CHAPTER SIX

Lola Programming System

The **H.O.T. Works Development System** requires the understanding of the both hardware design and software programming. In keeping with our goal, of making configurable computing accessible to the software and computer science engineer, VCC has choosen to support Logic Language (LoLa) HDL, and the Lola Programming Development System created around it. We hope you find it useful. VCC wishes to recognize Niklaus Wirth, Stephan Gehring and Stefan Ludwig for their fine work in creating the original Lola/Trianus/Hades Programming System.

We highly recommend the following book on Digital Design:

Digital Circuit Design, An Introductory Textbook
by Niklaus Wirth
ISBN 3-540-58577-X Springer-Verlag

CONTENTS

BLANK PAGE

Lola Programming System

Overview

There are various design entry, editing, mapping, placing and routing tools available from third party vendors for FPGAs. For the users of the XC6200RPU, one of the benefits of Open Architecture is the creation of new types of development software. The Lola Programming System is one of the first of these newer development systems.

The Lola Programming System is located on your harddisk at the following location:

```
C:/Hotworks/Lola
```

Features

- Suite of Tightly Integrated Tools
- Efficient design & implementation

of algorithms on the H.O.T. Works Platform

Contains

- Lola HDL & Lola Compiler,
- Layout Editor,
- Circuit Checker,
- a technology mapper,
- a placer & router,
- and a bit-stream generator

We recommend you read through this entire chapter before trying the Lola Programming System installed on your harddisk. There are various On-Line tutorials.

⇒ Under Windows, the Hotworks Setup installs a folder on the Start Menu Called **Hotworks**. In this folder one of the items is the **Lola Program**

The following figures represent an overview of the Oberon Version of the Lola Programming System. Oberon is a platform independent operating system upon which the original Lola Programming System was written.

Getting started with Oberon for Windows

Oberon is an extensible system. There is no difference between functions offered by the operating system and functions added by a user which enhance the basic system. As a matter of fact, Oberon abandons the notion of a Program altogether. Atomic actions in Oberon are called Commands.

Commands are parameterless procedures which may be executed directly from the operating system interface. Procedures from anywhere in a module hierarchy may be called in this way and there is no need for a "top" module which distributes commands to other modules. Instead, command distribution is built into the operating system. When a command terminates, control is passed back to the operating system's main loop. However, the module that contained the last command remains loaded in the system and all of its global variables remain unaltered. Subsequent commands may reference these variables freely.

The granularity of commands may be quite fine. Typical commands display the directory of a storage device, increase the font size of the text last selected or compile the contents of the active window. A user may execute commands in any sequence and may thus be working on completely different problems in different windows at the same time. We call this "one-process multitasking".

Using Commands

Commands are activated by the use of an Action Button. The Action Button is either the middle mouse button on a three-button mouse or, if no three-button mouse is available, the control key on the keyboard. Pressing the Action Button while pointing at any text anywhere on the screen will try to execute a command of that name. This may be text in a pull-down menu, in a word processor document, or even the title of a window.

The system will then attempt to interpret the text that is pointed at as "Modulename.Commandname" and search for the corresponding module, possibly loading it from disk if it cannot be found in memory. If the module is present or can be loaded, the system will then search for the requested command and execute it. If the module or the command cannot be found, the system will display an appropriate error message.

For example, the text written in bold following this paragraph is a command that will display the current time in the System-Log (the window in the upper right corner). Execute this command by moving the mouse over it and pressing the Action Button within the Oberon window:

System.Time

Extensibility

The main feature of Oberon is extensibility. A user can add an arbitrary number of commands to the system at any time simply by writing a module and compiling it. There is no need to install the command (it may be activated as soon as compilation is completed) and modules containing

commands are not added to the system until an actual call is made. The <working menus> of commands that users keep (called Tools in Oberon) are regular text documents and freely editable.

Even the basic resources of the system are extensible. For example, the editor in the Oberon window supports "live" extensions of characters to which messages are sent when editing operations occur.

To Probe Further

A lot of documentation is supplied in this distribution. Execute the command below in the Oberon window (by pressing the Action Button over the word "Edit.Open") for viewing a first introduction to the built-in mouse commands and the most important basic commands of Oberon:

Edit.Open Oberon.Starter.Text

We have also prepared a Tool that leads you to further documentation about different parts of the system. Please:

System.Open Documentation.Tool

An overview of the Oberon Version of the Lola Programming System follows.

Figure 1	The first window you see when you run **`Lola.exe`**. This is the System Window.
Figure 2	The User Window is where most of the work is done.
Figure3	The System Window is divided into three sub windows.
Figure 4	The System Log Window lets you know what the Lola Programming System is doing.
Figure 5-6	The System Tool Window contains most system commands. It features System.Quit to exit from the Lola Programming System and various Open Commands
Figure 7-8	The Tutorial Tool Window is where you find the On-Line Tutorials. On-Line Tutorial, Directory Buttons and Users X6K Tools Guide & the Oberon Three Button Mouse Guide

The Lola Programming Environment --- Oberon Version

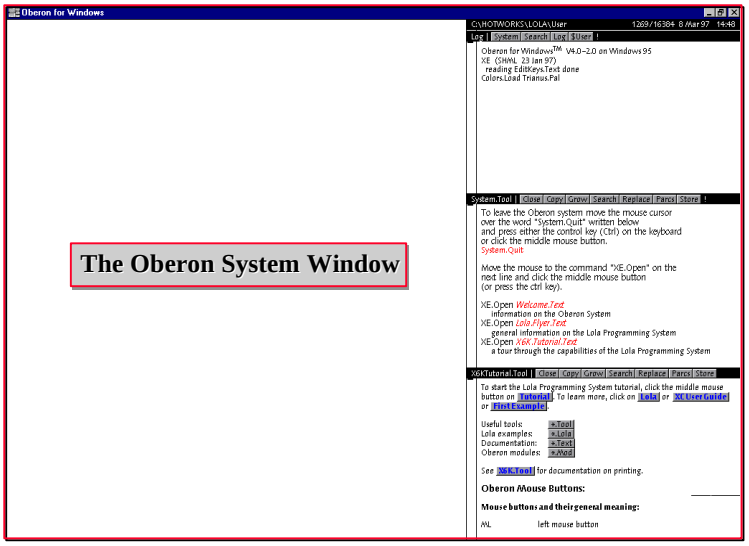


Figure 1

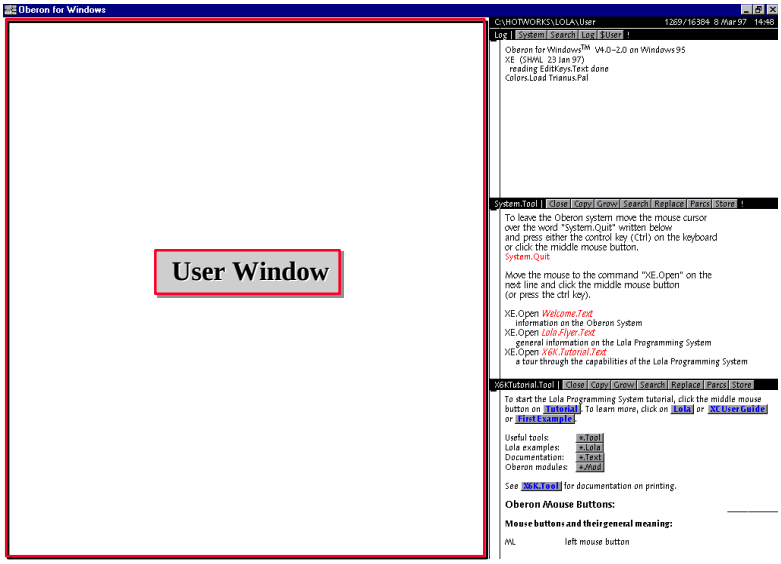
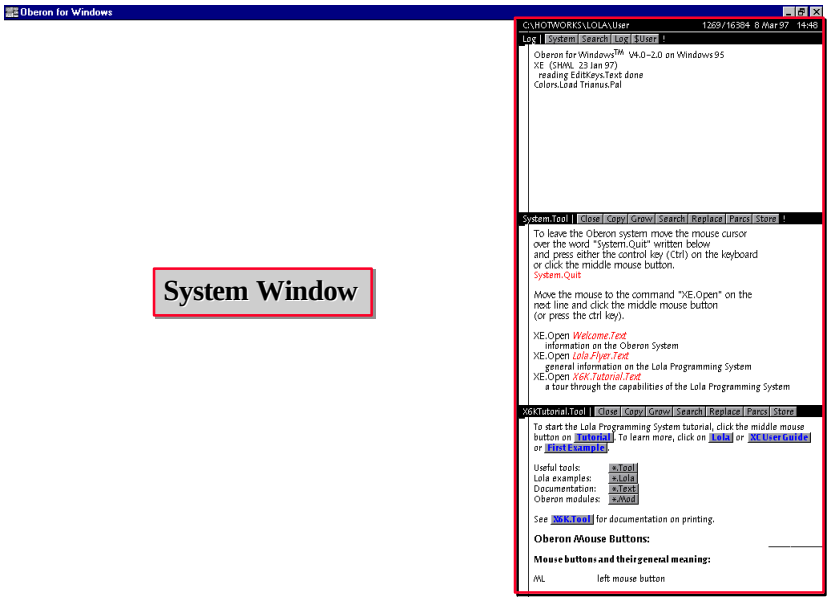
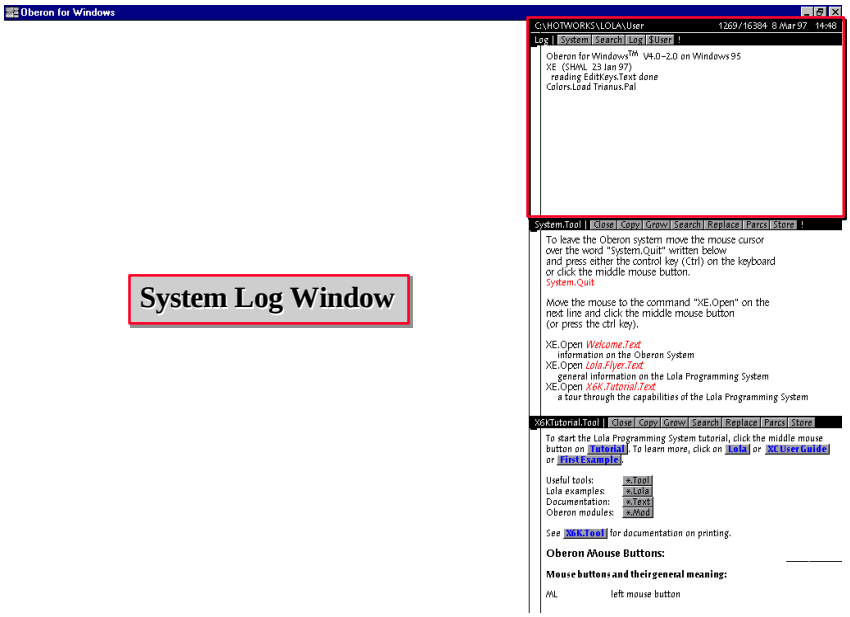


Figure 2



System Window

Figure 3



System Log Window

Figure 4

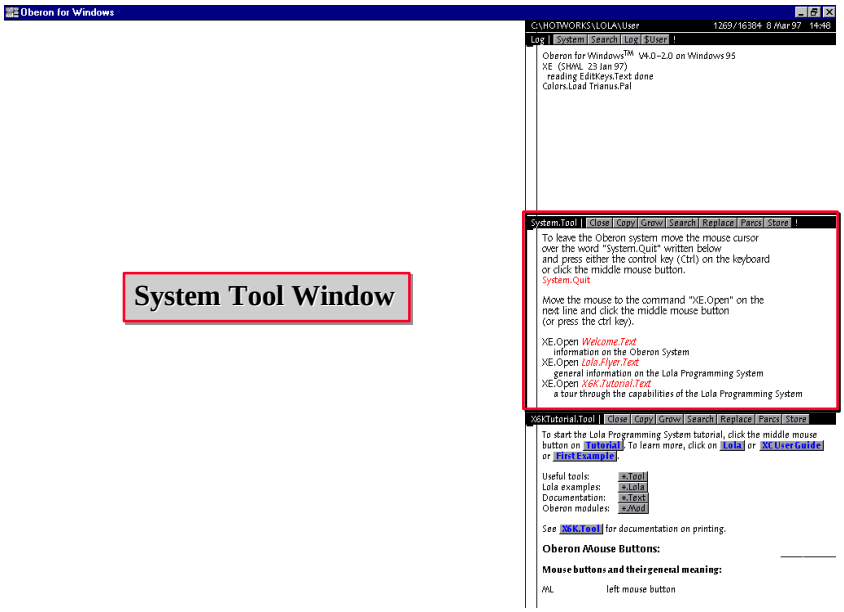


Figure 5

The System Tool Window

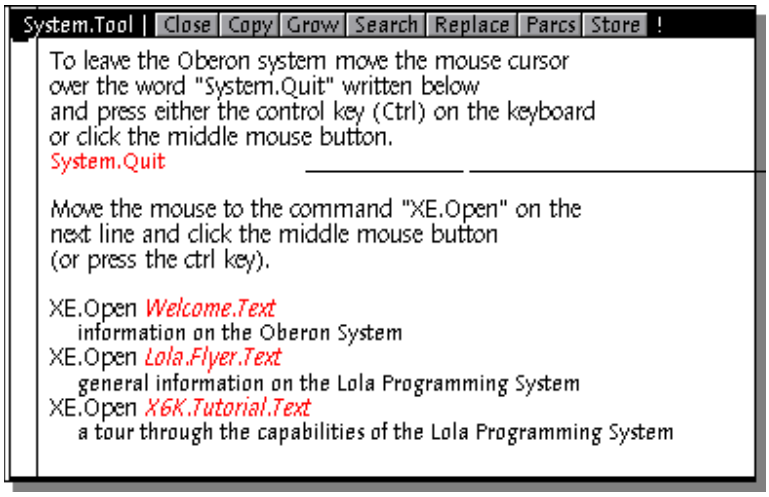


Figure 6

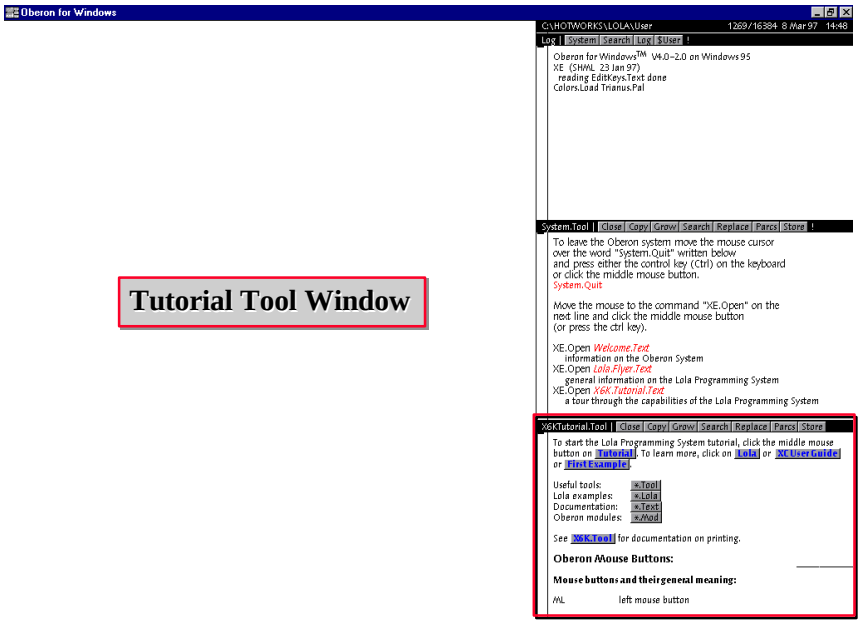


Figure 7

The Tutorial Tool Window

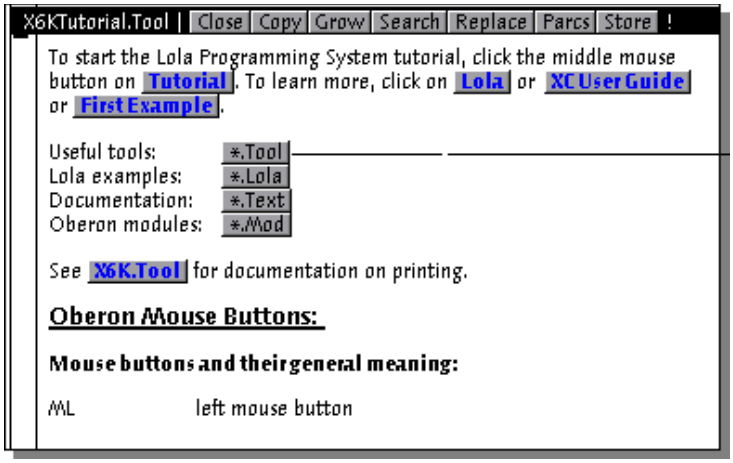


Figure 8

Using the Mouse with the Oberon Version of Lola Programming System

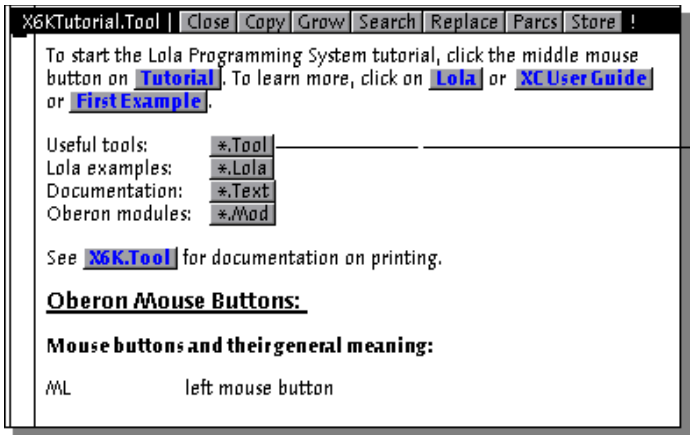


Figure 9

Action
Mouse Scrolls Window Up & Down when the arrow is placed in the left margin.

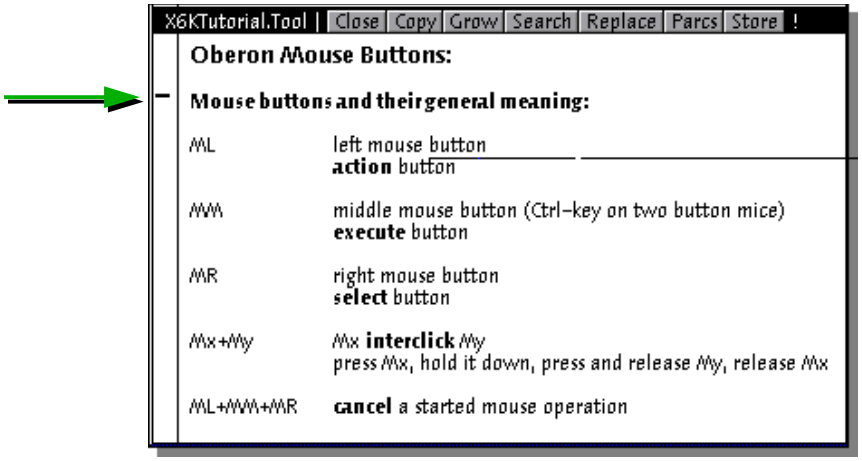


Figure 10

System actions, selections and execution command may all be given with the mouse. You may use a three button mouse or a two button mouse and the Control Key on the keyboard.

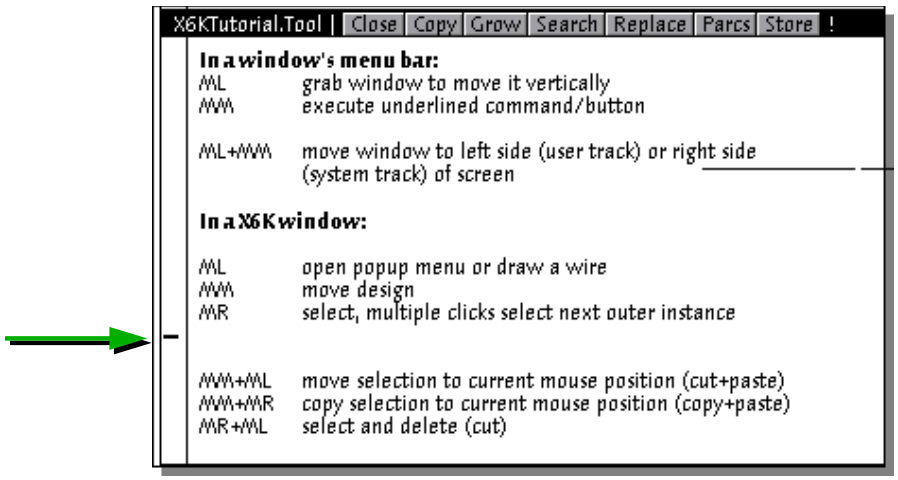


Figure 11

Screen controls are handled by the mouse. You can control window size as well as menu selections.

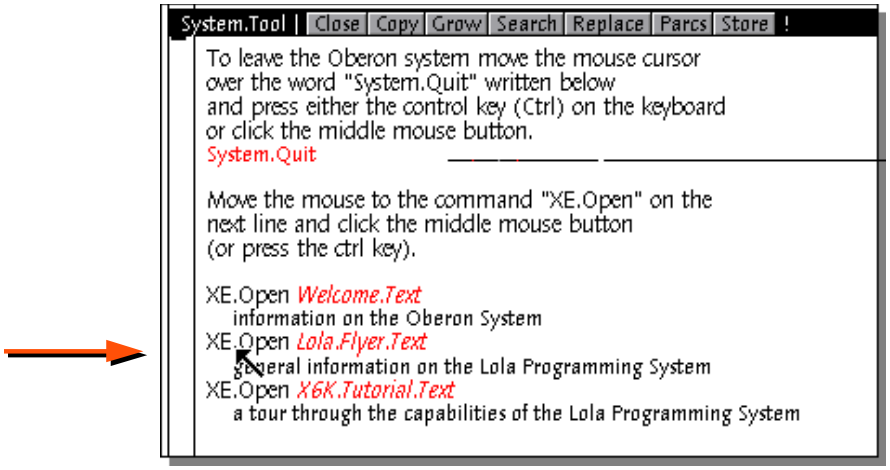


Figure 12

Go to the System.Tool Window and Open the Lola Description Text -*Lola.Flyer.Text*by placing the cursor over***XE.Open Lola Flyer.Text***and press the Middle Mouse Button or Control Key (on a two button mouse).

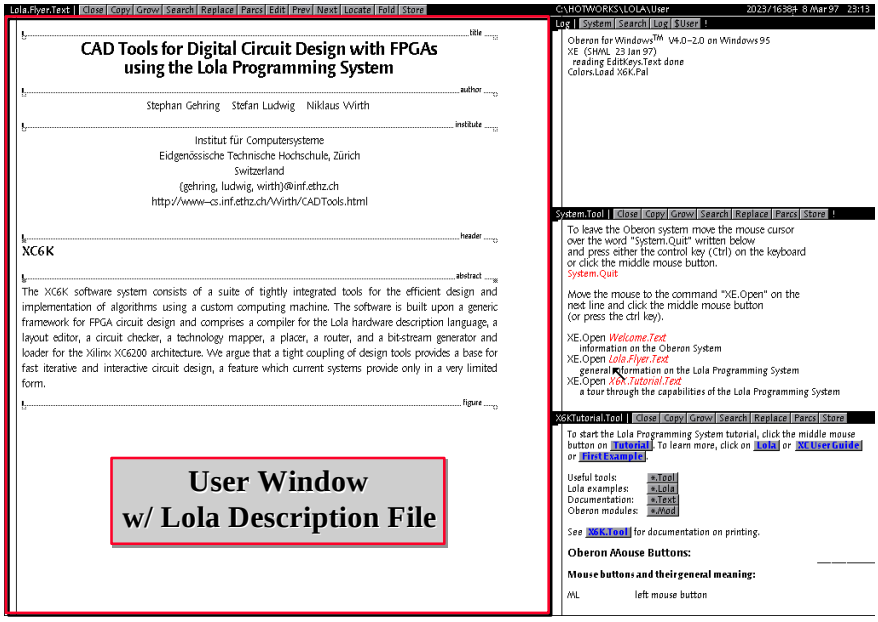


Figure 13

You have just opened the On-Line Tutorial. The following is from the Lola Description Text -- *Lola.Flyer.Text*

CAD Tools for Digital Circuit Design with FPGAs using the Lola Programming System

Stephan Gehring, Stefan Ludwig &Niklaus Wirth
at the Institut für Computersysteme Eidgenössische Technische Hochschule, Zürich
Switzerland

X6K Programming Tools

The **X6K** software system consists of a suite of tightly integrated tools for the efficient design and implementation of algorithms using a custom computing machine. The software is built upon a generic framework for FPGA circuit design and comprises a compiler for the Lola hardware description language, a layout editor, a circuit checker, a technology mapper, a placer, a router, and a bit_stream generator and loader for the Xilinx XC6200 architecture. We argue that a tight coupling of design tools provides a base for fast iterative and interactive circuit design, a feature which current systems provide only in a very limited form.

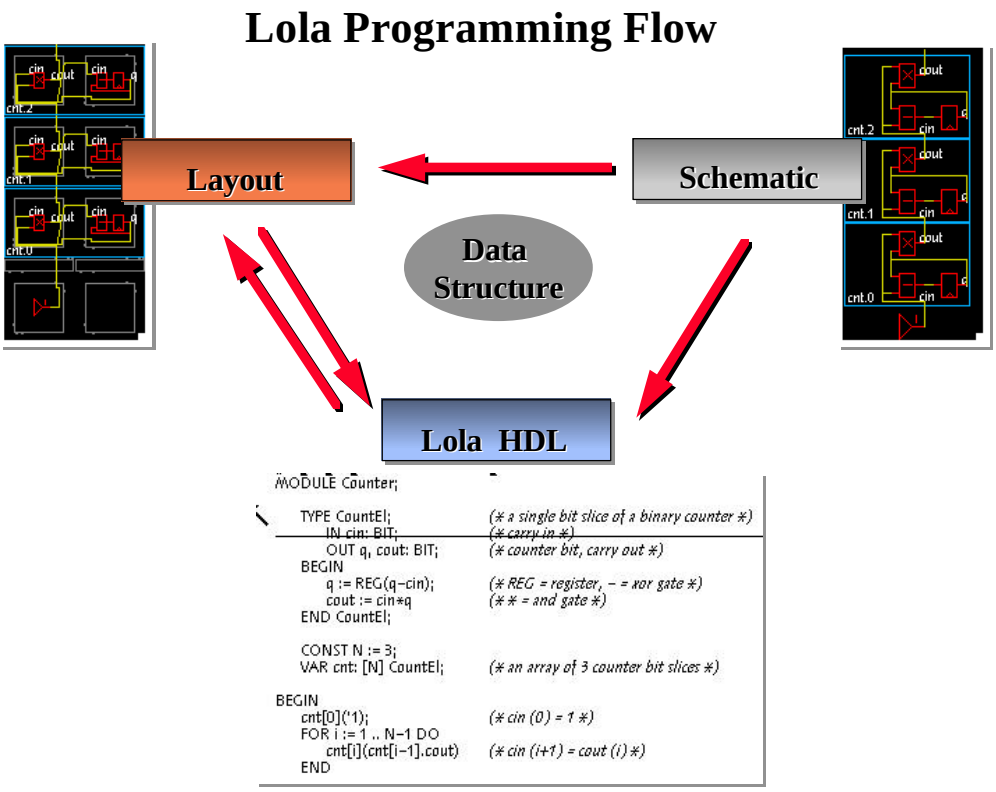


Figure 14

Hades {Some History}

The Hades coprocessor board uses the new XC6200 FPGA family from Xilinx. The board contains a single XC6216 with 64x64 cells and 256 KB of fast local memory. It is accessed from the host (an ETH Ceres-2 workstation) via a conventional memory interface. Due to this, variables in a hardware design can be accessed just like normal program variables.

The current Lola Programming System software release is integrated with the H.O.T. Works Board.

Lola

Lola was designed as a simple, easily learnable hardware description language for describing synchronous, digital circuits. In addition to its use in a digital design course for second year

computer science students at ETH Zürich, the Institute for Computer Systems uses it as a hardware description language for describing hardware designs in general and coprocessor applications in particular. The purpose of Lola is to statically describe the structure and functionality of hardware components and of the connections between them. A Lola text is composed of declarations and statements. It describes the hardware on the gate level in the form of signal assignments. Signals are combined using operators and assigned to other signals. Signals and the respective assignments can be grouped together into types. An instance of a type is a hardware component. Types can be composed of instances of other types, thereby supporting a hierarchical design style and they can be generic (e.g. parametrizable with the word-width of a circuit).

Opening the Lola Descritpion Text

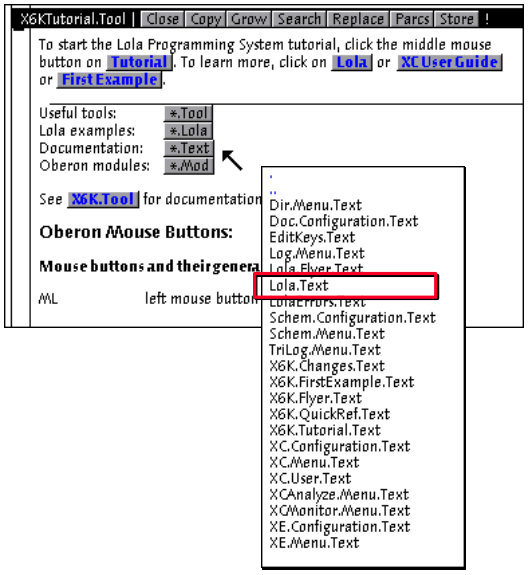


Figure 15

Let's open another window using a second approach. You can also open the Text Window by selecting Documentation: *.Text in the X6K.Tutorial.Tools Window. Place the mouse over the *.Text button and press the Middle Mouse Button or Control Key. Then Select the Lola.Text. The complete Lola.Text follows.

Lola: A Logic Description Language

By Niklaus Wirth Institut für Computersysteme ETH Zürich

Lola is a notation (language) for specifying synchronous, digital circuits (logic). In many ways it resembles a procedural programming language. However, Lola texts describe static circuits rather than dynamic processes. The objects occurring in a description are variables representing signals, operators representing gates, and composite structures of such objects.

1. Identifiers, Integers, Logical Values, and Comments

Identifiers are used to denote constants, variables, and types.

```
identifier =    letter {letter | digit} [""].
integer =       digit {digit}.
LogicValue =    " '0" | " '1".
```

Comments are sequences of characters enclosed by the brackets (* and *), and they may occur between any two symbols within a Lola text.

2. Basic and Simple Types

Every variable in Lola has a type. It is either a basic, predefined type or a structured type defined in the program text. The basic types are denoted by BIT, TS, or OC, which differ only in their rules governing assignment. (TS denotes a tri-state bus, OC an open-collector bus). Variables of a basic type have a logic value, denoted by '0 or '1.

```
SimpleType =      BasicType | identifier ["(" ExpressionList ")"].
BasicType =       "BIT" | "TS" | "OC".
ExpressionList =  expression {"," expression}.
```

3. Array types

Array types consist of an array of elements, all of which have the same type. A numeric expression indicates the number of elements in the array. Elements are identified by an index value. Indices range from 0 to the array's length minus 1.

```
type =          { "[" expression "]" } SimpleType.
```

4. Constant Declarations

Constant declarations serve to introduce identifiers denoting a constant, numeric value.

```
ConstDeclaration =   identifier ":=" expression ";".
```

5. Variable Declarations

Variable declarations serve to introduce identifiers denoting a logic variable and to associate it with a type. All variables declared in an identifier list have the same type.

```
VarDeclaration =     IdList ":" type ";".
IdList =             identifier {"," identifier}.
```

6. Expressions

Expressions serve to combine variables with logical operators to define new values. The operators are negation, logical conjunction (and), disjunction (or), and difference (xor). Operands are of any basic type. Elements of an array are selected by an index: a.5, a.i. If the index is an expression, the form a[exp] is used.

- \+ logical disjunction (or)
- \- logical difference (exclusive or)
- * logical conjunction (and)
- ~ negation (not)

A multiplexer is denoted by MUX(s: a, b) and is equal to ~s*a + s*b.

A register provides the means to specify a value depending on previous values in time (sequential circuit). The value of REG(ck: en, d) in the next clock cycle is equal to d in the current clock cycle, if en = '1. If en = '0, the previous value is retained. en is called the register's enable signal. In the short form REG(en, d), the clock signal does not appear explicitly and is assumed to be a global clock, as is typical for synchronous circuits. The short notation REG(d) stands for REG('1, d).

A latch, denoted by LATCH(g, d), is a storage element which holds a logic value while g = '0. If g = '1, the value d is acquired (and the latch is transparent).A basic SR-latch with (active-low) set and reset signals is expressed as SR(s', r').

Apart from logic expressions there exist numeric expressions. They follow the same rules of composition; their operators are those of addition, subtraction, multiplication, division, and exponentiation and (integer) logarithm to the base 2.

```
selector =      {"." identifier | "." integer | "[" expression [ ".." expression] "]"}.
factor =        identifier selector | LogicValue | integer | "(" expression ")" |
                "MUX" "(" expression ":" expression "," expression ")" |
                "REG" "(" [expression ":"][expression ","] expression ")" |
                "LATCH" "(" expression "," expression ")" |
                "SR" "(" expression "," expression ")" |
                "~" factor | "EXP" factor | "LOG" factor.
term =          factor {("*" | "/" | "DIV" | "MOD") factor}.
expression =    term {("+" | "-") term}.
```

Note that the selector parameter(s) of MUX is followed by a colon instead of a comma.

A selector of the form [m .. n], applied to an array variable a, denotes the subarray a[m], a[m+1], ... , a[n]. It can be used as actual parameter in unit assignments (see 7.2.8) corresponding to a formal input variable.

LOG n (n >= 0) is the binary logarithm, rounded down to the next integer. LOG 0 = 0. EXP x (x >= 0) is equal to 2^x.

7. Assignments

Assignments serve to define a variable's value, which is specified as that of an expression. The form v := x stands for "let v be equal to x". Hence, an assignment must be understood as a variable's definition (in contrast to an identifier's declaration). v and x do not have the same roles, and this asymmetry is emphasized by the use of the symbol := instead of the symmetric equal sign (=).

If a variable is of type BIT, the expression must be of any basic type, and only a single assignment (definition) is allowed.

If the variable's type is TS, the statement must specify a condition (representing a tri-state gate). Arbitrarily many assignment to the same variable are permitted. However, the value of the bus is defined only if some condition's value is '1.

```
assignment =    identifier selector ":=" [condition "|"] expression.
condition =     expression.
```

If the variable's type is OC (open-collector), arbitrarily many assignments to the same bus variable are permitted. The bus value is '1, unless any one of the assigned expressions has the value '0 (wired-or).

8. Control Statements

Statements are either assignments or composites of assignments, namely repeated or conditional assignments.

```
relation =              expression ("=" | "#" | "<" | "<=" | ">" | ">=") expression.
IfStatement =           "IF" relation "THEN" StatementSequence
                        {"ELSIF" relation "THEN" StatementSequence}
                        ["ELSE" StatementSequence]
                        "END" .
constructor =           "[" expression {"," expression} "]" .
parameter =             expression | constructor.
ForStatement =          "FOR" identifier ":=" expression ".." expression "DO"
StatementSequence
                        "END" .
UnitAssignment =        identifier selector "(" parameter {"," parameter} ")".
statement =             [assignment | UnitAssignment | PosAssignment | IfStatement |
ForStatement].
StatementSequence =     statement {";" statement}.
```

The expressions in a for-statement must be numeric, and they specify the range of integer values for which the constituent statement sequence is defined. The identifier associated with the control variable is considered as being local to the for-statement, i.e. does not exist in the for-statement's context. The control variable typically serves as index to array variables.

A constructor denotes a list of signals. It can be used as an actual parameter corresponding to a formal array variable with the same number of elements.

9. Modules

A module specifies variables and a circuit involving these variables. A module may also contain definitions of composite types. Modules are the textual units for compilation.

```
InType =            {"[" expression "]"}"BIT".
InOutType =         {"[" expression "]"} ("TS" | "OC").
OutType =           {"[" expression "]"} ("BIT" | "TS" | "OC").
ImportList  =       "IMPORT" identifier {"," identifier} ";" .
module  =           "MODULE" identifier ";" [ImportList]
```

```
{TypeDeclaration ";"}
["CONST" {ConstDeclaration}]
["IN" {IdList ":" InType ";"}]
["INOUT" {IdList ":" InOutType ";"}]
["OUT" {IdList ":" OutType ";"}]
["VAR" {VarDeclaration}]
["CLOCK" expression ";"]
["BEGIN" StatementSequence]
"END" identifier "." .
```

Note that declarations introduce identifiers for variables, and statements define their values. The identifier at the end of the module's declaration must match the one following the symbol MODULE.

The clock declaration serves to specify the value of the global clock, which is taken as default in a factor of the form REG(en, d) (see Sect. 6).

Example: The following circuit represents an 8-bit binary adder with inputs x (x.0 ... x.7), y (y.0 ... y.7), and the carry ci. Its outputs are the sum s (s.0 ... s.7) and the carry co.

```
MODULE Adder;
    CONST N := 8;
    IN x, y: [N] BIT; ci: BIT;
    OUT s: [N] BIT; co: BIT;
    VAR c: [N] BIT;
BEGIN
    s.0 := x.0 - y.0 - ci; c.0 := (x.0 * y.0) + (x.0 - y.0)*ci;
    FOR i := 1 .. N-1 DO
        s.i := x.i - y.i - c[i-1]; c.i := (x.i - y.i) + (x.i - y.i) * c[i-1]
    END ;
    co := c[N-1]
END Adder.
```

10. Composite Types and Unit Assignments

In addition to basic types and array types, composite types can be declared. This facility may be compared to record types in programming languages, and variables (instances) of such types correspond to components of circuits, i.e. to objects being part of a circuit. A type declaration specifies a composite type, of which instances are introduced by variable declarations. The heading of a type declaration contains up to four sections:

1. The section headed by the symbol IN declares input signals to which no assignments within the type declaration are permitted. The identifiers act as formal names for expressions specified externally in unit assignments, where the expressions appear in the form of parameters. The types of the formal names must be BIT or arrays thereof. The corresponding actual expressions must be of any basic type, or an array thereof.

2. The section headed by the symbol INOUT declares signals to which assignments within the type declaration are permitted. As in the case of inputs, the identifiers act as formal names for signals declared outside the type declaration. Their types must be TS or OC or arrays thereof.

3. The section headed by the symbol OUT declares actual variables. Their type must be BIT, TS, OC, or an array thereof. These output variables are accessible in the scope (type declaration) in which the composite variable is declared. There they are denoted by the composite variable's identifier followed by the output identifier as selector (the latter acting like a field identifier of a record). No assignments are permitted outside the declaration in which the output is declared.

4. The section headed by the symbol VAR declares actual variables. They are not accessible outside the type declaration.

Summary

mode	allowed types	types of corresponding actual parameters
IN	BIT	BIT, TS, OC
INOUT	TS, OC	TS, OC
OUT	BIT, TS, OC	
VAR	BIT, TS, OC, declared type	

Consider the following example:

```
TYPE AddElem;
     IN x, y, ci: BIT;
     OUT z, co: BIT;
     VAR h: BIT;
BEGIN h := x - y; z := h - ci; co := (x * y) + (h * ci)
END AddElem
```

A variable u of type AddElem (i.e. an instance of an AddElem) is introduced by the declaration:

```
u: AddElem
```

The inputs appear in the form of parameters (expressions) in a statement called unit assignment:

```
u(a, b, c)
```

The components of u are obtained by substitution of the actual expressions for the corresponding formal identifiers:

```
u.h := a - b;
u.z := u.h - c;
u.co := (a * b) + (u.h * c)
```

An 8-bit adder with inputs X and Y can now be declared as consisting of 8 identical elements

```
U: [8] AddElem
```

defined by the following assignments:

```
U.0(X.0, Y.0, '0);
FOR i := 1 .. 7 DO U.i(X.i, Y.i, U[i-1].co) END
```

and the sum is represented by the variables U.0.z ... U.7.z .

```
TypeDeclaration =   "TYPE" identifier ["*"] ["(" IdList ")"] ";"
                    ["CONST" {ConstDeclaration}]
                    ["IN" {IdList ":" InType ";"}]
                    ["INOUT" {IdList ":" InOutType ";"}]
                    ["OUT" {IdList ":" OutType ";"}]
                    ["VAR" {VarDeclaration}]
                    ["BEGIN" StatementSequence]
                    "END" identifier.
```

The number of bracket pairs in a formal type specifies the number of indices used for this parameter. If an expression is specified, it indicates the length of the corresponding actual arrays given in unit assignments. The identifier at the end of the declaration must match the one following the symbol TYPE.

11. Generic Types

Declared types can be supplied with parameters and therefore define an entire class of types. The parameters are numeric quantities and are used, for example, to specify the length of arrays. Example:

```
TYPE Counter(N);
    IN ci: BIT;
    OUT co: BIT; q: [N] BIT;
    VAR c: [N] BIT;
BEGIN
    q.0 := REG(q.0 - ci); c.0 := q.0 * ci;
    FOR i := 1 .. N-1 DO q.i := REG(q.i - c[i-1]); ci := q.0 * c[i-1] END ;
    co := c[N-1]
END Counter;
```

An instance u of a counter with 8 elements is declared as

```
u: Counter(8)
```

yielding the variables
u.co, u.q.0. ... , u.q.7 and u.c.0, .. , u.c.7

Note that u.c is local, i.e. not accessible outside the type declaration. A corresponding unit assignment with enable signal e is now expressed by

```
u(e)
```

12. Placement Information

Variables may be attributed with a list of integer values. The language does not specify their semantics. However, their typical use is as coordinates of the positions of parts generating the respective signals in physical devices. The values are assigned to a variable by a position assignment.

```
position =          expression {"," expression} | "[" position {";" position} "]" .
PosAssignment =     identifier selector "::" position.
```

If the attributed variable is an array, a list of positions can be specified, whose elements are attributed to the corresponding elements of the array.

Examples for variables x, y: BIT; z: [4] BIT:

```
x :: 2;    y :: 3, 4, 5;   z :: [6,7; 8,9; 10,11]
```

If variables local to a type declaration are attributed, the specified values are added to the corresponding attributes of every instance of the type. This implies that the attributed coordinates of the local variables are relative to the origin of the instance.

13. Lola Syntax

```
identifier  =          letter {letter | digit} [""].
integer  =             digit {digit}.
LogicValue =           "'0" | "'1".
BasicType =            "BIT" | "TS" | "OC".
SimpleType =           BasicType | identifier ["(" ExpressionList ")"].
ExpressionList =       expression {"," expression}.
type =                 { "[" expression "]" } SimpleType.

ConstDeclaration =     identifier ":=" expression ";".
VarDeclaration =       IdList ":" type ";".
IdList =               identifier {"," identifier}.

selector =             {"." identifier | "." integer | "[" expression [ ".." expression] "]"}.
factor =               identifier selector | LogicValue | integer | "(" expression ")" |
                       "MUX" "(" expression ":" expression "," expression ")" |
                       "REG" "(" [expression ":"][expression ","] expression ")" |
                       "LATCH" "(" expression "," expression ")" |
                       "SR" "(" expression "," expression ")" |
                       "~" factor | "EXP" factor | "LOG" factor.

term =                 factor {("*" | "/" | "DIV" | "MOD") factor}.
expression =           term {("+" | "-") term}.
condition =            expression.
assignment =           identifier selector ":=" [condition "|"] expression.
relation =             expression ("=" | "#" | "<" | "<=" | ">" | ">=") expression.

IfStatement =          "IF" relation "THEN" StatementSequence
                       {"ELSIF" relation "THEN" StatementSequence}
                       ["ELSE" StatementSequence]
                       "END" .
ForStatement =         "FOR" identifier ":=" expression ".." expression "DO"
                       StatementSequence "END" .
constructor =          "[" expression {"," expression} "]".
parameter =            expression | constructor.
UnitAssignment =       identifier selector "(" parameter {"," parameter} ")".
position =             expression {"," expression} | "[" position {";" position} "]" .
PosAssignment =        identifier selector "::" position.
```

```
statement =                 [assignment | UnitAssignment | PosAssignment | IfStatement |
ForStatement].
StatementSequence =         statement {";" statement}.

InType =                    {"[" expression "]"} "BIT".
InOutType =                 {"[" expression "]"} ("TS" | "OC").
OutType =                   {"[" expression "]"} ("BIT" | "TS" | "OC").
ImportList  =               "IMPORT" identifier {"," identifier} ";" .

module  =                   "MODULE" identifier ";" [ImportList]
                            {TypeDeclaration ";"}
                            ["CONST" {ConstDeclaration}]
                            ["IN" {IdList ":" InType ";"}]
                            ["INOUT" {IdList ":" InOutType ";"}]
                            ["OUT" {IdList ":" OutType ";"}]
                            ["VAR" {VarDeclaration}]
                            ["CLOCK" expression ";"]
                            ["BEGIN" StatementSequence]
                            "END" identifier "." .

TypeDeclaration =           "TYPE" identifier ["*"] ["(" IdList ")"] ";"
                            ["CONST" {ConstDeclaration}]
                            ["IN" {IdList ":" InType ";"}]
                            ["INOUT" {IdList ":" InOutType ";"}]
                            ["OUT" {IdList ":" OutType ";"}]
                            ["VAR" {VarDeclaration}]
                            ["BEGIN" [StatementSequence]
                            "END" identifier.
```