  \item \texttt{Lola.Compile name} compiling the named text file
\end{itemize}

6. Interpretation of the compiled data structure

Suitable interpretation of the compiled data leads to the desired data structure representing the specified circuit. We recall that in this structure, each circuit element (gate, register) is represented by a node. In contrast, in the compiler’s output, which represents the source text mapped into a tree, a variable of a composite type (array, record) is represented by a single node. Therefore, the interpretation (execution) of the code structure essentially constitutes an expansion process, to a flattening of the data structure. In particular, every structured variable is expanded into nodes, one for each component variable. Since components of structured variables may themselves be structured, the expansion procedure is necessarily recursive.

As an example, we show the result of the expansion of the code of Fig. 8. Instead of a data structure, the textual form produced by \texttt{Lola.Show} is chosen:

\begin{align*}
  H.x & := v \\
  G.x & := w \\
  H.y & := (H.a.1+H.x) \\
  G.y & := (G.a.1+G.x) \\
  H.a.0 & \\
  G.a.0 & \\
  H.a.1 & \\
  G.a.1 & \\
  H.a.2 & \\
  w & \\
  v & \\
  u & := H.y
\end{align*}

In our implementation, expansion follows compilation automatically, i.e. is implied in the command \texttt{Lola.Compile}. In addition, two further steps are implied following expansion, namely simplification and checking for combinational loops.

The expansion process starts with the first element of the list of global variables (B in Fig. 8), and then proceeds through the list. For constants at the head of the list (tag = par), interpretation consists of the evaluation of the numeric expression for which the constant stands. This is the purpose of procedure \texttt{V(x)}. For variables in the list, expansion is performed by the recursive procedure \texttt{NewVar}.

After all variables have been generated, they act as anchors for the data structures representing the assigned expressions. The data representing Lola statements are interpreted by procedure \texttt{S(x)}, first applied to \texttt{LSC.body}. Procedure \texttt{S} distinguishes between the different kinds of statements: assignments, calls, if−, and for statements. \texttt{S} calls procedure \texttt{E} interpreting data corresponding to Lola expressions. Whereas the interpretation of assignments is reasonably straightforward, the interpretation of calls is quite complicated. It includes the evaluation of actual parameters and their assignments to the corresponding input variables. This is done by procedure \texttt{Link}.

We note that the assignment of constants 0 and 1 to inputs may give rise to expressions assigned to local variables becoming amenable to simplification. This is the reason why simplification during source compilation alone is insufficient.
7. Circuit Simulation

The described data structure is also a suitable basis for simulating circuits. The principle of simulation is quite simple in the case of synchronous circuits, to which we restrict our attention. Time progresses in discrete steps, and in each step all registers are clocked simultaneously. Therefore, the behaviour of the entire circuit can be represented by a sequence of binary values for each variable. The goal of a simulator is to generate these sequences of binary values.

In order to understand the principle of the simulator, we consider the scheme of a general, synchronous, sequential machine, called a state machine. F and G are arbitrary combinational circuits, i.e. Boolean functions. R denotes the set of simultaneously clocked registers.

![State Machine Diagram](image)

Fig. 9. State machine

The simulation program is derived from Fig. 9. The following algorithm represents a single step:

For each variable compute its value according to the defining Boolean expression, from the values of other variables and from the states of the registers. Then do likewise for the register inputs. Finally, let the register values become equal to their inputs. This represents the clock tick.

The assignment of its value to a variable \( v \) is expressed as \( v.val := value(v.x) \), where the function recursively traverses the tree rooted in \( v.x \).

```pascal
PROCEDURE value(s: Signal): SHORTINT;
VAR w: SHORTINT;
BEGIN
  IF s # NIL THEN
    IF s IS Variable THEN assign(s(Variable)); w := s.val
    ELSE
      CASE s.fct OF
        | or: w := or[value(s.x), value(s.y)]
        | and: w := and[value(s.x), value(s.y)]
        | reg: w := s.val
        | ...
      END
    END
  ELSE w := undef
  END;
  RETURN w
END value;
```

An advantage of simulation is that a variable may be marked as undefined, and that undefined values may be propagated, detected and diagnosed. This is typically done not by providing an additional, Boolean attribute \( defined \), but by extending the range of values of \( val \). We use \( undefined = 3 \).

If the tree traversal reaches a variable node, then the process continues with the evaluation of that variable's expression. As this may lead to a never ending computation, nodes once visited must be marked. We assume that all nodes representing variables in the data structure are initially marked by, say, "color" black (\( val = 5 \)). During traversal, all visited variable nodes are brightened to grey (\( val = 4 \)). When the expression has been evaluated, the resulting value (0, 1, or 3) is assigned (\( val <= 3 \) is considered as white). If a grey node is reached, the evaluation process is evidently in a loop, which would have been detected by a prior test for combinational loops. We realize that indeed the coloring scheme is the same...
as the one used in procedure _LSB.Loops_

```pascal
PROCEDURE assign(v: Variable);
VAR lnk, lsg: Signal;
  w, h: SHORTINT;
BEGIN
  IF v.val = black THEN
    v.val := grey; v.val := value(v.x)
  ELSIF v.val = grey THEN
    WriteName(W, v); Texts.WriteString(W, " in loop"); Texts.WriteLn(W)
  END
END assign;
```

Tree traversal stops when a _white_ variable, an input variable, or a register is reached. At this point we realize that register nodes must hold two values, namely the current one and the newly computed one. Fortunately, in our data structure a register is represented by a double node. By convention, the current value of a register \( r \) is stored as \( r.val \) \( (r.fct = \text{reg}) \), and the new value as \( r.y.val \) \( (r.y.fct = \text{reg1}) \). A consequence is that before each evaluation step (clock tick) all register input values must be computed, and thereafter the effect of the tick must be simulated by replacing the old register values by the input values. Simulating \( N \) steps then becomes:

```
Initialize all variables to black;
FOR each variable \( v \) DO assign(\( v \)) END;
i := 0;
REPEAT (*step*)
  FOR each register \( r \) DO
    IF value(\( r.y.x \)) = 1 THEN (*enabled*) \( r.y.val := \) value(\( r.y.y \)) END
  END;
  FOR each register \( r \) DO \( r.val := r.y.val \) END; (*the tick*)
  Initialize all variables to black;
  FOR each variable \( v \) DO assign(\( v \)) END;
  FOR each variable \( v \) DO output(\( v.val \)) END;
  INC(i)
UNTIL i = N
```

How is the set of registers recognized? The question arises, because the data structure contains no explicit list of register nodes. Fortunately, it is easy to construct such a list during each traversal. For the register list, link field \( x \) is used, which in general represents a register's clock signal. Since we restrict simulation to synchronous circuits, clock signals can be ignored, since there is only one, implicit clock for all registers.

The presented scheme may indeed be regarded as the "poor man's solution". A more effective program might first topologically sort the variables such that whenever \( v = F(v_0, \ldots, v_n) \) is computed, all variables \( v_0 \ldots v_n \) have been assigned their values beforehand. Although this measure will speed up the simulation program, we refrain from presenting it in detail, because it complicates the program considerably. This is particularly so in view of the fact that we do not deal with a list, but with a tree of variables.

Some further considerations and explanations follow. The reader is referred to the listings for details.

1. The value \( w \) of a latch node \( s \) is defined as
   
   ```pascal
   IF value(\( s.x \)) = 0 THEN \( w := s.val \) ELSE \( w := value(s.y); s.val := w \) END
   ```

2. The value \( w \) of a SR node \( s \) is defined as
   
   ```pascal
   IF value(\( s.x \)) = 0 THEN
     IF value(\( s.y \)) = 0 THEN \( w := \text{clash} \) ELSE \( w := 1; s.val := 1 \) END
   ELSIF value(\( s.y \)) = 0 THEN \( w := 0; s.val := 0 \)
   ELSE w := s.val
   END
   ```
We note that the "illegal" case of both inputs of an SR latch being active is detected and results in a value called clash (2).

3. Assignment is expressed by \( v.val := \text{value}(v.x) \). This becomes more complicated in the case of bus variables, which allow that several signals define the variable. Hence, \( v.x \) does not denote the single expression tree like in the case of regular variables, but rather a list of expression trees. The list consists of nodes with fct-value list. Evaluation of a list of tri-state gates generates the value clash, if more than a single gate is enabled, and the value undef, if none is enabled.

4. A facility to initialize input variables is mandatory. This is represented by the command Set. For example

   Simulator.Set \( x = 0, y = 1, a = 2 \)

assigns the given values to the listed input variables.

5. Typically, the designer is interested in tracing the values of a few selected variables only. This possibility is offered through the command Select. The node field \( u \) is used to mark variables for tracing (s. procedure list).

Consider the following counter as an example:

\[
\begin{align*}
\text{MODULE} \hspace{1em} \text{Counter}; \\
\text{CONST} \hspace{1em} N := 4; \\
\text{IN} \hspace{1em} \text{en}: \text{BIT}; \\
\text{VAR} \hspace{1em} Q, c: [N] \text{BIT}; \\
\text{BEGIN} \hspace{1em} Q.0 := \text{REG}(Q.0 \cdot \text{en}); \hspace{1em} c.0 := Q.0 \cdot \text{en}; \\
\hspace{2em} \text{FOR} \hspace{1em} i := 1 \ldots N-1 \hspace{1em} \text{DO} \hspace{1em} Q.i := \text{REG}(Q.i \cdot c[i-1]); \hspace{1em} c.i := Q.i \cdot c[i-1] \hspace{1em} \text{END} \\
\text{END} \hspace{1em} \text{Counter}.
\end{align*}
\]

Executing the commands

   Simulator.Set \( \text{en} = 1 \)
   Simulator.Select Q.0 Q.1 Q.2 Q.3
   Simulator.Label
   Simulator.Step 8

yields the table of values

\[
\begin{array}{cccc}
\text{Q.0} & \text{Q.1} & \text{Q.2} & \text{Q.3} \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

8. Circuit implementation

The ultimate purpose of design tools is to generate implementations from given circuit specifications automatically. Given the compiler translating textual specifications into a suitable data structure, this task is slightly reduced into constructing a program further translating the data structure into a circuit implementation.

Circuit implementations generally take the form of layouts, i.e. exact specifications of the positions of circuit components and of their interconnections. Such layouts strongly depend on the technology used, but the general considerations are similar. Among the available technologies are printed circuit board's
and custom VLSI design using standard cells. Of particular interest and promise, however, are standard arrays of configurable cells, so-called programmable gate arrays. If the cells and their interconnects are programmable by the user "in the field", they are called field programmable gate arrays (FPGA). Automatic generation of an implementation using an FPGA consists of finding an assignment of gates and registers to cells in such a way that (1) the cells are capable of representing the function assigned to them, and (2) that the provided routing facilities allow to connect the cells according to the given circuit specifications.

This is in general an extremely difficult task, in particular, if a reasonably high utilization of the available cells and routing facilities is requested. An approach more amenable to the current state of the art is to lay out a circuit "by hand" with the aid of a suitable layout editor, and then to verify with the aid of a program that the implementation represents the circuit as specified by the Lola text. The best way to do this is to extract from the layout a data structure of the known format, and thereupon to compare it with the one derived from the Lola specification. We call such an extraction and comparison tool a Checker.

9. Fuse–Map Generation for a PLD

The task of automatically generating a layout, i.e. of finding a mapping of functions to gates, is easiest if all gates are the same and if the routing facilities are completely regular and general. An extreme case of this kind is the well known programmable logic array (PLA) consisting of an And- and an Or-matrix. It directly reflects Boolean expressions in disjunctive normal form. Hence, the generation of an implementation is straightforward, given the restriction that the (Lola) program consist of a (small) set of variables defined by expressions in normal form. If the array is field programmable, the circuit's implementation occurs entirely by software. The device having a fixed set of inputs $x_0 \ldots x_{n-1}$ and outputs $y_0 \ldots y_{m-1}$, the latter are defined by

$$y_i := F_i(x_0 \ldots x_{n-1}) \quad i = 0 \ldots m-1$$

where $F$ is a disjunctive normal form, i.e. a sum of products of the arguments and their inverses.

The usefulness of this scheme is significantly enlarged by applying it to PLDs, i.e. devices containing registers. These devices are primarily designed to implement state machines. Typically, the $y$ are fed into a so-called macro cell generating the output. The industry standard GAL22V10 [7] provides the four options shown in Fig. 10. Note that we ignore the macro cell's tri-state gate, i.e. always consider them as active. This is because in the intended laboratory application a fixed set of pins is connected as inputs from the processor bus, and another fixed set is connected to the bus as outputs (see Fig. 13). However, we retain the important facility of feedbacks which may be selected as arguments in $F$.

![Fig. 10. The 4 options of macro cells in the GAL22V10](image)

Consequently, the options for the specification of the outputs grow too:

$$y_i := F(x_0 \ldots x_{n-1}, y_0 \ldots y_{m-1})$$
$$y_i := \neg F(x_0 \ldots x_{n-1}, y_0 \ldots y_{m-1})$$
$$y_i := \text{REG}(F(x_0 \ldots x_{n-1}, y_0 \ldots y_{m-1}))$$
$$y_i := \neg \text{REG}(F(x_0 \ldots x_{n-1}, y_0 \ldots y_{m-1}))$$

The syntax of these forms is a subset of Lola:

- assignment = variable "=" definition.
- definition = expression | "~" expression | "REG" "(" expression ")" | "~" "REG" "(" expression ")".
- expression = term {"+" term}. 

The syntax of these forms is a subset of Lola:
term = factor {"*" factor}.
factor = variable | "~" variable | "'0" | "'1" .

The following are simple examples of Lola specifications that can quite easily be translated into a PLD implementation:

**MODULE Adder;**
IN x0, x1, x2, x3, y0, y1, y2, y3, ci: BIT;
OUT s0, s1, s2, s3, c0, c1, c2, c3: BIT;
POS x0=3; x1=4; x2=5; x3=6; y0=7; y1=9; y2=10; y3=11; ci=12;
s0=27; s1=26; s2=25; s3=24; c0=23; c1=21; c2=20; c3=19;
BEGIN
  s0 := ˜ci*˜x0*y0 + ˜ci*x0*˜y0 + ci*˜x0*˜y0 + ci+x0*y0;
  s1 := ˜c0*˜x1*y1 + ˜c0*x1*˜y1 + c0*˜x1*y1 + c0*x1 + c0*y1;
  s2 := ˜c1*˜x2*y2 + ˜c1*x2*˜y2 + c1*˜x2*˜y2 + c1*x2*y2;
  s3 := ˜c2*˜x3*y3 + ˜c2*x3*˜y3 + c2*˜x3*y3 + c2*x3 + c2*y3;
END Adder.

**MODULE Barrel;**
IN s0, s1, s2: BIT;
d0, d1, d2, d3, d4, d5, d6, d7: BIT;
OUT q0, q1, q2, q3, q4, q5, q6, q7: BIT;
POS d0=3; d1=4; d2=5; d3=6; d4=7; d5=9; d6=10; d7=11; s0 = 12; s1=13; s2=16;
q0=27; q1=26; q2=25; q3=24; q4=23; q5=21; q6=20; q7=19;
BEGIN
  q0 := ˜s2*˜s1*˜s0*d0 + ˜s2*˜s1*s0*d1 + ˜s2*s1*˜s0*d2 + ˜s2*s1*s0*d3 + ˜s2*˜s1*˜s0*d4 + ˜s2*˜s1*s0*d5 + s2*˜s1*˜s0*d6 + s2*˜s1*s0*d7;
  q1 := ˜s2*˜s1*˜s0*d1 + ˜s2*˜s1*s0*d2 + ˜s2*s1*˜s0*d3 + ˜s2*s1*s0*d4 + ˜s2*˜s1*˜s0*d5 + ˜s2*˜s1*s0*d6 + s2*˜s1*˜s0*d7 + s2*˜s1*s0*d0;
  q2 := ˜s2*˜s1*˜s0*d2 + ˜s2*˜s1*s0*d3 + ˜s2*s1*˜s0*d4 + ˜s2*s1*s0*d5 + ˜s2*˜s1*˜s0*d6 + ˜s2*˜s1*s0*d7 + s2*˜s1*˜s0*d0 + s2*˜s1*s0*d1;
  q3 := ˜s2*˜s1*˜s0*d3 + ˜s2*˜s1*s0*d4 + ˜s2*s1*˜s0*d5 + ˜s2*s1*s0*d6 + ˜s2*˜s1*˜s0*d7 + ˜s2*˜s1*s0*d0 + s2*˜s1*˜s0*d1 + s2*˜s1*s0*d2;
  q4 := ˜s2*˜s1*˜s0*d4 + ˜s2*˜s1*s0*d5 + ˜s2*s1*˜s0*d6 + ˜s2*s1*s0*d7 + ˜s2*˜s1*˜s0*d0 + ˜s2*˜s1*s0*d1 + s2*˜s1*˜s0*d2 + s2*˜s1*s0*d3;
  q5 := ˜s2*˜s1*˜s0*d5 + ˜s2*˜s1*s0*d6 + ˜s2*s1*˜s0*d7 + ˜s2*s1*s0*d0 + ˜s2*˜s1*˜s0*d1 + ˜s2*˜s1*s0*d2 + s2*˜s1*˜s0*d3 + s2*˜s1*s0*d4;
  q6 := ˜s2*˜s1*˜s0*d6 + ˜s2*˜s1*s0*d7 + ˜s2*s1*˜s0*d0 + ˜s2*s1*s0*d1 + ˜s2*˜s1*˜s0*d2 + ˜s2*˜s1*s0*d3 + s2*˜s1*˜s0*d4 + s2*˜s1*s0*d5;
  q7 := ˜s2*˜s1*˜s0*d7 + ˜s2*˜s1*s0*d0 + ˜s2*s1*˜s0*d1 + ˜s2*s1*s0*d2 + ˜s2*˜s1*˜s0*d3 + ˜s2*˜s1*s0*d4 + s2*˜s1*˜s0*d5 + s2*˜s1*s0*d6;
END Barrel.

**MODULE Counter;**
IN ci: BIT;
OUT q0, q1, q2, q3, q4, q5, q6, q7: BIT;
POS ci=3; q0=27; q1=26; q2=25; q3=24; q4=23; q5=21; q6=20; q7=19;
BEGIN
  q0 := REG(q0*˜ci + ˜q0*ci);
  q1 := REG(q1*˜q0 + q1*˜ci + ˜q1*q0*ci);
  q2 := REG(q2*˜q1 + q2*˜q0 + q2*˜ci + ˜q2*q1*q0*ci);
  q3 := REG(q3*˜q2 + q3*˜q1 + q3*˜q0 + q3*˜ci + ˜q3*q2*q1*q0*ci);
  q4 := REG(q4*˜q3 + q4*˜q2 + q4*˜q1 + q4*˜q0 + q4*˜ci + ˜q4*q3*q2*q1*q0*ci);
  q5 := REG(q5*˜q4 + q5*˜q3 + q5*˜q2 + q5*˜q1 + q5*˜q0 + q5*˜ci + ˜q5*q4*q3*q2*q1*q0*ci);
  q6 := REG(q6*˜q5 + q6*˜q4 + q6*˜q3 + q6*˜q2 + q6*˜q1 + q6*˜q0 + q6*˜ci + ˜q6*q5*q4*q3*q2*q1*q0*ci);
  q7 := REG(q7*˜q6 + q7*˜q5 + q7*˜q4 + q7*˜q3 + q7*˜q2 + q7*˜q1 + q7*˜q0 + q7*˜ci + ˜q7*q6*q5*q4*q3*q2*q1*q0*ci);
END Counter.
We subsequently show the development of a program which configures (programs) the PLD, i.e. derives a fuse map from a given Lola specification. The fuse map is the matrix of Boolean values standing for open or closed connections in the And−matrix. Since these “fuses” can be determined dynamically by “loading” the Boolean array into the device, we can imagine that we are generating a circuit automatically according to a specification given in Lola. What makes this scheme attractive for teaching and experimenting is the fact that the device can be reset to a null state by erasing the connections. This can be done electrically, i.e. without the need for removing the chip from the board.

The generator program is used in conjunction with a PLD connected to a workstation serving as a test site for experimentation. The PLD appears to the programmer as an I/O device, and it is connected to the computer’s bus with fixed sets of 12 input and 8 output pins. The PLD chosen is the industry standard part GAL22V10B. This setup implies that the flexibility of the PLD is not used to its full extent. (The device actually allows to be configured with up to \( n = 22 \) inputs and at most \( m = 10 \) outputs, whereby \( m+n = 22 \)). This self-imposed restriction has the side-effect that we may ignore the tri-state gates controlling the pin functions. Furthermore, we decided to dedicate one of the inputs to serve exclusively as clock signal.

Signals in a circuit specification (Lola program) must by some means become related to the signals of the device identified by their pin numbers. The simplest solution is to assign fixed names to pins, such as D0, D1, … . Not wishing to infringe on the designer’s freedom to choose meaningful identifiers, we instead extended the Lola language by a construct allowing to associate pin numbers with declared input and output variables. The program GAL then checks whether inputs are assigned to input pins and outputs to output pins. (The pin number of a variable \( v \) is stored in the record field \( v.u \)).

Next, we need to explain the structure of the fuse map. It consists of two parts, namely the matrix \( D \) representing the And−matrix, and two arrays \( S0 \) and \( S1 \) determining the macro cell options of inversion and register (see Fig. 10). Matrix \( D \) consists of \( M = 44 \) rows of \( N = 132 \) elements with value 0 or 1. Each row represents an argument (input or fed back output) or its inverse. There are potentially 22 signals, yielding 44 rows, each of which can be connected to any of the 132 product terms, yielding rows with 132 elements (see Fig. 11). \( D[i, j] = 0 \) means that variable \( i \) (if even, or its inverse, if \( i \) odd) is a factor in term \( j \), i.e. is connected to product line \( j \). \( D[i, j] = 1 \) means that the connection is open. Note that in typical pictures of PLD fuse maps, product lines run horizontally, whereas in matrix \( D \) a product line is represented by a column). Initially, \( D \) contains all ones, representing the device’s state after an erase operation.
We are now ready to look at program GAL which proceeds as follows: Procedure \textit{evaluate} visits all variables in the Lola data structure. For each output variable \(v\), its internal signal number (index) \(k\) is obtained from the specified pin number \(v.u\) through the constant mapping \(k = \text{smap}[v.u]\). If node \(s \equiv v.x\) is a negation, the respective macro cell option is selected by setting \(S0[k]\) to 0, and if the node specifies a register, the register option is selected by setting \(S1[k]\) to 0. Then procedure \textit{expression} is called to process the respective tree structure. Its parameters are the tree's root \(s\) and the output variable's number \(k\).

\begin{verbatim}
PROCEDURE evaluate(v: Variable);
VAR s: LSB.Signal;
BEGIN (*traverse list of variables*)
WHILE v # NIL DO
  IF v.class = out THEN
    IF v.fct = bit THEN s := v.x;
    IF s.fct = not THEN s := s.y END ;
    IF s.fct = reg THEN
      IF (s.x = NIL) OR (s.x # clk) OR (s.y.x # one) THEN
 (*error: register with clock or enable specification*)
      END ;
      s := s.y.y
    END ;
    expression(s, smap[v.u])
  ELSIF v.fct = array THEN evaluate(v.dsc)
  ELSE (*error: bus variable*)
  END
  END
  v := v.next
END evaluate;
\end{verbatim}

An expression consists of a sum of product terms, each of which is processed by a call to procedure \textit{term}. Each term is represented by a column in matrix \(D\), and since every variable is the sum of a limited number of terms, the term's number must be determined and then checked against its limit. Procedure \textit{expression}(\(k\)) obtains the number \(j\) of its first term from the mapping \(jmap[k]\), and then increments \(j\) for each new term up to the maximum specified by \(jmap[k+1] - 1\). (Note that this explicit mapping is needed because the maximum number of terms differs among the variables). After processing the last term, the remaining product lines, if any, must be set to yield 0 by selecting any input and its inverse (\(\sim xx = 0\)).

\begin{verbatim}
PROCEDURE expression(s: Signal; k: INTEGER);
VAR j, lim: INTEGER;
BEGIN j := jmap[k]+1; lim := jmap[k+1];
  WHILE s.fct = or DO term(s.y, j); INC(j); s := s.x
  IF j = lim THEN (*error: too many product terms*) END
  term(s, j); INC(j)
  WHILE j < lim DO Zero(j); INC(j) END
END expression;
\end{verbatim}

Procedure \textit{term} further traverses the tree structure and calls upon \textit{factor}. Note that the same product line index \(j\) is used for all factors.

\begin{verbatim}
PROCEDURE term(s: Signal; j: INTEGER);
BEGIN
  WHILE s.fct = and DO factor(s.y, j); s := s.x END ;
factor(s, j)
END term;
\end{verbatim}
Procedure factor finally assigns zeroes to elements of D, i.e. registers connections. $D[i, j]$ becomes 0, if variable $i = imap[s.u]$ is a factor in term j, and $D[i+1, j]$ becomes 0, if the inverse of variable $i$ is a factor.

```pascal
PROCEDURE factor(s: Signal; j: INTEGER);
VAR inv: SHORTINT;
BEGIN
  IF s.fct = LSB.not THEN inv := 1; s := s.y ELSE inv := 0 END;
  IF s IS LSB.Variable THEN
    IF s.u IN inputs + outputs THEN D[imap[s.u]+inv, j] := 0
    ELSIF s = zero THEN Zero(j)
    ELSIF s = one THEN (*do nothing; unconnected terms have value 1*)
    ELSE (*error: variable neither input nor output*)
    END
  ELSE (*error: illegal expression*)
  END
END factor;
```

For details and a complete presentation, the reader is referred to the program listings at the end of this text, where it becomes evident that a specific detail has been omitted from the above discussion. This omission causes the device to be configured wrongly when a macro cell is chosen to include a register but no inverter. We note that then the feedback term is not identical to the output, but unfortunately is its inverse, for no apparent reason. This forces a prescan of the macro cell configurations before the matrix is generated. Procedure PreScan sets the values of arrays $S0$ and $S1$. An auxiliary array $S2$ is used, and factor requires a corresponding adjustment.

10. The GAL circuit board for Ceres, and loading the fuse map

The implementation of an interface between the Ceres Bus and the GAL22V10B is shown in Fig. 12. Input and output addresses (relative to a base address for the GAL) are assigned as follows. Outputs are latched in registers (74LS574).

- **write**: 0 write 8 bits to GAL inputs at pins 3–11
- 4 neg. pulse to GAL input at pin 2, used as register clock
- 12 write 8 bits to GAL inputs at pins 12, 13, 16
- program step, latch SDI, MODE, and SCLK

- **read**: 0 read 8 bits from GAL outputs at pins 27–19
- 12 read GAL shift register output SDO as bit 0 for configuration verification
Fig. 12. GAL-interface for Ceres computer

Loading of the fuse map is controlled by a state machine in the ispGAL22V10B part. It is clocked by the clock signal SCLK, and it has 3 states: idle, read command, and execute command. State transitions occur when the input signal MODE is 1. When in read command mode, a command consisting of 5 bits is shifted (LSB first) into the command register of the programming machinery. This is done by procedure `EnterCmd` (see Fig 13). There are the following commands:

00010 shift data into (and out of) the 13 2+6 bits shift register (row data and row number)
00011 erase the fuse maps
00111 program the fuse map row specified by the last 6 bits of the shift register
01010 load the shift register from the specified fuse map row
10100 shift data into (and out of) the 20 bit architecture shift register

Program GAL uses procedures `WriteRow(d)` and `Arch` to load row d and the architecture (cell options) S0 and S1 respectively. They in turn use procedure P (for pulse) which first outputs and latches the programming machinery's data signal SDI (bit 0), then sets the SCLK signal to 1 (rising edge), and finally
resets it to 0. Further details are contained in the GAL data books and in the listing of the program. We merely note that every command, after being loaded, requires at least one clock pulse for being executed. Also, timing constraints must be observed, and the clock signal must return to 0 for taking effect.

Fig. 13. Signals for loading and executing a programming command

11. Converting expressions into normal form

Program GAL requires that expressions defining the output variables be in disjunctive normal form (DNF). In order to remove this restriction, a suitable preprocessor is needed which converts arbitrary expressions into their equivalent normal form. This is the purpose of module DNF and its procedure `Convert`.

This command calls procedure `list`, which in turn calls procedure `R(s) - via ekm` - for each output variable v, s is the root of the expression tree defining v, if a possible initial inverter and/or a register are ignored. R then calls upon procedures P and Q, which perform the actual conversions on the tree.

The recursive procedure P serves to remove multiplexers and Xor-gates by replacing them by And- and Or-gates, and to remove inversions as far as possible. P has as parameters the structure to which it is applied, and the Boolean `inv` indicating whether the true value or its inversion is to be generated. The different cases handled by P are best represented by recursive equivalences. The reader is referred to the program listing at the end of this text. We note that the data structure is modified in situ.

\[
\begin{align*}
P(\neg x, F) & \rightarrow P(x, T) \\
P(\neg x, T) & \rightarrow P(x, F) \\
P(x \cdot y, F) & \rightarrow P(x, F) \cdot P(y, F) \\
P(x \cdot y, T) & \rightarrow P(x, T) + P(y, T) \quad \text{de Morgan} \\
P(x + y, F) & \rightarrow P(x, F) + P(y, F) \\
P(x + y, T) & \rightarrow P(x, T) \cdot P(y, T) \quad \text{de Morgan} \\
P(MUX(s: x, y), F) & \rightarrow P(s, T) \cdot P(x, F) + P(s, F) \cdot P(y, F) \\
P(MUX(s: x, y), T) & \rightarrow P(s, T) \cdot P(x, T) + P(s, F) \cdot P(y, T) \\
P(v, F) & \rightarrow v \\
P(v, T) & \rightarrow \neg v
\end{align*}
\]

The recursive procedure Q performs the actual expansions necessary to obtain normal forms. For each node representing an And- or an Or-gate, it first applies itself to the two operands, and then recognizes the following patterns and performs structural transformations:

\[
\begin{align*}
x \cdot (y + z) & \rightarrow (x \cdot y) + z \\
x \cdot (y \cdot z) & \rightarrow (x \cdot y) \cdot z \\
(x \cdot y) \cdot z & \rightarrow x \cdot z + y \cdot z \\
x \cdot (y + z) & \rightarrow x \cdot y + x \cdot z
\end{align*}
\]
Concluding, we show the three examples of circuit specifications from the preceding section again. The possibility to express them in terms of expressions not in normal form improves readability significantly. However, the user must keep in mind that the expansion process brings them “back” into their expanded form. Using a PLD, one must be aware that the limits in the number of product terms may easily be reached when the necessary conversions are hidden.

\[
\begin{align*}
\text{MODULE Adder; } \\
\text{IN } x, y: \left[4\right] \text{ BIT}; c_i: \text{ BIT}; \\
\text{OUT } s, c: \left[4\right] \text{ BIT}; \\
\text{POS } x.0=3; x.1=4; x.2=5; x.3=6; y.0=7; y.1=9; y.2=10; y.3=11; c_i=12; \\
s.0=0; s.1=26; s.2=24; s.3=22; c.0=23; c.1=21; c.2=20; c.3=19; \\
\text{BEGIN } s.0 := x.0 − y.0 − c_i; c.0 := x.0*y.0 + (x.0 − y.0)*c_i; \\
\text{FOR } i := 1 .. 3 \text{ DO } s.i := x.i − y.i − c[i−1]; c.i := x.i*y.i + (x.i − y.i) * c[i−1] \text{ END} \\
\end{align*}
\]

END Adder.

\[
\begin{align*}
\text{MODULE Barrel; } \\
\text{IN } s: \left[3\right] \text{ BIT}; d: \left[8\right] \text{ BIT}; \\
\text{OUT } q: \left[8\right] \text{ BIT}; \\
\text{POS } d.0=3; d.1=4; d.2=5; d.3=6; d.4=7; d.5=9; d.6=10; d.7=11; \\
s.0 = 12; s.1=13; s.2=16; \\
q.0=27; q.1=26; q.2=25; q.3=24; q.4=23; q.5=21; q.6=20; q.7=19; \\
\text{BEGIN } \\
\text{FOR } i := 0 .. 7 \text{ DO } q.i := \text{MUX}(s.2: \\
\text{MUX}(s.0: d[i], d[(i+1) MOD 8]), \\
\text{MUX}(s.0: d[(i+2) MOD 8], d[(i+3) MOD 8])), \\
\text{MUX}(s.1: \\
\text{MUX}(s.0: d[(i+4) MOD 8], d[(i+5) MOD 8]), \\
\text{MUX}(s.0: d[(i+6) MOD 8], d[(i+7) MOD 8]))) \\
\text{END } \\
\text{END Barrel.} \\
\text{MODULE Counter; } \\
\text{IN } c_i: \text{ BIT}; \\
\text{OUT } q: \left[8\right] \text{ BIT}; \\
\text{VAR } c: \left[8\right] \text{ BIT}; \\
\text{POS } c_i=3; q.0=27; q.1=26; q.2=25; q.3=24; q.4=23; q.5=21; q.6=20; q.7=19; \\
\text{BEGIN } q.0 := \text{REG}(q.0 − c_i); c.0 := q.0 * c_i; \\
\text{FOR } i := 1 .. 7 \text{ DO } q.i := \text{REG}(q.i − c[i−1]); c.i := q.i * c[i−1] \text{ END} \\
\text{END Counter.}
\end{align*}
\]

12. References


2. Lola: An object-oriented logic description language,

3. S. Ludwig. An Editor for the CLi6000 FPGA, and its implementation.

4. S. Ludwig. CL-Editor user manual.

5. S. Gehring. CLChecker user manual.
13. Program Listings

MODULE LSB; (*Lola System Base NW 252.95 / 15.4.95*)
IMPORT Texts, Oberon, MenuViewers, TextFrames;

CONST NameLen* = 7;
black* = 5; grey* = 4; (*node values used in loop search*)
(*function codes*)
bit* = 0; ts* = 1; oc* = 2; integer* = 3; array* = 4; record* = 5; buf* = 7; not* = 8; and* = 9;
or* = 10; xor* = 11; mux* = 12; mux1* = 13; reg* = 14; reg1* = 15; latch* = 16; sr* = 17; tsg* = 18; lin* = 19;
(*class codes*) var* = 0; in* = 1; out* = 2; io* = 3;

TYPE Name* = ARRAY NameLen OF CHAR;
Signal* = POINTER TO SignalDesc;
Variable* = POINTER TO VarDesc;
(*fct field: bits 0−4: function code, bits 5−7: class code;
val field: not used; u,v fields: position data*)
SignalDesc* = RECORD
  x*, y*: Signal;
  fct*, val*, u*, v*: SHORTINT
END;
VarDesc* = RECORD (SignalDesc)
  name*: Name;
  class*: SHORTINT;
  next*, dsc*: Variable
END;
VAR org*, zero*, one*, clk*: Variable;
Log*: Texts.Text;
change: BOOLEAN;
code: ARRAY 24 OF CHAR;
W: Texts.Writer;
PROCEDURE Init*;
BEGIN org := NIL; clk.x := NIL; Oberon.Collect(0)
END Init;
PROCEDURE WriteName*(VAR W: Texts.Writer; v: Variable);
BEGIN
  IF v.y # NIL THEN WriteName(W, v.y(Variable)); Texts.Write(W, ".") END;
  Texts.WriteString(W, v.name)
END WriteName;
PROCEDURE This*(org: Variable; VAR name: ARRAY OF CHAR): Variable;
VAR v: Variable; i, j: INTEGER;
id: ARRAY 16 OF CHAR;
BEGIN v := org.dsc; i := 0;
  LOOP j := 0;
    WHILE (name[i] > ") & (name[i] # ".") DO id[j] := name[i]; INC(j); INC(i) END;
    id[j] := 0X;
    WHILE (v # NIL) & (v.name # id) DO v := v.next END;
    IF name[i] = "." THEN
      IF (v # NIL) & (v.fct IN {array, record}) THEN v := v.dsc; INC(i)
      ELSE v := NIL; EXIT
    ELSE EXIT
  END;
END;
RETURN v
END This;
PROCEDURE New*(f: SHORTINT; x, y: Signal): Signal;
VAR z: Signal;
BEGIN NEW(z); z.fct := f; z.x := x; z.y := y; RETURN z
END New;
PROCEDURE NewVar*(f, val: SHORTINT; x, y: Signal;
  next: Variable; VAR name: ARRAY OF CHAR): Variable;
VAR v: Variable;
BEGIN NEW(v); v.fct := f; v.val := val; v.x := x; v.y := y;
v.next := next; COPY(name, v.name); RETURN v
END NewVar;

(*------------------------ Simplify ------------------------*)

PROCEDURE traverse(VAR s: Signal);
BEGIN
  IF s # NIL THEN
    IF s IS Variable THEN
      IF s.x = zero THEN s := zero
      ELSIF s.x = one THEN s := one
    END
    ELSE traverse(s.x); traverse(s.y);
    IF s.fct = not THEN
      IF s.y.fct = not THEN s := s.y.y
      ELSIF s.y = zero THEN s := one
      ELSIF s.y = one THEN s := zero
    END
    ELSIF s.fct = or THEN
      IF s.x = one THEN s := one
      ELSIF s.x = zero THEN s := s.y
      ELSIF s.y = one THEN s := one
      ELSIF s.y = zero THEN s := s.x
    END
    ELSIF s.fct = xor THEN
      IF s.x = zero THEN s := s.y
      ELSIF s.x = one THEN
        IF s.y.fct = not THEN s := s.y.y ELSE s.fct := not; s.x := NIL END
      ELSIF s.y = zero THEN s := s.x
      ELSIF s.y = one THEN
        IF s.x.fct = not THEN s := s.x.y ELSE s.fct := not; s.y := s.x; s.x := NIL END
    END
    ELSIF s.fct = and THEN
      IF s.x = zero THEN s := zero
      ELSIF s.x = one THEN s := s.y
      ELSIF s.y = zero THEN s := zero
      ELSIF s.y = one THEN s := s.x
    END
    ELSIF s.fct = mux THEN
      IF s.x = zero THEN s := s.y.x ELSIF s.x = one THEN s := s.y.y
    END
    ELSE s.fct = reg THEN
      IF (s.x = zero) OR (s.x = one) OR (s.y.x = zero) THEN
        Texts.WriteString(W, " dead reg'"); Texts.WriteLn(W)
      END
    END
    ELSE s.fct = latch THEN
      IF s.x = zero THEN Texts.WriteString(W, " dead latch'"); Texts.WriteLn(W)
      END
    END
    ELSE s.fct = sr THEN
      IF (s.x = zero) OR (s.y = zero) THEN Texts.WriteString(W, " dead SR'"); Texts.WriteLn(W)
    END
    ELSE s.fct = tsg THEN
      IF (s.x = zero) OR (s.x = one) THEN
        Texts.WriteString(W, " dead tri-state'"); Texts.WriteLn(W)
      END
    END
    END
  END
END traverse;

PROCEDURE simp(v: Variable);
BEGIN
  IF v.fct IN {array, record} THEN v := v.dsc;
  WHILE v # NIL DO simp(v); v := v.next END
  ELSIF (v.x # zero) & (v.x # one) THEN
    IF v.fct = link THEN traverse(v.x); traverse(v.y)
    ELSE traverse(v.x);
      IF (v.x = zero) OR (v.x = one) THEN change := TRUE END
    END
  END
END simp;

PROCEDURE Simplify*(org: Variable);
VAR n: INTEGER;
BEGIN n := 0;
  REPEAT INC(n); change := FALSE; simp(org) UNTIL ~change;
  Texts.Append(Log, W.buf)
END Simplify;

(*------------------- Find Loops -------------------*)
PROCEDURE loop(s: Signal);
BEGIN
  IF s # NIL THEN
    IF s IS Variable THEN
      IF s.val = black THEN s.val := grey; loop(s.x); s.val := 0
      ELSEIF s.val = grey THEN
        WriteName(W, s(Variable)); Texts.WriteString(W, " in loop"); Texts.WriteLn(W);
        Texts.Append(Log, W.buf)
      END
    ELSEIF s.fct # reg THEN
      loop(s.x);
      IF s.fct # tsg THEN loop(s.y) END
    END
  END
END loop;

PROCEDURE Loops*(v: Variable);
BEGIN
  IF v.fct IN {array, record} THEN v := v.dsc;
  WHILE v # NIL DO Loops(v); v := v.next END
  ELSIF v.val = black THEN loop(v)
END Loops;

(*------------------- Show -------------------*)
PROCEDURE ShowTree(x: Signal);
BEGIN
  IF x # NIL THEN
    IF x IS Variable THEN WriteName(W, x(Variable))
    ELSE Texts.Write(W, ";"); ShowTree(x.x); Texts.Write(W, code[x.fct]); ShowTree(x.y); Texts.Write(W, ")")
    END
  END
END ShowTree;

PROCEDURE Show*(x: Variable; lev: INTEGER);
VAR typ: SHORTINT;
BEGIN typ := x.fct;
  IF typ = record THEN
    x := x.dsc;
    WHILE x # NIL DO Show(x, lev+1); x := x.next END
    Texts.Append(Log, W.buf)
  ELSEIF typ = array THEN
    x := x.dsc;
    WHILE x # NIL DO Show(x, lev); x := x.next END
  ELSEIF typ # integer THEN
    WriteName(W, x);
    IF (lev = 0) & (x.class # var) THEN Texts.Write(W, "*") END;
    IF xx # NIL THEN Texts.WriteString(W, " := "); ShowTree(xx) END;
    Texts.WriteLn(W); Texts.Append(Log, W.buf)
  END
END Show;

(*------------------- Open -------------------*)
PROCEDURE OpenLog*;
VAR V: MenuViewers.Viewer; X, Y: INTEGER;
BEGIN Oberon.AllocateSystemViewer(Oberon.Mouse.X, X, Y);
  V := MenuViewers.New (
    TextFrames.NewMenu("Lola.Log",
    TextFrames.NewText(Log, 0), TextFrames.menuH, X, Y)
END OpenLog;

PROCEDURE ClearLog*;
MODULE Lola; (*NW 2.2.95 / 13.4.95*)
IMPORT Texts, Oberon, Viewers, MenuViewers, TextFrames, LSB, LSC;

VAR scope: LSB.Variable; clk: LSB.Signal;
null: ARRAY 2 OF CHAR;
W: Texts.Writer;

PROCEDURE V(x: LSC.Item): INTEGER;
VAR y: INTEGER; v: LSB.Variable; id: LSB.Name;
BEGIN
IF x IS LSC.Object THEN
v := scope; COPY(x(LSC.Object).name, id);
WHILE v.name # id DO v := v.next END;
y := v.val
ELSE
CASE x.tag OF
 | LSC.lit: y := x.val
 | LSC.add: y := V(x.a) + V(x.b)
 | LSC.sub: y := V(x.a) − V(x.b)
 | LSC.neg: y := −V(x.b)
 | LSC.mul: y := V(x.a) * V(x.b)
 | LSC.div: y := V(x.a) DIV V(x.b)
 | LSC.mod: y := V(x.a) MOD V(x.b)
 | LSC.pwr: y := SHORT(ASH(1, V(x.b)))
END
END;
RETURN y
END V;

PROCEDURE Index(n: INTEGER; VAR name: ARRAY OF CHAR);
VAR i, j: INTEGER;
d: ARRAY 4 OF INTEGER;
BEGIN i := 0; j := 0;
REPEAT d[i] := n MOD 10; INC(i); n := n DIV 10 UNTIL n = 0;
REPEAT DEC(i); name[j] := CHR(d[i] + 30H); INC(j) UNTIL i = 0;
END Index;

PROCEDURE NewVar(typ: LSC.Item; mode: INTEGER; VAR id: ARRAY OF CHAR; link, anc: LSB.Variable):
LSB.Variable;
VAR form, len: INTEGER;
fp: LSC.Object; ap: LSC.Item;
v, el, x, y, tempscope: LSB.Variable;
BEGIN
form := typ.tag;
v := LSB.NewVar(SHORT(form), LSB.black, NIL, anc, id); v.class := SHORT(mode-LSC.var);
IF form = LSB.array THEN
len := V(typ.b); el := NIL;
WHILE len > 0 DO DEC(len); el := NewVar(typ.a, mode, null, el, v); Index(len, el.name) END;
v.dsc := el
ELSIF form = LSB.record THEN
COPY(id, v.name); el := NIL;
ap := typ.a.a(LSC.Object);
WHILE ap # NIL DO
el := LSB.NewVar(LSB.integer, SHORT(V(ap.a)), NIL, v, el, fp.name); fp := fp.next; ap := ap.b
END;
tempscope := scope; scope := el;
WHILE fp.tag = LSC.const DO
el := LSB.NewVar(LSB.integer, SHORT(V(fp.b)), NIL, v, el, fp.name); scope := el; fp := fp.next
END;
WHILE fp # LSC.guard DO el := NewVar(fp.a, fp.tag, fp.name, el, v); fp := fp.next END;
y := NIL; (*invert list*)
WHILE el # NIL DO x := el; el := x.next; x.next := y; y := x END;
scope := tempscope; v.dsc := y
END;
RETURN v
END NewVar;

PROCEDURE E(x: LSC.Item): LSB.Signal;
VAR y: LSB.Signal; v: LSB.Variable;
tag, k: INTEGER; id: LSB.Name;
BEGIN
tag := x.tag;
IF x IS LSC.Object THEN
v := scope; COPY(x(LSC.Object).name, id);
WHILE v.name # id DO v := v.next END;
ELSE
CASE x.tag OF
 | LSC.lit: y := x.val
 | LSC.add: y := E(x.a) + E(x.b)
 | LSC.sub: y := E(x.a) − E(x.b)
 | LSC.neg: y := −E(x.b)
 | LSC.mul: y := E(x.a) * E(x.b)
 | LSC.div: y := E(x.a) DIV E(x.b)
 | LSC.mod: y := E(x.a) MOD E(x.b)
 | LSC.pwr: y := SHORT(ASH(1, E(x.b)))
END
END;
RETURN y
END E;
y := v
ELSE
    CASE x.tag OF
        LSC.asel: y := E(x.a); v := y(LSB.Variable).dsc; k := V(x.b);
        WHILE (k > 0) & (v # NIL) DO v := v.next; DEC(k) END;
        IF (v = NIL) OR (k < 0) THEN
            Texts.WriteString(W, 'index off range in ');
            LSB.WriteName(W, v); Texts.WriteLn(W);
            Texts.Append(LSB.Log, W.buf);
            v := v.dsc
        END;
        y := v
    | LSC.rsel: y := E(x.a); v := y(LSB.Variable).dsc;
    COPY(x.b(LSC.Object).name, id);
    WHILE v.name # id DO v := v.next END;
    y := v
    | LSB.and, LSB.or, LSB.mux, LSB.not, LSB.buf: y := LSB.New(SHORT(x.tag), E(x.a), E(x.b));
    | LSB.reg, LSB.mux1, LSB.reg1, LSB.latch, LSB.tsg: y := LSB.New(SHORT(x.tag), E(x.a), NIL, E(x.b));
    | LSC.lit: IF x.val = 0 THEN y := LSB.zero ELSE IF x.val = 1 THEN y := LSB.one ELSE y := clk END END;
    RETURN y
END E;
PROCEDURE Link(fp: LSB.Variable; ap: LSB.Signal);
VAR fel, ael: LSB.Variable;
BEGIN
    IF fp.fct = LSB.array THEN
        fel := fp.dsc; ael := ap(LSB.Variable).dsc;
        WHILE (fel # NIL) & (ael # NIL) DO Link(fel, ael); ael := ael.next; fel := fel.next END;
        IF (fel # NIL) OR (ael # NIL) THEN
            Texts.WriteString(W, 'array mismatch ');
            LSB.WriteName(W, fp);
            Texts.WriteLn(W);
            Texts.Append(LSB.Log, W.buf);
        END;
        fp.x := ap
    END;
END Link;
PROCEDURE S(s: LSC.Item);
VAR tag, lim, u, v: INTEGER;
    x, cond, cv, ap: LSC.Item;
    y: LSB.Signal;
    tempscope, fp: LSB.Variable;
BEGIN
    WHILE s # NIL DO
        x := s.a; s := s.b; tag := x.tag;
        CASE tag OF
            LSC.assign: y := E(x.a);
            IF y.x # NIL THEN
                Texts.WriteString(W, 'mult ass ');
                LSB.WriteName(W, y(LSB.Variable));
                Texts.WriteLn(W);
                Texts.Append(LSB.Log, W.buf);
            END;
            y.x := E(x.b)
            | LSC.tsass, LSC.ocass: y := E(x.a);
            | LSC.clkass: clk := E(x.a);
            | LSC.posass: y := E(x.a);
            | LSC.if: cond := x.a;
            | x := x.b;
            | CASE cond.tag OF
                | LSC.eql: IF u = v THEN S(x.a) ELSE S(x.b) END
                | LSC.neq: IF u # v THEN S(x.a) ELSE S(x.b) END
                | LSC.lss: IF u < v THEN S(x.a) ELSE S(x.b) END
                | LSC.geq: IF u >= v THEN S(x.a) ELSE S(x.b) END
                | LSC.leq: IF u <= v THEN S(x.a) ELSE S(x.b) END
                | LSC.gtr: IF u > v THEN S(x.a) ELSE S(x.b) END
            END
            | LSC.for: cv := x.a;
            | x := x.b;
            | scope := LSB.NewVar(LSB.integer, SHORT(V(x.a)), NIL, NIL, scope, cv(LSC.Object).name);
            | x := x.b;
            | WHILE scope.val <= lim DO S(x.a);
            | INC(scope.val) END;
            | scope := scope.next
            | LSC.call: y := E(x.a);
            | IF y(LSB.Variable).dsc # NIL THEN
            | S(x.b); ap := x.b;
            | WHILE fp.fct = LSB.integer DO fp := fp.next END;
            | WHILE (ap # NIL) & (ap.tag # LSC.type) DO
            | Link(fp, E(ap.a)); ap := ap.b
        END;
    END;
    RETURN;
END S;
PROCEDURE Expand;
VAR obj: LSC.Object;
root, new: LSB.Variable;
y: LSB.Signal;
BEGIN Texts.WriteString(W, "expanding "); Texts.WriteString(W, LSC.globalScope.name);
  Texts.WriteLn(W); Texts.Append(LSB.Log, W.buf);
  obj := LSC.localScope.next; new := NIL;
  WHILE (obj # LSC.guard) & (obj.tag = LSC.const) DO
    new := LSB.NewVar(LSB.integer, SHORT(V(obj.b)), NIL, NIL, new, obj.name);
    scope := new; obj := obj.next
  END;
  WHILE obj # LSC.guard DO
    new := NewVar(obj.a, obj.tag, obj.name, new, NIL); obj := obj.next
  END;
  scope := new; clk := LSB.clk; S(LSC.body);
NEW(root); root.fct := LSB.record; COPY(LSC.globalScope.name, root.name); root.dsc := new;
LSB.Assign(root);
LSB.Simplify(LSB.org);
LSB.Loops(LSB.org)
END Expand;

PROCEDURE Compile;
VAR beg, end, time: LONGINT;
S: Texts.Scanner; T: Texts.Text; v: Viewers.Viewer;
BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
  IF S.class = Texts.Char THEN
    IF S.c = '*' THEN
      v := Oberon.MarkedViewer();
      IF (v.dsc # NIL) & (v.dsc.next IS TextFrames.Frame) THEN
        LSC.Module(v.dsc.next(TextFrames.Frame).text, 0); Expand
      END
    END
    ELSIF S.c = '@' THEN
      Oberon.GetSelection(T, beg, end, time);
      IF time >= 0 THEN LSC.Module(T, beg); Expand END
    END
  END
END Compile;

PROCEDURE Show;
BEGIN
  IF LSB.org # NIL THEN LSB.Show(LSB.org, −1) END
END Show;

BEGIN Texts.OpenWriter(W); Texts.WriteString(W, "Lola–System NW 15.4.95");
  Texts.WriteLn(W); Texts.Append(LSB.Log, W.buf)
END Lola.
MODULE Simulator; (*NW 27.12.92 / 15.4.95*)
IMPORT Texts, Oberon, TextFrames, ParcElems, LSB;

CONST clash = 2; undef = 3; (*signal values*)
BaseTyps = {LSB.bit, LSB.ts, LSB.oc}; Struct = {LSB.array, LSB.record};
Tab = 9X;

VAR rorg: LSB.Signal; (*register list*)
stepno: INTEGER;
sym: ARRAY 4 OF CHAR;
and, or, xor: ARRAY 4, 4 OF SHORTINT;
not: ARRAY 4 OF SHORTINT;
W: Texts.Writer;

PROCEDURE assign(v: LSB.Variable);
PROCEDURE value(s: LSB.Signal): SHORTINT;
VAR w, h: SHORTINT;
BEGIN
IF s # NIL THEN
  IF s IS LSB.Variable THEN assign(s(LSB.Variable)); w := s.val
  ELSE CASE s.fct OF
  | LSB.or: w := or[value(s.x), value(s.y)]
  | LSB.xor: w := xor[value(s.x), value(s.y)]
  | LSB.and: w := and[value(s.x), value(s.y)]
  | LSB.not: w := not[value(s.y)]
  | LSBmux: h := value(s.x);
    IF h = undef THEN w := undef
    ELSIF h = 0 THEN w := value(s.y.x)
    ELSE w := value(s.y.y)
  END
  | LSB.reg: w := s.val; s.x := rorg; rorg := s
  | LSB.latch: h := value(s.x);
    IF h = undef THEN w := undef
    ELSIF h = 0 THEN w := s.val
    ELSE w := value(s.y); s.val := w
  END
  | LSB.sr: h := value(s.x); w := value(s.y);
    IF (h = undef) OR (w = undef) THEN w := undef
    ELSIF h = 0 THEN
      IF w = 0 THEN w := clash ELSE w := 1; s.val := 1 END
    ELSIF w = 0 THEN w := 0; s.val := 0
    ELSE w := s.val
  END
  END
  ELSE w := undef
  END
  RETURN w
END value;
PROCEDURE assign(v: LSB.Variable);
VAR lnk, tsg: LSB.Signal;
BEGIN
  IF v.val = LSB.black THEN
    v.val := LSB.grey;
    IF v.fct = LSB.bit THEN w := value(v.x);
    ELSIF v.fct = LSB.ts THEN
      lnk := v.x; h := 0; w := undef;
      LOOP
        IF lnk = NIL THEN EXIT END;
        tsg := lnk.x; h := value(tsg.x);
        IF h = 1 THEN
          w := value(tsg.y);
          REPEAT lnk := lnk.y UNTIL (lnk = NIL) OR (value(lnk.x) # 0);
          IF lnk # NIL THEN w := clash; EXIT END
        ELSIF h = 0 THEN lnk := lnk.y
        ELSE EXIT
      END
    END
    ELSE w := undef
  END;
  RETURN w
END assign;
lnk := v.x; w := 1;
WHILE (lnk # NIL) & (w = 1) DO w := value(lnk.x); lnk := lnk.y END
END;
v.val := w
ELSIF v.val = LSB.grey THEN
LSB.WriteName(W, v); Texts.WriteString(W, ' in loop'); Texts.WriteLn(W)
END
END assign;
PROCEDURE evaluate(v: LSB.Variable);
BEGIN (*compute new values of variables*)
IF v.fct IN BaseTyps THEN assign(v)
ELSIF v.fct IN Struct THEN v := v.dsc;
WHILE v # NIL DO evaluate(v); v := v.next END
END
END evaluate;
PROCEDURE initval(v: LSB.Variable);
BEGIN
IF v.fct IN BaseTyps THEN
IF v.x # NIL THEN v.val := LSB.black END
ELSIF v.fct IN Struct THEN v := v.dsc;
WHILE v # NIL DO initval(v); v := v.next END
END
END initval;
PROCEDURE list(v: LSB.Variable);
BEGIN
IF v.fct IN BaseTyps THEN
IF v.u = 0 THEN Texts.Write(W, Tab); Texts.Write(W, sym[v.val]) END
ELSIF v.fct IN Struct THEN v := v.dsc;
WHILE v # NIL DO list(v); v := v.next END
END
END list;
PROCEDURE Step*;
VAR i: LONGINT; r: LSB.Signal;
S: Texts.Scanner;
BEGIN
IF LSB.org # NIL THEN
Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
IF S.class = Texts.Int THEN
initval(LSB.org); rorg := NIL; evaluate(LSB.org); i := S.i;
WHILE i > 0 DO
r := rorg; (*compute new values of register inputs*)
WHILE r # NIL DO
IF value(r.y.x) = 1 THEN (*enabled*) r.y.val := value(r.y.y) END;
   r := r.x
END;
   r := rorg; (*tick: replace old values of registers by new values*)
WHILE r # NIL DO list(r); r := r.next END
END;
END list;
PROCEDURE rst1(s: LSB.Signal);
BEGIN
IF (s # NIL) & ˜(s IS LSB.Variable) THEN
IF s.fct = LSB.reg THEN s.val := 0 END;
rst1(s.x); rst1(s.y)
END
END rst1;
PROCEDURE rst0(v: LSB.Variable);
BEGIN
IF v.fct IN BaseTyps THEN rst1(v.x)
ELSIF v.fct IN Struct THEN v := v.dsc;
WHILE v # NIL DO rst0(v); v := v.next END
END
PROCEDURE Reset; BEGIN IF LSB.org # NIL THEN stepno := 0; rst0(LSB.org) END Reset; END Reset;

PROCEDURE Set; VAR v: LSB.Variable; S: Texts.Scanner; BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); LOOP Texts.Scan(S); IF (S.class # Texts.Name) & (S.class # Texts.String) THEN EXIT END; v := LSB.This(LSB.org, S.s); Texts.Scan(S); IF (S.class = Texts.Char) & (S.c = '=') THEN Texts.Scan(S) END; IF S.class # Texts.Int THEN EXIT END; IF v # NIL THEN Texts.WriteString(W, ' '); LSB.WriteName(W, v); IF (v.x = NIL) OR (v.fct = LSB.ts) THEN IF S.i = 0 THEN v.val := 0 ELSEIF S.i = 1 THEN v.val := 1 ELSE v.val := undef END; Texts.Write(W, '='); Texts.Write(W, sym[v.val MOD 4]) ELSE Texts.WriteString(W, ' not an input') END END END; Texts.WriteLn(W); Texts.Append(LSB.Log, W.buf) END Set;

PROCEDURE lab(v: LSB.Variable); BEGIN IF v.fct IN BaseTyps THEN IF v.u = 0 THEN Texts.Write(W, Tab); LSB.WriteName(W, v) END ELSEIF v.fct IN Struct THEN v := v.dsc; WHILE v # NIL DO lab(v); v := v.next END END lab; END Lab;

PROCEDURE Labels; BEGIN IF LSB.org # NIL THEN lab(LSB.org); Texts.WriteLn(W); Texts.Append(LSB.Log, W.buf) END END Labels;

PROCEDURE clrSel(v: LSB.Variable); BEGIN IF v.fct IN BaseTyps THEN v.u := 1 ELSEIF v.fct IN Struct THEN v := v.dsc; WHILE v # NIL DO clrSel(v); v := v.next END END clrSel;

PROCEDURE Select; VAR i: LONGINT; v: LSB.Variable; S: Texts.Scanner; BEGIN IF LSB.org # NIL THEN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S); clrSel(LSB.org); i := 0; WHILE (S.class = Texts.Name) & (i < 14) DO v := LSB.This(LSB.org, S.s); IF v # NIL THEN v.u := 0; INC(i) END; Texts.Scan(S) END END Select; END Select;

PROCEDURE GetStepNo; BEGIN Texts.WriteString(W, 'step no = '); Texts.WriteInt(W, stepno, 4); Texts.WriteLn(W); Texts.Append(LSB.Log, W.buf) END GetStepNo;

PROCEDURE Init; BEGIN stepno := 0; rorg := NIL END Init;
END Init;

PROCEDURE DefOps;
VAR i, j: INTEGER;
BEGIN
  FOR i := 0 TO undef DO
    FOR j := 0 TO undef DO
      or[i, j] := undef; and[i, j] := undef; xor[i, j] := undef
    END;
    not[i] := undef
  END;
  or[0, 0] := 0; or[0, 1] := 1; or[1, 0] := 1; or[1, 1] := undef;
  and[0, 0] := 0; and[0, 1] := 0; and[1, 0] := 0; and[1, 1] := 1;
  xor[0, 0] := 0; xor[0, 1] := 1; xor[1, 0] := 1; xor[1, 1] := 0;
  not[0] := 1; not[1] := 0
END DefOps;

PROCEDURE SetTabs;
CONST mm = 36000; u = 10000;
VAR i: INTEGER; p: TextFrames.Parc;
BEGIN
  NEW(p); p.W := 100; p.H := 3*mm; p.handle := ParcElems.Handle;
  p.first := 0; p.left := 0; p.width := 165*mm; p.lead := mm; p.lsp := 14*u;
  p.dsr := 2*u; p.opts := {1}; p.nofTabs := 16; i := 0;
  REPEAT p.tab[i] := 11*mm*(i+1); INC(i) UNTIL i = 18;
  Texts.WriteElem(W, p); Texts.Append(LSB.Log, W.buf)
END SetTabs;

BEGIN Texts.OpenWriter(W); SetTabs; DefOps;
END Simulator.
MODULE GAL1; (*NW 26.2.95 / 10.3.95*)
IMPORT SYSTEM, LSB, Texts, Oberon;

CONST K = 10; (*nof registers*)
M = 44; (*nof And terms per row*)
N = 132; (*nof And rows*)
G0 = 0FFFF0000H; (*output to latch, input from GAL*)
G1 = 0FFFF0040H; (*pulse to GAL clock input, pin 2*)
G3 = 0FFFF00C0H; (*program port, d0 = SDI, d1 = MODE, d4 = SCLK*)
inputs = {2, 7, 9, 13, 16}; outputs = {17, 21, 23, 27}; (*pin numbers*)

TYPE Row = ARRAY N OF SHORTINT; (*in map diagram this is a column*)
VAR res: INTEGER;
W: Texts.Writer;
imap, smap: ARRAY 28 OF SHORTINT; (*index is pin number*)
jmap: ARRAY K+1 OF INTEGER; (*index is OMLC no.*)
S0, S1, S2: ARRAY K OF SHORTINT; (*architecture rows*)
D: ARRAY M OF Row; (*and−matrix, first index horizontal in map diagram*)

(* A zero entry in D means 'connection', a one means 'no connection'; initially, all
D[i, j] = 1, as set by Erase command. Programming can create zeroes only.
A term with value 1 is obtained by leaving all connections open, a zero by
connecting to some signal and to its inverse also. All tri−state enables active (1).
Note that input at pin 2 is clock. Preset and Reset set to zero *)

PROCEDURE wait(n: LONGINT);
VAR T: LONGINT;
BEGIN T := Oberon.Time() + n DIV 3;
REPEAT UNTIL Oberon.Time() > T
END wait;

PROCEDURE P(x: SHORTINT);
BEGIN (*pulse*)
SYSTEM.PUT(G3, x); SYSTEM.PUT(G3, x+10H); SYSTEM.PUT(G3, x)
END P;

PROCEDURE EnterCmd(c: SHORTINT);
VAR k: INTEGER;
BEGIN P(3); k := 0; (*enter Load Cmd state, then shift in 5 bits*)
REPEAT P(c MOD 2); c := c DIV 2; INC(k) UNTIL k=5 ;
P(3) (*enter exec state*)
END EnterCmd;

PROCEDURE WriteRow(s: SHORTINT; VAR d: ARRAY OF SHORTINT);
VAR k: INTEGER;
BEGIN EnterCmd(2); (*shift data in*)
k := 0;
REPEAT P(d[k]); INC(k) UNTIL k = N;
k := 0;
REPEAT P(s MOD 2); s := s DIV 2; INC(k) UNTIL k = 6;
EnterCmd(7); (*program*)
P(0); wait(80)
END WriteRow;

PROCEDURE Arch;
VAR k: INTEGER;
BEGIN EnterCmd(20); (*shift in architecture data*)
k := 0;
REPEAT P(S1[k]); P(S0[k]); INC(k) UNTIL k = 10;
EnterCmd(7); (*program*)
P(0); wait(80)
END Arch;

PROCEDURE Erase;
BEGIN EnterCmd(3); P(0); wait(200)
END Erase;

(*----------------------------------------------*)

PROCEDURE Msg(v: LSB.Variable; text: ARRAY OF CHAR);
BEGIN LSB.WriteNam(e(W, v)); Texts.WriteString(W, text); Texts.WriteLn(W)
END Msg;
PROCEDURE zero(e: INTEGER);
BEGIN D[0, e] := 0; D[1, e] := 0
END zero;

PROCEDURE factor(s: LSB.Signal; j: INTEGER);
VAR inv: SHORTINT;
BEGIN
  IF s.fct = LSB.not THEN inv := 1; s := s.y ELSE inv := 0 END;
  IF s IS LSB.Variable THEN
    IF s IN inputs + outputs THEN
      IF (s.u IN outputs) & (S2[smap[s.u]] = 0) THEN inv := 1 - inv END;
      D[imap[s.u]+inv, j] := 0
    ELSEIF s = LSB.zero THEN zero(j)
    ELSEIF s # LSB.one THEN Msg(s(LSB.Variable), 'with bad pin number'); res := 0
    END
    ELSE Texts.WriteString(W, 'illegal expression'); Texts.WriteLn(W); res := 0
    END
  END factor;

PROCEDURE term(s: LSB.Signal; j: INTEGER);
BEGIN
  WHILE s.fct = LSB.and DO factor(s.y, j); s := s.x END;
  factor(s, j)
END term;

PROCEDURE expression(s: LSB.Signal; k: INTEGER);
VAR j, lim: INTEGER;
BEGIN
  j := jmap[k]+1; lim := jmap[k+1];
  WHILE s.fct = LSB.or DO
    term(s.y, j); INC(j);
    IF j = lim THEN
      Texts.WriteString(W, 'too many terms'); Texts.WriteLn(W); DEC(j, 8); res := 0
      END
      s := s.x
    END;
  term(s, j); INC(j)
END expression;

PROCEDURE Prescan(v: LSB.Variable);
VAR s: LSB.Signal; k: INTEGER;
BEGIN
  WHILE v # NIL DO
    IF v.class = LSB.out THEN
      IF v.fct = LSB.bit THEN
        v := v.next
        END
      END
      ELSEIF v.fct = LSB.array THEN Prescan(v.dsc)
      ELSE Msg(v, 'bad signal type'); res := 0
      END
      END
    END;
  Texts.Append(LSB.Log, W.buf)
END Prescan;

PROCEDURE Evaluate(v: LSB.Variable);
VAR s: LSB.Signal;
BEGIN
  WHILE v # NIL DO
    IF v.class = LSB.out THEN
      IF v.fct = LSB.bit THEN
        LSB.WriteName(W, v); Texts.WriteLine(W);
        s := v.x
        IF s.fct = LSB.not THEN s := s.y END;
        IF s.fct = LSB.reg THEN
          IF (s.x = NIL) OR (s.x # LSB.clk) OR (s.y # LSB.one) THEN
            Msg(v, 'register with clock or enable specification'); res := 0
          END
        END
      ELSEIF v.fct = LSB.array THEN Prescan(v.dsc)
      ELSE Msg(v, 'bad signal type'); res := 0
      END
      END
    END;
END Evaluate;
PROCEDURE Make;  
VAR i, j: INTEGER;  
BEGIN res := 1; Texts.WriteString(W, 'making map');  
Texts.WriteLine(W); Texts.Append(LSB.Log, W.buf);  
FOR i := 0 TO M−1 DO (*erase all*)  
FOR j := 0 TO N−1 DO D[i, j] := 1 END  
END;  
FOR i := 0 TO K−1 DO S0[i] := 1; S1[i] := 1; S2[i] := 1 END ;  
D[M−1, 0] := 0; (*pin 16 to asyn reset*)  
zero(0); zero(N−1); (*zero to preset and reset*)  
Prescan(LSB.org.dsc);  
IF res = 1 THEN  
Evaluate(LSB.org.dsc);  
IF res = 1 THEN  
Texts.WriteString(W, 'programming');  
Texts.WriteLine(W); Texts.Append(LSB.Log, W.buf);  
Erase; j := 0;  
REPEAT WriteRow(SHORT(j), D[j]); INC(j) UNTIL j = M;  
Arch; EnterCmd(0); P(2); SYSTEM.PUT(G3, 0) (*idle*)  
END;  
END;  
END Make;  
PROCEDURE Verify;  
VAR k: INTEGER; i, s, n: SHORTINT;  
d: ARRAY 6 OF SHORTINT;  
v: ARRAY N OF SHORTINT;  
BEGIN i := 0;  
REPEAT EnterCmd(2); k := 0; (*shift zeroes in*)  
REPEAT P(0); INC(k) UNTIL k = N;  
s := i; k := 0; (*shift in row number*)  
REPEAT P(s MOD 2); s := s DIV 2; INC(k) UNTIL k = 6;  
EnterCmd(10); P(0); wait(4); (*verify*)  
EnterCmd(2); SYSTEM.PUT(G3, 0); (*shift out*) k := 0;  
REPEAT SYSTEM.GET(G3, d[k]); P(0); INC(k) UNTIL k = 6;  
REPEAT DEC(k); n := n*2 + (d[k] MOD 2) UNTIL k = 0;  
(*compare*)  
IF n # i THEN  
Texts.WriteString(W, 'error row no.'); Texts.WriteInt(W, s, 4); Texts.WriteLine(W);  
END;  
END;  
WHILE (k < N) & (V[k] MOD 2 = D[i, k]) DO INC(k) END;  
IF k < N THEN  
Texts.WriteString(W, 'error in row'); Texts.WriteInt(W, i, 4); Texts.WriteInt(W, k, 4);  
Texts.WriteInt(W, D[i, k], 4); Texts.WriteInt(W, V[k], 6);  
Texts.WriteLine(W)  
END;  
INC(i)  
END;  
END Verify;  
(*--------------------------*)  
PROCEDURE ShowMap;  
VAR i, j: INTEGER;  
BEGIN j := 0;  
REPEAT Texts.WriteLine(W, j, 4); Texts.Write(W, ':'); i := 0;  
REPEAT  
IF D[i, j] = 0 THEN Texts.WriteLine(W, i, 4) END;  
INC(i)  
UNTIL i = M;  
EnterCmd(0); P(2); Texts.Append(LSB.Log, W.buf)  
END ShowMap;
UNTIL i = M;
  Texts.WriteLn(W); INC(j)
UNTIL j = N;
  Texts.WriteString(W, 'SO:'); i := 0;
REPEAT Texts.WriteInt(W, S0[i], 2); INC(i) UNTIL i = K;
  Texts.WriteString(W, 'S1:'); i := 0;
REPEAT Texts.WriteInt(W, S1[i], 2); INC(i) UNTIL i = K;
Texts.WriteString(W, 'STOP'); Texts.Append(LSB.Log, W.buf)
END ShowMap;

PROCEDURE WriteByte(x: INTEGER);
  VAR s0, s1: SHORTINT;
  BEGIN s0 := SHORT(x) MOD 10H; s1 := SHORT(x DIV 10H) MOD 10H;
    IF s0 < 10 THEN INC(s0, 30H) ELSE INC(s0, 37H) END;
    IF s1 < 10 THEN INC(s1, 30H) ELSE INC(s1, 37H) END;
    Texts.Write(W, ' '); Texts.Write(W, CHR(s1)); Texts.Write(W, CHR(s0))
  END WriteByte;

PROCEDURE PutS;
  VAR S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
    SYSTEM.PUT(G3, SHORT(SHORT(S.i)) MOD 10H)
  END PutS;

PROCEDURE Get;
  VAR x: SHORTINT;
  BEGIN SYSTEM.GET(G0, x); WriteByte(x); Texts.Append(Oberon.Log, W.buf)
  END Get;

PROCEDURE Clock;
  BEGIN SYSTEM.PUT(G1, 0)
  END Clock;

BEGIN Texts.OpenWriter(W);
  imap[26] := 6; imap[27] := 2;
  smap[18] := 8; smap[17] := 9;
END GAL1.
module DNF; (*NW 25.3.95*)
(*convert Boolean expression to disjunctive normal form*)
import Texts, LSB;

procedure copy(s: LSB.Signal): LSB.Signal;
begin
  if (s IS LSB.Variable) OR (s.fct = LSB.not) then return s
  else return LSB.New(s.fct, copy(s.x), copy(s.y))
end
end copy;

procedure P(var s: LSB.Signal; inv: BOOLEAN);
begin (*remove not, xor, and mux*)
  if s IS LSB.Variable then
    if s(LSB.Variable).class = LSB.var then s := copy(s.x); P(s, inv)
    elif inv then s := LSB.New(LSB.not, NIL, s)
    end
  elsif s.fct = LSB.not then
    if inv then s := s.y; P(s, FALSE)
    else s := LSB.New(LSB.not, NIL, s)
    end
  elsif s.fct = LSB.and then
    if inv then s.fct := LSB.or end;
    P(s.x, inv); P(s.y, inv)
  elsif s.fct = LSB.or then
    if inv then s.fct := LSB.and end;
    P(s.x, inv); P(s.y, inv)
  elsif s.fct = LSB.xor then
    s.fct := LSB.or;
    s.x := LSB.New(LSB.and, s.x, s.y);
    s.y := LSB.New(LSB.and, copy(s.x.x), copy(s.x.y));
    P(s.x.x, FALSE); P(s.x.y, inv); P(s.y.x, TRUE); P(s.y.y, TRUE)
  elsif s.fct = LSB.mux then
    s.fct := LSB.or; s.y.fct := LSB.and;
    s.x := LSB.New(LSB.and, s.x, s.y);
    s.y.x := copy(s.x.x);
    P(s.x.x, TRUE); P(s.x.y, inv); P(s.y.x, FALSE); P(s.y.y, TRUE)
  else halt(88)
  end
end P;

procedure Q(var s: LSB.Signal);
var t: LSB.Signal;
begin
  if ~(s IS LSB.Variable) & (s.fct # LSB.not) then
    if s.fct = LSB.or then
      Q(s.x); Q(s.y);
      if s.y.fct = LSB.or then
        t := s.y; s.y := tx; tx := s; s := t; Q(s)
      end
    end
    elsif s.fct = LSB.and then
      Q(s.x); Q(s.y);
      if s.x.fct = LSB.or then
        t := s.x; s.x := ty; ty := s;
        tx := LSB.New(LSB.and, s.x, copy(s.y)); s := t; Q(s)
      elsif s.y.fct = LSB.and then
        t := s.y; s.y := tx; tx := s;
        ty := LSB.New(LSB.and, copy(s.x), ty); s := t; Q(s)
      end
    else halt(99)
  end
end Q;

procedure R(var s: LSB.Signal);
begin
  P(s, FALSE); Q(s)
end R;

procedure elem(var s: LSB.Signal);
begin
  if s.fct = LSB.not then
IF s.y.fct = LSB.reg THEN R(s.y) ELSE R(s.y) END
ELSIF s.fct = LSB.reg THEN R(s.y)
ELSE R(s)
END
END elem;

PROCEDURE list(v: LSB.Variable);
BEGIN
  WHILE v # NIL DO
    IF v.class = LSB.out THEN
      IF v.fct = LSB.array THEN list(v.dsc)
      ELSIF (v.fct = LSB.bit) & (v.x # NIL) THEN
        elem(v.x); LSB.Show(v, 0)
      END
    END
  END;
  v := v.next
END
END list;

PROCEDURE Convert;
VAR v: LSB.Variable;
BEGIN list(LSB.org.dsc)
END Convert;

END DNF.