Report

An Extension-Board with an FPGA for Experimental Circuit Design

Author(s):
Wirth, Niklaus; Ludwig, Stefan

Publication Date:
1993-07

Permanent Link:
https://doi.org/10.3929/ethz-a-000900466

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
An Extension-Board with an FPGA for Experimental Circuit Design

CL - An Editor for the CLi6000 Field Programmable Gate Array and its Implementation

CL-Editor User Manual

July 1993
Author's address:

Institut für Computersysteme, ETH Zentrum, CH-8092 Zurich, Switzerland
{wirth,ludwig@inf.ethz.ch}

© 1993 Departement Informatik, ETH Zürich
An Extension-Board with an FPGA for Experimental Circuit Design

N. Wirth

Abstract

We describe the design of an extension board for the workstation Ceres-3 containing a Concurrent Logic CLI6002 FPGA. The board is used in a laboratory for an introductory digital design course equipped with Ceres-3 workstations. An analogous board was designed for PCs.

Introduction

Introductory design courses in digital circuit design typically use laboratories based on modules containing basic TTL-components which can be plugged or wired together to represent the desired circuits. Their drawback is a fairly large inventory of modules, if the subject of designing is to be at the center of attention, i.e. if non-trivial exercises are to be performed. The advent of field-programmable gate arrays opens a new opportunity, at the same time making reasonably large designs possible and also drastically reducing the amount of necessary hardware and thereby the cost of the entire laboratory. For each workstation, a single extension board containing an FPGA suffices. Such a board is subsequently described; it uses a Concurrent Logic FPGA CLI6002 with 1024 programmable cells. The FPGA is configured with the aid of a graphical editor described in the companion paper.

The extension board contains the following components:

1. The CLI6002 field-programmable gate array.
2. Interface circuits between the FPGA and the Ceres-3 system bus.
3. "Peripherals" to the FPGA, namely
   1. A 32K x 8 static memory (SRAM).
   2. A driver for an RS-232C line.
   3. A driver for an RS-485 network.
   4. A clock generator (typically 3.6864 MHz for line interface circuits).

The presence of "peripherals" is useful for the design of more complex exercises, such as serial line interfaces (UARTs), fifo-memories, and even simple microprocessors, just to name a few.

Designer's view of the FPGA

The CLI6002 FPGA consists of a 32x32 matrix of cells and I/O cells connecting the cells at the periphery to pads (pins). On each of the four sides, there exist 16 I/O cells, of which we make 15 available. Their external connections are specified in Fig. 1.

I/O cells on the left (West) connect to the Ceres system bus. The data channel is 8 bits wide, and 2 address lines are made available. Evidently, the FPGA is to be viewed from the computer like a peripheral device with 4 possible address values.

<table>
<thead>
<tr>
<th>D0 – D7</th>
<th>A2, A3</th>
<th>RD’</th>
<th>WR’</th>
<th>SEL’</th>
<th>INT’</th>
<th>CW’</th>
</tr>
</thead>
<tbody>
<tr>
<td>data lines</td>
<td>address lines</td>
<td>read strobe</td>
<td>write strobe</td>
<td>chip select</td>
<td>interrupt</td>
<td>continuous wait</td>
</tr>
<tr>
<td>input/output</td>
<td>input</td>
<td>input</td>
<td>input</td>
<td>input</td>
<td>output</td>
<td>output used to delay processor access cycle</td>
</tr>
</tbody>
</table>
I/O cells on the top (North) and right (East) connect the external SRAM.

<table>
<thead>
<tr>
<th>a14</th>
<th>a13</th>
<th>a12</th>
<th>a11</th>
<th>a10</th>
<th>a9</th>
<th>a8</th>
<th>a7</th>
<th>a6</th>
<th>a5</th>
<th>a4</th>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

- **d0 - d7**: data lines
- **a0 - a14**: address lines
- **OE'**: RAM output enable
- **WE'**: RAM write enable
- **CS'**: RAM chip select

The remaining named I/O cells are assigned as follows:

- **Rx0**: RS-232C Receiver data
- **Tx0**: RS-232C Transmitter data
- **Rx1**: RS-485 Receiver data
- **Tx1**: RS-485 Transmitter data
- **RxE'**: RS-485 Receiver enable
- **TxE**: RS-485 Transmitter enable

Note that pins labelled as "input" must not be configured with pad tri-state enabled.

**Implementation**

The entire board circuit is shown in detail in Fig. 2. The bus–control signals A2, A3, INT0', IOSei0', (decoded signal addressing the extension board), and CWAIT' are directly connected to the FPGA. The bus clock (25 MHz) is fed directly to the FPGA's global clock input. The bus data lines D0 – D7 are connected to the FPGA via a transceiver (74BCT245). The FPGA control signals CS', RESET', CCLK, CON', and M0 – M2 are derived from bus control signals by circuits implemented through a programmable device (Altera EP610).
On the right side, the "peripherals" are an SRAM (HM62256), a differential RS–485 bus driver/receiver (DS3695), and an RS–232C line driver/receiver (DS1228).

The decoding circuitry is implemented using a PLD; its function is shown in detail in Fig. 3. The flipflops hold state and loading mode of the FPGA. To start the downloading process, CON' is pulled low and mode M is 6, signalling sequential loading clocked by CCLK; a clock pulse is generated upon every write instruction. After loading has been started, the FPGA drives CON' low, and when loading is terminated, the FPGA releases the signal. Its value can be read on the D0 data bus line, allowing to determine...
whether or not downloading has been completed successfully.

The following program statements, written in Oberon, are used in communicating with the FPGA.

```plaintext
CONST A = 0F8000000H;  base address of FPGA
SYSTEM.PUT(A + a, x)  output x to FPGA, a = 0, 4, 8, 12
SYSTEM.GET(A + a, x)  input x from FPGA, a = 0, 4, 8, 12
SYSTEM.PUT(A + 80H, 0)  reset (global reset signal)
SYSTEM.PUT(A + 0COH, 7)  start downloading process
SYSTEM.PUT(A + 40H, x)  download x
SYSTEM.BIT(A + 0COH, 0)  downloading completed
SYSTEM.PUT(A + 0COH, 4)  set normal mode
```
CL – An Editor for the CLi6000 Field Programmable Gate Array and its Implementation

Stefan H.-M. Ludwig

Abstract

We describe the development of an interactive editor for small to intermediate circuit design using the Field Programmable Gate Array CLi6000 of Concurrent Logic, Inc.. A short introduction to FPGAs is given and the requirements for a low-level hardware editing tool are stated. The implementation of such an editor is presented in detail and it is contrasted to an editor for the CAL1 FPGA architecture. Upon this comparison some conclusions are drawn.

Keywords: Field Programmable Gate Arrays, Digital Design, Editor, Oberon

Introduction

With the advent of Field Programmable Gate Arrays (FPGAs), experimental hardware design has become a matter of moving the mouse pointer around a computer screen and clicking buttons. The flexibility of static RAM-based FPGAs has made the introduction of hardware editors feasible. The designer can edit, view, verify, and download a circuit to real hardware within minutes. This work aims at the development of an editor for hardware design used in an introductory course in digital design for computer science students, as well as for the design of small to intermediate circuits in our Institute. The paper first presents FPGAs in general and then proceeds with a detailed implementation of an editor for low-level hardware design. The motivation for this project as well as a description of the used hardware is given in the companion paper.

FPGAs – The CLi6000 Architecture

The first commercial static RAM based Field Programmable Gate Array was introduced in 1985 by Xilinx [Xil90]. Since then, many different architectures have been proposed, among them Concurrent Logic's CLi6000 architecture [CLI92]. We will present the concepts behind FPGAs by means of this particular architecture.

An FPGA is a programmable logic device. It consists of a matrix of programmable cells and a programmable interconnect between these cells. The configuration of the cells and interconnect is stored in an on-chip SRAM. (There are other types of on-chip store which will not be discussed here.) On the perimeter of the matrix are input and output cells which connect external signals with cells in the chip. These IO-cells are programmable as well. Figure 1 shows the general layout of an FPGA.
If we view the array from the side (e.g. from the south), we can imagine seeing two layers – the cell layer on top, and the configuration layer on the bottom (see figure 2). The bottom layer controls the functionality of the top layer by means of bits stored in RAM-cells. We regard the values of the RAM-cells as constant after downloading.

General Layout and Connection Network

The CLI6002 FPGA of Concurrent Logic, Inc. [CLI92] consists of an array of 32 by 32 identical cells. A cell implements two functions of up to three inputs (A, B, and L). These functions can either be combinational or sequential (i.e. involving a register). Both outputs (A and B) of a cell are connected to inputs of its four neighbors (north, south, east, and west). In addition to the neighbor connections, there is a bussing network connecting inputs or 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 express busses via repeaters at eight cell boundaries. Figure 3 shows the connection network.
Cell

A detailed view of a CLI6000 cell is shown in figure 4 above. The multiplexers (rectangles in the figure)
are controlled by SRAM cells, as are the pass gates to the local bus (NS1/2, EW1/2). We will discuss the
details from left to right. The input to the cell is selected by the input multiplexers (mux for short). There
are two inputs, named A and B. The A-mux selects from which cell (i.e. from which direction N, S, E, or
W) the input comes from, or if a logical '1' is feed into the cell. The same can be selected for the B-input.
The two 2-input mux in the middle select between reading from the L-bus or supplying a constant (either
"1" or "0").

Together with the 3-input mux controlling the tri-state on the right, these 5 multiplexers constitute the
so-called routing mode of a cell. Figure 5 presents the different routing modes in a more schematic way.

![Routing modes](image)

**Fig. 5: Routing modes**

The different modes have the following semantics:

- **Write**: write to the local bus
- **Tri-State**: write to the local bus under control of the B-input
- **Read**: read from the local bus "anding" it with the A-input
- **Mux**: pass either the A- or B-input to the inner cell
- **Turn-B**: the cell performs a corner turn on two L-busses
  (e.g. connecting north with west L-bus)
- **Turn-0**: same as above, but feed in a "0" instead of the B-input

The two 4-input multiplexers in the inner cell select the state of the cell — i.e. what function the cell is
implementing. Note that both multiplexers are controlled in tandem. States 0 and 1 are used for routing
the two inputs (A and B) to the two outputs A and B directly (state 0), or by crossing over (state 1). State
2 implements an exclusive Or for the A-output and a Nand for the B-output. In state 3, the exclusive Or
feeds into a register for the A-output, and the And-function is implemented for the B-output (see figure
6).

![Function states](image)

**Fig. 6: Function states**

A cell is configured, when both mode and state are set. For instance, by combining the Mux mode with
State 2, the cell implements a multiplexer circuit on output A, and a logical "1" on output B (figure 7).
Repeaters

At eight cell boundaries, there are so-called *repeaters* for connecting busses together. Almost all combinations of connecting a local or express bus with another local or express bus are possible. A repeater can either be off or have one of 28 states shown in figure 8. In general, busses and therefore repeaters are uni-directional; special cases where a bus can be bi-directional are not discussed here.

*Fig. 8: Possible states of an east-west repeater (local bus below)*

Input / Output

*Pads* are located at the chip boundaries for bringing signals into or off the chip. Pads receive a signal from an *output cell* through a tri-state gate which can be either on, off, or controlled by a local bus. An *input cell* receives the signal from a pad and can feed it into the cell array. Again, the mux controlling the tri-state is configurable via bits in the RAM.

Note: Input and output cells are just like normal cells, except that their A- or B-input or A- or B-output signal on one side is connected to a pad instead of to another cell.
Clock / Reset

The sources of clock and reset signals for the 32 registers in a column can be selected individually for each column. The source of a clock signal can be specified by a mux near the top-most cell, and the source of a reset signal by a mux near the bottom-most cell in each column. The source of these signals can be the A-output of the cell in the top (bottom) row in the given column, a global clock or reset signal, an express bus in the top (bottom) row, or the constant "1".

For further details regarding the architecture, the reader is referred to the data sheet.

Editor – Overview

Having discussed the general structure of the CLi-FPGA, we will continue with our requirements for a tool which allows the construction of hardware designs on a computer screen by means of interactive editing. This editor was intended to be used in a second year introductory course in digital design for computer science students, as well as for experimental hardware design on a small to intermediate scale. We came up with the following requirements:

1. The physical array of cells should appear on the screen in a one-to-one correspondence, in order to give the user flexibility when designing a circuit.

2. Changes made in the design should appear immediately on the display, such that the system's responsiveness is never felt to be slow.

3. Local changes in the design should cause local updates on the screen to avoid flickering.

4. A cell's appearance should reflect the implemented function as closely as possible to hide unnecessary details.

5. Unused resources should not be displayed to avoid a cluttered view.

6. Multiple views of the same design in various viewers should be possible.

7. To improve the speed of editing, a collection of cells implementing a certain operation should be copyable.

8. It should be possible to give certain signals in a design a descriptive name such that the design can be viewed from a higher level of abstraction.

More features may seem desirable, for instance cell macros, but since we did not intend to develop large designs, the offered possibilities were felt sufficient. The editor has been implemented for the Oberon System [Rei91] and has now been used by students and researchers for some time without requiring major changes.

After a short review of the editor's structure, the following sections present the general mode of operation of the CL-Editor. Based on this knowledge, we will present the implementation of the requirements above.
Module Structure

The CL-Editor package consists of five modules. The import relation is shown in figure 10 where modules on top import modules below. CLGAs contains the definition of the necessary data structures representing a CLI6002 FPGA with corresponding procedures for loading and storing a design in a portable file format. CLFrames and CLFramesD conceptually belong together, as they make up the display and editing component of the editor. Due to the size of this component, it was split into two modules. (CLFrames will stand for both modules from now on.) CLLoader is used to download a design to the actual FPGA hardware and for communicating with the chip. It also performs some rudimentary checks on the design to avoid damaging the chip (→ Consistency Checks). CL is the top-level module, containing commands for opening, storing, and loading designs, as well as commands for interfacing to the hardware.

General Mode of Operation

When opening an empty design, the physical array is displayed in a viewer as an array of 32 by 32 cells (figure 11). Due to the size of a single cell, not all 1024 cells can be viewed simultaneously, but the view onto the array can be shifted with the mouse.
In the above figure, we see the upper left corner of the chip. Each cell has two outputs in each direction, where the A-output is represented by a fat (\(\text{fat}\)) and the B-output by a thin terminal (\(\text{thin}\)). When cells have no routing and state associated with them, they are displayed as blank boxes (requirement 1). When the designer wants to change a cell, a simple click on the left mouse button while on a cell suffices to bring up a menu presenting all possible routing modes and states a cell can have (figure 12). Menus are used to configure pads and repeaters (figures 13 and 14). Pads can be seen at the top and on the left side. Labels appearing below or to the right of a pad give a descriptive name to the signal the pad represents (requirement 8). Labels can also be assigned to a cell's outputs.
Menus highlight the selected state of the edited cell, pad, or repeater with a thick frame (see cell or pad menu). If the user releases the mouse outside the menu area, no changes are made. For a closer look at menus, refer to section *Menus* below.

Most operations can be performed with the mouse. Figure 15 presents the sensitive areas of a cell where mouse clicks perform an action. Similar areas exist at pads and repeaters.

**Fig. 14: Repeater menu**

**Fig. 15: Sensitive areas**

Shifting, Selecting, Copying

The view can be shifted by simply pressing the middle mouse button, dragging the mouse to the desired location, and releasing the button again. The location under the mouse before dragging will appear at the release location.

Selecting stretches of cells is simply done with the right mouse button. A rectangle is spanned when dragging the mouse and all cells in the rectangle get selected.
A selection can be moved or copied to a different location by dragging with the middle button held down and interclicking the left (move) or right (copy) button at the desired location (requirement 7).

A detailed user guide to the editor is included in the Appendix.

Editor – Implementation

Representation of a Design

The necessary data structures representing the 1024 cells of a CLI6002 chip are presented below. GADesc contains all the information needed to configure the chip correctly without respect to displaying it. Based on these data, the CLoader can initialize the chip with the required configuration bits. The constant and type definitions are straightforward; more information can be found in [CLI92].

```
CONST
  Sector = 8; Dim = 4 * Sector;  (* chip consists of 4x4 sectors with 8x8 cells each *)
  None = -1; North = 0; South = 1; West = 2; East = 3;   (* cell input directions *)
  Write = 0; TS = 1; Read = 2; Mux = 3; TurnB = 4; TurnO = 5;  (* cell routing modes *)
  State0 = 0; State1 = 1; State2 = 2; State3 = 3;  (* cell states *)
  EEWE = 1; LEWE = 2; LLWE = 4; ELWE = 8;
  EEEW = 16; LEEW = 32; LLEW = 64; ELEW = 128;  (* basic repeater states *)
  ClkOne = 1; ClkAout = 2; ClkGlobal = 4; ClkExpress = 8;  (* clock/reset selectors *)
  TSOff = 0; TSSerial = 1; TSParallel = 2; TSON = 4;  (* I/O pad output selectors *)
  TTL = 1; Pullup = 2; TSOopen = 4; TSFast = 8;  (* I/O pad flags *)

TYPE
  Cell = RECORD
    routing, state: SHORTINT;  (* function *)
    a, b, wel, nsL: SHORTINT  (* input directions/cornerturn output direction *)
  END;

  VRepeater = RECORD w, e: INTEGER END;  (* vertical repeaters, west and east *)
  HRepeater = RECORD n, s: INTEGER END;  (* horizontal repeaters, north and south *)

  Pad = RECORD
    selector: SHORTINT;  (* TSoff.TSON *)
    flags: SHORTINT  (* TTL.TSFast *)
  END;

  Label = POINTER TO LabelDesc;
  LabelDesc = RECORD
    next: Label;
    u, v: INTEGER;  (* cell coordinates *)
    sig: SHORTINT;  (* labeled output signal (AOu~, BOut) *)
    name: ARRAY 12 OF CHAR
  END;

  GA = POINTER TO GADesc;
  GADesc = RECORD  (* data structure representing a CLI6002 FPGA *)
    c: ARRAY Dim, Dim OF Cell;  (* cells *)
    vr: ARRAY Dim, (Dim DIV Sector) – 1 OF VRepeater;  (* vertical repeaters *)
    hr: ARRAY (Dim DIV Sector) – 1, Dim OF HRepeater;  (* horizontal repeaters *)
    p: ARRAY 4, Dim DIV 2 OF Pad;  (* I/O pads (N/S/W/E) *)
```
Based on the above data structures the editor has to present a view of the chip in an intuitive way. FrameDesc contains a pointer to the design to be displayed in field a, as well as geometrical information of which part of the design is displayed (x, y, etc.). Furthermore, to support the selection of cells, pads, and repeaters, there are fields representing the most recent selection including its type (sel, time). The field cache and its corresponding data type BusCache are needed for displaying bus lines correctly. This problem will be explained in more detail below. The field print is used to requested the printing of the frame (→ Printing).

Frame = POINTER TO FrameDesc;
FrameDesc = RECORD (Display.FrameDesc)
  a: CLGAs.GA; (* the design, this frame displays *)
  cache: BusCache; (* used to draw bus lines *)
  selU, selV, selL, selH, selTyp: INTEGER; (* array coordinates & type of selection *)
  time: LONGINT; (* of selection *)
  x, y, xa, ya, X1, Y1: INTEGER; (* model coordinates x, y, xa, ya, upper right corner x1, y1 *)
  mode: SET; (* of display *)
  selected, print: BOOLEAN (* selection exists, print frame *)
END;

BusCache = POINTER TO RECORD (* is a local/express bus used in the design? *)
localNS, expNS: ARRAY Dim DIV Sector, Dim, 2 OF BOOLEAN;
localWE, expWE: ARRAY Dim, Dim DIV Sector, 2 OF BOOLEAN
END;

Displaying a Design

Upon request the command module CL opens a design reading from a disk file, and initializes a frame to display the design with a call to Open. Then, the frame can be installed inside a menu viewer. The Oberon System sends a message to the frame requesting it to display itself (MenuViewers:ModifyMsg). This message gets handled by the frames message handler (procedure Handle in section Editing a Design), which in turn calls the editor’s Restore procedure. Restore will redraw the whole design. By relying on Oberon viewers we get multiple views onto the same design for free (requirement 6). For details on how the Oberon System manages viewers, the reader is referred to [WG92].

PROCEDURE Open(F: Frame; a: CLGAs.GA); (* initialize F for displaying a *)
PROCEDURE Restore(F: Frame); (* redraw the whole design *)
PROCEDURE Translate(F: Frame; x, y: INTEGER;
  VAR repU, repV, cellU, cellV, secU, secV: INTEGER; VAR repCol, repRow: BOOLEAN);
  (* translate screen coordinates into model coordinates *)
PROCEDURE Cell(F: Frame; x, y, u, v: INTEGER); (* draw the u, v-th cell at x, y *)
PROCEDURE NPad(F: Frame; x, y, k: INTEGER); (* draw the k-th north pad at x, y *)
PROCEDURE SPad(F: Frame; x, y, k: INTEGER); (* draw the k-th south pad at x, y *)
PROCEDURE WPad(F: Frame; x, y, k: INTEGER); (* draw the k-th west pad at x, y *)
PROCEDURE EPad(F: Frame; x, y, k: INTEGER); (* draw the k-th east pad at x, y *)

PROCEDURE NSRep(F: Frame; col, x, y, rep: INTEGER; east: BOOLEAN); (* draw the rep-th repeater at x, y in the east, if desired *)
PROCEDURE WERep(F: Frame; col, x, y, rep: INTEGER; north: BOOLEAN); (* draw the rep-th repeater at x, y in the north, if desired *)

We will now discuss procedure Restore in further detail. It makes use of procedures Translate, Cell, xPad, and *Rep shown above. First, the procedure removes all visible marks inside the frame. Then, it updates the coordinate fields appropriately and clears the necessary area on the screen.

The display area is divided into small tiles by a grid of side length RepW. All symbols appearing on the screen are aligned to that grid and their size is a multiple of the basic grid size (cells, pads, repeaters). This way, determining the position of the mouse and the mapping of its coordinates to model coordinates becomes much easier (see Edit below). Procedure Translate does this mapping.

After translation, the design is drawn row after row inside a loop, until the frame is filled with cells, repeaters, and pads.

PROCEDURE Restore(F: Frame);
VAR
  pen: Pen; (* turtle style pen *)
  u, v, cellU, cellV, secU, secV: INTEGER;
  x, x0, y, y0: INTEGER;
  repCol, repCol0, repRow: BOOLEAN;
BEGIN
  Oberon.RemoveMarks(F.X, F.Y, F.W, F.H); (* erase pointer and mouse cursor *)
  F.X1 := F.X + F.W; F.Y1 := F.Y + F.H; F.x := F.X + F.xa; F.y := F.Y1 + F.ya;

  IF ~F.print THEN Display.ReplConst(Display.black, F.X, F.Y, F.W, F.H, Display.replace) END; (* erase *)
  Translate(F, F.X, F.Y, u, v, cellU, cellV, secU, secV, repCol0, repRow);
  (* u, v: F.X, F.Y mapped to repeater grid 
     cellU, cellV: ~ cell grid 
     secU, secV: ~ sector grid 
     repCol0, repRow -> it's in a repeater column/row *)
  IF secU < 0 THEN x0 := F.X - PadW; repCol0 := TRUE; cellU := 0
  (* model's x-origin is visible, start with W-pads *)
  ELSE x0 := F.X + cellU*CellW + secU*RepW (* model's x-origin is not visible, start within cells *)
  END;
  IF secV < 0 THEN y0 := F.y - PadW; repRow := TRUE; cellV := 0
  (* model's y-origin is visible, start with S-pads *)
  System.Close;
ELSE y0 := F.y + cellV*cellW + secV*repW (*model’s y-origin is not visible, start within cells*)
END;
v := cellV; y := y0; pen.F := F; pen.col := ConnCol;
LOOP
IF (y >= F.Y1) OR ~repRow & (v >= Dim) OR (v > Dim) THEN EXITEND; (*finished with this design*)
END;
v := cellV; y := y0; pen.F := F; pen.col := ConnCol;
END;

```
ELSE
yO := F.y + cellV*cellW + secV*repW (*model’s y-origin is not visible, start within cells*)
END;
v := cellV; y := y0; pen.F := F; pen.col := ConnCol;
LOOP
IF (y >= F.Y1) OR ~repRow & (v >= Dim) OR (v > Dim) THEN EXITEND; (*finished with this design*)
END;
v := cellV; y := y0; pen.F := F; pen.col := ConnCol;
END;

```

For a better understanding of the editor's complex drawing operations, procedure Cell and the procedure it uses are explained in more detail. Each cell is responsible for the area it takes up on the screen. Therefore, it draws its contents — i.e. the routing mode and state it implements — as well as the local and express busses running along its side. This way, no global drawing has to take place for busses and updates can be kept local to single or multiple cells. Furthermore, the source code can be structured in an intelligible way.

Due to the cell's complex nature, the patterns inside it are drawn as such: i.e. we have defined a font containing patterns for displaying the various states and routing modes. Indices into this font are used to draw the appropriate patterns. Procedure CellPattern is responsible for drawing the contents of a cell (e.g. Xor and Nand gate, figure 16 on the left). The procedure must determine which inputs are activated in
the cell and use the corresponding pattern indices to draw the state accordingly (requirement 4). If only one input is chosen for the same state (figure 16 in the middle), the cell negates this input, as for the other input a logical "1" is supplied. Also, despite the fact that the routing mode is Write for all three cells, the local bus symbol is only drawn for the cell that actually writes to the bus (figure 16 on the right, requirement 5). This in turn constitutes a problem for the cells in the same column or row. If any cell in the column or row writes to or reads from the local bus, all cells in that column or row must draw their bus segment. E.g. both cells to the left of the rightmost cell in figure 16 must draw a line for the local bus, despite the fact that they neither read from nor write to it. The aforementioned field cache in the frame descriptor points to a data structure containing the additional information for drawing bus segments. In procedure Cell below, the use of this cache can be seen clearly.

Note: One may argue that we could have used "normal" drawing routines for displaying the state and routing mode of a cell. Printing a design would then come nearly for free and would be scalable as well. But the complex patterns to be drawn would have slowed down screen updates considerably, so this option was considered inapplicable. A less expressive display could have been chosen ("~" for Not, "x" for And, "[]" for registers, etc.) but it was felt to be crucial to see clearly what the cell implements without having one more level of interpretation between seeing and understanding.

The following presents the detailed implementation of the cell-drawing procedures. CellPattern has to decide what pattern indices to use for drawing the contents of the cell. Therefore, it merely consists of two large case statements which look more complicated than they are.

**Fig. 16: Different views for different inputs and outputs**

The following presents the detailed implementation of the cell-drawing procedures. CellPattern has to decide what pattern indices to use for drawing the contents of the cell. Therefore, it merely consists of two large case statements which look more complicated than they are. Cell has to decide whether to draw the cell or not. This is used to implement requirement 5. The drawing mode for cells can be toggled with a command in the top-level module CL.

```
PROCEDURE CellPattern(F: Frame; col, x, y: INTEGER; VAR cell: CLGAs.Cell);
("draw contents of cell at x, y with color col")
VAR left, right, IL, IR, UL, UR: INTEGER;
in: BOOLEAN;
BEGIN
  IF cell.routing # CLGAs.None THEN (* something has to be drawn *)
    IL := 1; IR := 4; UL := 6; UR := 0BH;
    CASE cell.routing OF
      | CLGAs.Turn0: UR := 0AH
      | CLGAs.TurnB:
      | CLGAs.Read: UL := 8
      | CLGAs.Mux:
        IF cell.state IN (CLGAs.State2, CLGAs.State3) THEN UL := 16H
        ELSE UL := 9; UR := 0CH
      END
      | CLGAs.Write: IF (cell.ns1 # CLGAs.None) OR (cell.wel # CLGAs.None) THEN IL := 2 END
      | CLGAs.TS:
        IF (cell.ns1 # CLGAs.None) OR (cell.wel # CLGAs.None) THEN IL := 3; IR := 5; UL := 7; UR := 0DH
    END
  END;
```
ELSE uR := OAH
END
END;
IF cell.a = CLGAs.None THEN INC(uL, 60H) END; (* constant 1 *)
IF cell.b = CLGAs.None THEN INC(uR, 60H) END; (* constant 1 *)
IF F.print THEN ... print patterns ...
ELSE ... copy patterns IL, IR, uL, uR to the screen ...
END
END;
IF cell.state # CLGAs.None THEN (* something has to be drawn *)
in := TRUE; INC(y, 11);
CASE cell.state OF
| CLGAs.State0: left := 0EH; right := 0FH
| CLGAs.State1: left := 10H; right := 11H
| CLGAs.State2:
   IF cell.routing = CLGAs.Mux THEN left := 17H; right := 19H
   ELSIF cell.routing IN {CLGAs.TS, CLGAs.Turn0} THEN left := 0EH; right := 75H; in := FALSE
   ELSIF cell.routing = CLGAs.Read THEN
       in := FALSE;
       IF cell.b = CLGAs.None THEN left := 1EH; right := 1EH
       ELSE left := 12H; right := 13H
       END
   ELSIF cell.a = CLGAs.None THEN left := 1BH; right := 1DH
   ELSIF cell.b = CLGAs.None THEN left := 1CH; right := 1EH
   ELSE left := 12H; right := 13H
   END
| CLGAs.State3:
   IF cell.routing = CLGAs.Mux THEN left := 18H; right := 1AH
   ELSIF cell.routing IN {CLGAs.TS, CLGAs.Turn0} THEN left := 23H; right := 73H; in := FALSE
   ELSIF cell.routing = CLGAs.Read THEN
       in := FALSE;
       IF cell.b = CLGAs.None THEN left := 20H; right := 22H
       ELSE left := 14H; right := 15H
       END
   ELSIF cell.a = CLGAs.None THEN left := 20H; right := 22H
   ELSIF cell.b = CLGAs.None THEN left := 14H; right := 15H
   ELSE left := 14H; right := 15H
   END
END;
IF in & (cell.a = CLGAs.None) & (cell.b = CLGAs.None) THEN
   INC(left, 60H); INC(right, 60H)
END;
IF F.print THEN ... print patterns ...
ELSE ... copy patterns left, right to the screen ...
END
END
END CellPattern;

PROCEDURE Label(F: Frame; u, v, x, y: INTEGER; sig: SHORTINT);
... draw label at cell or pad u, v annotating signal sig at location x, y ...

PROCEDURE Cell(F: Frame; x, y, u, v: INTEGER); (* draw cell u, v at x, y (cell's lower left corner) *)
VAR cell: CLGAs.Cell; su, sv, len: INTEGER; drawCell: BOOLEAN; cache: BusCache;
PROCEDURE LOut(dir: SHORTINT); (* draw arrow to local bus *)
PROCEDURE Lln(dir: SHORTINT); (* draw arrow from local bus *)
PROCEDURE LConn(dir: SHORTINT); (* draw line to/from local bus *)
BEGIN

cell := F.a.c[u, v]; cache := F.cache;
drawCell := (cellMode IN F.mode) OR (cell.routing+cell.state+cell.a+cell.b+cell.wel+cell.nsl # 6*CLGAs.None);
(* draw cell only if all cells should be drawn or if it contains something *)

IF drawCell THEN

Rect(F, CellCol, x+12, y+12, CellLen+2, CellLen+2, 2); (* cell frame *)
CellPattern(F, ConnCol, x+15, y+14, cell); (* contents *)
CASE cell.a OF
(u-input"
I
CLGAs.None: I
CLGAs.North:
IF (v MOD Sector = Sector-1) & (v II Dim-1) THEN len := 11+RepW ELSE len := 11 END;
(* if at repeater boundaries, draw a long arrow *)
DnArrow(F, ConnCol, x+AOff, y+(CeiiW-11), len)
... similar for south, east, west ...
END;
... similar for b-input ...
IF cell.routing IN {CLGAs.Write, CLGAs.TS} THEN LOut(cell.nsl); LOut(cell.wel)
ELSIF (cell.routing = CLGAs.None) OR (cell.routing IN {CLGAs.TumB, CLGAs.TurnO}) THEN
LConn(cell.nsl); LConn(cell.wel)
ELSE Ln(cell.nsl); Ln(cell.wel)
END;
Block(F, ConnCol, x+(BOff-2), y+(CeiiW-10), 5, 1);
Block(F, ConnCol, x+(BOff-2), y+(CeiiW-9), 1, 9); (* north output markers, B *)
Block(F, ConnCol, x+(CellW–(AOff+2)), y+(CellW–11), 5, 2);
Block(F, ConnCol, x+(CellW–AOff), y+(CellW–9), 1, 9); (* north output markers, A *)
... similar for south, east, west ...
END;
su := u DIV Sector; sv := v DIV Sector;
(* draw local and express bus lines, if it is necessary *)
IF cache.locaiWE[u, sv, CLGAs.West MOD 2] THEN Block(F, LBusCol, x+3, y, 1, CellW) END;
IF cache.locaiWE[u, sv, CLGAs.East MOD 2] THEN Block(F, LBusCol, x+(CellW–3), y, 1, CellW) END;
(* NS local bus lines *)
... similar for north-south express bus lines ...
... similar for east-west local and express bus lines ...
(* labels have to be drawn after all lines! *)
IF drawCell THEN (* draw labels for A- and B-output *)
Label(F, u, v, x+(AOff–10), y+2, CLGAs.AOut);
Label(F, u, v, x+(CellW–Boff–2), y+2, CLGAs.BOut)
END;
END Cell;

Editing a Design

When working on a display oriented problem, quick response from the computer is mandatory.
Interaction is by definition taking place between two parties, and if one party should wait for the other, it should clearly not be the user! Therefore, during the development of interactive programs – and especially graphics editors – one must be careful when designing the part that reacts to the user’s input. Locality of updates is one major design criterion. In the CL-Editor, for instance, upon editing a cell’s contents, it is the cell’s responsibility to redraw itself. Not the whole design should get redrawn, but only those parts that are affected by the operation (requirement 2, 3). As stated earlier, a problem arises with local busses. They may affect a whole column or row of eight cells. When a cell writes to the local bus, all cells in the same column or row must draw their local bus segment as well. In that case we update the whole column or row.
In the Oberon System, each visible frame has an associated handler which processes messages sent by the system. The handler dispatches actions according to the type of the message. Currently, only Oberon.InputMsg is of our concern. If the user moves the mouse into a frame, the Oberon loop sends an Input message to that frame containing the mouse's coordinates and pressed keys. Our handler calls procedure Edit, if keys were depressed.

PROCEDURE Edit(F: Frame; x, y: INTEGER; keys: SET); (* handle mouse at x, y in F with initial keys *)

PROCEDURE Handle(f: Display.Frame; VAR M: Display.FrameMsg);
VAR F: Frame;
BEGIN
  F := f(Frame);
  IF M IS Oberon.InputMsg THEN
    WITH M: Oberon.InputMsg DO
      IF M.id = Oberon.track THEN
        IF M.keys # {} THEN Edit(F, M.X, M.Y, M.keys) (* buttons were clicked *)
        ELSE Oberon.DrawCursor(Oberon.Mouse, Oberon.Arrow, M.X, M.Y) END
      END
    END
  END
  ... other messages ...
END Handle;

Procedure Edit makes up the lion's share of the editor's code dealing with user interaction. It has to distinguish between three different classes of operations: Editing a cell, a pad, or a repeater; setting labels to cells or pads; selecting, moving, or copying a cell stretch. Editing is performed with the help of two procedures – EditConnections and Menu – presented below. Setting labels is done with the local procedures SetLabel and PadLabel. Selecting, moving, and copying is done with SelectCells (which is not presented) and Copy. Due to the grid structure of the screen introduced in section Displaying a Design, the positions of sensitive areas can be specified in grid coordinates relative to a symbol's origin (constants left, middle, right below). Thus, determining the mouse position becomes easier. Despite this method, the procedure is still very large and contains many comparisons for the location of the mouse.

PROCEDURE EditConnections(F: Frame; cellU, cellV, u, v: INTEGER);
CONST left = 1; middle = RepPerCell # 2; right = RepPerCell# 2; (* coordinates of A, B, L-output south *)

PROCEDURE Toggle(VAR selector: SHORTINT; dir: SHORTINT); (* set direction or toggle if same *)
BEGIN
  IF selector = dir THEN selector := CLGAs.None ELSE selector := dir END;
  UpdateCells(F, cellU, cellV, 1, 1) (* update this cell only *)
END Toggle;

PROCEDURE Setl(dir: SHORTINT);
... similar to toggle, has to deal with corner-turns, though ...
BEGIN
  IF v = RepPerCell# 1 THEN (* north *)
    IF u = left THEN Toggle(F.a.c[cellU, cellV].a, CLGAs.North) (* toggle A-input *)
    ELSIF u = middle THEN Setl(CLGAs.North) (* set/toggle L-input/output *)
    ELSIF u = right THEN Toggle(F.a.c[cellU, cellV].b, CLGAs.North) (* toggle B-input *)
  END
  ... similar for south, east, west ...
END EditConnections;
PROCEDURE Menu(F: Frame; x0, y0, u, v, xn, yn, xw, yw: INTEGER; pattern: PatternProc; update: UpdateProc);
... see section Menus ...

PROCEDURE Edit(F: Frame; x, y: INTEGER; keys: SET);
CONST A = 1; B = RepPerCell-2;  (* coords of A & B-output south *)
VAR
  V: Viewers.Viewer; dF: Frame; cell: CLGAs.Cell;
  keySum: SET;
  x1, y1, repU, repV, cellU, cellV, secU, secV, u, v, h, du, dv: INTEGER;
  repU1, repV1, cellU1, cellV1, secU1, secV1, u1, v1, rep, pad, padM: INTEGER;
  repCol, repRow, repCol1, repRow1, ml: BOOLEAN;
  sig: SHORTINT;

PROCEDURE TrackMouse(VAR sum: SET; VAR xO, yO: INTEGER);
... draw mouse cursor and sum up pressed keys ...

PROCEDURE Setlabel(lu, lv: INTEGER; s: SHORTINT);
VAR
  old: CLGAs.Label; name: ARRAY CLGAs.LabelLen OF CHAR;
  Sel: Texts.Text; S: Texts.Scanner;
  keySum: SET;
  x1, y1, repU, repV, cellU, cellV, secU, secV, u, v, w, h, du, dv: INTEGER;
  repU1, repV1, cellU1, cellV1, secU1, secV1, u1, v1, rep, pad, padM: INTEGER;
BEGIN
  Oberon.GetSelection(Sel, beg, end, time);
  If (time >= 0) & (keySum = {ML, MM}) THEN Texts.OpenScanner(S, Sel, beg);
  Texts.Scan(S);
  If (S.class = Texts.Name) OR (S.class = Texts.String) THEN old := CLGAs.This(F.a, S.s);
  IF old # NIL THEN Str(S.s); Str(" already defined at"); Int(old.u); Int(old.v); Ln ELSE CLGAs.Delete(F.a, lu, lv, s, name);
  CLGAs.Insert(F.a, S.s, lu, lv, s, old) END ELSE RETURN END
ELSIF keySum = {ML, MR} THEN CLGAs.Delete(F.a, lu, lv, s, name) END;

PROCEDURE PadLabel(pu, pv: INTEGER);
... set label at pad pu, pv ...

PROCEDURE Copy;
(* uses u, v, du, dv, F, dF *)
VAR u0, v0: INTEGER;

PROCEDURE MoveLabel(s: SHORTINT);
VAR l: CLGAs.Label; name: ARRAY CLGAs.LabelLen OF CHAR;
BEGIN
  l := CLGAs.LabelAt(F.a, u, v, s);
  IF l # NIL THEN
    IF ml THEN CLGAs.Delete(F.a, u, v, s, name) END;
    CLGAs.Delete(dF.a, u0, v0, s, name); CLGAs.Insert(dF.a, l.name, u0, v0, s, l)
  END
END MoveLabel;
BEGIN
u0 := u+du; v0 := v+dv; f.a[c[u0, v0]] := f.a[c[u, v]]

IF ml THEN MoveLabel(CLGAs.AOut); MoveLabel(CLGAs.BOut); f.a[c[u, v]] := cell
ELSIF f.a # d.f.a THEN MoveLabel(CLGAs.AOut); MoveLabel(CLGAs.BOut)
END
END

BEGIN

Translate(F, x, y, repU, repV, secU, secV, repCol, repRow);  (* map x, y to model coordinates *)
IF keys{ML, MR} # () THEN  (* edit/select *)
  ml := keys = {ML};
IF (repU >= SWPadRep) & (repU <= NEPadRep) & (repV >= SWPadRep) & (repV <= NEPadRep) THEN
  IF A <= u & u <= B & A <= v & v <= B THEN
    (* we're inside a cell *)
    u := (repU-secU) MOD RepPerCell;
    v := (repV-secV) MOD RepPerCell;
    IF (A <= u) & (u <= B) & (A <= v) & (v <= B) THEN
      SelectCells(F, cellU, cellV, 1, 1, FALSE);
    END
    (* extend selection from lower left to upper right *)
    InputMouse(keys, x1, y1);
    Oberon.DrawCursor(Oberon.Mouse, Oberon.Arrow, x1, y1);
    Translate(F, x1, y1, repU1, repV1, secU1, secV1, repCol1, repRow1);
    u1 := (repU1-secU1) MOD RepPerCell;
    v1 := (repV1-secV1) MOD RepPerCell;
    IF (A <= u1) & (u1 <= B) & (A <= v1) & (v1 <= B) THEN
      SelectCells(F, cellU1, cellV1, Max(1, cellU1-cellU+1), Max(1, cellV1-cellV+1), TRUE)
    END
    END
  UNTIL keys ={}
END
ELSIF ml THEN  (* we want to edit connections/labels *)
  TrackMouse(keySum, x1, y1);
  IF keySum = cancel THEN RETURN END;
  repU := (x1-F.x) DIV RepW; repV1 := (y1-F.y) DIV RepW;
  IF (repU = repU1) & (repV = repV1) THEN
    IF keySum = {ML} THEN EditConnections(F, cellU1, cellV1, u, v)
    ELSIF keySum = {MM, MR} # () & (v = 0) THEN  (* set label *)
      IF u = A THEN sig := CLGAs.AOut
      ELSIF u = B THEN sig := CLGAs.BOut
      ELSE RETURN END
    ELSE TrackMouse(keySum, x1, y1)
  END
ELSIF ml THEN  (* not cell *)
  IF (SWPadRep-repV >= -1) OR (NEPadRep-repV <= 1) THEN  (* north or south pad *)
    Pad(repU-secU);
  IF (pad >= 0) & (pad < Dim-3) & (padM < RepPerCell-1) & ~ODD(pad) THEN
    IF repV = NEPadRep THEN  (* north pad *)
      Menu(F, x, y, pad DIV 2, Dim, 4, 1, 28, 25, PadPattern, PadUpdate)
    ELSIF repV = NEPadRep-1 THEN PadLabel(pad DIV 2, Dim)  (* set label at pad *)
END
ELSIF ... similar for south pad ...
ELSIF ... similar for east and west pads ...
ELSIF ~repCol & repRow THEN (* north-south repeater *)
    rep := (repU-secU) MOD 5;
    IF rep = 4 THEN (* east part of north-south repeater *)
        Menu(F, x, y, cellU, cellV DIV 8 – 1, 7, 4, RepW, RepW+2, RepPatternE, RepUpdateE)
        (* edit repeater *)
    ELSIF rep = 0 THEN ... similar for west part ...
ELSIF ... similar for east-west repeater ...
ELSE TrackMouse(keySum, x1, y1) (* we’re not near anything we can edit, track the mouse *)
END
ELSE TrackMouse(keySum, x1, y1) (* we cannot edit without mouse left, track the mouse *)
END (* IF repU >= ... *)
ELSIF keys = {MM} THEN (* shift plane / move/copy selection *)
    TrackMouse(keySum, x1, y1);
    IF keySum = cancel THEN RETURN END;
    IF (keySum = {MM}) & ((x # x1) OR (y # y1)) THEN
        INC(F.Xa, x1-x); INC(F.Ya, y1-y);
    END;
ELSIF keySum # {MM} THEN (* MM, ML or MM, MR, copy or move selection *)
    ml := ML IN keySum;
    V := Viewers.This(x1, y1); (* which viewer are we in? *)
    IF V.dsc.next IS Frame THEN dF := V.dsc.next(Frame); F := Selected()
    (* get source frame containing most recent selection, dF is destination frame dF could be the same as F *)
    ELSE F := NIL END;
END (* IF cell *)
ELSE TrackMouse(keySum, x1, y1) (* we’re not near anything we can edit, track the mouse *)
END (* IF repU >= ... *)
ELSIF keys = {MM} THEN (* shift plane / move/copy selection *)
    TrackMouse(keySum, x1, y1);
    IF keySum = cancel THEN RETURN END;
    IF (keySum = {MM}) & ((x # x1) OR (y # y1)) THEN
        INC(F.Xa, x1-x); INC(F.Ya, y1-y);
    END;
ELSIF keySum # {MM} THEN (* MM, ML or MM, MR, copy or move selection *)
    ml := ML IN keySum;
    V := Viewers.This(x1, y1); (* which viewer are we in? *)
    IF V.dsc.next IS Frame THEN dF := V.dsc.next(Frame); F := Selected()
    (* get source frame containing most recent selection, dF is destination frame dF could be the same as F *)
    ELSE F := NIL END;
END (* IF cell *)
ELSE TrackMouse(keySum, x1, y1) (* we cannot edit without mouse left, track the mouse *)
END (* IF repU >= ... *)
ELSIF keys = {MM} THEN (* shift plane / move/copy selection *)
    TrackMouse(keySum, x1, y1);
    IF keySum = cancel THEN RETURN END;
    IF (keySum = {MM}) & ((x # x1) OR (y # y1)) THEN
        INC(F.Xa, x1-x); INC(F.Ya, y1-y);
    END;
ELSIF keySum # {MM} THEN (* MM, ML or MM, MR, copy or move selection *)
    ml := ML IN keySum;
    V := Viewers.This(x1, y1); (* which viewer are we in? *)
    IF V.dsc.next IS Frame THEN dF := V.dsc.next(Frame); F := Selected()
    (* get source frame containing most recent selection, dF is destination frame dF could be the same as F *)
    ELSE F := NIL END;
END (* IF cell *)
ELSE TrackMouse(keySum, x1, y1) (* we cannot edit without mouse left, track the mouse *)
END (* IF repU >= ... *)
ELSIF keys = {MM} THEN (* shift plane / move/copy selection *)
    TrackMouse(keySum, x1, y1);
    IF keySum = cancel THEN RETURN END;
    IF (keySum = {MM}) & ((x # x1) OR (y # y1)) THEN
        INC(F.Xa, x1-x); INC(F.Ya, y1-y);
    END;
ELSIF keySum # {MM} THEN (* MM, ML or MM, MR, copy or move selection *)
    ml := ML IN keySum;
    V := Viewers.This(x1, y1); (* which viewer are we in? *)
    IF V.dsc.next IS Frame THEN dF := V.dsc.next(Frame); F := Selected()
    (* get source frame containing most recent selection, dF is destination frame dF could be the same as F *)
    ELSE F := NIL END;
END (* IF cell *)
ELSE TrackMouse(keySum, x1, y1) (* we cannot edit without mouse left, track the mouse *)
END (* IF repU >= ... *)
ELSIF keys = {MM} THEN (* shift plane / move/copy selection *)
    TrackMouse(keySum, x1, y1);
    IF keySum = cancel THEN RETURN END;
    IF (keySum = {MM}) & ((x # x1) OR (y # y1)) THEN
        INC(F.Xa, x1-x); INC(F.Ya, y1-y);
    END;
ELSIF keySum # {MM} THEN (* MM, ML or MM, MR, copy or move selection *)
    ml := ML IN keySum;
    V := Viewers.This(x1, y1); (* which viewer are we in? *)
    IF V.dsc.next IS Frame THEN dF := V.dsc.next(Frame); F := Selected()
    (* get source frame containing most recent selection, dF is destination frame dF could be the same as F *)
    ELSE F := NIL END;
END (* IF cell *)
ELSE TrackMouse(keySum, x1, y1) (* we cannot edit without mouse left, track the mouse *)
END (* IF repU >= ... *)
Although we have already split Edit into smaller parts, it is still a long procedure. One may note that non-trivial comparisons are needed to determine whether the mouse is inside a sensitive area presented in figure 15. These comparisons blow up the code quite a lot, but it was felt advantageous to concentrate this distinction in one place to narrow down the scope when trying to find an error. As can be seen in the implementation, there is no magic involved but only a lot of code. Still, the complexity of the basic cell and the different possibilities occurring during an editing operation make this procedure complex. Splitting it further into smaller procedures would not have helped because of the many variables used in the different parts of the procedure. These could have been passed as parameters to the smaller procedures, but we did not see any clear advantage in that proceeding.

Updates via Messages

After having updated the data structure describing a cell, a pad, or a repeater, the Edit procedure has to notify all views displaying the changed design. Since an editor cannot — and should not — know all designs it is currently displaying, Oberon introduced the concept of broadcasts using message records. There are three kinds of update messages all derived from the base type UpdateMsg. In order to check if a frame is affected by the message, the changed design is stored in field a of the message. By comparing this field against its local field a, the frame can change its display according to the change in the design. The fields u and v denote the coordinates of the affected cell, pad, or repeater. As can be seen in the implementation of Edit above, UpdateCells gets called when a cell has been changed. This procedure simply broadcasts a CellMsg to all viewers. The two additional fields w and h in CellMsg are used to update multiple cells with one message send, which is used after copying a cell stretch, or when a local bus has to be drawn in a column or a row.

```
UpdateMsg = RECORD (Display.FrameMsg)
  a: CLGAs.GA;  (* design being changed *)
  u, v: INTEGER  (* coordinates of change *)
END;

CellMsg = RECORD (UpdateMsg) w, h: INTEGER END;  (* cell stretch u, v, w, h *)

PadMsg = RECORD (UpdateMsg) side: SHORTINT END;  (* pad on side (N, S, E, W) *)

RepMsg = RECORD (UpdateMsg) typ: SHORTINT END;  (* repeater with typ (N, S, E, W) *)

PROCEDURE UpdateCells(F: Frame; u, v, w, h: INTEGER);
  VAR msg: CellMsg;
BEGIN
  InitLocalBusCache(F);  (* update cache for displaying local busses *)
  msg.a := F.a; msg.u := u; msg.v := v; msg.w := w; msg.h := h;
  Viewers.Broadcast(msg)
END UpdateCells;

PROCEDURE UpdatePad(F: Frame; u, v: INTEGER; side: SHORTINT);  ... similar ...
PROCEDURE UpdateRep(F: Frame; u, v: INTEGER; typ: SHORTINT);  ... similar ...
```

When copying cell stretches, the editor has to find the most recent selection. For that a SelMsg is broadcast to the viewer system. If a selection exists, the field f in the message record points to the frame containing the most recent selection. The time field is used to find the most recent selection.
SelMsg = RECORD (Display.FrameMsg)
    f: Frame;  (* NIL or frame containing most recent selection *)
    time: LONGINT (* time of selection *)
END;

Finally, we present the implementation of the message handler. We have already presented this procedure in section Displaying a Design and left out the update messages then. According to the message's type the corresponding redraw procedure can be called which performs the necessary screen update. MarkMenu is used to append an exclamation mark to the end of the menu text to indicate the change of a design to the user.

PROCEDURE Handle(f: Display.Frame; VAR M: Display.FrameMsg);
    VAR F: Frame;
BEGIN
    F := f(Frame);
    ... other messages ...
    ELSIF M IS SelMsg THEN (* does this frame contain the most recent selection? *)
        WITH M: SelMsg DO
            IF F.selected & (M.time < F.time) THEN M.f := F; M.time := F.time END
        END
    ELSIF M IS CellMsg THEN (* if this frame displays the design in the message, update it *)
        WITH M: CellMsg DO
            IF M.a = F.a THEN RedrawCells(F, M.u, M.v, M.w, M.h); MarkMenu(F) END
        END
    ELSIF M IS PadMsg THEN
        WITH M: PadMsg DO
            IF M.a = F.a THEN RedrawPad(F, M.u, M.v, M.side); MarkMenu(F) END
        END
    ELSIF M IS RepMsg THEN
        WITH M: RepMsg DO
            IF M.a = F.a THEN RedrawRep(F, M.u, M.v, M.typ); MarkMenu(F) END
        END
    ... other messages ...
END Handle;

Menus

All menus are displayed by means of a single generic procedure Menu. By supplying different draw and update procedures at call time, this procedure can edit cells, pads, and repeaters with the same code. The usage of procedure Menu can be seen in procedure Edit above. For the appearance of menus cf. with figures 12-14. If a menu of the desired size can be displayed in the frame, the procedure saves the current foreground. Then, the area the menu will cover is cleared and a frame surrounding the menu as well as horizontal and vertical lines separating the individual menu items are drawn. By means of the draw procedure supplied in the procedure variable pattern, the menu items are drawn and highlighted with a frame, if necessary. Once the user has selected an item, the foreground is restored and the selected item's coordinates are passed to an update procedure supplied in procedure variable update. This procedure changes the design and calls the necessary view-update procedure which in turn sends an update message to all frames.

PROCEDURE Menu(F: Frame; xo, yo, u, v, xn, yn, xw, yw: INTEGER; pattern: PatternProc; update: UpdateProc);
(* display a menu at xo, yo with xn items of width xw per row and yn rows of height yw for editing
 item u, v of a design. *)
Use pattern for displaying the various items and update to change the data structure
VAR
  keySum, keys: SET;
  width, height, x, y, x1, y1, X, Y, oldX, oldY, i, j: INTEGER;
PROCEDURE Flip(i, j: INTEGER); (* invert menu item i, j *)
BEGIN
  IF (0 <= i) & (i < x) & (0 <= j) & (j < y) THEN
    Display.ReplConst(MenuCol, x+i*xw+i, y+j*yw+j, xw-2, yw-2, Display.invert)
END
END Flip;

BEGIN
  width := xn*xw+(xn-1)+4; height := yn*yw+(yn-1)+4;
  IF (F.W < width) OR (F.H < height) THEN RETURN END; (* frame too small *)
  IF x0+width > F.X1 THEN x0 := F.X1-width ELSIF x0 < F.X THEN x0 := F.X END;
  IF y0+height > F.Y1 THEN y0 := F.Y1-height ELSIF y0 < F.Y THEN y0 := F.Y END;
  (* move menu to appropriate position within frame *)
  Oberon.RemoveMarks(x0, y0, width, height); Oberon.FadeCursor(Oberon.Mouse);
  Display.CopyBlock(x0, y0, width, height, x0, -height, Display.replace); (* save foreground *)
  Display.ReplConst(MenuCol, x0, y0, width, height, Display.replace); (* clear *)
  Rect(F, MenuCol, x0, y0, width, height, 2);
  x := x0+2+xw; y := y0+2; x1 := x+(xn-1)*(XW+1);
  (vertical lines *)
  WHILE x < x1 DO Display.ReplConst(MenuCol, x, y, 1, height-4, drawMode); INC(x, xw+1) END;
  (horizontal lines *)
  x := x0+2; y := y0+2+yw; y1 := y+(yn-1)*(yw+1);
  WHILE y < y1 DO Display.ReplConst(MenuCol, x, y, width-4, 1, drawMode); INC(y, yw+1) END;
  j := 0; y := y0+3;
  WHILE j < yn DO
    i := 0; x := x0+3;
    WHILE i < xn DO
      IF pattern(F, u, v, x, y, i, j) THEN Rect(F, MenuCol1, x-1, y-1, xw, yw, 2) END;
      (* if changed item currently has this mode, highlight it *)
      INC(i); INC(x, xw+1)
    END;
    INC(j); INC(y, yw+1)
  END;
  x := x0+3; y := y0+3; oldX := 0; oldY := 0; keySum := {ML}; Flip(0, 0);
REPEAT
  InputMouse(keys, x1, y1); keySum := keySum + keys;
  Oberon.DrawCursor(Oberon.Mouse, Oberon.Arrow, x1, y1);
  X := (x1-(x0+2)) DIV (xw+1); Y := (y1-(y0+2)) DIV (yw+1);
  IF (X # oldX) OR (Y # oldY) THEN Flip(oldX, oldY); Flip(X, Y); oldX := X; oldY := Y END
UNTIL keys = {};
  Oberon.FadeCursor(Oberon.Mouse);
  Display.CopyBlock(x0, -height, width, height, x0, y0, Display.replace); (* restore foreground *)
  IF (keySum # cancel) & (0 <= oldX) & (oldX < xn) & (0 <= oldY) & (oldY < yn)
    OR (keySum = {ML, MM}) THEN
    update(F, u, v, oldX, oldY, keySum)
  END
END
END Menu;

PROCEDURE CellMenuPattern(F: Frame; u, v, x, y, i, j: INTEGER): BOOLEAN;
  (* draw menu picture i, j at x, y, return cell u, v has current mode or state *)
VAR cell: CLGAs.Cell;
BEGIN
  cell.a := CLGAs.North; cell.b := CLGAs.North;
  cell.a := CLGAs.North; cell.b := CLGAs.North;
cell.routing := CLGAs.None; cell.state := CLGAs.None;

IF j = 0 THEN
  CASE i OF
  | 0: cell.state := SHORT(i+CLGAs.State0); CellPattern(F, MenuCol, x+5, y, cell)
  | 1..3: cell.state := CLGAs.State2; cell.routing := CLGAs.Mux; CellPattern(F, MenuCol, x, y, cell)
  | 5: cell.state := CLGAs.State3; cell.routing := CLGAs.Mux; CellPattern(F, MenuCol, x, y, cell)
END
ELSE
  cell.routing := SHORT(i+CLGAs.Write); CellPattern(F, MenuCol, x, y, cell);
  IF i >= 4 THEN CopyPattern(MenuCol, x+10, y+13, 53H) END (* corner turn *)
END;
RETURN cell u, v has current mode or state

END CellMenuPattern;

PROCEDURE CellUpdate(F: Frame; u, v, i, j: INTEGER; keySum: SET);
(* update design and display of cell u, v with menu selection i, j depending on keys keySum *)
VAR oldR, routing, state:
BEGIN
  IF MM IN keySum THEN (* mouse left, mouse middle -> initialize cell *)
    Fac[u, v].routing := CLGAs.None; Fac[u, v].state := CLGAs.None;
    Fac[u, v].a := CLGAs.None; Fac[u, v].b := CLGAs.None;
    CLGAs.Delete(F.a, u, v, name); CLGAs.Delete(F.a, u, v, BOut, name);
    UpdateCells(F, u-u MOD Sector, v, Sector, 1); UpdateCells(F, u, v-v MOD Sector, 1, Sector)
  ELSE
    routing := Fac[u, v].routing; state := Fac[u, v].state;
    IF j = 0 THEN (* selected state or mux in lower row of menu *)
      CASE i OF
      | 0..3: state := SHORT(i)
      | 4: state := CLGAs.State2; routing := CLGAs.Mux (* mux routing and xor state *)
      | 5: state := CLGAs.State3; routing := CLGAs.Mux (* mux routing and register state *)
      END
    ELSE routing := SHORT(i) (* selected routing in upper row of menu *)
    END;
    oldR := Fac[u, v].routing;
    IF (oldR # routing) OR (Fac[u, v].state # state) THEN (* if something was changed *)
      IF (routing IN {CLGAs.Turn0, CLGAs.TurnB}) (* special treatment for L-busses *)
        & ~(oldR IN {CLGAs.Turn0, CLGAs.TurnB}) THEN
        IF Fac[u, v].nsl = CLGAs.None THEN
          Fac[u, v].nsl := CLGAs.North; UpdateCells(F, u-u MOD Sector, v, Sector, 1)
        END;
        IF Fac[u, v].wel = CLGAs.None THEN
          Fac[u, v].wel := CLGAs.West; UpdateCells(F, u, v-v MOD Sector, 1, Sector)
        END
      ELSEIF (routing IN {CLGAs.Read, CLGAs.Mux}) & ~(oldR IN {CLGAs.Read, CLGAs.Mux}) OR
        ~(routing IN {CLGAs.Read, CLGAs.Mux}) & (oldR IN {CLGAs.Read, CLGAs.Mux}) THEN
        Fac[u, v].nsl := CLGAs.None; Fac[u, v].wel := CLGAs.None; (* init if no cornerturn *)
        UpdateCells(F, u-u MOD Sector, v, Sector, 1);
        UpdateCells(F, u, v-v MOD Sector, 1, Sector)
      END;
      Fac[u, v].routing := routing; Fac[u, v].state := state; UpdateCells(F, u, v, 1, 1)
    END
  END;
END CellUpdate;

... similar procedures for pads and repeaters ...
Printing

A design can be printed by setting the print field in a frame to true and calling the Redraw procedure for that frame. Since we are using a font to display cell contents, it has to be present on the printer as well. Currently, we simply download the screen font to the printer and redraw everything. Therefore, designs on the printer appear about four times smaller in each direction than on the screen due to the different resolutions used. However, since printing a design is mainly used for getting the big picture, this mechanism was felt to be sufficient and did not raise major complaints from the users. In the future, a scalable display and print mechanism would be desirable. This would require some coding effort and a good solution to the font problem, for instance scalable fonts.

Consistency Checks

The electrical viability of designs could be checked by the editor. For instance, if two cells write to the same bus, an electrical conflict arises when one cell drives the bus low while the other one is driving it high. Also, unconnected inputs should be detected, e.g. in a cell implementing a multiplexer, where the selector signal is not connected. All these types of design errors could potentially be detected by the editor with more or less effort. However, we have taken a different approach. While a design is under development, checking for these errors would be a waste of processor cycles, since a design is changed again and again. Only when it is complete and to be downloaded onto the chip, certain checks are performed. Therefore, we delay consistency checks until load time, i.e. the design is checked by module CLIloader. Further tools for checking a design against a formal description are available as well (CLChecker), but do not interfere with the editor's normal operation.

Note: The problem of following busses alone requires considerable coding effort and is currently implemented in CLChecker. The reader may try to find a simple solution for deciding, whether a bus running over repeater boundaries and cornerturns is written to twice or not! Beside that, for tri-stated bus-writes, no statements can be given on the electrical viability!

Anomalies in the Architecture

The CLI6002 input/output architecture consists of two types of IO cells: The A-type and the B-type. In figure 9, we did not specify the type of the involved cells, and in the editor, we treat all IO cells being of type A. The chip we are using in the student labs (CLI6002 84-pins), however, has an additional B-type cell between A-type cell 7 and 8, such that there are actually 16 IOs on each side of the chip. The editor supports only 15 IOs, as this additional B-type cell causes special cases throughout the whole editor. In addition, a design cannot be regularly laid out on the chip, if it uses this special B-type IO cell. Furthermore, if we used the 132-pin version of the CLI6002 FPGA, the problem would get even worse: In that chip, there are 24 IOs on each side, but the sequence of A- and B-types is as follows: A B A B A A B B A A B A B A B A A A B A B A A B A B A A B A. This has more similarity with a DNA sequence than with a clear IO structure for an FPGA! Apart from destroying the regularity of a program displaying such a chip, it is not at all clear how to design a circuit's input/output structure when there is no such regularity in the IO structure of the chip. The chosen solution of simply discarding the single B-type IO cell from our view was only possible with the smaller pin-out chip. However, we never noticed its absence.

Note: The data structure in module CLGAs supports 16 IOs but the editor uses only the first 15. Therefore, tools that want to use all 16 IOs are not restricted by the implementation of the data structure.
Extensibility

Being a one-target software solution, no provisions were taken to allow easy extensions of the editor or its data structures. For a bigger chip, a few constants could be changed in the source code and the whole package recompiled to allow editing the new chip. These constants could be changed into variables to allow configuration at run-time, but if a chip with a different IO architecture is used, some modifications to the drawing and editing procedures would have to be done as well. Through procedure variable hooks in the editor, even this problem could be solved at run-time but we did not provide this possibility. For the intended purpose the chosen approach worked out very well.

To support different FPGA architectures, a whole editor framework would have to be constructed. The approach taken in the CALLAS tools [Pfl93] would have to be extended to allow for extension of the editor itself. Currently, it is only possible to install different layout algorithms. All modules in such a framework, starting with the ones handling the basic data structures, would have to be configurable or programmed in an object-oriented fashion to allow for later extension. The long term goal could be a framework or test bed for evaluating different FPGA architectures and synthesis and layout algorithms for them.

Size and Comparison to CAL1

Many concepts in our editor come from an earlier version of an FPGA editor, namely the editor by N. Wirth for the CAL1 architecture [Kea89][Aig90]. CAL1’s simple cell, its regular overall design in general, and IO structure in particular make it very suitable for editors. A typical view of a design is shown in figure 17.
Due to its very nature, the CAL1 architecture demands little effort on the editor's side. The program can therefore be kept simple and easily manageable. A comparison of our editor with the CAL1 editor is made on a source and object code basis in the table below.

<table>
<thead>
<tr>
<th></th>
<th>CAL1 (lines)</th>
<th>obj</th>
<th>CL (lines)</th>
<th>obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structures</td>
<td>176</td>
<td>1656</td>
<td>297</td>
<td>4152</td>
</tr>
<tr>
<td>Editor</td>
<td>869</td>
<td>14804</td>
<td>1777</td>
<td>32240</td>
</tr>
<tr>
<td>Loader</td>
<td>463</td>
<td>4768</td>
<td>481</td>
<td>5068</td>
</tr>
<tr>
<td>Command Module</td>
<td>154</td>
<td>1952</td>
<td>408</td>
<td>6248</td>
</tr>
<tr>
<td>Total</td>
<td>1662</td>
<td>23180</td>
<td>2963</td>
<td>47708</td>
</tr>
</tbody>
</table>

As can be clearly seen, the editor modules differ by more than a factor of two. Our editor part consists of two separate modules, CLFramesD and CLFrames. This fact alone tells us something about the size difference. Much of the additional code is due to the complex drawing routines (notably Cell, CellPattern) which have to draw the contents of a cell according to the selected inputs. But also the additional architectural features like busses and repeaters add to the editor's size. The other modules are slightly bigger due to the additional features provided: The data structure module CLGAs stores designs in a machine-independent format and is therefore larger than its CAL1 counterpart. The loader is of equal size, although the bitstream format for CAL1 is very irregular. CL's command module provides additional commands for statistics and for setting clocks and resets.

Conclusions

In retrospect, we would hardly have done anything different than we did. The solution to the given problems are straightforward and work well in practise. If one had scalable fonts in the Oberon System, better printing and zooming could be implemented without much effort, but as it is now, the chosen solution seems good enough. The usage of the editor in the student course posed no problems at all. We ported the editor to MacOberon and DOSOberon, such that students could develop their designs at home. The fast adaption of all users to our system was encouraging and the positive feedback very rewarding.

The anomalies of the chosen target architecture gave us some problems that we wished to avoid. Future FPGA architectures should strive for a clear, orthogonal, and regular design of the basic cell as well as of the IO architecture. Clearly, the complexity of a Xilinx cell [Xil90] is even worse than CLI's and therefore, research directions in FPGA design should apply some of the lessons learnt by implementers of RISC microprocessor architectures. Building editors for low-level design of circuits is one problem. But the long-term goal would be the design and implementation of silicon compilers for FPGAs. Having a small, regular cell and a regular IO structure in the target architecture is felt mandatory for the successful implementation of such a system. Still, much work remains until satisfactory architectures appear on the market.

Acknowledgements

I would like to thank N. Wirth, for his guidance during this project. His pragmatism and focus on the essential were invaluable. Giving me the right hint at the right time without hindering creativity or innovative ideas was very helpful for the successful completion of this work. His CAL1 editor was our
starting point and always a source for compact and elegant solutions to a given problem.

I would like to thank my colleague, Stephan Gehring, for his collaboration. His stable implementations of the data structure module and the loader were the solid foundation I could build the editor on.

The CLI-Board used in the student labs was built by Immo Noack. His enthusiasm and constructive criticism were always encouraging.

References


Xil90  Xilinx, Inc., San Jose, CA. XC 4000 Logic Cell Array Family. 1990.
Appendix

Adder.Cli – A Small Example

The following figure presents 3 bits of an 8-bit adder/subtractor. The two arguments are fed from $D_i$ into registers at $x_i$ and $y_i$, then these values are added or subtracted with the circuitry on the right (4 cells for a full adder, marked with a rectangle). The result is put on a local bus by the rightmost cell and read out at the pad $D_i$. In column 5, one can see the carry chain. The vertical busses are used to distinguish between adding and subtracting and for controlling the multiplexers before the registers. This circuit has been implemented by students during the second exercise of the mentioned course.
CL-Editor User Manual

Stefan H.-M. Ludwig

The CL-Editor is a program package for the graphical design of circuits for the CLI6000 Field Programmable Gate Array architecture. The functionality of the software supports the fast design of circuits on the computer screen, downloading of the circuit onto the hardware of a CLI-Board for Ceres-3, and testing. Parts of designs can be copied into other designs, thereby allowing the construction of libraries of tested components. Before reading this manual, the user should be acquainted with the CLI6000 architecture.

Getting the Necessary Files from the Server

The editor package consists of the following files which are available on the Pluto server:

- CLGAs.obj  Data structures representing designs, loading and storing of designs
- CLLoader.obj  Loader for the CLI-Board for Ceres-3
- CLI.Scn.Fnt  Patterns for the editor
- CLFramesD.obj  Display procedures used by the editor
- CLFrames.obj  Editing operations
- CL.obj  Command module and programming interface for the CLI-Board
- CL.sym  Tool text with explanations
- Reg.Cli  Example design

First, CLTool has to be loaded from Pluto with the command Net.ReceiveFiles Pluto CLTool~. The corresponding Net.ReceiveFiles command in CLTool can then be activated to retrieve the remaining files from the server.

Opening, Storing, and Loading of Designs

The command CLOpen Name.Cli loads a design from disk and opens a viewer with a graphical representation of the design. File names of designs should end on Cli (convention). Like with other Oberon commands, an arrow (+) indicates the last text selection to be the argument of the command (CLOpen +).

If a design gets changed, an exclamation mark in the menu bar of the corresponding viewer indicates this fact. This way, all viewers showing modified designs are clearly marked.

Command CLStore in the menu bar stores a design to a file using the name shown in the menu bar. A backup of the old design is kept on disk under the name Name.Cli.Bak.

Hint: To rename a design, one can simply edit the name in the menu bar with the mouse (delete the name in the menu bar using the mouse and type the new name on the keyboard). After storing the design, it is available under the changed name.
CLLoad, another command in the menu bar, loads the displayed design to the CLI-Board of the Ceres-3 and notifies the user of its completion through a message in the Log viewer (Configuration successful/Configuration failed). If the design contains errors, a message with the coordinates of the erroneous cell is printed (→ CLLocate).

Editing: General Remarks

The editor is used consistently with the mouse. Mouse button assignments and their functions were kept compatible with the normal Oberon text operations as closely as possible. By pressing all three buttons at the same time, a started operation can be canceled.

The left button is used to edit cells, pads, or repeaters. In most cases, a menu will appear from which a certain item can be chosen. By leaving the menu with the mouse and releasing the button, the operation is canceled and no changes take place.

The middle button is used to shift the view of a design or to move or copy a cell selection.

The right button is used to select a cell stretch.

The command CLCells in the menu bar toggles between two display modes. Either all cells are shown – even the ones not used – or only those cells which actually have a content or get used for routing a bus. Furthermore, only those local and express busses are shown which get used by a cell or are connected together via a repeater (independent of the toggle state set by CLCells).

Editing Cells

Cells are edited with the left mouse button. Depending on the location where the button is pressed, different actions take place: Either the inputs or outputs of a cell are edited or the cell's content itself. Figure 1 below shows the sensitive areas of a cell. By pressing the left button in the respective area, the involved states get changed:

A/B inputs: The input is turned on or off (toggled). Depending on the selection of inputs the picture inside the cell changes as not selecting an input results in a "1" being supplied at that input. The inputs can be toggled on all four sides of the cell. Near the A output (fat terminal □) lies the B input and near the B output (thin terminal □) the A input. The lower outputs and the upper inputs correspond graphically with the patterns inside the cell. If other directions are chosen, one has to imagine the signal flow since the pattern is not rotated accordingly.

L-bus input/output: If the cell reads from a local bus (Read, Mux) (middle right and middle left cell in fig. 1) the arrow is drawn from the L-bus to the cell. It is only possible to select one connection from the bus to the cell. If the cell writes to the bus (Write, TS) (middle cell in fig. 1) the arrow is drawn from the cell to the L-bus and it is possible to have two connections simultaneously. If the cell implements one of the two cornerturns (lower left and upper left cell in fig. 1) two connections can be set, but no arrow is drawn because the bus is bi-directional.

It is possible to have the cell perform a cornerturn without choosing the cell's content. This can be seen in the lower left cell in fig. 1. The corresponding directions at the cell are chosen without first giving the cell a content.

Note: If two connections are made, these must be perpendicular to each other, e.g. north/west or south/west but not north/south.
Labels for A and B outputs: The last text selection can be copied to the lower A or B output by additionally pressing the middle button (left/middle interclick). This is analogous to Set Caret/Copy with normal texts. With left/right interclick the label at that position is cleared. If a label exists in a design already, an error message is printed to the Log.

Cell menu: When pressing the left button, a popup menu appears on the screen (figure 2). If a cell has a content already, these will be highlighted with a frame (see figure 2). The first four items in the lower row constitute the State and the upper row constitutes the Routing mode of a cell. The two Mux on the lower right are shortcuts for state Xor/Reg and routing Mux. If the left button is released outside the menu, no selection takes place.

By interclicking the middle button during the menu selection, the cell can be brought into its initial configuration (no routing/state/input/output).

The content of the cell is drawn depending on which routing and state is chosen and which inputs are active. E.g. the Xor and the Nand in the middle cell of figure 1 becomes a Not if input B is not active ($A \text{XOR} 1 = \neg A$).
Editing Pads

The pads, too, can be edited with the left mouse button. Also, a menu appears if the button is pressed inside a pad (figure 3). The tri-state function can then be chosen. The current state of the pad is highlighted with a frame. If the left button is released outside the menu, no selection takes place.

By interclicking the *middle button* during the menu selection, the pad can be brought into its initial configuration (0, no output).

1. *item (left in menu in fig. 3):* The pad acts as input only. "0" is drawn inside the pad box (output off). This is the most secure setup of a pad since no signals can leave the chip.

2. *item:* The pad is controlled by the L-bus running perpendicular to the pad’s side (the right, vertical (east) L-bus of the output cell in the figure). This state is indicated with a bar "|" which is perpendicular to the pad’s side.

3. *item:* The pad is controlled by the L-bus running parallel to the pad’s side. A dash "-" parallel to the pad’s side is shown in this state which is the normal case for data pads (D0..D7) on the left side of the chip on the CLI-Board.

4. *item (right):* The signal of the output cell drives the pad always which is indicated by a "1". Care must be taken with this state: Only the address lines (a0..a14) on the top of the chip should have this state. For all other pads this state should not be selected, as external logic or the chip itself could be destroyed.

Labels can be set at pads, too. The proceeding is analogous to the one with cells. The sensitive area for pads is at the input or output. Labels are drawn to the right or below a pad (cf. figure 3: sig).

Editing Repeaters

The state of repeaters can be chosen with a menu (figure 4). All possible connections between local and express busses are presented in the menu. The current state is highlighted with a frame. If the left button is released outside the menu, no selection takes place.

By interclicking the *middle button* during the menu selection, the repeater can be initialized (no connections).

The four last columns in the menu (from left to right) call for an explanation:

1: L-bus and E-bus are crossed over (E->L L->E)
2: L-bus writes to L- and E-bus or vice versa (L->E and L->L or E->E and E->L)
3: L-bus writes to L-bus in same direction and E-bus in the other or vice versa
   (L->L and E<=L or E->E and L<=E)
4: L-bus writes to E-bus in same and other direction or vice versa
   (L->E and E<=L or E->L and L<=E)
Selection

With the right mouse button, multiple cells can be selected. Certain commands require cells as arguments and these are specified by the last selection.

*Single cells* can be selected by simply pressing the *right button* inside the menu area. If *multiple cells* should be selected, the mouse should be brought to the lower left cell, the right button should be depressed and *held down*, and the mouse should then be dragged to the upper right cell one wishes to select. This way, a rectangular area can be selected even over repeater boundaries.

Shifting the View, Moving and Copying Cells

When the mouse is moved with the *middle button* held down, the visible portion of the design can be *shifted*. The location where the button is pressed down appears at the location where the button is released again.

If the middle and right buttons are depressed (*middle/right interclick*) the most recently selected cells are *copied* to the location where the buttons are released. Care should be taken in releasing the buttons only over a cell. If a rectangular area should be copied, the target cell must be the *lower left cell*. If the area cannot be copied as a whole, the operation is canceled.
If the middle and left buttons are depressed (*middle/left interclick*) the most recently selected cells are moved to the location where the buttons are released, i.e. they are deleted at the old location. Labels at outputs are moved as well. Attention: The editor has no Undo. Deleted cells are lost.

During these operations (moving or copying), repeaters are not moved or copied along. It is possible, though, to move or copy cells into another viewer. This way, designs from one viewer can be copied into another.

List of Commands

*Commands in the menu bar*

- **CL Locate**: cf. *Locate* below
- **CL Cells**: show all cells / only used cells
- **CL Load**: load design to the CLI-Board
- **CL Store**: store design to file using name in the menu bar

*Commands in CL Tool*

Many commands in **CL Tool** take the marked (*"*"*) viewer as an argument. If no viewer is marked, the one containing the most recent selection is used. With this, the star does not have to be setup every time, since a selection exists already. *Viewer* written in italics indicates if a command expects this kind of viewer.

- **CL Open Name**: open a viewer displaying the design in file *Name*
- **CL Open ***: the last text selection is taken as argument
- **CL Print Pluto col**: *Viewer* is printed on server Pluto, starting with column *col* (0 <= *col* <= 15).

Command *Locate* is used to find labels or cells by means of their names or coordinates.

- **CL Locate name**: *name* is searched for as a label in *Viewer* and the corresponding cell is selected
- **CL Locate xy**: the cell or pad at *xy* in *Viewer* is selected
- **CL Locate ***: the last text selection is taken as argument
- **CL Locate**: in the menu bar is the same as **CL Locate ***

*SetClock* and *SetReset* are used to set the value of the clock/reset signals in the selected column(s). It is important that the top most (31.) row is selected for setting the clock, and the bottom most (0.) row for setting the reset signal of a column. The command is applied to *all selected* columns. Note: The reset signal is active low (*"0"*).

- **CL Set Clock Aout**: set Clock of column(s) to A-Output of the top most cell in the column(s)
- **CL Set Clock Global**: set Clock of column(s) to global clock signal (Ceres-Clock)
- **CL Set Clock Express**: set Clock of column(s) to express bus south of top most row
- **CL Set Clock Off**: turn Clock of column(s) off
- **CL Set Clock ***: the last text selection is taken as argument
- **CL Set Reset Aout**: set Reset of column(s) to A-Output of the bottom most cell in the column(s)
- **CL Set Reset Global**: set Reset of column(s) to global reset signal
- **CL Set Reset Express**: set Reset of column(s) to express bus north of bottom most row
- **CL Set Reset Off**: turn Reset of column(s) off
- **CL Set Reset ***: the last text selection is taken as argument
CL Reset: all flip-flops in the chip are reset (global reset)

The following commands write information about a design to the Log:

- CLInspect: detailed information about the selection (cell/pad)
- CLClocks: clock assignments for all columns in Viewer
- CLResets: reset assignments for all columns in Viewer
- CLLabels: used labels in Viewer
- CLStatistics: number of used cells and busses in Viewer

Commands `Put` and `Get` are used to read and write the 4 CLI-ports of the CLI-Board. Procedures `PutInt` and `GetInt` are the programming interface to the board.

- `CLPut [port].val` write val to CLI-port port (0 default) (0 <= val <= 255, 0 <= port <= 3)
- `CLPut [port]` the last text selection is taken as argument
- `CLGet [port]` read a byte from CLI-port port (0 default) and write it to the Log (0 <= port <= 3)
- `CLGet [port]` the last text selection is taken as argument

- CLPutInt(port, val: INTEGER); Write val to CLI-port port (0 <= val <= 255, 0 <= port <= 3)
- CLGetInt(port: INTEGER): INTEGER; Read a byte from CLI-port port (0 <= port <= 3)

List of Mouse Button Combinations

- **left button**: menu for cell/pad/repeater
  (release outside menu to cancel)
  toggle A/B inputs
  toggle L-bus input/output

- **middle (interclick)** initialize cell/pad/repeater if inside menu
  copy text selection to label

- **right (interclick)** delete label

- **middle button**: shift view

- **left (interclick)** move cell selection (incl. labels)

- **right (interclick)** copy cell selection

- **right button**: select

- **all three buttons**: cancel started operation
| 177 | J. Burse | ProQuel: Using Prolog to Implement a Deductive Database System |
| 178 | P. Arbenz, M. Oettli | Block Implementations of the Symmetric QR and Jacobi Algorithms |
| 179 | B. Hösli | 3-wertige Logiken und stabile Logik |
| 180 | K. Zuse | Computerarchitektur aus damaliger und heutiger Sicht |
| 181 | B. Sanders | A Predicate Transformer Approach to Knowledge and Knowledge-based Protocols |
| 182 | O. Lorenz | Retrieval von Grafikdokumenten |
| 184 | L. Knecht, G.H. Gonnet | Alignment of Nucleotide with Peptide Sequences |
| 185 | T. Roos, P. Widmayer | Computing the Minimum of the k-Level of an Arrangement with Applications |
| 186 | E. Margulis | Using nP-based Analysis in Information Retrieval |
| 187 | D. Gruntz | Limit Computation in Computer Algebra |
| 188 | S. Mentzer | Analyse von Methoden und Werkzeugen zur Entwicklung grosser Datenbank-Anwendungs-Systeme |
| 189 | S.J. Leon | Maximizing Bilinear Forms Subject to Linear Constraints |
| 191 | M. Böhlen, R. Marti | A Temporal Extension of the Deductive Database System ProQuel |
| 192 | R.H. Güting | Second-order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization |
| 194 | H.P. Frei, D. Stieger | A Semantic Link Model for Hypertext Retrieval |
| 196 | R. Gross, R. Marti | Intensional Answers in Generalized Deductive Databases |
| 197 | K.M. Chandi, B.A. Sanders | Conjunctive Predicate Transformers for Reasoning about Concurrent Computation |