A Laboratory for a Digital Design Course Using FPGAs

Stephan Gehring Stefan Ludwig Niklaus Wirth

Institute for Computer Systems, Federal Institute of Technology (ETH)
CH–8092 Zurich, Switzerland
{gehring ludwig wirth}@inf.ethz.ch

Abstract. *In our digital design laboratory we have replaced the traditional wired circuit modules by workstations equipped with an extension board containing a single FPGA. This hardware is supplemented with a set of software tools consisting of a compiler for the circuit specification language Lola, a graphical layout editor for design entry, and a checker to verify conformity of a layout with its specification in Lola. The new laboratory has been used with considerable success in digital design courses for computer science students. Not only is this solution much cheaper than collections of modules to be wired, but it also allows for more substantial and challenging exercises.*

1 Introduction

In order to demonstrate that what had been learnt in the classroom can actually be materialized into useful, correctly operating circuits, digital circuit design courses are accompanied by exercises in the laboratory. There, students select building elements from an available collection and assemble circuits by plugging them together, by wire-wrapping, or by soldering. We have replaced this setup by workstations used in programming courses [1] and equipping them with an FPGA on a simple extension board. Not only is this replacement substantially less expensive, but it allows for the implementation of considerably more realistic and challenging designs. This is due to the large number of available building elements in the form of FPGA cells. Instead of plugging units together, cells are configured and connected using a graphical circuit editor. Indeed we consider this laboratory as *the* application of SRAM-based FPGAs, where their inherent flexibility is not merely an advantage, but a simple necessity. After all, a design is not only changed for correction or improvement, but also discarded upon successful completion, whereafter the FPGA is reused for a next exercise. Our experience also shows that learning effect and motivation surpass our expectations, and that simulation by software can no longer be justified as a substitute for actual circuit implementation. Furthermore, the concurrent design of test programs on the host computer helps to bridge the perceived gap between hardware and software, and is a strongly motivating factor, in particular for Computer Science students.

Whereas the construction of the FPGA-board was a rather trivial matter, most of the project's efforts were spent on the design of adequate software tools. They comprise not only a graphical layout editor, but also a small circuit specification language called *Lola* and its compiler (Sect. 2). A typical exercise starts with the formulation of the informally described circuit in terms of this (textual) notation. The second step consists of mapping it onto the FPGA, i.e. of finding a layout and entering it with the aid of the *layout editor* (Sect. 3). Before testing the circuit with test programs, a second tool, the *Checker* is applied to verify the consistency of the layout with the circuit's specification in terms of Lola (Sect. 4).

We stress the fact that these tools have not only proved most useful in digital design courses, but also adequate and effective in practice.

2 The Circuit Specification Language Lola

In the design of Lola we have made a deliberate effort to let the basic notions of digital circuits be expressed as concisely and as regularly as possible, making use of constructs of programming languages, while omitting unnecessary and redundant features and facilities. The similarity of its appearance (syntax) with that of structured programming languages is intentional and facilitates the learning process. However, the reader is reminded that "programs" describe static circuits rather than algorithmic processes. Although the entire language is defined in a report of some six pages only, we here choose to convey its "flavor" by showing a few examples rather than by presenting a comprehensive tutorial.

2.1 Declarations, Expressions, and Assignments

Every variable (signal) is explicitly declared. Its declaration specifies a type (binary, tri-state, open-collector) and possibly a structure (array dimension). Variables occur in expressions defining new signal values. The available operators are those of Boolean algebra: not (~), and (*), or (+), and xor (−). Expressions are assigned to variables, thereby defining their value depending on other variables. The frequently encountered multiplexer operation is defined as

```
MUX(s: x, y) = ~s * x + s * y
```

The following basic operators allow the specification of storage elements and registers, and thereby of (synchronous) sequential circuits.

SR(s', r')	set-reset flipflop
LATCH(g, d)	transparent latch
REG(en, d)	D-type register with enable and implied clock

2.2 Type Declarations

If a certain subcircuit appears repeatedly, it can be defined as an explicit circuit type (pattern), whereafter it can be instantiated by a simple statement. Declaration and instantiation resemble the procedure declaration and call in programming languages. Inputs appear in an explicit list of parameters. Outputs do not. Instead, they are treated like local variables, with the difference, however, that they can also be referenced in the context of the instantiation, namely by their name qualified by the instance's identification.

Of particular value is the easy *scalability* of declared types. This is achieved by supplying a declaration with numeric parameters, typically used to indicate array dimensions. This kind of parametrization embodies the most essential advantage of textual specifications over circuit schematics.

2.3 Examples

The first example is a binary adder consisting of N identical units of type *ASElement*. Input *cin* denotes the input carry, and *s* controls whether *z* is the sum of *x* and *y* or their difference (Fig. 1).

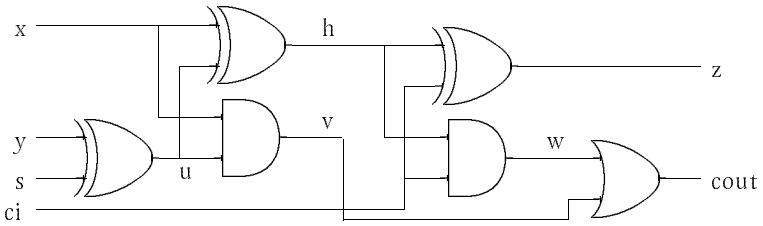

Fig. 1. Add-Subtract Element

```
TYPE ASElement;
    IN x, y, ci, s: BIT;
    OUT z, co: BIT;
    VAR u, h: BIT;
BEGIN u := y − s; h := x − u; z := h − ci; co := (x ∗ u) + (h ∗ ci)
END ASElement;

TYPE Adder(N);
    IN cin, sub: BIT;
        x, y: [N] BIT;
    OUT cout: BIT;
        z: [N] BIT;
    VAR AS: [N] ASElement;
BEGIN AS.0(x.0, y.0, sub, sub);
    FOR i := 1 .. N−1 DO AS.i(x.i, y.i, AS[i−1].co, sub); z.i := AS.i.z END ;
    cout := AS[N−1].co
END Adder
```

The second example shows a multiplier with N-bit inputs *x* and *y* and a 2N-bit output *z*. The circuit consists of a matrix of identical adder elements (Fig. 2). The first parameter is the product of multiplicand and multiplier.

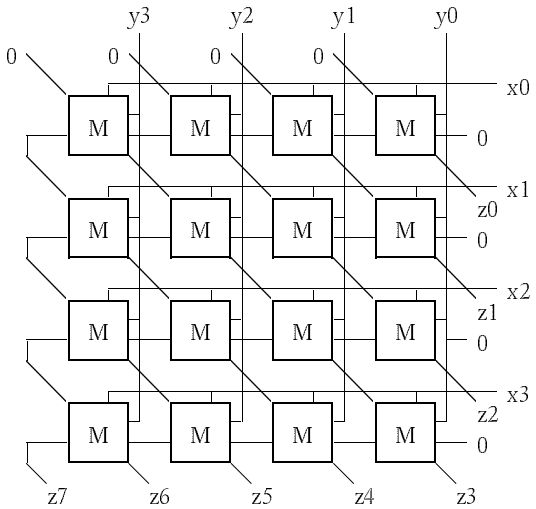

Fig. 2. Multiplier

```
TYPE AddElement;
    IN x, y, ci: BIT;
    OUT z, co: BIT;
BEGIN z := (x–y) – ci; co := (x * y) + ((x–y) * ci)
END AddElement;

TYPE Multiplier(N);
    IN x, y: [N] BIT;
    OUT z: [2*N] BIT;
    VAR M: [N][N] AddElement;
BEGIN
    FOR j := 0 .. N–1 DO M.0.j (x.0 * y.j, '0, '0) END ;
    FOR i := 1 .. N–1 DO
        M.i.0 (x.i * y.0, M[i–1].1.z, '0);
        FOR j := 1 .. N–2 DO  M.i.j (x.i * y.j, M[i–1][j+1].z, M[i][j–1].co) END ;
        M[i][N–1] (x.i * y[N–1], M[i–1][N–1].co, M[i][N–2].co)
    END ;
    FOR i := 0 .. N–2 DO z.i := M.i.0.z;  z[i+N] := M[N–1][i+1].z END ;
    z[N–1] := M[N–1].0.z;  z[2*N–1] := M[N–1][N–1].co
END Multiplier
```

Our last example is a binary up/down counter with the three control inputs *en* (enable, carry input), *clr*' (clear), and *up* (indicating the counting direction).

```
TYPE UpDownCounter(N);   (*with load, enable and clear*)
    IN ld', en, clr', up: BIT; x: [N] BIT;
    OUT Q: [N] BIT;
    VAR cu, cd: [N] BIT;
BEGIN
    Q.0 := REG(MUX(ld': x.0, Q.0 * clr' – en)); cu.0 := Q.0 * en; cd.0 := ~Q.0 * en;
    FOR i := 1 .. N–1 DO
        Q.i := REG((MUX(ld': x.i, Q.i – MUX(up: cd[i–1], cu[i–1]))) * clr');
        cu.i := Q.0 * cu[i–1]; cd.i := ~Q.i * cd[i–1]
    END
END UpDownCounter
```

2.4 The Compiler

Unlike a compiler for a programming language, which generates executable code, the Lola compiler generates a data structure representing the circuit that is most appropriate for further processing by various design tools, ideally by an automatic layout generator. Other tools are timing analyzers, fanout checkers, and simulators. In our case, the most important tool is the Checker, which verifies a given layout rather than generating one. The data structure generated by the compiler consists of a binary tree for each variable occurring in the design. Hence the compiler flattens the structured description. It also applies obvious simplification rules. They take effect, for example, at the edges of the matrix of the second example above, where some of the input parameters are zeroes.

3 The Layout Editor

A graphical editor is used to enter and modify circuit specifications implemented on an FPGA. It presents the FPGA at a low level, as close to the real hardware as possible. We first present the used FPGA architecture and then give a description of

the editor's mode of operation and its implementation.

3.1 The Hardware

In our laboratory, an extension board containing an FPGA of Atmel (formerly Concurrent Logic Inc.) is used [2]. The AT6002 chip in an 84-pin package consists of a matrix of 32 by 32 identical cells. A cell implements *two functions of up to three inputs* (A, B, and L). These functions can be combinational and sequential (i.e. involving a register). Two outputs (A and B) of a cell are connected to the inputs of its four neighbors (north, south, east, and west). In addition to the *neighbor connections*, there is a bussing network connecting bus inputs and outputs of eight cells in a row or column. These so-called *local busses* are used to transport signals over longer distances between cells. They can be connected to other local busses or to additional *express busses* via *repeaters* at 8-cell boundaries. Surrounding the array of cells are 16 *programmable IO pads* on each side. These connect to the bus of the host workstation and to components on the extension board, such as an SRAM and an RS–232 line driver.

3.2 Design Representation and Modification

The editor presents the gate array in a viewer as an excerpt of the 1024 cells (Fig. 3). Every eight cells, a repeater column or row is displayed, and surrounding the array, the programmable pads are shown. Each component's contents reflect the implemented function as closely as possible - e.g. an Exclusive-Or in a cell with a constant one input is displayed as a Not-gate. To show the signal flow, connections between cells and to and from local busses, and connections with repeaters are displayed as arrows. By giving neighboring connections a different color (yellow) than local (green) and express busses (red), a visual feedback on the speed of a specific connection is suggested. Inside a cell, the same picture is displayed regardless of the source and destination direction of signals. For instance, even if signals enter a cell from below and flow to the top, the picture inside the cell suggests a flow from top to bottom. The reason for this will be explained in Sect. 3.3.2. To give signals a meaningful name – and to enable a link to a Lola description of a circuit (see Sect. 4) – textual labels can be placed at cell and pad outputs.

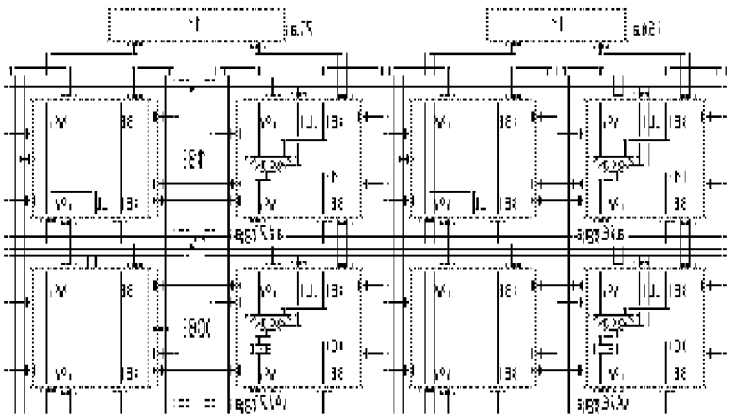

Fig. 3. Editor View with Cells, Pads, Repeaters, and Labels

The mouse is used as the primary input device to change a design. Cells, pads, and repeaters can be edited using popup menus (Figs. 4, 5). The top row of the menu in Fig. 4 shows the six different *routing modes* possible in a cell, and the four items on the left of the bottom row show the *state* of a cell [2]. The two multiplexers on the right are an often used combination of routing mode Mux and states Xor or Xor with register. Similarly, all possible configurations for repeaters (Fig. 5) and pads (not shown) are presented through a menu. The current configuration of the edited resource is highlighted in the menu with a frame. Connections between cells must be entered manually as no automatic router is provided. Thus, students learn about the problems of placement and routing in FPGAs. Fast replication of data path elements is available by selecting and copying bit slices of the layout. Cells can also be moved or copied across viewer boundaries in which a different design or a different excerpt of the same design is shown.

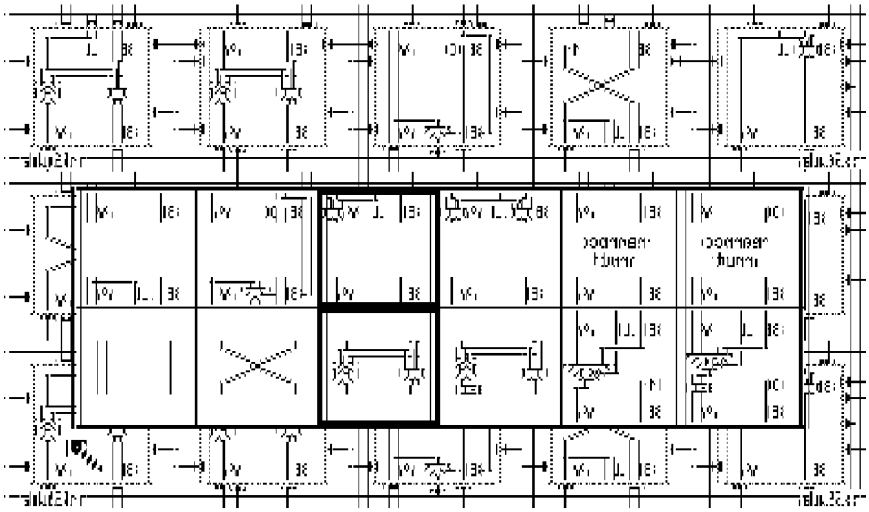

Fig. 4. Cell Menu

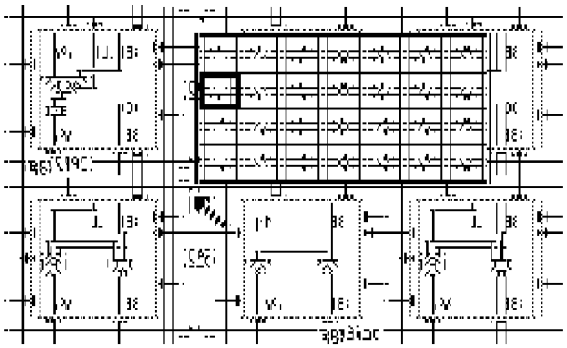

Fig. 5. Repeater Menu

3.3 Implementation

The editor consists of five modules comprising roughly 65KB of object code. The following sections discuss some of the finer points of the implementation.

3.3.1 Data Structures

We use a straight-forward data structure to represent the various resources on the FPGA. A two-dimensional array of cell records represents the matrix of cells. This allows for fast iteration over the data structure when displaying it. Similarly, the repeaters and pads are represented as arrays of records. The labels, however, are a linked list of records containing the position and caption of a label. Designs are saved to disk using a portable data format. A simple run-length encoding of empty cells, pads, and repeaters compresses typical files to 23% of their original size. Even large designs take up only 8KB, whereas smaller designs remain well under 1KB.

3.3.2 Drawing Operations

For drawing the contents of a cell, we use a special font containing only the patterns of signals flowing from top to bottom. Thereby, we get fast drawing of a design without having to distinguish between the 384 possible signal flow directions, but at the cost of a fixed aspect ratio and non-optimal print output. Making the distinction and drawing a cell's contents with multiple lines and dots slows down the performance by 50% and increases the program size by 100%. Repeaters are drawn using a font as well, but here, a special pattern exists for each possible signal flow. Despite the disadvantages when using a font, the chosen solution works well in practise. A special display option can be set where only used cells and busses are drawn. Not only does this improve display speed, but it also avoids a cluttered view.

3.3.3 Editing Operations and Undo

The problem of displaying three different menus has an elegant solution using a generic procedure. This procedure takes two procedure variables as parameters, one for displaying the contents of each menu item, and one for updating the data structure according to the chosen item. Thereby, the code for configuring cells, pads, and repeaters remains the same, only the procedure variables and the number of rows and columns in the menu change.

Each editing operation can be undone. This is accomplished by backing up the data structure before executing the operation. Then, a simple swap between the backup and the primary data structure implements the undo (and redo) operation.

3.4 Command Module and Queries

Operations that are not frequently used are provided through a command module [3]. Clock and reset lines are set with commands. Labels, cells according to their coordinates, and whole arrays according to a prefix, can be located in a design. Statistics on the design are also provided, with which different implementations of the same specification can be compared against each other (according to bus utilization and the number of cells used for routing, logic, and registers).

3.5 Downloading to the Extension Board

Once a design is finished, it can be downloaded onto the FPGA in a few milli-seconds. Only during this step, simple electrical consistency checks are performed, such as multiple sources writing to a bus unconditionally, and incompletely configured cells.

3.6 Discussion

For the intended purpose the chosen implementation worked out very well. The fast adaption of all users to our system was encouraging and the positive feedback very rewarding. In the future, we will provide configurability of the editor to support various chip sizes and IO configurations. Research-wise, we intend to develop design automation tools that support a seamless integration between the specification of a circuit and the automatically laid out design.

4 The Checker and Analysis Tools

4.1 The Checker

In a digital design laboratory, a typical design cycle might look as in Fig. 6. After initial design entry with the editor, the designer downloads the design onto the FPGA. By configuring the FPGA, the circuit is implemented and can be tested subsequently. If the test fails, the design is corrected, downloaded, and tested anew.

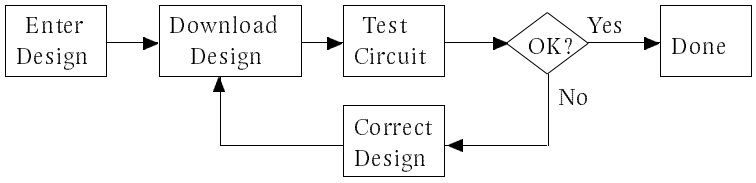

Fig. 6. Design Cycle

While downloading and testing a design is usually a matter of seconds or minutes, correcting a faulty design can be very tedious. Mostly, this comes from the fact that, while it is easy to detect an error, it is hard to find its location in the design. In traditional laboratories with electronic components being plugged together, the designer must verify manually that each component is properly wired. Our software-based approach, by contrast, offers the opportunity to construct a circuit checker program that helps the designer not only to detect, but also to locate implementation errors.

4.1.1 Representing and Checking Designs

A digital circuit is characterized by its inputs, outputs, and a set of Boolean functions combining the inputs. Each circuit output is associated with the result of such a function. The function can be represented as a *binary tree* with nodes consisting of Boolean constants, operators, variables, and units composed of several operators (e.g. multiplexers, registers). Each output forms the root of such a binary tree. A

complete circuit can thus be represented as a set of trees, one for each output. Inner tree nodes represent operators with edges pointing towards the node's inputs, while leaf nodes represent constants and input variables.
Fig. 7 illustrates the equivalence between a Boolean function represented as a set of interconnected gates, a binary tree, and a Boolean formula.

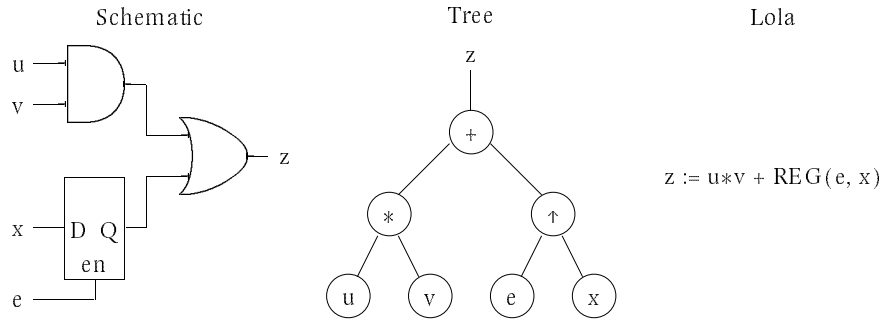

Fig. 7. Circuit Representations

Since the above representations of a circuit are equivalent, both a circuit layout and a Lola program can be transformed into a set of trees. Corresponding trees can then be compared to detect inconsistencies. Under the assumption that the Lola program describes the circuit correctly, i.e. it properly reflects a *circuit specification*, inconsistencies between corresponding pairs of trees are interpreted as errors in the layout, i.e. the *circuit implementation*.

The checker strives to find a *structural equivalence* between the specification and the implementation trees. It starts at the roots of two respective trees and, in parallel, traverses both trees from the roots towards the inputs. At each pair of nodes, the checker verifies that the two nodes match. If they match, the nodes' subtrees are checked for equivalence recursively. The procedure terminates when all nodes have been visited or a mismatch is detected.

Existing verifiers, such as automated theorem-provers [4], attempt to find an equivalence between Boolean equations by transforming them until equivalence (or its opposite) is inferred. This scheme is more flexible than matching for structural equivalence and allows for different levels of abstraction between the specification and the implementation. While such verifiers are well suited to detect inconsistencies, they typically fail in pinpointing the fault in the layout. The information needed for this purpose is either left out or lost during the transformations applied to the Boolean equations. This loss makes it impossible to locate an implementation error automatically and leaves the designer with the labor of locating it in the layout manually.

The checker, by contrast, keeps the information required to locate a part in the layout within each node. With this information available, an implementation error can not only be detected but also located in the faulty layout.

4.1.2 Using the Checker

The first step in the checking process is writing a Lola specification for the circuit. This program is compiled by the Lola compiler which generates a set of trees as its output. The trees can be viewed in a textual format as a set of Boolean equations. The output can be used as a reference in the next step when entering the design with the editor. The checker is then invoked to check the implementation for compatibility with the specification. Inconsistencies between the two are displayed textually and

graphically in the layout. The checker can check complete layouts but may also be used during design entry to check partial layouts (e.g. for checking bit slices of a data path).

4.1.3 Implementation

In order to make the implementation of the checker simple and extensible, the *architecture-dependent extraction* part is decoupled from the *architecture-independent matching* part. The extractor converts the FPGA-dependent representation of circuits used by the editor into an architecture-independent set of trees. The matcher then verifies compatibility with the set of trees generated by the Lola compiler. This separation allows easy adaption to a new FPGA architecture by simply exchanging the extractor component. The extractor follows the signals from the output towards the inputs. Extraction stops at labels and constants found in the layout. When returning from the leaf nodes, the tree is constructed. Already during extraction certain checks are performed, such as detecting unconditional outputs to a tri-state bus or reading from an undefined source. The extractor also recognizes certain combinations of gates and converts them to more abstract operators, such as

$$q := \mathrm{MUX}(en : q, x) \quad \rightarrow \quad q := \mathrm{LATCH}(en, x)$$
$$q := \sim(s' * \sim(r' * q)) \quad \rightarrow \quad q := \mathrm{SR}(s', r')$$

Once the trees are extracted, the matcher checks corresponding pairs of trees for compatibility. The trees generated by the Lola compiler are used as a reference while the trees extracted from the circuit are examined.

Earlier, we mentioned that the checker searches for a structural match between two corresponding trees. Demanding an exact structural match would require the designer to specify the circuit exactly the same way as it is later implemented. As this is too restrictive, the checker allows a number of *transformations* being applied to the trees. Since the goal is still to locate detected errors in the layout automatically, transformations must preserve the information needed for this purpose. The structural matching rules are relaxed and allow the following transformations:

1. *Inverters.* Architectural constraints imposed by FPGAs sometimes require the designer to connect parts of a circuit through successive inverters. For example, if an AND gate is implemented with a NAND gate, an inverter must follow the NAND gate, hence there are two inverters in series. The checker allows an arbitrary number of inverter nodes between any two nodes.
   $x = \sim(\sim x)$
2. *DeMorgan's Laws.* The checker applies the laws of DeMorgan when necessary. For instance, the AT6002 FPGA cell lacks an OR gate. An OR gate is therefore usually implemented as a NAND gate with inverted inputs. This architecture-dependency should, however, not reflect in the specification where the OR operator is used instead.
   $x + y = \sim(\sim x * \sim y) \qquad x * y = \sim(\sim x + \sim y)$
3. *Commutativity.* The representation of a dyadic Boolean operator as a node of a binary tree introduces an inherent order, by which its subtrees are compared (e.g. "compare left specification subtree with left implementation subtree"). For commutative operators (AND, OR, XOR), this order cannot be determined beforehand and the checker potentially matches both possibilities. Since the trees generated by the Lola compiler have a typical height of less than five, there is no apparent performance penalty associated with commutativity.
   $x * y = y * x \qquad x + y = y + x \qquad x - y = y - x$

4. *Associativity*. As with commutativity, associativity is an inherent property of binary trees. The checker supports only simple cases of associativity.
   $y * (x * (u + v)) = (u + v) * (x * y)$
5. *MUX selectors*. For greater flexibility, multiplexers may be implemented with an inverted selector signal and accordingly exchanged input signals.
   $MUX(s: x, y) = MUX(\sim s: y, x)$
6. *OR/AND with MUX*. It is sometimes more convenient to implement OR gates or AND gates using multiplexers. The checker recognizes the MUX representations as equivalent.
   $x + y = MUX(x: y, '1) \qquad x * y = MUX(x: '0, y)$

All of these transformations can be applied to trees without losing information needed to locate errors in the layout after a mismatch.

Combined, the transformations make the checker a flexible and efficient tool for checking layouts. Its speed and its capability to check only parts of a design make it well suited for interactive use during design entry.

Design	AT6002 Cells Used	Lola Variables	Total Checking Time
UART	240	100	< 1 s
8x8 Multiplier	440	230	< 2 s
Microcontroller	770	240	< 4 s

Table 1. Checking Performance (80486, 33MHz)

4.2 The Timing Analyzer

Once a circuit is designed with the editor and its correct layout verified with the checker, the question about the circuit's performance arises. To determine the maximum operating speed of a given synchronous circuit a timing analysis tool is required. We have developed a timing analyzer which is capable of analyzing combinational and sequential circuits efficiently. It can be used interactively from within the editor during design entry but also provides a simple programming interface which can be used by future design automation tools. It provides commands to determine the maximum input delay between a given output and all of its inputs or only a specific input. If a circuit contains parts with fan-outs greater than one ("common subexpressions") their input delays are calculated only once to save computation time.

5 Conclusions

We presented an FPGA system consisting of an extension-board with an Atmel AT6002 FPGA and a set of simple and efficient software tools used to develop circuits for the board. The software consists of a compiler for the Lola language, a small hardware description language for synchronous digital circuits, an easy-to-use graphical editor with which layouts are entered with simple mouse manipulations, and a loader to configure the FPGA with layouts entered with the editor. Additionally, a circuit checker was implemented which performs a consistency check between a circuit specification in the form of a Lola program and its implementation within the editor. Inconsistencies are not only detected but also located within the layout displayed in the editor.

The software part was designed and implemented in Oberon [3] by three people in three months and consists of 13 modules containing about 6500 lines of code. Two weeks were spent developing the extension-board.

We have been using the system successfully in a laboratory for introductory courses in digital design. Due to its simplicity, the students learned to use the system quickly and were able to solve the given exercises. The exercises range from simple binary counters to a UART. At the Institute, we use the same system for experiments with programmable hardware.

All in all, we can only recommend using FPGAs in education. Their flexibility and quick reprogrammability allow interesting and diverse problem statements. By using real hardware instead of a simulator, the students also have to cope with the "real" problems of digital design such as good placement, economical routing, timing, and synchronization between components. Last, but not least, the chosen solution is an order of magnitude more cost effective than conventional laboratories using discrete MSI components and physical wiring.

Acknowledgements

We wish to thank I. Noack for implementing and testing the extension board.

References

1. B. Heeb, I. Noack, *Hardware Description of the Workstation Ceres–3*,Technical Report 168, Institute for Computer Systems, ETH Zurich, Switzerland, October 1991
2. Atmel Corporation, San Jose, CA. *Field-Programmable Gate Arrays, AT6000 Series.* 1993
3. M. Reiser. *The Oberon System – User's Guide and Programmer's Manual.* Addison-Wesley, Reading, MA. 1991.
4. R.S. Boyer, J. Strother Moore, *Proof-Checking, Therorem-Proving, and Program Verification*, Contemporary Mathematics, Vol. 29, American Math. Society, 1984, 119-132