Contents

List of Figures

1

Introduction

CADIC is a hierarchical top–down design system for VLSI circuits. It has been developed at the Lehrstuhl of Prof. Dr. Günter Hotz at the Department of Computer Science, University of Saarland, Saarbrücken, Germany. The development of CADIC is supported by the Deutsche Forschungsgemeinschaft (DFG) within the Sonderforschungsbereich 124 *"VLSI–Entwurfsmethoden und Parallelität"* since 1983.

1.1 Features of the CADIC System

In comparison to other VLSI design systems, the CADIC system possesses the following distinctive features :

Parameterization

The design level of CADIC especially supports the specification of whole families of circuits which can depend on a set of parameters. For example these parameters can be used to have operands of flexible bit length (the user can design an n bit adder) or generic descriptions of circuits (a sorting network for arbitrary types of elements).

The designer can use sub–circuits which depend on arithmetical expressions over the set of circuit parameters. As well as the sub–circuits, the user may parameterize the width of a bundle of wires (equivalent to busses). With the help of parameterization multiple descriptions of the same circuit with

different sizes can be avoided. The use of parameterized sub–circuits and wires makes it possible to define a class of even very large circuits by only a few schematic inputs.

Recursive Specification

Recursion is one of the most impressive properties of CADIC . This feature admits recursive design operations in a topological framework and makes it possible for a hardware designer to easily specify regular circuits in a very compact and easy way. Like the recursive definition of functions in programming languages, references on the calling sub–circuit itself are allowed. The base of a recursive description are sub–circuits with fixed values for the parameters.

As well as the sub–circuits the designer has to parameterize the width of bundles of wires within a recursive specification. For the situation that it is too difficult to derive an arithmetical expression for a bundle of wires which holds for all stages of the recursion, CADIC offers the designer a very easy and comprehensive way of parameterizing wires. It allows the designer to use formal variables, denoted as wire types, to describe the width of a bus. It is possible to use an expression containing both indicated variables and arithmetical expressions over the set of circuit parameters. Within the recursive specification the indices of these wire types can be arithmetical expressions over the set of circuit parameters.

It is very significant that the recursive description together with the use of wire variables enables a user to setup generic specifications of algorithmic structures as for example sorting networks or parallel prefix computation parts ([Bur94]). These circuits for example can be specified independent from the realization of the basic operation. Later they can be configured according to special needs. The basic operation of a parallel prefix computation circuit can be chosen in order to construct a carry look ahead adder or a leading zero counter etc. without changing any other input of the parallel prefix network.

With the help of this description method the designer can setup his own library of reusable circuit structures. By this CADIC makes the graphical design methods become as strong and flexible as the text design methods.

Hierarchical Representation

An important problem during the design of large circuits is to control the amount of generated data. In CADIC we take advantage from the fact that large circuits often contain very regular structures. This can be used to setup a very compact representation in the form of an internal hierarchical data structure which is given by a directed acyclic graph (DAG). The DAG is a folded tree structure which is constructed in a top–down process over all sub–circuits of the design. Its root node is the given circuit C. All sub–circuits of C are internal nodes of the tree, in which same sub–circuits appear only once. The leaves of the tree are the basic cells or sub–circuits which only contain wiring structures. To construct this data structure for the required circuit, its parameters, if it depends on those, must be set to fixed values. This means that from the whole family of circuits given by a parameterized description we will now select one special representative (construct a 32 bit adder out of the n bit adder family).

After the hierarchical data structure is built, the system calculates all arithmetic expressions in the parameterized description, including the indices of the formal variables. If the designer has used wire variables to denote bundles of wires, the system will now derive a set of equations from the hierarchical representation in order to compute a legal solution for the variables. If this last step is successful, the created hierarchical data structure serves as the basic structure for the integrated design tools.

Integrated Design Environment

In order to make the integration of newly developed design tools possible, the graphical design environment of CADIC is implemented like a workbench. The core of the system is a graphical user interface which is basic for all design steps. For special needs of certain design tools it can be

enlarged by tool specific graphical components.

The modular implementation of CADIC supports the configurability of the whole system. Each tool is given by its own object code libraries, which can be linked into the graphical surface. Prerequisite is that each tool creates a relation between its internal data structures and the central hierarchical data structure of CADIC or that it directly works on it. Beside this a design tool may also be a stand alone program. In this case it can be connected to the graphical surface via interprocess communication.

Based on this open hierarchical data structure, CADIC has formed an integrated design environment which manages the different steps of the design process. The system CADIC 3 0 includes integrated tools for

- logic simulation
- timing analysis
- layer assignment
- power supply
- place and route
- net–list converters

The hierarchical data structure of CADIC provides a basis, on which new tools can be developed and easily integrated.

Visualization and Navigation

The integrated design tools of CADIC can directly display the calculated results and design data within the graphical user input, such that a relation between the design data and the specification is created.

The visualization of the data in combination with the navigation through the circuit hierarchy enables the user to find critical parts of a specification, where the design tools compute bad results. For example you can easily locate such regions of a design, where a logic simulator indicates wrong results in the specification.

In order to enable the visualization of data, the types of design data are specified by a simple description language, from which a parser generates the necessary library of graphical functions, that is linked into the surface of CADIC . If a tool is directly integrated in the graphical environment it can visualize its design data by calling the functions from this generated library. If a tool is not integrated into the surface and works independently, the data can be transferred to the surface of CADIC by interprocess communication (IPC).

Compared with pure text–form representation, there are especially possibilities to graphically display data using timing diagrams, point diagrams, line diagrams or histograms, etc.

Conversion Interfaces

There exist a lot of conversion interfaces in CADIC to export a circuit specification in the net–list formats of commercial systems. Especially this means that the designer cat setup parameterized circuit specifications and extract special representatives in a standard exchange format. CADIC 3 0 can generate the following different net–list formats:

- Electronic Design Interchange Format (EDIF 2 0 0)
- Desgin Exchange Format (DEF) for TANGATE Placement and Routing System from CADENCE
- Genrad Hardware Description Language (GHDL) for HILO Simulation Tool from GENRAD
- StrukturBeschreibungsSprache (SBS) for Venus Design System from SIEMENS
- Xilinx Netlist Format (XNF) for XACT FPGA Development System of Xilinx

Thereby, the CADIC system could be used as graphical front end for many commercial VLSI design systems. By selecting a certain library of basic

cells the hardware designer can choose any required technology to realize his design.

1.2 The Structure of CADIC

The integration of the given features into a common graphical environment is shown in figure 1.1.

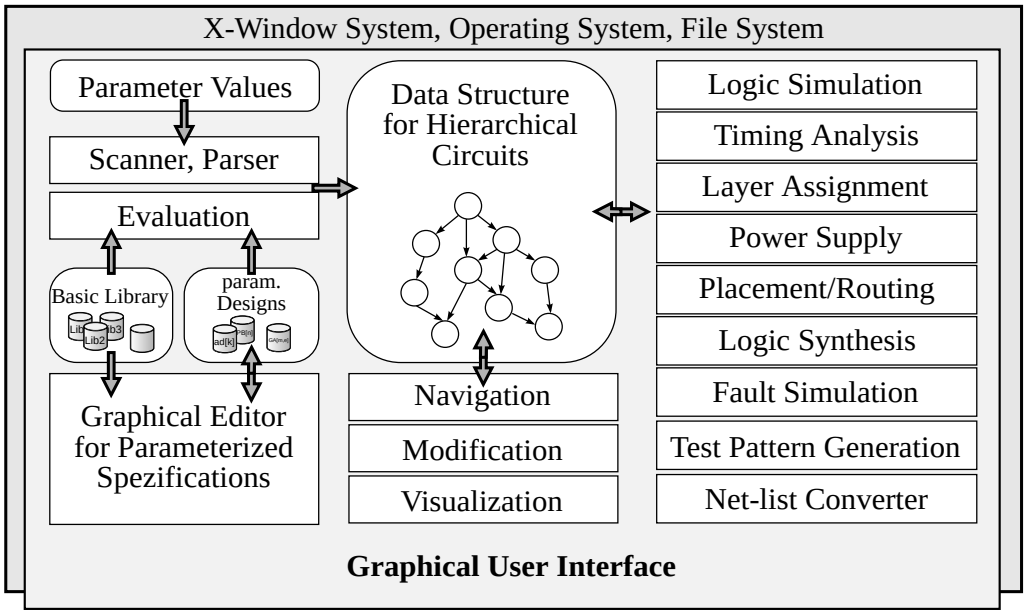


Figure 1.1: Integrated Design Environment of CADIC

With the help of the graphical editor the user can setup parameterized circuit designs over an arbitrary library of basic cells. From such a parameterized description the designer can select a specific element, i.e. a fixed circuit, by giving explicit values for its parameters. According to these parameter values the system will build a hierarchical data structure which is the base for all integrated design tools. This data structure is used to visualize the desgin results. With the help of navigation functions the hierarchical strucutre can be traversed in every direction in order to completely examine the design data.

1.3 The Structure of this Book

The structure of this book is orientated at the structure of the CADIC system as it is shown in figure 1.1.

In chapter 2 we will describe how the system can be installed on your hard disk.

In chapter 3 the main components of the graphical user interface and the setup of CADIC 's environment are explained.

In chapter 4 we will show, how you can setup a parameterized specification with the help of the graphical editor of CADIC . An adder for n bit operands is specified in order to introduce you to the work with the graphical editor.

In chapter 5 CADIC 's hierarchical data structure is explained. It is shown, how the system extracts one element of a parameterized circuit specification and builds up its basic hierarchical representation in the machine's memory.

In chapter 6 the logic simulation tool is introduced. This tools provides a simple method for checking your design on the base of the graphical specification, i.e. you can control the results of a simulation step graphically at the inputs and outputs of each subcircuit.

In chapters 7 to 9 we describe the integrated tools for layer assignment, for generation of the power supply nets and for calculation of a geometrical layout.

Chapter 10 gives you an impression of the parameterized design mehtod of CADIC . With the help of three examples we will demonstrate the power of this description method by giving short and handy specifications for even large circuits.

In chapter 11 CADIC 's interfaces to commercial systems and standard exchange formats are shown. With the help of these interfaces you can exchange a CADIC design with other design systems for further processing.

Chapter 12 contains a reference manual for the graphical editor. The basic syntactical structures for parameterized designs are explained as well as

the handling of the different editor functions. This chapter in addition to chapter 4 and chapter 10 enables you to setup parameterized descriptions yourself.

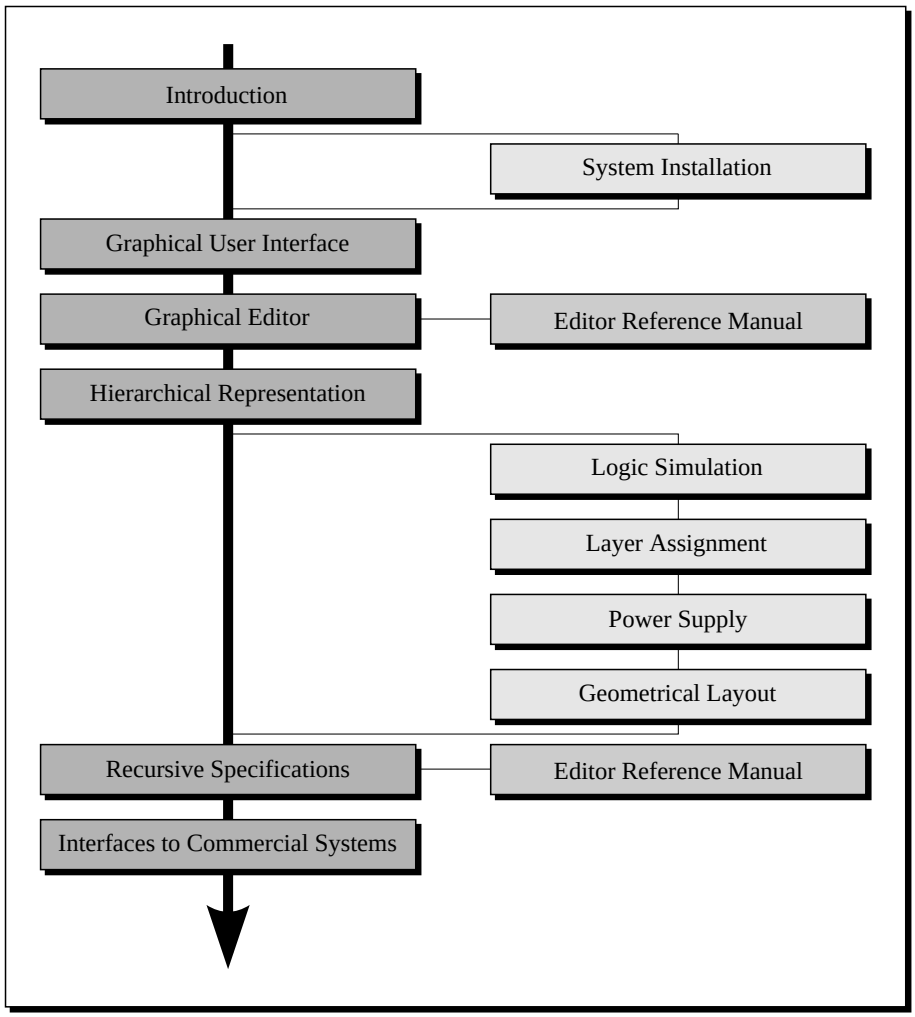


Figure 1.2: Readers guide for the CADIC manual

Figure 1.2 gives you a guide through the chapters of this book. For the understanding of the principles of CADIC you should read the main chapters (dark grey boxes) at least. The chapters about the system installation and the integrated design tools can be read later, when you want to perform the corresponding design task for a given specification. During the reading of the chapters about the graphical editor and the recursive specifications you can directly work with the system in order to setup these specifications

youself. In that case it may be helpful to have a look at the editor refrence manual.

1.4 Notations

In this book we use the following typographical conventions:

The *italic font* is used to introduce important new terminologies defined in the book which are used in following paragraphs.

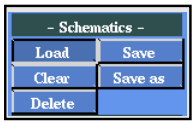
The `typewriter font` is used to indicate system messages that appear on your screen. It also denotes user input to CADIC as well as system files, directories and environment variables.

Beside different fonts we use the following signs to give you some hints within the text:

This sign is used to mark important parts in the text. For example some requirements, without which the system would not work, are marked in this way.

⇝ 3,5.1

This sign gives you a reference to other paragraphs in the book, where you can find more information about the current subjects.



The extracted pictures from the menuline inserted in the text help you to identify the corresponding entries in the system menus and illustrate which button is currently to be selected. Within the text it is described how the corresponding function can be invoked and terminated.

2

System Installation

2.1 Hardware Requirements

For the installation of the CADIC system you need one of the following hardware configurations:

- Sun Sparc Station 2, 10 or 20 with SunOS 4. 1. x
- PC486/586 with Linux 1.x
- HP Apollo Series 700

The system has been developed and tested on the first configuration. On the two other configurations it has been compiled and installed successfully, but not all components have been tested. There could occur some troubles which might be caused by specific sources of the operating system.

You should have at least 90MB of free disk space and 8MB of memory. For best use of the system your workstation or PC should have a color screen with a resolution of at least 1024×768 pixels. This is an important aspect, if you plan to install the system on a linux PC, because your VGA graphics card may not support this resolution.

If you get a copy of the executable CADIC system, you can skip the following software requirements. In this case you only have to perform the unpacking steps in section 2.3. After that you can start the system by calling one of the example scripts in the directory `Examples`.

2.2 Software Requirements

In order to compile the sources of the system you need a C– and C++– Compiler as well as a scanner and parser generator. They have been successfully compiled with one of the following configurations:

- SunOS `cc` and At&T CC with `yacc` and `lex`
- `gcc` and `g++` with `bison` and `flex`

If you use the GNU software the version of the compilers should be 2.5.8 or newer.

For the graphical surface of the system you need an installation of the X–Window System X11R5/R6.

Make sure that the compiler commands `cc` (`gcc`) and `CC` (`g++`) are in your command path as well as `lex` (`flex`) and `yacc` (`bison`). Further you need access to the X-Window libraries and header files.

2.3 Installation Procedure

If you have the required hardware and software environment you can begin with the installation procedure. You can install the system anywhere in the directory tree, i.e. you need not to have root permissions to perform the installation. If you want to take a first glance at the CADIC system we recommend that you install it within a subdirectory of your home directory. You can do this with the following commands:

```
cd $HOME

mkdir Cadic

cd Cadic
```

In the following we call the newly created subdirectory `Cadic` the top installation directory or top directory for short. Now you copy the CADIC archive file into the top directory and unpack the sources. Here we assume

that you have got the sources via magnetic tape. The commands to copy it to the hard disk are:

```
mt -f /dev/nrst0 rewind
```

```
dd if=/dev/nrst0 of=./cadic.tgz
```

You also may get the archive file `cadic.tgz` directly via anonymous ftp from `hamster.cs.uni-sb.de`.

Now you can unpack the archive file with the following command sequence:

```
cat cadic.tgz | gunzip | tar xvf -
```

After that you can remove the original archive file in order to save disk space:

```
rm -f cadic.tgz
```

After unpacking the archive file you will see the following list of directories and files in the top directory:

```
Data          Imakefile  Libs     System       cadic.tmpl
Examples      Includes   Src      Tools
INSTALLATION  Leda       Surface  cadic.rules
```

`Data`: This directory contains subdirectories for global and temporal data of the system. The descriptions of the basic cell libraries are located here as well as some example designs.

`Examples`: This directory contains some example shell scripts which setup the environment variables of CADIC and start the system. You can call these scripts after the installation is completed.

`INSTALLATION`: This file contains a short description of the installation commands. If you are really impatient and familiar with installing software on a unix machine, this file shows to you the steps to perform the installation.

`Includes`: This directory subtree contains the header files of the system.

`Leda`: This directory subtree contains version 3.0 of the LEDA [MN89] library. CADIC depends on this special version. Because you might not

have an installation of the LEDA library, it is included in the CADIC distribution. It will be compiled during the installation, but it will not replace any previous installation of LEDA.

`Libs`: Initially this directory is empty. It will contain the object code libraries of CADIC and LEDA which are created during the compilation of the system sources. If you plan to write programs using the basic data structures of CADIC , you will find the corresponding libraries in this directory and have to set the linker options appropriately.

`Src`: This directory subtree contains the sources of CADIC , divided into several libraries which contain the functions of the basic data structure and the integrated design tools.

`Surface`: This directory contains the description of the graphical surface of CADIC . For each design tool there exists a file which configures its menu entries.

`System`: This directory contains the main part of the CADIC system. After the installation is completed it will contain the executable program which is called `cadic`. You can call it directly from this directory, but it is more convenient to use a shell script from the `Examples` subdirectory which sets up a correct environment for the system, before it is started.

`Tools`: This directory contains some tools which are needed for the installation. The main tool is a compiler which translates the surface descriptions from the `Surface` subdirectory into a C module. This module serves for the menu display, selection and activation of the corresponding functions. It is included into the main part of the surface.

`cadic.rules`: This file contains some macros which are used by `imake` to create the Makefiles.

`cadic.tmpl`: This file contains a templative description of the installation environment. You have to edit this file in order to set up the compiler commands etc. as you will see in the following.

Before you can start the compilation of the sources you have to edit the

macros in `cadic.tmpl`. The macro `CC` is set to the C compiler command. In the default distribution it is set to `gcc`, change it, if you want to use the traditional SunOS C compiler. The next macro `CPLUSPLUS` points to the C++ compiler, it is set to `g++`. In the same way change the macros `YACC` and `LEX`, which represent the commands for the parser and scanner generator.

Finally take a look at the macro `CDEBUGFLAGS`. This defaults to `-g` in order to create a debugging version of the system. If you do not want this, just erase any text behind the equal sign in this line and set `CDEBUGFLAGS` to the empty string. This results in much smaller object files and executables and is recommended, if you have less free disk space.

Now start the installation from the top directory with the command:

```
xmkmf
```

Note that you need an installation of the X–Window system to use this command. The directory `/usr/bin/X11` must be in your command path. After that, you create the makefiles for CADIC with the command:

```
make Makefiles
```

After this command is completed, you can start the compilation of the source files and the graphical surface simply with the command:

```
make >& cadic.log &
```

Depending on your hardware platform this command will take several hours to complete. If there are any error messages, you can read them on the file `cadic.log`.

If it has finished without error, you can find the executable CADIC system in the subdirectory `System`. It is called `cadic`. Before you can start the program, you have to setup an environment, i.e. you have to define a list of environment variables. In the subdirectory `Examples` you can find some sample shell scripts which set these variables for some example designs and then start the system.

3

The Graphical User Interface

3.1 Environment Setup

Before you can start the CADIC system you have to set some environment variables in order to tell the system where its temporal and global data files are located. For example you have to specify a basic cell library which you want to use during a design. You can find some sample environment setups in the shell scripts within the directory `Examples`, a subdirectory of the top installation directory. The contents of the sample startup file `arith_circuits` is the following. As the name indicates this file sets the environment for some sample designs of arithmetical circuits (conditional sum adder as it is shown in chapter 4 and Wallace tree multiplier).

```
#!/bin/csh

setenv DAGDIR      ../Data/Designs/Arithmetic
setenv CELLDIR     ../Data/Cells/Ims/Right
setenv TECHDIR     ../Data/Cells/Ims/Power/Tech
setenv SLICEDIR    ../Data/Slices
setenv SIMDIR      ../Data/Sim
setenv MACROLIB    ../Data/Designs/Arithmetic
setenv PARLIB      ../Data/Designs/Arithmetic
setenv PWRDIR      ../Data/Power
setenv HILODIR     ../Data/Formats
```

```
setenv NBSDIR      ../Data/Formats
setenv CADENCEDIR  ../Data/Formats
setenv VENUSDIR    ../Data/Formats
setenv EDIFDIR     ../Data/Formats

../System/cadic
```

You can create yourself such a shell script for a special design. Another method to setup the environment variables is to add the needed commands to your `.cshrc` which is executed, when you start a new shell.

For the correct setup of the system environment you have to give values to the following variables:

- `CELLDIR`: This variable points to the library of basic cells, you want to use during the design. The cell libraries in the current CADIC distribution are contained in the subdirectories `Data/Cells` of the top installation directory. A description of the available libraries is given in chapter 11. Note that you cannot use the system, if `CELLDIR` has no legal value.

- `DAGDIR`: This variable points to the directory, which contains the schematic inputs of your designs. Initially this directory may be empty. Note that you must set this variable to a legal value, i.e. the given directory must exist. If this is not yet the case, you must create it with the command

  `mkdir -p <`*directoryname*`>`

- `TECHDIR`: This variable points to the subdirectory which contains technological data about the cells in the basic cell library. Normally it is just a subdirectory of `CELLDIR` which is called `Tech`. If you only want to enter a design, it is not necessary to set this variable. In the case that you do not set it, you cannot use the tools for power supply and place&route. If you have already set the variable `CELLDIR`, you can simply use the command

```
setenv TECHDIR $CELLDIR/Tech
```

- SLICEDIR: This variable points to a directory, which will contain data files that are temporarily generated during the design process. If you do not set this variable to a legal value, i.e. an existing directory, you cannot use the tools for power supply and place&route.
- PWRDIR: Just like SLICEDIR this variable points to a directory for temporarily generated data. If you do not set this variable to a legal value (existing directory) you cannot use the tool for power supply.
- SIMDIR: The directory given by this variable contains data which is generated by the tool for logic simulation. If you want to use the simulator, you must set this variable to a legal value.
- MACROLIB, PARLIB: These two files contain information about created macros and parameterized macro cells. They are used by the schematic editor. It is recommended to set these variables to legal file names.
- HILODIR, NBSDIR, CADENCEDIR, VENUSDIR, EDIFDIR: These variables can be set to directories for files, which are created by the netlist converter. They contain specific exchange formats which can be used as input for commercial design systems. In order to distinguish between the different formats, you can set each of these variable to its own value.

If you do not like to distinguish between the different directories for the temporarily generated data, you can set all these variables (SLICEDIR, PWRDIR, SIMDIR, HILODIR, NBSDIR, CADENCEDIR, VENUSDIR, EDIFDIR) to the same value, e.g. to /tmp or /usr/tmp.

3.2 The Components of the Graphical User Interface

If you startup the CADIC system, the basic graphical user interface is displayed on your screen as it is shown in figure 3.1.

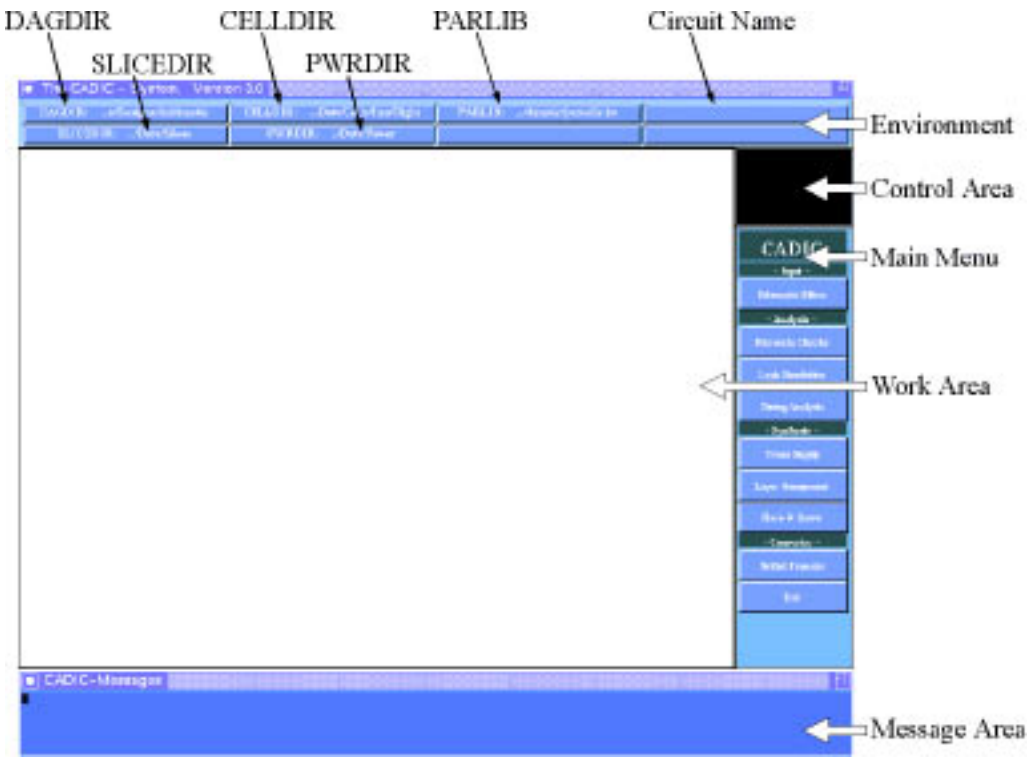


Figure 3.1: Basic structure of the graphical user interface

The structure of the user interface is basic to all integrated design tools. But each tool can add its own components, as dialog windows or information displays. The basic graphical user interface can be divided into the following parts:

- *Work area*: This is the main part of the user interface. It is used to display the currently loaded circuit. The display mode depends on the active tool. During an editor session you will see the netgraph of the circuit which can be interactively modified, as it is described in chapter 4. This mode is also used for other design tools, for example during logic simulation. Here the simulation results are displayed directly on the graphical input of the circuit. Another display mode is used, when the place&route tool is active. In this case the wires have physical widths, and they will be displayed as polygons as it is described in chapter 9.

- *Environment*: The environment area is divided into eight panels. Here you can see the values of the environment variables of CADIC . The library of basic cells currently used is displayed as well as the directory where your design files are located. The right panel in the first row which is initially empty, will contain the name of the currently loaded circuit.

- *Control area*: If you change the current view of a loaded circuit by zooming or scrolling in the work area, you can see in this window the relation between the currently visible part of the circuit and its total area.

- *Message area*: Control and error messages of CADIC are displayed in this area. Every integrated design tool prints information messages while it is performing its task. If you are using a window manager (`twm`, `mwm`, `fvwm`, ...), you will see that this window is decorated as a usual shell window. This enables you to change its size, if the default height of five text lines is not sufficient. You can also add a scroll bar to the window, if you want to take a look at some previously displayed messages. For information about how to change the size of a window or how to add a scroll bar, you should look at the manual of your window manager.

- *Main menu*: The menu line of CADIC is organized hierarchically. At the topmost level you will see the list of integrated design tools which are grouped according to their specific tasks (e.g. there are some tools for analysis and others for synthesis of the currently loaded circuit). Each tool has its own menu, sometimes including several sub menus. You activate a certain menu or function by pressing the left mouse button, when the appropriate push button is highlighted. To leave an activated menu or sub menu you simply must press the right mouse button within the menu line, when you are in menu selection mode.

4

The Graphical Editor

4.1 Introduction

Today the specification of integrated circuits is based on two different methods: procedural or graphical. In the first case the specification is given in a hardware description language (HDL). In most cases these languages are derived from imperative or functional programming languages. They can be classified according to the represented properties of the circuit:

- languages that only describe the functional behaviour of the circuit without considering any structural properties, e. g. ISPS ([Bar78]) or Macpitts ([SSC82]).
- languages that describe both functional and structural properties, e. g. Zeus ([GL85]), Hades ([Wir82]), VHDL ([LSU89], [Sha82]) or EDIF ([Com87]).
- languages that only describe the structure of the circuit, e. g. HISDL ([Lim82]).

Common to all these languages is a great flexibility, because they allow the full power and generality of a programming language. Procedures are written to describe each part of the circuit and are combined hierarchically using the methods defined for programming languages. Parameters may be passed to procedures in order to generate different versions of the same structure.

But the procedural approach suffers from two drawbacks. First it is hard to embed graphical information in a textual procedure, in other words topological information is difficult to visualize when specified textually. Second the results of even small changes in the input cannot be seen without recompiling the whole circuit which can be a time–consuming process. Thus procedural systems tend not to be very interactive.

Systems which base on the graphical specification method offer a graphical editor to the user. This editor is used to specify structural information of a circuit, where subcircuits can be placed on the input screen and can be connected with lines. Some editors also allow the specification of behavioural aspects of a circuit. For instance in the system Gdl ([DBR+88]) the user can draw petri nets in order to specify control and data flow of a circuit. In most cases however the graphical input is translated into interchange formats such that the knowledge of the designer given by the topological information will be lost.

Such editors normally build the frontends of commercial design systems. Although they support hierarchical specification they cannot be regarded as easy to use input tools. An important reason for this fact is that these commercial editors do not support the extraction of regular parts of circuits in order to set up parameterized and recursive circuit descriptions.

An exception to the rule is the layout system ESCHER ([CF86]) which allows parameterized circuit specifications. In this system the user can group cells together which are somewhat like one dimensional arrays in programming languages. The disadvantages of ESCHER follow from the fact that it is not based on a well defined mathematical calculus which leads to ambiguities in the interconnection of the above groups. Moreover ESCHER only supports the use of only one single parameter. This is an ugly constraint because a lot of interesting circuits can be easily described using more than one parameter. A well known example is the Wallace tree multiplier ([Wal64]) where a short recursive description with two parameters is given in [LV83]. A parameterized specification of this multiplier with the help of CADIC 's graphical editor will be introduced in section 10.2.

⇝ 10.2

4.2 Basic Concepts of CADIC

Because both approaches have their disadvantages we will now introduce a specification method which powerfully combines the flexibility of a programming language with an intuitive graphical representation. For this purpose the design level of CADIC is based on a mathematical calculus, which has been developed within the project B1 of the SFB124 and which serves as a programming language for the specification of integrated circuits. The properties of this calculus, which has originally been introduced in [Hot65] and [Hot74], are explicitly described in [Mol86]. In the following we will summarize the main aspects in order to understand the design level of CADIC .

The importance of boolean algebra in the field of the design of logic circuits is well known. It is the base of a method, which allows us to describe the logic of circuits and to exploit these objects for further calculations. This has been sufficient as long as the arrangement of the elementary gates and the connections between these did not play an important role. Now that technology has been improved and the integration of millions of transistors on a single chip can be achieved, the geometrical layout of the wires and the gates can no longer be neglected. In the design of very large circuits the geometrical layout directly influences the function of the circuit. From this point of view it is desirable to have a calculus, which combines the functional and geometrical aspects of a design.

In order to get an easy to handle method there will be taken some abstractions in a circuit description. Each circuit will be laid out into a rectangle, at the border of which its external connectors are placed. Within the rectangle there are components which are connected by wires. The whole layout is projected into the Euclidean plane, such that there are no different layers first. In order to receive a planar embedding, we consider branches and crossings of wires as basic components. Further we neglect the physical width of wires and look at them as sequences of line segments. Each component as well as the surrounding rectangle has a northern, eastern, southern and western border. It also has a name, and sub–circuits with the same

names must have the same number and sequence of external connectors at each border. This abstract method of describing a circuit is called *logic topographical net*.

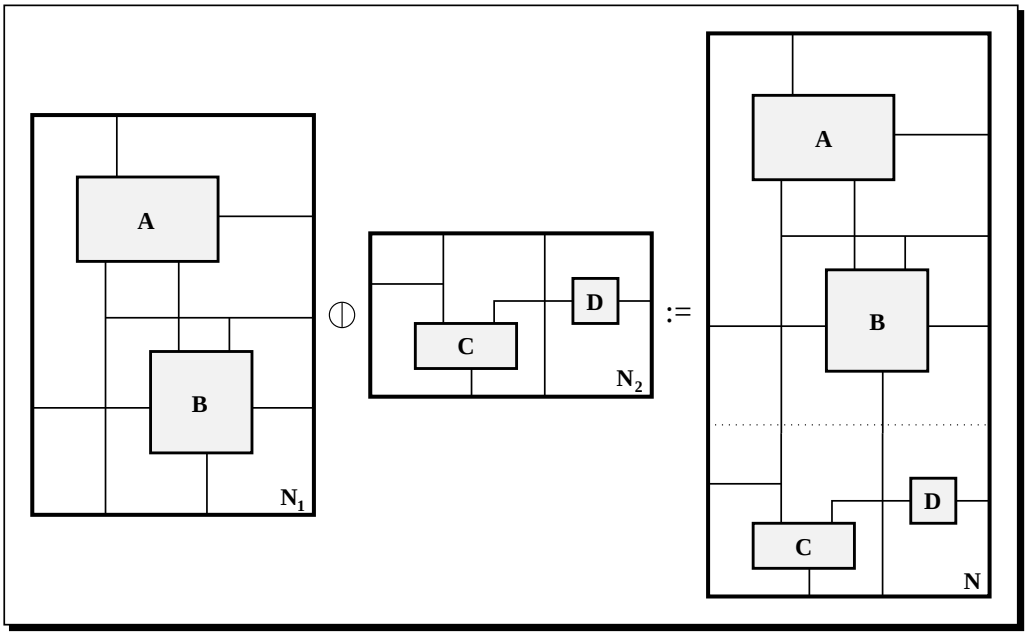


Figure 4.1: Vertical composition of two logic topographical nets

In order to construct larger nets we define two simple operations on logic topographical nets. Let N_1 and N_2 be two topographical nets. Then we define the vertical composition $N := N_1 \oplus N_2$, if and only if the southern border of N_1 and the northern border of N_2 fit together (c.f. figure 4.1).

In the same way we can define the horizontal composition $N := N_1 \ominus N_2$, if and only if the eastern border of N_1 and the western border of N_2 fit together (c.f. figure 4.2).

The resulting net N is given by the two parts N_1 and N_2, which are connected at their borders, i.e. the external connectors at these borders are identified and finally they are erased.

With the help of these operations we could create large logic topographical nets over the set of basic cells, branches, crossings and wire segments.

A disadvantage of this method is that we need an exact topographical de-

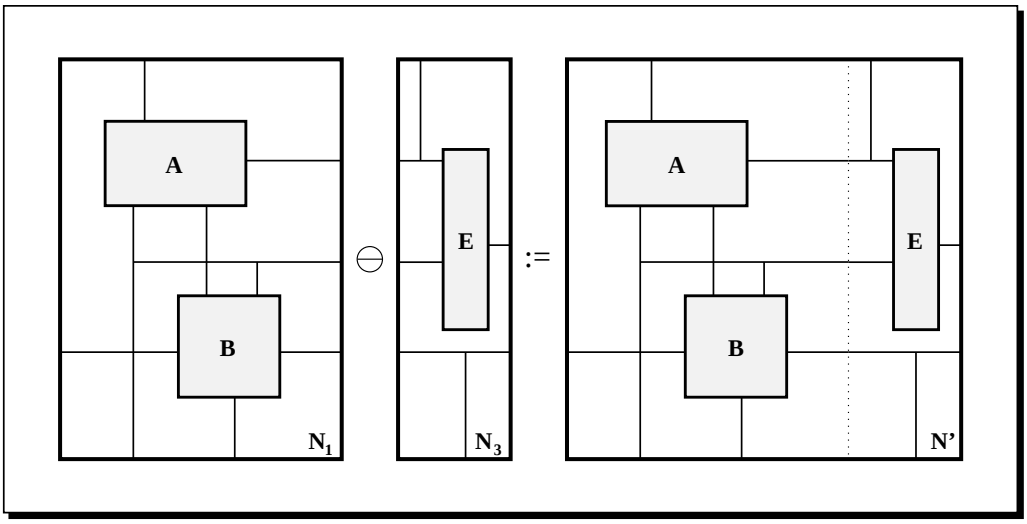


Figure 4.2: Horizontal composition of two logic topographical nets

scription of the resulting net. To overcome this restriction we introduce an equivalence relation between topographical nets. By that relation we consider two topographical nets to be equivalent, if they can be transformed into each another by a sequence of elementary deformations. Legal deformation steps are deformations of wires, moving, rotating and scaling of cells as well as repositioning external connectors at the borders of cells. A deformation is illegal, if for example the sequence of the connectors is changed of if additional crossings of wires are created. All resulting nets during a sequence of deformations must be legal topographical nets.

According to this set of elementary deformations the two logic topographical nets in figure 4.3 are equivalent. With this equivalence relation we obtain classes of logic topographical nets. Each class is called *logic topological net* and an element of a class is called a representative of a logic topological net. The vertical and horizontal composition can be applied to logic topological nets, if there exist two topographical representatives such that the operation is defined as shown above, i.e. if there exist two logic topographical nets, where the connectors can me moved to appropriate positions, such that the borders can be glued together.

In most cases the designer not only has to look at these numbers of con-

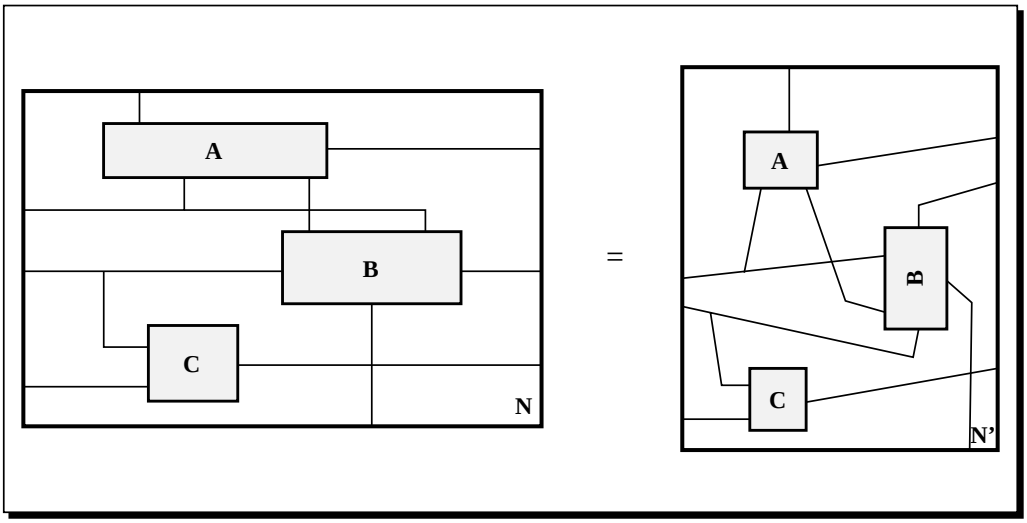


Figure 4.3: Two representatives of a logic topological net

nectors, but the connectors may have a specific type, such as clock signal or power supply signals. This fact is represented in the calculus by defining the composition operations, if the sequences of the signal types, read from left to right or top to bottom respectively, at the corresponding borders are identical.

The underlying mathematical calculus is called the *bicategory* of logic topological nets ([Mol86]). The description of a circuit within this calculus is done by *bicategorial expressions* which consist of the operations $\ominus$ and $\odot$ composing basic cells and wiring elements. The following is an example for a correct bicategorial expression, which we will call *net equation*, because it defines a *net variable* by the corresponding expression.

$$HalfAdder = (\vdash \ominus \urcorner \ominus 1_1^{\mathfrak{o}}) \odot (1_1^{\mathfrak{o}} \ominus \ulcorner \ominus + \ominus \dashv) \odot (AND \ominus EXOR)$$

In this example the symbol $\vdash$ stands for an east branch of a wire, $\urcorner$ denotes a south–west knee and *EXOR* is an element from the library of basic cells. The notation $1_1^{\mathfrak{o}}$ represents a vertival wire segment of width 1 which is the unit for the vertical composition.

This small example already indicates that bicategorial expressions can become very large and difficult to handle. To solve this problem we first

introduce the hierarchy concept for bicategorial expressions.

In the sense of bicategories hierarchy means to transform a net over a set of cells A into a description over a set of cells B. This transformation is done by a homomorphic function between bicategories which is called *bifunctor*. Let for example

$$HalfAdder = (\vdash \ominus \urcorner \ominus 1_1^{\Phi}) \oplus (1_1^{\Phi} \ominus \ulcorner \ominus + \ominus \dashv) \oplus (AND \ominus EXOR)$$

$$FullAdder = (HalfAdder \ominus 1_1^{\Phi}) \oplus (1_1^{\Phi} \ominus HalfAdder) \oplus (OR \ominus 1_1^{\Phi})$$

be two bicategorial expressions. *HalfAdder* is described over the set of basic wiring elements plus the basic gates *AND* and *EXOR*. *FullAdder* also uses basic wiring elements, but beside the basic gate *OR* it contains the cell *HalfAdder* which itself is given by a bicategorial expression. In order to get a description of *FullAdder* over the set of basic gates, we have to substitute each occurrence of *HalfAdder* by the corresponding bicategorial expression. This substitution step is also called *expansion* (applying the bifunctor) because we expand the macro *HalfAdder* by its representation. The following figure 4.4 illustrates the expansion step.

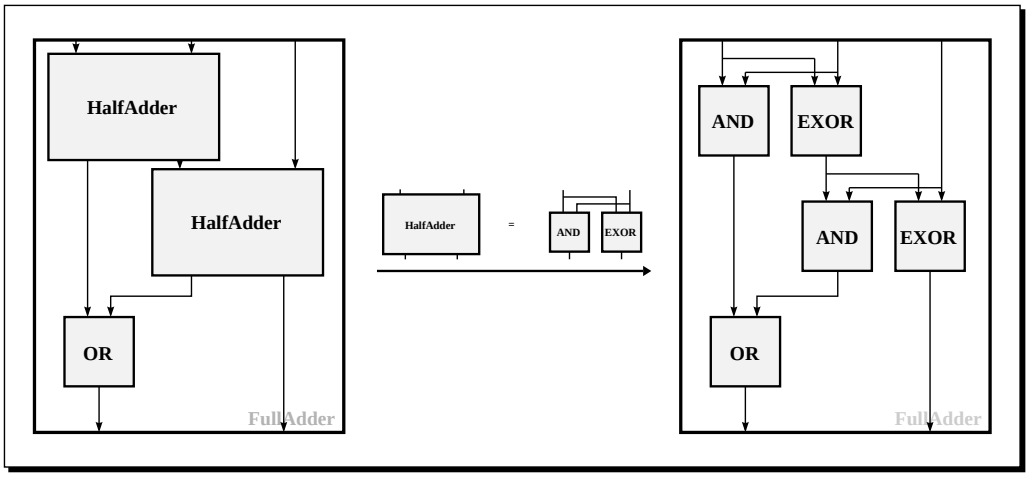


Figure 4.4: Refinement operator for the example of a *FullAdder*

With this method of discrete hierarchy we still have problems in describing very large circuits. Moreover we can only specify fixed realizations of a circuit. In order to get a more flexible specification level we now allow parameterizable net variables and bicategorial expressions. On the one hand this gives us the possibility to describe whole families of circuits with one fixed set of net equations. On the other hand we can setup recursive specifications which lead to very short and handy descriptions for regular circuits, such as adders, multipliers, sorting networks or memory, etc.

The following example demonstrates the use of parameterized net variables:

$$Tree[0] = EXOR$$

$$Tree[i] = (Tree[i-1] \ominus Tree[i-1]) \oplus OR$$

This recursive system of net equations has a free parameter i which can be set to an arbitrary nonnegative integer value. If for example we set $i = 3$, then we can apply the expansion functor from above, until we get a bicategorial expression which only contains basic gates as active components. This will take the following steps:

$$\begin{aligned}
Tree[3] &= (Tree[2] \ominus Tree[2]) \oplus OR \\
&= ((Tree[1] \ominus Tree[1]) \oplus OR \ominus (Tree[1] \ominus Tree[1]) \oplus OR) \oplus OR \\
&= (((Tree[0] \ominus Tree[0]) \oplus OR \ominus (Tree[0] \ominus Tree[0]) \oplus OR) \oplus OR \ominus \\
&\quad ((Tree[0] \ominus Tree[0]) \oplus OR \ominus (Tree[0] \ominus Tree[0]) \oplus OR) \oplus OR \\
&= (((EXOR \ominus EXOR) \oplus OR \ominus (EXOR \ominus EXOR) \oplus OR) \oplus OR \ominus \\
&\quad ((EXOR \ominus EXOR) \oplus OR \ominus (EXOR \ominus EXOR) \oplus OR) \oplus OR) \oplus OR
\end{aligned}$$

The graphical representation of $Tree[3]$ is given in figure 4.5. This simple system of two net equations describes every binary comparison tree for two operands of bit–length 2^i. The result of the circuit is 1, if the two binary numbers differ in at least one bit.

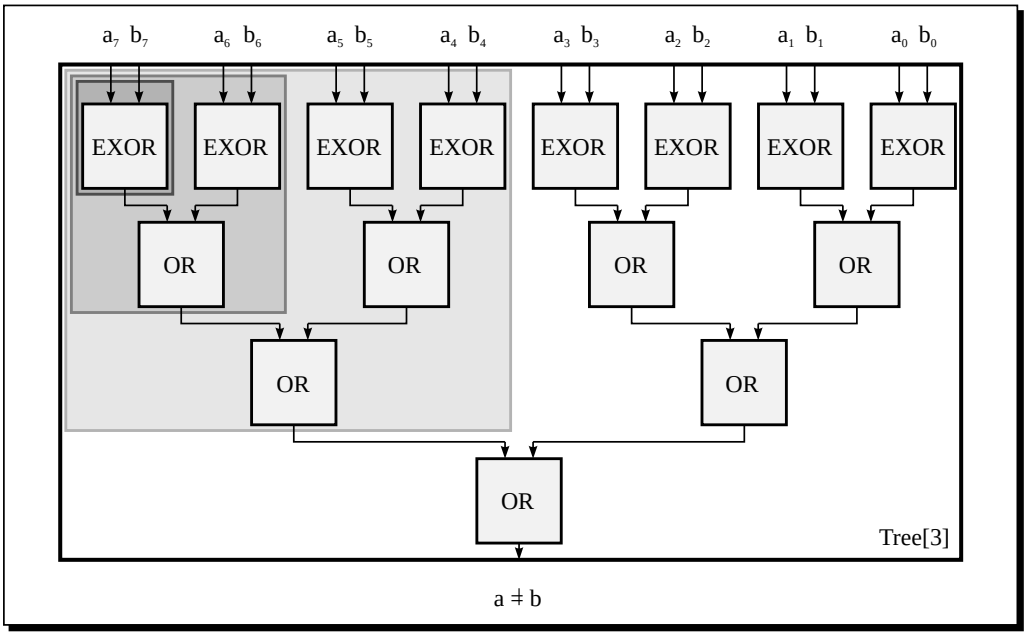


Figure 4.5: Expanded representation of the binary comparison tree *Tree*[3]

This small example shows that the method of recursive net equation systems enables the designer to specify even large circuits by a small number of equations. But the input via bicategorial expressions has some disadvantages:

- the designer creates such an expression after drawing the net on a sheet of paper and then slicing it according to the rules of the calculus.
- for small nets he can already get uncomfortably large expressions.
- short modifications in the circuit may cause tremendous changes in the expressions. Usually they are newly calculated which tends not to be an interactive way of working.
- the initial input may contain a lot of errors, especially if the circuit contains complex wiring components.
- the communication with other design tools is difficult, because the designer has to setup the relation between the calculated results and the given input himself.

To overcome these disadvantages the CADIC system has a graphical input tool, a schematic editor which supports all the features of the underlying mathematical calculus, especially the parameterized and recursive design methodology. In the following sections of this chapter we will introduce the work with this graphical editor of CADIC . We begin our session with some small examples which will show you the basic functions of the editor. During the main part of the chapter we will design an arithmetical circuit as a realistic example to demonstrate, how the user can setup flexible specifications of circuits.

4.3 Getting Started

⇝ 3

If you have followed the instructions given in chapter 3 , then you will see the graphical user interface with its components. In the menuline the topmost menu is displayed which contains a push button for every tool that is integrated in the current graphical environment. From this menu you should select the entry `Schematic Editor` in order to activate the submenu for the graphical editor. After pressing the left mouse button the contents of the menuline changes and the functions of the graphical editor are displayed. These functions are grouped according to the objects they will affect.

In the following design examples we will demonstrate the use of the most important editor functions step by step. A user manual for all editor functions is given in chapter 12.

⇝ 12

4.4 Graphical Specification

From the functional description of a circuit we will now directly derive the schematic inputs for the graphical editor of CADIC . The procedure of entering a schematic usually takes the following basic steps:

- open a new schematic sheet with an appropriate name and parameter list
- select and manipulate basic cells or parameterized macros

- ☐ connect the cells, macros and the schematic border with wires
- ☐ enter and place comments for the documentation of the schematic
- ☐ save the schematic

The sequence of these steps can be changed at some positions. You can enter some cells, connect them with wires, add new cells, save the schematic during the session, etc. Especially you can delete cells, wires or comments in the case of mistakes.

In the following sections we will concentrate on these basic steps of drawing a schematic. During this sample session you will especially learn how to open and save schematics, enter, resize, copy, move, rotate and rename basic cells and parameterized macros, enter and delete wires, enter comments and enter additional equations about wire variables. Beside these operations the graphical editor offers more functionality, which will be explained in detail in chapter 12.

4.5 The First Design: A HalfAdder

In our first design we want to show the graphical input for the halfadder given by the equation

$$HalfAdder = (\vdash \ominus \neg \ominus 1_1^{\mathfrak{o}}) \oplus (1_1^{\mathfrak{o}} \ominus \vdash \ominus + \ominus \dashv) \oplus (AND \ominus EXOR)$$

Opening a New Schematic

⇝ 12.1,12.2.1

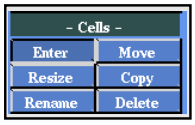
The first step we have to do is to open a new schematic sheet. This is done by selecting the entry `Load` from the submenu `-Schematics-`. You will see a table of the schematics in the directory given by the environment variable `DAGDIR`. At this point of the session this table will only contain a single entry labelled `**** New ****` as it is shown in figure 4.6. Select this entry by moving the pointer onto it and press the left mouse button as soon as it is highlighted. After that a text input window will appear on your screen and you have to specify the name of the new schematic.

```
Please enter New Schematic Name:  <HalfAdder    >
```

Type in the text `HalfAdder` and press the return key. Then you have opened a new schematic which will first be given by an empty frame. Note, that the name in the upper right field of your environment has changed to `HalfAdder` which is the name of the new schematic.

Entering Basic Cells

⇝ 12.1,12.3.1



In the next step we select the appropriate cells for the specification of the halfadder and place them within the given frame. For this purpose select the entry `Enter` from the submenu `-Cells-` in the menuline. After activating this point with the left mouse button, you will see three new windows upon your working area. Each of these windows contains a list of cells of a certain type (c.f. figure 4.7).

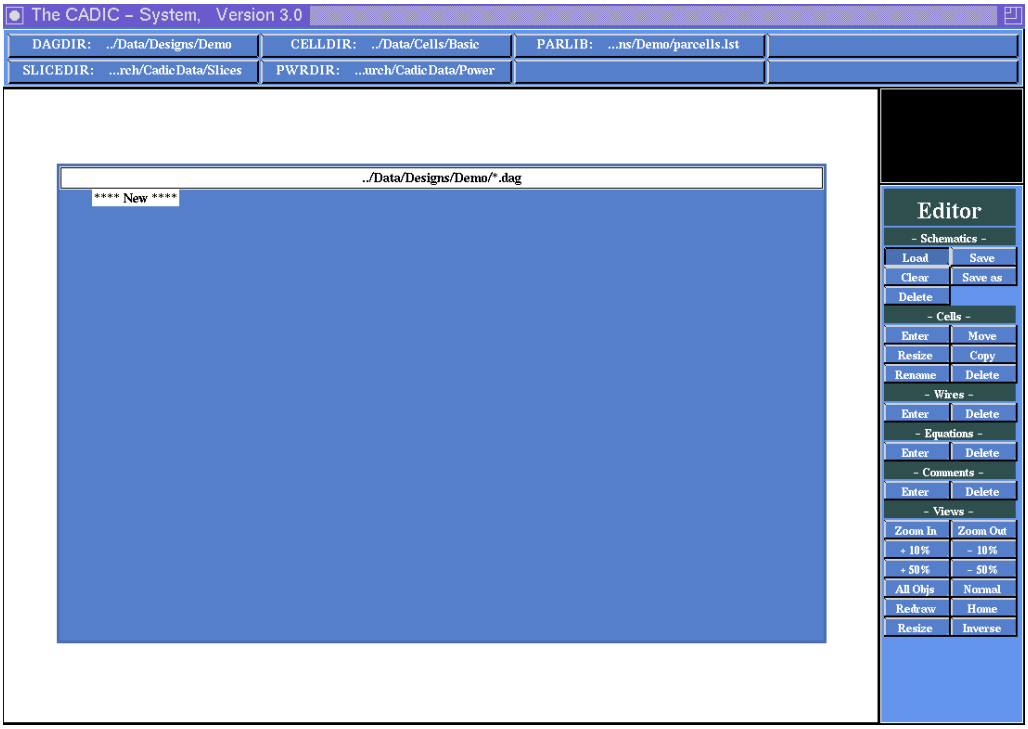


Figure 4.6: Dialog for opening a new schematic

In the upmost window you see the names of the cells in the basic cell library which are located in the directory described by the environment variable

`CELLDIR`. The second window shows the names of macro cells which have been defined by a previous input of a schematic. At this point of the session the macro list is empty. In the third window the names of the parameterized macro cells are listed which have been used during the current session. Actually this list contains one single entry labelled `--- New Cell ---`.

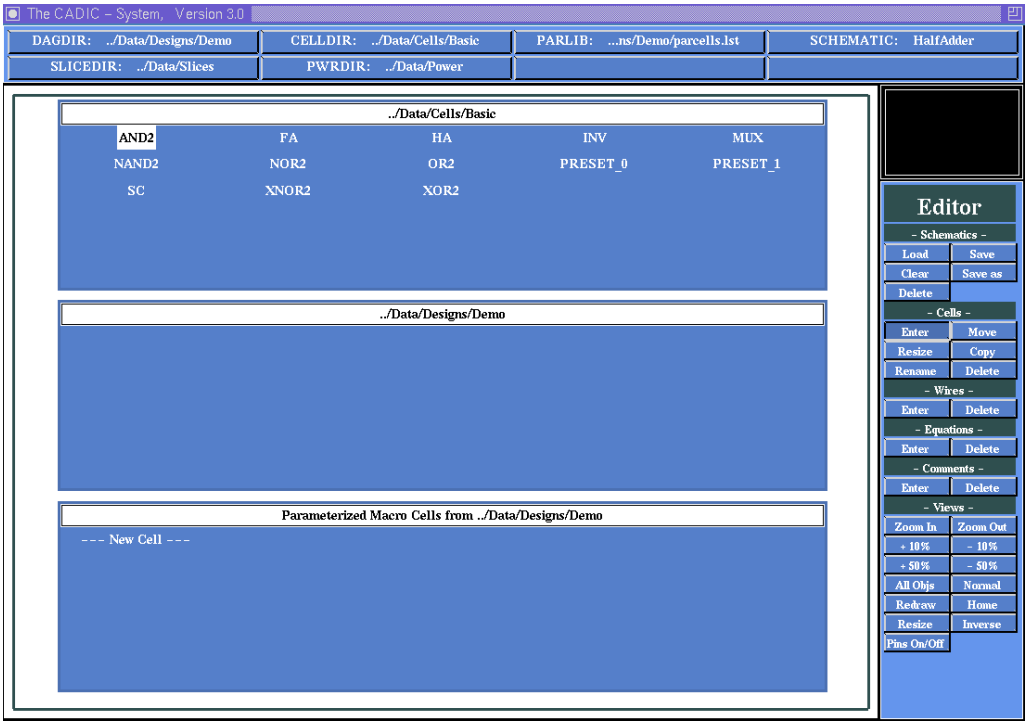


Figure 4.7: Dialogs for cell and macro selection

If you look at the equation for the halfadder, the first cell we need is and AND gate which computes the carry of the two inputs. Move the pointer onto the name `AND2` within the first window and press the left mouse button if the name is highlighted. The cell list windows will be closed and you see the shape of an AND gate in the workarea which will follow the pointer motion within the workarea. Now you can pick the right position to place it (c.f. figure 4.8).

If you drop the gate by pressing the left mouse button, it is redrawn with two input pins at its northern border and an output pin at its southern border, indicated by green points. The pins are labelled with the names `I1`,

I2 and O1. Small arrows indicate the directions of the pins.

The gate has two more pins called VDD and VSS (blue points), which represent the power/ground supply ports. These pins are uninteresting for the logical function of our circuit and we will not use them during the editor session. The power supply nets which will connect all these supply pins are automatically generated by an appropriate tool (c.f. chapter 8).

⇝ 8

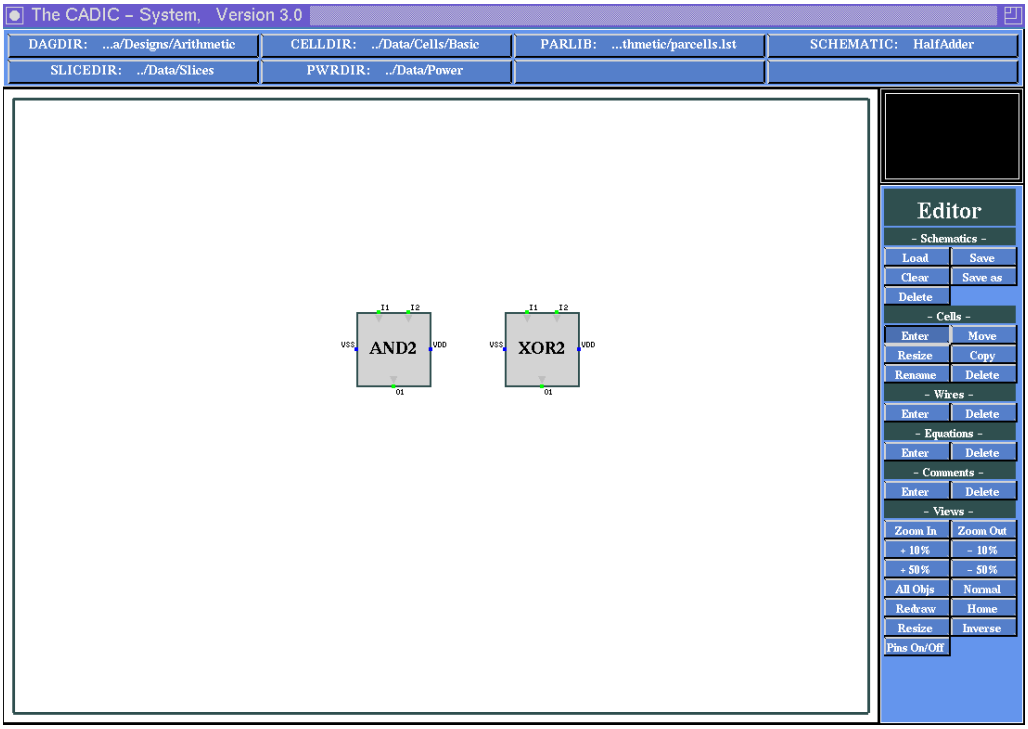


Figure 4.8: Placement of the basic cells for HalfAdder

After the placement the cell list windows are popped up again in order to select another gate. Now move the pointer to the entry XOR2 to get an EXOR gate for two inputs. We need this gate to compute the sum modulo 2 of the two input bits, so place it right beside the AND gate as it is shown in figure 4.8. Because this is the last gate we need for the halfadder, you should abort the cell selection mode by pressing the right mouse button within one of the three list windows.

Entering Wires

⇝ 12.4.1

To draw the wiring of `HalfAdder` select the entry `Enter` from the `-Wires-` submenu. First we connect the output pins of the two gates directly with the southern border of the schematic. Move the pointer near to the output pin `O1` of the `AND2` gate. If the distance to the pin is less than a certain tolerance value the crosshair cursor will snap to the pin and you can fix the start point of the wire by pressing the left mouse button. Now move vertically down to the southern border of the schematic. During the pointer motion you will notice a grey rubberbanding line following the pointer. This line indicates the current connection which will be established, if you press the left mouse button to fix the end point of the wire. In some cases the rubberbanding line will dissappear in order to show you that the current connection is illegal. For example this is the case, if the connection would cut a cell or the schematic border. Pressing the left mouse button with an invisible rubberband would result in nothing but the selection of a new start point.

If you move the pointer near to the southern border of the schematic it will snap again to the border line and you can fix the end point by pressing the left mouse button. The newly created wire is established and redrawn as a yellow connection between the pin and the schematic border. In this case of connecting a pin of a basic cell the width of the wire is automatically set to 1. Later you will see that the width of a wire can also be set to flexible values.

After connecting the output pin `O1` of the `XOR2` gate with the southern border of the schematic we turn to the input pins. Connect the left input `I1` of the `AND2` gate as well as the right input `I2` of the `XOR2` gate directly with the northern border of the schematic. These two wires represent the inputs a and b of the halfadder. As shown above we have to feed these values into both the `AND2` and `XOR2` gate, i.e. we branch the wires and connect them with the appropriate input pins. To do this move the pointer onto the left wire. You can notice that it snaps to the wire if the distance is less than a certain tolerance value. Press the left mouse button to fix

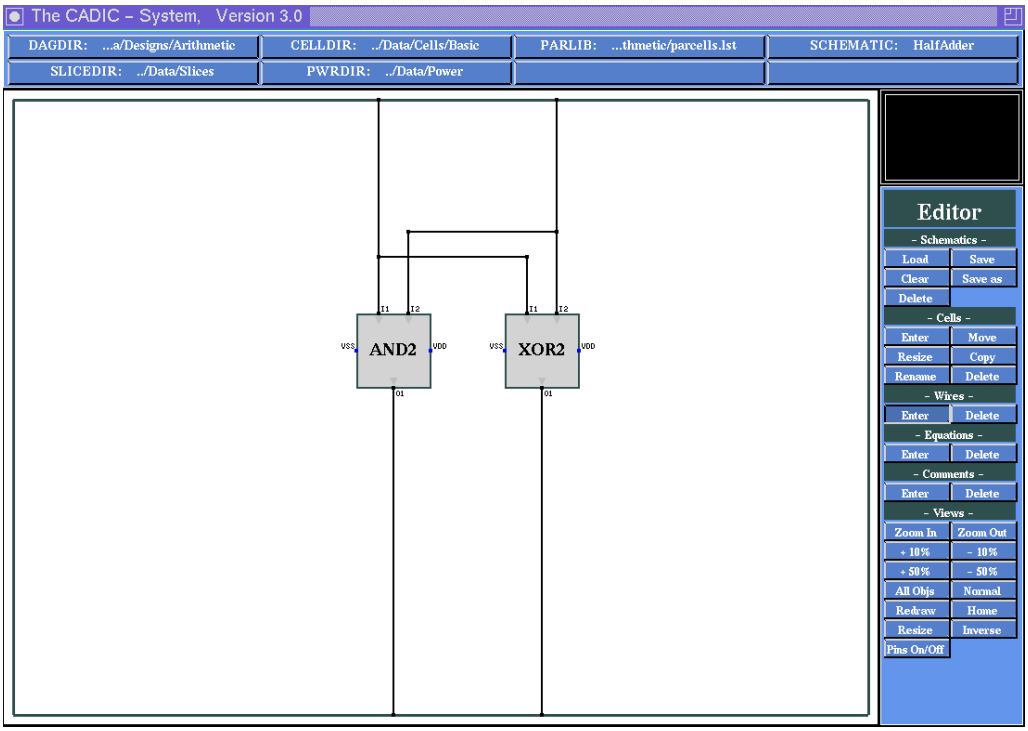


Figure 4.9: Wiring for `HalfAdder`

the start point and move the pointer to the left input pin of the `XOR2` gate. The connection is indicated by a rubberbanding line between the start point and the current position of the pointer. If you select an illegal position the rubberbanding line will disappear. If you have setup the right connection to the left input of the `XOR2` gate press the left mouse button to fix the end point of the wire. The yellow dot at the start point of the new wire indicates a branching of the previous vertical wire. Implicitly the branch node is considered to be a basic operation in the underlying calculus. In this special case we have entered an east–branch.

In the same way you can connect the right vertical wire with the right input pin of the `AND2` gate. You should have entered a schematic similar to that shown in figure 4.9. Terminate the enter wire mode by pressing the right mouse button two times (the first time you press the right mouse terminates the selection of an end point and turns to the selection of a new start point).

Saving the Schematic

⇝ 12.2.2

Save the schematic by selecting the entry `Save` from the `-Schematics-` submenu. The graphical input is then written to the file `HalfAdder.dag` in the directory given by the environment variable `DAGDIR`.

4.6 Going into Hierarchy: A FullAdder

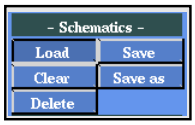
In this section we will specify a small circuit which consists of two hierarchy levels. According to the description from above

$$FullAdder = (HalfAdder \ominus 1_1^{\Phi}) \oplus (1_1^{\Phi} \ominus HalfAdder) \oplus (OR \ominus 1_1^{\Phi})$$

will we build a fulladder with the help of the halfadder from the previous section.

Opening a New Schematic

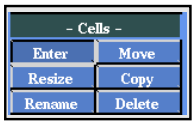
⇝ 12.2.1



We call this new schematic `FullAdder` and open it by following the steps in section 4.5. Note that the list of schematics now contains the entry `HalfAdder` beside the position of `**** New ****`. After selecting a new schematic you should type in the name `FullAdder` and press return. Now we have created an empty frame for entering the fulladder design.

Entering Macro Cells

⇝ 12.1,12.3.1



In the next step we select the appropriate cells for the specification of the fulladder and place them within the given frame. For this purpose select the entry `Enter` from the submenu `-Cells-` in the menuline. As shown above we need two instances of a halfadder together with an instance of an OR gate. The halfadder is described by its own schematic. Therefore you will now see an entry `HalfAdder` in the second cell selection window. This is the window for discrete macro cells, i.e. macro cells which do not depend on any free parameter.

Move the pointer onto this entry and select it by pressing the left mouse button after it is highlighted. Then the cell selection windows are popped

down and you notice a cell of a certain size following the pointer motion within the workarea. Place it as it is shown in figure 4.11 and fix the position by pressing the left mouse button. After that the cell selection windows are popped up again in order to choose another cell.

Select a second instance of the halfadder and place it lower right to the first one (c.f. figure 4.10). Finally select an OR gate (`OR2`) from the first cell selection window which contains the list of basic cells. Place it at the position shown in figure 4.10 below the second half adder.

Now terminate the cell selection mode by pressing the right mouse button within one of the selection windows.

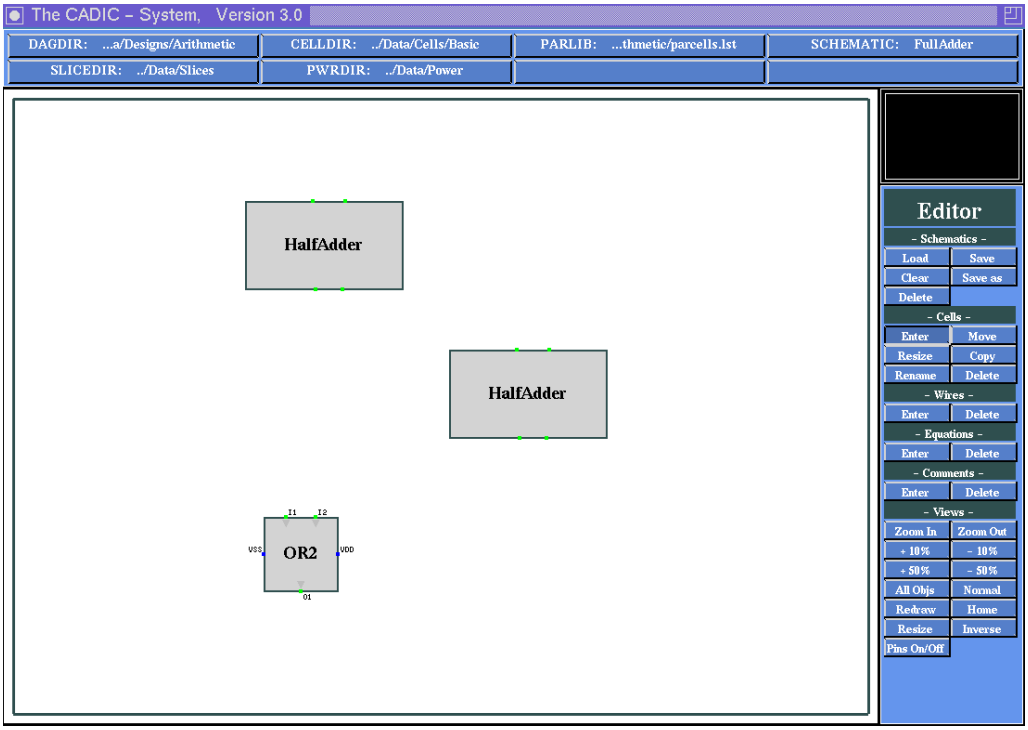


Figure 4.10: Using the macro `HalfAdder` to construct a fulladder

Before we continue you should take a short look at the macro cells and compare them to the basic cells. You notice that the macro `HalfAdder` has two pin connectors at its northern border and southern border. These pins correspond to the wires we have connected with the schematic borders of the corresponding schematic for the halfadder. The position of the pins is

relative to the position of the wire connection at the schematic frame.

In contrary to the basic cells there are no pin names shown for the macro cells. This does not mean that the pin connectors have no names, but they are automatically generated and omitted for reasons of simplicity. The pin names of macro cells contain the exact position of the corresponding wire. For example the upper left pin of `HalfAdder` has the name `n[0,0]` which means that it belongs to the northern border and represents the interval of wires from `0` to `0` inclusively, i.e. it represents the first wire connected to the northern border and this wire has the width 1. For the conventions of naming pins of macro cells see also 12.3.1.

⇝ 12.3.1

Entering Wires

⇝ 12.4.1

To draw the wiring of `FullAdder` select the entry `Enter` from the `-Wires-` submenu. First we connect the output pin of the OR gate and the right output pin (`s[1,1]`) of the lower halfadder macro directly with the southern border of the schematic. Move the pointer near to the output pin of the `OR2` gate. If the distance to the pin is less than a certain tolerance value the crosshair will snap to the pin and you can fix the start point of the wire by pressing the left mouse button. Move vertically to the southern border of the schematic, dragging the grey rubberbanding line, and fix the end point in the same way. As mentioned above the width of this wire is automatically set to 1.

Now move the pointer to the right pin at the southern border of the lower halfadder and fix the start point with the left mouse button. Draw the rubberbanding line vertically down to the southern schematic border and fix the end point. Because the pin of the macro cell represents one single binary value, the width of this wire is also automatically set to 1.

After connecting these output pins with the southern border of the schematic we turn to the input pins of the fulladder. Connect the two input pins of the upper left halfadder as well as the right input pin of the lower right halfadder directly with the northern schematic border. These three wires represent the inputs of the fulladder.

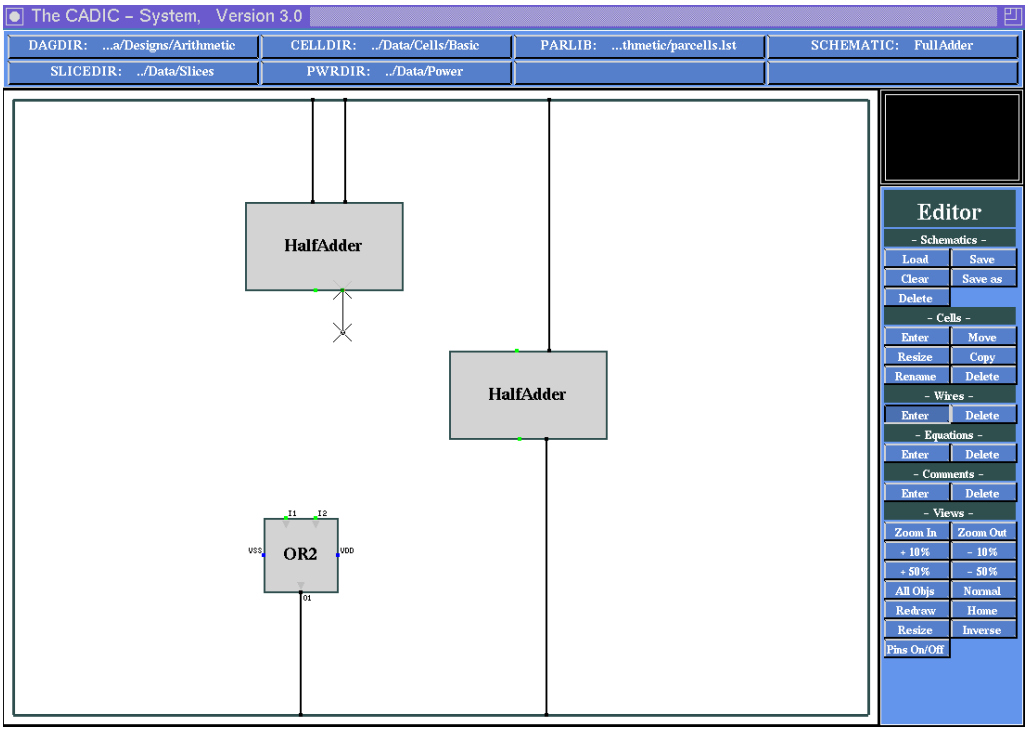


Figure 4.11: Wiring of `FullAdder` and creating a knee for a connection between two pins

According to the sketch from figure 4.4 we still have to connect the right output pin of the left halfadder with the left input pin of the right halfadder and the left output pins of both halfadders with the input pins of the OR gate.

To do this wiring move the pointer onto the right output pin of the left halfadder. You can notice that it snaps to the pin if the distance is less than a certain tolerance value. Press the left mouse button to fix the start point and move the pointer vertically downwards. The actual connection is indicated by the rubberbanding line between the start point and the current position of the pointer. If you select an illegal position the rubberbanding line will disappear. This is the case if the chosen connection intersects a cell or moves along an already existing wire or the border of the schematic.

In our case we first create a small vertical wire, from which we will complete the connection between the two pins. Fix the end point as it is shown in

figure 4.12. This end point now serves as the start point for the next wire, i.e. just move the pointer onto the left input pin of the right half adder. You will notice that the rubberbanding line will automatically create a knee for the proper connection. Fix the end point at the left input pin of the halfadder by pressing the left mouse button.

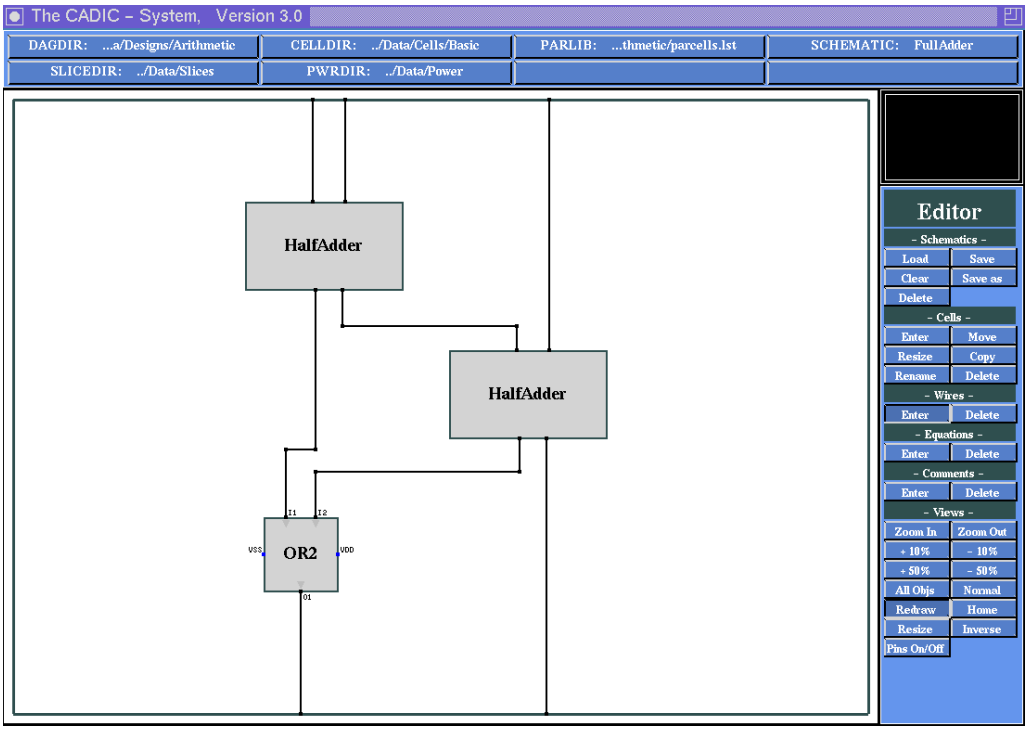


Figure 4.12: Final wiring for `FullAdder`

We also apply this method of connecting two pins for the remaining two wires between the left output pins of the halfadders and the input pins of the OR gate. Finally you can terminate the enter wire mode by pressing the right mouse button two times within the workarea.

Saving the Schematic

⇝ 12.2.2

Save the schematic by selecting the entry `Save` from the `-Schematics-` submenu. The graphical input is then written to the file `FullAdder.dag` in the directory given by the environment variable `DAGDIR`.

4.7 Parameterization: An n Bit Comparison Tree

In this section we will show how you can setup a parameterized specification of a circuit. In contrast to the previous two sections, where we designed a discrete hierarchical circuit, we will now specify a whole family of circuits. The advantage of CADIC is that we can do this with the help of a finite number of graphical inputs.

The parameterization is done with the help of recursive equations, i.e. we use a regularity in the circuit structure to get a compact description (a small number of inputs). Normally a recursive specification consists of basic equations which serve as end conditions of a recursion. These are equations, where at least one parameter of the subcircuit is given by a fixed nonnegative integer value. You can have more than one basic equation for the same subcircuit, for example if there are different basic elements for different initial parameter values. The recursion itself is given by so called general equations. These are equations, where all parameters are free. For each subcircuit there may exist only one general equations. If this is not the case the refinement operation would ambiguous.

The example of the n bit comparison tree, we will show in this section, can be described by two parameterized equations:

$$Tree[0] = EXOR$$

$$Tree[i] = (Tree[i-1] \ominus Tree[i-1]) \oplus OR$$

The first equation is a basic equation which serves as the end of the recursion. It handles the case that we have to compare two single bits a_0 and b_0 ($n = 2^0$). The second equation is the general recursive equation, where we reduce the comparison of two $n = 2^i$ bit numbers to the comparison of the upper and lower halfs and combining these intermediate results by an OR gate. In the following sections we will now show, how you can enter a schematic input for these two equations.

4.7.1 Basic Equation for the n Bit Comparison Tree

Opening a New Schematic

⇝ 12.2.1

We begin with the basic equation for the recursive description of the n bit comparison tree. This will be a compare subcircuit for $n = 2^0$ bit operands. Therefore we call this schematic `Tree[0]` which will be opened following the steps in section 4.5. Note that the list of schematics now contains the entries `HalfAdder` and `FullAdder` from our previous work beside the position of `**** New ****`. After selecting a new schematic you should type in the name `Tree[0]` and press return.

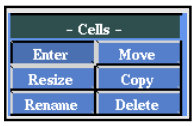
```
Please enter New Schematic Name:  <Tree[0]    >
```

The name of the new schematic is `Tree`. The square brackets introduce a list of parameters which can be used during the specification of the schematic. In this case we have one single parameter which is set to the constant value `0`. In general you can use up to 32 parameters for one schematic.

⇝ 12.1

Entering Basic Cells

⇝ 12.1,12.3.1



In the next step we select the appropriate cells for the basic compare element and place them within the given frame. For this purpose select the entry `Enter` from the submenu `-Cells-` in the menuline. After picking this point with the left mouse button, you will see the three cell selection windows upon your working area (c.f. figure 4.7).

The task to compare two single bits a_0 and b_0 is trivial and can be solved with the help of an EXOR gate. The result of this gate is 1, if the input bits are different, and 0, if they are equal.

Select the entry `XOR2` from the first cell selection window and press the left mouse button, if it is highlighted. Move the gate into the middle of the working area and drop it there by pressing the left mouse button again. Because this is the only cell we need in this schematic you can terminate the cell selection by pressing the right mouse button within one of three selection windows.

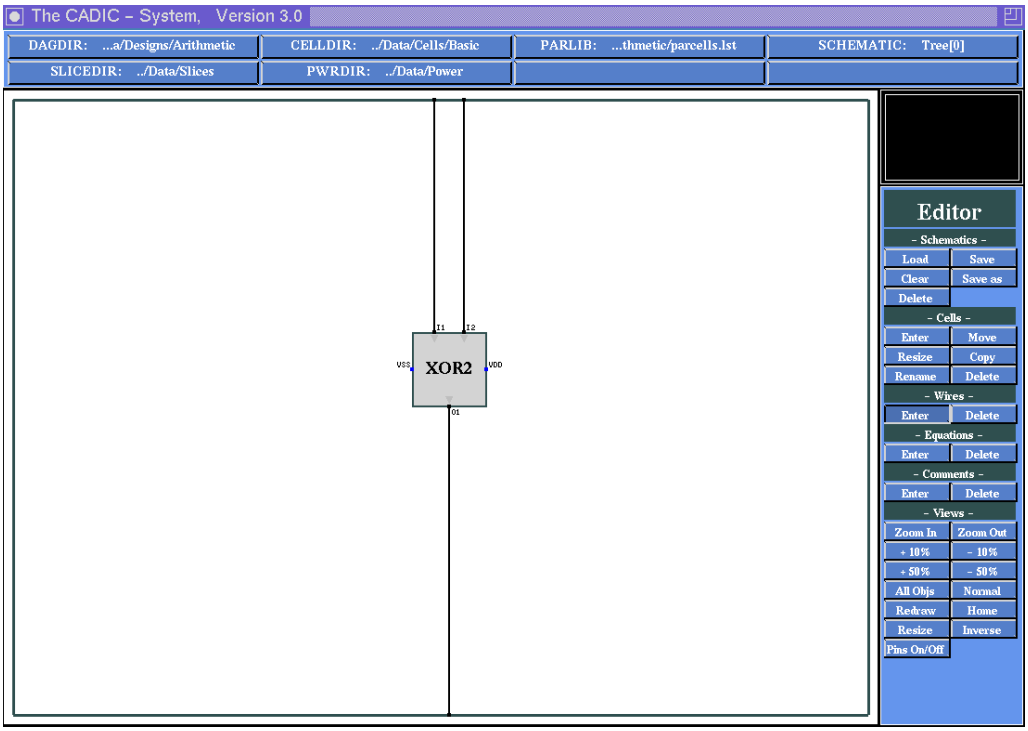


Figure 4.13: The basic equation for the n bit comparison tree

Entering Wires

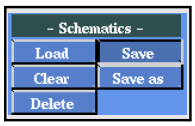
⇝ 12.4.1

To draw the wiring of `Tree[0]` select the entry `Enter` from the `-Wires-` submenu. The wiring in this case is very simple. We just have to connect the pins of the EXOR gate with the corresponding schematic borders. Move the pointer near to the output pin of the `XOR2` gate. If the distance to the pin is less than a certain tolerance value the crosshair will snap to the pin and you can fix the start point of the wire by pressing the left mouse button. Move vertically to the southern border of the schematic and fix the end point in the same way. In the same way connect the input pins of the gate with the northern border of the schematic. The width of all three wires is automatically set to 1.

The schematic should look like that shown in figure 4.13. Finally terminate the enter wire mode by pressing the right mouse button twice within the working area.

Saving the Schematic

↝ 12.2.2

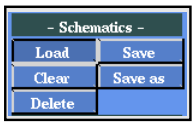


Save the schematic by selecting the entry `Save` from the `-Schematics-` submenu. The graphical input is then written to the file `Tree[0].dag` in the directory given by the environment variable `DAGDIR`.

4.7.2 General Equation for the n Bit Comparison Tree

Opening a New Schematic

↝ 12.1,12.2.1



We will now enter the recursive equation of the n bit comparison tree, for which we first have to open a new schematic sheet. After the selection of the entry `Load` from the submenu `-Schematics-` you should type in `Tree[n]`.

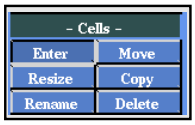
As in the case of the basic equation the square brackets introduce the parameter list of the schematic. Here we have one free parameter `n` which can be used within the elements of the schematic as macros cells and wires.

You may use any legal identifier as the name for a parameter. The syntactical structure of an identifier is defined as you may know it from programming languages.

↝ 12.1

Entering and Manipulating Cells

↝ 12.1,12.3.1



In our recursive description we reduce the comparison of two 2^n bit numbers to the parallel comparison of the upper 2^{n-1} and lower 2^{n-1} bits of the numbers. This means that we need two instances `Tree[n-1]` of 2^{n-1} comparison trees within `Tree[n]`.

In this next step we select the appropriate cells for the recursive specification of the comparison tree and place them within the given frame. For this purpose select the entry `Enter` from the submenu `-Cells-` in the menuline.

The input of the parameterized macros for the 2^{n-1} bit comparison trees is done within the third of the cell selection windows. At this point of the session this windows contains a single entry `--- New Cell ---` for the definition of a new parameterized macro cell. Select this in the same way

as you have selected to open a new schematic in order to define a new parameterized macro cell. After hitting the left mouse button you will be prompted to enter a new macro name.

```
Please enter Parameterized Name:  <Tree[n-1]     >
```

Type in `Tree[n-1]` to specify a macro for a comparison tree for 2^{n-1} bit numbers.

Note that you can only use the parameters of the schematic to build arithmetic expressions for the parameters of the macro cells. Try to specify a cellname `Tree[k-1]` and the system will respond with the message

```
!  Error !  "Tree[k-1]" uses illegal parameter "k"
```

The syntax diagrams for the expressions allowed as parameters of these macro cells are given in chapter 12.

After the specification of a correct cellname you will see the graphical representation of the macro, i.e. a red bounded box with the cellname in its center. You can move this symbol inside the frame and place it at the desired position by clicking the left mouse button. The macro can only be placed on legal positions, i.e. there must not be an overlapping with other macros, wires or the border of the schematic. Try to move the macro onto the schematic frame and you will notice a change of its colour from red to grey. This change indicates the selection of an illegal position. Now place the macro onto the position you see in figure 4.21.

The size of the macro is chosen by default, because the selected parameterized macro represents a whole class of macros. Its exact size can be determined, if the values for the parameters are known. For the abstract specification level of the editor the size of a cell or a macro is not important. In order to setup a comfortable schematic which is easy to understand, we will now resize the macro, such that it gives us an impression of the relative sizes. Before we can perform the resize operation, we have to terminate the enter cell function (third mouse button in one of the cell selection windows).

⇝ 12.3.3

For the resize operation push the button `Resize` in the `-Cells-` submenu.

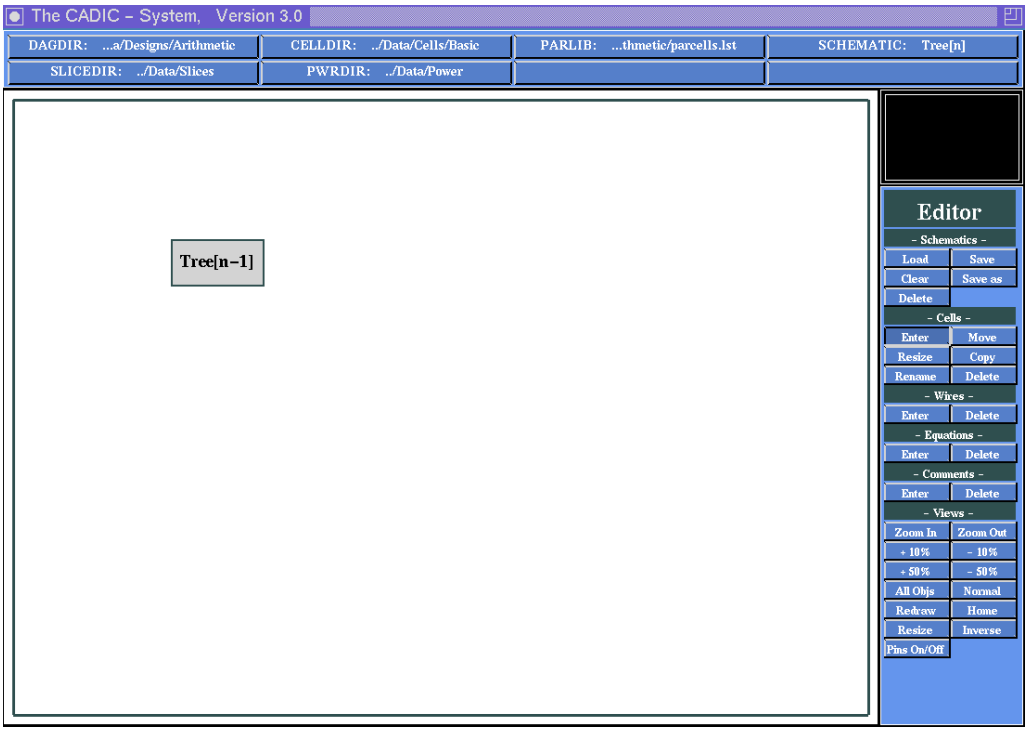
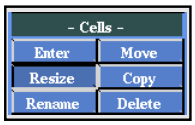


Figure 4.14: Placing a parameterized macro cell for the recursive specification of the comparison tree



Then you are in the cell selection mode and can pick the desired cell within the workarea. Move the pointer onto the workarea and you will see the macro being highlighted (its colour changes from grey to blue). In cell selection mode always the macro nearest to the pointer will be highlightet, you need not be inside it. At this point of our session cell selection is trivial because we only have a single macro within the workarea. The selection is done by clicking the left mouse button.

After the selection you can draw a rubberbanding frame which is anchored in the upper left corner of the macro and represents its new size. During the resize operation you need not hold down the left button. Just move the pointer to the desired position and confirm the new size by clicking the left mouse button or cancel the operation with the right mouse button (you will be returned to cell selection mode).

If you confirm a new size for the macro you will be in cell placement mode.

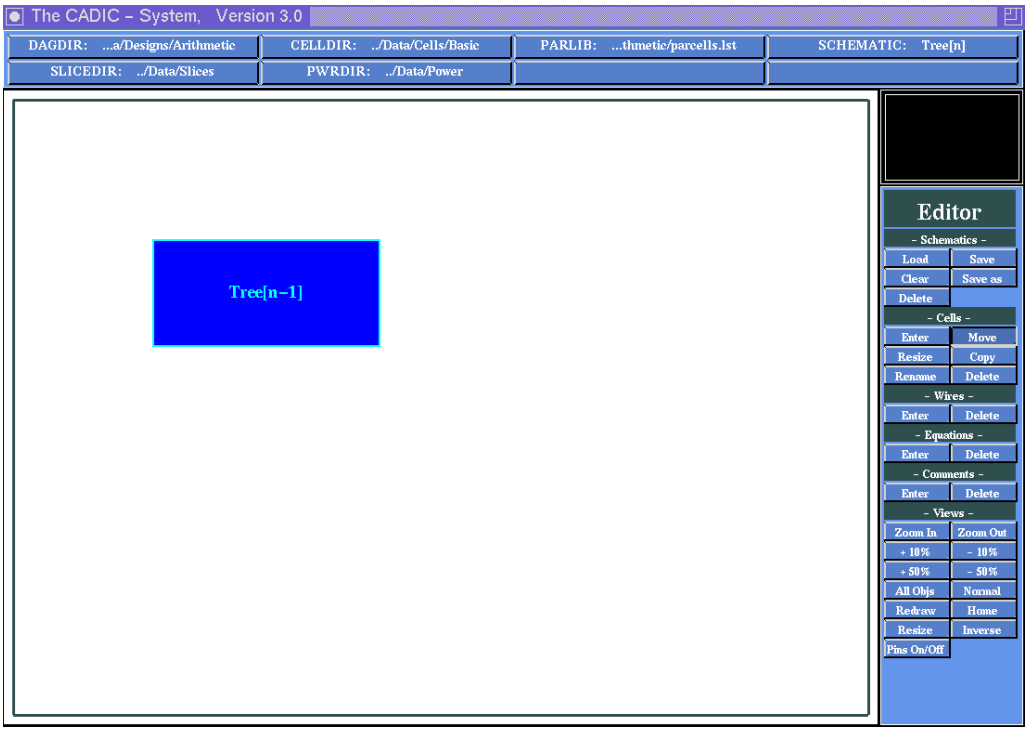


Figure 4.15: Resizing a parameterized macro cell

The reason for this are possible overlappings of the resized macro with its neighbours or with wires. You have to choose a new position for it which is only possible within the given constraints (i.e. no overlapping with macros, wires or the border). Sometimes it can happen that you have no enough free space in your workarea to perform the resize operation without creating overlaps with other cells. In this case you should terminate the resize operation and enlarge the workarea with the help of the functions from the submenu `-Views-` which are explained in section 12.7.1.

⇝ 12.7.1

Move the resized macro to the previous position and confirm it with the left mouse button. Now your schematic should look like that shown in figure 4.15.

⇝ 12.3.4

As described above the 2^n bit comparison tree simultaneously compares the upper 2^{n-1} and the lower 2^{n-1} bits. We now have positioned the macro `Tree[n-1]` for the upper half of the operands. A second instance of this macro will be needed for the lower half. We could create this instance by

following the above steps (selecting a parameterized macro, placing and resizing it). But the editor offers a much simplier way to perform this operation. Push the entry `Copy` from the `-Cells-` submenu. This enables the cell selection mode where you can select an instance to be doubled. Press the left mouse button and you will get a second macro labelled `Tree[n-1]` which has the same size as our first macro. Place it within the workarea in order to have the schematic look like that in figure 4.16. During the placement operation you can watch the macro changing its colour, if it is placed on the first `Tree[n-1]` or on the border. In this case you cannot drop it, i.e. pressing the left mouse button has no effect. You can cancel the operation by pressing the right mouse button. The copy will dissappear and you will return to cell selection mode. Pressing the right mouse button again will terminate the whole copy operation.

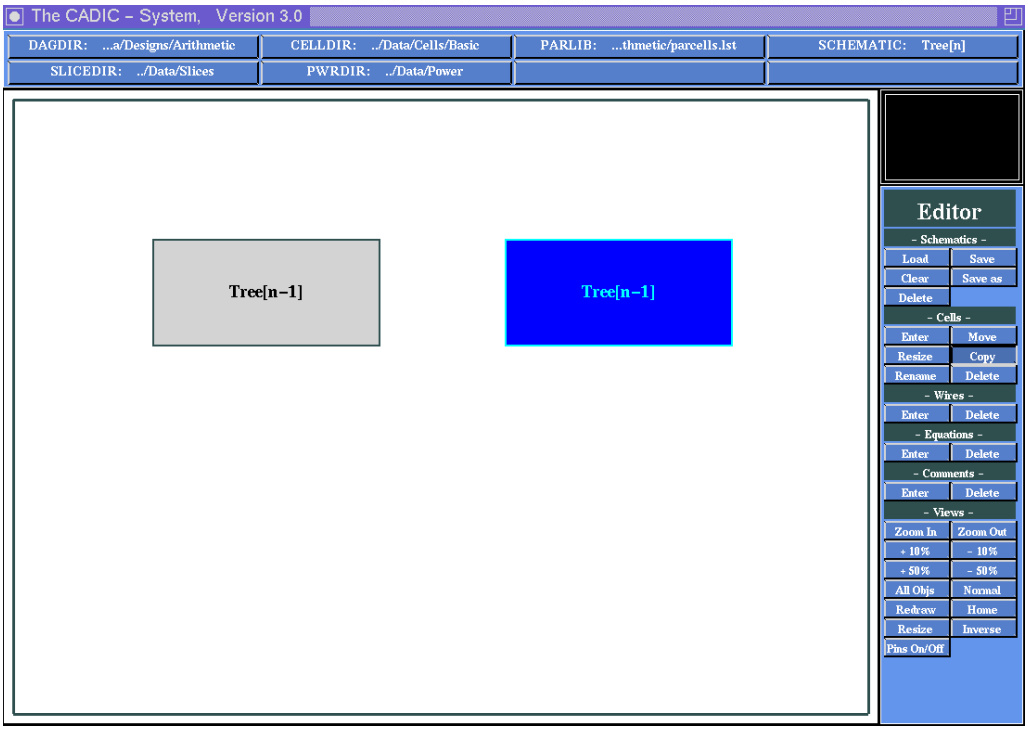
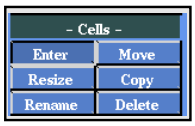


Figure 4.16: Creating a copy of an exisiting macro cell

⇝ 12.1,12.3.1



Finally we need an OR gate to combine the results of the two 2^{n-1} comparison trees. For this purpose select the entry `Enter` from the submenu `-Cells-` in the menuline again. Select an OR gate (`OR2`) from the first cell

selection window which contains the list of basic cells. Place it at the position shown in figure 4.17 below the two 2^{n-1} bit comparison trees, such that we get a tree–like structure. Now terminate the enter cell mode by pressing the right mouse button within one of the three selection windows.

Entering Wires

⇝ 12.4.1

In this section we show how to connect the modules with wires. The main point you will observe during this procedure is that because of the parameterized description the lines drawn will have variable widths. The width of a line specifies the number of parallel wires contained in this bundle.

We start with the left `Tree[n-1]` which compares the upper half of the bits of the the operands. First select the point `Enter` from the `-Wires-` submenu and move the pointer into the workarea. The cursor has changed to a crosshair and you can select the start point for a new wire. Move the pointer to the northern side of the left `Tree[n-1]` macro and position it to the middle of this side. If you move the pointer slowly toward the border of the macro you can observe that it snaps onto the border, if the distance becomes less than a certain tolerance value. This indicates that you have selected a special position within the schematic. The same behaviour can be noticed if you reach the frame, a pin of a cell or another wire. If the pointer has snapped to the border of the macro press the left mouse button to fix the start point of the new wire. Now move the pointer vertically towards the northern border of the schematic. You will see that it is followed by a grey rubberband line which indicates the position of the new wire. The cursor snaps to the border if the distance is small enough.

Now you can fix the end point of the wire by pressing the left mouse button again. The grey rubberband line will disappear and you will be prompted to enter the width of this wire.

```
Please enter Wire Width:  <2^n      >
```

As mentioned above the width of a wire represents the number of parallel lines within this bundle. In our case this wire describes the input line to

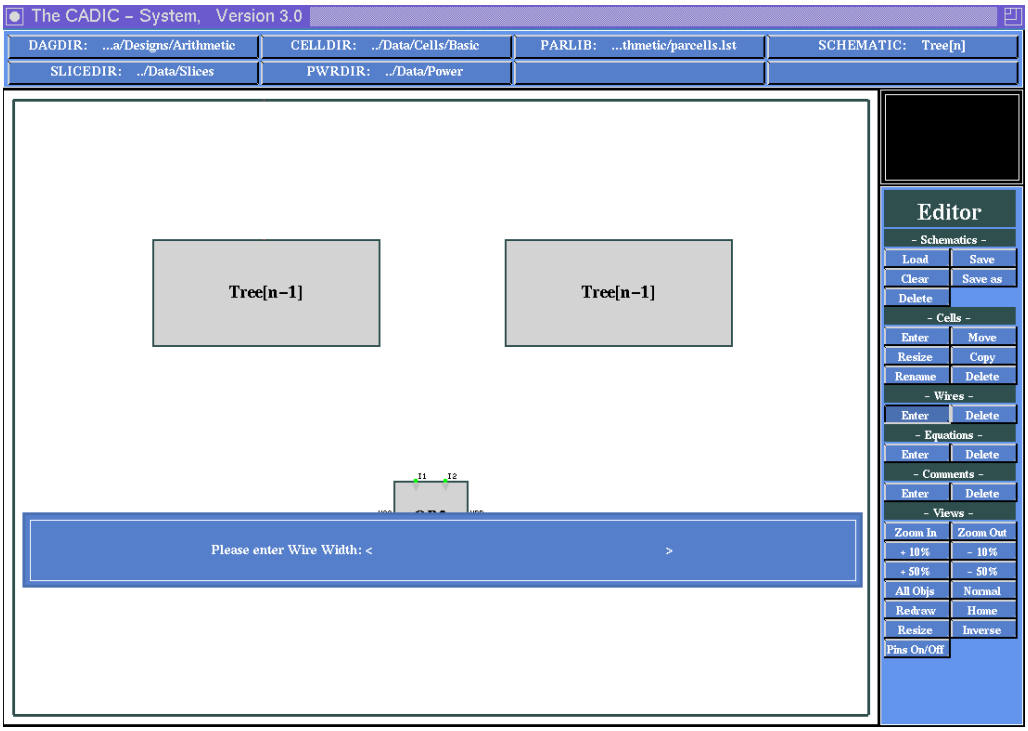


Figure 4.17: Entering a wire of parameterized width

the first comparison tree for the upper half of the operands. The width of a single operand is given by the parameter 2^n. The left `Tree[n]` gets two operands of length 2^{n-1}, i.e. the number of single wires is 2^n which you can type in in the form `2^n`. You will see the new wire as a yellow line from the border of the `Tree[n-1]` to the border of the schematic. In the middle of the wire you can notice the description of its width annotated by a small diagonal line with the appropriate expression (c.f. figure 4.18).

Note that you can only use the parameters of the schematic to build arithmetic expressions or wire variables. The exact syntax for these expressions is given by the diagrams in appendix.

↝ 12.1

We proceed with the other modules. The right instance of `Tree[n-1]` will also be connected with the northern border of the schematic. You can draw this wire in the same way as you did for the left `Tree[n-1]`. Type `2n̂` for its width in order to denote that it describes the wires for the lower parts of the operands.

Now we turn to the southern borders of the two `Tree[n-1]`. The output of both has to be connected with the input pins of the OR gate. Move the pointer near to the middle of the southern border of the left `Tree[n-1]` and fix the start point of the new wire. Because the connection to the input pin of the OR gate has to be drawn with two knees, we first create a small vertical wire segment. After fixing the end point of this segment you are prompted to enter the width of this wire, because the system can not automatically determine the width of this wire (it is connected to a parameterized macro cell).

```
Please enter Wire Width:  <1      >
```

Enter `1` because the result of the 2^{n-1} bit comparison tree is a single binary value. Now continue the drawing of this wire by moving the pointer to the left input pin of the OR gate. The rubberbanding line will automatically create the second knee for the correct connection. If you fix the end point you are not prompted for the width of this wire, because you continued an already existing wire.

The entering of the wire width `1` from above can be avoided, if we start the wire at the input pin of the OR gate as we will do this for the second `Tree[n-1]`. Fix the start point at the right input pin of `OR2` and move the pointer a short distance upwards. If you fix an end point here the wire automatically gets the width 1. Now continue the wire to the middle of the southern border of the right `Tree[n-1]`. Press the left mouse button again and fix the end point of the connection.

Finally connect the output pin of `OR2` with the southern border of the schematic as the output of the whole comparison tree `Tree[n]`.

To leave the enter wire mode press the right mouse button twice. The first button press will deactivate the selection of a wire end point and return to the selection of new start point. The second button press will abort the enter wire mode and return to the menu selection mode.

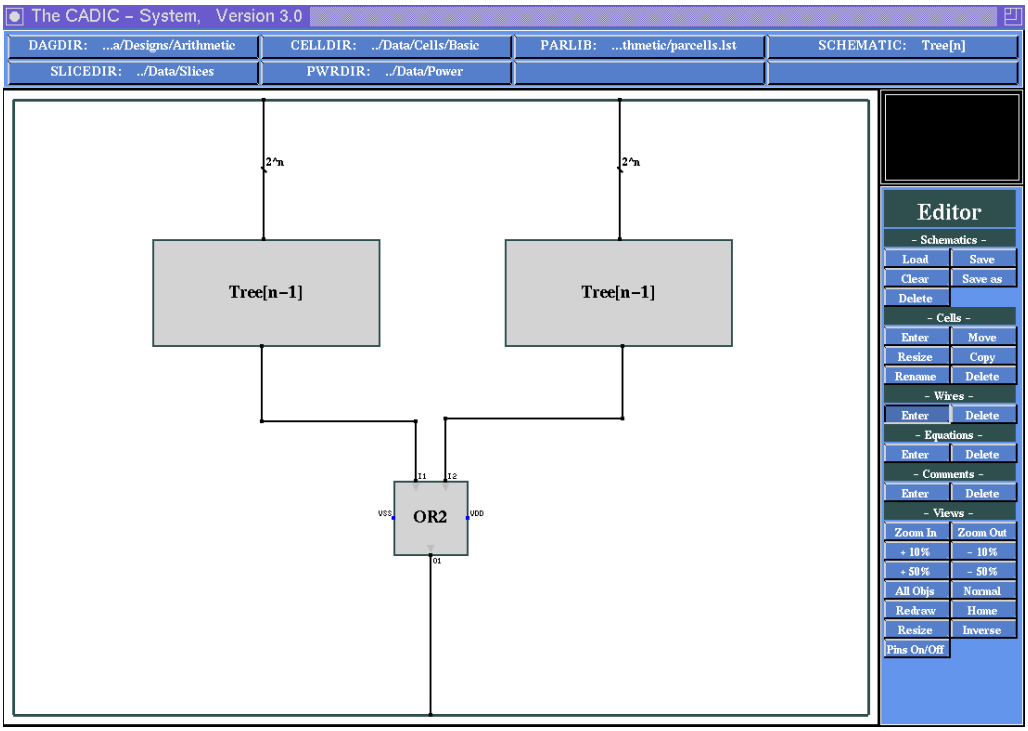


Figure 4.18: Final wiring for the 2^n bit comparison tree

Saving a Schematic

⇝ 12.2.2

Now that we have drawn all elements of the general recursive equation for the 2^n bit comparison tree we must save the schematic. This is simply done by selecting the entry `Save` from the `-Schematics-` submenu. The schematic will be saved to the file `Tree[n].dag` in the directory given by the environment variable `DAGDIR`.

4.8 Design of an n Bit Conditional Sum Adder

After the introduction of basic editor operations and specification methods, we will now enter a larger design. By the example of an addition circuit we will show, how the designer can set up short and handy specifications for often used components of larger designs such as microprocessors. He can perform that by using recursive systems of schematic equations.

4.8.1 Basic Concepts of the Conditional Sum Adder

Simple algorithms for addition which e.g. use the well–known carry ripple principle, have the disadvantage of worst (linear) time behaviour. If you need high performance, you have to consider other, more complex structures which will guarantee a better (logarithmic) runtime. We will show now that this is possible by using recursive definitions which will easily mirror the basic principles of the underlying algorithm.

A nice idea for parallelizing addition was given by Sklansky in 1960 ([Skl60]) and is well–known as conditional sum adder. This adder simultaneously computes the sum as well as the sum plus one for the leading parts of the two operands. This is done at the same time as the sum of the lower parts is generated. If the carry of the lower summation is known, you can select the right version of the sum of the upper parts. If this selection can be done in a small amount of time, i.e. by a subcircuit of constant depth, and if we apply this scheme recursively to the upper and lower part of the operands, we get an addition circuit with logrithmic depth, where the operands are split in parts of equal size.

Let $a = num_n(a_{n-1} \dots a_0)$ and $b = num_n(b_{n-1} \dots b_0)$ be two n bit binary numbers. For simplicity we only consider the case $n = 2^k$. Then

$$a_H := num_{\frac{n}{2}}(a_{n-1} \dots a_{\frac{n}{2}}), a_L := num_{\frac{n}{2}}(a_{\frac{n}{2}-1} \dots a_0)$$

denote the $\frac{n}{2}$ bit binary numbers which are represented by the upper (H) and lower (L) half of the operand a. In the same way we define b_H and b_L. Then it holds:

$$\begin{aligned}
a+b &= (2^{\frac{n}{2}}a_H + a_L) + (2^{\frac{n}{2}}b_H + b_L) \\
&= 2^{\frac{n}{2}}(a_H + b_H) + (a_L + b_L) \\
&= 2^{\frac{n}{2}}(a_H + b_H) + 2^{\frac{n}{2}}((a_L + b_L) \div 2^{\frac{n}{2}}) + (a_L + b_L) \bmod 2^{\frac{n}{2}} \\
&= 2^{\frac{n}{2}}(a_H + b_H + (a_L + b_L) \div 2^{\frac{n}{2}}) + (a_L + b_L) \bmod 2^{\frac{n}{2}} \\
&= \begin{cases} 2^{\frac{n}{2}}(a_H + b_H) + (a_L + b_L) \bmod 2^{\frac{n}{2}} & \text{if } (a_L + b_L) \div 2^{\frac{n}{2}} = 0 \\ 2^{\frac{n}{2}}(a_H + b_H + 1) + (a_L + b_L) \bmod 2^{\frac{n}{2}} & \text{if } (a_L + b_L) \div 2^{\frac{n}{2}} = 1 \end{cases}
\end{aligned}$$

and

$$a+b+1 = \begin{cases} 2^{\frac{n}{2}}(a_H + b_H) + (a_L + b_L + 1) \bmod 2^{\frac{n}{2}} & \text{if } (a_L + b_L + 1) \div 2^{\frac{n}{2}} = 0 \\ 2^{\frac{n}{2}}(a_H + b_H + 1) + (a_L + b_L + 1) \bmod 2^{\frac{n}{2}} & \text{if } (a_L + b_L + 1) \div 2^{\frac{n}{2}} = 1 \end{cases}$$

Because $(a_L + b_L) \bmod 2^{\frac{n}{2}}$ (resp. $(a_L + b_L + 1) \bmod 2^{\frac{n}{2}}$) are the $\frac{n}{2}$ lowest bits of the binary representation of the sum $a_L + b_L$ (resp. $a_L + b_L + 1$) and $(a_L + b_L) \div 2^{\frac{n}{2}}$ (resp. $(a_L + b_L + 1) \div 2^{\frac{n}{2}}$) are the carries of $a_L + b_L$ (resp. $a_L + b_L + 1$) we get an arrangement of two $\frac{n}{2}$ bit conditional sum adders. The selection subcircuit for the leading parts of the sum can be constructed by a row of multiplexers which are controlled by the two carry values. This scheme is applied recursively to the two $\frac{n}{2}$ bit adders and leads to a splitting of the operands in very small parts. The most natural input sequence for the whole conditional sum adder therefore is a_{n-1},b_{n-1},...,a_i,b_i,...,a_0,b_0. With the same considerations the sequence of the output values of the adder are the bits of the sum plus one and the sum in alternating order ...,$(a+b+1)_i$, $(a+b)_i$,...,$(a+b+1)_0$,$(a+b)_0$.

Because these input and output sequences are not very intuitive for checking the correctness of our design we will specify two wiring subcircuits (sections 4.8.6,4.8.7, 4.8.8 and 4.8.9) for more convenient input/output behaviour of the conditional sum adder. If you only want to get a first glance at working with the graphical editor you can skip these sections. It is possible to read these sections later in order to complete the specification. The main part of the adder will be specified in sections 4.8.2,4.8.3,4.8.4 and 4.8.5 where the most important functions of the graphical editor are explained in detail.

4.8.2 General Equation for an n Bit Conditional Sum Adder

Opening a New Schematic

⇝ 12.1,12.2.1

We will now start with the recursive equation of an n bit conditional sum adder, for which we first have to open a new schematic sheet. This is done by selecting the entry `Load` from the submenu `-Schematics-`. You will see a table of the schematics in the directory given by the environment variable `DAGDIR`.

At this point of the session this table will only contain the entry labelled

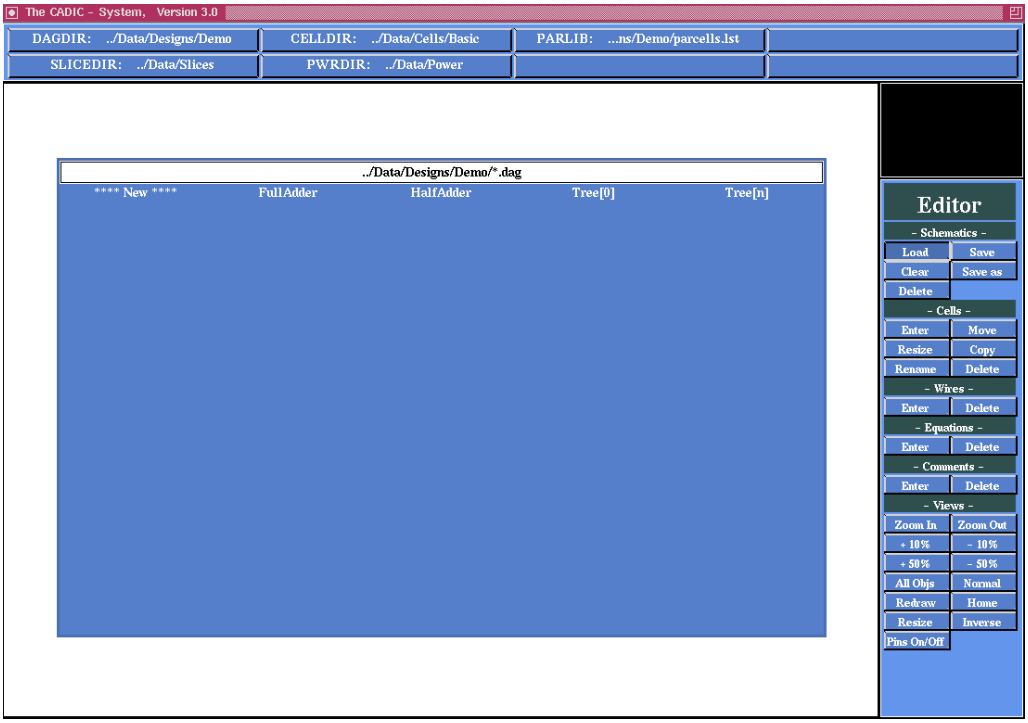


Figure 4.19: Dialog for opening a new schematic

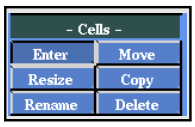
`**** New ****` beside the names of our designs from the previous sections as it is shown in figure 4.19. Select this entry by moving the pointer onto it and press the left mouse button as soon as it is highlighted. After that a text input window will appear on your screen and you have to specify the name of the new schematic. Type in `CSA[n]` and press the return key.

You have opened a new schematic which will first be given by an empty frame. Note, that the name in the upper left field of your environment has changed to `CSA[n]`. The name of the schematic is `CSA`. The square brackets introduce a list of parameters which can be used during the specification of the schematics. These parameters can appear in subcircuits or in the wiring. In our case we have a single parameter `n`, but in general you can use a list of up to 32 parameters.

Entering and Manipulating Cells

⇝ 12.1,12.3.1

In the next step we select the appropriate cells for the recursive specification



of the conditional sum adder and place them within the given frame. For this purpose select the entry `Enter` from the submenu `-Cells-` in the menuline. After picking this point with the left mouse button, you will see three new windows upon your working area. Each of these windows contains a list of cells of a certain type (c.f. figure 4.20).

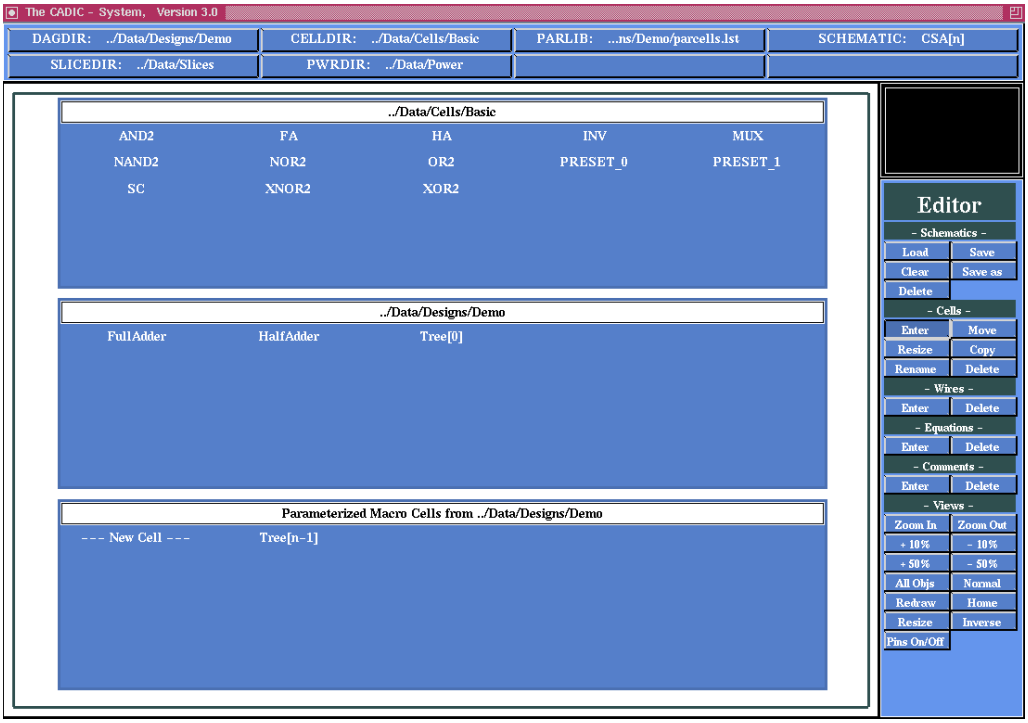


Figure 4.20: Dialogs for cell and macro selection

In the upmost window you see the names of the cells in the basic cell library which are located in the directory described by the environment variable `CELLDIR`. The second window shows the names of macro cells which have been defined by a previous input of a schematic. The system only shows such macros that are fixed, i.e. which do not depend on any formal parameters. This is the reason, why the macro `Tree[0]` appears in this list and the parameterized macro `Tree[n]` does not. In the third window the names of the parameterized macro cells are listed which have been used during the current session. Actually this list contains the entry labelled `--- New Cell ---` beside the previously specified parameterized macro `Tree[n-1]`. Select the first one in the same way as you have selected to open a new schematic

in order to define a new parameterized macro cell. After hitting the left mouse button you will be prompted to enter a new macro name. Type in `CSA[n/2]` to specify a macro for a conditional sum adder for $\frac{n}{2}$ bit operands.

Note that you can only use the parameters of the schematic to build arithmetic expressions for the parameters of the macro cells. Try to specify a cellname `CSA[k/2]` and the system will respond with the message

```
!  Error !  "CSA[k/2]" uses illegal parameter "k"
```

The syntax diagrams for the expressions allowed as parameters of these macro cells are given in chapter 12.

After the specification of a correct cellname you will see the graphical representation of the macro, i.e. a red bounded box with the cellname in its center. You can move this symbol inside the frame and place it at the desired position by clicking the left mouse button. The macro can only be placed on legal positions, i.e. there must not be an overlapping with other macros, wires or the border of the schematic. Try to move the macro onto the schematic frame and you will notice a change of its colour from red to grey. This change indicates the selection of an illegal position. Put the macro onto the position you see in figure 4.21.

The size of the macro is chosen by default, because the selected parameterized macro represents a whole class of macros. Its exact size can be determined, if the values for the parameters are known. For the abstract specification level of the editor the size of a cell or a macro is not important. In order to setup a comfortable schematic which is easy to understand, we will now resize the macro. This gives an impression of the relative sizes of the macros. Before we can resize the macro, we have to terminate the enter cell function. After you have placed the macro the three cell selection windows are popped up again in order to select another cell or macro. You can abort this function by pressing the right mouse button within one of the three windows.

⇝ 12.3.3

For the resize operation push the button `Resize` in the `-Cells-` submenu. Then you are in the cell selection mode and can pick the desired cell within

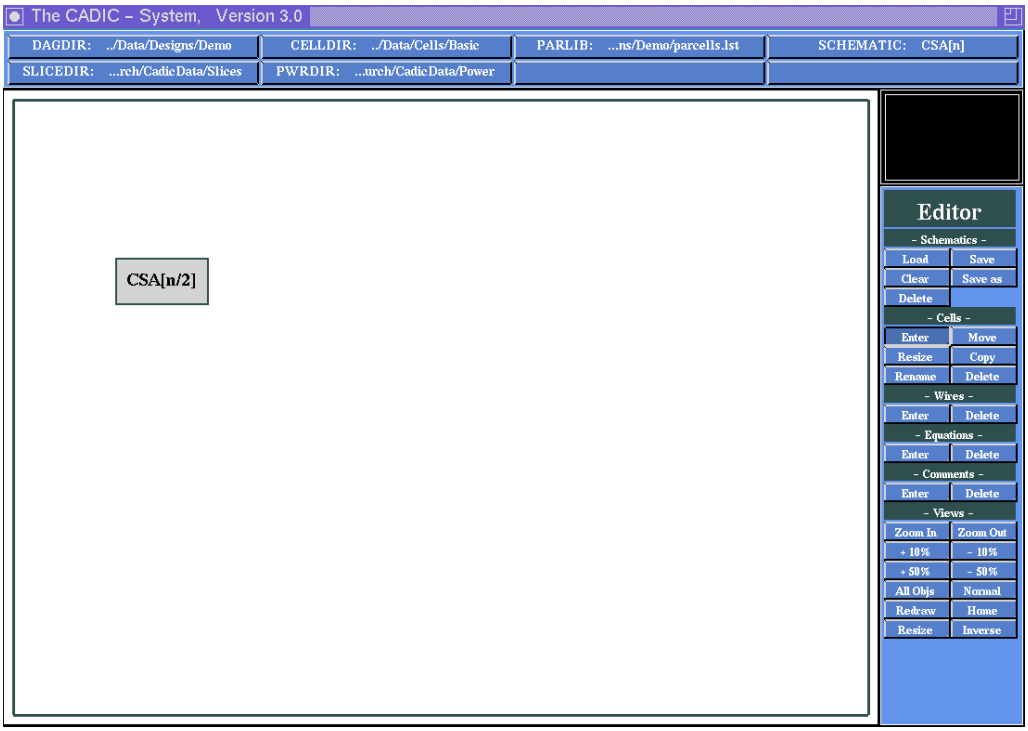
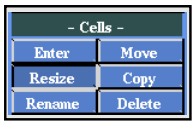


Figure 4.21: Placing a parameterized macro cell for the recursive specification of the conditional sum adder



the workarea. Move the pointer onto the workarea and you will see the macro being highlightet (its colour changes from grey to blue). In cell selection mode always the macro nearest to the pointer will be highlighted, you need not be inside it. At this point of our session cell selection is trivial because we only have a single macro within the workarea. The selection is done by clicking the left mouse button.

After the selection you can draw a rubberbanding frame which is anchored in the upper left corner of the macro and represents its new size. During the resize operation you need not hold down the left button. Just move the pointer to the desired position and confirm the new size by clicking the left mouse button or cancel the operation with the right mouse button (you will be returned to cell selection mode).

If you confirm a new size for the macro you will be in cell placement mode. The reason for this are possible overlappings of the resized macro with its

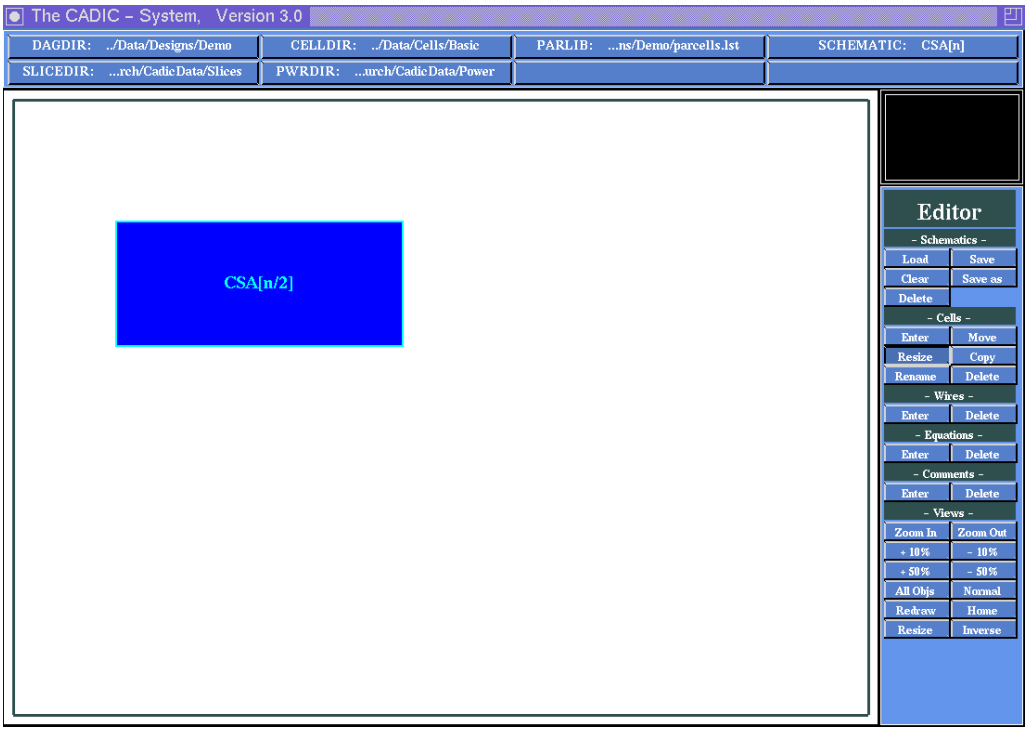
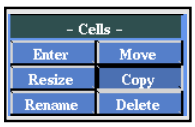


Figure 4.22: Resizing a parameterized macro cell

neighbours or with wires. You have to choose a new position for it which is only possible within the given constraints (i.e. no overlapping with macros, wires or the border). Move the resized macro to the previous position and confirm with the left mouse button. Now your schematic should look like that shown in figure 4.22.

⇝ 12.3.4



As described above the conditional sum adder simultaneously generates the sum and sum plus one for the upper and lower halfs of the operands. We now have positioned the macro `CSA[n/2]` for the upper half of the operands. A second instance of this macro will be needed for the lower half. We could create this instance by following the above steps (selecting a parameterized macro, placing and resizing it). But the editor offers a much simplier way to perform this operation. Push the entry `Copy` from the `-Cells-` submenu. This enables the cell selection mode where you can select an instance to be doubled. Press the left mouse button and you will get a second macro labelled `CSA[n/2]` which has the same size as our first macro.

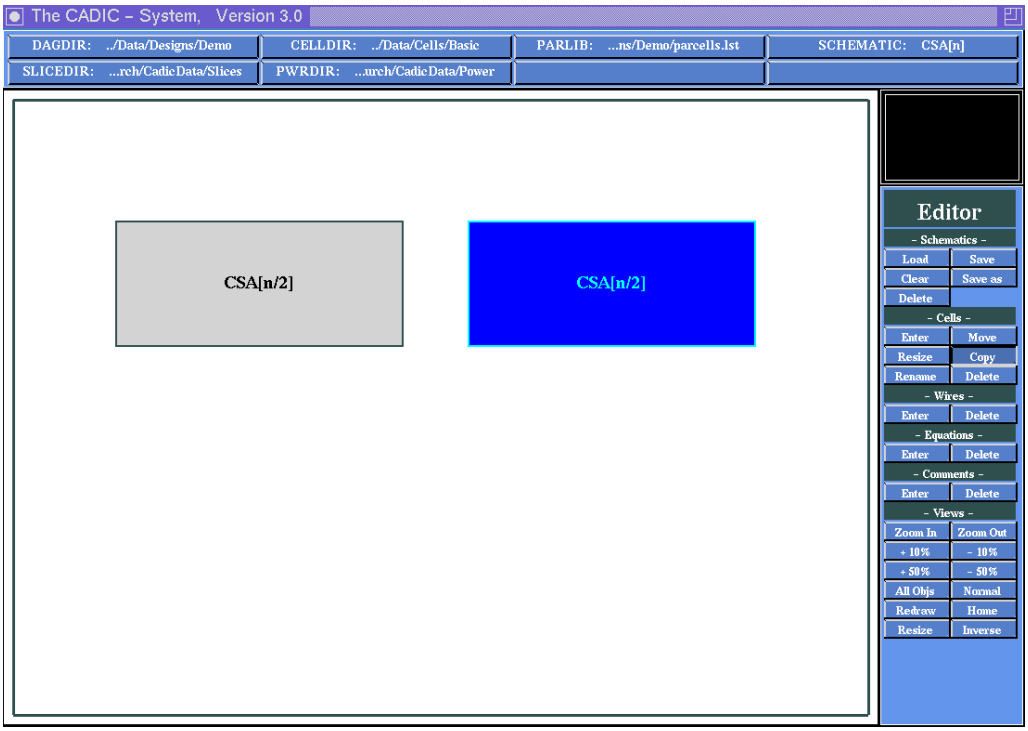
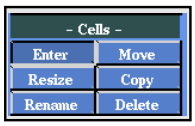


Figure 4.23: Creating a copy of an exisiting macro cell

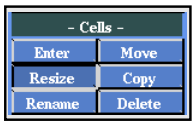
Place it within the workarea in order to have the schematic look like that in figure 4.23. During the placement operation you can watch the macro changing its colour, if it is placed on the first `CSA[n/2]` or on the border. In this case you cannot drop it, i.e. pressing the left mouse button has no effect. You can cancel the operation by pressing the right mouse button. The copy will disappear and you will return to cell selection mode. Pressing the right mouse button again will terminate the whole copy operation.

⇝ 12.3.1



In the next step we create an instance for the subcircuit which selects the sum or the sum plus one from the conditional sum adder for the upper halfs. This selection depends on the carry values of the addition of the lower halfs of the operands. This subcircuit will be represented by a new parameterized macro labelled `SEL[n/2+1]`. Create it in the same way as you have created the first `CSA[n/2]` macro and place it below the left $\frac{n}{2}$ bit conditional sum adder as it is shown in figure 4.24 (in this figure the macro is also resized as it will be described in the next paragraph).

⇝ 12.3.3



The width of this macro should be the same as that of `CSA[n/2]` which we achieve by an appropriate resize operation. The height of this macro should be smaller than the height of `CSA[n/2]` because it only will contain a line of multiplexers (the selection subcircuit must have constant depth as mentioned above). Now we have selected, resized and positioned the macros needed for the general recursive equation of the conditional sum adder. It is a recursive description because we need two instances of a macro which has the same name as the whole schematic (`CSA[n]` uses `CSA[n/2]`). Before we start to draw the wires between the macros, the schematic should look like that in figure 4.24.

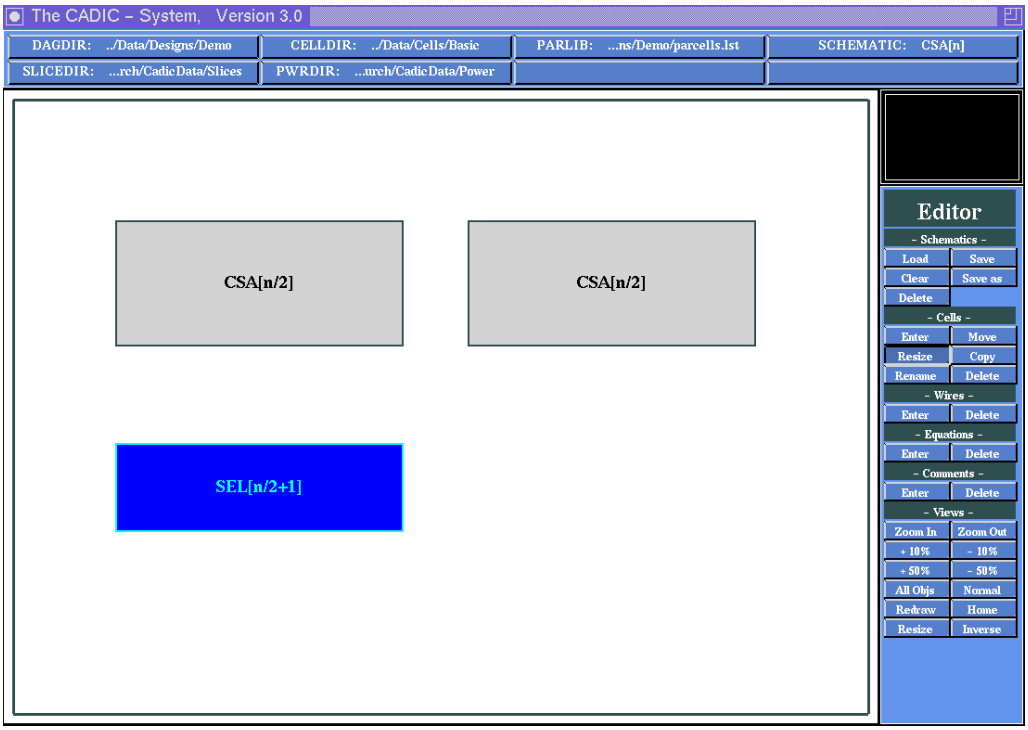


Figure 4.24: Parameterized macro cells for the n bit conditional sum adder

Entering Wires

⇝ 12.4.1

In this section we show how to connect the modules with wires. The main point you will observe during this procedure is that because of the parameterized description the lines drawn will have variable widths. The width of a line specifies the number of parallel wires contained in this bundle. We

start with the left `CSA[n/2]` which computes the sum and sum plus one of the upper half of the operands. First select the point `Enter` from the `-Wires-` submenu and move the pointer into the workarea. The cursor has changed to a crosshair and you can select the start point for a new wire. Move the pointer to the northern side of the left `CSA[n/2]` macro and position it to the middle of this side. If you move the pointer slowly toward the border of the macro you can observe that it snaps onto the border, if the distance becomes less than a certain tolerance value. This indicates that you have selected a special position within the schematic. The same behaviour can be noticed if you reach the frame, a pin of a cell or another wire. If the pointer has snapped to the border of the macro press the left mouse button to fix the start point of the new wire. Now move the pointer vertically towards the northern border of the schematic. You will see that it is followed by a grey rubberband line which indicates the position of the new wire. The cursor snaps to the border if the distance is small enough (your screen should look like that shown in figure 4.25).

Now you can fix the end point of the wire by pressing the left mouse button again. The grey rubberband line will disappear and you will be prompted to enter the width of this wire. As mentioned above the width of a wire represents the number of parallel lines within this bundle. In our case this wire describes the input line to the first conditional sum adder for the upper half of the operands. The width of a single operand is given by the parameter n. The left `CSA[n]` gets two operands of length $\frac{n}{2}$, i.e. the number of single wires is n. We describe this wire by a variable `@high[n]` which implicitly represents the two upper halfs of the operands on each stage of the recursion.

```
Please enter Wire Width:  <@high[n]      >
```

Type in `@high[n]` and confirm with the return key. You will see the new wire as a yellow line from the border of the `CSA[n]` to the border of the schematic. In the middle of the wire you can notice the description of its width annotated by a small diagonal line with the appropriate expression (c.f. figure 4.26).

Note that you can only use the parameters of the schematic to build arith-

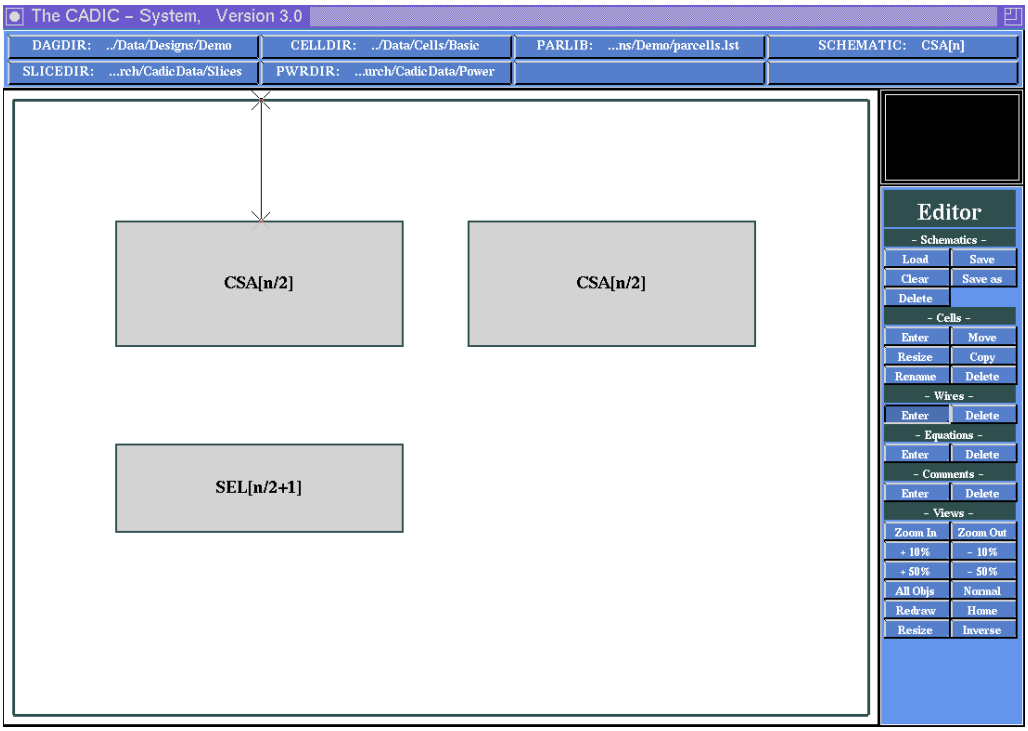


Figure 4.25: Connecting a module with the northern border of the schematic

metic expressions or wire variables. The exact syntax for these expressions is given by the diagrams in appendix.

We proceed with the other modules. The right instance of `CSA[n/2]` will also be connected with the northern border of the schematic. You can draw this wire in the same way as you did for the left `CSA[n/2]`. Type `@low[n]` for its width in order to denote that it describes the wires for the lower parts of the operands. Now we turn to the southern borders of the two `CSA[n/2]`. The left one must be connected with the northern border of the `SEL[n/2+1]`. Its width should describe the sum and sum plus one of the upper parts of the operands. Therefore we call it `@highsum[n]`. In this name we also have encoded the two carry bits for the higher part. A second wire with the width `@highsum[n]` has to be drawn from the southern border of `SEL[n/2+1]` to the southern border of the schematic. In the lower part we split the carry wires from the rest of the sums because we need the carries to control the selection subcircuit. First you draw a line from

the southern border of the right `CSA[n/2]` to the southern border of the schematic and call it `@lowsum[n]`. Next you draw a line which starts at the southern border of the right `CSA[n/2]` left of the line `@lowsum[n]` to the eastern side of the selection subcircuit `SEL[n/2+1]`. The width of this line will be denoted by `@carry` which describes the carry bits of the sum and sum plus one for the lower parts of the operands. You can directly connect the southern and the eastern side of the two macros. Note that the rubberbanding line will follow the movement of the pointer and there will be generated a knee when you change the motion direction from south to west.

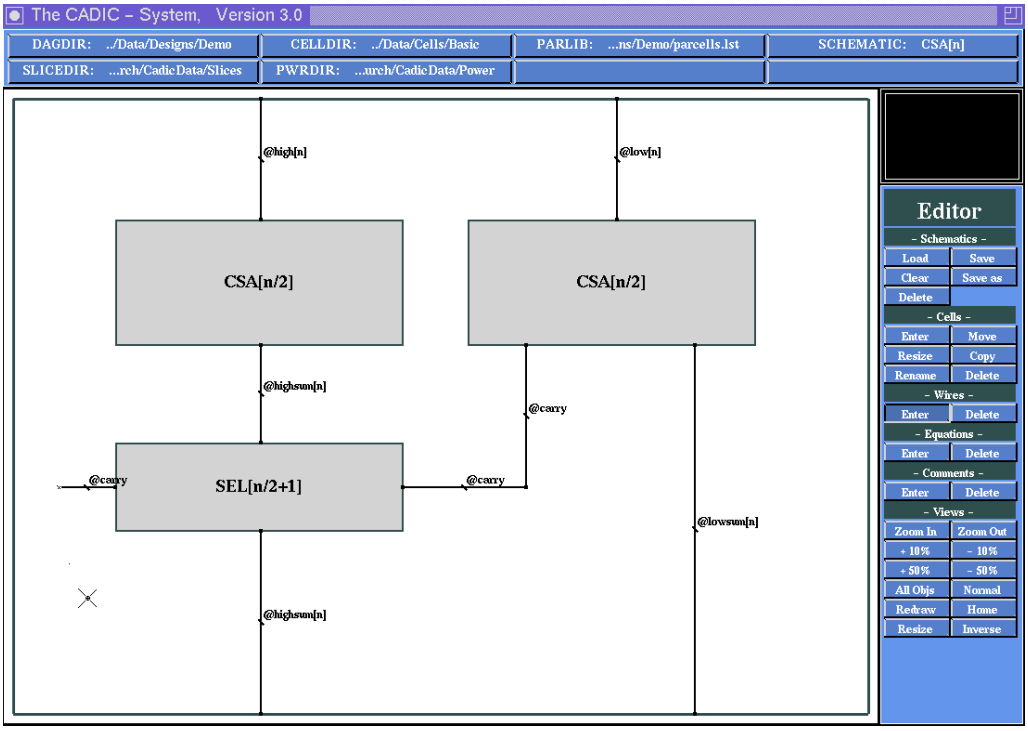


Figure 4.26: Final wiring for the conditional sum adder

The last wire we have to draw in this schematic starts at the western border of the selection subcircuit. It represents the carry lines from the `CSA[n/2]` for the lower halfs of the operands which are routed through the selection subcircuit. Because of the recursive specification of this subcircuit we have to connect a wire to its western border (c.f. section 4.8.4). This wire must not be connected to the western border of the schematic because of the

recursive definition of the conditional sum adder. This means that we have to cut the wire within the schematic. This is done by a so called projection cell which is a legal operation in the underlying calculus (c.f. section 4.2). The graphical notation for a projection cell is very simple. Just place the endpoint of the wire anywhere within the schematic and press the left mouse button. In our case we draw a small horizontal line towards the western border of the schematic as you can see it in figure 4.26.

To leave the enter wire mode press the right mouse button twice. The first button press will deactivate the selection of a wire end point and return to the selection of new start point. The second button press will abort the enter wire mode and return to the menu selection mode.

Entering Comments

⇝ 12.5.1

In order to document the schematic and especially the signals of the wiring you can place remarks at any position within the drawing. To do this select the entry `Enter` from the `-Comments-` submenu. Then you are prompted to enter some comment text. First we label the input wires of the schematic. For the left wire type in `a(n-1)b(n-1)...a(n/2)b(n/2)` and confirm this by pressing the return key. The comment text is now visible within the workarea and it follows the movement of the pointer. Move it near to the left input wire and fix its position by pressing the left mouse button. This causes the input window to popup again in order to enter another comment text. For the right input wire of the schematic type in `a(n/2-1)b(n/2-1)...a(0)b(0)` and place it at the appropriate position (c.f. figure 4.27).

Now we turn to the output signals of the schematic. The left wire, connected with the southern border of the schematic, will be labelled with the comment `c1Hc0Hs1(n-1)s0(n-1)...s1(n/2)s0(n/2)`. The right wire at the southern border is labelled with `s1(n-1)s0(n-1)...s1(0)s0(0)`. Finally the carry wire is marked with the comment text `c1Lc0L`. To terminate the enter comment mode simply press the return key within an empty input window. Your schematic should now look like that in figure 4.27.

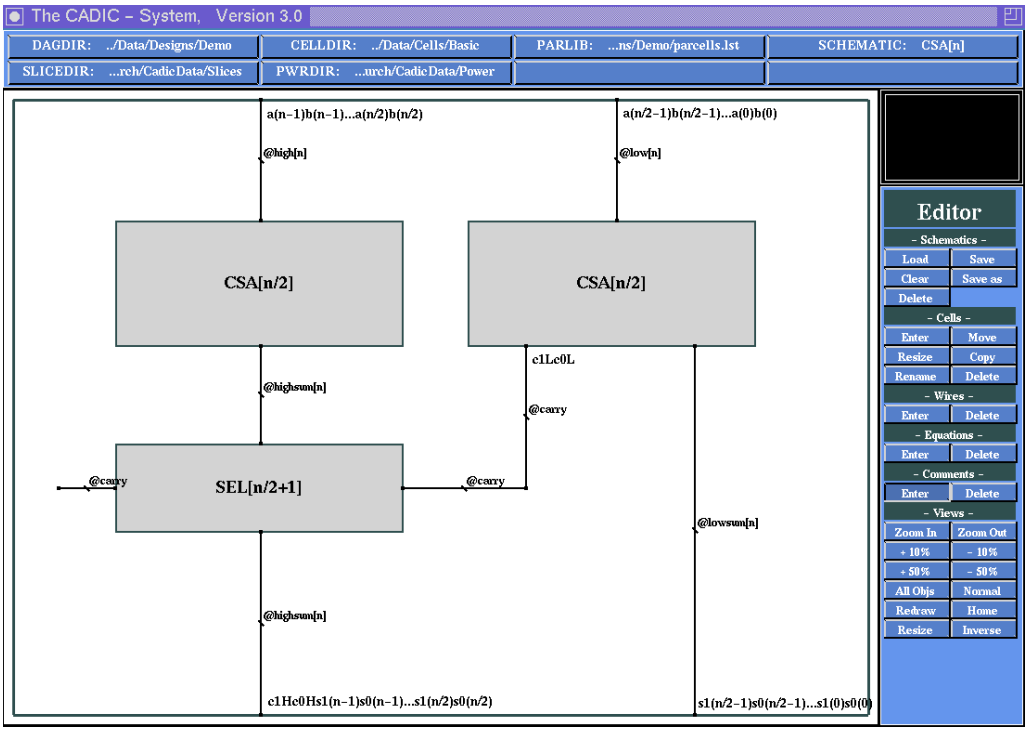


Figure 4.27: Comments placed at the input and output wires of `CSA[n]`

Note that the comment text are used to document the schematics. If you use the parameters of the schematic, the comment text will not be evaluated, if you fix the parameter values as it is shown in section 5.

Saving a Schematic

⇝ 12.2.2

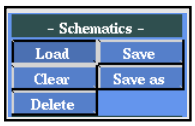
Now that we have drawn all elements of the general recursive equation for the n bit conditional sum adder we must save the schematic. This is simply done by selecting the entry `Save` from the `-Schematics-` submenu. The schematic will be saved to the file `CSA[n].dag` in the directory given by the environment variable `DAGDIR`.

4.8.3 Basic Equation for a 1 Bit Conditional Sum Adder

Opening a New Schematic

⇝ 12.2.1

After the specification of the general recursive equation for the conditional



sum adder we will now draw the schematic for the basic equation. This will be an adder for 1 bit operands. Therefore we call this schematic `CSA[1]` which will be opened following the steps from above. Note that the list of schematics now contains the entry `CSA[n]` beside the position of `**** New ****`. After selecting a new schematic you should type in the name `CSA[1]` and press return. Now we have created an empty frame for entering the 1 bit conditional sum adder.

Entering Basic Cells

The task of this adder is to compute the sum and the sum plus one of two single bits a_0 and b_0. The following table shows the values of this function:

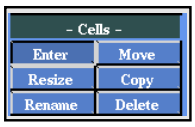
a_0	b_0	s_1^H	s_1^L	s_0^H	s_0^L
0	0	0	1	0	0
0	1	1	0	0	1
1	0	1	0	0	1
1	1	1	1	1	0

With $s_1 = (s_1^H, s_1^L) = a_0 + b_0 + 1$ and $s_0 = (s_0^H, s_0^L) = a_0 + b_0$ we can derive following boolean functions from this table

$$s_0^H = a_0 \wedge b_0, s_0^L = a_0 \oplus b_0, s_1^H = a_0 \vee b_0, s_1^L = \overline{s_0^L}.$$

We have to draw a schematic with two input signals (a_0, b_0) and four ouput signals $(s_1^H, s_0^H, s_1^L, s_0^L)$. The sequence of the output signals must have the given form because of the specification of the general equation for the conditional sum adder.

⇝ 12.3.1



First we select and place the needed basic cells. To do this select the entry `Enter` from the submenu `-Cells-`. This will open the three cell list windows (c.f. figure 4.7). Now we have to select the cells from the upmost window which contains the names of the available basic cells. These cells are located in the directory given by the environment variable `CELLDIR`.

The first cell we need is an OR gate with two inputs. Move the pointer to the name `OR2` and press the left mouse button if the name is highlighted.

The cell list windows will be closed and you see the shape of an OR gate in the workarea. The gate will follow the pointer motion within the workarea and you can pick the right position to place it. As shown above we need four gates to compute the output values of `CSA[1]`. The OR gate will be the leftmost (c.f. figure 4.28).

If you drop the gate by pressing the left mouse button, it is redrawn with two input pins at its northern border and an output pin at its southern border. The pins are labelled with the names `I1`, `I2` and `O1`. Small arrows indicate the directions of the pins.

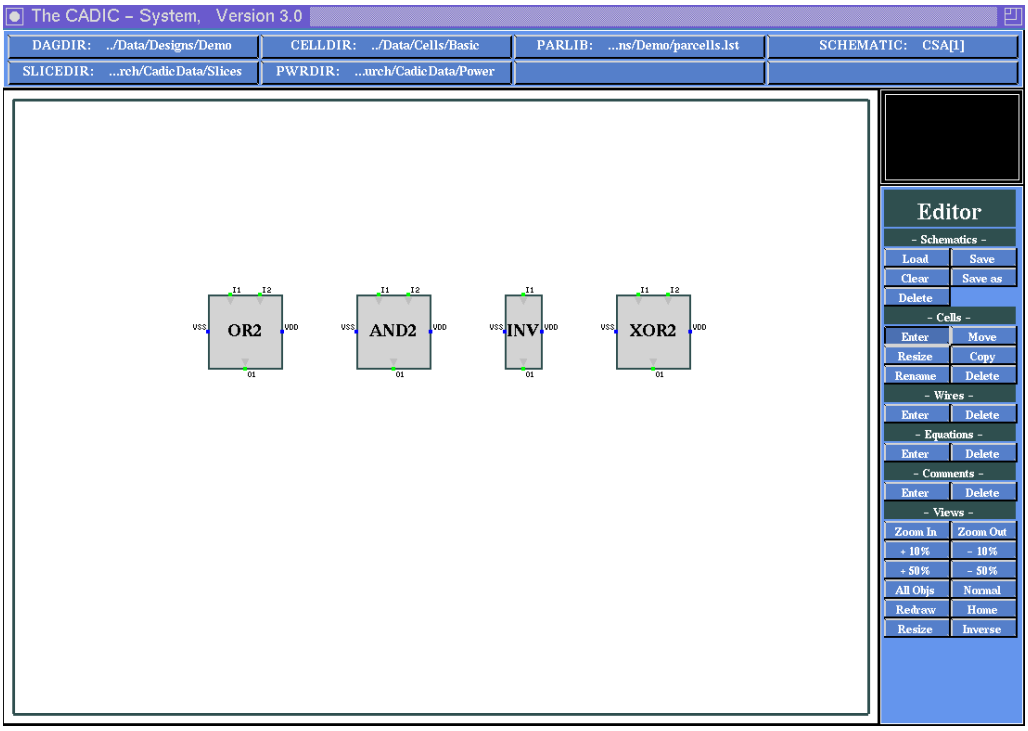


Figure 4.28: Placement of the basic cells for the 1 bit conditional sum adder

After the placement the cell list windows are popped up again in order to select another gate. Move the pointer to the entry `AND2` to get an AND gate with two inputs. We need this gate to compute the second value s_0^H, so place it right beside the OR gate. In the same way you should select an Inverter (`INV`) and an EXOR gate for two inputs (`XOR2`) and place them according to figure 4.28). After the placement of the last gate you can abort

the cell selection mode by pressing the right mouse button.

Entering Wires

↝ 12.4.1

To draw the wiring of `CSA[1]` select the entry `Enter` from the `-Wires-` submenu. First we connect the output pins of the gates directly with the southern border of the schematic. Move the pointer near to the output pin of the `OR2` gate. If the distance to the pin is less than a certain tolerance value the crosshair will snap to the pin and you can fix the start point of the wire by pressing the left mouse button. Move vertically to the southern border of the schematic and fix the end point in the same way.

When you drew the wires in the general equation `CSA[n]` you have been prompted to enter the width of the wire (e.g. `@carry`). In the case of connecting a pin of a basic cell this is not necessary because the wire represents a single binary value. The width is automatically set to 1.

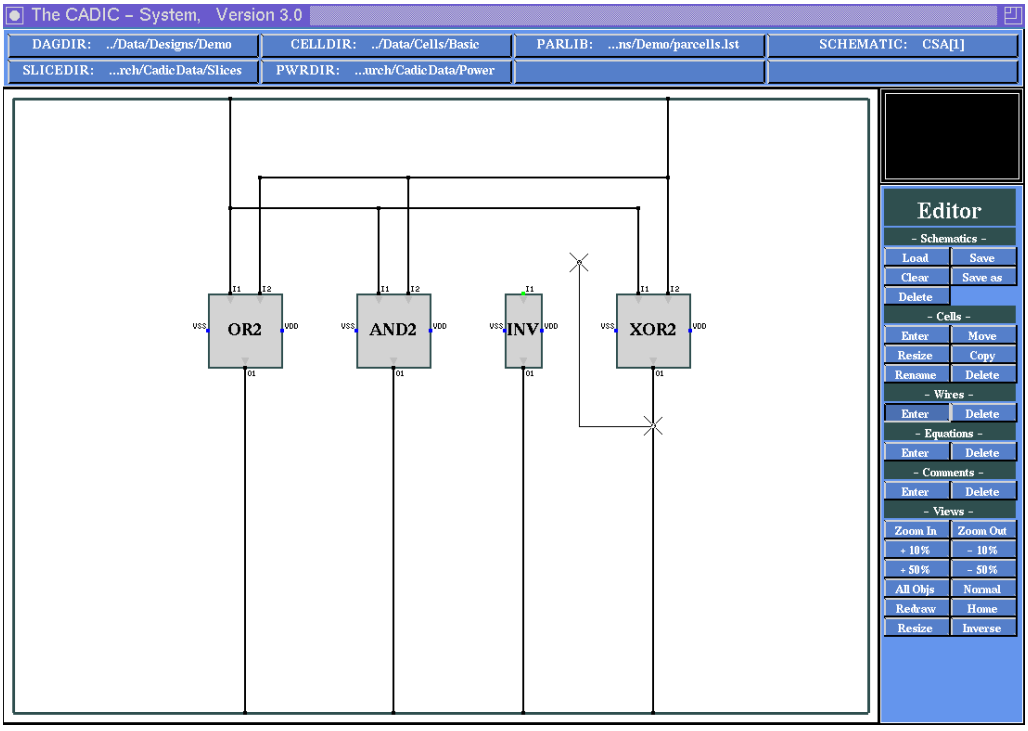


Figure 4.29: Wiring for the 1 bit conditional sum adder

After connecting the last three output pins of the gates with the southern

border of the schematic we turn to the input pins. Connect the left input (`I1`) of the `OR2` gate as well as the right input (`I2`) of the `XOR2` gate directly with the northern border of the schematic. These two wires represent the inputs a_0 and b_0 of the 1 bit adder. As shown above we have to feed these values into the `OR2`, `AND2` and `XOR2` gate. To do this we branch the wires and connect them with the appropriate input pins. To do this move the pointer onto the left wire. You can notice that it snaps to the wire if the distance is less than a certain tolerance value. Press the left mouse button to fix the start point and move the pointer to the left input pin of the `XOR2` gate. The connection is indicated by a rubberbanding line between the start point and the current position of the pointer. If you select an illegal position the rubberbanding line will disappear. This is the case if the chosen connection intersects a cell or moves along an already existing wire or the border of the schematic (try this by moving the pointer into the `AND2` gate). If you have setup the right connection to the left input of the `XOR2` gate press the left mouse button to fix the end point of the wire. The yellow dot at the start point of the new wire indicates a branching of the previous vertical wire. Implicitly the branch node is considered to be a basic operation in the underlying calculus just as the projection node mentioned above. In this special case we have entered an east–branch.

In the same way you can connect the right vertical wire with the right input pin of the `OR2` gate. Both new wires still have to be connected with the input pins of the `AND2` gate. This can be done by two simple vertical wires. Here it is important that you select the input pin of the `AND2` gate as the start point and move the pointer vertically toward the appropriate wire creating a new branch (south–branch). If you fix the start point on the horizontal wire you might miss the right position for a vertical connection. The position of the pin implies the position of the wire, so selecting it first is often very helpful. Now complete the wiring of `CSA[1]` by connecting the wire from the output of the `XOR2` gate with the input pin of the Inverter. As shown above the `XOR2` gate computes the value s_0^L and the value $s_1^L = \overline{s_0^L}$ is just the inverted signal. This connection has to be entered in two steps. First fix the start point of the wire at the output wire of the `XOR2` gate and

move the pointer upwards between the `INV` and the `XOR2` as it is indicated in figure 4.29. At this position fix the end point of this wire segment. From this end point move the pointer to the pin connector of the input of `INV`. The previous end point serves as the start point for the next wire segment which you can fix now by pressing the left mouse button. You should have entered a schematic similar to that shown in figure 4.30. Terminate the enter wire mode by pressing the right mouse button two times.

Entering Comments

⇝ 12.5.1

Before we save the schematic we want to document it by some comments in order to later remember the meaning of the different signals. For this purpose select the entry `Enter` from the `-Comments-` submenu. Now you are prompted to enter the comment text. First we will describe the input signals of the schematic. The left one represents the value a_0, so type in `a0` and press the return key. You will see the comment text in the upper left corner of the workarea. The text is connected to the pointer and will follow its movement within the workarea. Move it near to the connection of the left input signal with the northern border of the schematic and place it on the right side of the wire by pressing the left mouse button. After the placement of the comment the text field window is popped up again and you can enter the next comment text. For the second input wire type in `b0` and move it to the analogous position.

Now we turn to the output signals of `CSA[1]`. The sequence from left to right is (s_1^H, s_0^H, s_1^L, s_0^L). That is why we place the remaining four comments `s1H`, `s0H`, `s1L` and `s0L` near to the points where the wires are connected to the southern border of the schematic. After the placement of the last comment you can abort the loop of entering a comment text by simply pressing the return key to an empty input window. The schematic `CSA[1]` should now look like that shown in figure 4.30.

Saving the Schematic

⇝ 12.2.2

Save the schematic by selecting the entry `Save` from the `-Schematics-`

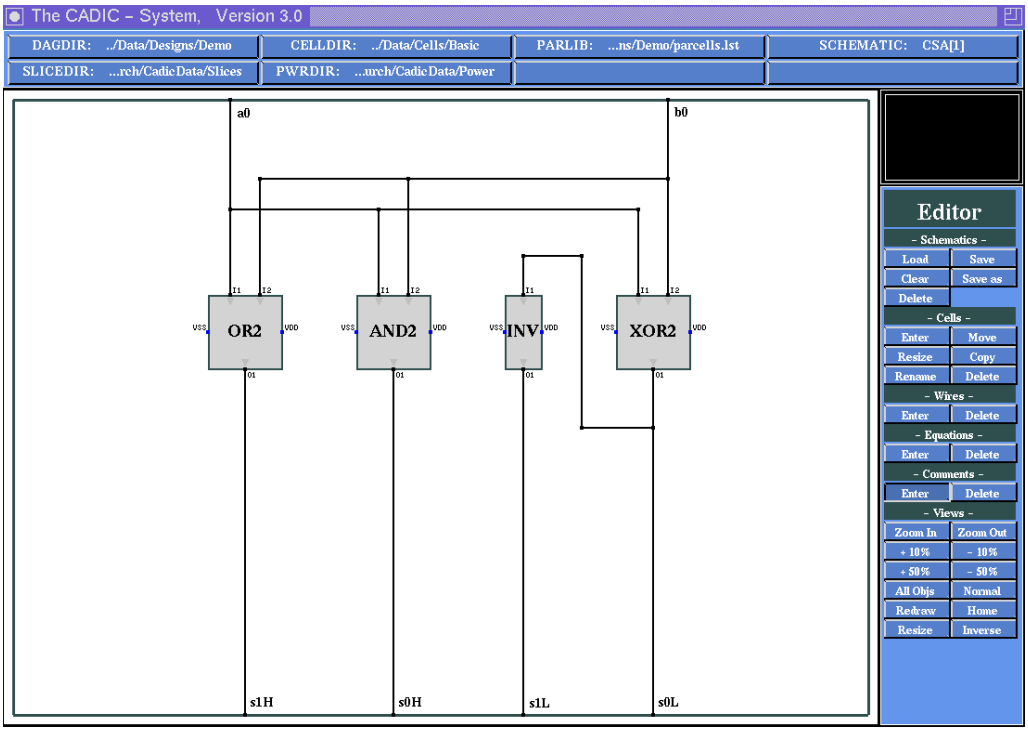


Figure 4.30: Remarks for the input and output signals of `CSA[1]`

submenu. The graphical input is then written to the file `CSA[1].dag` in the directory given by the environment variable `DAGDIR`.

4.8.4 General Equation for the Selection Subcircuit

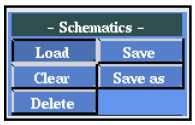
At this point of the session we have entered the general and the basic equation for the recursive description of an n bit conditional sum adder. In the general equation `CSA[n]` we used a parameterized subcircuit `SEL[n/2+1]` for the selection of either the sum or the sum plus one of the upper halfs of the operands. This selection circuit is controlled by the values of the carry bits of the sum of the lower half. In order to generate an adder with logarithmic depth the selection circuit must have constant depth. In the general equation of the conditional sum adder we used an instance of the selection subcircuit in advance, i.e. before we have defined this circuit.

We will construct this subcircuit by a row of multiplexers. The generation of a row of k elements is a basic recursive technique which is used in a

lot of parameterized designs. If the number of elements to be generated is a power of two the recursion is even more simple than in our case where we generate an arbitrary number k ($k > 0$) of elements. We need two schematic inputs for the specification, one for a general and one for a basic equation. The principle is a simple divide and conquer technique, i.e. a row of k elements consists of a row of $\lceil \frac{k}{2} \rceil$ elements followed by a row of $\lfloor \frac{k}{2} \rfloor$ elements. Here we use the reduction $k \longrightarrow \lceil \frac{k}{2} \rceil + \lfloor \frac{k}{2} \rfloor$ which leads to a structure of logarithmic size as we will show it in chapter 5. You could also choose the form $k \longrightarrow (k-1)+1$ which would result in a structure of linear size.

Opening a New Schematic

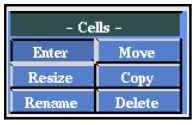
⇝ 12.2.1



An important feature of the graphical editor of the CADIC system is that such theoretical considerations can directly be used to set up an appropriate schematic. The general equation for the row of k multiplexers is called `SEL[k]`. You should open a new schematic now (select the entry `Load` from the `-Schematics-` submenu and `**** New ****` in the schematic list, then type in the appropriate name).

Entering and Manipulating Parameterized Macros

⇝ 12.3.1

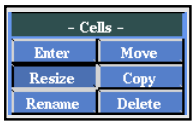


In the empty frame we first place a new parameterized macro cell. Select the entry `Enter` from the `-Cells-` submenu to popup the cell selection windows. In the third window which contains the names of parameterized macro cells, move the pointer onto the `--- New Cell ---` entry and press the left mouse button if it is highlighted. In the following input line type `SEL[upper(k/2)]` for the name of the new macro cell. The function call `upper(k/2)` computes the value $\lceil \frac{k}{2} \rceil$ (for more information about standard function calls see section 12.1). You get a new parameterized macro with the default size. This macro can be moved around within the workarea. Place it near to the western border of the schematic and press the left mouse button. After that the cell selection windows are popped up again. We will generate the second instance from this first one, so you can abort

⇝ 12.1

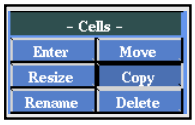
the cell selection mode by pressing the right mouse button within one of the three list windows.

⇝ 12.3.3



Before we create a copy of this instance we resize it to a convenient size. Select the entry `Resize` from the `-Cells-` submenu. You are in cell selection mode when you move the pointer into the workarea. Notice that the only instance `SEL[upper(k/2)]` is highlighted in blue. Select it by pressing the left mouse button. After that you can resize it in the same way as we did during the specification of `CSA[n]` for the first instance `CSA[n/2]` (c.f. page 62). The pointer motion is coupled with the drawing of a rubberbanding rectangle which indicates the new size of the macro cell. After confirming the new size with the left mouse button you are in the cell placement mode. Move the macro to its previous position and press the left mouse button again. You are returned to the cell selection mode for further resize operations. Abort this mode by pressing the right mouse button within the workarea.

⇝ 12.3.4



Now we create a copy of this instance for the second part which contains the last $\lfloor \frac{k}{2} \rfloor$ elements. If you wonder why we take an instance with the same name, you will later see that its label can be changed to the correct value. Select the entry `Copy` from the `-Cells-` submenu and move the pointer into the workarea to choose the instance `SEL[upper(k/2)]`. If you press the left mouse button you get a second instance of this name with the same size like the first one. Now the cell placement mode is active and you can move the new instance right beside the first `SEL[upper(k/2)]`. Fix its position by pressing the left mouse button, then abort the cell selection mode by clicking with the right mouse button.

⇝ 12.3.5

As shown above we will recursively construct a row of k elements by the sequence of $\lceil \frac{k}{2} \rceil$ and $\lfloor \frac{k}{2} \rfloor$ elements. For the second instance we have to change its parameter from `upper(k/2)` to `lower(k/2)` (the function call `lower(x)` computes the value $\lfloor x \rfloor$, c.f. section 12.1). Now we change the name of the instance in this way. For this purpose select the entry `Rename` from the `-Cells-` submenu. Select the right `SEL[upper(k/2)]` by moving the pointer near to it and press the left mouse button when the instance is highlighted. The text input window will popup on the screen and you

⇝ 12.1

can type in `SEL[lower(k/2)]` as its new name. Note that you have to type in the complete name, it is not possible to exchange only the expressions for the parameters. If the typed name is correct press the return key and now the selected instance has got the new name. The system returns to the cell selection mode in order to let you change the label of another macro. This mode can be aborted by pressing the right mouse button within the workarea.

Entering Wires

⇝ 12.4.1

Now we turn to the wiring of `SEL[k]` which is very simple. As you can see in the schematic for `CSA[n]` (c.f. figure 4.26) the macro `SEL[n/2+1]` receives at its eastern border the wires representing the carry bits of the sum and the sum plus one of the lower halfs of the operands. This wire has to be connected to all the multiplexers in the row. Therefore we first connect the eastern border of the right instance `SEL[lower(k/2)]` with the eastern border of the schematic. Then we connect the eastern border of the left instance `SEL[upper(k/2)]` with the western border of the right instance. Finally the western border of the left instance is connected with the western border of the schematic. To enter these wires select the entry `Enter` from the `-Wires-` submenu and move the pointer into the workarea. Fix the start point in the middle of the eastern border of the `SEL[lower(k/2)]` with the left mouse button and move the pointer horizontally onto the eastern border of the schematic. If you fix the end point of the wire the input window pops up on the screen. Type in `@carry` to denote that this wire represents the carry values. Now select a new start point for the next wire on the eastern border of the left instance and connect it with the western border of the right instance. At the text input window type in `@carry` to express that this wire represents the continuation of the carry wires which are fed through the `SEL[lower(k/2)]`. In the same way this wire is routed through the left instance, i.e. connect the western border of `SEL[upper(k/2)]` with the western border of the schematic and call it `@carry` as well. Now your schematic should look similar to the one shown in figure 4.31.

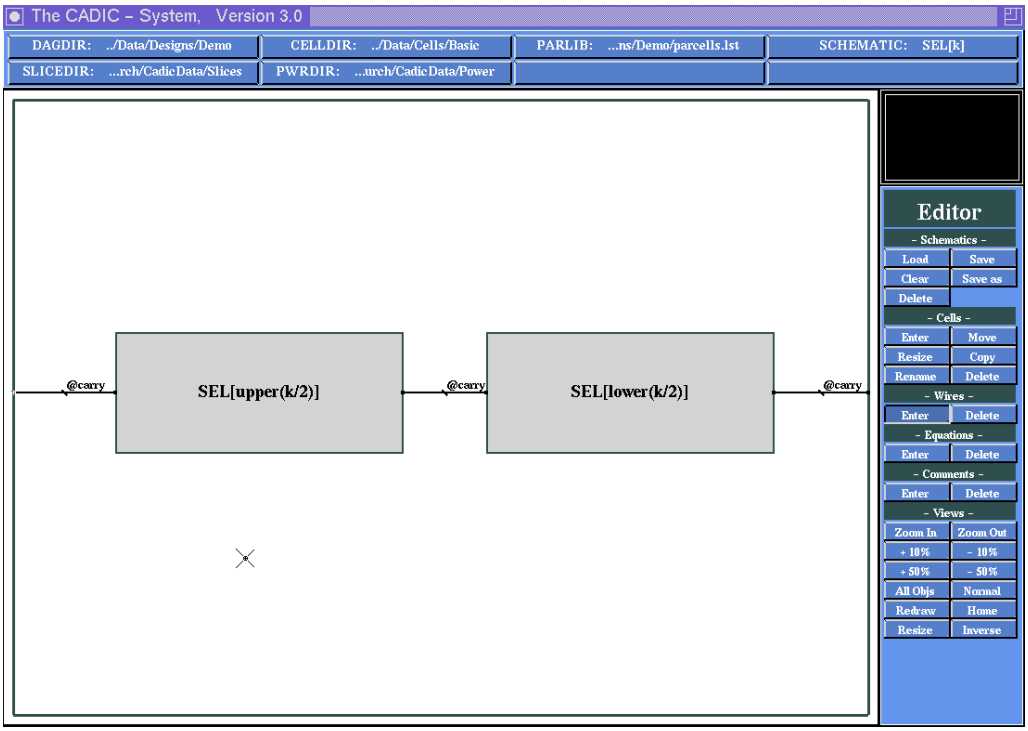


Figure 4.31: Feed through for the carry wires

Now we have drawn the carry wires which control the outputs of the multiplexers. To complete the wiring of `SEL[k]` we draw the data input and output wires. As you can see in the schematic for `CSA[n]` (c.f. figure 4.26) the inputs (outputs) of the selection subcircuit `SEL[n/2+1]` are connected to its northern (southern) border. These wires represent the sum and the sum plus one of the upper half of the operands in alternating sequence. First we draw a wire from the middle of the northern border of the left instance to the northern border of the schematic. Type in `@selinL[k]` for the name of this wire. We choose this name to denote that it represents the left (L) part of the input wires (in) to the selection subcircuit (sel). The number of single binary values represented by this wire depends on the value of the parameter k. This is the reason why we also use this parameter in the name of the wire variable.

Next we connect the southern border of `SEL[upper(k/2)]` with the southern border of the schematic by a wire named `@seloutL[k]`. In the same way

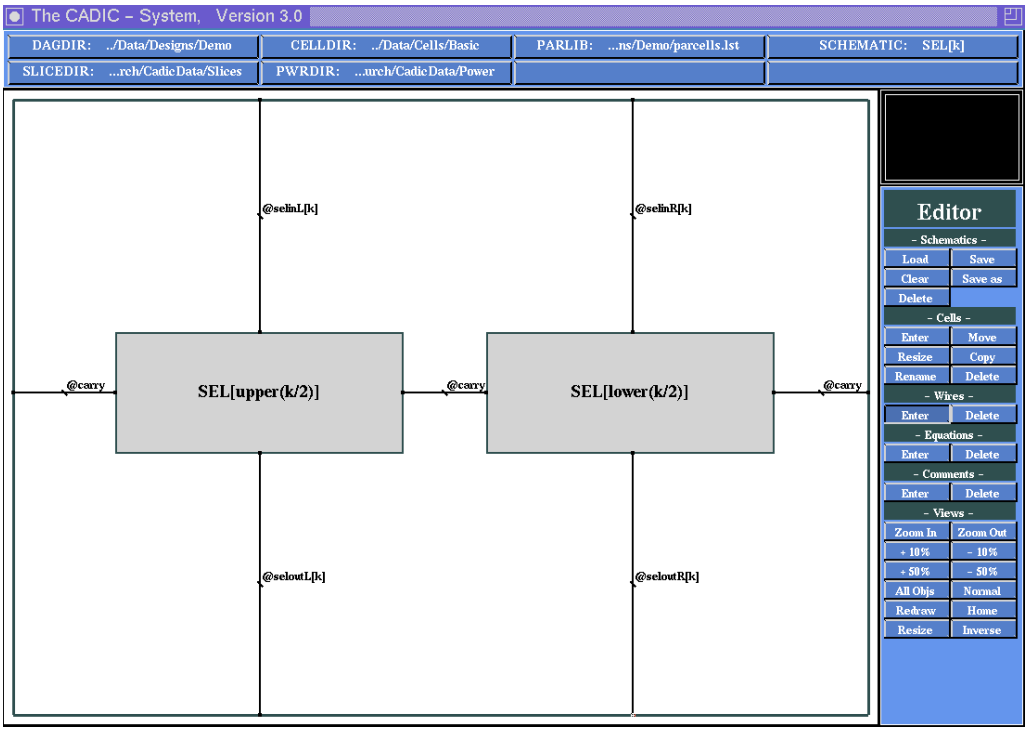


Figure 4.32: Final wiring for the selection subcircuit

we connect the right instance `SEL[lower(k/2)]` with the northern and the southern border of the schematic. These two wires are denoted by the names `@selinR[k]` (northern border) and `@seloutR[k]` (southern border) to indicate that they represent the inputs and outputs of the right part of the selection subcircuit. After you have drawn the last wire you can abort the enter wire mode by pressing the right mouse button twice within the workarea.

Entering Comments

⇝ 12.5.1

Now we will place comments to the input and output signals of `SEL[k]`. To do this select the entry `Enter` from the `-Comments-` submenu. Our first comment concerns the input wires at the northern border of the schematic. At the following popup input window type the text

```
sum and sum+1 in alternating order
```

and press the return key. After that you can move the comment text within the workarea. Place it between the two input wires near to the northern border of the schematic and confirm the position with the left mouse button.

Next we place a comment at the output wires at the southern border of the schematic. The popup window appears automatically after you placed the first comment. At its prompt type in

```
selected sum or sum+1
```

and press the return key. The text is now movable inside the workarea. Place it between the two output wires by pressing the left mouse button.

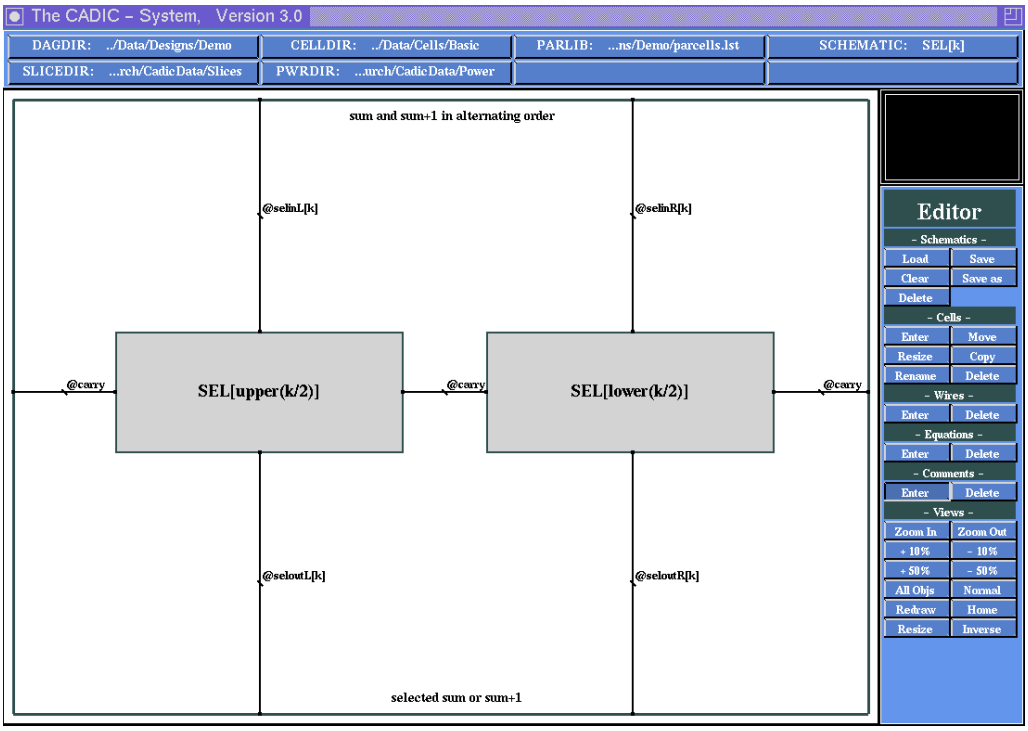


Figure 4.33: Comments for the input and output signals of `SEL[k]`

We will place a last comment at the inputs at the eastern border of the schematic. These control signals represent the carry values of the sum plus one and the sum of the addition of the lower operand parts. That is why we enter the comment `c1Lc0L` for this wire. `c1(0)L` stands for carry (`c`) of the sum plus one (sum) (`1 (0)`) of the addition of the lower (`L`) parts of

the operands. After pressing the return key move this comment near to the corresponding wire at the eastern border of the schematic where you can place it with the left mouse button.

Finally you can abort the enter comment mode by simply pressing the return key in the empty input window. You return to the menu selection mode.

Saving the Schematic

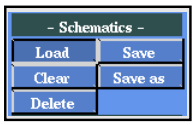
⇝ 12.2.2

If you have drawn a schematic similar to that in figure 4.33 you can save it by selecting the entry `Save` from the `-Schematics-` submenu. CADIC will put the file `SEL[k].dag` into `DAGDIR`.

4.8.5 Basic Equation for the Selection Subcircuit

Opening a New Schematic

⇝ 12.2.1



To complete the specification of the n bit conditional sum adder we have to draw a schematic for the basic equation of the selection subcircuit. Open a new schematic `SEL[1]` by choosing the `**** New ****` entry in the schematic list which will be displayed if you select the entry `Load` from the `-Schematics-` submenu.

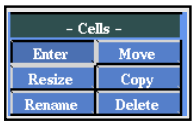
Entering and Manipulating Basic Cells

The basic element of the selection subcircuit has two input bits s_1, s_0 from the sum plus one and from the sum of the corresponding part of the operands. Between these two bits a selection has to be made according to the values of the carry bits c_1 and c_0. Both carries come from the sums of the lower part of the operands. c_1 is the carry of the sum plus one, c_0 is the carry of the sum. The selected values are the outputs a_1 and a_0 of the elementary selection macro. It holds:

$$a_1 \leftarrow (c_1 \wedge s_1) \vee (\overline{c_1} \wedge s_0), a_0 \leftarrow (c_0 \wedge s_1) \vee (\overline{c_0} \wedge s_0).$$

To perform this operations we need two multiplexers from the basic cell library. For this purpose select the entry `Enter` from the `-Cells-` submenu

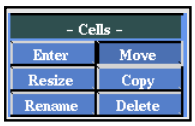
⇝ 12.3.1



. Move the pointer to the entry `MUX` within the basic cell window (upmost) and press the left mouse button when the name is highlighted. Place the cell in the left half of the schematic as shown in figure 4.34. A second multiplexer can directly be selected from the popped up cell selection windows and can be moved to the right half of the schematic. After placing it abort the cell selection by pressing the right mouse button within one of the cell list windows.

The multiplexer has three input pins `I1`, `I2` and `I3` at its northern border and one output pin `O1` at its southern border. `I3` is the selection input with the following functionality:

$$\texttt{O1} \longleftarrow \begin{cases} \texttt{I1} & \text{if } \texttt{I3} = 0 \\ \texttt{I2} & \text{if } \texttt{I3} = 1 \end{cases}$$



⇝ 12.3.2

Before we draw the wiring of `SEL[1]` we change the orientation of the left multiplexer in order to minimize the number of wire crossings. Another reason for this action is to demonstrate how you can change the orientation of an instance. Select the entry `Move` from the `-Cells-` submenu . Although this entry is called `Move` it contains the movement and rotating and flipping of instances. Move the pointer near to the left multiplexer and press the left mouse button when it is highlighted. After that you are in the cell placement mode, i.e. you can select a new position for the cell. But at the same time you can change its orientation by pressing the middle mouse button several time. Each time you press it the instance changes to one of eight possible orientations. The current orientation is indicated by a suffix to the cellname. The cell is in normal orientation if there is no suffix to its name. Pressing the middle mouse button for the first time the cell is turned clockwise by 90 degree, indicated by the suffix `(90->)`.

button presses	suffix	meaning
0		normal orientation
1	(90->)	turned clockwise 90^o
2	(180)	turned 180^o
3	(<-90)	turned counterclockwise 90^o
4	(^\v)	flipped at the horizontal axis
5	(<-90^\v)	turned counterclockwise 90^o, then flipped at the horizontal axis
6	(<\>)	flipped at the vertical axis
7	(<-90<\>)	turned counterclockwise 90^o, then flipped at the vertical axis
$8 \equiv 0$		normal orientation

In our case we want to flip the multiplexer at its vertical axis in order to invert the sequence of its input pins. According to the tabular given above you get this orientation by pressing the middle mouse button six times. During this operation you are in the cell placement mode and the instance will follow any movement of the pointer within the workarea. If you have chosen the correct orientation you can move the instance to its previous position and place it by pressing the left mouse button. If you made a mistake in the selection of the orientation you can abort the operation by pressing the right mouse button. Then the cell is returned to its original position and orientation and you will be in cell selection mode again to restart with the operation.

After placing the instance you are in the cell selection mode again in order to change the orientation of another cell. You can abort this mode by pressing the right mouse button within the workarea.

Entering Wires

⇝ 12.4.1

Now we begin with the wiring of `SEL[1]`. Select the entry `Enter` from the `-Wires-` submenu. First we connect the output pins of the multiplexers with the southern border of the schematic. Move the pointer near to the pin at the southern border of the left multiplexer and press the left mouse

button when the crosshair cursor snaps to the pin. Now move vertically down towards the schematic border drawing the rubberbanding line and fix the end point of the wire. Note that you do not have to enter the width of the wire. Every wire which is connected to a pin of a basic cell is automatically given the width 1. In the same way we connect the output pin of the right multiplexer with the southern border of the schematic.

It has been shown above that the value s_1 has to be assigned to the outputs if c_1 or c_0 have the value 1. Respectively the value s_0 is the output if c_1 or c_0 are 0. From the description of the multiplexer it follows that we have to connect both inputs `I1` with the schematic input s_1 and both inputs `I2` with s_0. The selection input `I3` of the left multiplexer has to be connected with c_1 and the selection input of the right multiplexer with c_0. Fix the start point of the next wire at the input pin `I1` of the left multiplexer. Move vertically to the northern border of the schematic and fix the end point of this wire. Now select the input pin `I1` of the right multiplexer move a little bit upwards and then horizontally to the just drawn wire and fix the end point on it, creating an east–branch. To draw the next wire you first have to press the right mouse button once, because we do not want to use the end point of this wire as the new start point. After that we select the input pin `I2` of the right multiplexer and draw a vertical connection with the northern schematic border. From the input pin `I2` of the left multiplexer we draw a connection to the wire starting at the input pin `I2` of the right multiplexer and create a west–branch on this wire. After fixing the end point you must press the right mouse button once in order to return to the selection of a new start point.

Finally we have to enter the wires for the control signals which have to be connected with the `I3` input pins of the multiplexers. It has been shown above that these wires have to be fed through the whole selection subcircuit because they are used to control all the multiplexers in the row. For this purpose we first draw the wires from the eastern to the western border of the schematic. Fix the start point of the first wire (c_1) on the eastern border. Leave enough space to enter the second wire between this one and

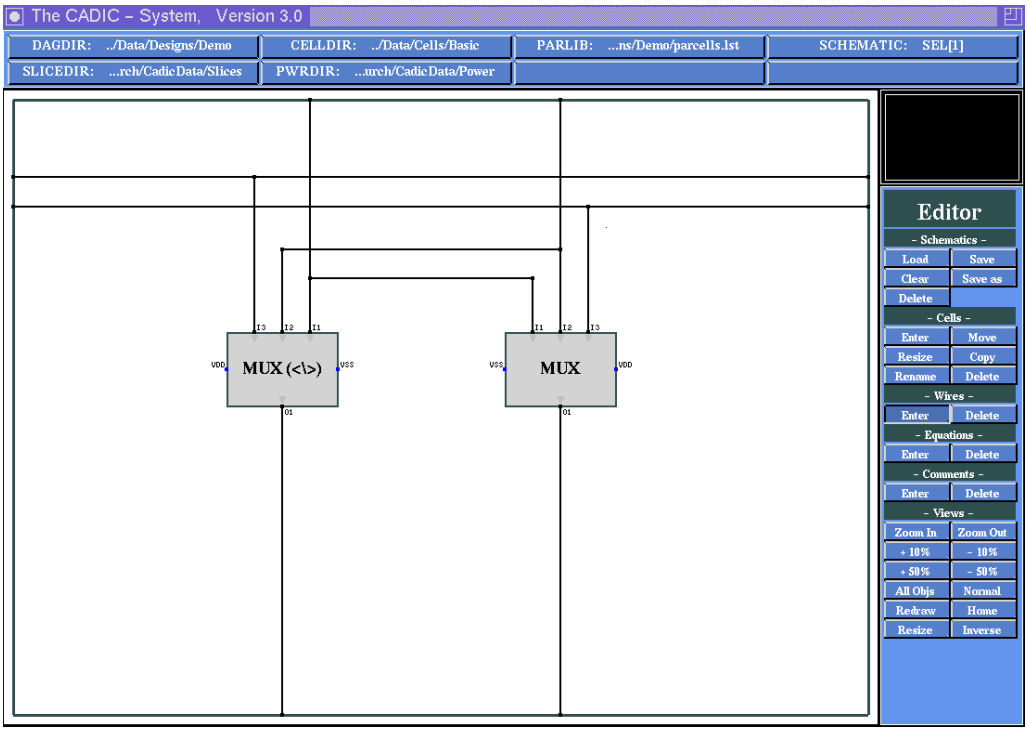


Figure 4.34: Final wiring of `SEL[1]`

the horizontal parts of the two input signals. Move the pointer horizontally to the western border and fix the end point. At the following input window type in `1` as the width of this new wire.

Now you draw a second connection from the eastern to the western border for the carry signal c_0 below the first one (type in `1` to the input window as the width of this wire).

The connection to the inputs `I3` of the multiplexers is done by first selecting the pin and then moving vertically to the appropriate wire (the left (right) multiplexer has to be connected with the upper (lower) wire) creating a south–branch. Compare your schematic with that shown in figure 4.34.

Entering Comments

⇝ 12.5.1

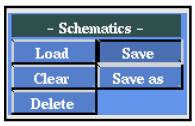
Now we place comments at the input and output signals of `SEL[1]` according to the equations shown above. First we label the left input wire at the northern border of the schematic. To do this select the entry `Enter` from

the `-Comments-` submenu and type `s1` at the following input window. After you have hit the return key, you can move the comment text within the workarea. Place it near to the connection of the left input wire with the northern border of the schematic. The text `s1` denotes that this signal represents a single bit of the sum plus one. In the same way the right input wire at the northern border is labelled `s0` to show that it is a single bit of the sum.

The schematic `SEL[1]` has two more input wires which represent the incoming carry values. These carries are input from the eastern border and are routed through `SEL[1]` to its western border in order to be available in the selection subcircuit on the left side of the current. The upper one of the two carry wires represents the value of the carry of the sum plus one. Therefore we place a comment `c1L` near to the connection of this wire with the eastern border of the schematic. This comment marks this wire as being the carry (`c`) of the sum plus one (`1`) of the lower part (`L`) of the operands. For analog reasons the lower carry wire is labelled with `c0L`.

Finally we mark the output wires of `SEL[1]`. According to our equation from above we place the comment text `a1` near to the connection of the left output wire with the southern border of the schematic. The right output wire gets the comment `a0`. Before we save the schematic you should compare your input with that shown in figure 4.35.

Saving the Schematic



To complete the specification of the n bit conditional sum adder we save the last schematic to the file `SEL[1].dag` in `DAGDIR` now. Do this by selecting the entry `Save` from the `-Schematics-` submenu.

Now we have drawn the four schematics needed to describe an arbitrary conditional sum adder for operands of length $n = 2^k$ where $k \geq 0$. Note that you do not have to change any of the inputs to extract e.g. a 32 bit adder instead of a 16 bit adder. This is true if we only consider the functional behaviour of the circuit. In reality the change of the bitlength of the operands influences the timing behaviour of the signals because of

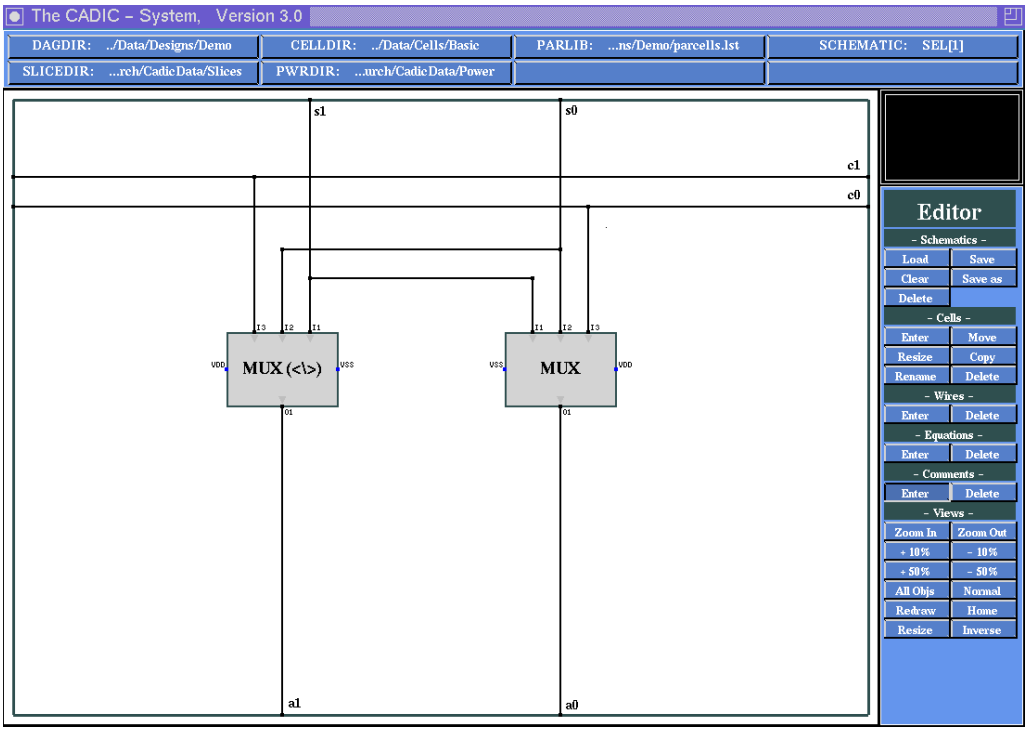


Figure 4.35: Remarks for the input and output signals of `SEL[1]`

increasing fanout for example. We will consider such problems when we describe the integrated tools.

Another problem is the succession of the input and output signals of our design of the conditional sum adder. As shown above the inputs are given in the sequence a_{n-1},b_{n-1},..., a_i,b_i,...,a_0,b_0 because of the recursive splitting of the operands. This is a restriction if we want to visualize simulation results in order to check the correctness of our design. We will demonstrate those simulations in chapter 6. A more natural sequence for the inputs of the adder would be a_{n-1},...,a_0 followed by b_{n-1}, ...,b_0. In the same way the sequence of the output signals c_1,c_0,$s_{1,n-1}$,$s_{0,n-1}$, ...,$s_{1,0}$,$s_{0,0}$ which contains the bits of the sum plus one and the sum in alternating order should be changed to the bits of the sum only preceeded by the corresponding carry bit.

We can achieve these signal sequences for the inputs and outputs of the conditional sum adder by two simple subcircuits which only contain wiring

elements. These subcircuits which are also parameterized by the length n of the operands can be described by very simple recursive equations as we will show in the following paragraphs.

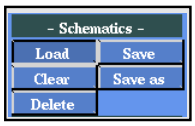
4.8.6 General Equation for Shuffle Subcircuit

We want to specify a circuit with the following functionality: given the input sequence $a_{n-1},\ldots,a_0,b_{n-1},\ldots,b_0$ it should produce the output sequence $a_{n-1},b_{n-1},\ldots,a_0,b_0$. This permutation operation is used very often and is called shuffling of the input wires.

A recursive specification can be derived by the following considerations. To shuffle two sequences a and b of n bits we take the first half of a and the first half of b and shuffle them in an appropriate shuffle subcircuit. In the same way we take the second half of a and the second half of b and shuffle them together.

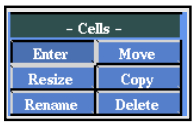
Opening a New Schematic

⇝ 12.2.1



Open a new schematic for the general recursive equation of the shuffling subcircuit. We call this schematic `SHUFFLE[n]` to show that it can be used to shuffle two input sequences of length n.

Entering and Manipulating Parameterized Macros



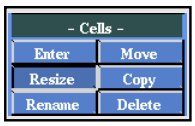
We obtain a simple recursion, because the length of the input sequences of the shuffling subcircuit have the same size as the length of the operands of the conditional sum adder ($n = 2^k$). As shown above we need two parameterized macro cells `SHUFFLE[n/2]`. We will enter the first one by selecting

⇝ 12.3.1

the entry `Enter` from the `-Cells-` submenu . Move the pointer into the window with the parameterized macro names (third cell selection window) and select the entry `--- New Cell ---` by pressing the left mouse button when it is highlighted. In the following input window type in the name `SHUFFLE[n/2]` and press the return key. Now you can move the graphical representation of the new parameterized macro cell within the workarea. Place it near to the western border of the schematic. We will create the

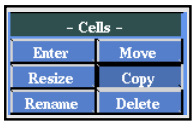
second instance of `SHUFFLE[n/2]` with the help of the copy operation and you can abort the cell selection mode by pressing the right mouse button within one of the three cell selection windows.

⇝ 12.3.3



Before we create a copy we will change the default size of the macro. Select the entry `Resize` from the `-Cells-` submenu. If you move the pointer into the workarea you are in cell selection mode and the only instance within the schematic is highlighted in blue. Select it for the resize operation by pressing the left mouse button. After the selection you can draw a rubberbanding frame which indicates the new size of the macro. Choose an appropriate size and confirm it with the left mouse button. After that you return to cell selection mode which can be aborted by pressing the right mouse button within the workarea.

⇝ 12.3.4



The second instance is now created by copying the first one. To do this select the entry `Copy` from the `-Cells-` submenu. Move the pointer into the workarea to select the first `SHUFFLE[n/2]` with the left mouse button. Now you get a second macro with the same size and name as the first one. You are in the cell placement mode, so you can choose an appropriate position for the new instance (c.f. figure 4.36) and confirm this with the left mouse button. The following cell selection mode can be aborted by pressing the right mouse button within the workarea.

Entering Wires

⇝ 12.4.1

We start entering the wiring for `SHUFFLE[n]` at the southern borders of the two instances. These are the output wires of the shuffling subcircuit. To activate enter wire mode select the entry `Enter` from the `-Wires-` submenu. First we select the start point on the southern border of the left `SHUFFLE[n/2]`. Fix it with the left mouse button and move the pointer vertically downwards until it snaps to the southern border of the schematic. After fixing the end point the input window will popup in order to specify the width of this wire. Type in `@abH[n]` to denote that this wire represents the higher part of the shuffled sequence of two input sequences a and b. In the same way you should connect the southern border of the right

`SHUFFLE[n/2]` with the southern border of the schematic. The width of this wire is described by `@abL[n]`.

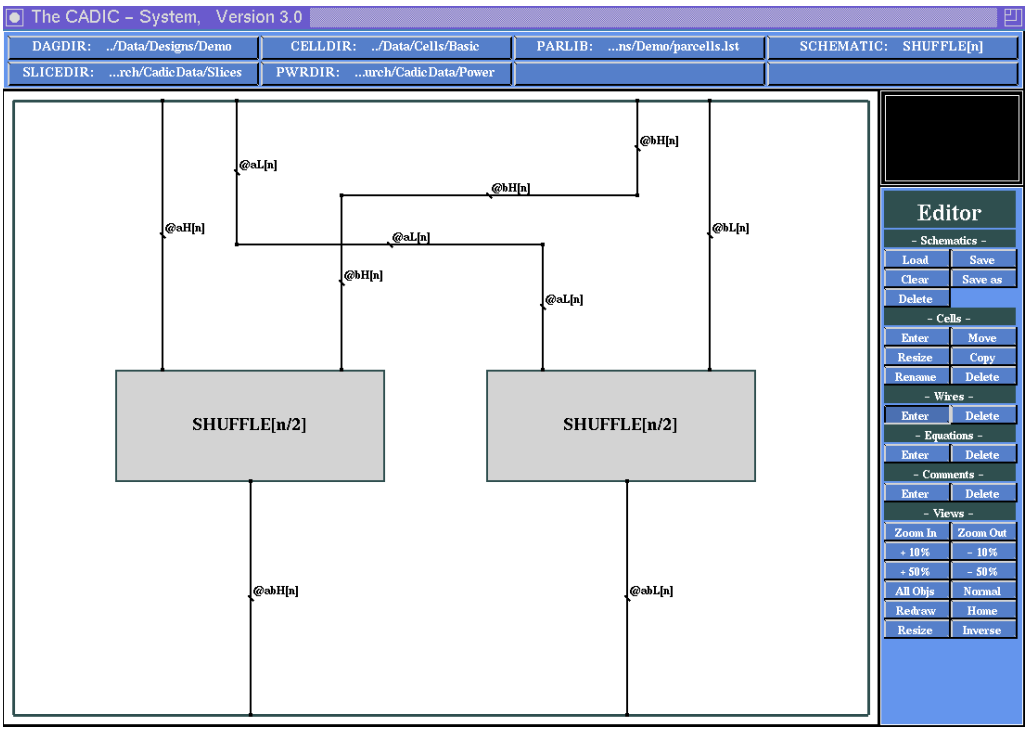


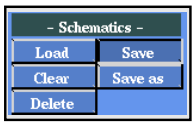
Figure 4.36: Schematic for the general equation `SHUFFLE[n]` of the shuffling subcircuit

Now we turn to the input wires of the two shuffle instances. As shown above we have to shuffle the first half of the first input sequence with the first half of the second input sequence in the left `SHUFFLE[n/2]`. For these purposes we connect the northern border of the left instance with the northern border of the schematic with a wire of width `@aH[n]` to denote that it represents the upper (H) half of the input sequence a. In the same way we connect the northern border of the right instance with the northern border of the schematic by a wire of width `@bL[n]` (c.f. figure 4.36). Next we draw a wire from the right part of the northern border of the left instance near to the point where the wire `@bL[n]` is connected to the schematic border. This can be done by fixing the start point on the border of the left instance and moving the pointer a short distance upwards and then to the right near to the wire `@bL[n]`. Here we fix a wire point by pressing the left mouse button.

Enter `@bH[n]` as the width of this wiring which represents the upper half of the input sequence b. From this point we connect the wire to the northern border of the schematic now. This can easily be done because a fixed end point serves as start point for the next wire. So we have to click one time to fix the end point of the previous wire and use this point to continue drawing. You do not have to type in the width again because it can be derived from the continued wire. In the same way you should connect the northern border of the right `SHUFFLE[n/2]` with a position on the northern border of the schematic near to the position of the wire `@aH[n]`. The width of this wire is denoted by `@aL[n]`. Abort the enter wire mode by pressing the right mouse button twice within the workarea.

Saving the Schematic

⇝ 12.2.2



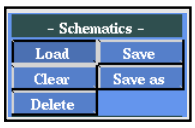
In this schematic we do not enter comments for the reasons of simplicity. Now you can save the general equation for the shuffling subcircuit into `DAGDIR` by selecting the entry `Save` from the `-Schematics-` submenu.

4.8.7 Basic Equation for Shuffling Subcircuit

The basic equation for the shuffling subcircuit concerns the merging of two input sequences of length 2. It is obvious that in case of sequences of length 1 we have nothing to do (the input is equal to the output). For two sequences of length 2 a_1,a_0 and b_1,b_0 the output sequence is a_1,b_1,a_0,b_0. This can be achieved by a simple schematic input which only contains wiring elements.

Opening a New Schematic

⇝ 12.2.1



For the basic equation of the shuffling subcircuit we create a new schematic named `SHUFFLE[2]`. This denotes that it shuffles two input sequences of length 2 (compare this with the general equation where `SHUFFLE[n]` shuffles two sequences of length n). Open the new schematic by selecting `**** New ****` from the list of schematic names which will be displayed when you select the entry `Load` from the `-Schematics-` submenu. Type `SHUFFLE[2]` as input to the popup window and press the return key.

Entering Wires

↝ 12.4.1

In this schematic we need no macros nor basic cells, so we proceed with entering the wiring. As shown above `SHUFFLE[2]` has four input and four output signals. The output values are the same as the inputs, but their order has changed. From the comparison between the order of the inputs and the outputs we obtain the desired schematic. First we notice that the first output signal a_1 is the same signal as the first input signal. Therefore we draw a vertical wire from the northern border to the southern border of the schematic. To do this select the entry `Enter` from the `-Wires-` submenu. Move the pointer onto the northern border of the schematic and fix the start point near to the western border by pressing the left mouse button. Draw the rubberbanding line vertically down towards the southern border and fix its end point in the same way. Because this wire represents one single bit of the input sequence we type `1` as the input to the window asking the width of the wire.

For the same reasons we can draw the wire for the last input bit b_0 near to the eastern border straight from the northern to the southern border of the schematic. The width of this wire also is `1`.

As shown above the two bits in the middle of the input sequence a_0 and b_1 have to be exchanged. We achieve this by drawing two crossed wires. Fix the start point for the input a_0 at the northern border of the schematic right beside a_1 and move the pointer vertically down a certain distance. Fix the end point and type in `1` for its width. From this end point we continue the wire horizontally towards the right wire for b_0 and then down to the southern border of the schematic where it is fixed by pressing the left mouse button (c.f. figure 4.37). In the same way you should draw the wire for the third input bit b_1 from the northern to the southern border of the schematic creating a crossing with the previous wire. Now your schematic should look like that shown in figure 4.37.

Saving the Schematic

↝ 12.2.2

In this schematic we do not enter comments for the reasons of simplic-

ity. Now you should save the basic equation for the shuffling subcircuit to the file `SHUFFLE[2].dag` in `DAGDIR` by selecting the entry `Save` from the `-Schematics-` submenu.

The shuffling subcircuit generates the correct input sequence a_{n-1},b_{n-1},...,a_0,b_0 for our design of the conditional sum adder. This subcircuit helps us to interprete the simulation results in chapter 6 because we can keep the bits of the operands a and b together.

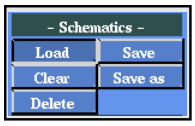
For a similar reason we combine the bits of the output values of the conditional sum adder. As shown above the sequence of the output bits is c_1,c_0, $s_{1,n-1}$,$s_{0,n-1}$,...,$s_{1,0}$,$s_{0,0}$. The first two bits represent the carries of the sum plus one and the sum of the addition. The next $2n$ bits represent the sum plus one and the sum in alternating order. As result we only want to have the carry and the bits of the sum c_0,$s_{0,n-1}$,...,$s_{0,0}$ as outputs of our addition circuit. To perform this task we will now specify a subcircuit which cuts the not desired wires.

4.8.8 General Equation for the Cutting Subcircuit

Similar to the row of multiplexers from section 4.8.4 this subcircuit can be recursively specified. The task of the whole circuit is to cut each wire on an odd position from a sequence w_1,w_2,w_3,...,w_k in order to generate the sequence w_2,w_4,...,w_k (if k is an even number as in our special case). That means we have to create a row of $\frac{k}{2}$ elements where each element cuts the first wire and feeds through the second.

Opening a Schematic

⇝ 12.2.1



We use the schematic `SEL[k]` to create the new equation because the structure of the general equation of the cutting subcircuit is very similar to that of the row of multiplexers. To open it select the entry `Load` from the `-Schematics-` submenu and move the pointer onto the name `SEL[k]`. Select it by pressing the left mouse button when it is highlighted.

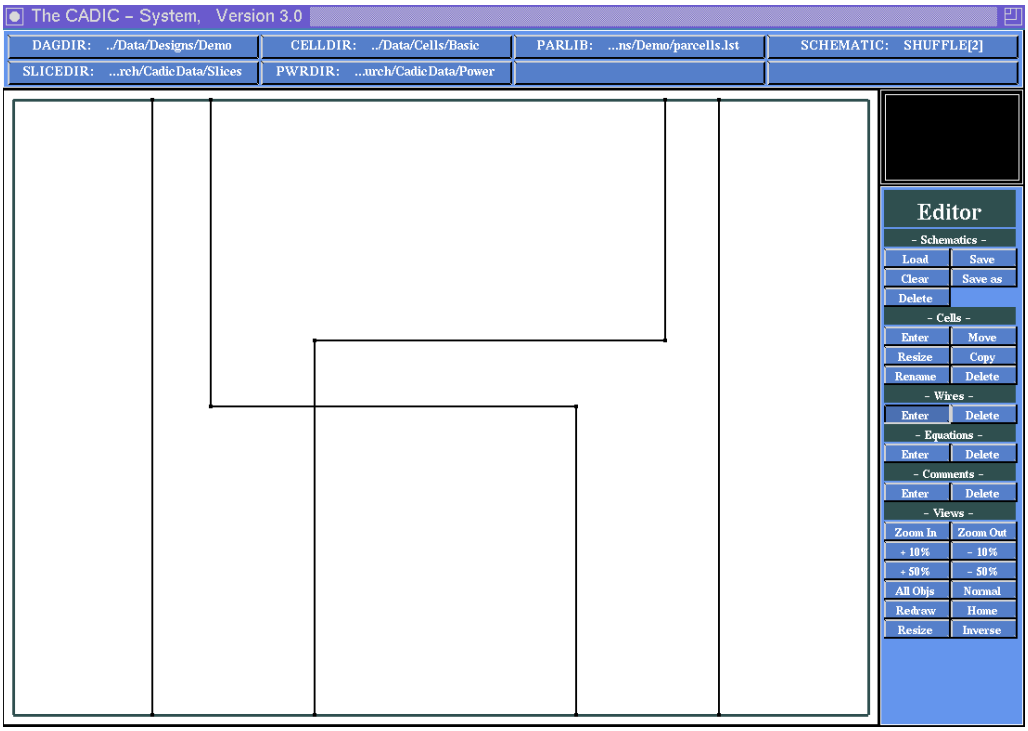


Figure 4.37: Schematic for the basic equation `SHUFFLE[2]`

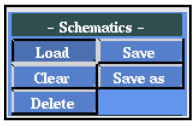
Saving a Schematic Under a New Name

⇝ 12.2.3

In the workarea the schematic input of `SEL[k]` is visible. We want to copy this schematic to a new sheet in order to construct the cutting subcircuit. You can do this by selecting the entry `Save As` from the `-Schematics-` submenu. In the following input window you should type in a new name for the schematic, here it should be `CUT[k]`. Note that after you pressed the return key the name of the loaded schematic still is `SEL[k]`.

Opening a Schematic

⇝ 12.2.1

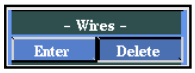


In order not to overwrite `SEL[k]` first we have to load the copy which is saved under the name `CUT[k]`. To do this select the entry `Load` from the `-Schematics-` submenu. In the list window move the pointer onto the name `CUT[k]` and press the left mouse button when the name is highlighted. Note that you will see the same schematic as before but the schematic name in the upper right corner of the graphical surface has changed to `CUT[k]` which

is now the schematic under construction.

Deleting Wires

⇝ 12.4.2



Before we draw all the wires for `CUT[k]` we have to delete the wires from `SEL[k]`. To do this select the entry `Delete` from the `-Wires-` submenu. If you move the pointer into the workarea you are in the wire selection mode. The wire nearest to the pointer is highlighted (it changes its colour from yellow to cyan). By pressing the left mouse button you can delete the highlighted wire. After the deletion operation a new wire is highlighted which is the closest wire to the pointer position. Proceed with this operation until all wires are deleted. After the deletion of the last wire the wire selection mode is automatically terminated and you return to menu selection mode. Alternatively the wire deletion mode can be terminated by pressing the right mouse button within the workarea.

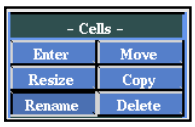
Deleting Comments

⇝ 12.5.2

Just as the wires we have to delete the two comment texts in the schematic before we change the labels of the macros. To do this select the entry `Delete` from the `-Comments-` submenu. If you move the pointer into the workarea you are in comment selection mode and the comment text nearest to the pointer is highlighted (its colour changes from white to cyan). By pressing the left mouse button this text can be erased from the schematic. After the deletion the next comment text which is nearest to the pointer is highlighted. Proceed with this operation until all comments are erased. After the deletion of the last comment the function is automatically terminated as well as when you press the right mouse button within the workarea.

Manipulating Parameterized Macros

⇝ 12.3.5



Next we have to rename the two instances which are still labelled `SEL[upper(k/2)]` and `SEL[lower(k/2)]`. According to the name `CUT` of our schematic we have to change the names of these instances. Select the entry `Rename` from the `-Cells-` submenu. If you move the pointer into

the workarea you are in cell selection mode where the instance closest to the pointer position is highlighted in blue. First we select the left instance and press the left mouse button. In the following input window we type in `CUT[upper(k/2)]` as its new name and press the return key to confirm the input. Now select the right instance in the same way and change its name to `CUT[lower(k/2)]`. After that you can abort cell selection mode by pressing the right mouse button within the workarea.

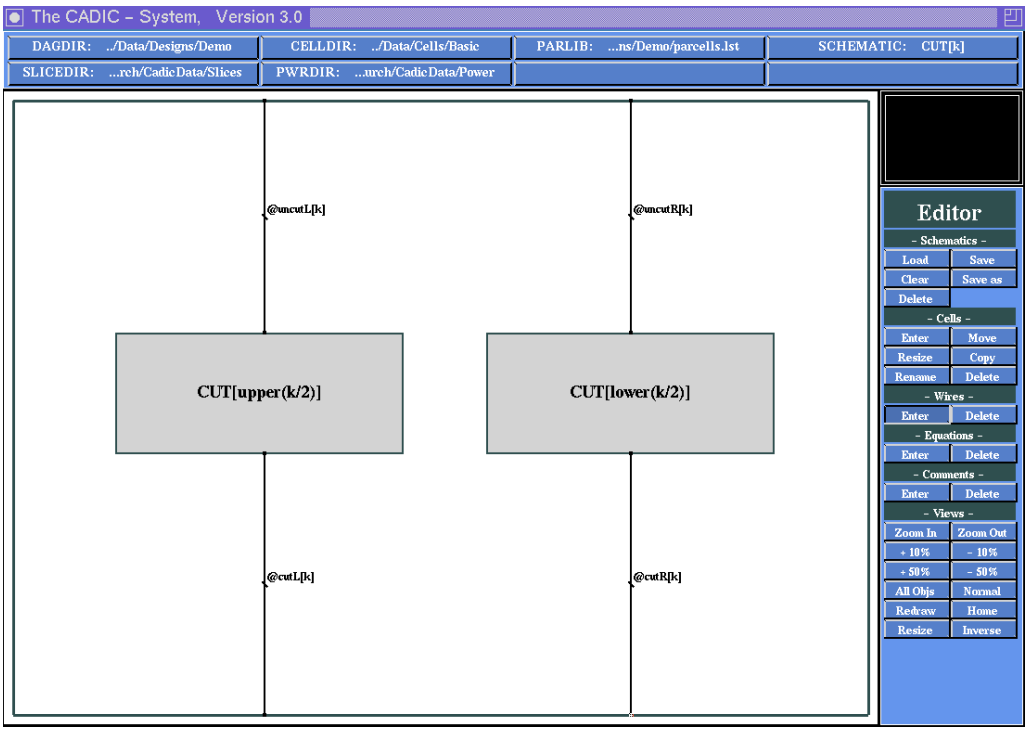
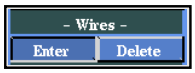


Figure 4.38: Graphical input for the general equation of the cutting subcircuit `CUT[k]`

Entering Wires

⇝ 12.4.1



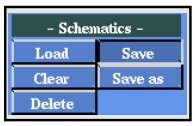
The wiring of `CUT[k]` is very simple and only consists of vertical wires connecting the two instances with the schematic borders. To draw it select the entry `Enter` from the `-Wires-` submenu. First we connect the northern border of the left instance with the northern border of the schematic. This wire represents the input signals to `CUT[upper(k/2)]`, i.e. from this bundle each wire at an odd position has to be cut. At the prompt for the width of

this bundle type in `@uncutL[k]` to denote that it represents the left half of the uncut wires. Now select the start point for the next wire at the southern border of the left instance and move the pointer vertically down towards the southern border of the schematic where you fix the end point of this wire. Type in `@cutL[k]` as its width in order to show that the output of the left instance are the cut wires.

In the same way we connect the right instance with the corresponding schematic borders. These wires have the width `@uncutR[k]` (`@cutR[k]`) for the connection between the northern (southern) borders. Your graphical input should look similar to that shown in figure 4.38. Abort the wire enter mode by pressing the right mouse button twice within the workarea.

Saving the Schematic

⇝ 12.2.2



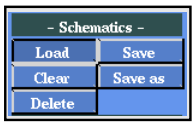
Save the schematic `CUT[k]` into `DAGDIR` by selecting the entry `Save` from the `-Schematics-` submenu.

4.8.9 Basic Equation for the Cutting Subcircuit

The basic element for the cutting subcircuit is very simple. It takes two input signals w_i,w_{i+1} (where i is an odd number) of the whole input sequence, cuts the first one and feeds through the second w_{i+1} as its output signal.

Opening a New Schematic

⇝ 12.2.1



For the basic equation of the cutting subcircuit we now open a new schematic `CUT[1]`. Select the entry `Load` from the `-Schematics-` submenu and move the pointer to the position of `**** New ****` within the list window for the schematic names. When it is highlighted press the left mouse button. At the following input window type `CUT[1]` as the new name and press the return key.

Entering Wires

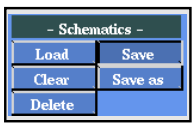
↝ 12.4.1

Because this schematic does not contain macros we proceed with entering the wiring just as we did it in the case of `SHUFFLE[2]`. `CUT[1]` has two input signals from which the first one is cut. To draw the needed wires select the entry `Enter` from the `-Wires-` submenu. Move the pointer into the workarea near onto the middle of the northern border of the schematic and fix the start point of the first wire with the left mouse button. Move vertically down until you have reached the middle of the schematic where we fix the end point of this wire. Now we have drawn a wire which starts at the northern border of the schematic and ends within the schematic. As you have already learned during the input of the general equation for the conditional sum adder `CSA[n]` the end point of this wire is a legal operation within the underlying calculus. Cutting a wire to an open end is called a projection operation. For the width of this wire type in `1` at the following input window.

The second wire in `CUT[k]` has to connect the northern and the southern border of the schematic. Before you can fix the start point of this wire you have to press the right mouse button. This is necessary to tell the editor that you want to fix a new start point for the next wire and that you do not want to continue the just drawn wire (as we did it in the case of the wiring of `SHUFFLE[2]`). Note that you have to select a start point of the wire on the northern border right beside the start point of the previous wire. The width of this wire is `1` as well. Abort the enter wire mode by pressing the right mouse button twice within the workarea.

Saving the Schematic

↝ 12.2.2



If you have drawn a schematic similar to that shown in figure 4.39 you can save it into `DAGDIR` by selecting the entry `Save` from the `-Schematics-` submenu.

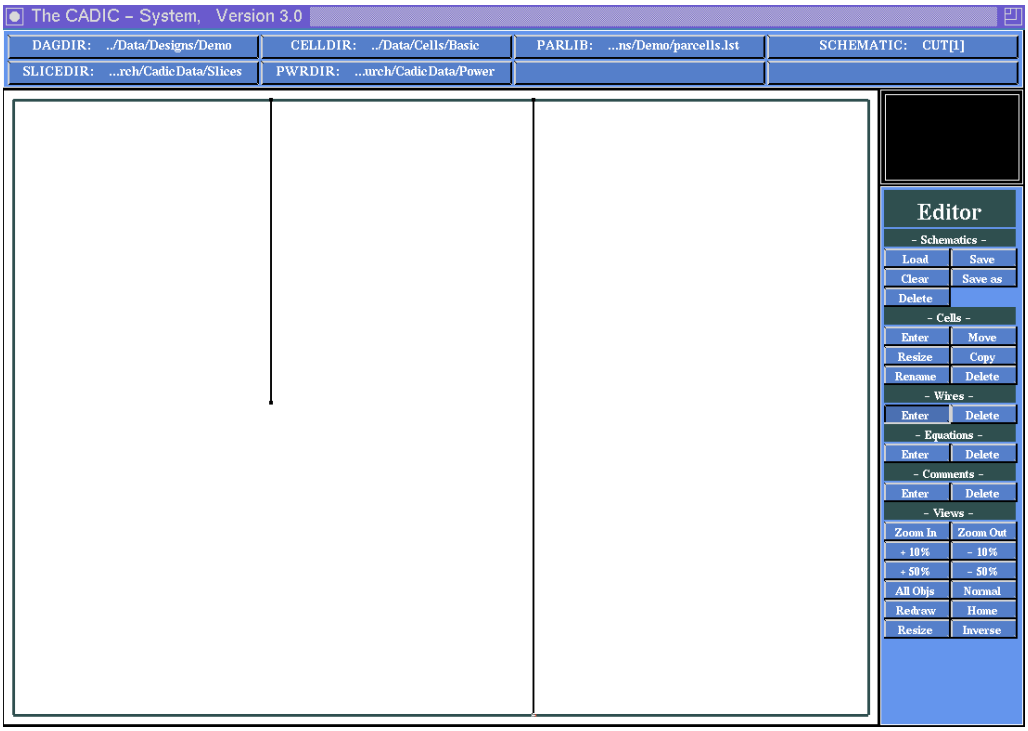


Figure 4.39: Projection of a single wire in the basic equation of the cutting subcircuit `CUT[1]`

4.8.10 Equation for the complete n bit adder

In this section we will combine the main part of the conditional sum adder, i.e. the circuit `CSA[n]` for n bit operands with the two wiring subcircuits `SHUFFLE[n]` and `CUT[k]` to obtain an addition circuit with the following functionality. It receives two binary vectors $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$ as inputs and generates an output vector of the form $s = cs_{n-1} \dots s_0$ with $s = a + b$. c represents the carry bit of the summation. We choose this form of the adder because it allows us to control simulation results very easily. We will demonstrate this in chapter 6.

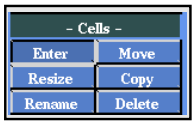
Opening a New Schematic

⇝ 12.2.1

For the complete n bit adder we use a simple parameterized equation (not a recursive specification). That is the reason why we only need one equation where we map the parameters of the subcircuits in an appropriate way. We

need no basic equation in this case. Open a new schematic `CSADDER[n]` by selecting the entry `Load` from the `-Schematics-` submenu and typing this name at the input window which pops up when you select the entry `**** New ****` in the schematic list window.

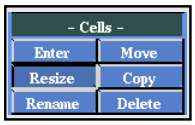
Entering and Manipulating Parameterized Macros



We need three parameterized macro cells in `CSADDER[n]` which will be vertically arranged. We begin with the shuffling of the operands a and b. The length of the operands is n that is why we need an instance `SHUFFLE[n]` to shuffle two sequences of length n. Select the entry `Enter` from the `-Cells-` submenu . Now choose the entry `--- New Cell ---` from the cell list window for the parameterized macros (bottom window). At the input window type the name `SHUFFLE[n]` as input and move the appearing macro to a position near to the northern border of the schematic. Before we select the other macros we want to resize this first one. Therefore you should abort cell selection mode by pressing the right mouse button within one of the three popup windows.

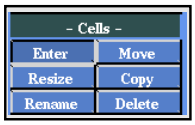
↝ 12.3.1

↝ 12.3.3



Now select the entry `Resize` from the `-Cells-` submenu. Move the pointer into the workarea and press the left mouse button when the instance `SHUFFLE[n]` is highlighted. Now you can draw the rubberbanding frame to the new macro size. We select a small height for this macro to indicate that it only contains wiring elements and no basic cells. If you have chosen a size similar to that shown in figure 4.40 can should confirm the size with the left mouse button. After that abort the cell selection mode by pressing the right mouse button within the workarea.

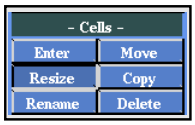
↝ 12.3.1



Now we enter the macro for the main part of our adder which is an instance of the general equation for the conditional sum adder. Select the entry `Enter` from the `-Cells-` submenu again in order to create a new parameterized macro (entry `--- New Cell ---` from the third popup window). Type `CSA[n]` at the input window and move the macro with its upper left corner below the instance `SHUFFLE[n]`. Leave enough space between the two instances for the wiring and confirm the position with the left mouse

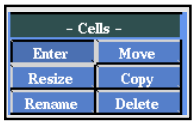
button. After that abort cell selection mode (right mouse button in one of the popup windows) in order to resize `CSA[n]`.

↝ 12.3.3



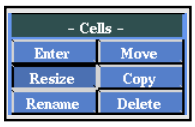
Select the entry `Resize` from the `-Cells-` submenu and move the pointer into the workarea. When the instance `CSA[n]` is highlighted press the left mouse button and select a new size for this macro. Choose the same width as for `SHUFFLE[n]` but make this macro much higher in order to denote that it contains the main part of the whole adder (c.f. figure 4.40). Confirm the new size with the left mouse button and abort any further cell selection with the right mouse button.

↝ 12.3.1



The third macro we need is an instance of the cutting subcircuit. To create it select the entry `Enter` from the `-Cells-` submenu and move the pointer onto the position `--- New Cell ---` in the third popup window. As shown above the output of `CSA[n]` is a sequence of length $2 \cdot (n + 1)$ because we have two sums of length n and one carry bit for each of the sums. That is the reason why we need an instance `CUT[n+1]` of the cutting subcircuit (type `CUT[n+1]` at the popup input window). Move the new macro with its upper left corner under the `CSA[n]` and leave enough space for the wiring. After you confirm the position with the left mouse button the cell selection windows popup again. We do not need another instance so you can abort cell selection by pressing the right mouse button within one of the three popup windows.

↝ 12.3.3



The last operation concerning the macros consists in resizing the instance `CUT[n+1]`. Select the entry `Resize` from the `-Cells-` submenu and move the pointer into the workarea so that `CUT[n+1]` is highlighted. Confirm the selection by pressing the left mouse button. With the help of the rubber-banding frame we now select a new size for this macro. Choose the same width as for the other two instances but a much smaller height than that of `CSA[n]`. The height can approximately be the same as that of `SHUFFLE[n]` to denote that this subcircuit only contains wiring elements as well. Compare the sizes of the macros with figure 4.40 and abort cell selection mode if they are correct (right mouse button in workarea).

Entering Wires

↝ 12.4.1

To draw the wiring of `CSADDER[n]` select the entry `Enter` from the `-Wires-` submenu. First we draw the input wires of the instance `SHUFFLE[n]`. Fix a start point at the left half of the northern border of `SHUFFLE[n]` and move the pointer vertically up onto the northern border of the schematic. After fixing the end point with the left mouse button you should type `@a[n]` as width of this wire to denote that it represents the first operand a of length n. In the same way we draw a second wire from the northern border of `SHUFFLE[n]` to the northern border of the schematic. This wire should be located right beside the previous wire. For its width type in `@b[n]` to denote that it represents the second operand b of length n.

Next we connect the southern border of `SHUFFLE[n]` with the northern border of `CSA[n]`. For the width of this wire type in `@ab[n]` to denote that it represents the bits of the operands a and b in alternating order as it is generated by the shuffling subcircuit.

In the same way we draw a wire from the southern border of `CSA[n]` to the northern border of `CUT[n+1]`. We describe the width of this wire by `@csums[n]`, because the output of `CSA[n]` is the sum and sum plus one, preceeded by the corresponding carries.

The last wire of this schematic connects the southern border of the instance `CUT[n+1]` with the southern border of the schematic. For the width of this wire type in `@csum[n]` to denote that it only represents the bits of the sum including the leading carry bit. Finally abort the enter wire mode by pressing the right mouse button twice within the workarea.

Entering an Equation

↝ 12.6.1

In the wiring above we have specified the width of the input operands by the wire variables `@a[n]` and `@b[n]`. The specification of the n bit conditional sum adder will give us the equation that `@a[n] + @b[n] = 2n`. This equation is not sufficient to uniquely determine the value of the two variables. One possible solution would be `@a[n] = 2n` and `@b[n] = 0` but

this is not the desired one. To obtain a unique solution we enter an equation about the width of the two variables. To do this select `Enter` from the `-Equations-` submenu. In the same way as you entered a comment text now you can type in an equation about relationships between wire variables in the popup window. Type in `@a[n] = @b[n]` and press the return key. By this equation we tell the system that the width of operand a is equal to that of operand b. The equation is movable within the workarea like comment text. Select an appropriate place like that shown in figure 4.40 and confirm the position by pressing the left mouse button. After that the input window will popup again. Because we do not need any further equation in our specification just press the return key to the empty input window in order to abort the enter equation mode.

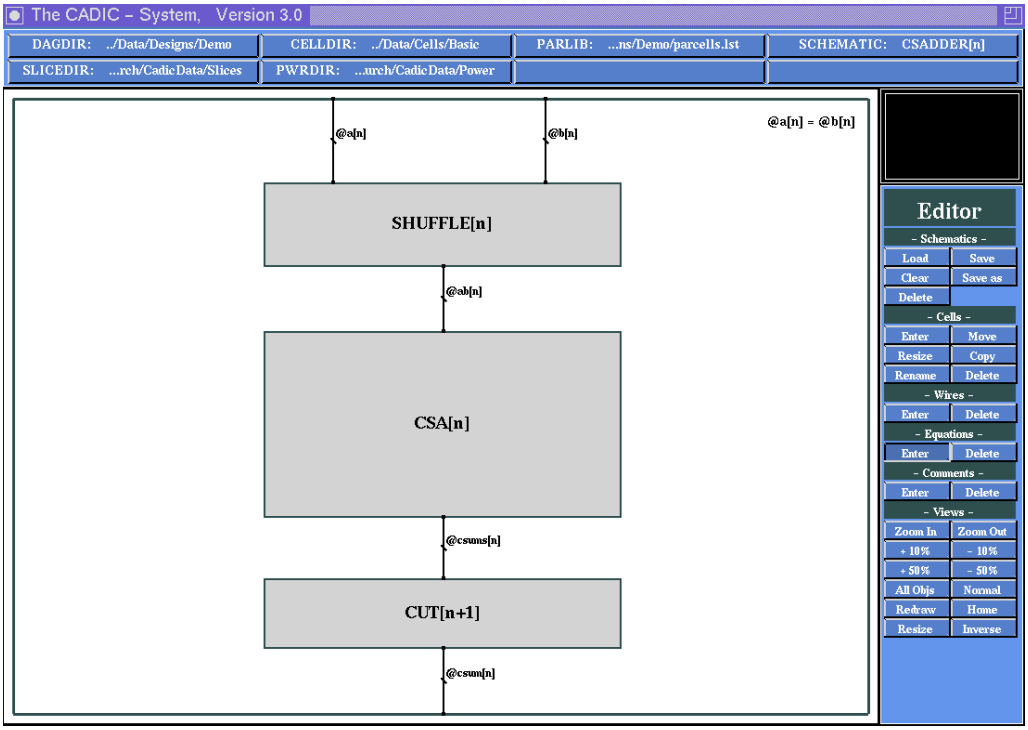


Figure 4.40: Graphical input for `CSADDER[n]` with an additional equation for unique specification of the wire variables `@a[n]` and `@b[n]`

Saving the Schematic

⇝ 12.2.2

You can check all the operations we used above (entering and resizing macros

and entering the wires) by comparing your schematic with that shown in figure 4.40. If everything is correct you can save the schematic into `DAGDIR` by selecting the entry `Save` from the `-Schematics-` submenu.

4.8.11 Summary

Now we have completed the specification of the n bit conditional sum adder and we will close our editing session. This can be done by simply pressing the right mouse button within the menuline. Then the submenu for the graphical editor will disappear and the system's main menu is displayed again.

Note that our design of the conditional sum adder only works for parameter values $n = 2^k$. If you enter any other value for n, the system may not be able to construct the corresponding adder. For more information about selecting parameter values you can read the section 5.8.

⇝ 5.8

Here you can stop your tour through the CADIC system by selecting the entry `Exit` from the main menu. You can also proceed with using another tool of the system in order to analyze or synthesize your design. From this point of the manual you can skip to another chapter describing the special tool. We recommend to you to continue with chapter 5 where the extraction of a hierarchical representation for fixed values of the design parameters is shown. This hierarchical representation is the basic structure for the integrated tools. It is used to generate and visualize the results during the design process.

5

Hierarchy Representation

With the help of the graphical editor the user can setup hierarchical specifications of circuits. Especially he can specify whole families of circuits. As we have shown in chapter 4 this can be done by the graphical input of recursive schematic equations which may depend on an arbitrary set of parameters.

The design tools which are integrated in the graphical surface of CADIC , do their work on a specific member of such a circuit family. For this circuit the system will build up a hierarchical representation which is given by an appropriate data structure. The main aspect of this data structure is that the circuit hierarchy is represented by a *directed acyclic graph* (DAG). In this chapter we explain the properties of this DAG structure and show how it is built. This structure results from folding the circuit hierarchy by identifying nodes which represent the same subcircuits. We respect the underlying tree structure in the following notations, when we characterize the elements of the DAG as *root*, *leaves* or *inner nodes*.

5.1 The DAG Data Structure

For a given circuit C its DAG structure includes not only the description of C, but also all the subcircuits S of C, the subcircuits of S and so on down to the elements of the basic cell library. The DAG structure for C has the following properties:

- the root of the tree is C.

- the leaves of the tree are either elements from the basic cell library or subcircuits which only consist of wiring components.
- for all subcircuits S of C there exists one inner node; multiple appearances of a subcircuit are represented by pointers to the corresponding node of the tree.

Each node of the DAG structure is described by the elements shown in figure 5.1 which we call *treenode*. It consists of the following components:

- the subcircuit *name*, including an optional list of parameters which are set to nonnegative integer values.
- a *key* which is a unique index for all treenodes within a hierarchical circuit description. Although the name of a treenode (with included parameter values) is also unique over the hierarchy, an integer index can be used better for adressing the nodes.
- the number of *instances* and the corresponding *list of instances* which represent the macros and basic cells on this hierarchy level. The entries in each instance list are numbered from 0 to `Instances-1`, such that each instance has a unique key on this hierarchy level. In the list entries there is stored local information about the instances (e.g. orientation).
- the number of *references* which represents the number of appearences of this treenode as an instance within the hierarchical description.
- pointer to specific data (views) about the treenode as for example the *graph description* which is created by the graphical editor and used for the graphical display of the treenode. There are other views (netlist, bicategorial expression, ...), which can be generated, when they are needed.

The hierarchical structure of a circuit is built over pointers from a treenode to its instance list and from each instance to a corresponding treenodes at a lower level. It is possible that more than one instance points to the same treenode which results in the DAG structure. As mentioned above there

may exist treenodes with an empty instance list which are either elements from the basic library or simple wiring components. These treenodes build the lowest level of the hierarchy.

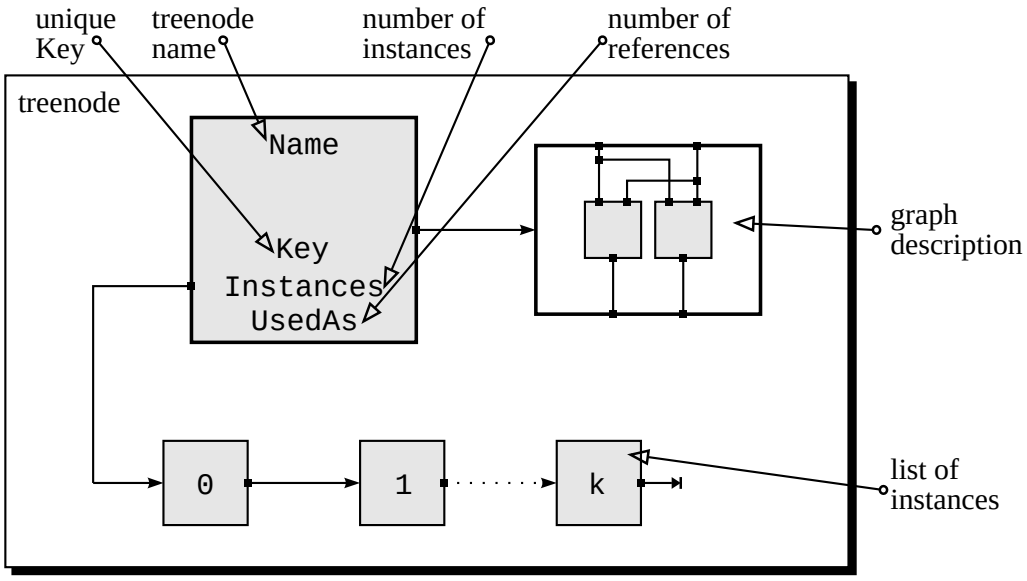


Figure 5.1: Basic structure *treenode* of the hierarchical representation

Depending on the circuit structure, we obtain a very compact description for even large circuits. To give you an impression: a 64 bit integer multiplier which consists of about 50000 basic gates, can be represented by a DAG structure of about 20 treenodes.

Figure 5.2 shows the hierarchical transition between a treenode `C` and the successor treenode `S`. `C` has a list of `k` instances, where the ith instance represents the treenode `S`. If `C` would contain more occurences of `S`, there would be further instances pointing to the treenode `S`. In the reverse direction in `S` we count the number of occurences together with corresponding pointers to the instances, which represent an appearence of `S`.

As shown in figure 5.2 there exists a reference in the instance list of a treenode for each attached successor treenode, such that any path through the hierarchical representation can be traced in a top–down direction. Additionally the data structure contains references in ascendant direction from a treenode to all locations, where it is used as an instance and from each

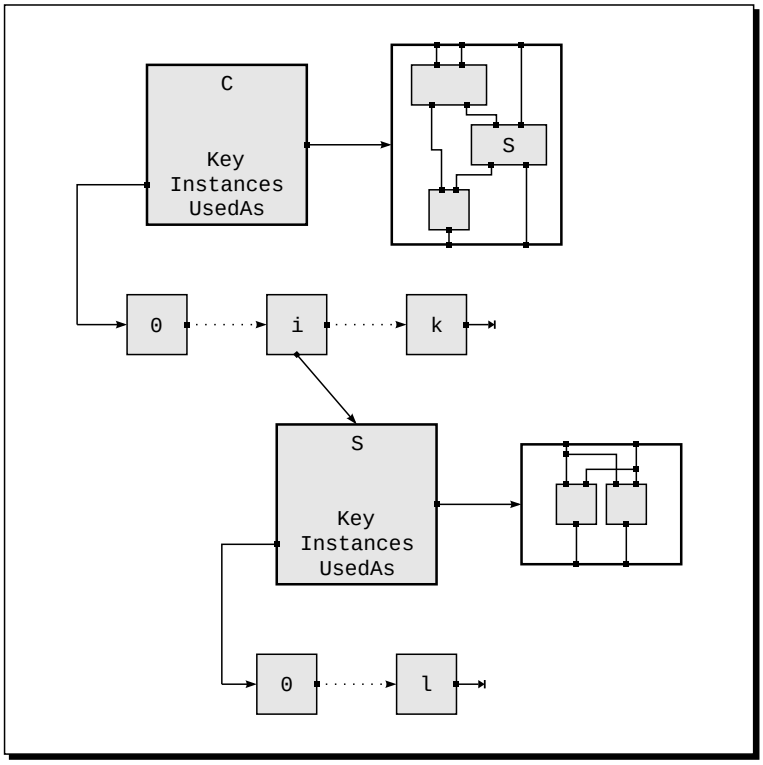


Figure 5.2: Transition from an instance of a treenode at level i to the corresponding treenode at level $i + 1$

instance to the treenode, to which it belongs. Thus, it is possible to traverse the DAG structure in every desired manner. For reasons of simplicity these upward links are not shown in figure 5.2.

The description of each hierarchy level is given by a *Cgraph* which either is interactively created during an editor session or can automatically be generated by programs, e.g. logic synthesis tools ([Sch96],[Biw94]). The Cgraph is the topographical description of the concerned circuit. Especially, it describes size information, exact wiring, signal net connections, and special layout blocks.

From the Cgraph we can obtain a representation in form of a hierarchical or flat netlist. It reflects the connection structure over hierarchy levels and represents signal net connections, dimensions and relations among subnets. This information is important for simulation tools, where we need the logical

connectivity of the circuit and can abstract from the topographical layout of the wires.

The Cgraph can also be used to create the description in form of a bicategorial expression as they are introduced in section 4.2. As mentioned there bicategorial expressions are difficult to handle by the user, because they can become very large and unreadable. But they build the base for some efficient design tools, as for example the place&route tool ([Kol86],[Fet95],[Wan95]) shown in chapter 9.

5.2 Building the DAG Structure

The building of the DAG structure is done in two steps. First the treenodes for each subcircuit are created in a top–down process starting a the highest hierarchy level. After that the pointers from the instances at each hierachy level to the corresponding treenode are setup. If you used formal variables for the width of wires (as in our design of the conditional sum adder), a system of linear equations over these variables is derived from the hierarchical structure and finally solved.

In the following we denote by (C, φ_C) a treenode C with a corresponding list of values φ_C for its parameters. If C has no parameters, we set $\varphi_C = \emptyset$.

The algorithm for building the DAG structure starts with such a pair (C, φ_C) which is pushed onto a stack. For each element on the stack there will be performed the following steps:

- pop the upmost element from the stack
- check, whether there exists already a treenode for this pair; if this is the case continue with the next element on the stack.
- find the corresponding Cgraph for the current element. This may be a macro cell, for which the Cgraph is located in the directory `DAGDIR` or a basic cell, for which the Cgraph is stored in `CELLDIR`. If it is a macro cell and the Cgraph contains elements which depend

on formal parameters, these elements are evaluated according to the current values of the treenode.

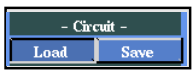
- push all instances of the current element, i.e. all its successors in the hierarchy, together with their parameter list onto the stack.
- insert the just created treenode into a hash table, such that we can check that is not creted more than once.

These steps are done as long as there are elements on the stack. As shown in [Bur94] this process will terminate, if there is a check for a maximum hierarchy level which is set by the system to an appropriate value. Finally we use the hash table to create the pointers from the instances to the corresponding treenodes which are stored in the table.

5.2.1 A First DAG: The HalfAdder

To show you how the system will build the hierarchical data structure we will load the examples from the editor session. In order to load the hierarchy of circuits, you first should select the entry `Hierarchy Checks` from the submenu `-Analysis-` within the main menu. The menu point to load the hierarchy of a circuit is also contained in the submenus of the integrated design tools as you will see in the following chapters. In this chapter we will demonstrate some basic functions concerning the circuit hierarchy. After the selection a new submenu named `Hierarchy` is displayed.

Load Circuit Hierarchy



To load a circuit hierarchically you must select the entry `Load` from the submenu `-Circuit-` and press the left mouse button. After that the list window for the circuits from the `DAGDIR` directory are shown as you have already seen during your editor session. But note that the entry `**** New ****` does not appear, i.e. you can not enter a new schematic, but can only load an already existing circuit.

Select the entry `HalfAdder` and press the left mouse button if the name is highlighted. If you look at the message window you can read the following

system messages:

```
Reading Circuit Description of "HalfAdder"
Reading Dag-File:  HalfAdder
Reading Dag-File:  AND2
Reading Dag-File:  XOR2
Total Number of Dag-Files:  3
Last Modification Date:  Fri Jan 19 17:15:54 1996
CtnTypes:  No matrix elements in convert_matrix
Circuit "HalfAdder" loaded, Total Data Size:  7422 Bytes
```

These messages reflect the building of the DAG structure. The algorithm starts with the element $(\texttt{HalfAdder}, \emptyset)$, which is pushed onto the stack. Here $\varphi_{\texttt{HalfAdder}} = \emptyset$, because `HalfAdder` has no parameters. For this element there is a treenode created and the corresponding Cgraph is read out of `DAGDIR` which corresponds to the message `Reading Dag-File: HalfAdder`. After that the instances of `HalfAdder`, i.e. `AND2` and `XOR2` are pushed onto the stack together with an empty set of parameter values. Next the element $(\texttt{AND2}, \emptyset)$ is read from the stack. The corresponding treenode is created, but no successors are pushed onto the stack, because it is a basic cell. In the same way `XOR2` is treated, and after that the stack is empty, i.e. all elements for the description of `HalfAdder` have been loaded. The stack during the loading of `HalfAdder` is shown in figure 5.3.

Figure 5.3: The stack during the loading of `HalfAdder`

Finally the system tells you that it had to load 3 files from the directories `DAGDIR` and `CELLDIR` and it gives you the last modification time. This is the time of the newest file which had to be read. The message `CtnTypes: No matrix elements in convert_matrix` indicates that there are no formal variables for the width of wires in this discription. All wires have the constant width 1, i.e. there is no matrix for an equation system to be solved. The last message `Circuit "HalfAdder" loaded, Total Data`

`Size: 7422 Bytes` tells you the total amount of memory which is used by the `HalfAdder`. This is the sum of bytes for the hierarchical structure, i.e. the treenodes, instance lists, etc. and the Cgraphs for the graphical representation.

After the setup of the pointers from the instances to the corresponding treenodes the DAG for `HalfAdder` has the form shown in figure 5.4.

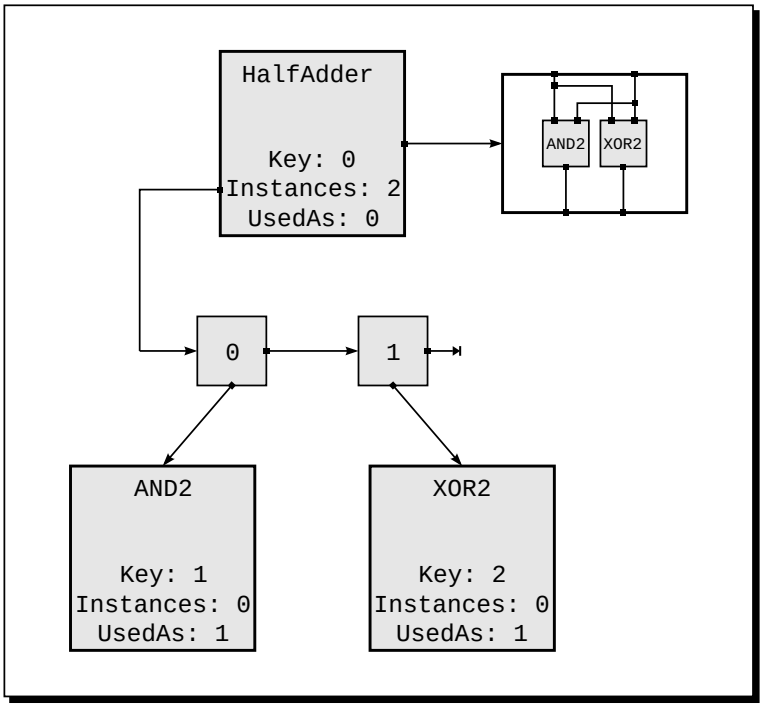


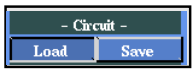
Figure 5.4: DAG structure for `HalfAdder`

In this case we have an overhead in our description: we need three treenodes which represent a circuit of only two basic gates. This negative effect only appears for small circuits and will drastically change to our advantage as you will see later.

5.2.2 More Hierarchy Levels: The FullAdder

Now we will build the DAG structure for our design of a fulladder which contains the halfadder as a subcircuit.

Load Circuit Hierarchy



Again you must select the entry `Load` from the submenu `-Circuit-` and press the left mouse button. From the list window select the entry `FullAdder` and press the left mouse button if the name is highlighted. Note that the graphical display of the halfadder turns to grey which indicates that its DAG structure is destroyed. In the message window the system displays the following informations:

```
Reading Circuit Description of "FullAdder"
Reading Dag-File:  FullAdder
Reading Dag-File:  HalfAdder
Reading Dag-File:  AND2
Reading Dag-File:  XOR2
Reading Dag-File:  OR2
Total Number of Dag-Files:  5
Last Modification Date:  Fri Jan 19 17:20:55 1996
CtnTypes:  No matrix elements in convert_matrix
Circuit "FullAdder" loaded, Total Data Size:  14698 Bytes
```

Note that although the `FullAdder` contains `HalfAdder` twice, the corresponding Cgraph is only read once. Here the first entry on the stack is $(\texttt{FullAdder}, \emptyset)$. This entry is replaced by its instances, i.e. by two elements $(\texttt{HalfAdder}, \emptyset)$ and one element $(\texttt{OR2}, \emptyset)$. After that the treenode for `HalfAdder` is created and the first pair $(\texttt{HalfAdder}, \emptyset)$ is popped from the stack. For this element the steps from the previous section are applied. For the second pair, there is done nothing, but it is removed from the stack. The reason for this is that the treenode for `HalfAdder` is already created and stored within the hash table. Finally the `OR2` is loaded and popped from the stack which has the states shown in figure 5.5 during the loading of `FullAdder`.

For the fulladder, the system had to load 5 files in total. Here you see again the message `CtnTypes:  No matrix elements in convert_matrix`, because we did not use wire variables within the fulladder design. The size

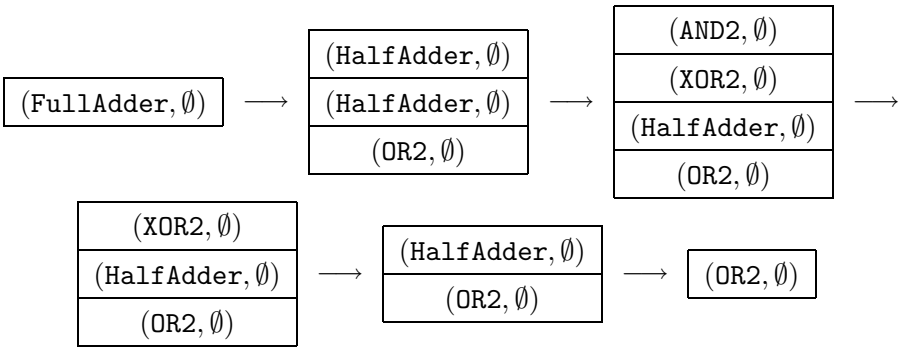


Figure 5.5: The stack during the loading of `FullAdder`

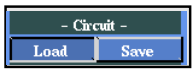
of the circuit hierarchy is given by `Circuit "FullAdder" loaded, Total Data Size: 14698 Bytes`. After the setup of the pointers from the instances to the corresponding treenodes the DAG for `FullAdder` has the form shown in figure 5.6.

For the DAG of `FullAdder` the overhead, we had in the case of the `HalfAdder` description has vanished. Figure 5.6 shows, that we need five treenodes for the hierarchical description which also represents five basic gates. In the next section you will see that this relation will improve to the side of hierarchy.

5.2.3 Parameterized Levels: The 2^n Bit Comparison Tree

In this section we will demonstrate how you can load a parameterized design. The main difference to the previous two sections is that you will have to specify nonnegative integer values for the formal parameters of the design.

Load Circuit Hierarchy



As you already know you must select the entry `Load` from the submenu `-Circuit-` and press the left mouse button. From the list window select the entry `Tree[n]` and press the left mouse button if the name is highlighted.

Note that you have selected a parameterized description with a free parameter `n`. In order to build the hierarchical data structure this parameter has

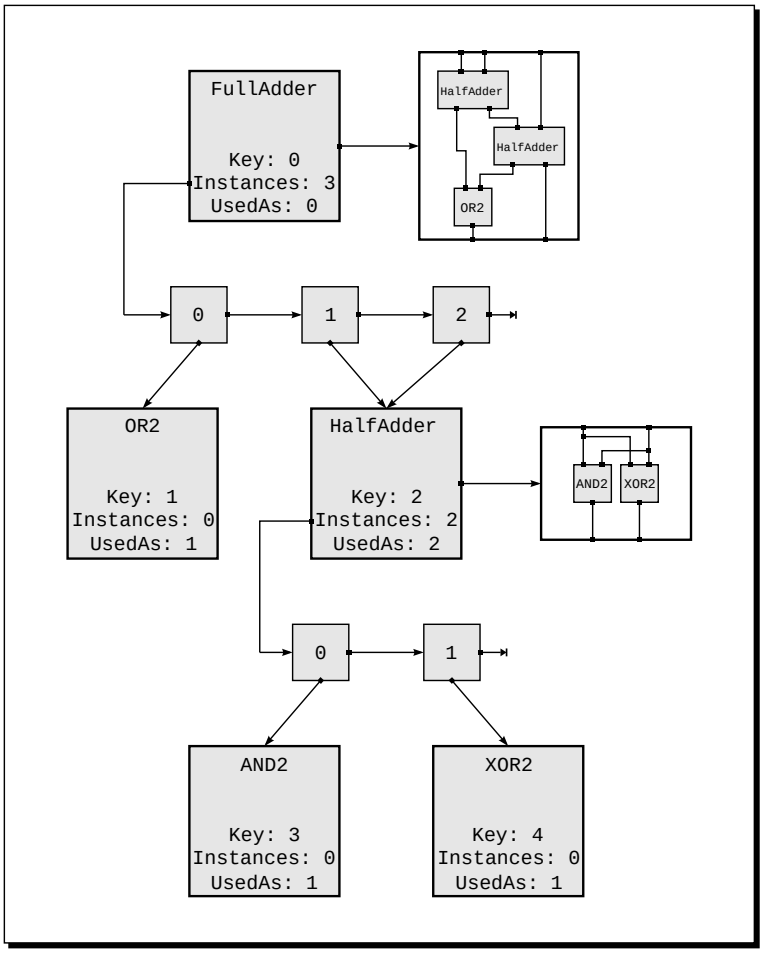


Figure 5.6: DAG structure for `FullAdder`

to be set to a nonnegative integer value. This is why the system pops up the input window onto the workarea with the prompt

```
Enter Parameter Values for "Tree[n]":<2      >
```

Type in `Tree[2]` or simply `2` to set the value of `n`=2. After confirming this entry with the return key, the previous DAG structure of `FullAdder` is destroyed and the system will display the following informations in the message window:

```
Reading Circuit Description of "Tree[2]"
Reading Dag-File:  Tree[2]
Reading Dag-File:  Tree[1]
```

```
Reading Dag-File:  Tree[0]
Reading Dag-File:  XOR2
Reading Dag-File:  OR2
Total Number of Dag-Files:  5
Last Modification Date:  Fri Jan 19 17:26:08 1996
CtnTypes:  (2 x 1) - Matrix
Circuit "Tree[2]" loaded, Total Data Size:  14907 Bytes
```

Note that all Cgraphs are loaded only once, although the level `Tree[n]` uses `Tree[n-1]` twice and each level of the hierarchy contains one OR gate.

In contrast to the loading of `HalfAdder` and `FullAdder` the first entry on the stack contains not the empty set as its second component. The algorithm starts with the pair $(\texttt{Tree}, n = 2)$. The corresponding Cgraph which is given by the schematic `Tree[n]`, is loaded. In the next step all elements within this schematic which depend on the formal parameter `n` are evaluated, according to its current value `n`=2. The parameter is used in the names of two instances `Tree[n-1]` and in the width of two wires (`2^n`). After the treenode for `Tree[2]` is created the evaluated names of its instances are pushed onto the stack as two pairs $(\texttt{Tree}, n = 1)$ as well as the instance of the OR gate given by $(\texttt{OR2}, \emptyset)$ (c.f. figure 5.7).

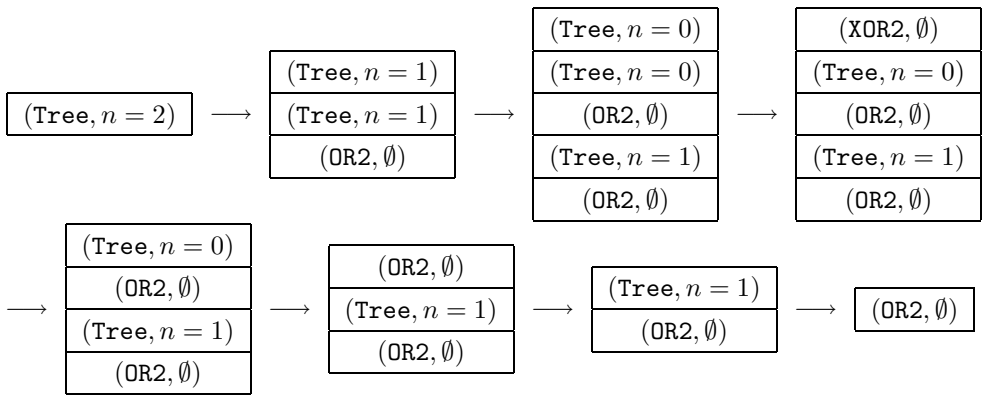


Figure 5.7: The stack during the loading of `Tree[2]`

In the next iteration of the algorithm the pair $(\texttt{Tree}, n = 1)$ is popped from the stack. The corresponding Cgraph is also given by the schematic for `Tree[n]` and is now loaded again. All elements are evaluated with the new

parameter value $n = 1$, i.e. two pairs $(\texttt{Tree}, n = 0)$ are pushed onto the stack and another OR gate and the treenode `Tree[1]` is stored in the hash table.

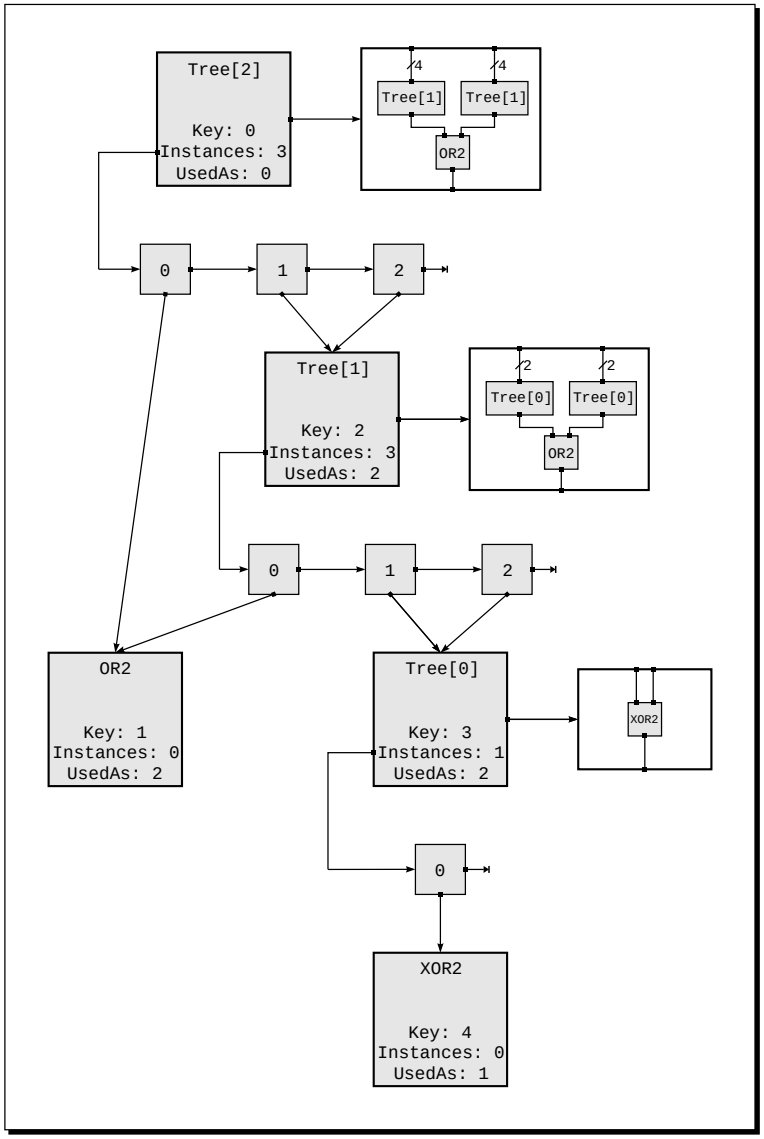


Figure 5.8: DAG structure for `Tree[2]`

Now the algorithm has to manage the pair $(\texttt{Tree}, n = 0)$, for which the corresponding Cgraph is given by the schematic `Tree[0]`. Note that the algorithm does not read in the general equation `Tree[n]` again. It takes `Tree[0]`, because the fixed parameter value `0` is more restrictive than the

free parameter `n`. We call this *choosing the best matching schematic*. After reading `Tree[0]` there are no evaluations needed, because all elements are fixed. The only instance of `Tree[0]` is an EXOR gate, such that the pair $(\texttt{XOR2}, \emptyset)$ is pushed onto the stack. Finally the treenode `Tree[0]` is stored in the hash table.

From this point the algorithm only has to read the basic gates `XOR2` and `OR2` to complete the hierarchical structure. All other components on the stack are already created, such that they can be removed without any further operation. After creating the links between the instances of each hierarchy level and the corresponding treenodes we get the DAG structure shown in figure 5.8.

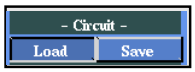
In the example of the comparison tree the hierarchical description is already smaller than the flat representation of the circuit. Figure 5.8 shows, that we need five treenodes for the hierarchical description of `Tree[2]` which also represents seven basic gates. If we would build the DAG for the next level, i.e. for `Tree[3]`, we would create only one additional treenode. But the whole DAG structure for `Tree[3]` represents 15 basic gates.

In general we have $3 + n$ treenodes in the DAG of the 2^n bit comparison tree ($n > 0$) which consists of $2^{n+1} - 1$ gates, i.e. we have a description of logarithmic size.

5.3 A Larger Design: The 16 Bit Conditional Sum Adder

While we have loaded some small designs in the previous sections in order to explain the algorithm for building the DAG structure, we will now use a larger circuit to show some other useful function concerning the circuit hierarchy. Especially you will see, how the equation system for the wire variables is extracted and solved.

Load Circuit Hierarchy



We will explain the hierarchy operations with the help of a 16 bit conditional sum adder. First we have to load this element from our parameterized specification of the n bit conditional sum adder. To do this you have to select the entry `Load` from the submenu `-Circuit-` and press the left mouse button. From the list window select the entry `CSA[n]` and press the left mouse button if the name is highlighted.

Again you you have selected a parameterized description with a free parameter `n`. In order to build the hierarchical data structure this parameter has to be set to a nonnegative integer value within the input window:

```
Enter Parameter Values for "CSA[n]":<16    >
```

Type in `CSA[16]` or simply `16` to set the value of `n`=16. After confirming this entry with the return key the system will display the following informations in the message window:

```
Reading Circuit Description of "CSA[16]"
Reading Dag-File:  CSA[16]
Reading Dag-File:  SEL[9]
Reading Dag-File:  SEL[4]
Reading Dag-File:  SEL[2]
Reading Dag-File:  SEL[1]
Reading Dag-File:  MUX
Reading Dag-File:  SEL[5]
Reading Dag-File:  SEL[3]
Reading Dag-File:  CSA[8]
Reading Dag-File:  CSA[4]
Reading Dag-File:  CSA[2]
Reading Dag-File:  CSA[1]
Reading Dag-File:  XOR2
Reading Dag-File:  INV
Reading Dag-File:  AND2
Reading Dag-File:  OR2
```

```
Total Number of Dag-Files:  16
Last Modification Date:  Thu Jan 25 17:08:34 1996
CtnTypes:  (45 x 38) - Matrix
Circuit "CSA[16]" loaded, Total Data Size:  64813 Bytes
```

You already know the meaning of these messages from the previous reading with the exception of the line `CtnTypes:  (45 x 38) - Matrix`. This indicates that a 45×38 matrix has been setup from the equations about the wire variables. The position of this message in the output during the loading of the circuit implies that the system of equations over the wire variables is solved after the DAG structure has been built. At this time all elements which directly depend on the circuit parameters are evaluated. For the wire variables this means that their indices (if existing) are replaced by nonnegative integer values. The value of a variable depends on the values of the circuit parameters and the values of the other wire variables. This second dependency is calculated now by extracting and solving a system of linear equations over the variables.

In the following we will show how these equations can be derived from the hierarchical specification.

5.3.1 A System of Linear Equations for the Wire Variables

To explain how the equations about the wire variables are extracted we examine the hierarchy transitions from the instances on one level to their successor treenodes as it is shown in figure 5.9. There we have the hierarchy level of the treenode `CSA[8]`, which contains three instances, namely two occurences of `CSA[4]` and one `SEL[5]`. After the evaluation step we have the wire variables `@high[8]`, `@low[8]`, `@highsum[8]`, `@lowsum[8]` and `@carry` in the corresponding Cgraph. Each of these variables represents a certain number of parallel binary wires of width 1.

On the next lower level `CSA[4]` we have the variables `@high[4]`, `@low[4]`, `@highsum[4]`, `@lowsum[4]` and `@carry`. In the Cgraph of `SEL[5]` there are the variables `@selinL[5]`, `@selinR[5]`, `@seloutL[5]`, `@seloutR[5]` and

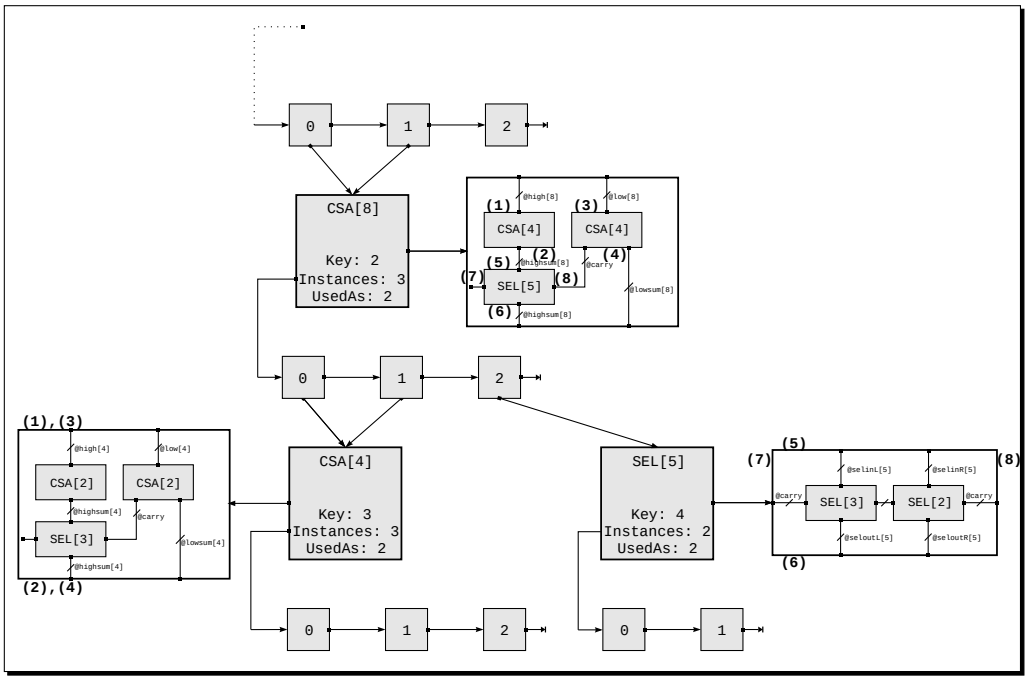


Figure 5.9: Deriving equations about the wire variables

again `@carry`.

For the correct connection of the wires on one hierarchy level to the wires on the successor level, it is necessary that all the single binary wires can be connected in the right way. This is only possible, if the number of single wires at level i is equal to the number of single wires at level $i + 1$. This leads to the condition that the sum of wire variables connected to the outer border of an instance is equal to the sum of wire variables connect to the schematic border (inner border) at the corresponding side.

In figure 5.9 this means that the variable `@high[8]` which is connected to the northern border of the left instance `CSA[4]` in the Cgraph for `CSA[8]`, must be equal to the sum `@high[4]+@low[4]` which are connected to the northern border of the Cgraph for `CSA[4]`. In figure 5.9 this condition is indicated by the small number 1 which identifies these borders as the first equation.

The corresponding transition at the northern border of the left instance

CSA[4] gives us the equation

(1) @high[8] = @high[4] + @low[4].

In the same way we examine the southern border of this instance (condition 2). Here we have the variable @highsum[8] at the level CSA[8] and the variables @highsum[4] and @lowsum[4] at the southern border of the schematic for CSA[4]. This leads to the equation

(2) @highsum[8] = @highsum[4] + @lowsum[4].

The northern border of the right instance CSA[4] gives us the third condition. Note that the right side of this equation is given by the same sum of wire variables as in the first equation, because this is a second occurence of the treenode CSA[4]. We receive

(3) @low[8] = @high[4] + @low[4].

If you examine the remaining borders of the instances in CSA[8] you will easily find the equations

(4) @carry + @lowsum[8] = @highsum[4] + @lowsum[4]

(5) @highsum[8] = @selinL[5] + @selinR[5]

(6) @highsum[8] = @seloutL[5] + @seloutR[5]

and the trivial equation

(7,8) @carry = @carry.

If we only get such equations between variables, the equation system would have the trivial solution, that all variables are set to 0. This is not the case for the lowest hierarchy transitions as shown in figure 5.10.

Here the northern border of the left CSA[1] implies the equation

(1) @high[2] = 2.

The remaining borders of the two CSA[2] instances give us the equations

(2) @highsum[2] = 4

(3) @low[2] = 2

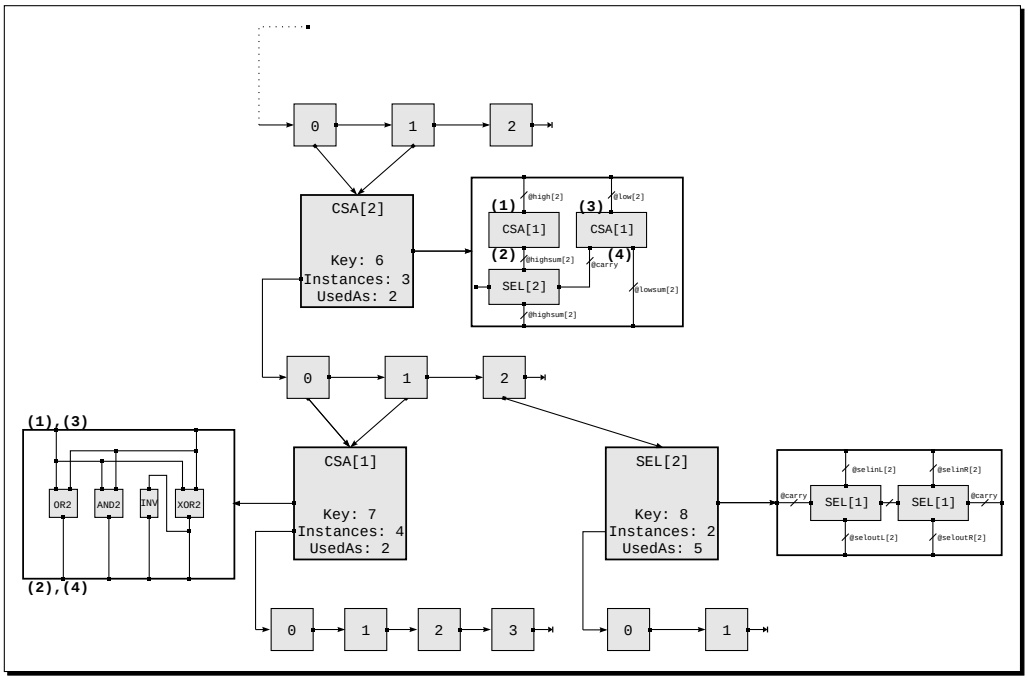


Figure 5.10: Equations for the wire variables at the lowest hierarchy level

and

(4) `@carry` + `@lowsum[2]` = 4.

When the whole equation system is solved, these basic values of the variables are propagated to the higher levels of the hierarchy. As the message `CtnTypes: (45 x 38) - Matrix` indicates the whole equation system for the `CSA[16]` includes 45 equations over 38 variables.

If the system can solve the equation system the wire variables are substituted by the calculated values and now shown as integer values. After the loading of `CSA[16]` you will see the highest hierarchy level as it is shown in figure 5.11. All wires have a constant width as well as all parameters of the macro names are evaluated to integer values.

There are some cases, where the system of equations can not be solved:

- all values for the wire variables must be nonnegative integer numbers.
- the system is indeterminable, i.e. there are some variables without a

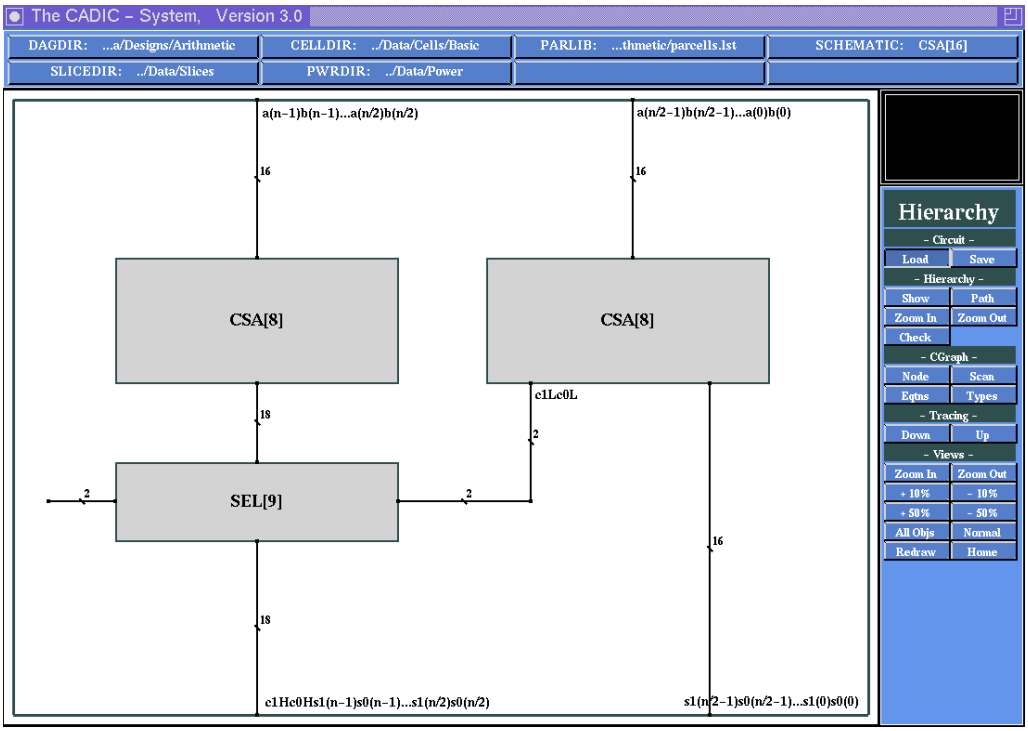


Figure 5.11: Evaluated wire variables at the top level of `CSA[16]`

unique solution.

- the system has no solution.

We will illustrate the first case with the help of the following example specification. If you look at the hierarchy transition shown in figure 5.12, you can extract the following equations from the borders of the subcircuit `S`:

$$\texttt{@s} + \texttt{@s} = \texttt{@t}$$

$$3 + \texttt{@u} = 1 + \texttt{@t}$$

$$\texttt{@u} = 1$$

The trivial equation at the western border of `S` can be neglected. If we solve this system of three equations over three variable, we get the solution

$$\texttt{@s} = 1.5,\ \texttt{@t} = 3,\ \texttt{@u} = 1$$

It is obvious that the value for `@s` is not a legal wire width. In such cases the system will respond with a message

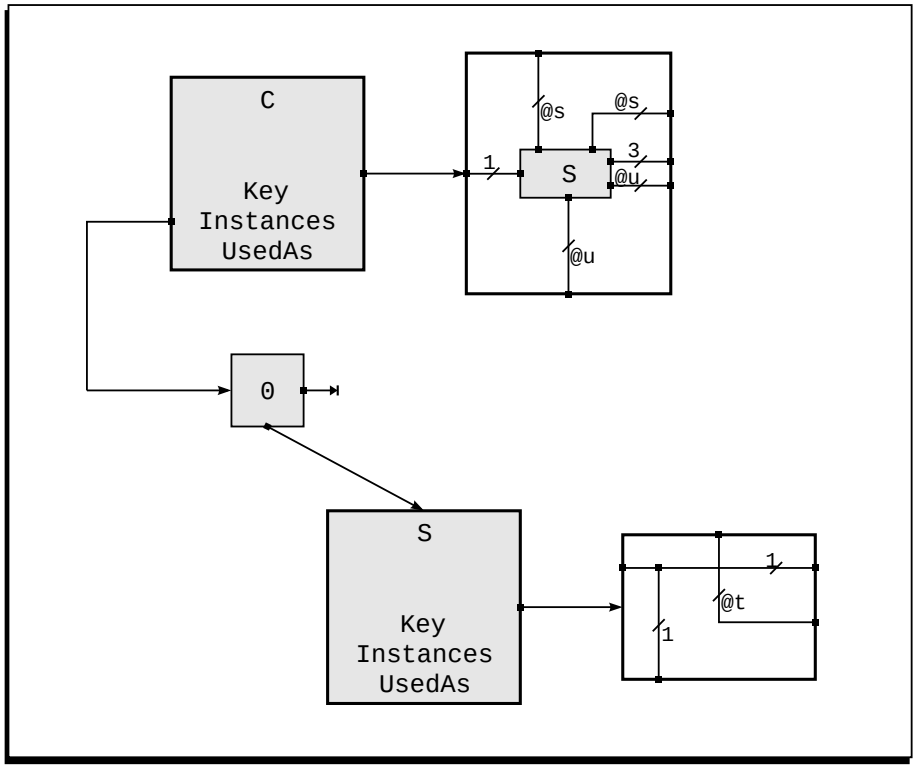
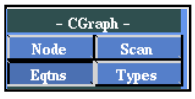


Figure 5.12: Specification with illegal solution for the system of linear equations over its wire variables

```
Error:  Value for Wire Variable @s is illegal
```

The DAG structure for the current circuit can not be used for any other design step in the system. You first have to fix the specification by calling the schematic editor.

Display Equations for Wire Variables



With the help of the function `Eqtns` from the submenu `-CGraph-` you can control the equations, which are implied by the hierarchy transitions at the borders of the instances. After selecting this menu point the system will display the equations within the schematic as it is shown in figure 5.13.

In figure 5.13 you see the highest level `CSA[16]` of the 16 bit conditional sum adder. According to the considerations from above you can see the equation

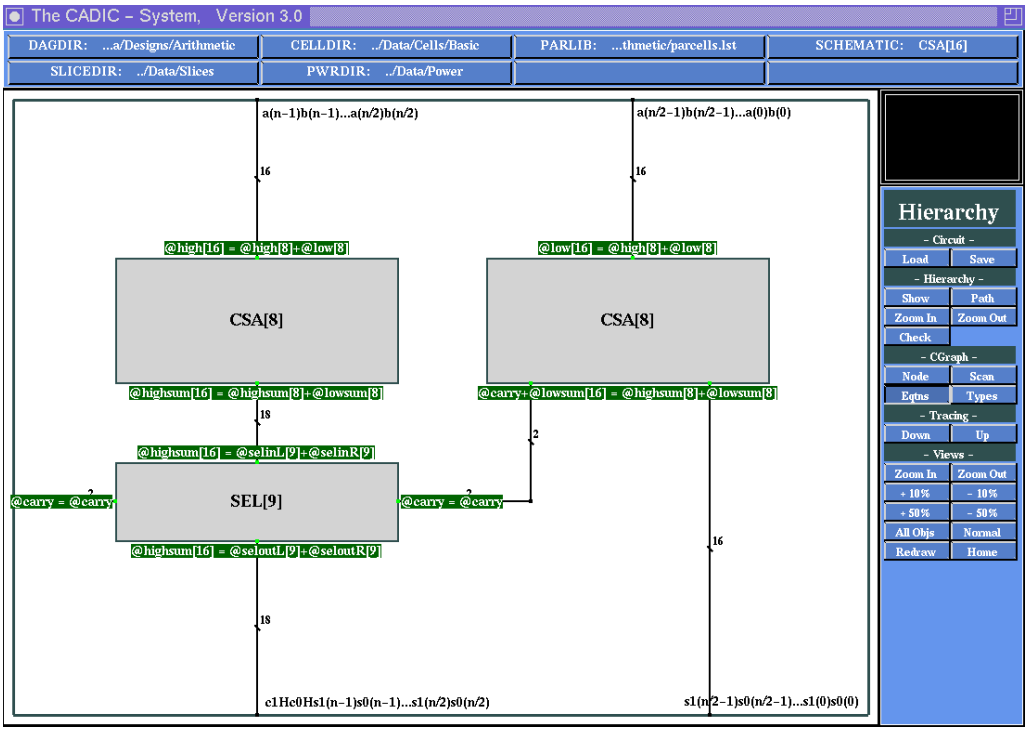


Figure 5.13: Display of the equations about the wire variables

`@high[16]` = `@high[8]` + `@low[8]`

at the northern border of the left `CSA[8]` and

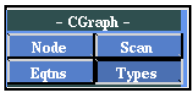
`@highsum[16]` = `@highsum[8]` + `@lowsum[8]`

at the southern border. The left side of the equation represents the sum of the wire variables at the current hierarchy level and the right side is the sum of the variables at the successor level.

Note that at the borders of basic cells there are no equations shown, because each wire has a constant width and is connected to exactly one pin of the instance.

After you have executed the function `Eqtns` the system automatically returns to the menu selection mode. The display mode for the equations will be active until you turn it off by selecting the menu point `Eqtns` again. Then the display mode is toggled and after a refresh of the workarea the equations at the borders of the instances will disappear.

Display the Wire Variables



After the system of equations has been solved, the wire variables are substituted by the calculated values. If you want to check the values of the variables, you can use the function `Types` from the submenu `-Cgraph-`. After the selection of this menu point the names of the variables are displayed at every wire as it is shown in figure 5.14.

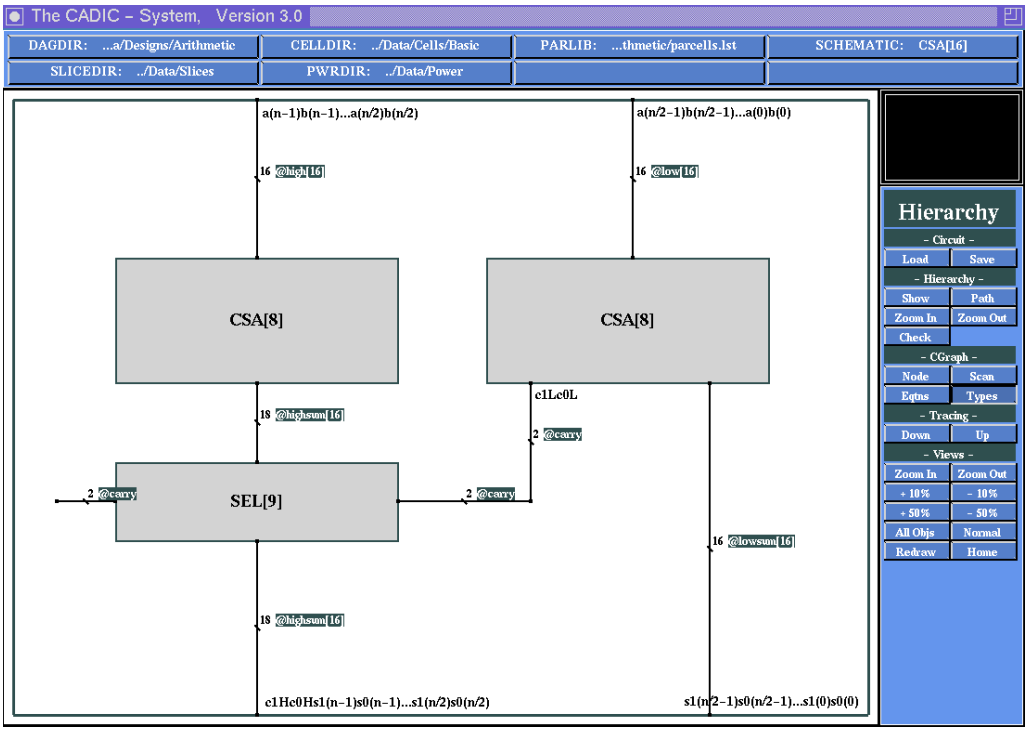


Figure 5.14: Display of the names of wire variables

Together with the display mode for the equations this function helps you to check the values of the wire variables. These two functions are very helpful to check the specification especially, if you use them in cooperation with the tracing functions, which are explained in section 5.6.

After you have executed the function `Types` the system automatically returns to the menu selection mode. The display mode for the variable names will be active until you turn it off by selecting the menu point `Types` again. Then the display mode is toggled and after a refresh of the workarea the names of the variables at the wires will disappear.

5.3.2 A Note on Wire Variables

The most important aspect of the specification level of the graphical editor is the possibility to describe whole families of circuits with a fixed set of schematic inputs. This can be realized by the use of parameterized macro cells and bundles of wires. The width of a bundle of wires can be given in two different ways:

- by an arithmetical expression over the set of circuit parameters
- by a formal wire variable which may be indicated by arithmetical expressions

If we only allowed the use of arithmetical expressions for the width of wires, this would have the disadvantage, that the user had to derive complicated expressions within a recursive circuit description. This can easily lead to mistakes in the specification. Here the use of formal wire variables has ernormous advantages:

- the user chooses a variable name for a bundle of wires and leaves it to the system to derive the values of the variables after the circuit parameters are given integer values.
- with the help of wire variables the user can describe algorithmic structures which are the base of some circuits. Especially these structures can be specified independently of basic operations.

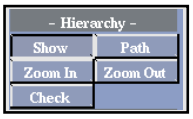
  An example for such a generic circuit is an odd–even-mergesort network which can be specified independently of the type of elements it should sort. The type (e.g. boolean, 32 bit integer, etc.) of the elements is simply represented by an appropriate wire variable `@t`. To configure the sorting network for a certain type of elements, the user has to give a basic compare function for two elements.
- the wire variables can be characterized as global variables. This can be used to setup relations between independent components of a circuit specification. Especially this can be used to describe reusable structures and configure them within a larger description be specify-

ing additional equations about the wire variables.

5.4 Visualization of the DAG Structure

In this section we introduce some functions which will visualize the DAG structure of the circuit hierarchy according to the transitions from instances on one level to the treenodes at the next lower level.

Display of the DAG Structure



The function `Show` from the `-Hierarchy-` submenu will display the hierarchical structure of the loaded circuit. After the selection of this menu point a new window frame will appear on your screen which can be moved around and dropped by pressing the left mouse button. This behaviour depends on your window manager (e.g. `twm` or `fvwm`) and in some cases the window will be automatically displayed (e.g. `mwm`).

In the open window the DAG structure of the circuit is displayed as it is shown in figure 5.15 for our example of the 16 bit conditional sum adder.

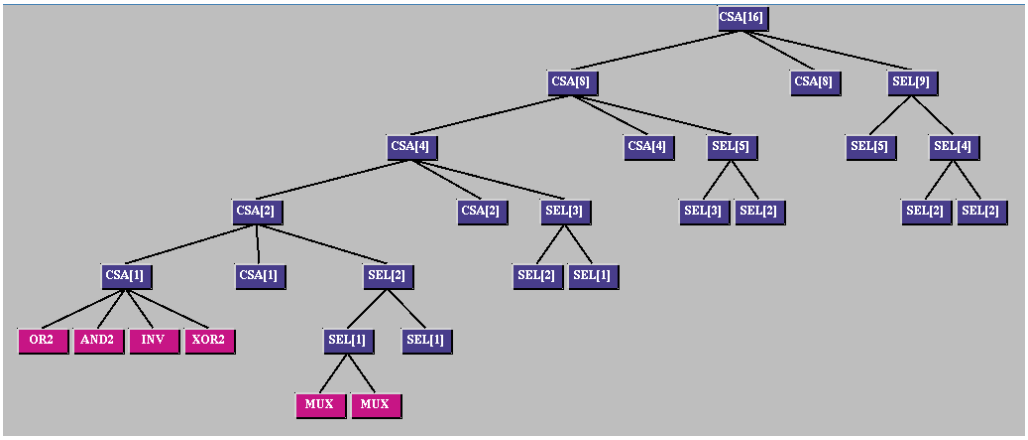


Figure 5.15: Visualization of the DAG structure for `CSA[16]`

In the window, the hierarchical structure of `CSA[16]` in form of the DAG data structure is shown. The folded structure is broken up for better visualization, but each node is refined only once. For example, there are two

nodes CSA[8] as successors for CSA[16] but only the left one is refined. Internally each node in the visualization corresponds to exactly one treenode, whereas nodes with the same name represent the same treenode.

In our example the root of the structure is the instance CSA[16], which is followed by two instances CSA[8] and one instance SEL[9]. CSA[8] consists of two instances CSA[4] and one instance SEL[5], while SEL[9] is composed of SEL[5] and SEL[4], etc. In the lowest level of the structure, it is shown that CSA[1] contains four basic cells: OR2, AND2, INV and XOR2 as well as SEL[1] contains two instances of the basic cell MUX. Thus, the specification of a design consists of one recursive and several simple equations. Such a refinement also defines a very compact hierarchical representation.

Since e.g. a 16 bit adder consists of two 8 bit adders, four 4 bit adders etc., we can represent it by storing the representation of each part only once.

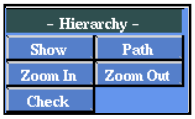
Changing the Visible Area

The size of the window for the visualization of the hierarchical structure is fix, i.e. it does not depend on the size of DAG. But you can scroll the visible area and select the part of the DAG you are interested in.

As you can see in figure 5.16 the window for the DAG structure is devided into a 3×3 grid. If you press the second mouse button within one of these nine parts of the window, the corresponding action from figure 5.16 is performed.

For example if you want to scroll the DAG structure to the right, you must press the second mouse in the left field in the middle row.

Zooming the Visualization of the DAG



If you want to change the graphical representation of the DAG, you can resize the nodes with the help of the functions Zoom In and Zoom Out from the -Hierarchy- submenu.

You can enlarge the size of the nodes by selecting Zoom In. You can repeatedly press the left mouse button until you obtain an ideal size of DAG.

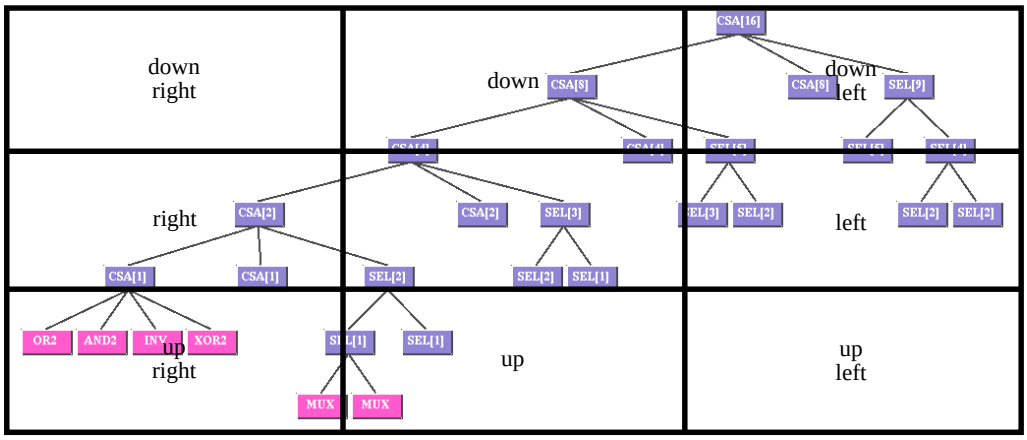


Figure 5.16: Grid of the hierarchy window for scrolling the visible area

In the same way you can shrink the size of the node with the help of the function `Zoom Out`. If the size of the nodes becomes to small, the names of the treenodes are no longer displayed. Sizing down a very large DAG can give you a hint about the balance of the hierarchical description, i.e. if the hierarchy is very deep (many different hierarchy levels) or very wide (many instances at one hierarchy level).

Note: when you use the functions `Zoom In` and `Zoom Out` from the submenu `-Hierarchy-` within a scrolled DAG, this scrolled DAG will first return to its initial position, and then perform the zooming operation.

If you want to close the hierarchical structure window, move the pointer into this window and press the right mouse button. This operation terminates the `Show` mode, and you return to menu selection mode again.

5.5 Hierarchy Check

After you have completed the design of a circuit you should check whether its hierarchical specification is correct or not. Of course, the system can only help you to check the syntactical correctness of a circuit. In our case this means that CADIC provides a function, which checks, whether all wires connected to an instance have a corresponding wire inside the Cgraph of this instance. With this function you can not check, if your circuit computes

the desired values. The task to check the logical or timing behaviour of the circuit falls to simulation tools.

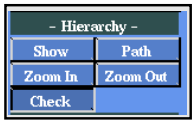
Because the number of external connectors at each border of an instance which may be a basic cell, macro or parameterized macro, must be equal to the number of its internal connectors, this is a powerful check to find out whether the recursive specification is free of errors.

Errors can occur, if for example the designer uses arithmetical expressions for the width of wires within a recursive description. The used expressions may not be correct for all iterations of the recursion, such that there are different values of "outer" and "inner" bundles of wires at some instance borders.

If you use variables for the width of wires, the values of these variables are calculated by solving an appropriate system of linear equations. This system may not have a unique solution, such that there are conflicting equations for some variables. These conflicts result in different numbers of wires at the instance borders. Errors can also occur, if you use variables and arithmetical expressions together in a description.

The hierarchical check function helps you to find instance borders, where the wires are not connected properly. The detected border must not necessary contain the errornous specification, because the same variable may be used elsewhere in the hierarchical design.

Check the Hierarchical Specification



In order to check the loaded circuit `CSA[16]`, move the pointer onto the push button named `Check` from the submenu `-Hierarchy-` and press the left mouse button.

If the specification, i.e the recursive definition, is correct, the following information is displayed in the message window:

```
Specification is correct.
```

Note: If the specification is correct, it only shows that the specification is

correct for the given parameter values, but not in general. For instance, in the above example the syntactical correctness of the specification for `CSA[16]` and its lower hierarchy levels, i.e. `CSA[8]`, `CSA[4]`, `SEL[9]`, ..., is checked, but it is not sure, whether `CSA[32]` is also correct. It means that you should check a parameterized specification for a set of parameter values.

If there is an incorrect specification in an instance, the corresponding border of the instance will change the color from red to pink, and two numbers will be displayed at both sides of the border, one being the number of internal wires and the other being the number of external wires. At the same time, a corresponding error message will be displayed in the message window.

Here, we give an example to explain the function `Check`. Suppose that you made an error during the specification of the schematic `CSA[n]`, as shown in Figure 5.17, where the small wire from the western border of `SEL[n/2+1]` is forgotten. If you compare figure 5.17 to that from our editor session, you see that this small wire is needed to project the carry wires to an open end. The width of this wire is denoted by the variable `@carry`.

If you want to control the behaviour of CADIC in this situation, you can go back to the graphical editor, load the schematic `CSA[n]` and remove this wire with the help of the function `Delete` from the submenu `-Wires-`. After that save the schematic back to `DAGDIR` with the function `Save` from the submenu `-Schematics-`. Then you can return to the `Hierarchy` menu by pressing the right mouse button within the editor submenu and selecting the hierarchy menu again.

If we load the circuit `CSA[16]` again and check it, the following informations are displayed in the message window:

```
Error in hierarchical Specification:

In Treenode CSA[8] Instance SEL[5] has 12 Pins and 10 Pads

on Side Northside

Error in hierarchical Specification:
```

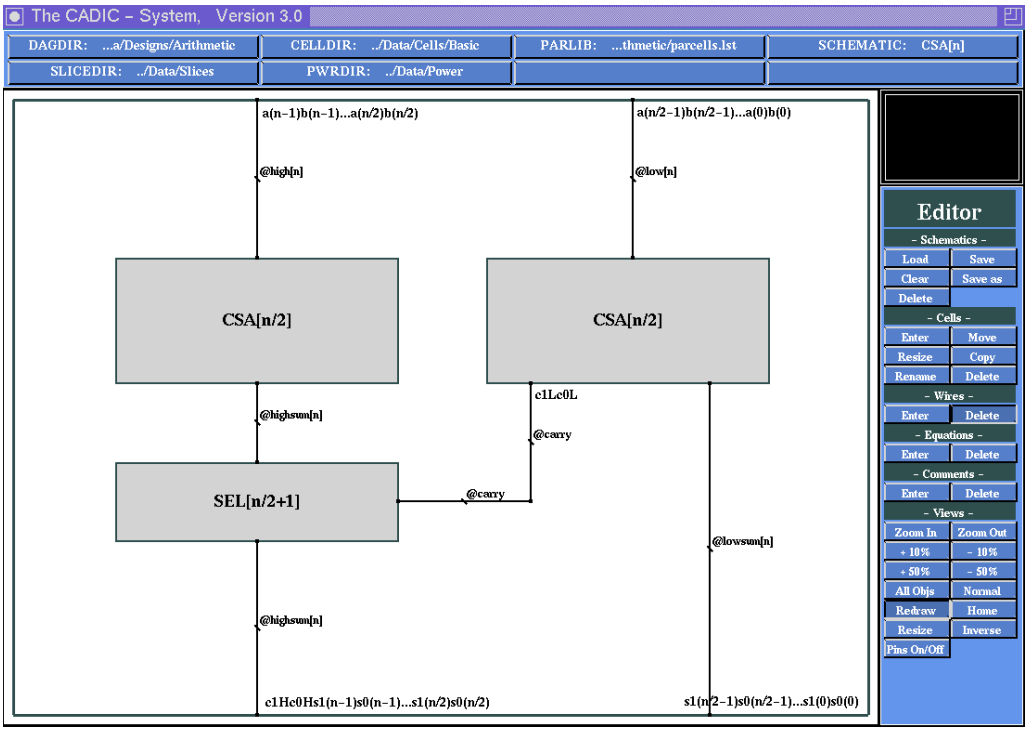


Figure 5.17: An error in the schematic `CSA[n]` at the western border of the instance `SEL[n/2+1]`

```
In Treenode CSA[8] Instance SEL[5] has 12 Pins and 10 Pads

on Side Southside

Error in hierarchical Specification:

In Treenode CSA[2] Instance CSA[1] has 2 Pins and 4 Pads

on Side Southside

Error in hierarchical Specification:

In Treenode SEL[2] Instance SEL[1] has 0 Pins and 2 Pads

on Side Eastside
```

and some more. The check report shows that there are errors in the hierarchical specification for `CSA[16]`. You can already see this in the schematic of the topmost hierarchy level. Here the carry wire from the right instance `CSA[8]` to the eastern border of the selection subcircuit `SEL[9]` has disap-

peared. This means that the system has set the value of the variable `@carry` to 0 and then has removed this wire. The equation system derived from the hierarchy transitions implies that `@carry`=0, but this leads to further conflicts in the hierarchical descriptions. These conflicts are given in the check report.

The first message tells you that there is an error in the hierarchy level `CSA[8]` at the instance `SEL[9]`. According to our manipulation from above this instance has 12 pins, i.e. there are 12 single binary wires connected at the "outer" border of the instance, and 10 pads, i.e. only 10 binary wires are connected to the border of the corresponding schematic.

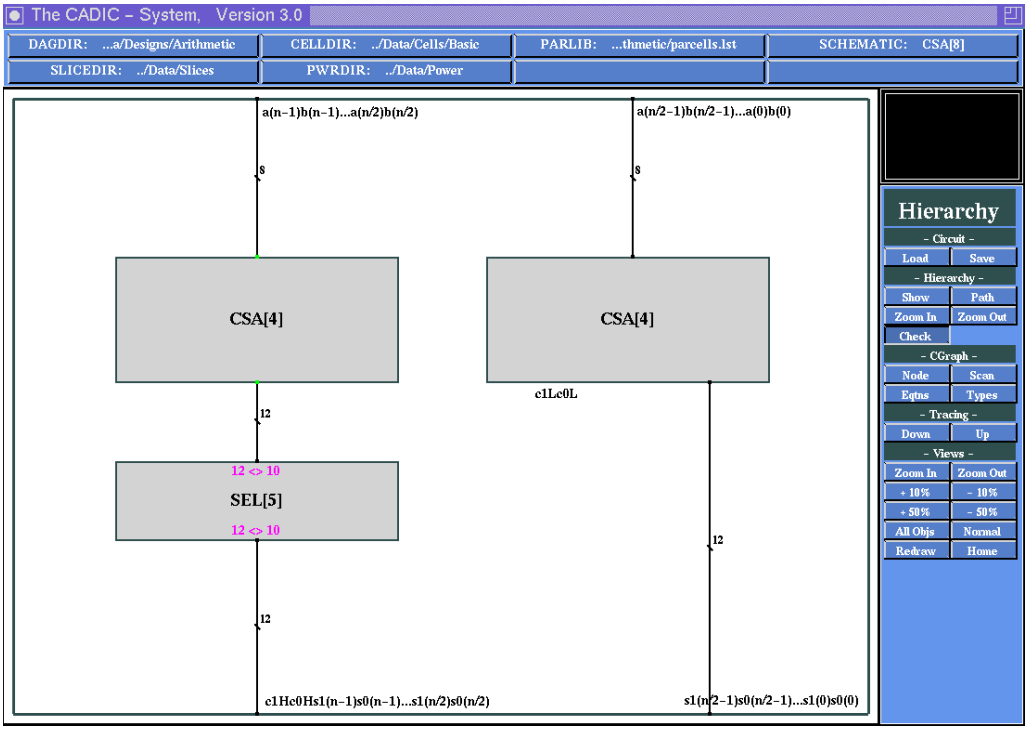


Figure 5.18: Visualization of specification errors at the hierarchy level `CSA[8]`

You can see in figure 5.18 that there is an error prompt `12<>10` at the northern and the southern border of the instance `SEL[5]`. The symbol `<` represents the number of external connections (pins), and the symbol `>` denotes the number of internal connections (pads). The error prompt

`12<>10` indicates that there are twelve external and ten internal connections.

Note: because the schematic in figure 5.18 is not at the highest hierarchy level of the circuit, you have to go down into this level first. This can be done with the help of the tracing functions which are explained in section 5.6. In the list of messages from above there are further errors at lower hierarchy levels. To look at the corresponding instance borders you must use these functions for tracing through the hierarchy levels.

⇝ 5.6

If there are some errors in the specification of your circuit, you must return to the `Schematic Editor` and correct these errors. In the menu selection mode, you can obtain the main menu by pressing the right mouse button in the menu window. Then call the editor by selecting `Schematic Editor` from the main menu.

Note: Only if the specification is correct according to the hierarchy check, you can use the integrated tools for further processing the design.

5.6 Navigation through the Hierarchy

With the help of the functions from the submenu `-Tracing-` you can conveniently trace through the DAG structure of a hierarchical design. *Tracing down* through the hierarchy means following the pointers from an instance at one level to the corresponding treenode at the next lower level. Each tracing down operation is pushed onto a stack, such that the complete path from the root of the design to the current treenode is stored. *Tracing up* means to go back one step on the path, i.e. to pop the upmost element from the trace stack and make it the current treenode.

Tracing Down

From the highest level of our 16 bit conditional sum adder `CSA[16]` we will now trace down into one of its instances. Because the trace stack is empty for the root of the DAG you can only use the trace down function. Select the entry `Down` from the submenu `-Tracing-` and press the left mouse button. Then you are in cell selection mode in order to pick the instance,

into which you want to descend. As you already know from the editor session the currently selected instance is highlighted in blue during your motion through the workarea.

Select the instance `SEL[9]` in this way and press the left mouse button. The schematic for `CSA[16]` disappears from the workarea and the Cgraph for `SEL[9]` is displayed instead (c.f. figure 5.19).

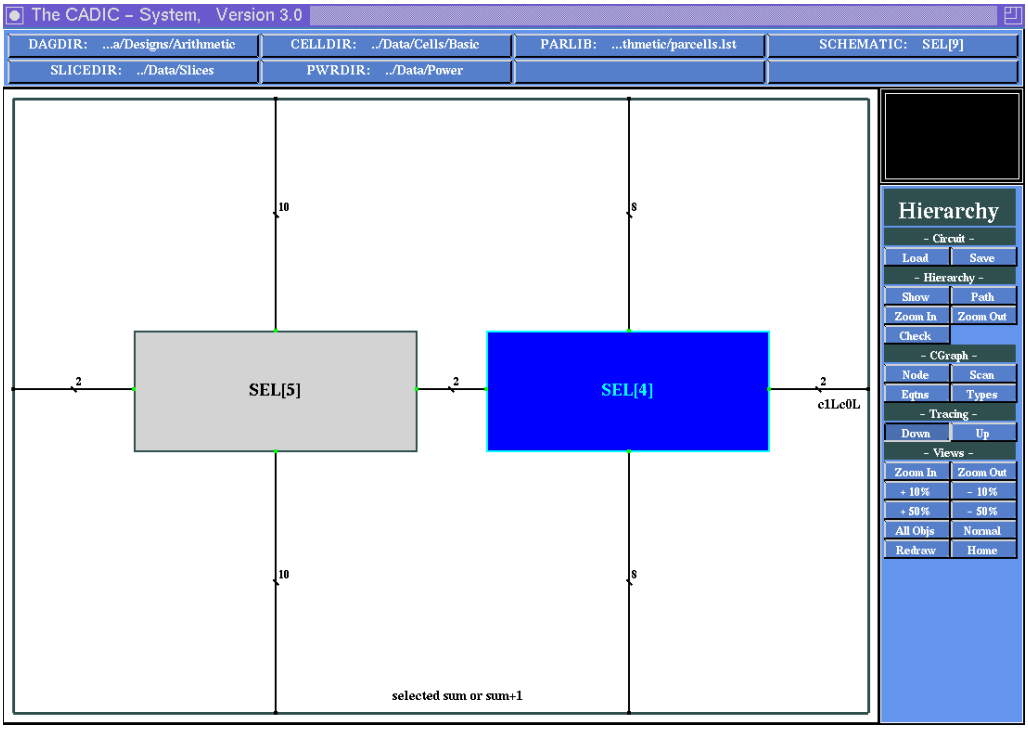


Figure 5.19: Tracing down from `CSA[16]` into `SEL[9]`

The new schematic shows that the macro cell `SEL[9]` consists of the two instances `SEL[5]` and `SEL[4]`. The cell selection mode is still active, such that you can choose another instance to trace into. For example, you can further trace into the instance `SEL[4]` by moving the pointer near to it and then confirming the selection with the left mouse button. Now the schematic `SEL[9]` is replaced by its successor `SEL[4]` which is given by the same Cgraph but evaluated with a different parameter value.

From this level you should continue to trace down. In `SEL[4]` select the left instance `SEL[2]` and in the following schematic pick the right instance

SEL[1]. After you have selected this instance the schematic shown in figure 5.20 appears in the workarea. It shows that SEL[1] consists of two basic cells MUX. Now you are at the lowest level of the hierarchy of our 16 bit conditional sum adder CSA[16]. The cell selection mode is still active which you can watch by moving the pointer from one multiplexer to the other. But selecting one of these instances by pressing the left mouse button has no effect, because there is no further hierarchy level to be displayed.

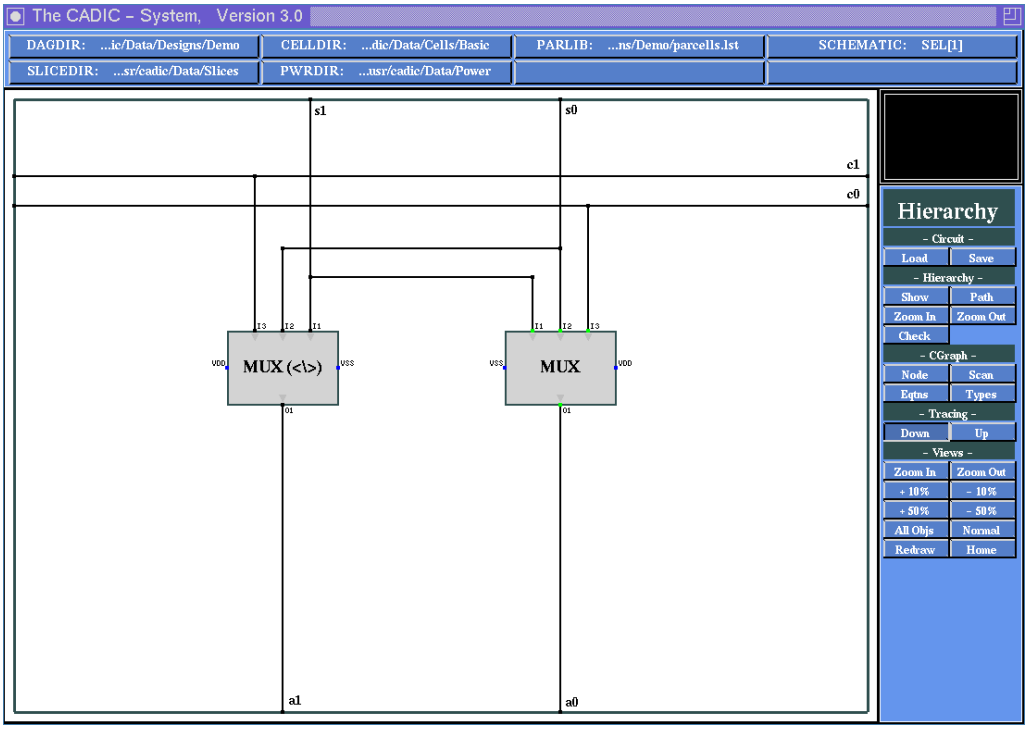


Figure 5.20: Tracing down from SEL[2] into SEL[1] and arriving at the lowest hierarchy level

If you want to abort the function for tracing down, you can move the pointer into the workarea and press the right mouse button. Then you return to the menu selection mode again.

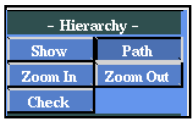
Tracing Up

If you are at a level of the DAG structure except the root level you can use the function for tracing up to go back some levels on the tracing path. In our

example from above we have reached to lowest level SEL[1] of the CSA[16]. To go back one step and show the schematic of the previous hierarchy level select the entry Up from the submenu -Tracing- and press the left mouse button. Then the level SEL[2], where we have selected the right instance SEL[1] is displayed in the workarea.

You can perform further trace up steps by selecting this entry again and again until the trace stack is empty, i.e. you have returned to the root level of the hierarchy.

Show Tracing Path



While you are tracing the hierarchical design you can view the current tracing path in order to understand the folded structure of the circuit. To do this select the entry Path from the submenu -Hierarchy-. After that a new window appears on the screen. Depending on your window manager you can move it anywhere and drop it by pressing the left mouse button.

This window named Trace View shows all schematics, which have been selected on the current tracing path. In our example, where we traced to down to the lowest level SEL[1] and then one step up you see four schematics in the window. In each schematic except of the last there is one instance highlighted in blue. This indicates the selected instance at this level, for which the next schematic represents its Cgraph. The last shown schematic is the current level of the hierarchy which is displayed in the workarea (c.f. figure 5.21).

In figure 5.21 you can see that the current trace path is CSA[16] $\longrightarrow$SEL[9] $\longrightarrow$SEL[4] $\longrightarrow$SEL[2].

After the trace view window is displayed on the screen you return to the menu selection mode and can perform further tracing operation. For example select the entry Down from the submenu -Tracing- again and step into the left instance SEL[1] of the current hierarchy level. Watch the trace view window, where the schematics are rearranged such that the new level is also shown. In the previous level SEL[2] you can see the selected instance

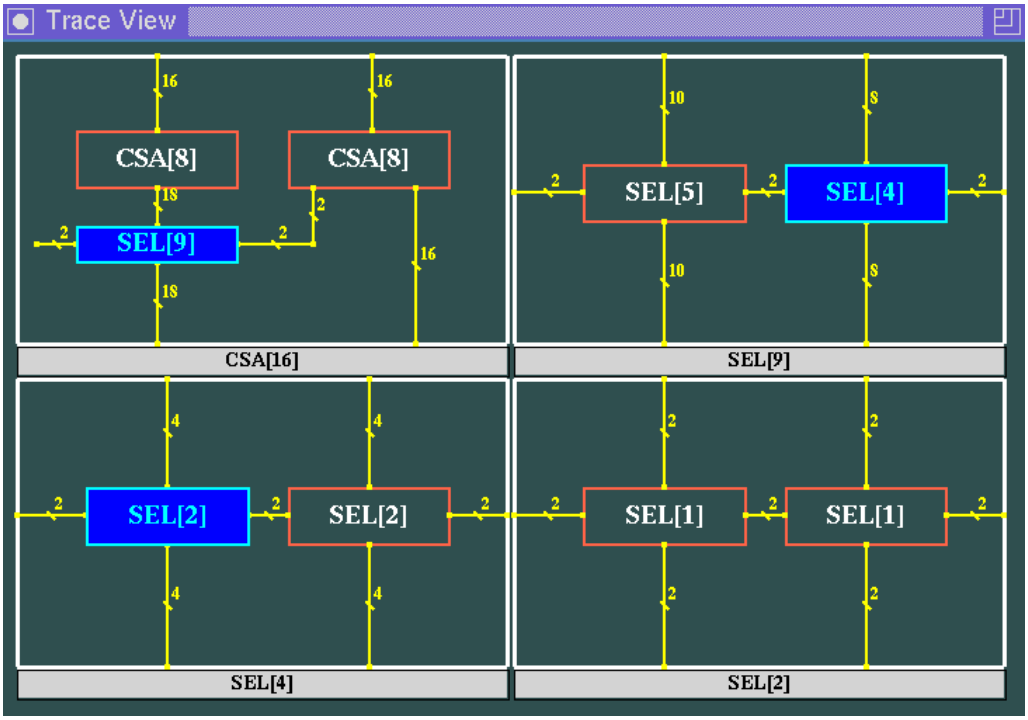


Figure 5.21: Trace view window with path from `CSA[4]`, then `SEL[3]` to `SEL[1]`

`SEL[1]` now being highlighted in blue.

In the same way you can trace up on the current tracing path. If you do this you can watch the corresponding schematic disappear from the trace view window and the previously highlighted instance is redrawn in normal mode.

If you want to close the trace view window, move the pointer to the submenu `-Hierarchy-` and select the button `Path` again. Then, the `Trace View` window disappears and you return to the menu selection mode again.

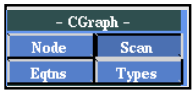
The tracing operations are essential for the work with CADIC . They allow you to inspect the hierarchy of a design in every desired way. In combination with the visualization of design results they are a powerful method for analyzing a circuit. We will use the tracing functions with nearly every integrated tool as you will see in the following chapters.

5.7 Examining the Cgraph

The Cgraph represents the topographical description of a circuit and gives its precise characterization which is sufficiently abstract by suppressing geometrical and physical details and which is sufficiently concrete to control the arrangement of cells and the global routing of wires. The description contains sizing informations as well as exact connections and special layout blocks. This information given in the Cgraph is used by the integrated synthesis tools (layer assignment, power supply, place&route, etc.). Whereas the tools for analyzing the circuit (logic simulation, timing analysis, etc.) do not depend on the topographical information, but they use the logical structure of the circuit which can be extracted from the Cgraph description.

The functions we introduce in this section help you to examine the Cgraph structure of a given circuit. Normally you will not need to use these functions, they are for checking the Cgraph structure, if you write your own programs to create a Cgraph. If there are some errors in a design, the `CGraph` functions can be used to check it and trace the design so as to help find out these errors.

Scan of a Hierarchy Level



With the function `Scan` from the submenu `-CGraph-` you can scan the Cgraph of the currently displayed hierarchy level. After activating this function you are in node selection mode. You can notice that by moving the pointer near to a node of the current Cgraph, for example to the corner of an instance, to a pin or the branching of a wire, etc. If the distance to a node is less than a certain tolerance value there will appear a small square surrounding the node in order to mark it as the currently selected one.

In our example of the 16 bit conditional sum adder we are at the topmost hierarchy level. Move the pointer near to the first pin at the southern border of the right instance `CSA[8]`. The wire connected at this pin represents the carry wires from the lower part of the adder to select the appropriate version of the sum of the higher part. Press the left mouse button, if the pin is surrounded by the small square mark. Now, a new window, called

Cgraph Window, appears close to this node, as shown in Figure 5.22.

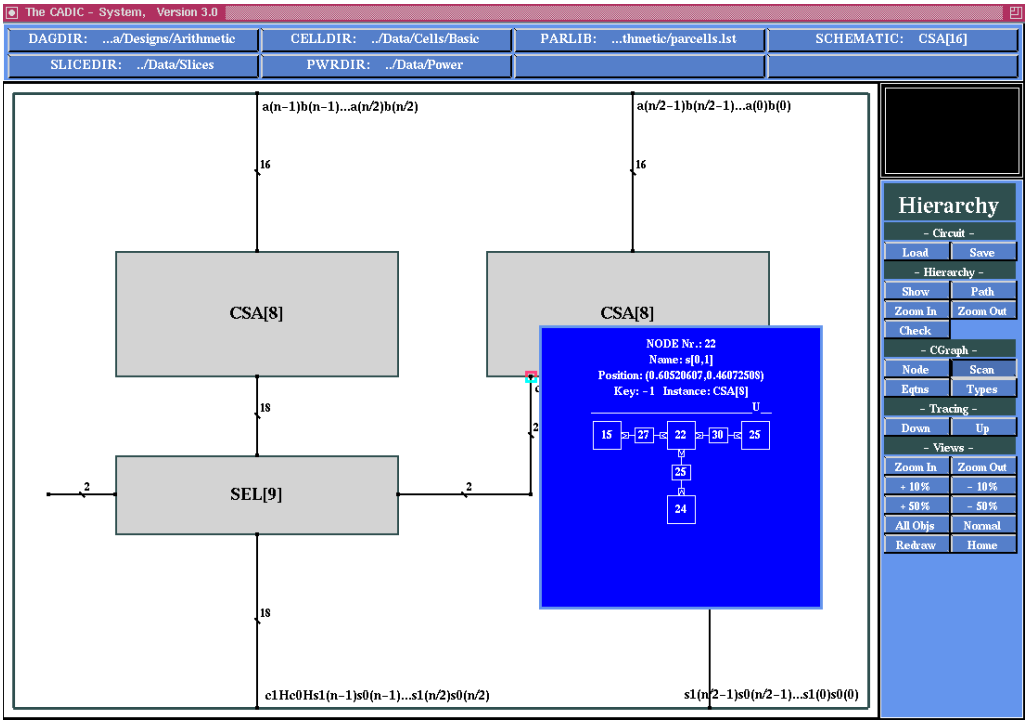


Figure 5.22: Cgraph window with node informations

The Cgraph window contains various Cgraph information about the selected node, such as its number, name, position, type and relation to the adjacent edges of the graph and the neighbour nodes. In detail there are the following informations shown in the Cgraph window:

- *node number* – the internal number of this node in the Cgraph of the current treenode.
- *node name* – the name of this node. If the node is a pin from an instance, its name is the same as the pin name, for example, `I1` from the basic cell `AND2` or `n[0,15]` from the instance `CSA[8]`. If the node is a pad at the border of the schematic, the node name is composed by the side to which it belongs and an interval that describes the position and the width of the pin. The order of the interval is from left to right at the northern and the southern borders and from top to bottom at the eastern and the western borders of the schematic, such that the

second northern pad of schematic `CSA[16]` is named `n[16,31]`. If the node is a nominal node, such as a wire crossing or a corner of an instance or the schematic border, the node has the name `__dummy__` to indicate that its name is not of interest.

- *position* – the coordinate `(x,y)` of the node. Since the schematic is enclosed by the coordinates of its four corners which are indicated by (0.0, 0.0) for the upper left corner, (0.0, 1.0) for the lower left corner, (1.0, 0.0) for the upper right corner, and (1.0, 1.0) for the lower right corner, the positions of the nodes of the Cgraph are within these intervals. Note that two nodes must not be located in a same place.
- *key* – an index for the nodes to classify them to signal nets, i.e. all nodes belonging to the same signal net have the same key. This information is calculated while the data structure for the netlist is created. Initially all nodes have the key -1.
- *instance* – the name of the instance to which the node belongs. If the node is a nominal node, such as a wire crossing or a corner, the node points to no instance and the entry will be `NULL`.
- *type* – the type of the node. There are the following types of nodes classified:
  (a) `I` – Input pin
  (b) `i` – Input pad
  (c) `O` – Output pin
  (d) `o` – Output pad
  (e) `W` – Wire crossing, branch, knee
  (f) `C` – Corner
  (g) `S` – VSS (−)
  (h) `D` – VDD (+)

(i) `U` – Unspecified pin

(j) `u` – Unspecified pad

- *relation graph* – the numbers of adjacent edges and neighbour nodes. In the Cgraph window the small rectangles represent the edge which connected to the current node in this direction. The square at the end of this edge represents the node at this position in the Cgraph.

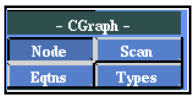
In our example, the node is the first pin at the southern border of the right instance `CSA[8]` in the schematic `CSA[16]`. Its internal number is `22`. The name is `s[0,1]` since the connected wire is a bundled wire of width 2. Its position coordinate is `(0.60520607, 0.46072508)`. The instance to which it belongs is `CSA[8]`. The key is `-1` because the netlist is not yet calculated. The node type is `U` because the node is a unspecified pin. In the relation graph, you can see that the western adjacent edge has the internal number `27` which is further linked to the node `15`. In the same way the eastern adjacent edge has number `30` and its end node is `25`. At the southern side the selected node has the neighbour node `24` which is connected via the edge with the number `25`.

From the current selected node you can scan to another adjacent node. From the node `22` you can easily move to the node `24` by simply pressing the left mouse button within the square surrounding the node number. Then the current Cgraph window is cleared and moved near to the newly selected node, where it displays the updated informations. In this way you can continue to scan through the whole Cgraph of the current hierarchy level.

If you want to close the Cgraph window and quit the `Scan` function, move the cursor into the Cgraph window and press the right mouse button. Then, the Cgraph window disappears and you return to the menu selection mode again.

Select a Cgraph Node by its Number

If you know the internal number of a node you want to examine, you can select the entry `Node` from the submenu `-Cgraph-`. Then an input window



pops up the working area with the following prompt (c.f. figure 5.23):

```
Please enter Node-Number :  <25     >
```

Type in the number of the node and press the return key. If a node with the given number exists the Cgraph window is opened nearby and shows the information mentioned above. If you would like to read the informations of another node, close the Cgraph window first by pressing the right mouse button within it. Then activate the menu entry `Node` again for a new node number.

Note: During this function you can not scan the whole schematic by pressing the left mouse button within the squares for the neighbour nodes. If you move the cursor into the square frame of an adjacent node in the Cgraph and press the left mouse button, the input window pops up again, and prompts you to enter the number for the next node.

If you want to close the Cgraph window and exit the function `Node`, you should first return to the input prompt situation. Move the cursor into the Cgraph window and press the left mouse button, the input window then appears. If you press the return key to the empty input window you return to the menu selection mode again.

5.8 Wrong Parameter Values?

Be careful with the values for the formal parameters of a specification. If you choose wrong values for the parameters, the system may not be able to further process the design.

For example the design of our n bit conditional sum adder from chapter 4 is only suited for parameter values $n = 2^k$. This is the case, because in the general equation `CSA[n]` we use two instances `CSA[n/2]`, i.e. we devide the parameter n by 2 at ech iteration of the recursive description. This is only possible, if the initial value for n is a power of 2.

If you would load the circuit `CSA[13]` the system would do this, but the DAG structure has errors at the hierarchy transitions. You can find these

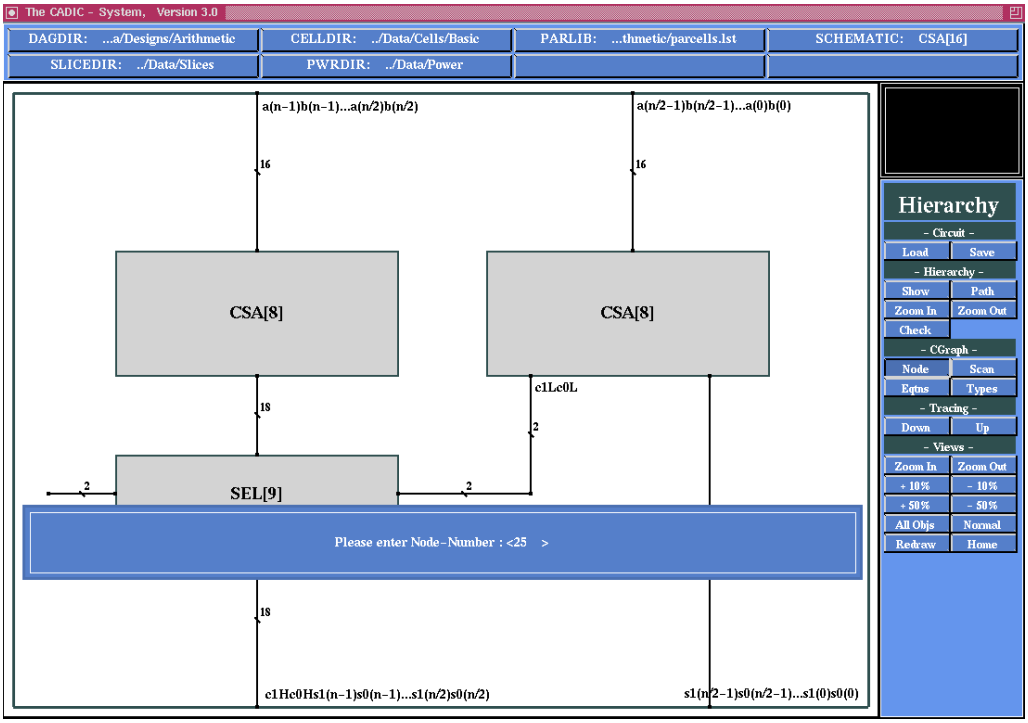


Figure 5.23: Input window with the prompt for showing the information of a specified node

⇝ 5.5

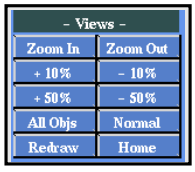
errors with the help of the check functions from section 5.5. You should always remember, for which set of parameter values your specification works fine. The specification level of the graphical editor provides a lot of functions, such that you also can manage more complicated recursions (not only reductions from n to $n/2$). The whole collection of expressions for manipulating the circuit parameters are given in section 12.

⇝ 12

5.9 Views

With the functions `Zoom In` and `Zoom Out` from the `-Views-` submenu you can select a viewport and change the graphical representation of your schematic.

Zoom In



If you activate the function `Zoom In` you are in viewport selection mode. Now move the pointer into the workarea and select a start point for a rectangular viewport. This must not be the upper left corner of the viewport, according to the selection of the second point it will be interpreted appropriately. After fixing the start point by pressing the left mouse button, move the pointer to the position of the opposite corner of the desired viewport. You notice a rubberbanding frame following the motion of the pointer. This frame indicates the size of the selected viewport and implies the factors by which the schematic will be scaled.

If you confirm the selection of the opposite corner by pressing the left mouse button, the schematic will be scaled and redrawn. The upper left corner of your selected viewport will be moved into the upper left corner of the workarea. You remain in viewport selection mode for further zooming operations. Each zooming operation is pushed onto a zooming stack, so that we can return to any previously selected viewport. The currently active viewport will be shown in the control window as a small yellow rectangle, representing the position and the size of the viewport relative to the size of the whole schematic.

The selection of the corners of a viewport can be aborted by pressing the right mouse button. If you do this during the selection of the first corner, the zooming function is terminated and you return to menu selection mode. If you press the right mouse button during the selection of the opposite corner, the start point is cancelled and you can select a new first point of the viewport.

Zoom Out

The inverse function `Zoom Out` can be used to restore the viewport on the top of the zooming stack. This function has no effect, if the zooming stack is empty. If you want to restore the original size of the schematic, you can directly call the function `Normal` which clears the zooming stack in one step.

Scaling

With the function `+10%`, `-10%`, `+50%` and `-50%` from the `-Views-` submenu you can change the size of the schematic without moving its upper left corner to a new position. The scaling factors are 1.1, 0.9, 1.5 and 0.5 respectively. After calling these functions for several times you can return to the original size of the schematic with the help of the function `Normal`.

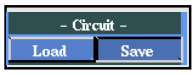
Miscellaneous

With the function `All Objs` from the `-Views-` submenu you get a representation of your schematic which will intuitively show the relation to the underlying mathematical calculus. The graphical represenation resembles an equation, where the left side is the macro representation of the current schematic and the right side is the schematic itself. This denotes that the macro is refined (refinement operator is indicated by an arrow) by the drawing of the schematic. You can return from this representation to the normal mode, if you call the functions `Normal` and `Home`. The call of `Normal` will restore the original size of the schematic and `Home` will move its upper left corner into the upper left corner of the workarea.

The function `Redraw` can be used to refresh the drawing of the schematic within the workarea. This function is useful, if there remains some dirt form entering and deleting objects as macros, wires or comments. Some functions automatically do a redraw operation, so that you do not have to call it very often.

To terminate the hierarchy menu you have to terminate any currently active function first. In most cases this can be done by pressing the right mouse button at most twice within the workarea (see also the description of the appropriate function). After that you are in menu selection mode and you can close the hierarchy menu by pressing the right mouse button within the menuline. Now you return to the main menu of the CADIC system. From there you can select another tool or you can terminate the whole system call by selecting the entry `Exit` in the main menu.

5.10 Save the Hierarchy Levels



In order to save a circuit you should select the entry `Save` from the submenu `-Circuit-`. If the circuit can be saved successfully into the directory given by `DAGDIR` the system will display the following message:

`Schematic <name> saved.`

When a schematic is saved, each hierarchy level is saved into a seperate file in `DAGDIR`. For example, if you save the circuit `CSA[16]`, files are created for `CSA[16]`, `CSA[8]`, `SEL[9]`, `CSA[4]`, `SEL[5]`, `SEL[4]`, `SEL[3]`, `CSA[2]`, `CSA[1]`, `SEL[2]` and `SEL[1]`.

Note: You must have write permissions for the directory `DAGDIR` to save the circuit hierarchy. If this is not the case, the system tries to save the files to the default directory `/tmp`.

6

Logic Simulation

6.1 Introduction

The simulation tool we describe in this chapter allows the user to check the functional behaviour of a design. Only the logical function of the basic cells is taken into account, there are no timing aspects, e.g. delays of the gates and the wires, represented in the simulation model. This simulation tool gives you a basic method to check the correctness of your design.

More detailed information about the timing behaviour of your design will be given by a simulation tool which will be included in the next distribution of the CADIC system. At this moment you can use one of the conversion routines to create a standard exchange format (e.g. EDIF 2 0 0 or GHDL), as they are described in chapter 11 and perform the simulation with a commercial design system.

6.2 Getting Started

If you use this simulation tool you have to consider some preconditions on your design. First of all you may only simulate combinational circuits. The design must not have virtual cycles (c.f. figure 6.1) and the data flow on a bundle of wires must be unique for all single wires of this bundle (c.f. figure 6.2).

If the first condition is violated the system is not able to calculate a topol-

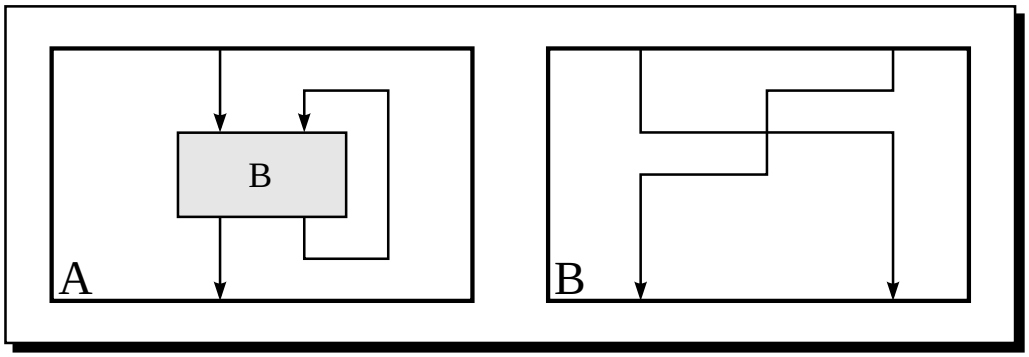


Figure 6.1: Macro A contains virtual cycle

gical sorting of the subcircuits for at least one hierarchy level. Because the simulation is done by a C program this sorting is needed to create the correct order of the function calls for the subcircuits as you will see in the following.

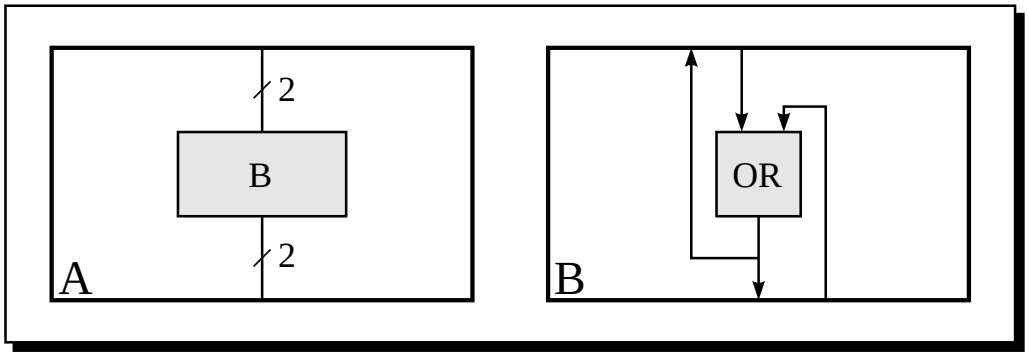


Figure 6.2: Two busses in macro A with non unique data flow

The second condition comes from the objects of the underlying netlist data structure which does not allow single subnets of a bus to have different directions. Both restrictions will be erased within a new simulation tool which is integrated in the new release of CADIC . But nevertheless this simulation provides a comprehensive method for a first checking of a design.

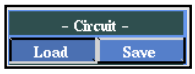
6.3 Sample Session

In this sample simulation session we will first analyze the fulladder design to show the basic functions of the simulation tool. Afterwards we will check the design of the n bit conditional sum adder from chapter 4. The simulation of a design usually takes the following steps

- load and prepare the design for simulation
- start the simulation for a single pattern or a list of patterns
- examine the simulation results by tracing through the circuit hierarchy

In the following we will explain how to perform these steps with the help of our special design example, where not all functions of the simulation tool are needed.

6.3.1 Load and Prepare a Design for Simulation



First we have to load and prepare a circuit for simulation. To do this select the entry `Load` from the submenu `-Circuit-`. After pressing the left mouse button you will see the list of schematics in your `DAGDIR` directory. From this list you should select the entry `FullAdder` by pressing the left mouse button onto the highlighted text.

Again you can read the messages during the loading of `FullAdder` and the construction of the hierarchical data structure. Finally you can read the message

```
Circuit "FullAdder" loaded, Total Data Size:  14698 Bytes
```

Now `FullAdder` is loaded and it will be prepared for simulation. The system will calculate the data flow through the circuit and will generate a C module for the execution of the circuit operations as it is shown in figure 6.3.

This module will be created in the directory given by the environment variable `SIMDIR`. In the message area you can see the following information:

```
cannot open file ../Data/Sim/FullAdder.out
```

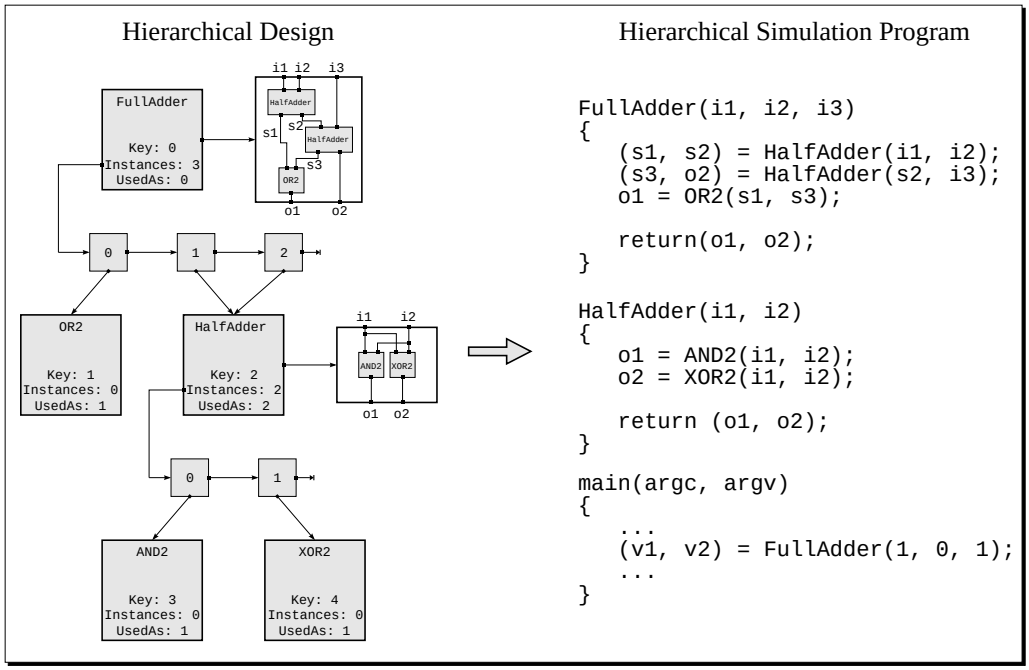


Figure 6.3: Creation of a hierarchical simulation program

This means that the executable simulation program does not yet exist. Here `../Data/Sim` is the value of the environment variable `SIMDIR` and `FullAdder.out` is the name of the executable simulation program. This program is created by translating the C module `../Data/Sim/FullAdder.c` and linking subsequently with an object file for the basic cells which is located in the `CELLDIR` directory and is called `basiccells.o`. For the compilation and the linking of the simulation program CADIC uses the same compiler as during the installation of the system [2]. If you get a copy of the executable system, then GNU `gcc` is the default compiler.

During the creation of the executable simulation program the system will display the message

`Generation of the simulation program:`

In the workarea the circuit is not fully shown, until the process of translating and linking of the simulation program has terminated. If the system displays the message

```
--- Finished
```

the executable file is created and the system is ready for starting the circuit simulation. At this moment your workarea should look like that shown in figure 6.4.

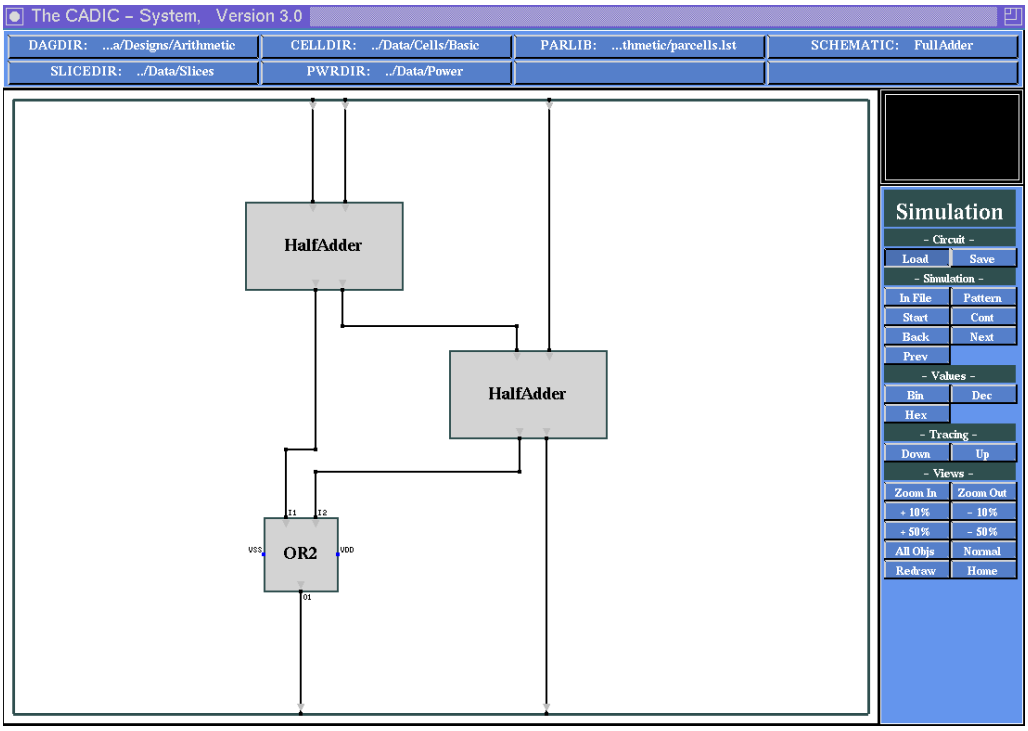


Figure 6.4: Circuit `FullAdder` loaded and prepared for simulation

In figure 6.4 you will see that the system has calculated the data flow through the circuit. This is indicated by the small arrows at the inputs and outputs of the two instances of `HalfAdder`.

6.3.2 Simulation of a Single Pattern

First we want to simulate the circuit for a single input pattern. For this purpose select the entry `Pattern` from the `-Simulation-` submenu. After activating this function the system asks you for the values at the input pads of the circuit. Your screen should look like that shown in figure 6.5. First you have to enter the input value for the most right pad at the northern

border of `FullAdder` which is highlighted by a small square. At the prompt window you can give a binary input value for this pad. Just type in

```
Give Input Value for this Pad <1        >
```

as it is shown in figure 6.5. Note that the binary string must have a length of 1, because of the width of the corresponding input wire.

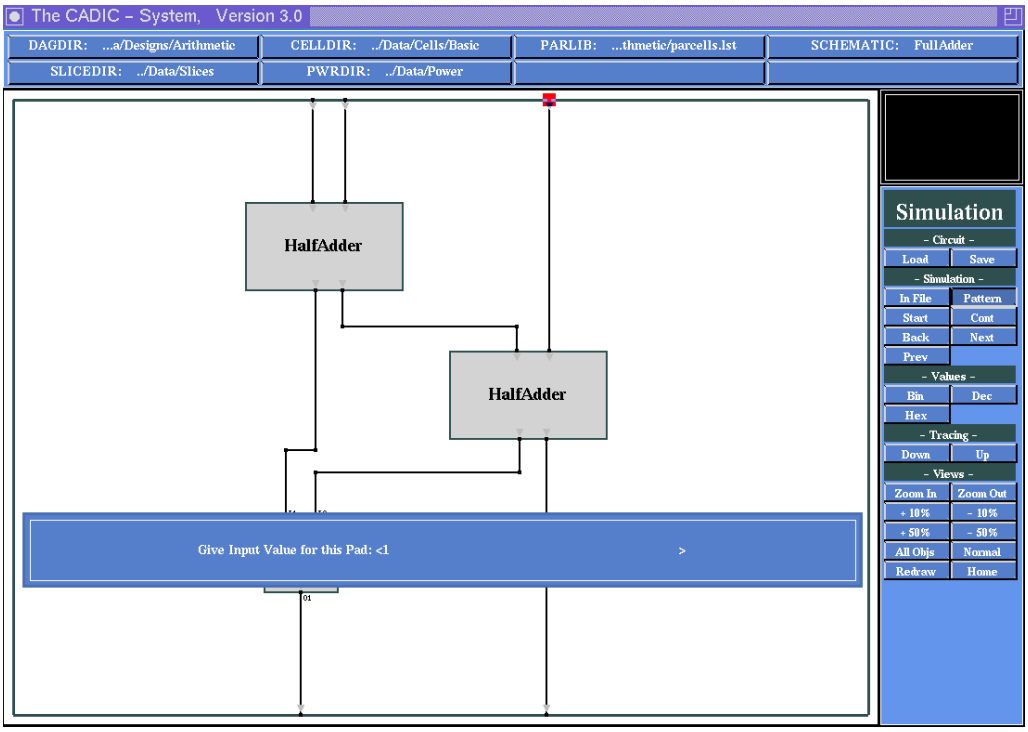


Figure 6.5: Setting input value for the right input pad

After pressing the return key, you will see a label beside the input pad, containing the binary input string. In the same way you have to specify the values for the middle and the left input pad. Here we enter the values `0` and `1` respectively, i.e. we want to check our fulladder design for the operation $1 + 0 + 1$. The pads are chosen according to the graphical input of the schematic, i.e. the wire which is connected to the border as first, is prompted first, etc.

A short time after you have input the last value, you will see the simulation results at the input and output pins of the macro cells. At each input

there exists a label containing a binary string, representing the values of the correspondig signals as it is shown in figure 6.6.

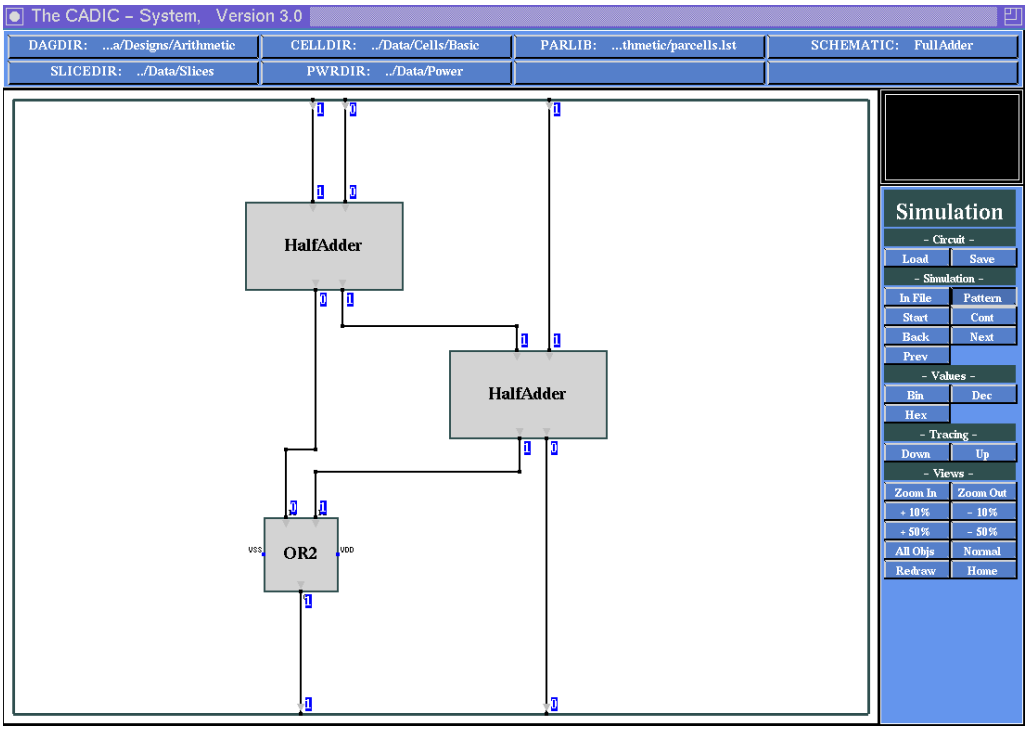


Figure 6.6: Simulation results for a single input pattern $1 + 0 + 1$

At the output signals of the circuit you see the values `1` and `0`, representing the carry value 1 and the sum value 0, i.e. for the input values 1, 0 and 1 the circuit computes the correct output values.

6.3.3 Tracing through Simulation Results

Now we want to examine the simulation results in detail. This can be done with the help of the tracing operations, which have already been explained in chapter 5. First we want to descend in the circuit hierarchy and control the simulation results at the next level. To do this select the entry `Down` from the submenu `-Tracing-`. Now you are in cell selection mode. Move the pointer into the workarea near to the upper instance `HalfAdder`. If it is highlighted in blue, press the left mouse button and descend into this macro cell. Now the schematic for `HalfAdder` is displayed and the simulation

results are shown at the input and output pins of the cells on this hierarchy level (c.f. figure 6.7).

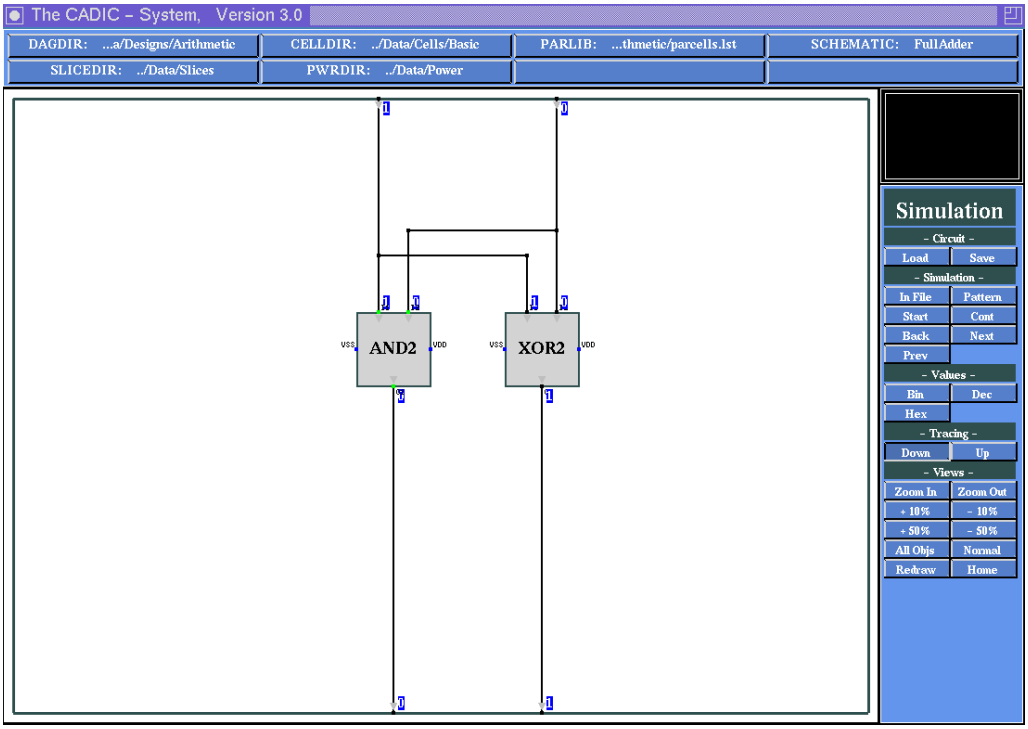


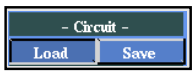
Figure 6.7: Simulation results at the `HalfAdder` level of the `FullAdder` specification

You can control the operation of the basic gates by comparing the input values to the generated output value. The task of `HalfAdder` is to compute the carry and sum of two single bits as we have shown in chapter 4. The first output value is the carry value, the second output is the sum. In this case the carry is `0` and the sum value is `1`, because we add `1 + 0`.

At the lowest level you terminate the trace down operation by pressing the right mouse button within the workarea. Now we will climb up the hierarchy to the topmost level in order to explain some other functions of the simulation tool. To do this move the pointer onto the entry `Up` in the submenu `-Tracing-` and press the left mouse button to reach the highest hierarchy level `FullAdder` (the current level name is shown in the upper right corner of the environment window).

6.4 More Simulation Functions

6.4.1 Load and Prepare for Simulation



In order to explain more functions of the simulation tool we will now examine the 4 bit conditional sum adder as a small example for a paramterized circuit. First we have to load and prepare this circuit for simulation. As shown above you can do this by selection of the entry `Load` from the submenu `-Circuit-`. From the schematic list window you should select the entry `CSADDER[n]`. We choose this schematic because it contains the additional wiring subcircuits which help us to check the simulation results very easily as it will become clear in the following.

Because this is a parameterized design, you will be prompted to enter the values for its parameters. After specifying these values the system will be able to build the hierarchical data structure as it is described in chapter 5.

In our case we choose $n = 4$, i.e. simply type `4` to the prompt window

```
Enter Parameter Values for "CSADDER[n]":  <4    >
```

and press the return key. In the message window you will see the hierarchy levels which are allocated for `CSADDER[4]`.

Now `CSADDER[4]` is loaded and it will be prepared for simulation. The system will calculate the data flow through the circuit and will generate a C module for the execution of the circuit operations (c.f. figure 6.3).

This module will be created in the directory given by the environment variable `SIMDIR`. In the message area you can see the following information:

```
cannot open file ../Data/Sim/CSADDER[4].out
```

This shows that the executable simulation program does not yet exist. During the creation of the executable simulation program the system will display the message

```
Generation of the simulation program:
```

In the workarea the circuit is not fully shown, until the process of translating

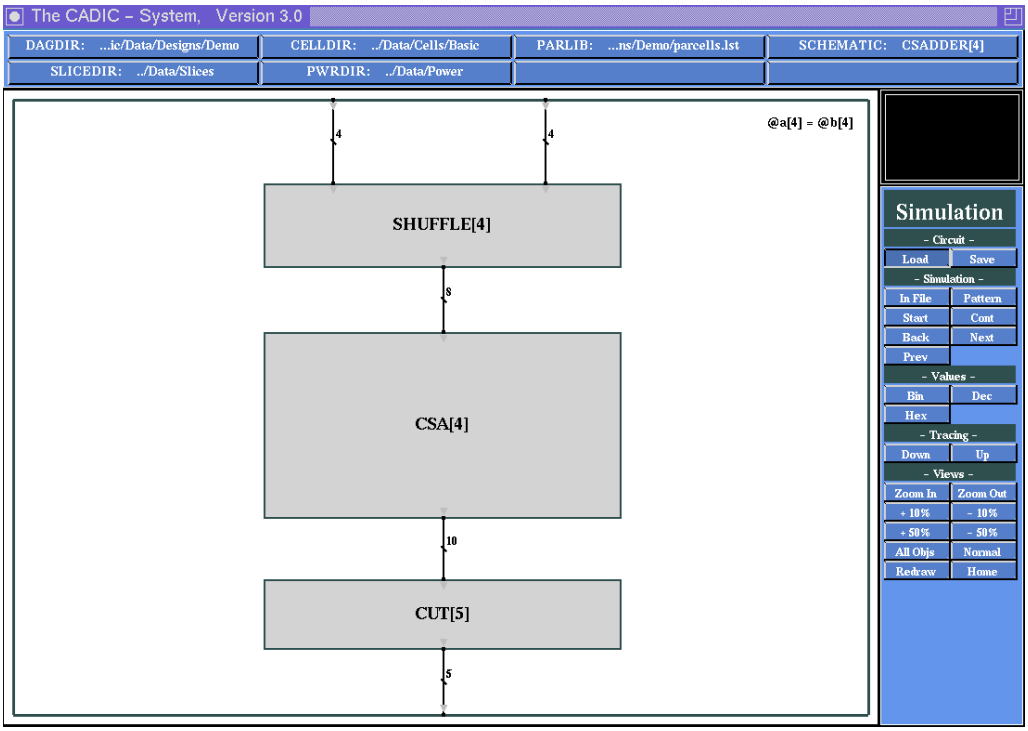


Figure 6.8: Circuit `CSADDER[4]` loaded and prepared for simulation

and linking of the simulation program has terminated. If the system displays the message

```
--- Finished
```

the executable file is created and the system is ready for starting the circuit simulation. At this moment your workarea should look like that shown in figure 6.8.

In figure 6.8 you will see that the system has calculated the data flow through the circuit. This is indicated by the small arrows at the inputs and outputs of the macro cells.

6.4.2 Simulation of a Single Pattern

Again we want to simulate the circuit for a single input pattern first. For this purpose select the entry `Pattern` from the `-Simulation-` submenu. After activating this function the system asks you for the values at the input pads

of the circuit. Your screen should look like that shown in figure 6.9. First you have to enter the input value for the left pad at the northern border of `CSADDER[4]` which is highlighted by a small square. At the prompt window you can give a binary input value for this pad. Just type in

```
Give Input Value for this Pad <11        >
```

as it is shown in figure 6.9. Note that the binary string must have a length of 4, because of the width of the corresponding input wire. For this reason the system will fill up your input with leading zeros.

If you enter more digits than given by the wire width, your input will be refused and the prompt window will popup again.

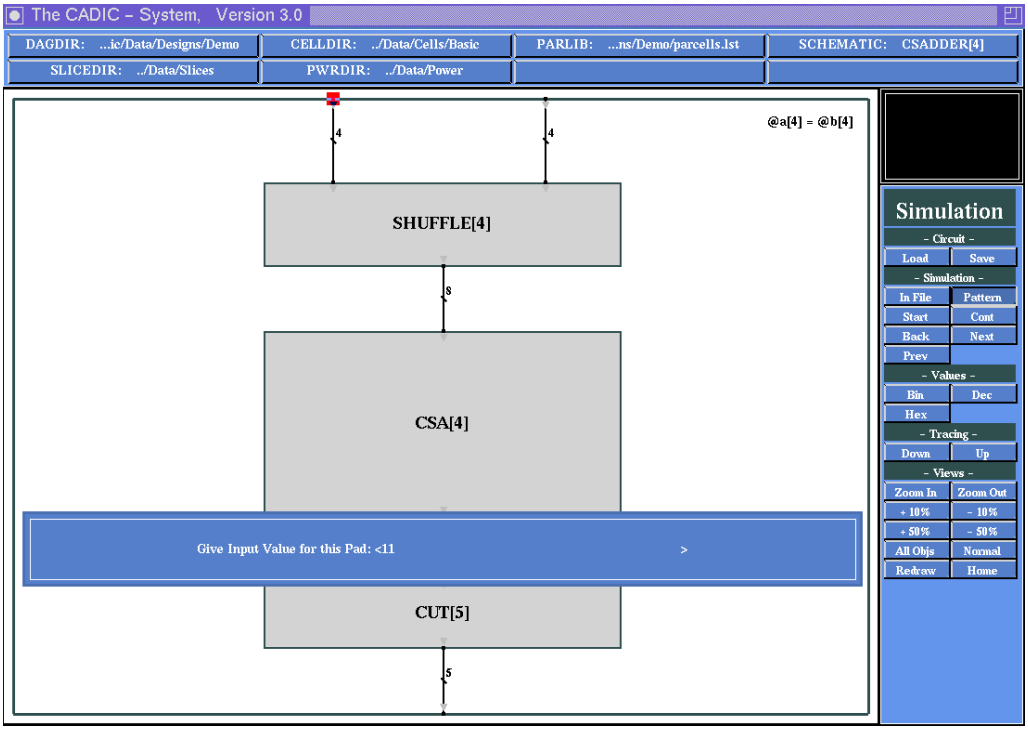


Figure 6.9: Setting input value for the left input pad

After pressing the return key, you will see a label beside the input pad, containing the binary input string. In the same way you have to specify the value for the right input pad. Here we enter the value `10`, i.e. we want to check our conditional sum adder design for the operation $11 + 10$.

A short time after you have input the last value, you will see the simulation results at the input and output pins of the macro cells. At each input there exists a label containing a binary string, representing the values of the corresponding signals as it is shown in figure 6.10.

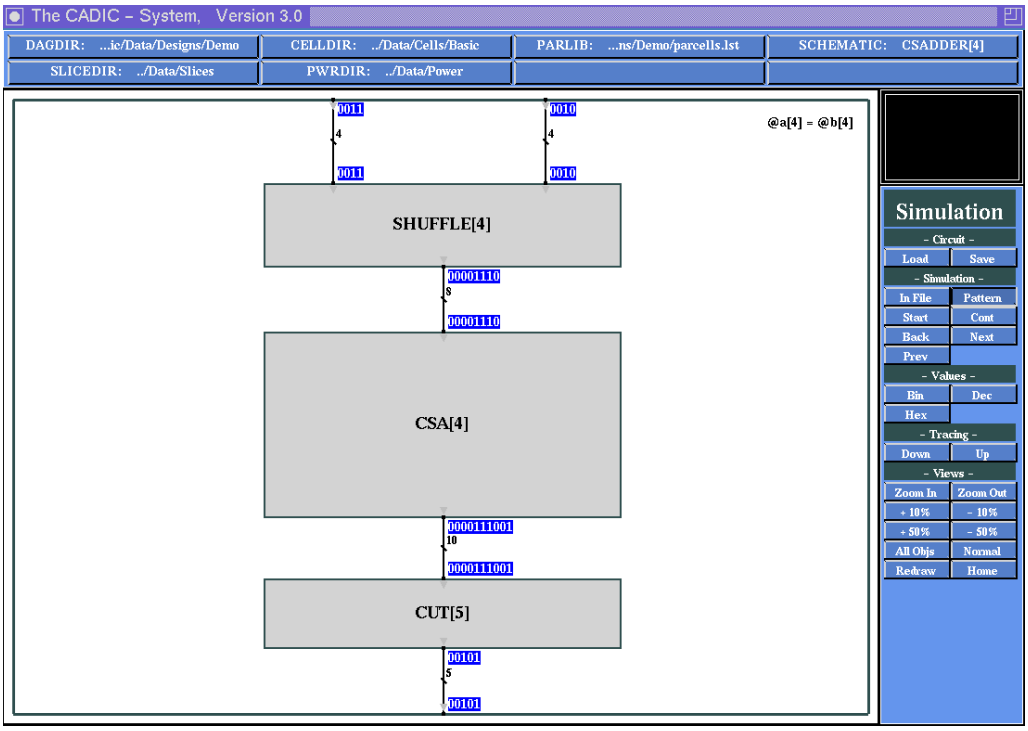
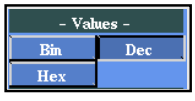


Figure 6.10: Simulation results for a single input pattern $11 + 10$

At the output signal of the circuit you see the value `00101`, representing the decimal value 5, i.e. for the input values `11` and `10` the circuit computes the correct output values. Again you can check the simulation results at the lower hierarchy levels with the help of the tracing operations. If you trace through the hierarchy you should return to the highest level before you continue with the following operations.

6.4.3 Changing the Display Mode



Currently the input and output values are displayed as binary strings, i.e. as sequences of zeros and ones. The length of each string is given by the width of the corresponding wire. In some cases (e.g. arithmetic circuits) it is more

convenient to interprete these binary strings and display them according to a certain base. Here we offer the modes decimal and hexadecimal. Now we change the display mode to decimal by pressing the left mouse button onto the entry `Dec` in the submenu `-Values-`. The bitstrings are converted to decimal numbers and you see the input values (`3`, `2`) and the output value `5`. Note that now the input and output is done in decimal mode .

We control this by entering a new single pattern for the simulation. Move the pointer onto the entry `Pattern` in the submenu `-Simulation-`. After pressing the left mouse button you are asked for the input value of the left input pad (c.f. figure 6.9). Now you can type in a decimal value, e.g. `9`. Note that the binary representation of the decimal value must not exceed the width of the corresponding wire. In our case of `CSADDER[4]`, where the input wire have width 4, you may input numbers in the range from `0` to `15`. for the second input pad enter the value `4` and press the return key.

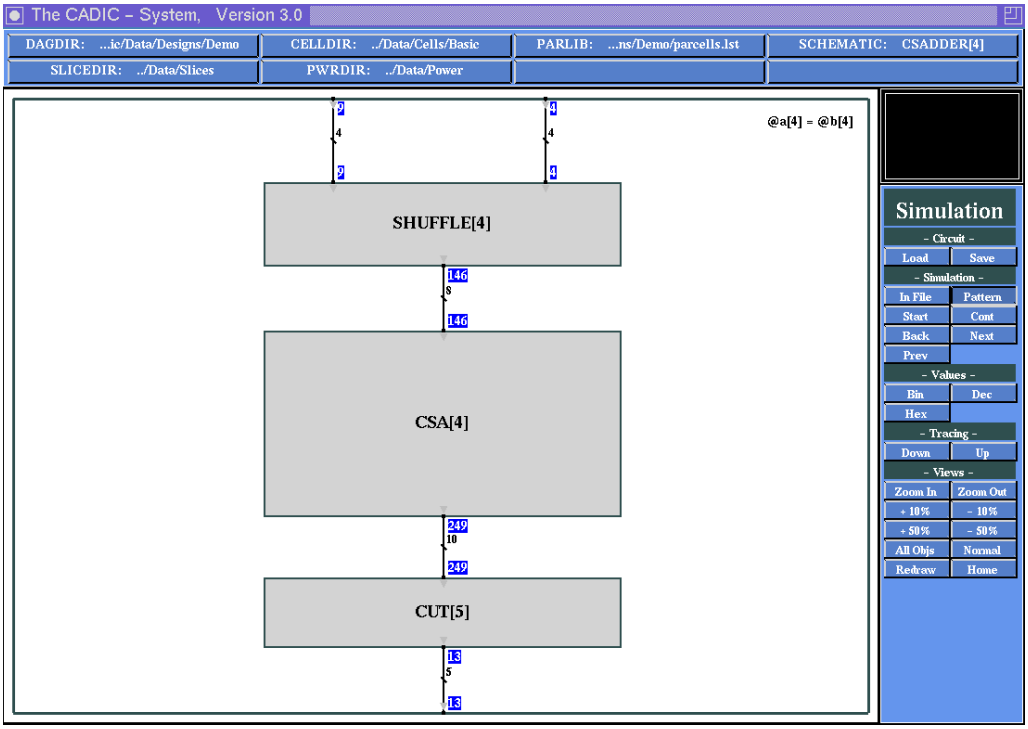


Figure 6.11: Display of simulation results in decimal mode

After the simulation has been finished you see the results at the input and

output pins of the macros which are all displayed in decimal mode (c.f. figure 6.11). The result of the addition is `13` which shows that our adder works correct for this input example.

The decimal display mode is very intuitive for controlling simulation results at a high level of the circuit hierarchy. When you descend in the hierarchy and the wires are more and more split into single binary values, it is more convenient to use the binary display mode.

6.4.4 Simulation of a List of Patterns

The simulation of a single pattern is useful, if you want to examine the behaviour of your design for special input values. But in general, you have to simulate it for a lot of patterns. Then it would be very tiresome to input all patterns in single pattern mode. The simulation of a list of patterns can easily be done with the help of the function `In File` from the `-Simulation-` submenu. After activating this mode, a window will be popped up onto the workarea. This window contains the system call

```
vi ../Data/Sim/CSADDER[4].pat
```

i.e. it calls the `vi` editor with the file `CSADDER[4].pat`, on which you can write simulation patterns for the execution by the simulation tool. This file is also located in the directory given by the environment variable `SIMDIR`. At this point of our session the file does not exist and in the bottom line of the new window you can read the message

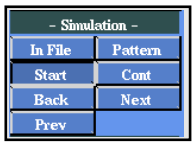
```
"../Data/Sim/CSADDER[4].pat" [New File]
```

which is the normal message of `vi` after opening a new file. In the following we suppose that you are familiar with the work with `vi`.

Now type in the following text:

```
DAG CSADDER[4];
PAR 4;
```

```
INPUT n[0:3], n[4:7];
OUTPUT s[0:4];
PATTERNS
00000000
00000
00000001
00001
00000010
00010
00010000
00001
00100000
00010
00100010
00100
00110011
00101
01000100
01000
01000101
010001
```



After you have closed the `vi` window you can start the simulation of the pattern list. For this purpose select the entry `Start` from the submenu `-Simulation-`. In the message area you can see, how many patterns have been simulated successfully. Each pattern is simulated and compared to the expected output in the pattern list. If there is a difference between the simulation result and the expected output, the simulation stops and the current simulation results are displayed at the input and output pins of the macro cells.

In this case there is a mistake on our pattern file. Therefore the simulation will stop when pattern 7 is simulated. In the message area the following

information is given

```
Wrong Simulation Values for pattern 7

Simulation Output:  00110011 0 0 00110

 Specified Output:  00110011 0 0 00101
```

The second line shows the simulation results for pattern 7 and the third line shows the expected results. These are grouped according to the borders of the circuit, beginning with the northern border, on which the inputs are located, i.e. input string is `00110011`. The next two elements are at the eastern and western border. These have no input or output pads, and their default string is `0`. The last string is located at the southern border which is the output of the circuit. Here is the difference between the simulation result `00110` and the expected result `00101`.

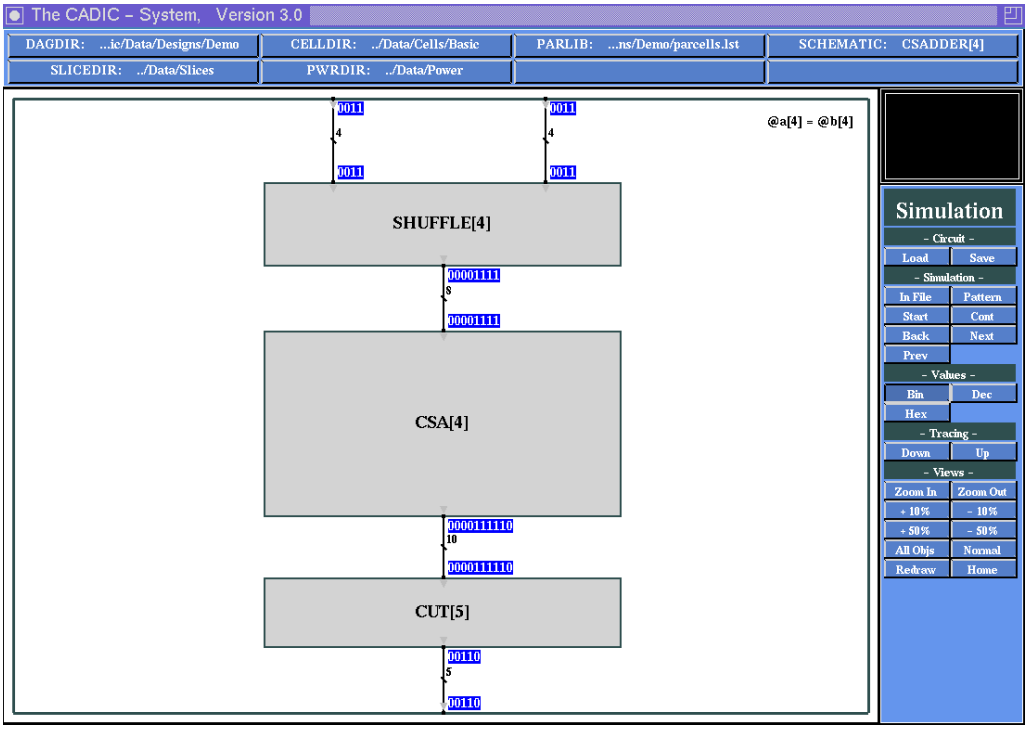
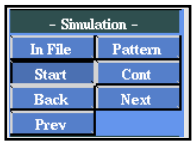


Figure 6.12: Simulation results for pattern 7

If you notice a difference between the simualtion result and the expected result, you can now examine your design with the help of the tracing func-

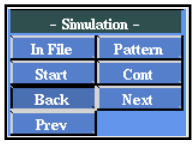
tions and locate the error. In our case it is not a design error, but we have a wrong entry in our pattern list. The inputs to our adder are namely `0011` and `0011`, i.e. the decimal numbers $(3, 3)$. The result of this addition should be 6, i.e. `00110` in binary representation. We have to change the corresponding line on our pattern file which can be done by calling the function `In File` from the submenu `-Simulation-` again.



Change the pattern file in the appropriate way and save it into `SIMDIR`. After that you can select the entry `Start` from the submenu `-Simulation-` again. Finally you will see the messages

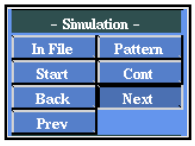
```
Simulation successful
```

in the message area. This means that all patterns have been sucessfully simulated and the simulation loop has reached the end of the pattern file.



If you want to check the patterns step by step you can first reset the simulation by selecting the entry `Back` from the submenu `-Simulation-`. With the help of this function you can return to a previously created wrong simulation result. In our case there is no such input pattern anymore and you return to the beginning of the pattern list. In the message area you can read the information

```
No previous error found.
```



Now you can step through the patterns in your list. For each pattern the simulation results are shown at the input and output pins of the macro cells. With the help of the tracing functions you can examine the design for each simulation pattern. If there is no difference between the simulation result and the expected result of the current pattern, you will see the message

```
Simulation sucessful
```

in the message area. If you repeat this function and reach the end of the pattern list, the system will display the message

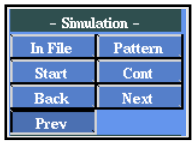
```
Simulation terminated
```

and the labels at the input and output pins will turn empty. Any further

activation of `Next` will result in the messages

```
Simulation terminated
```

as we have shown it above. This tells you that the end of the pattern list is reached. You can now step backwards with the help of the following function or restart the simulation by selecting the entry `Start`.



In the same way as you can step forward through the list of patterns you can go backwards to previous patterns. This can be done by selecting the entry `Prev` form the submenu `-Simulation-`.

7

Layer Assignment

7.1 Introduction

During the specification of a circuit we have neglected the layers, in which the wires are embedded, in order to support a comfortable description method. It is the task of a design tool to assign the signal nets to the given wiring layers (*layer assignment problem*). Wiring segments which are intersecting and which do not belong to the same signal net, must be embedded in two different layers. Parts of a signal net which are on different layers, must be connected by vias. Figure 7.1 shows a subcircuit which is embedded in two layers. In this case we need two vias to get a legal layer assignment.

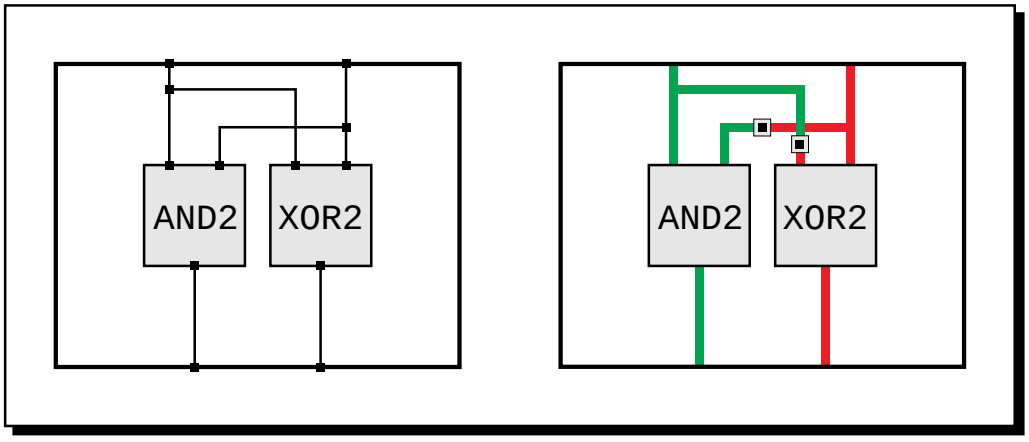


Figure 7.1: Circuit with layer assignment in two layers

An important aspect of algorithms for automatic generation of a layer assignment lies in the minimization of the number of vias. The reason for this is that vias delay the signals and they cause errors during the production process of the chip.

This optimization problem is called *constrained via minimization problem* (CVM), where we use n different layers in the general case (CVM_n). But industrial chip fabrication uses only two layers for the logical wiring of the chip and one additional metal layer for the supply nets. During the assignment of the signal nets to the layers, there may be some constraints which have to be taken into account. For example if you have layers with different conductivity, you should use the layer with higher conductivity for most parts of the signal nets (*layer assignment problem with layer preference*). Another constraint can be given by the basic cells, if you may only connect a pin in a certain layer (*layer assignment problem with pin preassignment*). A summary about the theory of layer assignment problems which in general are very difficult to solve, can be found in [Mol87],[CNR87].

7.2 The Algorithm integrated in CADIC

In the following we restrict to an algorithm which solves the problem CVM_2, where vias may only be set on wiring segments and not on branches or knees. In [KCS88] it is shown that there exists a polynomial algorithm for this restricted problem.

The computation of the layer assignment is done by a hierarchically working algorithm on the DAG structure of the circuit, i.e. each treenode in the hierarchy is treated only once. The algorithm works bottom up, i.e. the layers for a treenode are calculated, if they exist for all its instances. For the basic cells, the layers, in which the pins may be connected, are stored in the basic library. For each hierarchy level we have to solve the layer assignment problem with pin preassignment, because from the lower level the layers at the borders of the instances are implied. An optimal solution for this problem in polynomial time is unknown until now ([KMO89]).

In a simple approach the algorithm could be run without regarding the given preassignment of the pins. Afterwards we could adjust the layers at the instance borders by creating appropriate vias. It is obvious that this method would lead to too many vias. Therefore we hold for each Cgraph two different layer assignments which are "dual" to each other. The second version of the Cgraph is given by exchanging the layer for all signal nets, i.e. a net in the red layer in the normal version is in the green layer in the dual version and vice versa. From these two possibilities we choose this one, where fewest vias are needed to adjust the instance border to the surrounding Cgraph.

The integrated tool in CADIC ([Gra92]) is based on the algorithm described in [Mol93]. This algorithm works on a Cgraph as input, for which it calculates the dual graph first. The faces of the Cgraph which are represented by the nodes of the dual graph, are marked as "odd" or "even". This classification follows directly from a classification for the nodes at the border of each face. The edges of the dual graph represent the neighbourhood of two faces in the Cgraph, i.e. these faces have a common border. In [Mol93] it is shown that there is a legal layer assignment for the Cgraph, if for each pair of odd faces there exists a path within the dual graph. The edges on this path directly imply the locations of the vias. It is also shown in [Mol93] that such a "marriage" of the odd faces is always possible, because there is always an even number of odd faces in a given Cgraph. The problem of minimizing the number of vias for a legal layer assignment is now equivalent to finding shortest paths for the connection of two odd faces.

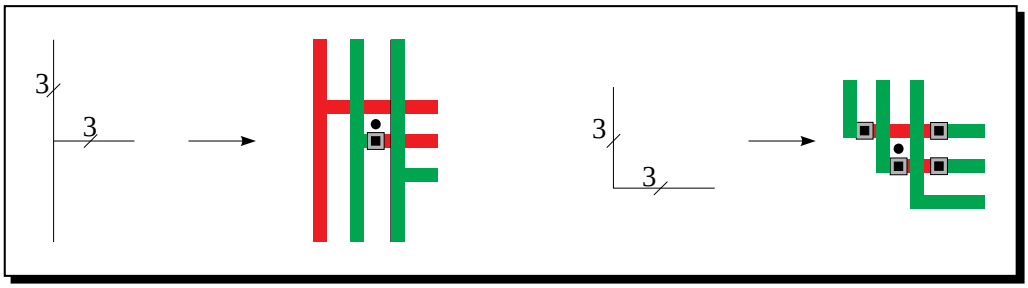


Figure 7.2: Layer assignment for multi wire nodes

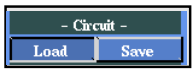
For a legal layer assignment we still have to consider the width of wires, because the branching or crossing of wires with a width larger than one implies more vias as figure 7.2 illustrates. The vias are needed, because the refinement of the multi wire node contains odd faces (marked by •). In order to reduce the number of vias we allow the use of *multi layer busses*, i.e. not all wires have to be in the same layer. In the same way a *bus via* represents a set of vias on the single wires of the bus.

7.3 Layer Assignment for the Halfadder

In order to invoke the layer assignment tool you first should select the entry `Layer Assignment` from the submenu `-Synthesis-` within the main menu.

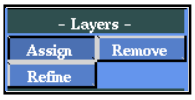
To show you how the system will display the information from the layer assignment algorithm we will first apply it to the simple design of the halfadder.

Load Circuit Hierarchy



First you have to load the DAG structure of the circuit `HalfAdder`. This can be done by selecting the entry `Load` from the submenu `-Circuit-` which calls the same function as the load button from the hierarchy menu (c.f. chapter 5). If you still have loaded the circuit `HalfAdder` you need not select this menu point.

Assign the Layers



To compute the layer assignment for `HalfAdder` you should activate the entry `Assign` from the submenu `-Layers-`. In the message window of CADIC you can read the following messages:

```
building_dualgraph HalfAdder
number of odd_faces 2
odd_face_marriage in dualgraph HalfAdder
inserting_layer_information HalfAdder
number of created via's:  2
```

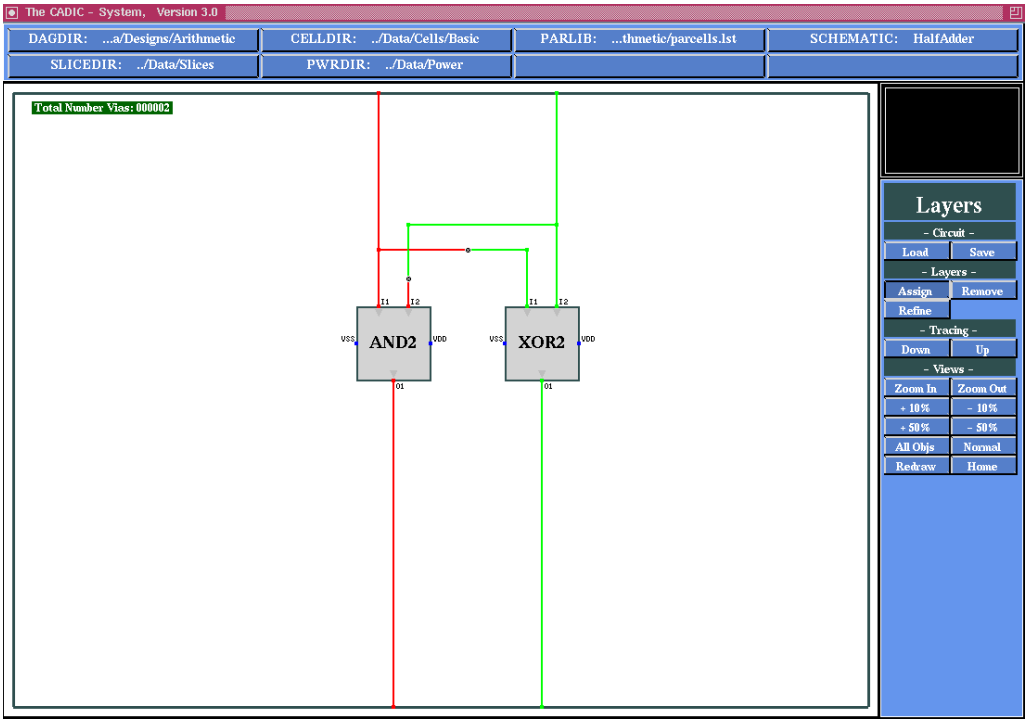


Figure 7.3: Layer assignment for `HalfAdder`

The first message shows that the system builds the dual graph from the Cgraph of `HalfAdder` and that it contains 2 odd faces. In the next step these faces are "married", i.e. the shortest path from one face to the other is calculated in the dual graph. This path implies the creation of two vias, i.e. the path consists of two edges in the dual graph.

After the last message is displayed the graphical representation of `HalfAdder` is changed in the workarea (c.f. figure 7.3). The wires which have been coloured yellow, are now drawn in red and green. These two colours represent the two different wiring layers, whereas yellow denotes the unspecified layer.

In the upper left corner of the workarea you can read a label `Total Number Vias: 000002` which indicates that for the whole circuit there have been created two vias. In this example of one single hierarchy level this number is equivalent to the vias on this level.

The vias are denoted by small squares on the wires in the Cgraph. At

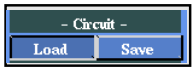
these positions the layer for the signal net changes from red to green. The algorithm puts the vias always in the middle of the corresponding Cgraph edge.

If you look at figure 7.3 you might ask yourself, why are the vias needed. We could layout the wire at the second input `I2` of the `AND2` gate in the green layer and the wire connected to the left input `I1` of the `XOR2` gate in the red layer. The reason for the vias is that the algorithm solves the layer assignment problem with pin preassignment. In our case the pins of the basic cells are preassigned within the basic cell library, such that all pins are either in the red layer or in the green layer.

7.4 Colouring the Fulladder

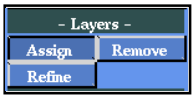
Now we apply the layer assignment algorithm to a small hierarchical circuit description.

Load Circuit Hierarchy



In the same way as above you should load the DAG structure for `FullAdder` by selecting the appropriate entry from the list window.

Assign the Layers



After the DAG structure has been built we can now compute the layer assignment for `FullAdder`. As shown above you can do this by activating the entry `Assign` from the submenu `-Layers-`. In the message window the system will now display the following informations:

```
building_dualgraph HalfAdder
number of odd_faces 2
odd_face_marriage in dualgraph HalfAdder
inserting_layer_information HalfAdder
building_dualgraph FullAdder
number of odd_faces 2
odd_face_marriage in dualgraph FullAdder
```

```
inserting_layer_information FullAdder
number of created via's:  5
```

In this case of a hierarchical circuit the system has to build the dual graphs for all treenodes, i.e. for `HalfAdder` and `FullAdder`. As mentioned above the algorithm works bottom–up, such that `HalfAdder` is treated first and the layer assignment for it is used to compute the layer assignment for the Cgraph of `FullAdder`. Both Cgraphs have two odd faces, which have to be "married". As we know from the previous section the algorithm needs two vias for a legal layer assignment of `HalfAdder`. One additional via is needed within the Cgraph of `FullAdder`, such that we have a total number of 5 vias within the whole hierarchy of the circuit.

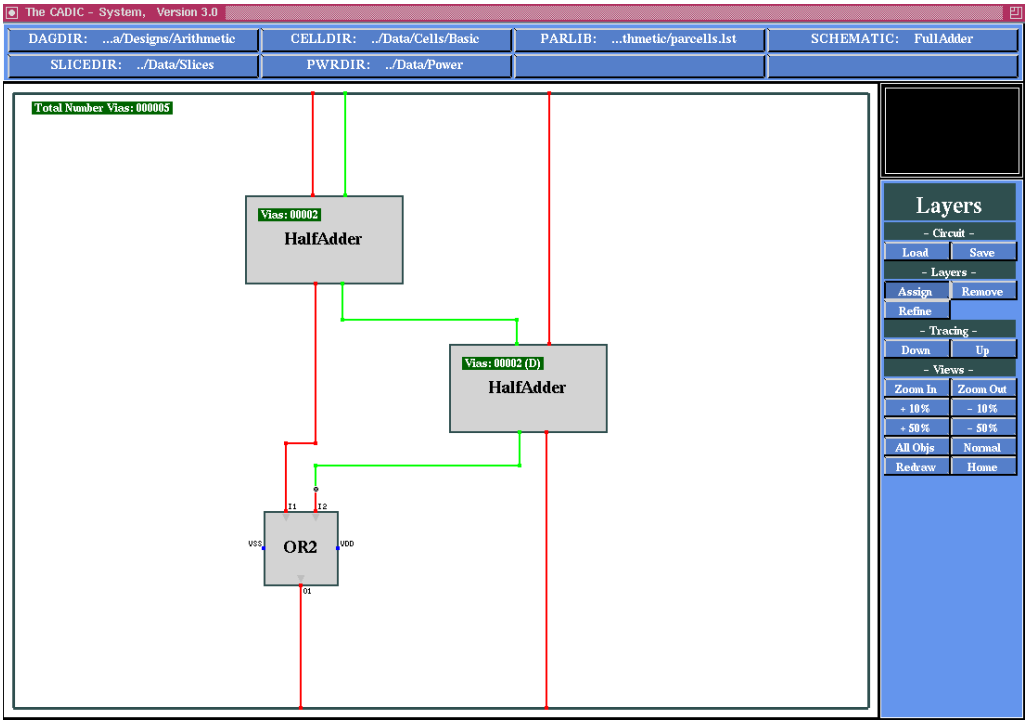


Figure 7.4: Layer assignment for `FullAdder`

In figure 7.4 the root level of the hierarchy for `FullAdder` is shown. If you look at the two instances `HalfAdder`, you can see that for the left instance the sequence of layers at the northern and southern border is (red, green). For the right `HalfAdder` the sequence is (green, red), i.e. for the second

instance we use the dual layer assignment, where the red and green layers are exchanged. This helps the algorithm to minimize the number of vias. If it had to use the same layer assignment for both instances, then it must create a via on the wire connecting the right output of the left `HalfAdder` with the first input of the right `HalfAdder`.

In each instance which represents a subcircuit, you can read the number of vias created in the corresponding DAG structure. Here both instances of `HalfAdder` contain two vias from the layer assignment of the previous section. The annotation `(D)` behind the number of vias in the label of the right `HalfAdder` indicates that for this instance the algorithm has chosen the dual layer assignment.

Tracing through the Results

We can control the layering of the `HalfAdder` with the help of the tracing functions from chapter 5. First select the entry `Down` from the submenu `-Tracing-` in order to descend into one halfadder. Move the pointer into the workarea near to the left instance and select it with the left mouse button. Now the system descends one level in the hierarchy and displays the Cgraph for the `HalfAdder`. Because the instance is in normal mode, you see the same layer assignment as we have created it in the previous section (c.f. figure 7.3).

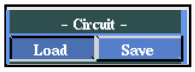
In the next step we want to look at the second `HalfAdder`. To do this, you must terminate the cell selection mode by pressing the right mouse button within the workarea. After that we trace up one level back to the `FullAdder` by selecting the entry `Up` from the submenu `-Tracing-`. Now you can step down into the second `HalfAdder` with the trace down operation. Compared to the first `HalfAdder` you will see that the layer assignment is inverted.

7.5 Multi Layer Wires: the 16 Bit Conditional Sum Adder

For the layer assignment algorithm the example of the 2^n bit comparison tree `Tree[n]` is an "uninteresting" circuit, because there are no crossings of wires in the tree structure. All wires can be layed out in the same layer, such that there are no vias needed.

We will now examine the results for the 16 bit conditional sum adder and especially show how bundles of wires may be assigned to the different layers.

Load Circuit Hierarchy

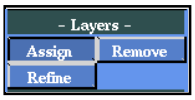


Load the 16 bit adder from our parameterized specification of the n bit conditional sum adder. As you already know from chapter 5 you have to activate the entry `Load` from the submenu `-Circuit-` and select the element `CSA[n]` from the list window. At the following prompt

```
Enter Parameter Values for "CSA[n]":<       >
```

type in `CSA[16]` or simply `16` to set the value of `n`=16. After confirming this entry with the return key the system will load the DAG structure for the 16 bit adder.

Assign the Layers



Now you can call the layer assignment algorithm for `CSA[16]` by activating the entry `Assign` from the submenu `-Layers-`. Following the hierarchical DAG structure, the algorithm displays a list of messages:

```
building_dualgraph CSA[1]
number of odd_faces 4
odd_face_marriage in dualgraph CSA[1]
inserting_layer_information CSA[1]
building_dualgraph SEL[1]
number of odd_faces 6
odd_face_marriage in dualgraph SEL[1]
```

```
inserting_layer_information SEL[1]
...
building_dualgraph CSA[16]
number of odd_faces 0
odd_face_marriage in dualgraph CSA[16]
inserting_layer_information CSA[16]
number of created via's:  402
```

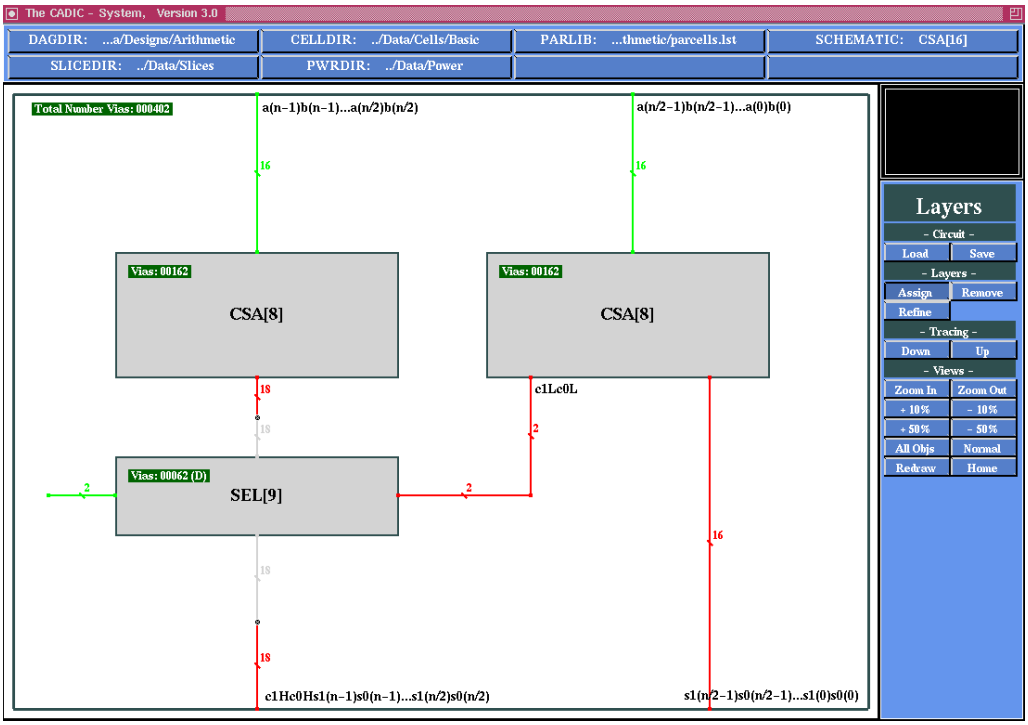


Figure 7.5: Layer assignment for the 16 bit conditional sum adder

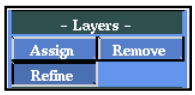
Figure 7.5 shows the result of the layer assignment algorithm for the topmost hierarchy level of `CSA[16]`. The label in the upleft corner of the workarea tells you that there have been inserted 402 via nodes in all Cgraphs of the hierarchy. The informations in the instances show, how these vias are split across the lower levels. Each `CSA[8]` contains 162 vias and the selection subcircuit `SEL[9]` needs 62 vias. If you sum up these numbers, you get $2\times162+62 = 386$. This means that the remaining 16 vias are on the topmost hierachy level. But in figure 7.5 you will only find two via nodes which are placed on the wire connecting the southern border of the left `CSA[8]` and

the northern border of `SEL[9]` and the wire between the southern border of `SEL[9]` and the southern border of the hierarchy level. These two via nodes represent not only one single via, because they are put onto a bundle of wires. As you can see in figure 7.6 each via node corresponds to 8 single vias on this bus wire.

The presence of a multi via node can be recognized by the change of the colours for a bus wire. In the example from figure 7.5 the colour of the bus between `CSA[8]` and `SEL[9]` changes from red to grey. This means that the bus is first completely layed out in the red layer and below the multi via node some single wires of the bus are in the green layer. The bus is then drawn in grey to denote that it is a multi layer wire. In the same way the bus wire at the southern border of `SEL[9]` is a multi layer wire. The remaining bus wires are all layed out in one layer, such as the two wires at the northern border of the `CSA[8]` instances are totally in the green layer.

You might wonder, why the bus via between the southern border of `SEL[9]` and the schematic border is needed. The multi layer wire could be directly connected to the border without changing the layers again. But one property of the underlying algorithm is that a bus wire must be uniformly connected to the schematic border in order to simplify the reuse of each Cgraph as an instance in another hierarchy level. If a multi layer wire has to be connected to the Cgraph border it is unified, such that the minimal number of vias is inserted. In our example the wire is totally layed out in the red layer, because the algorithm needs only 8 vias instead of 10 vias for a uniform wire in the green layer.

Refining Multi Layer Wires



If you want a detailed display of a multi layer wire node, you can select the entry `Refine` from the submenu `-Layers-`. After activating this function you are in wire selection mode. Move the pointer into the workarea near to the bus wire you want to examine. If the desired wire is highlighted (its colour changes to cyan) you can select it by pressing the left mouse button.

After that a small window is popped up upon the bus, in which you see the colours of all single wires. In figure 7.6 we have selected the multi layer wire at the southern border of the instance `SEL[9]` and refined it.

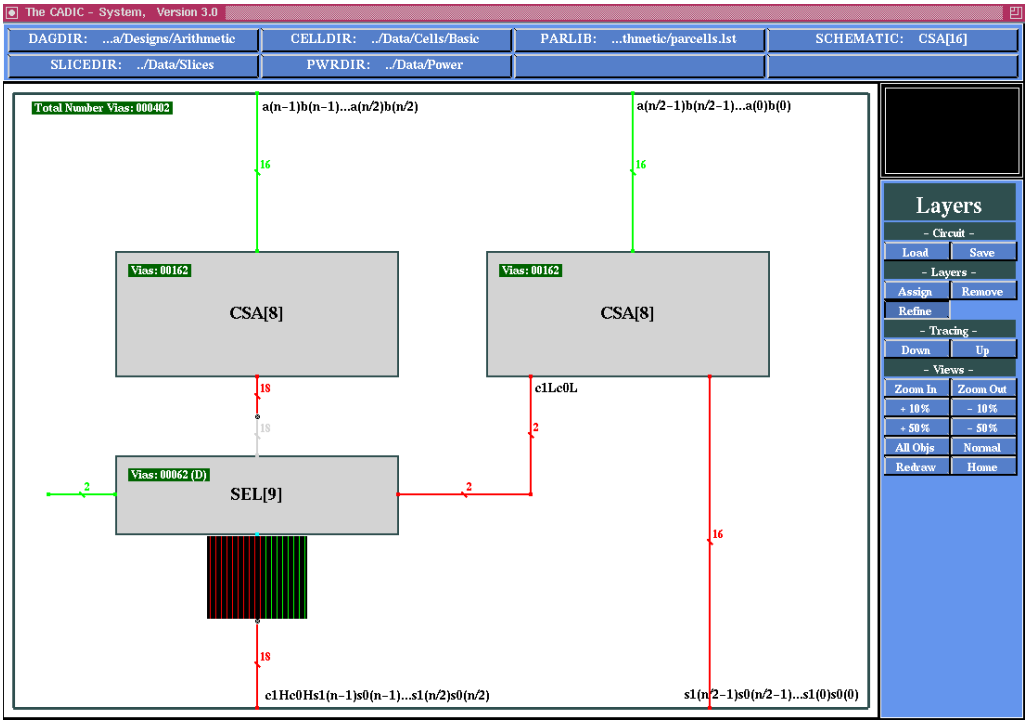


Figure 7.6: Refinement operation for multi layer wires

From the refinement you can also find out, where the vias of a bus via are placed. In figure 7.6 the vias are placed on the last 8 wires of the 18 bit wide bus, because in the bus refinement the last 8 wires are layed out in the green layer and after the bus via the whole wire is in the red layer.

After the popup window is opened in the workarea you remain in wire selection mode in order to refine another bus. For example move the pointer near to the other grey wire between `CSA[8]` and `SEL[9]`. If it is highlighted in cyan, press the left mouse button to select this one. Then the previous popup window disappears and a new window is creted at the position of the new selection. You can terminate the wire selection mode by pressing the right mouse button within the workarea. The current popup window is closed and you return to the menu selection mode.

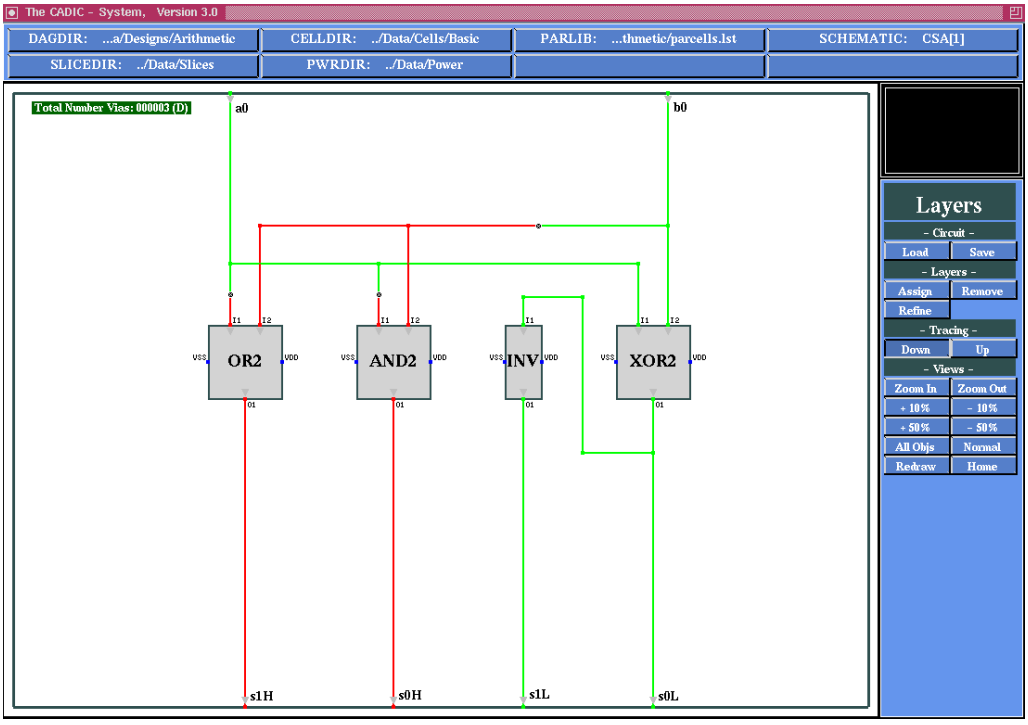


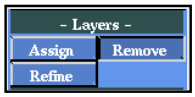
Figure 7.7: Navigation to the lowest hierarchy level `CSA[1]`

Navigation through the Results

Before we close the layer assignment session we want to take a look at the lower hierarchy levels of `CSA[16]`. Select the entry `Down` from the submenu `-Tracing-` in order to descend into the instances. Move the pointer near to the left `CSA[8]` and trace down this path until you arrive at the lowest hierarchy level `CSA[1]`. The layer assignment for the corresponding Cgraph is shown in figure 7.7.

The label in the upleft corner of `CSA[1]` tells you that there are three vias inserted and that the currently selected instance is in dual mode `(D)`. This is respected in the colouring of the wires, where the green and red layers are exchanged. You can notice this by first tracing up one level and then stepping down into the second `CSA[1]` which is in normal mode.

Removing the Layer Assignment



If you want to apply other design tools, where you do not use the layer assignment, you can remove the via nodes and the layer information from the Cgraphs. You also can load the circuit again from `DAGDIR`. To remove the layer assignment select the entry `Remove` from the submenu `-Layers-`. After activating this function the system will display the following informations in the message window:

```
2 Vias (Nodes/Edges) removed in Treenode CSA[16]
2 Vias (Nodes/Edges) removed in Treenode CSA[8]
3 Vias (Nodes/Edges) removed in Treenode CSA[4]
3 Vias (Nodes/Edges) removed in Treenode CSA[2]
3 Vias (Nodes/Edges) removed in Treenode CSA[1]
0 Vias (Nodes/Edges) removed in Treenode OR2
0 Vias (Nodes/Edges) removed in Treenode AND2
0 Vias (Nodes/Edges) removed in Treenode INV
0 Vias (Nodes/Edges) removed in Treenode XOR2
2 Vias (Nodes/Edges) removed in Treenode SEL[2]
3 Vias (Nodes/Edges) removed in Treenode SEL[1]
0 Vias (Nodes/Edges) removed in Treenode MUX
3 Vias (Nodes/Edges) removed in Treenode SEL[3]
3 Vias (Nodes/Edges) removed in Treenode SEL[5]
2 Vias (Nodes/Edges) removed in Treenode SEL[9]
4 Vias (Nodes/Edges) removed in Treenode SEL[4]
Total Number of removed Vias:  30
```

The number of removed vias concerns the via nodes, i.e. a bus via is regarded as one single node in this listing. For example at the hierarchy level `CSA[16]` there are only 2 vias removed which are exactly the two bus vias from above. Each of these two nodes represents 8 single vias as we have shown in the previous paragraph.

Note that the layer assignment is removed from the currently displayed hierarchy level down to the leaves of the DAG structure. If you want to

remove it for the whole DAG structure, then you have to trace up first, until you reach the root of the hierarchy.

Changing the View

With the help of the functions from the submenu `-Views-` you can change the graphical representation of the schematic. The functions in the menu of the layer assignment tool are identically to those in the hierarchy menu. For an explanation you can look at section 5.9.

To terminate the layer assignment menu you have to terminate any currently active function first. In most cases this can be done by pressing the right mouse button at most twice within the workarea (see also the description of the appropriate function). After that you are in menu selection mode and you can close the layer assignment menu by pressing the right mouse button within the menuline. Now you return to the main menu of the CADIC system. From there you can select another tool or you can terminate the whole system call by selecting the entry `Exit` in the main menu.

Normally the layer assignment and the power supply tools are called before you create the final layout of the circuit. When you have created the layer assignment for a circuit you can change to the menu for generating its layout by going to the main menu of CADIC and then to the layout submenu (c.f. chapter 9).

8

Power Supply

8.1 Introduction

During the specification of a circuit the designer does not enter the power supply nets. These can be generated automatically by an appropriate tool ([Sch92]) which has to perform the following two tasks:

- calculation of the topology and
- sizing of the wiring segments

The calculation of the topology is only possible, if all basic cells in the design have pins for the connection of the power supply nets. In our designs from the editor session, where we used the CADIC basic cell library, these pins are denoted by small blue squares at the cell borders, they are labelled with the names `VDD` and `VSS`.

In today's fabrication process both supply nets will be layed out in the same metal layer. This implies that the nets must be free of any crossings. The algorithm in CADIC achieves such a layout by a tree like structure for both nets which can be derived directly from the topology of the graphical specification.

Beside the calculation of the topology of the supply nets, the wiring segments of these nets have to be sized according to power consumption aspects. The sizing has to guarantee that the power drop from the pads at the border of the whole chip to the inner pins at the basic cells is smaller that a

given bound. At the same time the wire segments should not be oversized because of the limited chip area. Above this there is a minimum wire width which has to be respected. This means, that we have to regard different optimization goals which are influencing each other. If we choose a certain layout for the wires this implies their width and the power consumption. A certain wire width has a backdraw to the wire layout because of geometrical rules.

8.2 The Algorithm integrated in CADIC

The tool which is integrated in CADIC , generates the power supply nets in two independent steps. First it calculates the topology of the nets according to the graphical specification. In a second step the sizing for the nets is calculated such, that the power consumption and the chip area for the wires are balanced within the given topology.

8.2.1 Calculation of the Topology

As almost all integrated algorithms in CADIC the calculation of the topology for the power supply nets is done in a hierarchical manner. The algorithm constructs the nets in a bottom–up process following the DAG structure of the given circuit. This means, that for each treenode the power supply nets are created once. The location of the pads for these nets on the corresponding schematic borders are used on the next higher hierarchy level as the new starting points. For the macro cells on each hierarchy level the algorithm creates new pins for the connection of the power supply nets.

The underlying algorithm ([Kol86]) is based on the representation of each treenode by a bicategorial expression as we have introduced it in section 4.2. Before the topology is created, for each treenode a data structure for the bicategorial expression is generated by a slicing algorithm. This algorithm uses the graphical specification to split it according to the operations $\ominus$ and $\oplus$. This algorithm is stored in an appropriate function library, which can be used for other tools using the bicategorial representation. Another

example for such a tool is the place&route algorithm which will be explained in chapter 9.

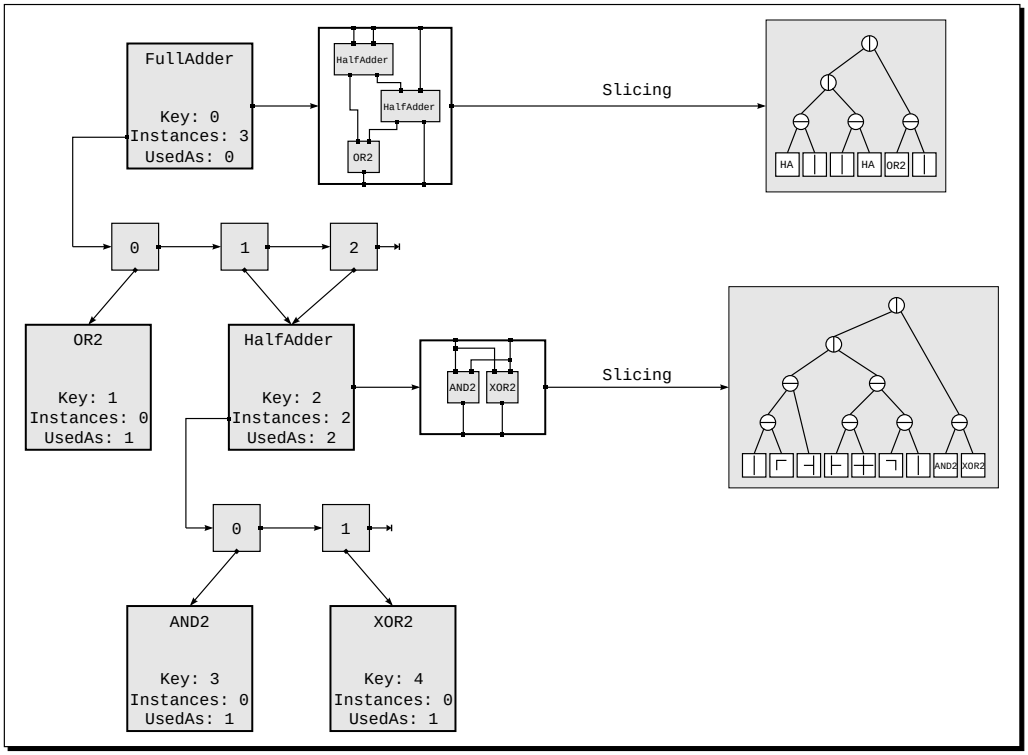


Figure 8.1: Generation of a slicing tree for each treenode in a hierarchical description

In figure 8.1 it is shown, how the representation in form of a bicategorial expression is created for each treenode in the hierarchy. By applying the slicing algorithm for each treenode a pointer to the data structure for the bicategorial expression is set. Each expression is represented by a tree like structure, where the leaves of the tree are built of nodes, edges and instances in the corresponding Cgraph. The inner nodes of the data structure, which is called the *syntax tree*, are operational nodes representing the concatenations of the corresponding subtrees.

On each hierarchy level the integrated algorithm works bottom–up in the syntax tree, where the operations at the inner nodes imply the local topology of the supply nets ([Kol86]). To create an easy to handle result, we assume that each treenode and each instance has exactly one `VDD` and one `VSS`

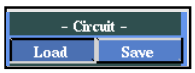
pad. The VDD (VSS) pins and pads are always located at the northern or the eastern (the southern and the western) border. This implies a simple method to create the two power nets without crossings.

8.2.2 Sizing of the Power Supply Nets

After the topology of the power supply nets is generated each wire segment of these nets is sized according to the power consumption of the connected elements. The sizing algorithm is also working hierarchically on the underlying DAG structure. It is based on heuristics which are explained in detail in [Kol86]. The calculated sizing results are stored as attributes to the created wiring edges in the corresponding Cgraphs, such that they can easily be displayed in the graphical specification.

8.3 Power Nets for the Fulladder

Load Circuit



Before we can construct the power supply nets for a circuit, you must load this circuit first. After pressing the push button `Load` in the submenu `-Circuit-`, a list window pops up onto the work area which contains the list of all existing circuits. Move the cursor near to the entry `FullAdder` and select by pressing the left mouse button.

Create the Topology of the Power Nets

After you have loaded the hierarchy for `FullAdder` you can create the topology for the power supply nets. You activate the algorithm by selecting the entry `Create` from the submenu `-Power-Tree-`. Then you can read the following informations in the message window:

```
Generate the Power-Tree for HalfAdder
Generating/Reading Slicing(HalfAdder)
Routing power_wire(HalfAdder)
Building Power-treenode(HalfAdder)
```

```
Generate the Power-Tree for FullAdder
Generating/Reading Slicing(FullAdder)
Routing power_wire(FullAdder)
Building Power-treenode(FullAdder)
```

These messages illustrate the bottom–up working method of the algorithm. First the power supply nets for the subcircuit `HalfAdder` are generated. For this purpose the algorithm creates or simply reads the slicing of `HalfAdder` as the second message line implies. If the graphical representation of `HalfAdder` has not changed since the last call of the slicing procedure, it can read the old information instead of calculating it from the scratch.

With the help of the slicing information the power supply nets are routed through the Cgraph of the corresponding hierarchy level (illustrated by the message `Routing power_wire(HalfAdder)`). In the final step for this hierarchy level the power nets are connected to the border of the schematic and the pads for the external connections to these nets are created.

Now that the power supply nets for `HalfAdder` exist, the nets for the next higher level `FullAdder` can be generated. Again the slicing for this level is created or read form the directory given by the environment variable `SLICEDIR` and then the power nets are routed through the Cgraph of `FullAdder`. The resulting topology of the supply nets can be seen in figure 8.2.

Note that the instances for `HalfAdder` have two new pins which are labelled `VDD` and `VSS`. These pins correspond to the pads created at the lower hierarchy level, where the supply nets are connected to the border of the schematic. As mentioned above the bottom–up working method of the algorithm implies that for all subcircuits at the current hierarchy level, the power supply nets must have been created, such that the power pins at all instances exist.

For the basic cells at the lowest hierarchy level the supply pins are given by the information from the cell library. An important condition is that the pins for `VDD` are always at the northern or the eastern border and the

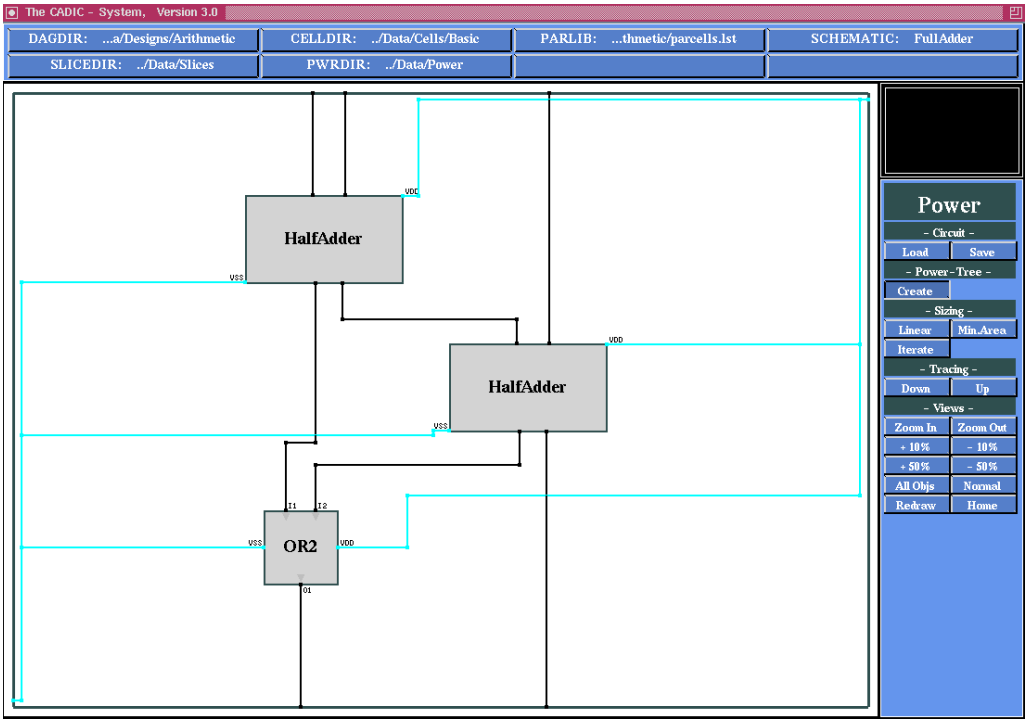
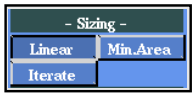


Figure 8.2: Topology of the power supply nets for the `FullAdder` level

pins for `VSS` are at the southern or the western border. For the macro cells this is automatically performed by the power supply algorithm, but for the basic cells this must be regarded during the design of a new cell library.

Sizing of the Supply Nets



After the topology for the supply nets has been created, the wire segments of these nets can be sized according to the power consumption rules. For this purpose CADIC offers three different heuristics which work on the hierarchical representation of the circuit. In our session we will use the linear sizing heuristic which can be activated by selecting the entry `Linear` from the submenu `-Sizing-`.

The results of the sizing heuristic are displayed by small labels at each wire segment of the power supply nets (c.f. figure 8.3). The width of each segment is given in μm, where the minimum width of a wire in CADIC 's layout system is 2μm.

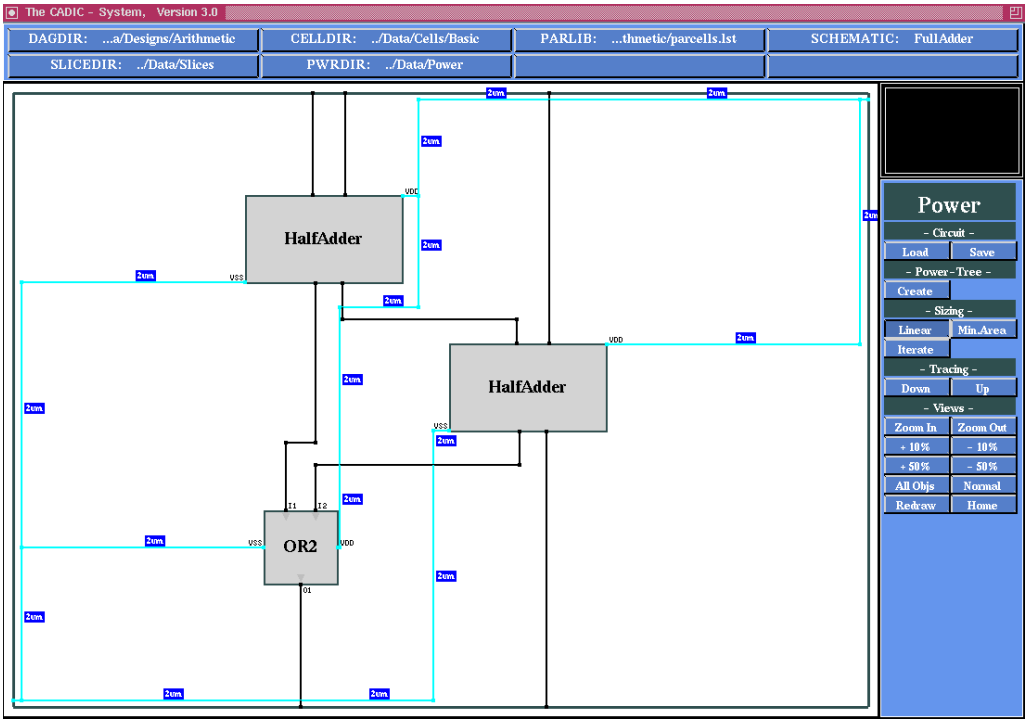
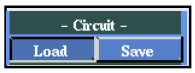


Figure 8.3: Sizing of the wire segments of the power supply nets for the `FullAdder`

In our small example of the fulladder, the sizing heuristic sets all wire widths to the same value of 2μm. Again you can inspect the results of the algorithm at the lower hierarchy level with the help of the tracing functions.

8.4 Power Supply for the Conditional Sum Adder

Load Circuit



Before we can create the power supply nets for the conditional sum adder, we have to load an element of the parameterized description. Select the entry `Load` from the submenu `-Circuit-` and choose `CSA[n]` from the schematic list window. For the parameter value n type in `16` at the corresponding input window.

Create the Topology of the Supply Nets

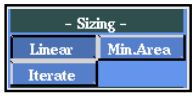
If you activate the entry `Create` in the `-Power-Tree-` submenu the topology for the supply nets on each hierarchy level of `CSA[16]` is created. In the message window the algorithm displays the informations for each treenode:

```
Generate the Power-Tree for CSA[1]
Generating/Reading Slicing(CSA[1])
Routing power_wire(CSA[1])
Building Power-treenode(CSA[1])

Generate the Power-Tree for SEL[1]
Generating/Reading Slicing(SEL[1])
Routing power_wire(SEL[1])
Building Power-treenode(SEL[1])
...
Generate the Power-Tree for SEL[9]
Generating/Reading Slicing(SEL[9])
Routing power_wire(SEL[9])
Building Power-treenode(SEL[9])

Generate the Power-Tree for CSA[16]
Generating/Reading Slicing(CSA[16])
Routing power_wire(CSA[16])
Building Power-treenode(CSA[16])
```

Sizing for the Power Nets



Now we call the sizing heuristic for the power supply nets of the 16 bit conditional sum adder. Again we will use the linear sizing algorithm which can be activated by the entry `Linear` form the submenu `-Sizing-`. After the algorithm has finished you can see the results within the small labels at each segment of the supply nets as it is illustrated in figure 8.4.

In this example the sizing algorithm has to enlarge the wire segments near to the border of the whole circuit. For example the VDD net has the width 22μm at the eastern border of the schematic. The label at the corresponding

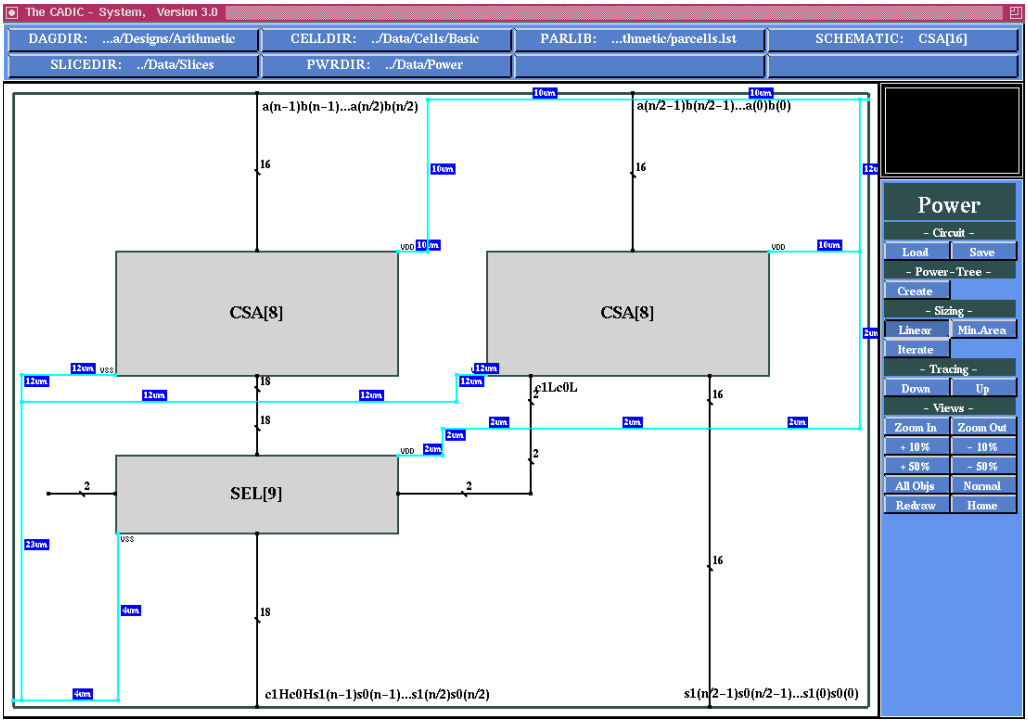


Figure 8.4: Power supply nets for `CSA[16]` with linear sizing of the wire segments

wire segment is not shown, because the length of the segment is too small. Nevertheless you can look at it with the help of the zooming function `Zoom In` from the submenu `-Views-`. After the branching node this wire segment is split into two segments of the width 12μm and 10μm, respectively.

The `VSS` net which starts at the western border of the schematic has the size 27μm. Again you must zoom in to see the label at this short segment. After the branch, it is divided into two segments of width 23μm and 4μm, the first of which is split again into two segments of width 12μm and then connected to the two instances `CSA[8]`.

With the help of the tracing function `Down` from the submenu `-Tracing-` you can follow this wire into the instances `CSA[8]`. Select the left 8 bit conditional sum adder and trace down into it. At this level you can control the connection of the power supply nets to the next subcircuits in the hierarchical description.

Now trace down until you reach the lowest level. For this purpose we select the left instance, the conditional sum adder for the upper half of the operands, at each hierarchy level until we reach the level `CSA[1]` as it is shown in figure 8.5.

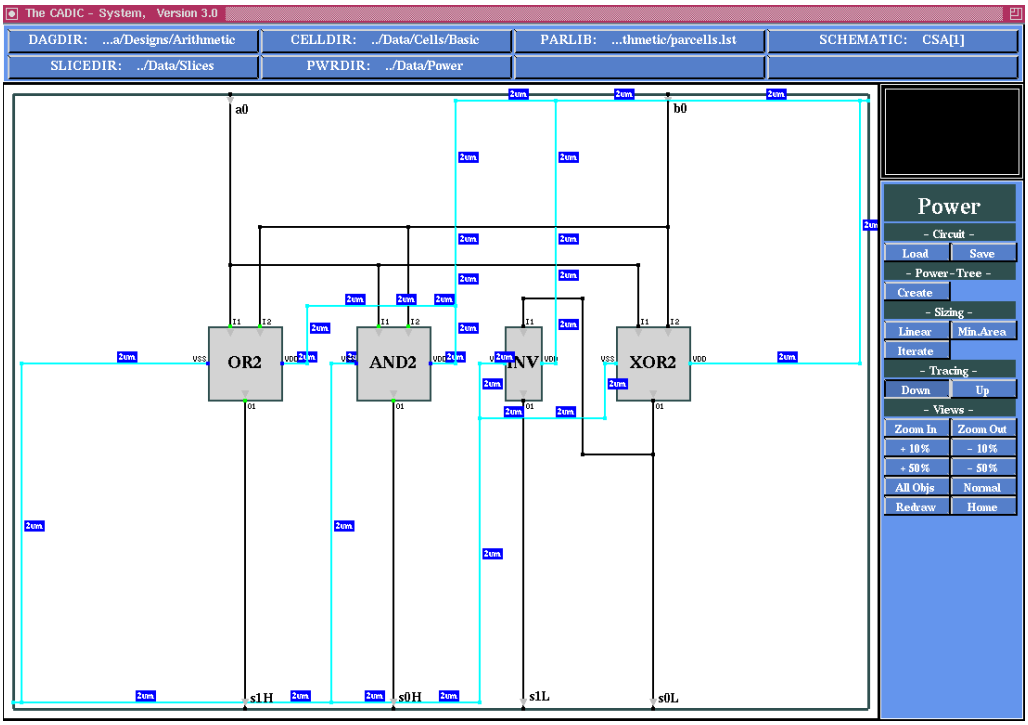


Figure 8.5: Power supply nets for the 16 bit conditional sum adder at the lowest hierarchy level `CSA[1]`

At this level which only contains basic cells, all wire segments are sized to the minimum wire width of 2μm. The width of the wires at higher levels are implied by these basic circuits, because of the bottom–up working method of the sizing algorithm.

Changing the View

With the help of the functions from the submenu `-Views-` you can change the graphical representation of the schematic. The functions in the menu of the power supply tool are identically to those in the hierarchy menu. For an explanation you can look at section 5.9.

To terminate the power supply menu you have to terminate any currently active function first. In most cases this can be done by pressing the right mouse button at most twice within the workarea (see also the description of the appropriate function). After that you are in menu selection mode and you can close the power supply menu by pressing the right mouse button within the menuline. Now you return to the main menu of the CADIC system. From there you can select another tool or you can terminate the whole system call by selecting the entry `Exit` in the main menu.

Normally the layer assignment and the power supply tools are called before you create the final layout of the circuit. When you have created the power supply nets for a circuit you can change to the menu for generating its layout by going to the main menu of CADIC and then to the layout submenu (c.f. chapter 9).

9

Geometrical Layout Design

9.1 Introduction

At the moment, the circuit layout in CADIC is only a topological design which is at the gate level and thereby independent of concrete technologies. In the fact, this layout design handles the problem of finding a "good" topographical representative of a logic topological net. The construction of the layout bases on the circuit representation by bicategorial expressions as it is shown in chapter 8. Here we also use the function library for the slicing of the Cgraphs in order to build the data structure shown in figure 8.1. If this structure has been calculated earlier and the Cgraphs have been changed by another synthesis tool as for example the power supply or the layer assignment tool, the slicing is newly created. The the generation of the layout is performed in the following two steps:

- placement ([Fet95]) and
- routing ([Wan95])

These two phases are decoupled, because during the placement the circuit is deformed step by step. This implies a manipulation of the geometrical data which would be an expensive operation, if it is performed on the final layout data. Instead the placement phase is a calculation phase, after which the positions of all objects in the circuit are known, such that the final layout can be generated by processing of these informations.

Not alike other design system for VLSI, CADIC 's layout system works hierarchically. The layout design in CADIC is also based on its DAG data structure. On the data structure, where the bicategorial expression for each treenode has been generated (c.f. figure 8.1), the system will now process each syntax tree in a bottom–up manner. The elements at the leaves of a syntax tree are basic components at the corresponding hierarchy level. The inner nodes of each tree which contain one of the operations $\ominus$ or $\oplus$ imply a local construction step for the two subtrees of this node. Mainly this construction step consists of a river routing algorithm which has to connect the borders of the two objects. The resulting wiring can be optimized according to the following aspects:

- minimization of the channel height
- minimization of the channel width
- minimization of the area for the new object which is created by the two subobjects and the channel between them

These optimization goals are respected during the placement phase, the result of which is a *placed syntax tree*. In this placed syntax tree all channel heights and sizes for the subcircuits are calculated.

Beside the representation as syntax trees, the layout design system in CADIC uses a data structure for the geometrical layout. This data structure which is called *BTG–Net* (basic topographical net), is created for each treenode in the hierarchy (c.f. figure 9.1). Again the hierarchical structure of the circuit is used, i.e. for each treenode there exists only one BTG–Net. For each instance of this treenode the corresponding BTG–Net is inserted.

9.2 The Layout for the Fulladder

In order to create the layout for a circuit select the entry `Place&Route` from the submenu `-Synthesis-` in the main menu. After you press the push button `Place&Route`, a new submenu named `Layout` is displayed in the menuline. You are now in the mode of function selection.

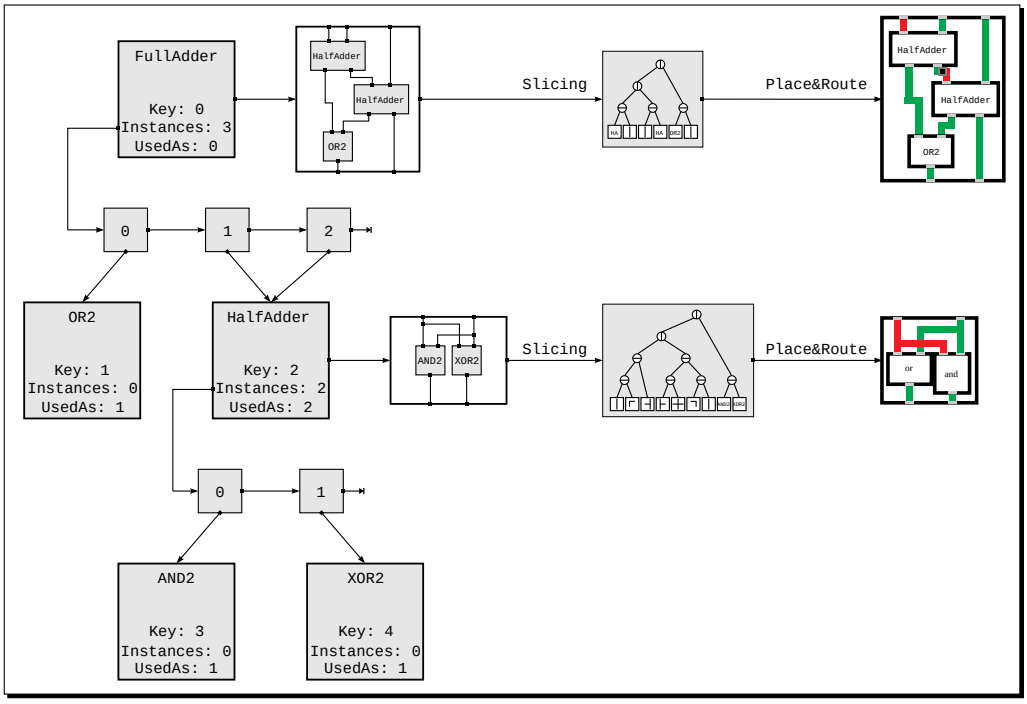
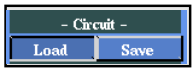


Figure 9.1: Generation of BTG–Nets for each treenode in a hierarchical description

Load Circuit



Before you can create the layout of a circuit you have to load it first. But you can use the DAG structure if you already loaded the circuit within another tool of the system, as for example the layer assignment tool. Especially you must not load it newly, if you want to use the results of a previous tool. In our case we will load again the hierarchy for `FullAdder` and create the layout without the results of the other tools. Select the push button `Load` from submenu `-Circuit-` and choose the entry `FullAdder` in the list window.

Hierarchical Placement

The function `hierarchic` from the submenu `-Placement-` will create a hierarchical geometrical layout. After you have loaded the circuit `FullAdder`, press the push button `hierarchic` within the submenu `-Placement-`. After

that the layout procedure starts up and you can read the following informations in the message window:

```
EMBED < FullAdder >
Slicing:
Generate Slicing for HalfAdder
Generate Slicing for FullAdder
Place the Slicing-Tree for FullAdder
(FullAdder(HalfAdder))
Layout-Generation:
Generate the Layout for HalfAdder
Generate the Layout for FullAdder
Data Space for BTG of FullAdder:  7940 Bytes
Data Space for BTG of HalfAdder:  5120 Bytes
Data Space for BTG of AND2:  1276 Bytes
Data Space for BTG of XOR2:  1276 Bytes
Data Space for BTG of OR2:  1276 Bytes
Total Data Space for BTG: 16888 Bytes
+----------------------------AREA-PROTOCOL----------------------------+
Lay 0:  1605.36 um^2 Lay 1:  0.00 um^2 Lay 2:  0.00 um^2 PLay:  0.00 um^2
+---------------------------------------------------------------------+
Total used area:  33605.36 um^2 Circuit Area:  58864.83 um^2 TA/CA:57.09%
+---------------------------------------------------------------------+
```

These messages illustrate the working of the place&route algorithm. First the slicing of the Cgraphs is generated in the same way as we have explained it in chapter 8. Then the slicing information is used for the placement phase, indicated by the message `Place the Slicing-Tree for FullAdder`. The placement is done in a bottom–up manner. This is indicated by the following line, where the expression `(FullAdder` indicates that the placement for `FullAdder` is started. But before it can be finished the placement for `HalfAdder` has be generated. An open bracket "`(`" indicates a step down in the hierarchy and the corresponding closing bracket "`)`" represents the backward direction. In our example the placement for `FullAdder` can be

finished after that for `HalfAdder` is finished.

After the placement is done, the system can generate the final layout of the circuit. Again this is done in a bottom–up manner. First the layout for `HalfAdder` is created, then that for `FullAdder`.

Finally the system displays information about the space used for the total layout of the circuit. Here you get the sizes for all treenodes in the hierarchy, i.e. also the sizes of the basic cells are shown, for which the layout is given within the corresponding basic cell library.

The tabular with the title `AREA-PROTOCOL` shows the size information of the circuit. In detail you can see, how much area is used for the wiring in the different layers. In our example, where we did not call the layer assignment and power supply tool, all wires are layed out in the default layer 0, such that you can read the information `Layer 0:  1605.36 um^2`. This indicates that all wires in the circuit use an area of $1605.36\mu m^2$. In the second line of the tabular the system displays the total used area of the layout, i.e. the area for the wires plus the area for the basic cells. In our case, this is $33605.36\mu m^2$. This value is put in relation to the enclosing rectangle of the circuit, such that we have a coverage of `57.09%`.

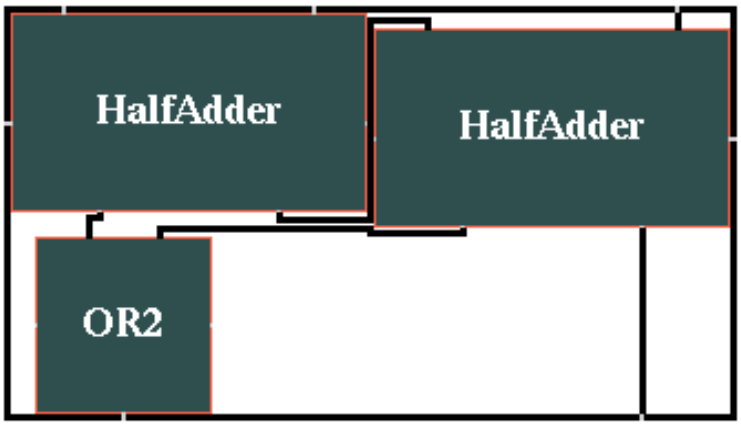


Figure 9.2: Hierarchical layout for `FullAdder`

Figure 9.2 shows the hierarchical layout for the `FullAdder`, i.e. you see the highest hierarchy level with the macro cells for the two `HalfAdder`.

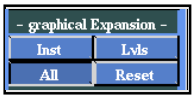
The layout differs from the schematic input of the circuit, because the used area is smaller, if the instances for the `HalfAdder` are placed beside each other. But note that the topology of the layout is the same as that for the schematic input. This means that the system does not insert new crossings of wires or totally changes the arrangement of the cells.

Graphical Expansion

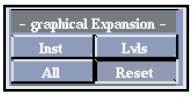
Because the display of the layout at the highest hierarchy level does not give a good impression of the structure of the circuit, you can expand the graphical representation. For the expansion of the layout the system offers you three different operations:

- expansion of a single instance
- expansion of one instance for some levels
- full expansion of the whole circuit

We will explain the first two operations later in this chapter. In the case of our small `FullAdder` example they nearly do the same as the full expansion operation.



For a complete expansion of the layout you should select the entry `All` from the submenu `-graphical Expansion-`. After the activation of this push button, the layout in the workarea is drawn again, where the two `HalfAdder` instances are replaced by their contents as you can see it in figure 9.3.

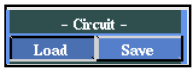


With the help of the entry `Reset` from the submenu `-graphical Expansion-` you can restore the display of the highest hierarchy level. After this first impression of the layout subsystem in CADIC we will turn to a larger example in order to explain the remaining menu functions.

Figure 9.3: Complete expansion of the layout for `FullAdder`

9.3 Layout for the 16 Bit Conditional Sum Adder

Load Circuit



Before you can create the layout of a circuit you have to load it first. But you can use the DAG structure if you already loaded the circuit within another tool of the system, as for example the layer assignment tool. Especially you must not load it newly, if you want to use the results of a previous tool. In order to load the 16 bit conditional sum adder you should select the entry `CSA[n]` from the schematic list window and type in `16` at the prompt window for the parameter values.

Hierarchical Placement

The function `hierarchic` from the submenu `-Placement-` will create a hierarchical geometrical layout. After you have created the DAG structure for `CSA[16]`, press the button `hierarchic` within the submenu `-Placement-`. After that the layout procedure starts up and you can read the following informations in the message window:

```
EMBED < CSA[16] >
Slicing:
Generate Slicing for CSA[1]
Generate Slicing for SEL[1]
```

```
Generate Slicing for SEL[2]
...
Generate Slicing for CSA[16]
Place the Slicing-Tree for CSA[16]
(CSA[16](CSA[8](CSA[4](CSA[2](CSA[1])(SEL[2](SEL[1])))(SEL[3]))
(SEL[5]))(SEL[9](SEL[4])))
Layout-Generation:
Generate the Layout for CSA[1]
Generate the Layout for SEL[1]
Generate the Layout for SEL[2]
...
Generate the Layout for CSA[16]
Data Space for BTG of CSA[16]:  66084 Bytes
Data Space for BTG of CSA[8]:  36336 Bytes
...
Total Data Space for BTG: 291192 Bytes
+----------------------------AREA-PROTOCOL----------------------------+
Lay 0:  0.00um^2 Lay 1:  112918.28um^2 Lay 2:  47372.92um^2 PLay:  0um^2
+---------------------------------------------------------------------+
Total area:  1425056.83um^2 Circuit Area:  5395939.12um^2 TA/CA:26.4%
+---------------------------------------------------------------------+
```

During layout design, many informations about the design process are displayed in the message window which include the information of slicing cells, placing the slicing-tree, layout generations for each subcircuits, and data space sizes of BTG. After the layout is successfully completed, the system will display an AREA-PROTOCOL table in the message window which gives the information about design areas. In the table, the areas for `Layer 0, Layer 1, Layer 2` and `Power-Layer` represent the areas of wires, and `Total used area` is the sum of areas of all cells and areas of all wires. `Circuit area` indicates the area used by the circuit after placement which is the enclosing rectangle region. The term `TA/CA` gives the rate of the total used area and the circuit area.

In our example we have called the layer assignment tool before the generation of the layout. Therefore in the table, the area of `Layer 0` is $0.00\mu m^2$. `Layer 1, Layer 2` are $112918.28\mu m^2$ and $47372.92\mu m^2$, repectively. The `Power-Layer` is not used. If you do not call the layer assignment tool, all the wires are embedded in the default layer 0.

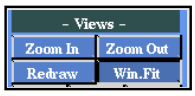
The total used area is $1425056.83\mu m^2$, and the circuit area is $5395939.12\mu m^2$. The rate between the total used area and the circuit area `TA/CA` is `26.4%`.

The complete area protocol is written in the directory defined by the environment variable `PROTDIR`. The name of the area protocol is the same as the name of the circuit and has a suffix `.areaprot`. In order to write the area protocol file, you should first set a value for the environment variable `PROTDIR`. A suggested value is `../Data/Prot`. If you do not set up the environment variable `PROTDIR`, the message window will display:

```
Environment Variable PROTDIR undefined
```

Now the hierarchical placement result of `CSA[16]` is shown in the work area. Only a part of it is visible, because the layout is too large to fit in the window in normal display mode.

Change the Display Mode



In order to see the whole layout of `CSA[16]` you can scale down the display. With the help of the entry `Win.Fit.` from the submenu `-Views-` the layout is scaled in both dimensions such that it totally fits in the workarea. The width and height are both scaled by the maximum factor.

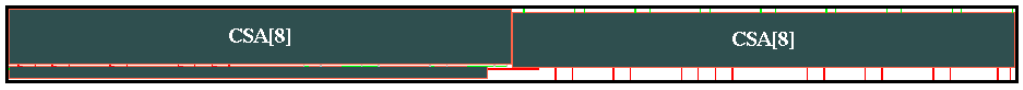
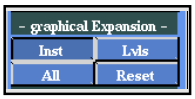


Figure 9.4: Fit the display of the layout `CSA[16]` in the workarea

9.3.1 Layout Expansion

After activating the window fit operation the layout of `CSA[16]` is displayed as it can be seen in figure 9.4. Now we will successively expand the graphical representation of this layout.

Expansion of a Single Instance



With the help of the entry `Inst` from the submenu `-graphical Expansion-` you can resolve the hierarchy level of one single instance. After you have activated this menu point you are in the instance selection mode. If you move the pointer into the workarea, the nearest instance in the layout will be highlighted, i.e. its border changes from red to cyan. Now press the left mouse button and the instance will be replaced by its contained subcircuits.

In figure 9.5 we have selected the right instance `CSA[8]` and substituted by the following hierarchy level. This operation illustrates the recursive description of the conditional sum adder, because `CSA[8]` is built by the same components as `CSA[16]`, but with smaller parameter values and therefore smaller layout areas.

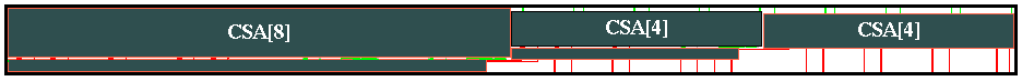


Figure 9.5: Expanding the right instance `CSA[8]` one single step

After the expansion you remain in the instance selection mode in order to expand more instances. Note that now the subcircuits from the just expanded instances are also selectable, because they are at the same level now. You can abort the selection mode by pressing the right mouse button within the workarea.

Expansion over more Levels

While you could perform a detailed expansion of the layout with the help of the previously explained function `Inst`, this operation is tiresome, if you want to expand the layout down to the lower levels. For this purpose CADIC

offers you the entry `Lvls` from the submenu `-graphical Expansion-`.

After the activation of this menu item you are in instance selection mode again in order to choose the candidate for the expansion operation. Pick the highlighted instance by pressing the left mouse button. Now an input window will popup on the workarea asking you the number of levels, for which the instance should be expanded:

```
Please enter the Depth of Expansion:  <3    >
```

In figure 9.6 we have expanded the left instance `CSA[8]` for three hierarchy levels, i.e. the instances you see at the top of the layout are 1 bit adders `CSA[1]`.

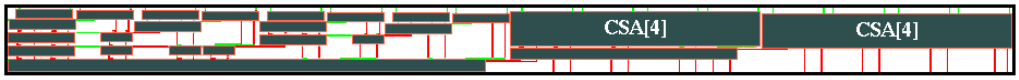
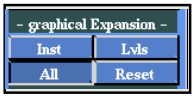


Figure 9.6: Expanding the left instance `CSA[8]` for three hierarchy levels

After the instance has been expanded you are still in instance selection mode which can be aborted by pressing the right mouse button in the workarea. You can use the other expansion operation on an already expanded layout, i.e. you can first expand one instance for some levels, then pick single instances and expand them step by step, etc.

With the help of the function `Lvls` you can expand one single instance down to the basic cell level. This is done, if you enter a depth for the expansion which is larger than the lowest level number. An easier way is to enter `-1`, especially if you do not know, how much hierarchy levels are contained in the corresponding instance.

Full Expansion of the Layout



As shown for the example of the `FullAdder` we can completely expand the layout with the help of the entry `All` from the submenu `-graphical Expansion-`. If you choose this menu item, the layout is fully expanded down to the level of basic cells as you can see it in figure 9.7.

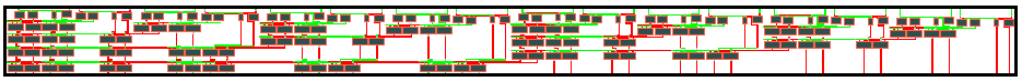


Figure 9.7: Complete expansion of the layout for `CSA[16]`

With the help of the entry `Reset` from the submenu `-graphical Expansion-` you can restore the display of the highest hierarchy level, i.e. the system will display the layout in the form of figure **??**, because you previously activated the window fit display mode.

9.3.2 Layout Tracing

You can hierarchically trace your layout design by using the function `Tracing`. As in the case of the Cgraph structure there are two directions to trace the layout: `Down` and `Up`. We still take the circuit `CSA[16]` as an example to explain this function. For these tracing functions yon can refer to the Section 5.6 for references.

Tracing Down

Because you are now in the highest level of the layout, you can use the tracing down function only. If you select the entry `Down` in the submenu `-Tracing-`, you are in instance selection mode. Move the cursor further to the subcircuit that you want to trace into, as for example the right instance `CSA[8]`, and make its border highlight in blue. Now press the left mouse button and the hierarchical layout of `CSA[8]` appears in the work area replacing that of `CSA[16]` as shown in figure 9.8.

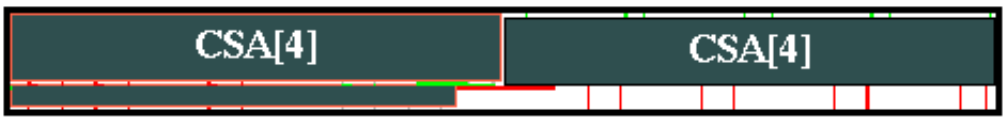


Figure 9.8: Tracing down to the layout of `CSA[8]`

Note: You can not use the expansion functions from the submenu `-graphical Expansion-` during the tracing of the layout. For example,

if you use the function `Inst` within the above `CSA[8]`, you will see the following warning the message window:

```
WARNING: no expansion while tracing!
```

If you want to exit the tracing down function, move the pointer to the work area and press the left mouse button. Then, you return to the menu selection mode again. If you are in the lowest level of the hierarchical layout, you can also press the left mouse button to quit the function of tracing down and return to the menu selection mode again.

At each level of the hierarchy you can enlarge the display of the layout by selecting the entry `Win.Fit` from the submenu `-Views-`. Then the current hierarchy level is scaled, such that it fits in the workarea. This is very helpful, if you trace down to the lower hierarchy levels, where the areas of the components are very small.

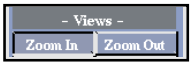
Tracing Up

If you are not at the highest level of the layout hierarchy, you can use the tracing up function to go back to higher levels. Each time you select the entry `Up` from the submenu `-Tracing-` you return to the next higher level of your tracing path. Thus, you can repeatedly press the push button `Up` until you are at the desired level.

9.3.3 Layout Views

The view functions consist of the zoom functions, the redraw function and the window fit function which we have already mentioned above. You can use the functions `Zoom In` and `Zoom Out` to enlarge or shrink the current display of the layout. The functions `Redraw` and `Win.Fit` can be used to adjust the display size to fit in the work area.

Zoom In



In order to enlarge the current display in the workarea, you can select the entry `Zoom In` in the submenu `-Views-`. Move the pointer into the workarea

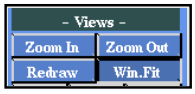
and you can locate the start position of the zoom window. After you have fixed the start point by pressing the left mouse button you can drag a rectangle frame in order to setup the area of the layout which will be scaled to fill the whole workarea. Locate the opposite corner of the rectangle by pressing the left mouse button again. Then, the chosen part is enlarged and appears in the work area to replace the former display. After that you automatically return to the menu selection mode again.

Zoom Out

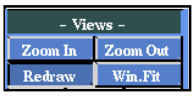
By using this function you can shrink the current display that has been enlarged. Press the push button `Zoom Out` in the submenu `-Views-`, then you return to the previous display mode, i.e. all selected areas during the zooming operations are pushed onto a stack. Each zoom out operation pops the upmost frame from the stack and restores the corresponding display of the layout.

Note: Only after you have enlarged a display, you can use the function `Zoom Out` to shrink it again. If there are no frames on the stack, the function `Zoom Out` will have no effects.

Redraw and Window Fit

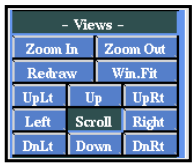


In general the display of a layout has not always a just suitable size for the workarea. For example, the initial display of the layout for `CSA[16]` shows only a part of the whole layout. In order to display the layout fully within the workarea you can select the entry `Win.Fit` in the submenu `-Views-` as we have already shown it in the previous sections. Figure `??` shows the result after executing the function `Win.Fit` for `CSA[16]`.



If you want to return to the initial display, you can use this function. By pressing the push button `Redraw` in the submenu `-Views-`, the initial display will appear in the workarea to replace the current display.

9.3.4 Layout Scrolling



You can scroll the display of the layout result in the work area by using the scroll functions. The scroll functions consist of eight directions: use the function `UpLt` to scroll directly to the upper left corner, `Up` to scroll up one screen, `UpRt` to the upper right corner, `Left` to the left, `Right` to the right, `DnLt` to the lower left corner, `Down` to scroll down one screen and `DnRt` to the lower right corner. If you want to use one of these functions, press the corresponding push button in the submenu `-View-`. You can repeatedly use the same function by pressing the corresponding push button many times until you obtain the desired position.

9.3.5 Output of the Layout Result

By using the output function you can obtain a postscript file of your layout design. Suppose that you want to output the layout result of `CSA[16]`. Move the cursor to the push button `PostScript` in the submenu `-Output-`, and press the left mouse button. Now, a input window pops up onto the work area with the following prompt:

```
Please enter Scaling Factor:  <1    >
```

The scaling factor must be a positive number. If you do not input any number, but direct press the `Return` key, the input window will disappear from the work area and abort the output function. Here we choose the scaling factor `1` first and press the `Return` key. Then, a prompt appears in the input window as follows:

```
Circuit needs 02×10 pages.  Accept (y/n)?:  <n>
```

It means that the circuit will be output in the size of twenty A4 pages. If you accept this size, type `y` and press the `Return` key. If you do not accept this size, type `n` and press the `Return` key. Now, the input window shows the prompt again:

```
Please enter Scaling Factor:  <0.25   >
```

Now we select the scaling factor `0.25` and press the `Return` key. After that, the following message appears in the input window:

```
Circuit needs 01×03 pages.  Accept (y/n)?:  <y>
```

It means that the circuit will be output in the size of three A4 pages. If you want to accept this size, you can type `y`. Otherwise, type `n` in order to select another output size.

After you type `y` and press the `Return` key, the layout that is currently displayed in the workarea will be output and stored in a file with the suffix `ps`, such as `CSA[16].ps` within the current directory.

10

Recursive Specifications

10.1 Odd–Even–Mergesort

In this section we will show the specification of a class of sorting networks, where we exploit the regularities in the underlying algorithm to get a handy recursive description. This class of sorting networks uses the algorithm of Batcher's Odd–Even–Merge ([Knu73]).

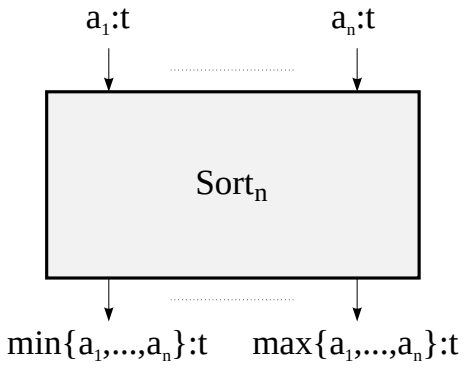


Figure 10.1: Sorting circuit for $n = 2^k$ elements of type t

The task of the circuit can be seen in figure 10.1. The input is a sequence of n elements $a_1, \ldots, a_n$ of an arbitrary, but fixed, type t. The output of the circuit should be the sorted sequence in ascending order. The most important property of our specification is that the sorting algorithm can be described independent of the type t of the elements. As mentioned in section 5.3.2 this is one of the advantages of the use of formal wire variables as it will become clear in the following.

⇝ 5.3.2

The whole circuit can be configured for sorting a specific type of elements by the declaration of a basic compare component `CMP`. This component has to compute the function

$$\texttt{CMP} : T \times T \longrightarrow T \times T \text{ mit } \texttt{CMP}\,(a, b) = (\min(a, b), \max(a, b))$$

Here T denotes the set of values for the elements of type t (e.g. $T = \{0, 1\}$).

An analogous mechanism can be found within programming languages, where a parameter of a procedure may be another procedure. For example you may use a construct like

`SortedList = MergeSort (List, CompareFunction)`,

where `CompareFunction` can be an arbitrary ordering operation on two elements.

10.1.1 The Sorting Algorithm

Our design task here is to specify a circuit that sorts a sequence of elements in increasing order. For simplicity we assume that the number of elements in the sequence is a power of two. The mathematical knowledge needed for this problem is the following:

- If the sequence has length one, then the output is equal to the input.
- To sort a larger sequence, it is sufficient to sort the first half and the second half and to merge the two sorted sequences.
- The merging of two sequences of length one can be done by one basic compare component.
- To merge two sequences a and b of length n greater than one, it is sufficient to merge the even–indexed elements of a with the odd–indexed elements of b, to merge the odd–indexed elements of a with the even–indexed elements of b and to send the outputs of the two half–sized merging circuits through an array of n basic compare components. The ith compare component is then connected to the ith outputs of the two half–sized merging circuits.

The basic algorithm begins with a recursive sort of the first half and the second half of the input sequence. The recursion ends with the input sequence of one element. The two sorted subsequences are then merged by the recursive odd–even merge technique. This recursion ends with the two input sequences each consisting of only one element.

10.1.2 Graphical Specification

Now we must translate these specifications into graphical inputs for CADIC . As in the case of the conditional sum adder (c.f. chapter 4) this can be easily done within a parameterized combinational circuit for sorting n numbers of type t.

As you directly see, the end of the recursion for sorting one single element, is given by a wire from the northern to the southern border of the schematic. The width of this wire is described by the variable `@t` to denote, that it represents a single element of type t.

The graphical input for the general equation for n elements is shown in figure 10.2. This schematic mirrors the second algorithmic statement from above. You see the splitting of the sorting of n elements into the parallel sorting of $\frac{n}{2}$ elements. The resulting presorted sequences are then merged together by an instance `Merge[n]`. The width of the wires in figure 10.2 is given by wire variables, because the number of single binary wires in each bus depends on the number of elements n and on the coding of the element type t. Implicitly the variable `@s[n]` represents $\frac{n}{2}$ elements of type t and `@t[n]` stands for n such elements.

From our mathematical description of the sorting algorithm follows that the merging subcircuit can also be given by a recursive specification. The task of the circuit `Merge[n]` is to merge two presorted sequences of length $\frac{n}{2}$ together to a completely sorted sequence of length n. As mentioned above this can be done by using two instances of `Merge[n/2]` in parallel. The first instances merges the even–index elements of the first sequence with the odd–indexed elements of the second sequence. The second `Merge[n/2]` merges

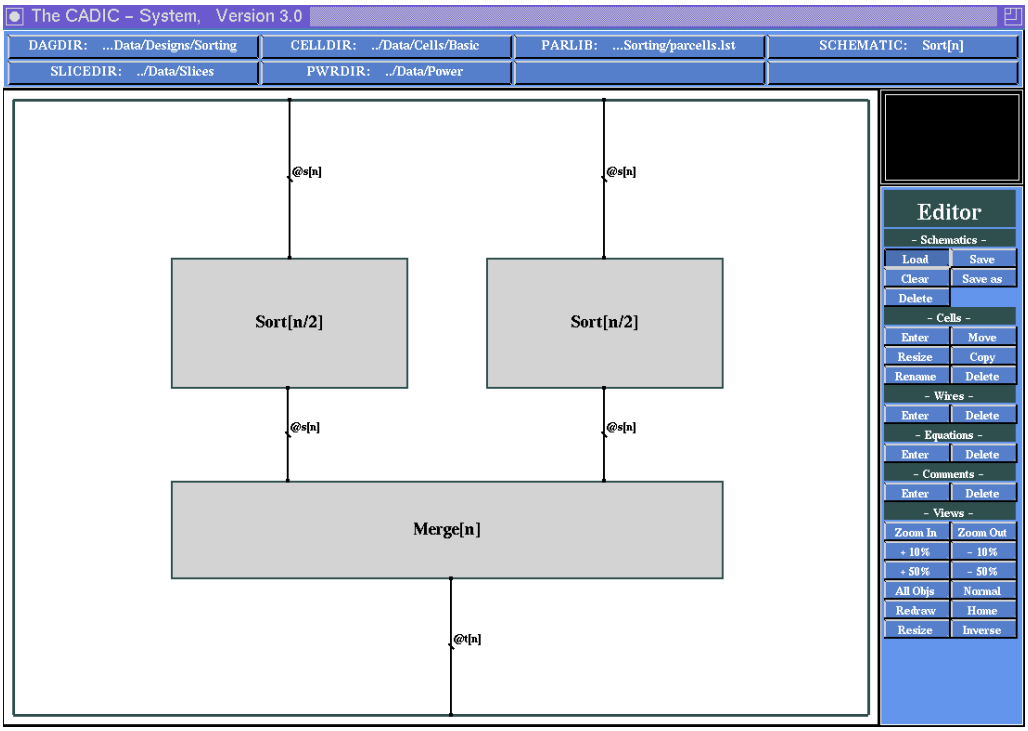


Figure 10.2: Recursive definition of the sorting of $n > 1$ elements

the odd–indexed elements of the first sequence with the even–indexed elements of the second sequence. Finally the ith outputs of these two instances have to be fed into a basic compare component ($0 \leq i < \frac{n}{2}$).

Figure 10.3 shows the graphical specification of `Merge[n]`. The splitting of the input sequences into even–indexed and odd–indexed elements is done by the instance `EvenOdd[n/2]`. In the subcircuit `Shuffle[n/2]` the ith outputs of the two `Merge[n/2]` are routed beside each other, such that they can be fed into an array of $\frac{n}{2}$ basic compare components in `Compare[n/2]`.

The subcircuits `EvenOdd[n]` and `Shuffle[n]` consist only of wiring elements. Their specifications are very similar. The task of `EvenOdd[n]` is to reorder two input sequences $a = a_0, \ldots, a_{n-1}$ and $b = b_0, \ldots, b_{n-1}$ into four subsequences of the form $s_1 = a_0, a_2, \ldots, a_{n-2}$, $s_2 = a_1, a_3, \ldots, a_{n-1}$, $s_3 = b_0, b_2, \ldots, b_{n-2}$ and $s_4 = b_1, b_3, \ldots, b_{n-1}$. `Shuffle[n]` is used to transform two input sequences $a = a_0, \ldots, a_{n-1}$ and $b = b_0, \ldots, b_{n-1}$ into one output sequence $s = a_0, b_0, \ldots, a_{n-1}, b_{n-1}$, i.e. the ith elements of both

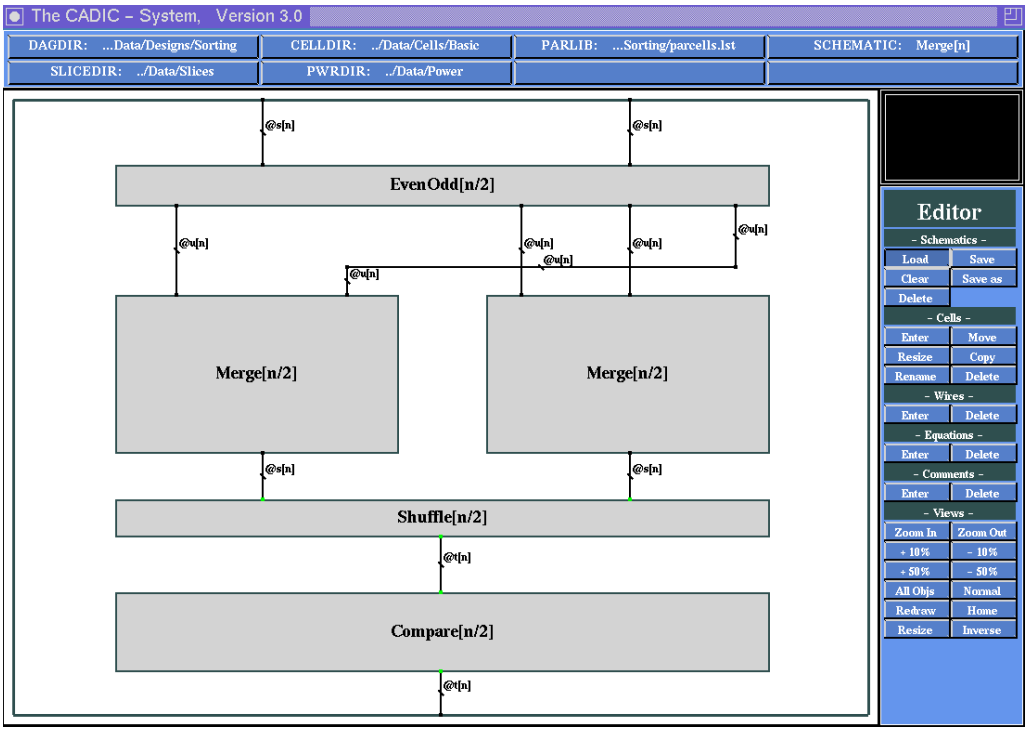


Figure 10.3: Recursive specification for the merging subcircuit `Merge[n]`

input sequences should be routed beside each other.

The shuffling subcircuit can be recursively defined in the following way: to shuffle two input sequences of length $n > 1$ we shuffle the first $\frac{n}{2}$ and the second $\frac{n}{2}$ elements of both sequences in parallel. Then the output sequences of these two instances form the shuffled sequence of `Shuffle[n]`. The basic shuffling operation for two input sequences $a = a_0$ and $b = b_0$ of length $n = 1$ are two simple feed throughs of the element type `@t`.

Figure 10.4 shows the general recursive equation for the shuffling subcircuit. In the left instance `Shuffle[n/2]` the first $\frac{n}{2}$ elements of both input sequences are shuffled together and the last $\frac{n}{2}$ elements in the right instance. The width of a wire representing one half of an input sequence is described by the variable `@r[n]`, and the output of each instance `Shuffle[n/2]` by the expression `2*@r[n]`, because it combines two sequences of width `@r[n]`.

The recursive specification of `EvenOdd[n]` can be derived from the fact that

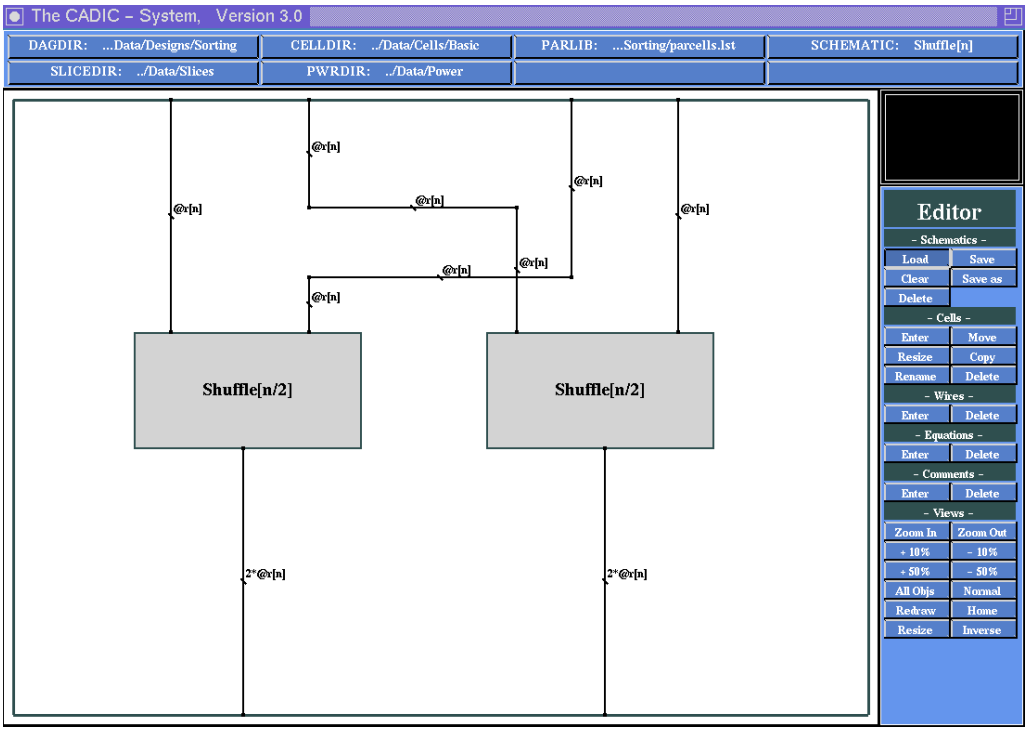


Figure 10.4: Recursive specification of the shuffling operation for two sequences of n elements

we reduce the splitting of two input sequences of length n into the parallel splitting of each sequence into even–indexed and odd–indexed elements. Then the final output of `EvenOdd[n]` is the combination of the even–indexed elements of sequence a with the even–indexed elements of sequence b. The same has to be done for the odd–indexed elements. This means that we have the same arrangement of macros `EvenOdd[n/2]` as in the shuffling subcircuit. But the input wires at the northern border have the width `2*@r[n]`, because it represents the two halfs of sequences a and b. The output of the instances `EvenOdd[n/2]` are four wires of width `@r[n]`, each representing the even–indexed and odd–indexed elements of the input sequences. To get the final result we still have to cross the wire representing the odd–indexed elements from the sequence a with the wire representing the even–indexed elements of sequence b which corresponds to the wiring above the instances in `Shuffle[n]`.

The last subcircuit in `Merge[n]` represents an array of $\frac{n}{2}$ basic compare components. We have already shown the specification of an array of k elements during the description of the conditional sum adder in chapter 4.

⇝ 4.8.4

There we had to specify an array of k pairs of multiplexers in the subcircuit `SEL[k]`. The base of the recursion in this case is given by a call of the basic compare component as it is shown in figure 10.5.

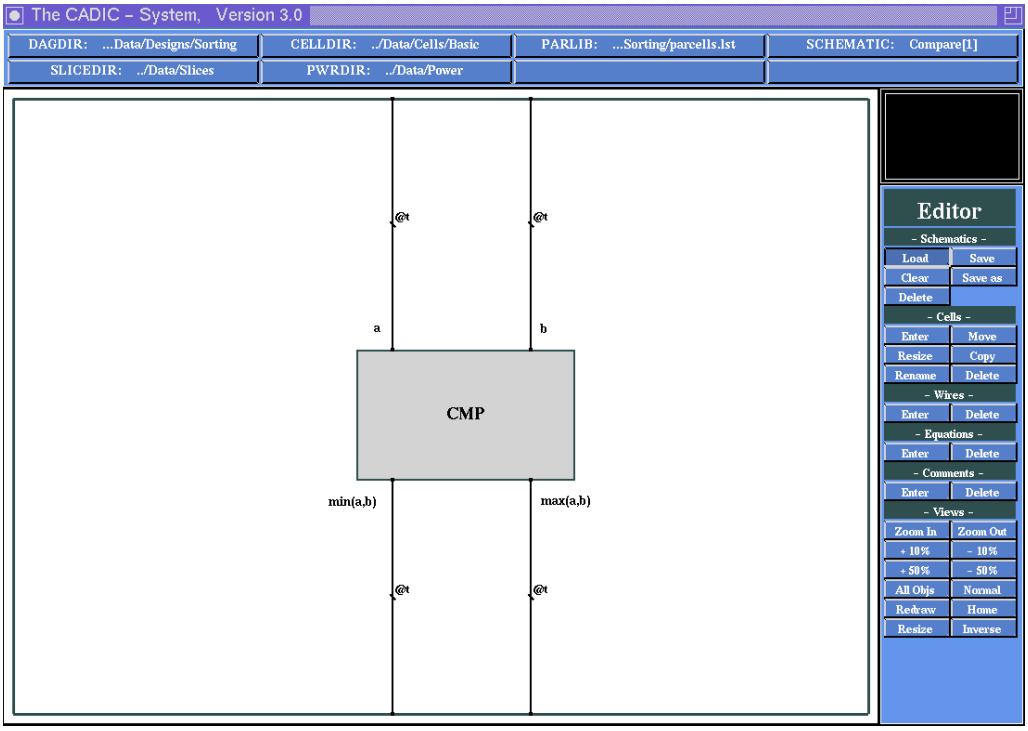


Figure 10.5: Single element of the array of compare components

The specification of the sorting network is independent of the type t of the elements. The inputs shown above can be used to sort a sequence of n elements of arbitrary type into increasing order. The type of the elements only has influence on the refinement of the instance `CMP`. Here we give a specification for the simple example that we want to sort single bit values. We have to implement the function

$$\texttt{CMP} : \{0,1\} \times \{0,1\} \longrightarrow \{0,1\} \times \{0,1\}$$

with

$$\texttt{CMP}(a,b) = (\min(a,b), \max(a,b)).$$

It is obvious that for two single bits the minimum of both is given by an AND gate and the maximum is the output of an OR gate. You can define an appropriate schematic `CMP` with two instances `AND2` and `OR2` which are both connected to the two input wires.

Figure 10.6 shows the layout for the sorting network for 64 single bit values. You can find all schematics for the description of `Sort[n]`, if you start the shell script `merge_sort` from the `Examples` directory. All schematics needed to create the layout from figure 10.6 are `Sort[n]`, `Sort[1]`, `Merge[n]`, `Merge[1]`, `EvenOdd[n]`, `EvenOdd[2]`, `Suffle[n]`, `Shuffle[2]`, `Compare[n]` and `Compare[1]`. The schematic `CMP` has to be replaced by the appropriate compare function for the specific type of elements.

In the `DAGDIR` from this shell script there is also a parameterized description of the compare component for binary values of k bit. If you want to sort 16 bit integer values for example, you must call the component `Cmp[16]` within the schematic `CMP`, i.e. replace the circuit from above (`AND2` and `OR2` gate) by this macro.

Lemma 10.1: *The complexity of the sorting network is given by*

$$size(\texttt{Sort[n]}) = \frac{\log n \cdot (\log n + 1)}{2} \cdot \frac{n}{2} \cdot C,$$

where C denotes the size of the basic compare component. □

Proof: From the recursive definition we get the following equations:

$$\begin{aligned}
size(\texttt{Sort[n]}) &= 2 \cdot size(\texttt{Sort[n/2]}) + size(\texttt{Merge[n]}),\\
size(\texttt{Merge[n]}) &= 2 \cdot size(\texttt{Merge[n/2]}) + size(\texttt{Compare[n/2]}),\\
size(\texttt{Sort[1]}) &= 0,\\
size(\texttt{Merge[2]}) &= C,\\
size(\texttt{Compare[n]}) &= 2 \cdot size(\texttt{Compare[n/2]}),\\
size(\texttt{Compare[1]}) &= C,
\end{aligned}$$

The subcircuits `EvenOdd[n]` and `Shuffle[n]` have size zero, because they do not contain any basic cell.

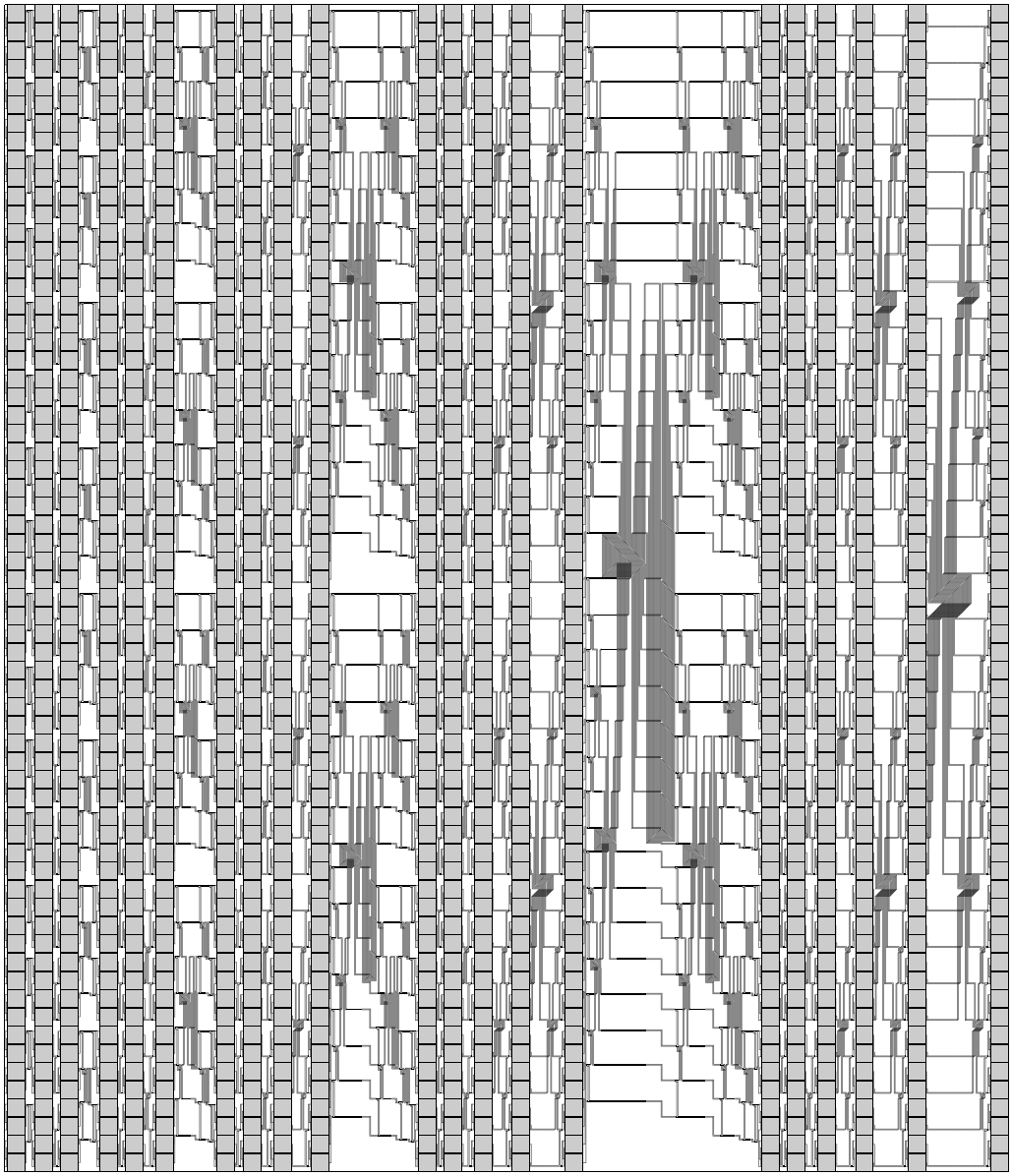


Figure 10.6: Layout for the sorting network for 64 elements of single bit values

From the first two equations it follows

$$size(\texttt{Sort[n]}) = \sum_{i=0}^{\log n-1} 2^i \cdot size(\texttt{Merge}[\frac{n}{2^i}])$$

and

$$size(\texttt{Merge[n]}) = \sum_{i=0}^{\log n-1} 2^i \cdot size(\texttt{Compare}[\frac{n}{2^{i+1}}])$$

Obviously it holds

$$size(\texttt{Compare[n]}) = n \cdot C$$

Then we have

$$\begin{aligned} size(\texttt{Merge[n]}) &= \sum_{i=0}^{\log n-1} 2^i \cdot \frac{n}{2^{i+1}} \cdot C \\ &= \log n \cdot \frac{n}{2} \cdot C \end{aligned}$$

and

$$\begin{aligned} size(\texttt{Sort[n]}) &= \sum_{i=0}^{\log n-1} 2^i \cdot \log \frac{n}{2^i} \cdot \frac{n}{2^{i+1}} \cdot C \\ &= \frac{n}{2} \cdot C \cdot \sum_{i=0}^{\log n-1} (\log n - i) \\ &= \frac{n}{2} \cdot C \cdot (\log^2 n - \frac{(\log n - 1) \cdot \log n}{2}) \\ &= \frac{\log n \cdot (\log n + 1)}{2} \cdot \frac{n}{2} \cdot C \end{aligned}$$

■

Lemma 10.2: *The complexity of the DAG structure for* `Sort[n]` *is* $5 \cdot \log n + 2$ *treenodes for all* $n > 1$. □

Proof: We leave the formal proof of this lemma to the reader. You can easily derive it, if you consider that there are five subcircuits, which have a recursive description of logarithmic size, namely `Sort`, `Merge`, `Shuffle`, `EvenOdd` and `Compare`. ■

10.2 Integer Multiplication

Now we will demonstrate the power of recursive specifications by the example of a family of multipliers for binary numbers. This type of multiplier is a tree multiplier which is a modified version of a Wallace tree multiplier ([Wal64]) made suitable for VLSI design by Luk and Vuillemin ([LV83]).

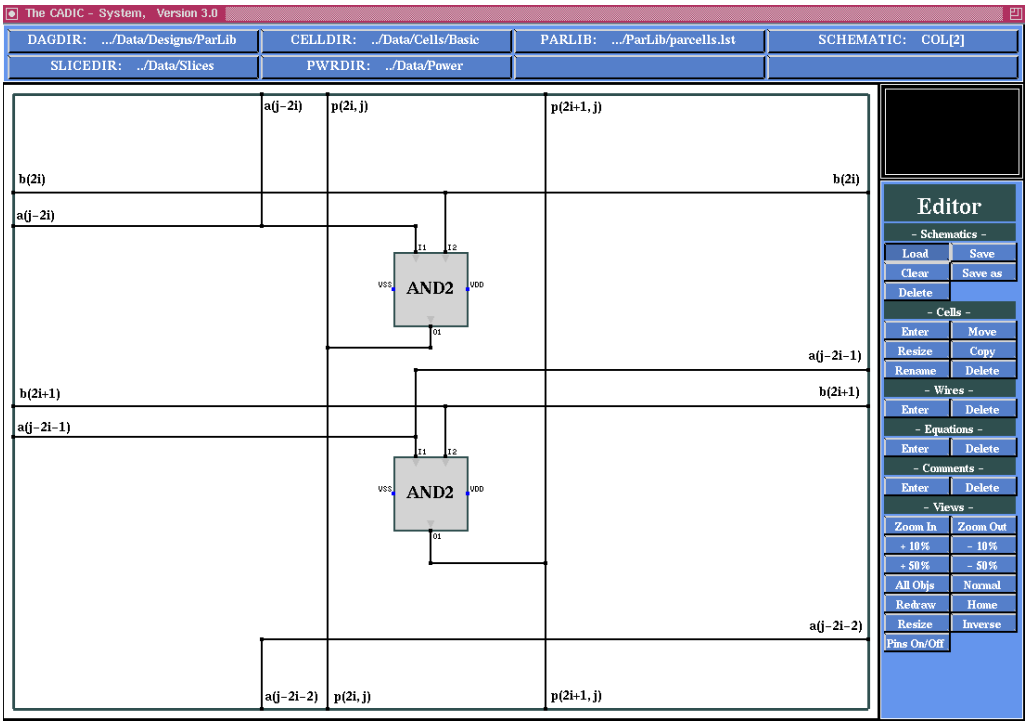


Figure 10.7: Computation of the partial products

Let $a = (a_{n-1}, \ldots, a_0)$ and $b = (b_{n-1}, \ldots, b_0)$ be the binary representations of the two factors. The product of a and b is equal to the sum of n binary numbers of length $2n$. These numbers are represented by the n lines of the following matrix P_n, where $a_i b_j$ denotes the *logical and* or the product of the bits a_i and b_j.

$$\begin{pmatrix} 0 & 0 & \cdots & 0 & 0 & a_{n-1}b_0 & a_{n-2}b_0 & \cdots & a_1b_0 & a_0b_0 \\ 0 & 0 & \cdots & 0 & a_{n-1}b_1 & a_{n-2}b_1 & a_{n-3}b_1 & \cdots & a_0b_1 & 0 \\ \vdots & & & & \vdots & \vdots & & & & \vdots \\ 0 & a_{n-1}b_{n-1} & \cdots & a_2b_{n-1} & a_1b_{n-1} & a_0b_{n-1} & 0 & \cdots & 0 & 0 \end{pmatrix}$$

The matrix P_n shows how to compose an n bit multiplier by $2n$ identical columns (or n identical rows). These n numbers are called the partial products of a and b. With the sequential method (shift and add algorithm) the partial products are accumulated one after the other. This can be speeded up by parallelizing the addition steps to a time of $O(\log n)$.

In a first step we have to compute the values of the partial products in each column. This will be done by the subcircuit `col[2]` as it is shown in figure 10.7. It is obvious that we need this subcircuit n^2 times in the multiplier.

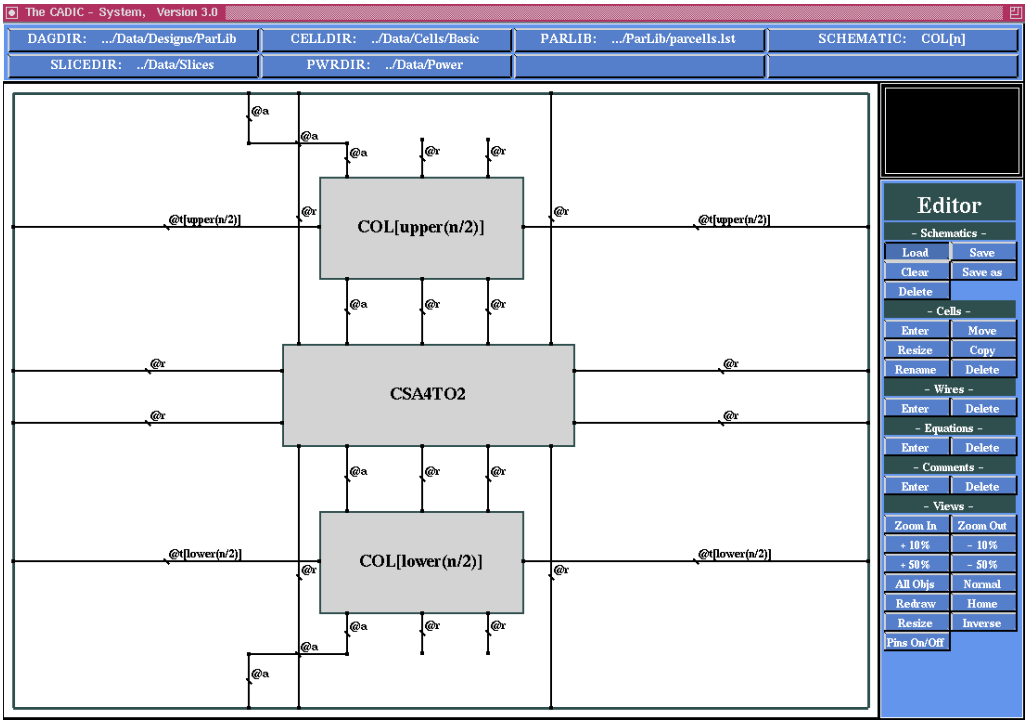


Figure 10.8: Recursive specification of one column of the partial multiplier

The idea of the tree multiplier is to reduce the number of rows in each step by the half. This is done in each column by combining four partial products to two, i.e. it holds the equation

$$p_{4i}^k + p_{4i+1}^k + p_{4i+2}^k + p_{4i+3}^k = p_{2i}^{k+1} + p_{2i+1}^{k+1}$$

where $i \in \{0, \ldots, \frac{n}{2^{k+2}} - 1\}$ denotes the matrix element and k the depth of

the reduction. Figure 10.8 shows the recursive specification of one single column of the n bit multiplier. You see there the reduction of two column parts of height $\frac{n}{2}$ by a subcircuit `CSA4to2`. At the end of the reduction we have two binary numbers of length $2n$ which have to be added to get the final result of the multiplication.

As the equation from above implies the reduction subcircuit `CSA4to2` can be realized by two full adders as it is shown in figure 10.9.

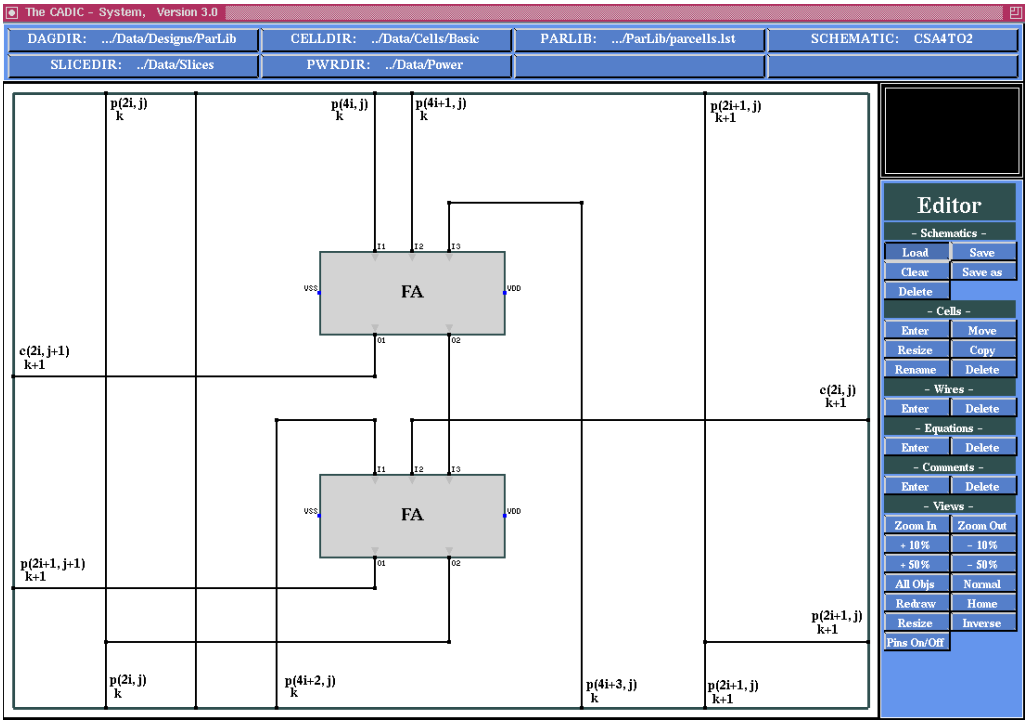


Figure 10.9: Recursive specification of one column of the partial multiplier

Now the n bit multiplier `mul[n]` is given by an instance of the subcircuit `c[n,2*n]`, where `c[n,i]` contains i columns of the n bit multiplier.

Because we proposed at the beginning of this section that $n = 2^k$ for some $k \geq 1$, we can compose i columns `c[n,i]` by two instances of `c[n,i/2]`. This shows that each wire segment of type `@k[i]` is replaced by two wires of type `@k[i/2]`.

Figure 10.10 shows the layout for the 16 bit integer multiplier. This circuit

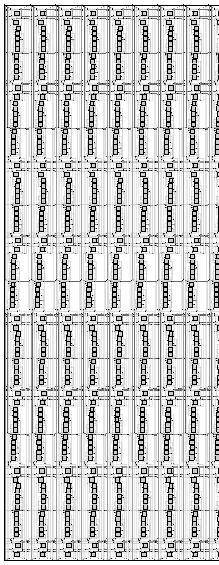


Figure 10.10: Layout for the 16 bit integer multiplier

contains about 3000 basic cells, the calculation of the shown layout takes nearly three minutes on a workstation.

Because of our paramterized description we could extract any version as for example an 1024 bit integer multiplier. It is obvious that this circuit would be too large to fit on one single chip. For that purpose in [HMZ91] it is shown how a very large integer multiplier ($n \gg 64$) can be described with the help of CADIC . The whole circuit is distributed over several chips, where the following goals have been respected:

- the multiplier should still have an optimal performance $O(\log n)$. An important fact to achieve this goal is a small amount of communication between the different chips.
- the design should be built over a small number of different types of chips. This is important to minimize the fabrication and test costs, because for each chip type we have to build an expensive prototype.
- the total number of chips should be minimized. This means that each chip type has to be very regular and compact in its layout in order to minimize the number boards.

The main problem with this design is to find a good splitting of the circuit, such that the number of different chip types and the communication between the chips are minimized. In [HMZ91] it is shown that from the basic principle of the Wallace tree multiplier as it is presented above a splitting into three different chip types can be found:

- *chip type 1* calculates the matrix of the partial products. For this type we can use an 32 bit integer multiplier as we have designed it above.
- *chip type 2* reduces the partial products according to the Wallace tree principle and produces two bit strings of length $2n$, which have to be added in a final step.
- *chip type 3* build a $2n$ bit adder for the final summation of the reduced partial products.

The following tabular illustrates the complexity of the whole design for an

512 bit and an 1024 bit multiplier. The number of chips is based on the fact that today a 32 bit multiplier or a 64 bit adder can be realized on one single chip.

bitlength	chip type 1	chip type 2	chip type 3	total number
512	64	76	9	149
1024	256	240	17	513

10.3 A Realization of a Fast Divider

10.3.1 Introduction

This is the description of the realization of a divider which has computation time $O(n)$ where n is the length of one operand in bits. It is faster than a conventional array–divider based on a subtract–and–shift algorithm (which has a computation time of $O(n^2)$) but needs a similar amount of hardware. The divider is based on an algorithm using a redundant binary number representation.

Since the divider is realized as a parameterized design using CADIC , it is possible to compute the layout for arbitrary wordlength ($n \geq 3$) of the operands by simply changing the parameter value of n.

For more information see: 'An On-Line Error-Detectable Array Divider with a Redundant Binary Representation and a Residue Code' by Naofumi Takagi and Shuzo Yajima of the Department of Information Science at the Kyoto University (Proceedings: International Symposium on Fault-Tolerant Computing 1988)

10.3.2 General Description of the Divider

Inputs

`Divider[n]` takes $2(n-1)$ bits as inputs which are numbered $x_1, \ldots, x_{n-1}$ and $y_1, \ldots, y_{n-1}$. These are interpreted as follows:

$$x = [1.x_1 \ldots x_{n-1}] \quad \text{with value}$$

$$\|x\| = 1 + \sum_{i=1}^{n-1} x_i \cdot 2^{-i}$$

$$y = [1.y_1 \dots y_{n-1}] \quad \text{with value}$$

$$\|y\| = 1 + \sum_{i=1}^{n-1} y_i \cdot 2^{-i}$$

Precondition: $\|x\| \geq \|y\|$

Outputs

`Divider[n]` has n bits as outputs numbered $z_1, \dots, z_n$. These are interpreted as follows:

$$z = [1.z_1 \dots z_{n-1}] \quad \text{with value}$$

$$\|z\| = 1 + \sum_{i=1}^{n} z_i \cdot 2^{-i}$$

$\|z\|$ has the following property:

$$\left| \|z\| - \frac{\|x\|}{\|y\|} \right| < 2^{-n}$$

Redundant binary numbers

The division–algorithm uses a redundant binary number representation. There are three different digits: $\bar{1}, 0$ and 1. The values of these digits are $-1, 0$ and 1 respectively.

The value of a bitstring x in this representation is denoted by $\|x\|_{SD2}$. If a string s is written $[s]_{SD2}$ the digits are redundant digits ($\bar{1}, 1$, or 0).

$$\text{If} \quad A = [a_0.a_1 \dots a_{n-1}]_{SD2}; \quad a_i \in \{\bar{1}, 0, 1\}$$

$$\text{then} \quad \|A\|_{SD2} = \sum_{i=0}^{n-1} a_i \cdot 2^{-i}$$

The digits of the representation are coded in binary as follows:

$$\begin{array}{lcll} \bar{1} & \rightarrow & 01 & \\ 0 & \rightarrow & 00 & \text{or} \quad 11 \\ 1 & \rightarrow & 10 & \end{array}$$

Using this representation, it is possible to perform parallel addition of two binary numbers in constant time independent of the wordlength of the operands. This causes the speedup in the algorithm.

The Division Algorithm

The iteration is done using the following rule:

$$R_j = R_{j-1} - q_j \cdot 2^{-j} \cdot Y \quad (j \geq 1)$$

and we start with $R_0 = X - Y; \quad q_0 = 1.$
(So R_j is the remainder at stage j)

The q_j are chosen so that the following inequality holds:

$$\left|\frac{R_j}{Y}\right| \leq 2^{-j}$$

Representation of the R_j:

$$R_{j-1} = [r_0^{j-1}.r_1^{j-1} \ldots r_{j-3}^{j-1} r_{j-2}^{j-1} r_{j-1}^{j-1} \ldots r_{n+j-2}^{j-1}]_{SD2}$$
$$\text{for} \quad j = 1, \ldots, n-1$$

Steps of the Algorithm

1. $q_0 = 1; \quad R_0 = X - Y$
   (Performed by the top–component)

2. for $j := 1$ to n do
   begin
   - $$q_j := \begin{cases} -1 & : \quad \|r_{j-2}^{j-1} r_{j-1}^{j-1} r_j^{j-1}\|_{SD2} < 0 \\ 0 & : \quad \|r_{j-2}^{j-1} r_{j-1}^{j-1} r_j^{j-1}\|_{SD2} = 0 \\ 1 & : \quad \|r_{j-2}^{j-1} r_{j-1}^{j-1} r_j^{j-1}\|_{SD2} < 0 \end{cases}$$

     (Performed by the DU-component)
   - $$R_j := R_{j-1} - q_j \cdot 2^{-j} \cdot Y$$

     (Performed in DU-, M- and L-Cells)

end

3. Transformation of $[q_1 \dots q_n]_{SD2}$ into a binary number $[z_1 \dots z_n]$ such that

$$\|1.q_1 \dots q_n\|_{SD2} = \|1.z_1 \dots z_n\|$$

(Performed in the Red2BinFast–component)

10.3.3 Graphical Specification of the Divider

Top Level: `Divider[n]`

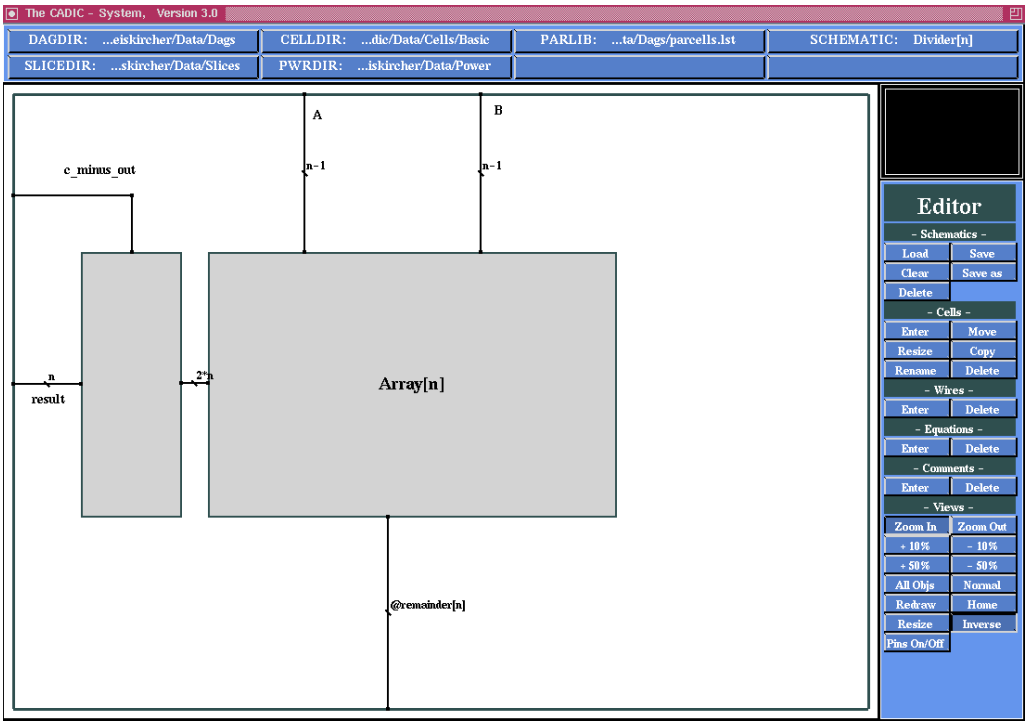


Figure 10.11: The top level of `Divider[n]`

The `Array[n]`–component (see figure 10.11) is the part where the division is performed. It gets the two operands A and B.

The `Red2BinFast[n]`–component (left of the `Array[n]`-component) transforms the result from redundant–binary format to standard binary format.

`Array[n]`

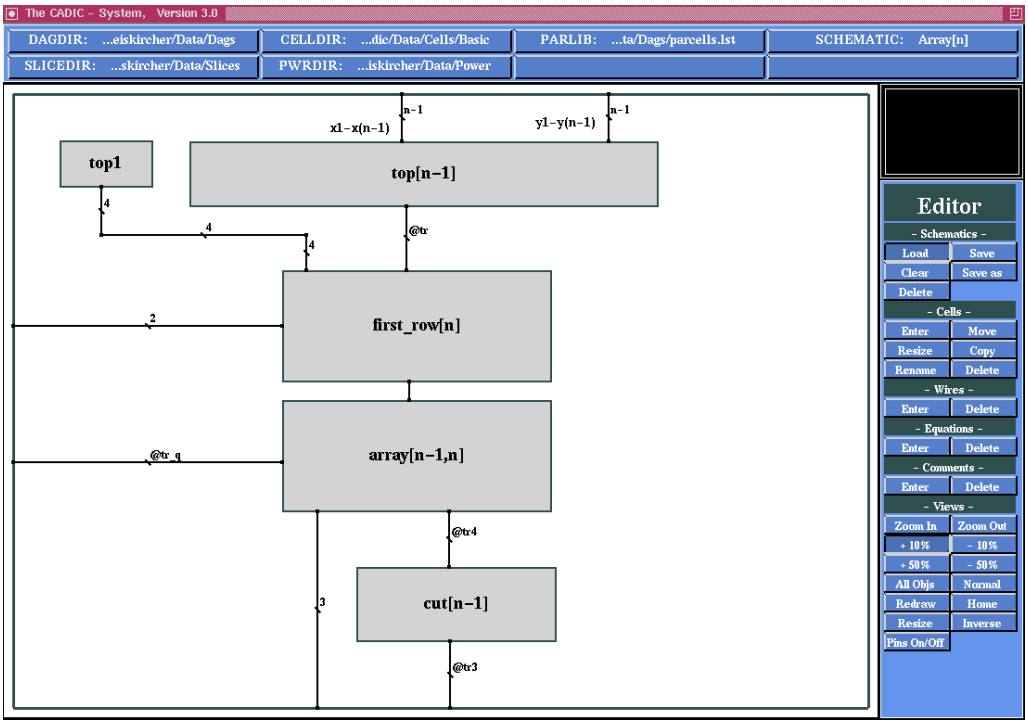


Figure 10.12: `Array[n]`

The `top1`–component (see figure 10.12) feeds 4 bits with value zero into the leftmost inputs of `first_row[n]`.

The `top[n-1]`–component performs the first step of the algorithm. B is subtracted from A and the result is transformed into the right format for the first row of the divider.

The `first_row[n]`–component is very similar to the remaining $n-1$ rows of the divider (in the `array[n-1,n]`–component), in fact it has the same components but it's wiring is slightly different because it is the uppermost row.

The `array[n-1,n]`–component consists of $n-1$ identical rows where the subtraction and the computation of the result bits is performed.

The `cut[n-1]`–component simply cuts every third bit of the output from the last row. This is done to get rid of the divisor–bits that are channeled though the whole array so that we only have the remainder–digits at the

bottom of the design.

`top[n]`

The `shuffle[n]`–component performs the first subtraction by simply transforming the input $a_{n-1} \ldots a_0 b_{n-1} \ldots b_0$ into $b_{n-1} a_{n-1} b_{n-2} a_{n-2} \ldots b_0 a_0$. The sequence is other than one would expect (a first and then b) because the inputs of the rows are oriented that way.

The `double_y[n]` component doubles every second bit in it's input. It transforms $x_{2n-1} x_{2n-2} x_{2n-3} \ldots$ into $x_{2n-1} x_{2n-2} x_{2n-2} x_{2n-3} \ldots$. This is done to fit the inputs of the first row.

`first_row[n]`, `array[1,n]`

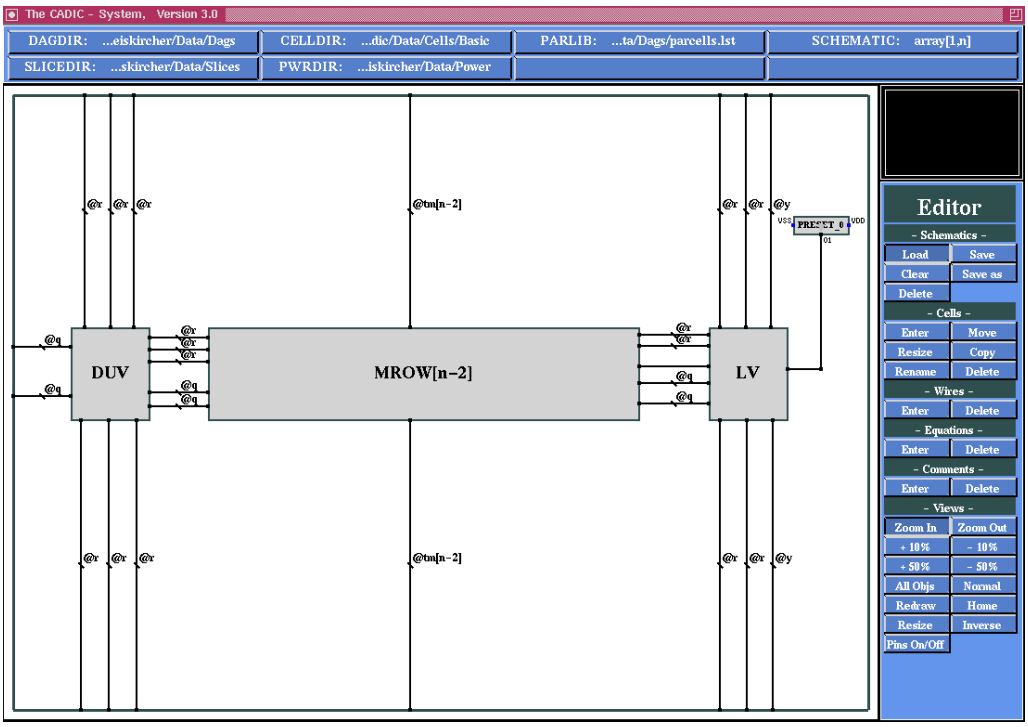


Figure 10.13: One row of the division array

The `DUV`–component is the `DU`–component with some additional wiring. The `LV`–component is the `L`–component with wiring. The `MROW[n-2]`–component consists of $n-2$ `MV`–components.

The only difference between the first row and the $n-1$ following rows is the one–bit line entering `LV` in the middle of the east side of the component. In all other rows (named `array[1,n]`) (see figure 10.13) this input is fed by a preset having value zero.

`DUV`

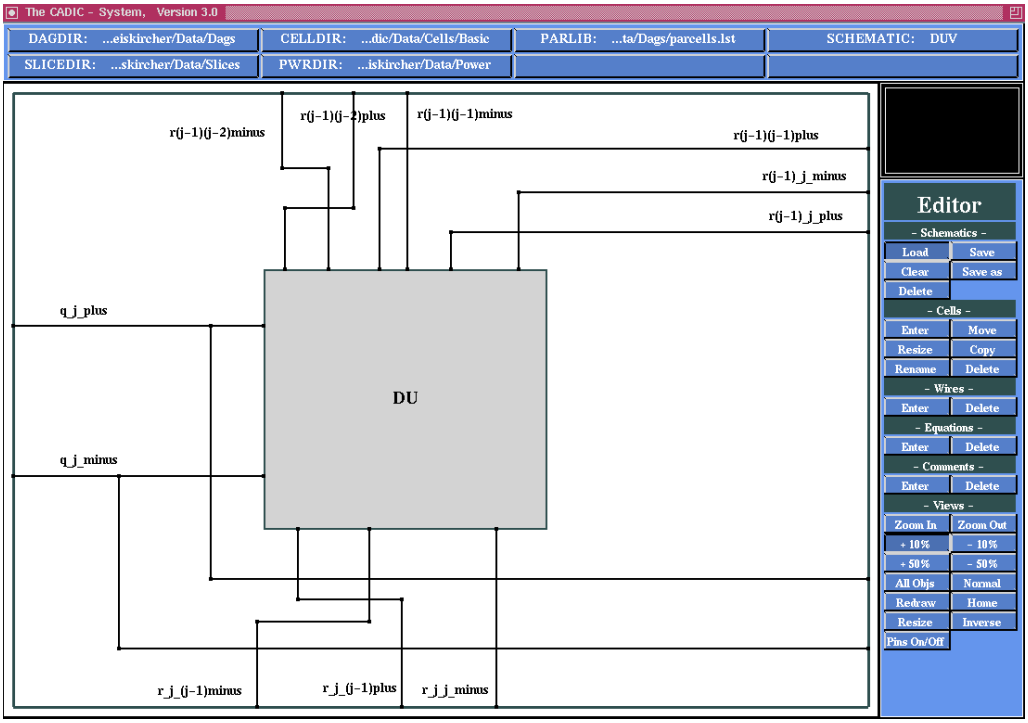


Figure 10.14: First part of each row `DUV`

The `DU`–component (see figure 10.14) has two tasks: It has two calculate the two result–bits for the row and it has to perform the subtraction at the three most significant digits. These digits are named $r_{j-2}^{j-1}, r_{j-1}^{j-1}$ and r_j^{j-1} (in order of significance). Because they are in redundant binary format, each digit is represented by two bits (minus and plus) with the meaning defined in section 10.3.2 (the plus–bit is the first bit of every digit). The result–digit q_j is calculated according to the algorithm in section 10.3.2.

Note that the high index of a remainder–digit r corresponds to the number of the row where the digit was computed while the low index corresponds to the index of the digit in the remainder of this row.

The remainder digit $r_{j-1}^{j}{}^{-}$ and the minus–bit of the remainder–digit r_j^j are calculated according to the following table:

r_{j-2}^{j-1}	r_{j-1}^{j-1}	$\bar{1}$	0	1
$\bar{1}$	0	/	$\bar{1},\bar{1}$	$\bar{1},0$
$\bar{1}$	1	$\bar{1},0$	$0,\bar{1}$	$0,0$
0	$\bar{1}$	$\bar{1},0$	$0,\bar{1}$	$0,0$
0	0	$0,0$	$0,0$	$0,\bar{1}$
0	1	$0,\bar{1}$	$0,0$	$1,\bar{1}$
1	$\bar{1}$	$0,\bar{1}$	$0,0$	$1,\bar{1}$
1	0	$1,\bar{1}$	$1,0$	/

The entries in the table are a, b where a is the redundant digit for r_{j-1}^j and b is the bit for $r_j^{j^-}$. The last three columns stand for the different values of the digit r_j^{j-1}.
So the digits of r_{j-2}^{j-1} and r_{j-1}^{j-1} determine the row and the digit for r_j^{j-1} determines the column.

`MV`

The `MV`–component (see figure 10.15) is an `M` component with some additional wiring. The `MROW[n]`–component consists of n such components. 3 of it's input–bits are simply forwarded to the next MV-cell (namely the digit r_{i-1}^{j-1} and the single bit $r_{i-2}^{j-1^+}$. The other inputs (the result–digit q_j, the divisor–bit y_{i-j} and the remainder–digit r_i^{j-1}) are used in the M-component to compute two bits of two adjacent remainder–digits by performing a subtraction in the redundant binary format according to the algorithm in section 10.3.2.

The computed function is shown using three tables (for space reasons). In the first step, a bit b_i^j is computed using the following rule:

$$b_i^j = \begin{cases} y_{i-j} & \text{if} \quad q_j = \bar{1} \\ 0 & \text{if} \quad q_j = 0 \\ \tilde{y}_{i-j} & \text{if} \quad q_j = 1 \end{cases}$$

This results in the following table for the computation of b_i^j:

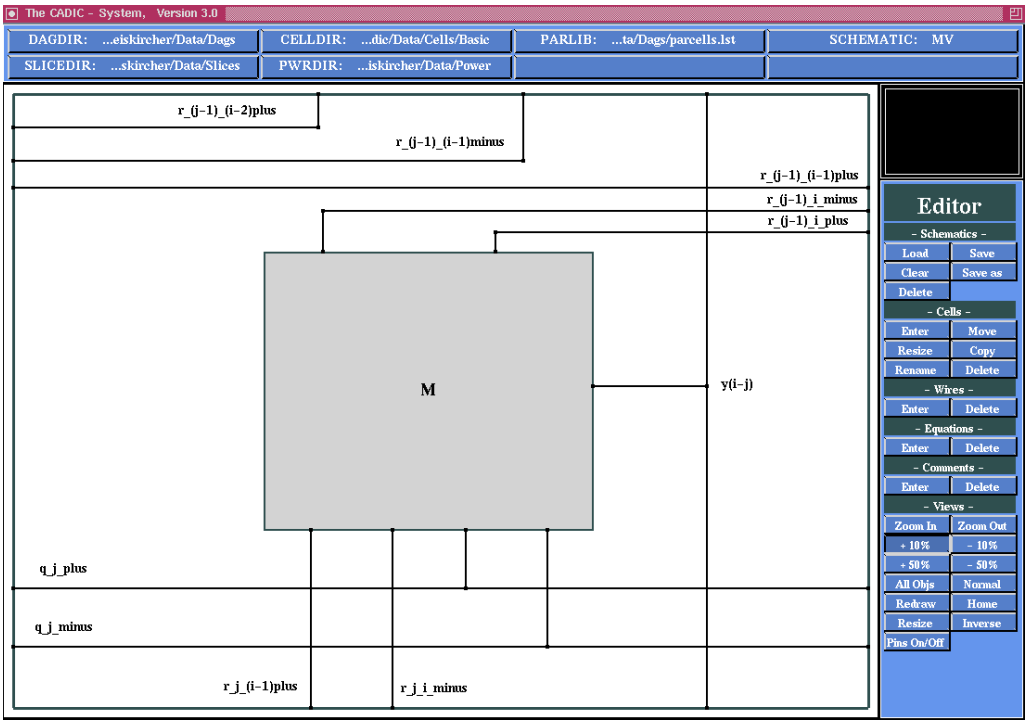


Figure 10.15: Elements in the middle of each row `MV`

q_j	0	1
1	1	0
0	0	0
$\bar{1}$	0	1

The columns 0 and 1 are determined by the value of y_{i-j}. Note that q_j is the result–bit of the row and y_{i-j} is a bit of the divisor.

Now we come to the subtraction rule:

r_i^{j-1}	0	1
1	$1, \bar{1}$	$1, 0$
0	$0, 0$	$1, \bar{1}$
$\bar{1}$	$0, \bar{1}$	$0, 0$

The columns 0 and 1 are determined by the value of b_i^j. The first component in every entry is a bit called c_i^j which corresponds to the bit ${r_{i-1}^j}^+$ in `MV`. The second component is a redundant binary digit called s_i^j which is used

together with the bit c_i^j to compute the redundant binary digit r_i^j in the following way:

s_i^j	0	1
0	0	1
$\bar{1}$	$\bar{1}$	0

The columns 0 and 1 are determined by the value of c_i^j. The output–bit ${r_{i-1}^j}^-$ is the minus–bit of the r_i^j, so it is 1 when the entry in the table is $\bar{1}$ and 0 otherwise.

LV

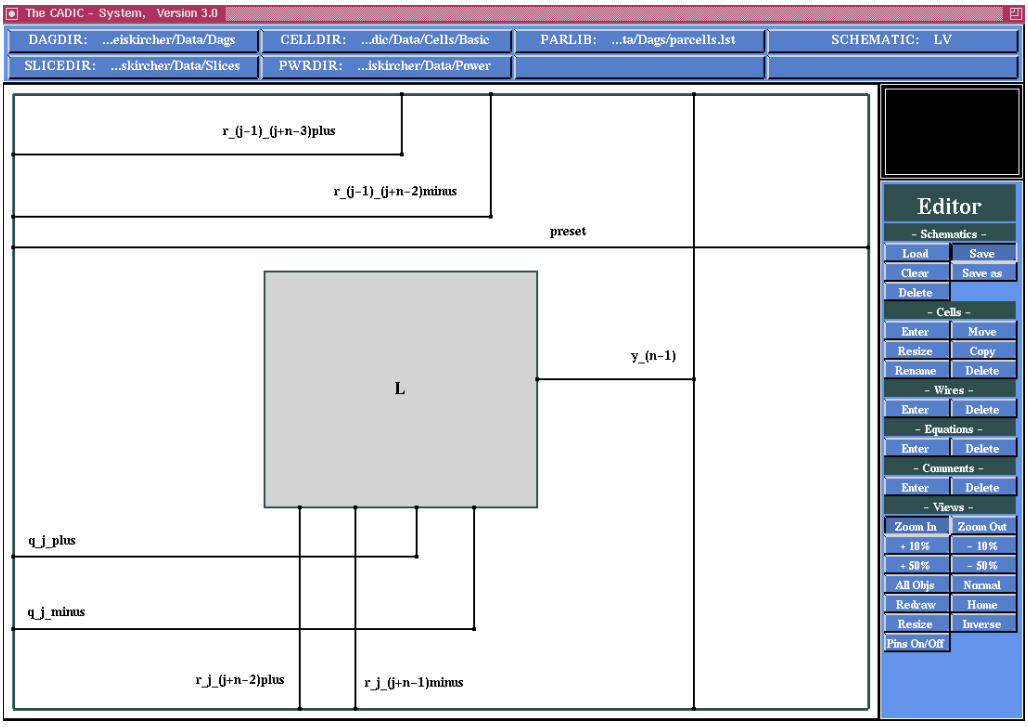


Figure 10.16: The last element in each row LV

The LV–component (see figure 10.16) is an L–component with additional wiring. The wiring forwards the 3 input–bits ${r_{j+n-3}^{j-1}}^+$, ${r_{j+n-2}^{j-1}}^-$ and *preset* to the first MV–component in the row. It is the task of the L–component to perform the subtraction at the least significant position. The L–component

takes the bit y_{n-1} and the redundant binary digit q_j (result–digit of the row) and computes it's output–bits according to the following table:

q_j	0	1
1	$1,0$	$1,\bar{1}$
0	$0,0$	$0,0$
$\bar{1}$	$0,0$	$1,\bar{1}$

The columns 0 and 1 are determined by the value of y_{n-1}. The first component in every entry is a bit called c^j_{j+n-1} which corresponds to the output–bit ${r^j_{j+n-2}}^+$ in `LV`. The second component is the redundant binary digit r^j_{j+n-1}. The output bit ${r^j_{j+n-1}}^-$ of `LV` is 1 if $r^j_{j+n-1} = \bar{1}$ and 0 otherwise.

`Red2BinFast[n]`

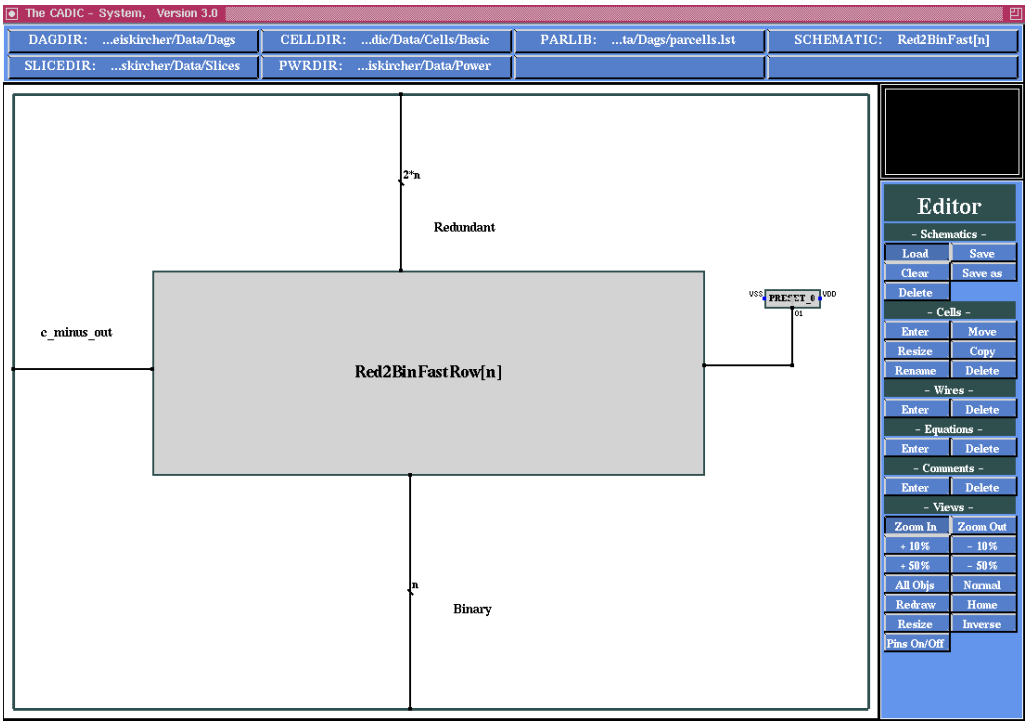


Figure 10.17: Reduction from the redundant representation to standard binary representation `Red2BinFast[n]`

This component (see figure 10.17) is used to transform a number $[x]_{SD2}$ with n redundant binary digits into a number y with n binary digits.

If $\|x\|_{SD2} \geq 0$ then $\|y\| = \|x\|_{SD2}$ and $c\_minus\_out = 0$,
otherwise $c\_minus\_out = 1$ (which will not occur if the second operand of the divider is not greater than the first operand, see section 10.3.2)

The preset(zero) gives the carry–in for the `Red2BinFastRow[n]` component.

`Red2BinFastRow[n]`

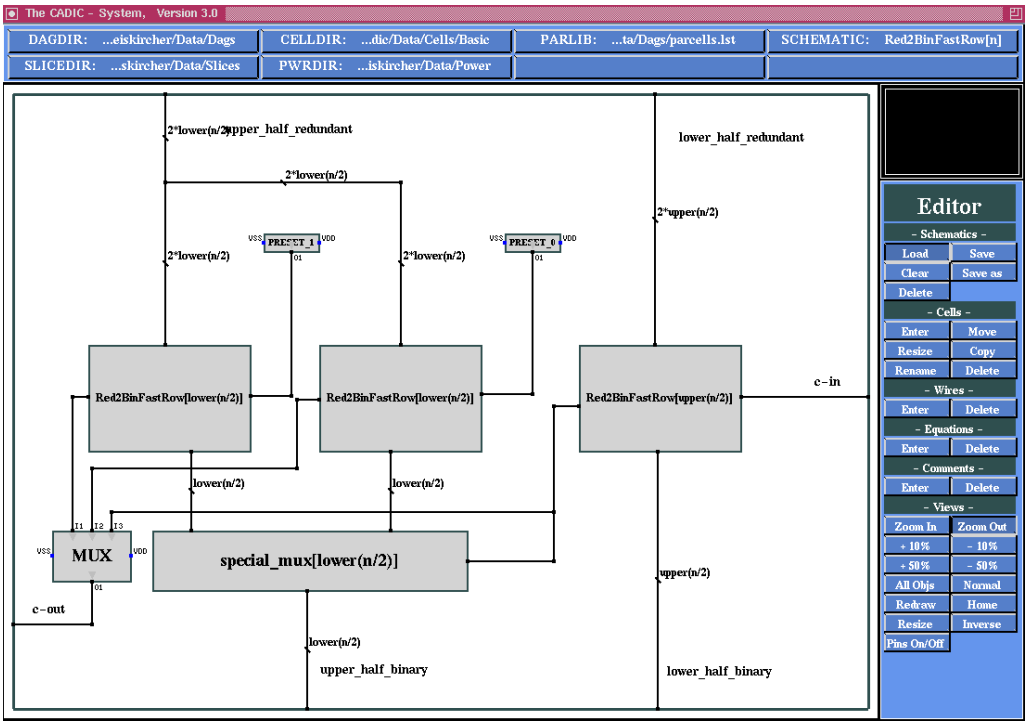


Figure 10.18: Recursive specification for `Red2BinFastRow[n]`

The transformation is done using the conditional–sum–principle. The component is defined recursively down to the component `Red2BinFastRow[1]` (see figure 10.18) exactly like a conditional–sum–adder. The function computed by the
`Red2BinFastRow[1]`–component is shown in the following table:

q	0	1
1	$1,0$	$0,0$
0	$0,0$	$1,1$
$\bar{1}$	$1,1$	$0,1$

The columns 0 and 1 are determined by the carry–in of the component. The first component of every entry is the sum–bit, the second component is the carry–out.

The $special\_mux[n]$-component gets input $x_{n-1} \ldots x_0 y_{n-1} \ldots y_0$. This is first transformed by a shuffle–component into $x_{n-1} y_{n-1} \ldots x_0 y_0$ and then given into n muxes using the same select–signal. So it forwards x if $select = 1$ and y otherwise.

Figure 10.19 shows the final layout for the 16 bit divider without the subcircuit for the reduction of the result from the redundant to the binary representation.

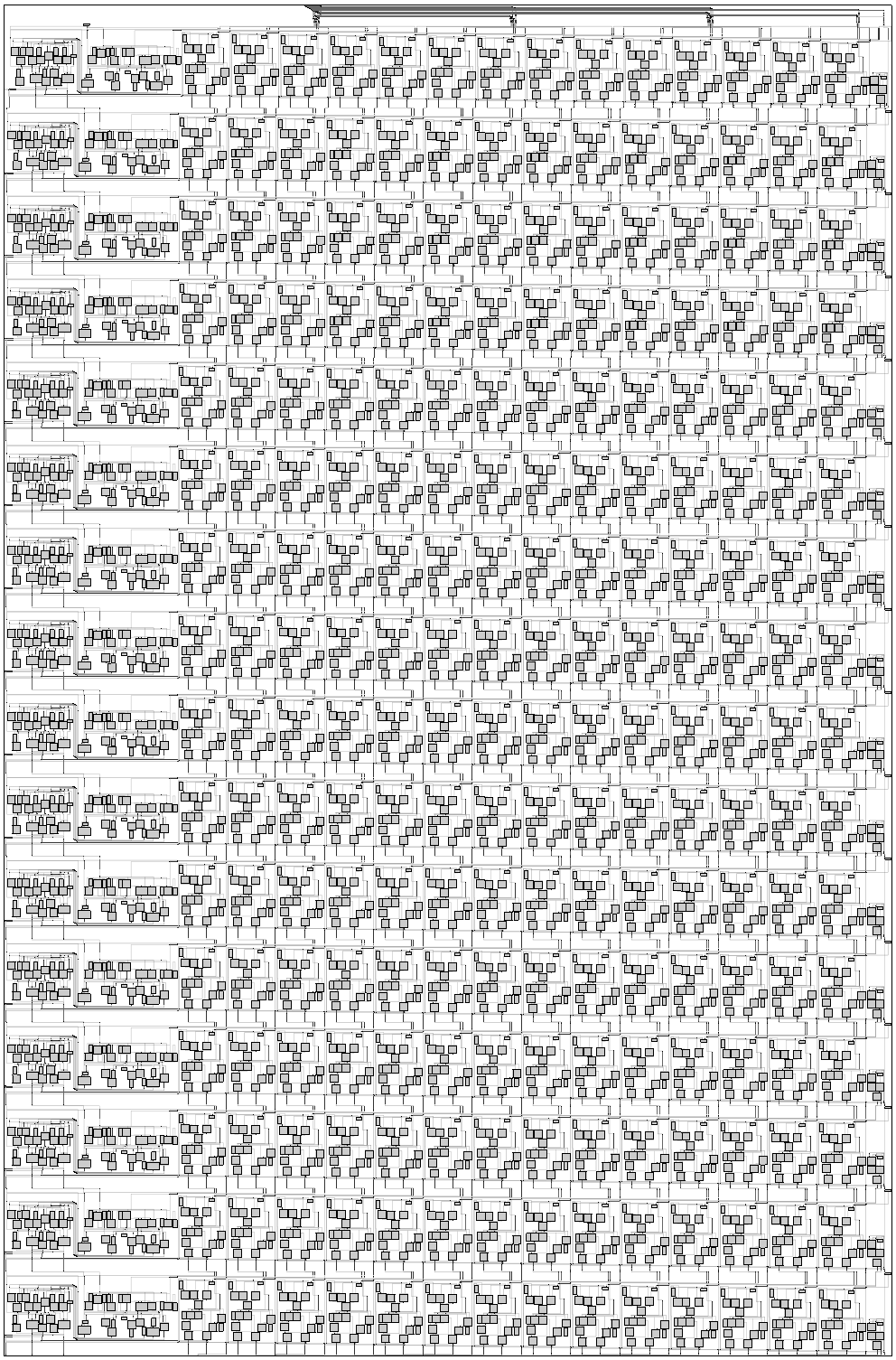


Figure 10.19: Layout for the 16 bit divider `Divider[16]`

11

Design Conversion

11.1 Introduction

In CADIC there are a lot of interfaces to convert its internal netlist format into the exchange formats of commercial design systems. With the help of these interfaces the designer can use the advantages of the graphical editor, especially its parametric design methods, in a comfortable graphical frontend.

The conversion tool in CADIC can generate different exchange formats out of the graphical specification of a circuit. The tool works on the DAG structure, from which it first builds a representation in form of a hierarchical netlist. In addition to the Cgraph structure each treenode in the DAG holds a pointer to a netlist data structure for the corresponding hierarchy level. Because some of the generated formats do not support a hierarchical description, we have to expand the hierarchy into a flat netlist. This operation can be done by an additional tool which can be called before the exchange format is written to the output file (c.f. figure 11.1).

The hierarchical netlist data structure in CADIC contains all informations needed to create all usual exchange formats. The structure of these formats can be classified into the following two groups:

- *cell list*: In the case of a flat netlist description this list only contains the names of the basic cells in the design. Within a hierarchical format this list contains the macros and basic cells for each hierarchy level.

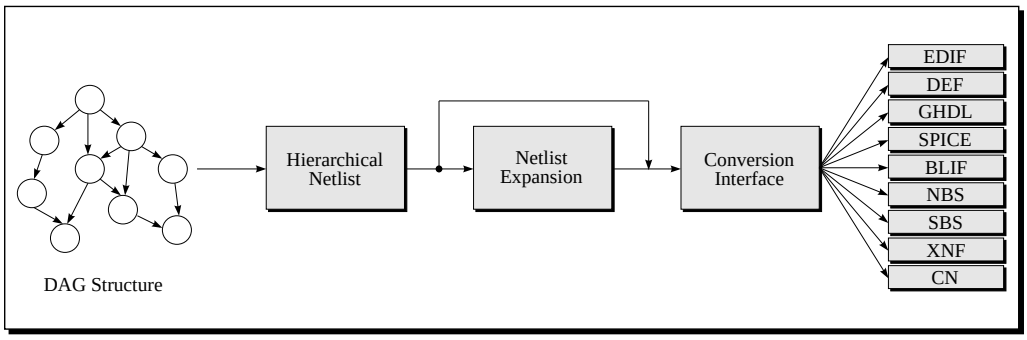


Figure 11.1: Components for generating exchange formats

- *logical connection*: The description of the connections between the cells bases on one of the following schemes:
  - *net oriented netlist*: The structure of the logical interconnections is described by a list of signal nets. For each net of this list it is denoted which pins of cells it combines.
  - *cell oriented netlist*: In this case the structure is given by a list of macros and basic cells. For each element of this list, the identifiers of the corresponding signal nets at its pins are denoted.

Beside the functional description of a design some exchange formats also support the specification of structural informations. For example this can be used to describe clusters of cells, which have to be placed in the same area. In our case this could be used to pack one hierarchy level in one cluster.

11.2 Supported Formats

As you can see from figure 11.1 CADIC currently supports the following exchange formats:

- *EDIF 2 0 0*: The exchange format EDIF (Electronic Design Interchange Format,[Com87]), developped from the Electronic Industries Association, meanwhile has established as standard format which is used in a lot of commercial desgin systems (e.g. Cadence Design

System,[CAD92]).

- *DEF*: The design system TANGATE ([SYS90]) uses the format DEF (Design Exchange Format) which has been defined by TANGENT SYSTEMS for the exchange of design data between the different tools in TANGATE.
- *GHDL*: The system HILO from GenRad ([Gen90]) contains the tools HISIM, HIFAULT and HITIME for different simulation models of integrated circuits. The input of a design to this system is done in form of a GHDL (GenRad Hardware Description Language) file, which is hierarchically organized.
- *SPICE*: The SPICE format has been developped for the circuit simulation program SPICE from the University of California, Berkeley.
- *BLIF*: The format BLIF is the input format for the logic synthesis tools in the OCTTOOLS package which has been developped by the University of California, Berkeley.
- *NBS*: The format NBS and the modified format NBS(f) are used within the layout system HULDA ([**?**]) from the Humboldt University of Berlin.
- *SBS*: With SBS (Strukturbeschreibungssprache) the graphical editor can be used as frontend for the design system VENUS from SIEMENS. It also can be used to address the logic simulator SMILE. In order to use this format the design has to be developped with the basic cell libraries ACMOS3 or ACMOS4. The ACMOS4 library is delivered with the basic CADIC distribution.
- *XNF*: The format XNF (Xilinx Netlist Format) is the netlist format of the XACT system which is a popular FPGA (Field Programmable Gate Arrays) development system from the Xilinx Company. With this conversion interface, a CADIC design can be transformed to the XACT system in order to realize the design with the FPGA technology.

- *CN*: The format CN is an alternative to SBS in order to give a CADIC design to the VENUS system. This format is used by the graphical editor SIGRED and from the layout subsystem in VENUS.

For a documentation of the structure of these different formats you should look at the corresponding manuals. For the following the syntax of the formats is not important.

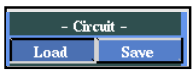
11.3 Design Conversion

In order to activate the design conversion, you should select the entry `Netlist Formats` from the submenu `-Converter-` within the main menu of CADIC . Then a new submenu, named `Converter` is displayed, and you are in the mode of the function selection.

The conversion routines create files for the different exchange formats. Theses files are placed in directories given by some environment variables. For example the files created by the EDIF converter are located in a directory which is given by the environment variable `EDIFDIR`. The formats with the corresponding enviroment variables are listed below. Note, that you must have write permission for these directories. If you do not want to distinguish between the different formats you can set all the variables to the same directory as it is done in the example shell scripts in the CADIC distribution.

⇝ 3

Load Circuit

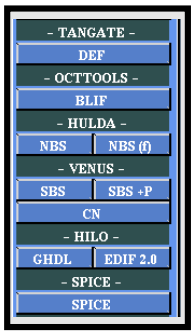


If you want to convert the internal representation of a CADIC circuit into other formats, you must first load its DAG data structure. As shown in the previous chapters this can be done with the entry `Load` from the submenu `-Circuit-`.

⇝ 5.2.1

In this section we will load the 16 bit conditional sum adder and transform it to various exchange formats. To load this adder you should select the entry `CSA[n]` from the circuit list window and type in `16` as the parameter value in the following input window.

Convert Design



In the `Converter` menu the netlist formats are placed under the corresponding name of the design system. For example, the format `DEF` is placed under the system name `-TANGATE-`. The format `EDIF 2 0 0` is placed under the system name `-HILO-`, because EDIF can also be used as input for the simulation package HILO.

After loading the hierarchy of a circuit you can now choose the desired exchange format. Then simply press the push button of the required netlist format. After calling the corresponding conversion routine the system will display a list of informations in the message window. The type of messages depends on the selected format.

For example we will now convert the loaded 16 bit conditional sum adder into the DEF format for the TANGATE design system. To do this select the entry `DEF` from the `-TANGATE-` submenu. In the message window you will see the following informations:

```
start of generating netlist for CADENCE
READY: netlist for CADENCE has been generated
```

The file of the generated format is put in the directory given by the environment variable `CADENCEDIR`. The file has the name `CSA0160.def`, i.e. the suffix of the created file corresponds to the exchange format. In some cases the square brackets and the commas in the parameter list of the circuit are replaced by other characters. In this case of the DEF format they are replaced by `0`. This replacement is done for the formats DEF, BLIF, NBS and SPICE. For GHDL the square brackets and commas are replaced by `_`.

The suffixes and their corresponding format names are grouped in the following way:

- DEF: `<name>.def` in `CADENCEDIR`
- BLIF: `<name>.netblif` in `EDIFDIR`
- NBS: `<name>.nbs` in `NBSDIR`

- NBS(f): `<name>.nbf` in `NBSDIR`
- GHDL: `<name>.cct` in `HILODIR`
- EDIF 2.0: `<name>.edif` in `EDIFDIR`
- SPICE: `<name>.cir` in `EDIFDIR`
- XNF: `<name>.xnf` in `EDIFDIR`

Here we will continue the example of the 16 bit conditional sum adder. We will now convert the DAG structure into an EDIF file. To do this select the entry `EDIF 2.0` from the submenu `-HILO-`. After activating the push button the system will display the following informations:

```
gen_edif:  Start Generation for CSA[16]
( header basiccells macrocells )
Done.
```

For small circuits as this example the generation of the exchange formats is done immediately. If you load a larger design (e.g. 256 bit conditional sum adder) this process will take more time and you can see the different sections of the generation process being displayed in the message window.

For the EDIF format the created file is in the directory `EDIFDIR`. It has the name `CSA[16].edif`, i.e. in the case of EDIF the square brackets and the commas in the parameter list are not replaced by other characters.

Changing the View

With the help of the functions from the submenu `-Views-` you can change the graphical representation of the schematic. The functions in the menu of the netlist conversion tool are identically to those in the hierarchy menu. For an explanation you can look at section 5.9.

To terminate the conversion menu you have to terminate any currently active function first. In most cases this can be done by pressing the right mouse button at most twice within the workarea (see also the description of the appropriate function). After that you are in menu selection mode

and you can close the conversion menu by pressing the right mouse button within the menuline. Now you return to the main menu of the CADIC system. From there you can select another tool or you can terminate the whole system call by selecting the entry `Exit` in the main menu.

12

Editor Reference

12.1 Basic Structures

In this section we use syntax diagrams to define the structure of user input as for example schematic names, parameters of macro cells, width of wires etc. We will show the corresponding syntax diagrams in the different sections. There are some basic syntactical structures, on which these syntax diagrams are based. We will describe these constructs in the following introduction.

The following basic symbols of the underlying programming language are defined:

`A ... Z, a ... z`	letter
`0 1 2 3 4 5 6 7 8 9`	digit
`+ - * / ^ %`	arithmetical operators
`< <= >= > == !=`	compare operators
`( )`	brackets in expressions
`[ ]`	brackets for parameter lists
`,`	split symbol in parameter lists
`@`	introduction of a wire variable

The following standard functions are defined:

`log`	logarithm to base 2
`upper`	next larger integer
`lower`	next smaller integer
`sqrt`	square root
`max`	maximum of two numbers
`min`	minimum of two numbers

With these basic elements we define the following structures which are essential for the syntax diagrams in the next few paragraphs.

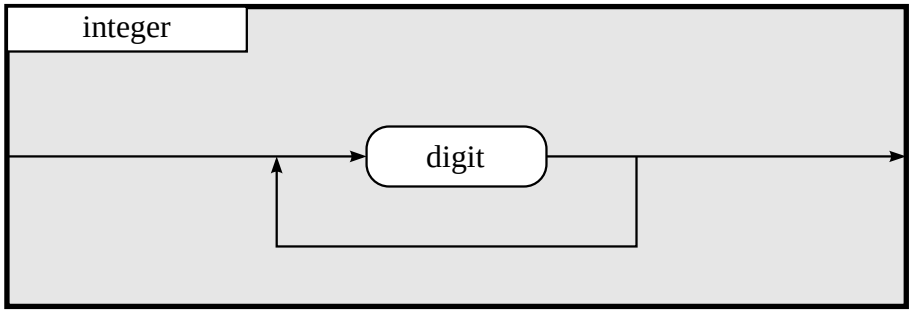


Figure 12.1: Syntactical structure of an unsigned integer

The diagram in figure 12.1 specifies the structure of an unsigned integer. From this diagram follows that an unsigned integer may be an arbitrary sequence of digits, but on a real machine an integer n must be in the range

`0 <=` n `<= 2147483648`

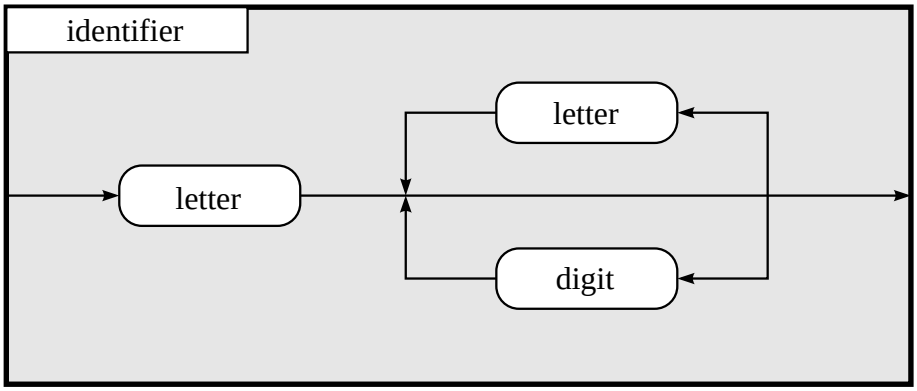


Figure 12.2: Syntactical structure of identifiers

A second important structure is the term identifier for which the diagram

is given in figure 12.2. An identifier may be an arbitrary sequence of letters and digits preceeded by a letter. Theoretically an identifier may have any length but in most cases the system will induce certain restrictions. For example the length of a schematic name must not exceed 80 characters. There are some other inputs where we have other restrictions to the length.

In the diagrams 12.1 and 12.2 the rounded elements represent elementary objects. In the following syntax diagrams there will be elements specified by rectangles. They represent objects which themselves are given by syntax diagrams.

12.2 Schematics

12.2.1 Load Schematic

This menupoint is used to load an existing schematic as well as to create a new one. If you select the entry `Load` from the `-Schematics-` submenu a list window will popup in the workarea. This window contains the list of schematics in the directory given by the environment variable `DAGDIR`. The list is in alphabetical order and it contains one special entry at its top which is called `**** New ****`. You must select this entry if you want to create a new schematic sheet, as it is shown in figure 12.3.

After the window is popped up you are in the name selection mode. When you move the pointer near to an entry in the schematic list it will be highlighted. You can notice that the colour of the schematic name changes from white to black and now is displayed on the background of a white bar. Move the pointer to the next entry in the list and you will notice the change of the highlighted item. Of course you cannot watch this behaviour if you have just started a new session of the editor where still no schematics have been created. In this case the only entry in the list is `**** New ****`.

The size of the list window is fixed, i.e. it does not depend on the number of items in `DAGDIR`. If there are more schematics in `DAGDIR` than can be displayed in the list window you can scroll the list up and down. You can

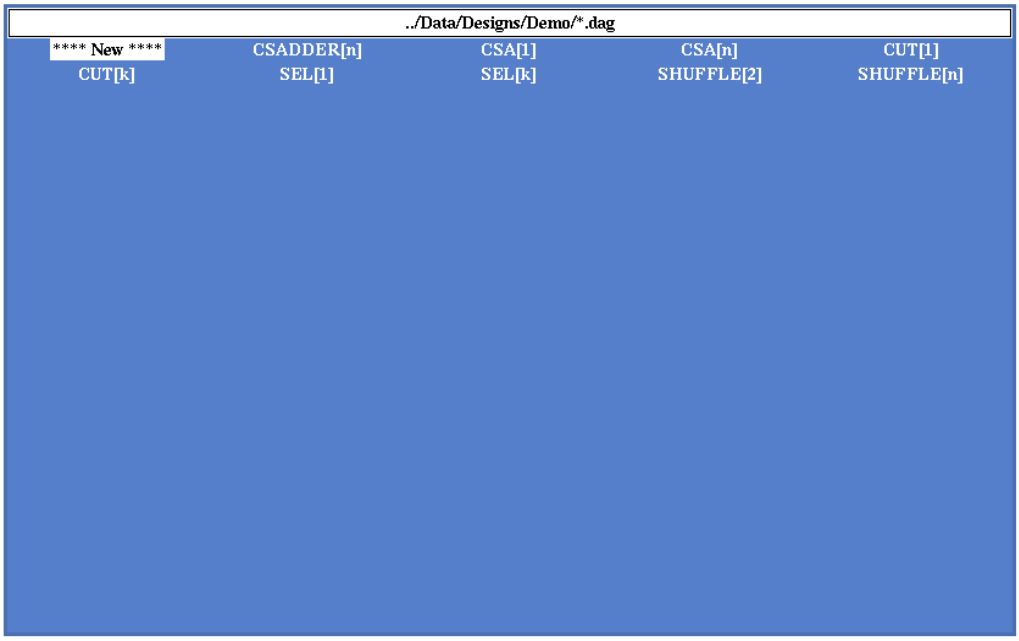


Figure 12.3: List window for schematic names in `DAGDIR` with selected entry `**** New ****`

scroll up the list by pressing the middle mouse button in the lower half of the list window. Note that you can repeatedly scroll up and the list window will be empty if you have scrolled beyond the bottom of the list . In the same way you can scroll down when you press the middle mouse button within the upper half of the list window until you reach the top of the list. The editor keeps the current scroll position in mind, so that further schematic selections are based on the same list offset. If you do not see the `**** New ****` entry the list is scrolled up, move the pointer in the upper half of the window and press the middle mouse button repeatedly until the entry appears in the window.

If you select the entry `**** New ****` an input window is popped up in the workarea with the following prompt (c.f. figure 12.4)

```
Please enter new Schematic Name:  <          >
```

The pointer has automatically been moved into this window and you can type in the name for a new schematic to be created. The distance between

the angle brackets < > indicates the possible length of a schematic name. This name must not be longer than 80 characters and the input routine does not allow you to type in more than this number.

Figure 12.4: Input window for the specification of a new schematic name

When you type in a schematic name you have to follow the syntactical rules, i.e. not every arbitrary sequence of characters is a correct schematic name. The following diagram in figure 12.5 shows the syntax of schematic names.

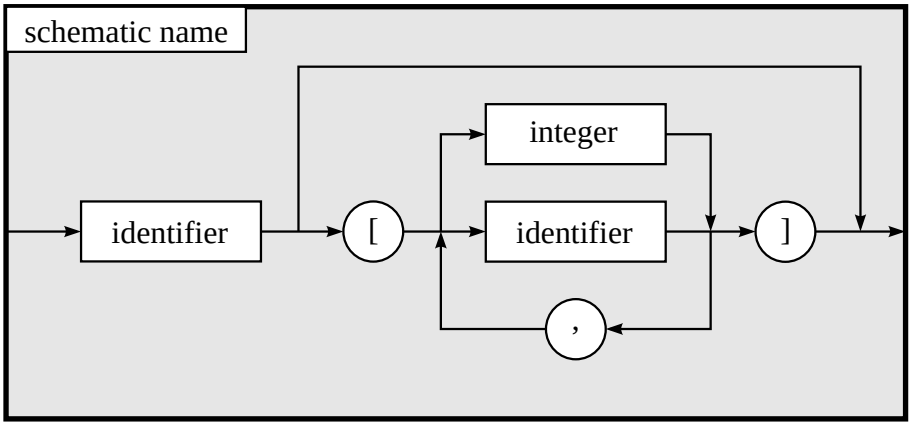


Figure 12.5: Syntax diagram for schematic names

A schematic name is given by an identifier and an optional list of parameters. This list of parameters may contain identifiers as well as integers. The diagram implies that you may use an infinite list of parameters, but in the editor you may only use up to 32 parameters. The parameters of the type identifier are called free, because they may be substituted by any non negative natural number. The other parameters are called fixed. All free parameters of a schematic may be used to create expressions for parameterized macro cells or the width of wires. This will be explained in detail in section 12.3.1.

Some examples for legal schematic names are the following:

```
CSADDER[n]
SEL[1]
NGrid[n,d1,d2,d3]
Matrix[rows,columns]
```

Illegal schematic names are the following:

`SHUFFLE[n`	parameter list not closed
`1CUT[k]`	name is not an identifier
`Matrix[-1,0]`	first parameter is not a non negative integer
`NGrid[,d1,d2]`	parameter list must not begin with a comma
`CSA[n/2]`	parameter must not be an arithmetical expression

If you type in a schematic name and press the return key the name will be analyzed by a parser. In the case that the name is syntactically not correct the system will display an error message in the message window below the workarea:

```
!  Error !  New Name "SHUFFLE[n" is syntactical incorrect
```

The input window will popup again, so that you can retry to enter the schematic name. If you made a mistake and do not want to enter a schematic name, you can cancel the input routine by simply pressing the return key to the empty input window.

If the name of the schematic is syntactically correct, the system will check, whether the new name is a redefinition of an existing schematic. You must not define two general equations with the same schematic identifier. If you have already defined a schematic `CSA[n]`, you are not allowed to define a second general equation of the form `CSA[k]`. In this case the system will respond with the error message

```
!  Error !  New Name "CSA[k]" is Redefinition of CSA[n]
```

If you want to redefine a schematic with new parameters, you have to delete all conflicting schematics .

⇝ 12.2.5

If there is no conflict with other schematic definitions, the input window will disappear and the system will display the message

```
New Schematic "CSA[n]" opened
```

in the message window below the workarea. Note: If you have selected the entry `**** New ****` from the schematic list, but typed in the name of an existing schematic, the corresponding drawing will be shown in the workarea (you not really opened a new schematic in this case).

The name of the schematic is displayed in the upper right corner of the graphical surface in one of the panels of the environment window. In the workarea you will see an empty white frame which represents the border of the schematic. Within this frame you have to place all elements of the schematic like macros, basic cells, wires etc.

12.2.2 Save Schematic

If you select the entry `Save` from the `-Schematics-` submenu the current schematic is saved to a file in the directory given by the environment variable `DAGDIR`. The file is named

```
<schematicname>.dag
```

The file contains the elements needed for the graphical representation of the schematic. These are position, orientation and names of the basic cells and macros, the wires, the comment texts and the additional equations. Each schematic is represented by a graph data structure, i.e. the borders of the schematic or the macros as well as the wires are the edges of this graph. The corners of the schematic and the cells, the wire points and the connections of wires to the cells and the border of the schematic are the nodes of this graph.

After the schematic has been saved to `DAGDIR` a macro cell will be generated, if the schematic has no or only fixed parameters. Such a macro is called a discrete macro, because its elements do not depend on any free parameters. The discrete macros can be selected as cells (menu entry `Enter` from `-Cells-` submenu) from the second cell list window.

Note: you need not to create schematics only with the editor. You can write programs yourself to create correct files in `DAGDIR`. This can be done by using the functions from the basic library for the graph data structure.

12.2.3 Save Schematic under New Name

You can save the current schematic under a new name with the entry `Save As` from the `-Schematics-` submenu. After selecting this point the input window will popup and you can type in a new name for the current schematic at the following prompt:

```
Please enter new Schematic-Name:<        >
```

For the new schematic name you have to consider the syntactical rules
↝ 12.2.1 defined by diagram 12.5 from section 12.2.1. If you have typed in a name which is not syntactically correct, the input window will popup again and in the message window you can read a message like

```
!  Error !  New Name "SHUFFLE[n" is syntactically incorrect
```

You can enter a new name for the schematic or abort the save function by pressing the return key to an empty input window.

↝ 12.2.2 When the schematic is saved to `DAGDIR` a macro cell with the name of the new schematic will be generated in the case of discrete macros.

12.2.4 Clear Schematic

With the help of the function `Clear` from the `-Schematics-` submenu you can erase all elements (macros, wires, comments and equations) within a schematic. It clears your workarea and leaves the empty schematic frame. If you have made mistakes during the drawing this function is a fast method to erase all elements.

This function gives an alternative method to create a new schematic. You can load an existing schematic, clear it and save it under a new name. The you have to load the newly created schematic in order not to overwrite the currently loaded schematic.

Note: there is no security check, when you call this function. If you called this function and did not want to clear the schematic, just load the schematic again without saving it. The old drawing will be restored.

12.2.5 Delete Schematic

With the function `Delete` from the `-Schematic-` submenu you can delete a schematic file in `DAGDIR`. After the selection of this entry the schematic list window will popup and you can select the name of a schematic. Note that the `**** New ****` entry does not appear in the list and the order of the schematic names is shifted by one to the left. If you press the left mouse button the current highlighted schematic is deleted. For the corresponding file in `DAGDIR` the command

```
> rm -f <schematicname>.dag
```

is executed. The deletion of the file can be noticed in the change of the schematic list where the name of the just deleted schematic will disappear. Then you can select another schematic for deletion or you can abort the function by pressing the right mouse button within the list window.

Note: be careful with this function. In the current implementation there is no security check which asks you, whether you really want to delete the schematic.

12.3 Cells and Macros

12.3.1 Enter Cells

With the function `Enter` from the `-Cells-` submenu you can select cells and macros and place them within the schematic frame. The cells are divided into the following three groups according to their functionality:

- *Basic cells* are elementary components which compute basic operations. The user can create a basic cell library with the help of a cell editor tool. With this editor it is very easy to make a cell library of

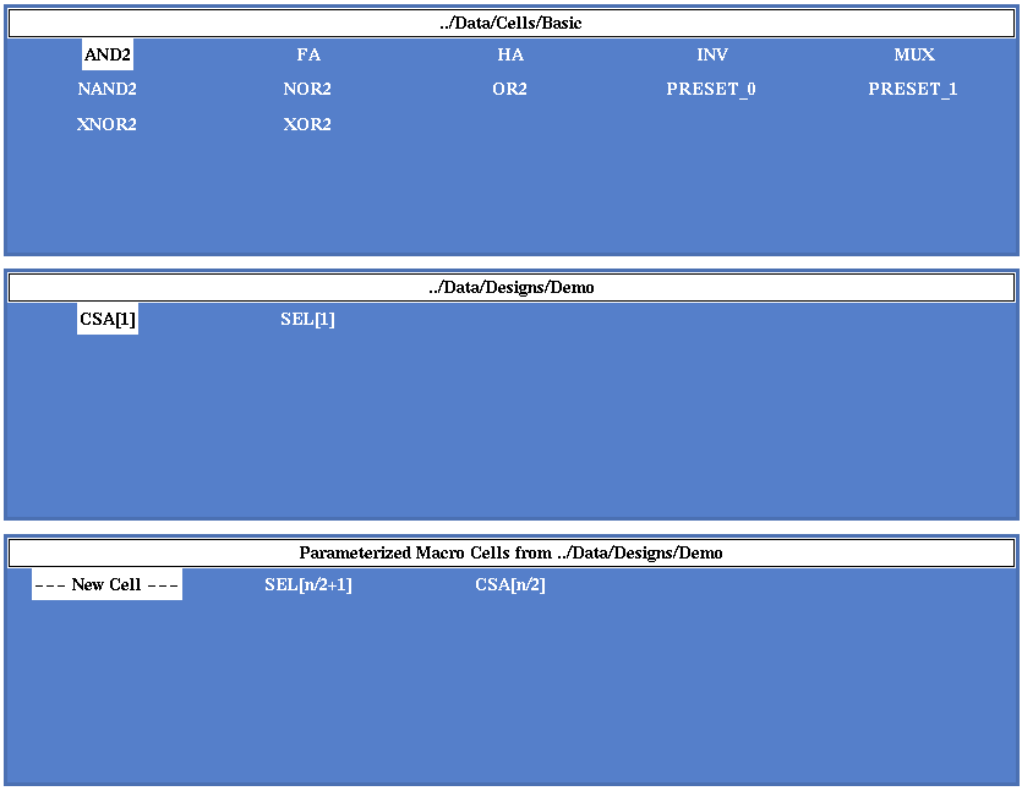


Figure 12.6: Cell selection windows for basic cells, discrete and parameterized macros

commercial design systems available for the CADIC system. Within the current distribution of the CADIC system there are some basic cell libraries of commercial design system contained.

- *Discrete macro cells* are components which are defined by the graphical input of a corresponding schematic. Discrete means that this macro does not depend on free parameters, i.e. the schematic name only contains integer parameters or it has no parameter at all. With the use of such a macro the designer creates a new hierarchy level in the design specification. The wires connected to the border of this schematic will appear as the pins of the graphical macro representation.

- *Parameterized macro cells* are components, which depend on different parameters of the current schematic. These macros can be used espe-

cially within recursive specifications, where you use an instance of the currently specified schematic. Parameterized macro cells can be compared with forward declarations known from programming languages.

Each of these cell groups is displayed in a separate window which is popped up upon the workarea (c.f. figure 12.6). The windows are arranged according to the listing above. In the upmost window the names of the available cells from the basic cell library are displayed, the second window contains the names of the defined discrete macro cells and in the third window the parameterized macro cells are listed, which have been defined during the current editor session.

If you move the pointer into one of the three windows near to the name of a cell or macro, the corresponding entry will be highlighted. Now you can select this item by pressing the left mouse button. After this a graphical representation of the cell or macro is displayed within the workarea. This representation depends on the type of the cell.

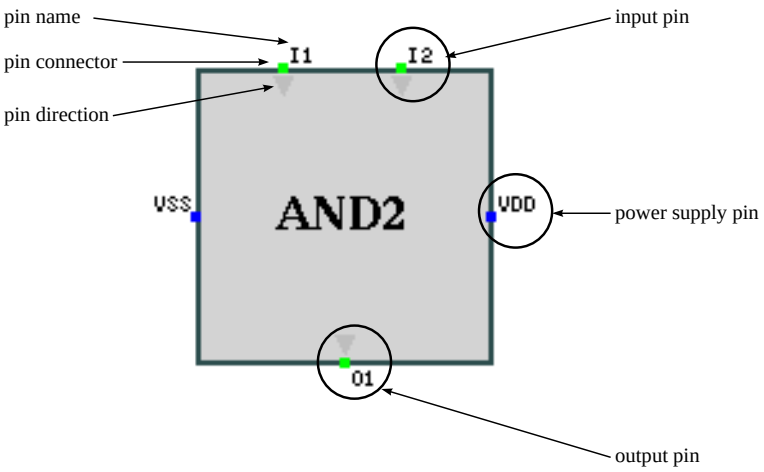


Figure 12.7: Graphical representation of an and gate with two inputs selected from the basic cell library

Figure 12.7 shows the typical layout of a basic cell. The displayed information is called the interface of the cell. It contains the size of the cell, the names and positions and directions of its pins. As shown in figure 12.7 we distinguish between signal and power supply pins. During an editor ses-

sion normallay only the signal pins are used and connected to wires. The power supply pins are left open, because the supply nets will be generated automatically by an appropriate tool (c.f. chapter 8).

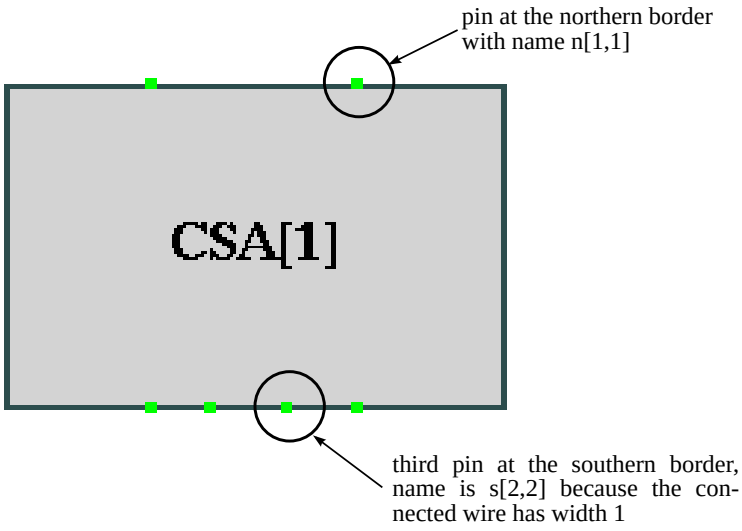


Figure 12.8: Graphical representation of a discrete macro cell in the case of a 1 bit conditional sum adder

The graphical representation of a discrete macro cell will be derived from
⇝ 12.2.2
the corresponding schematic. When the schematic is saved an entry in the discrete macro list will be generated. The position of the pins is given by the relative positions of the connections of the corresponding wires with the border of the schematic. The pin names are generated automatically. They consist of the side of the schematic to which the pin belongs and an interval which describes the position and the width of the pin. The width of the pin is given by the difference between the upper and lower bound of the interval plus one. The order of the intervals is from left to right at the northern and the southern border and from top to bottom at the western and eastern border of the macro. The order of the intervals is important in the case of rotations of the macro. Examples for pin names are the following:

`n[0,0]` first pin at the northern border of a macro, the connected wire has the width 1

`e[3,4]` this pin represents the connection of the fourth and fifth wire at the eastern border, which are bundled in a single wire of width 2

`s[1,1]` second pin at the southern border

`w[0,2]` first pin connector at the western border of the macro, representing a bundle of width 3

In the graphical representation of a macro cell, the pin names are not displayed, only the pin connectors are shown. Macro cells normally have no power supply pins, when they are selected from the macro library. Power supply pins are generated by invoking an appropriate tool (c.f. chapter 8).

The third group of cells you can use are parameterized macros, which depend on the free parameters of the schematic. For these macros in general there exists no definition from a basic library or a schematic input. That is why its size and the position of its pins are unknown when it is selected. Therefore it is displayed as a rectangle of default size which has no pin connectors first. Pins are created when you connect wires to the border of the macro which is treated like the border of the schematic. The size of a parameterized macro can be changed after it is placed within the workarea in order to create a more intuitive representation of the schematic.

⇝ 12.4.1

⇝ 12.3.3

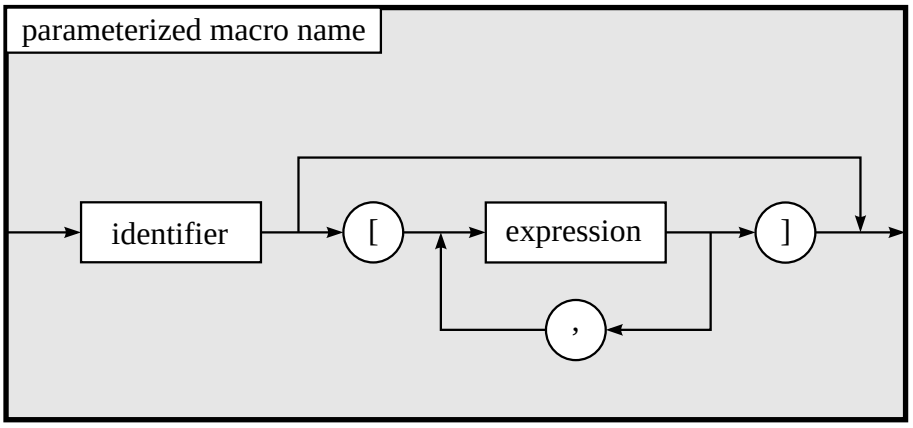


Figure 12.9: Syntax of parameterized macro names

To select a new parameterized macro, move the pointer into the third cell list window near to the entry `--- New Cell ---` and press the left mouse button if it is highlighted. The input window will popup and you are prompted to enter a new name for a parameterized macro cell:

```
Please enter Parameterized Name:  <        >
```

Now you can type in the name of a new parameterized macro cell. As shown above for the names of schematics there are syntactical rules which must be considered when selecting the name. The diagram in figure 12.9 shows the basic structure of a parameterized name.

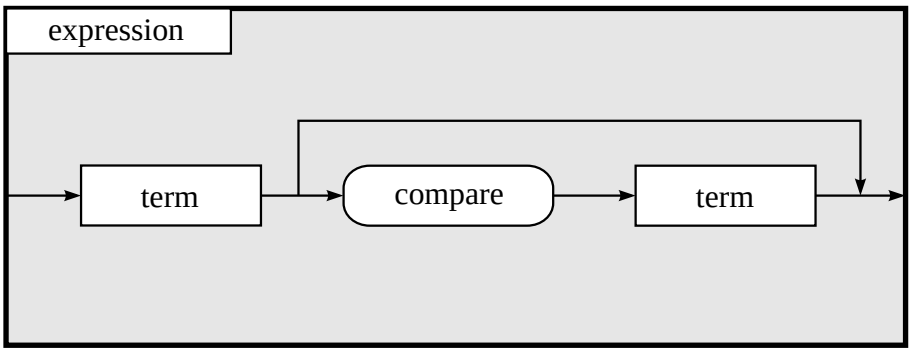


Figure 12.10: An expression consists of a single term or the comparison between two terms

The diagram in figure 12.10 defines that the name of a parameterized macro consists of an identifier followed by an optional list of parameters, separated by commas. Each parameter is given by a construct named expression which will be refined in the following. The diagram implies that a macro may have an infinite number of parameters, but as for schematic names, there is the limitation to a maximum of 32 parameters.

Each parameter of the type expression is defined by the following diagrams. We use the hierarchical description known from programming languages in order to describe the priorities between the available operators. In a first step an expression is refined according to the diagram in figure 12.10, where we use a construct named term. An expression may be a single term, or it may be the comparison of two terms. As shown in section 12.1 a comparison

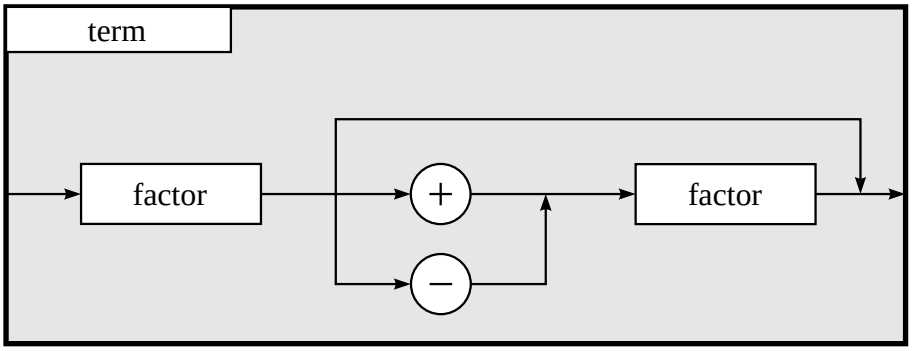


Figure 12.11: Syntactical structure of term which is the additive combination of two factors

operator may be one of `< <= >= > == !=`. The comparison between two terms is evaluated to zero or one, if it is false or true.

The parts of an expression are described by the construct term, for which the syntax diagram is given in figure 12.11. A term represents a single factor or the combination of two factors with the use of an additive operator (`+` or `-`). We have chosen this representation to express the higher priority of the elements in factor with regard to addition and subtraction.

A factor is defined by the syntax diagram in figure 12.12. It combines two objects of the type unit by using a multiplicative operator like multiplication (`*`), division (`/`), power (`^`) and division modulo (`%`). The use of a new object of type unit introduces a new priority level for this construct.

The object with the highest priority is called unit. Its syntactical structure is given by the diagram in figure 12.13. Beside the call of a standard function as they are listed in section 12.1 a unit can be an identifier or an unsigned integer. In the diagram any identifier is allowed but within a schematic you may only use the identifiers which appear as parameters in the schematic name.

⇝ 12.1

Legal names for parameterized macros are the following:

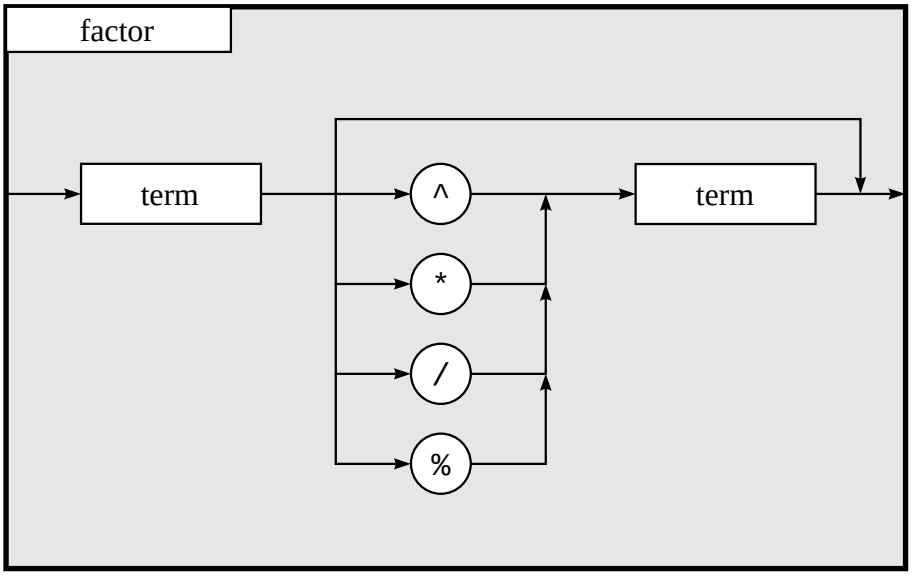


Figure 12.12: Syntactical structure of factor which is the multiplicative combination of two units

```
CSA[n/2]
SEL[upper(k/2)]
GChannel[n,min(n-2^i,2^i),i<(log(n)-1)]
```

After the selection of a cell from one of the three cell list windows, the corresponding graphical representation of the cell type is shown within the workarea. The cell follows the movement of the pointer in order to choose the position where you want to place it. During the motion you may notice that the cell changes its colour from red to grey. This indicates that you have selected an illegal position for the placement.

Illegal positions are overlappings of the current cell with other cells, with the border of the schematic or with wires. If the cell is on an illegal position you can not place it by pressing the left mouse button. This will have no effect and the cell will still be movable with the pointer. If you no possibility to place the cell you can abort placement mode by pressing the right mouse button within the workarea.

If you have selected a legal position for the cell and dropped it or if you aborted the cell placement mode, the three cell list windows are popped up

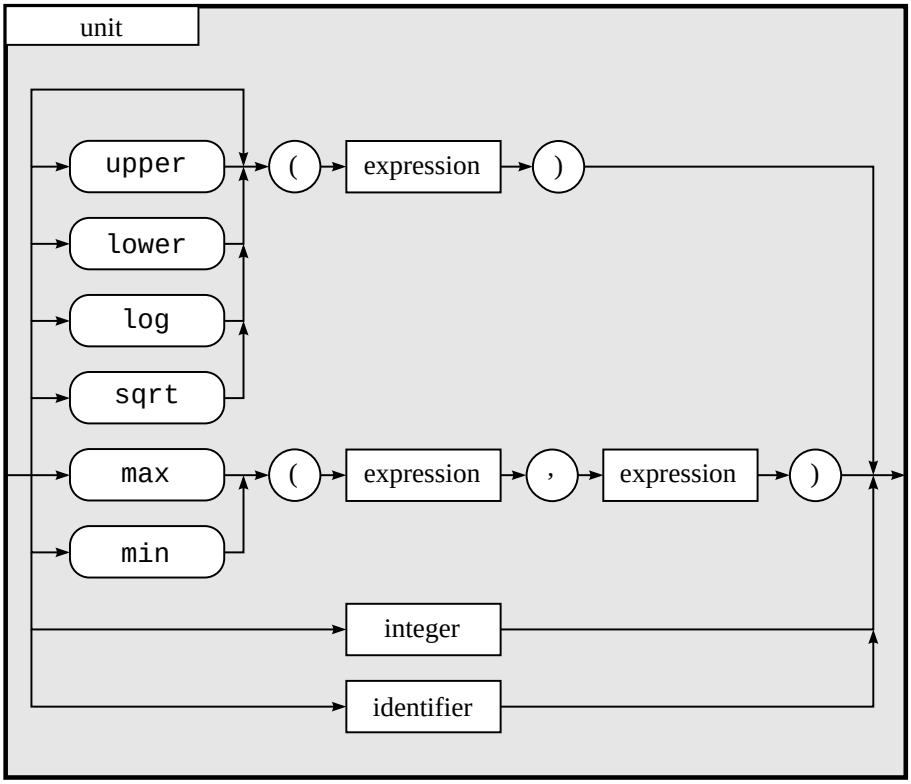


Figure 12.13: Syntactical structure of an object of type unit

again in order to select another cell. You can abort this mode by pressing the right mouse button within one of the three list windows. This operation terminates the enter cell mode and you return to menu selection mode.

12.3.2 Move and Rotate Cells

With the function `Move` from the `-Cells-` submenu you can select placed cells and macros within the workarea and move them to a new position or change their orientation. If you select this entry from the menu and move the pointer into the workarea you are in cell selection mode. This is indicated by the nearest cell to the pointer being highlighted in blue. If you press the left mouse button this cell will be selected.

Note: After the selection of a cell for movement or rotation, it is isolated from its context within the schematic. The wires are no longer connected to the cell. We recommend that you delete all wires, which are connected

⇝ 12.4.2

to the cell before you move it to a new position. But in most cases you will select cells from the cell list windows, place them within the schematic and change their positions or orientations before you enter the signal wires.

After the selection the instance is in cell placement mode and it can be moved freely within the workarea as we have shown it above in section 12.3.1 when we entered a new cell. The movable cell changes its colour, if you select an illegal position, where it intersects another cell, the border of the schematic or a wire. It is impossible to place it at an illegal position, the left mouse button has no effect in this case. If the cell is at a legal position, you can fix it there by pressing the left mouse button. You also can abort placement mode by pressing the right mouse button. In this the case the movable cell is returned to its old position and cell selection mode is activated again in order to select another cell for movement operation.

⇝ 12.3.1

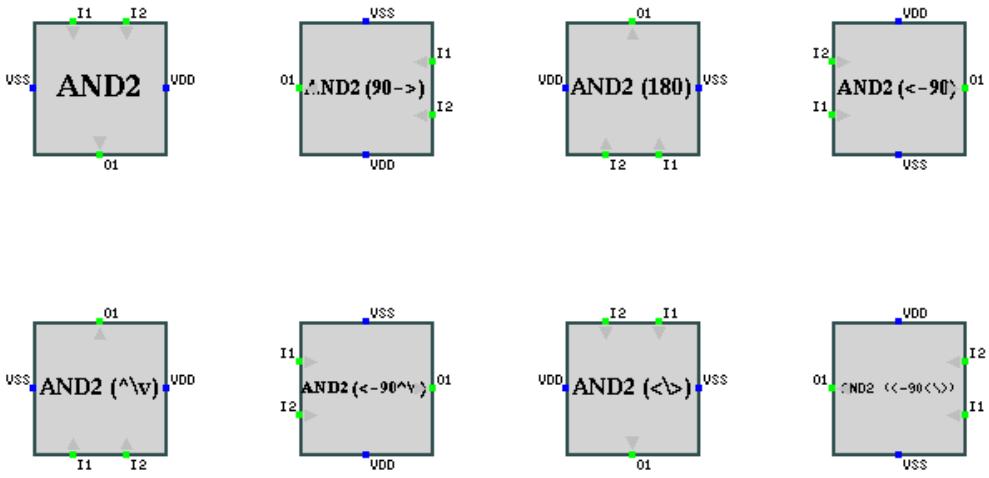


Figure 12.14: Sequence of possible orientations of a basic cell

With the help of this menu entry you can not only select cells and move them to a new position, you also can change their orientation. After the selection of a cell it is in placement mode. During the movement you can select a new orientation for it by pressing the middle mouse button several times. Each button press event changes the orientation of the cell to one of the eight possible representations shown in figure 12.14.

The current orientation is indicated by a suffix in the cell name. The description of these suffixes is given in figure 12.14. It is not a part of the cell name. You can notice this, when you rename a cell, which is not in normal orientation. In this case the cell with the new name has the same orientation as before and keeps its suffix.

⇝ 12.3.5

If you have chosen the desired orientation, you can move the cell to a new location and fix it there by pressing the left mouse button. If you do not want to change the orientation of the selected cell, you can abort the operation by pressing the right mouse button. In this case the cell is moved to its original position and orientation. You are in cell selection mode and you can choose another cell for movement or rotation.

You can terminate this operation by pressing the right mouse button within the workarea, if you are in cell selection mode. If you are in cell placement mode, i.e. a cell is following the motion of the pointer, you must abort this first (right mouse button).

12.3.3 Resize Cells

With the function `Resize` from the `-Cells-` submenu you can select placed cells or macros within the schematic and change their sizes. In most cases this function is used to resize parameterized macros, for which the system selects the default size. Changing the size of these macros is useful for creating an intuitive representation of a schematic. You can select appropriate sizes for the macros in order to indicate their different priorities.

If you activate the resize operation in the menu, you are in cell selection mode. Move the pointer into the workarea and choose the cell which you want to resize (nearest cell to the pointer is highlighted in blue) by pressing the left mouse button.

Now you are in resize mode which is indicated by a rubberbanding frame following the motion of the pointer. This frame is connected at the upper left corner of the selected cell. Its opposite corner is at the position of the pointer indicating the new size of the cell. You can confirm the current size

by pressing the left mouse button or you can abort the resize mode with the right mouse button.

If you have selected a new size for the macro, it is displayed now according to your selection. The resized macro is now in the cell placement mode and you have to choose a new position for it. This is necessary because the resizing of the macro could lead to overlappings with other cells, wires or the schematic border. Move the cell to a legal position (illegal positions are indicated by a change of the colour of the cell from red to grey) and place it there by pressing the left mouse button. If you abort cell placement mode, the cell is returned to its old position and its original size is restored.

After that you are in the cell selection mode again and can choose another cell for resize operation or you can terminate the resize operation by pressing the right mouse button within the workarea.

Note: As in the case of moving and rotating cell we recommend that you delete all wires, which are connected to the cell, before you select it for resizing.

12.3.4 Copy Cells

With the function `Copy` from the `-Cells-` submenu you can create a copy of an already placed cell or macro. This function is very useful, if you have changed the attributes of a cell (size or orientation) and you need a new cell with the same attributes. Without this function you would have to enter a new macro from the cell list windows, place it within the schematic and change the attributes.

⇝ 12.3.2

This function is very similar to the move operation from section 12.3.2. If you have activated the copy operation, you are in cell selection mode and can choose the macro, from which you want to create a copy.

After the selection with the left mouse button, a second instance of this cell is created. Then you are in the cell placement mode and can move the new macro to the desired position (confirm it by pressing the left mouse button). You can abort cell placement mode with the right mouse button. In this

case the newly created cell will disappear.

After that you are in cell selection mode again and you can select another cell to be copied. If you do not want to copy more macros, you can abort this function by pressing the right mouse button within the workarea.

12.3.5 Rename Cells

With the function `Rename` from the `-Cells-` submenu you can change the names of macros. This function is useful, if you have made a copy of an instance and you want to change its parameters. The size and the orientation are not modified by this operation.

If you activate this function, you are in cell selection mode in order to choose the cell, for which you want to change the name. Move the pointer near to the desired macro and press the left mouse button to select it.

Note: this function is only applicable with macros, not with basic cells. If the currently highlighted instance is a basic cell, pressing the left mouse button has no effect. You will remain in cell selection mode. You can not simply exchange the names of basic cells, because they have a fixed interface which is given by the descriptions in the selected library. This information is global to all users of this library.

After the selection of a macro cell, the input window will popup and ask you for the new name of the macro:

```
Please enter New Instance Name:<        >
```

Type in the new macro name and confirm your input by pressing the return key. Note that you have to consider the syntactical rules for macro names as they are described in section 12.1 and 12.3.1. If you typed in a syntactically wrong name, the name of the selected macro will not be changed and an error message is displayed in the message window:

⇝ 12.1,12.3.1

```
!  Error !  "SHUFFLE[n" is illegal Instance-Name
```

In this case the input window will popup again, and you can correct your

input. If you do not want to change the name of the selected macro any longer, you can cancel the input window by pressing the return key to an empty input line.

If your input is correct, the name of the selected macro is changed in this way. In both cases (cancelling the input window or choosing a correct name) the input window is popped down and you return to cell selection mode in order to select another macro, for which you want to change the name. You can abort this mode and thereby abort the whole rename function by pressing the right mouse button within the workarea.

12.3.6 Delete Cells

With the function `Delete` from the `-Cells-` submenu you can erase cells and macros from the current schematic. After activating this function you are in cell selection mode. Move the pointer into the workarea near to the cell you want to delete and press the left mouse button, when it is highlighted.

Note: only the cell or macro is deleted. The connected wires remain in the schematic. If you do not need them, call the function `Delete` from the `-Wires-` submenu .

⇝ 12.4.2

After the deletion of a cell you are in cell selection mode in order to delete more cells. You can abort this mode and the delete function by pressing the right mouse button within the workarea.

Note: the function is automatically terminated, if you have deleted the last cell in the schematic. Therefore you can not activate this function, if there is no cell or macro in your schematic.

12.4 Wires

12.4.1 Enter Wires

With the function `Enter` from the `-Wires-` submenu you can draw wires in order to connect cells and macros. Wires are build up of sequences of horizontal and vertical edges. The graphical representation of a wire may represent a bundle of single parallel wires. The number of these single wires is described by the width of the graphical wire. Concerning the width the wires can be clssified into three groups:

- wires with constant width
- wires with arithmetical expressions over the schematic parameters
- wires with formal variables

For the use of these wire types we define the following rules:

- at the pin connectors of basic cells and discrete macros you may only connect wires with a constant width. In the case of basic cells the width is always 1.
- at the borders of parameterized macro cells and the schematic you may connect wires with a width of all three types.

If the width of the wire is not equal to 1, it is displayed beside a small slash at the middle of the wire. The following figure 12.15 shows wires of all three types.

The enter wire function has to perform the following tasks

- selection of the start and end point of the wire
- determination of the wire width
- interactive error handling

If you activate the enter wire function, you first have to select the start point for a wire. If you move the pointer into the workarea you notice a crosshair cursor following the pointer motion. Because of the high resolution of the

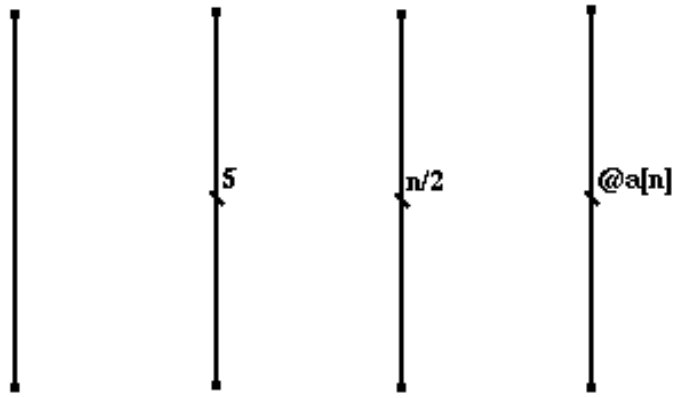


Figure 12.15: Wires with different widths

screen it is difficult for the user to select exactly a certain pixel position. Therefore the crosshair cursor not only specifies a pixel position, but it covers a region of pixels. If you move it near to a special position within the schematic, it will snap to that position. You can notice this behaviour, if you move the pointer slowly near to one of the following special points

- point on a wire (but not its start or end point)
- start or end point of a wire
- pin connector
- point on schematic or parameterized macro border

If the cursor is near to any of these points, the system will choose the coordinates of it. If this is not the case the position is the center of the cursor. The special points have increasing priority in the given order, i.e. a pin connector has higher priority than a point on a wire.

You can fix the start point of the wire by pressing the left mouse button. After that you can move the pointer to the end point of the wire. During the motion of the pointer you will notice a rubberbanding line connecting the start point with the current pointer position. The shape of this line indicates the wires which will be inserted, if you fix the end point. In some cases this

line will disappear. This indicates that you have chosen a connection of two points, which is impossible by insertion of at most two wires (e.g. one wire would intersect a cell). Note that the line will also disappear, if you connect two wires with different widths.

If you have chosen a legal connection (the rubberbanding line is visible), you can fix the end point by pressing the left mouse button again. If you do this for an illegal connection, the selected end point will become the start point for a new connection, i.e. the rubberbanding line will now be drawn from the expected end point to the current pointer position. You also can abort the selection of the end point by pressing the right mouse button and return to the selection of a new start point.

If you have drawn a legal connection between two wire points, the rubberbanding line will disappear and the system will now determine the width of this wire. In some cases this can be automatically derived from the points you have connected:

- if one of the wire points is a pin connector of a basic cell, the wire gets the width 1.
- if one of the wire points is a pin connector of a discrete macro cell, the wire gets the width of this pin.
- if one of the points is located on a wire, the new wire gets the same width as this wire.

In the other cases the system can not automatically determine the width of the new wire. Then the input window is popped up and you have to specify the wire width to the following prompt:

```
Please enter Wire Width:<        >
```

For the selection of the wire width you have to consider the following syntactical rules. As mentioned earlier the width of a wire may depend on the free parameters of the schematic. It may be an arithmetical expression in these parameters or it may be described by a wire variable. The diagram in figure 12.16 shows the syntactical structure of a wire width.

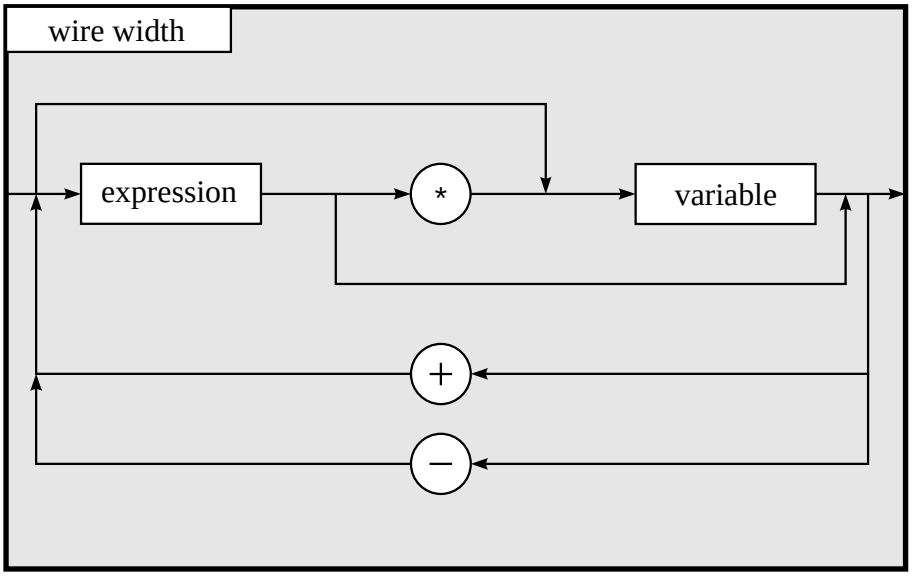


Figure 12.16: Syntactical structure for the width of a wire

This diagram shows that the width of a wire may be a simple expression as it is given by the diagrams in figure 12.10 to figure 12.13. But it also may be a linear combination of variables which have expressions as coefficients. The restriction to a linear combination is necessary, because we compute the values of the wire variables by solving an equation system.

The syntactical structure of a wire variable is given by the diagram in figure 12.17. A variable is introduced by the special character `@` followed by a construct which is similar to a parameterized macro name. Especially a wire variable may depend on parameters which are given by expressions over the schematic parameters. This allows you to specify flexible variables within a recursive description, whereas a variable without parameters would have the same value for all stages of the recursion.

The following inputs are legal wire widths:

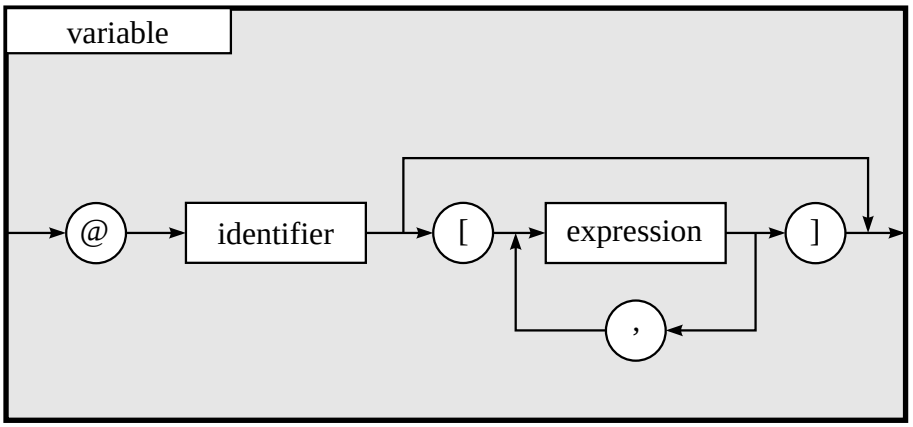


Figure 12.17: Structure of a wire variable

`5`	constant value
`n/2+1`	arithmetical expression
`@t`	constant wire variable
`@s[n]`	variable depending on schematic parameter
`@w[n>0,m==1]`	variable depending on comparisons
`@s[n]+n*@t[n-1]`	linear combination of variables

The following inputs are illegal wire widths:

`-1`	width may not be negative
`@s[@t[n]]`	illegal parameter
`@s[n/2]*@t[n]`	not a linear combination of variables
`0.5*@t[n-1]`	coefficients must be integers

If you enter a correct wire width and confirm this by pressing the return key, the new connection is drawn as one or two yellow edges. If the width is not equal to one, it is drawn as a label in the middle of each edge which is longer than 50 pixels (for very small edges the width is not shown on the screen, but you can look at it, if you zoom in the schematic with the help of the functions in section 12.7.1).

⇝ 12.7.1

After the new wire is created, you can draw another connection. Here the end point of the previous wire is taken as the start point for the next wire. This enables you to draw a sequence of more than one wire segment. If you do not want to use the end point of the previous wire as the start point

for the next wire, press the right mouse button once. Then the crosshair at the end point and the rubberbanding line will disappear and you can select a new start point for the next wire. Especially in some cases it is impossible to use the end point of the previous wire as the new start point (e.g. end point is located on the schematic border), then you will see no rubberbanding line (indicating that there is no legal connection possible). Then the next point selected will be the new start point of the next wire.

If you have entered an illegal wire width, the input window will popup again and you can correct your input. If you press the return key to the empty input window, the just created connection will be deleted.

You can terminate the enter wire function by pressing the right mouse button within the workarea. Note that you have to press it twice, if you are selecting the end point of a wire (rubberbanding line will follow the pointer motion).

12.4.2 Delete Wires

With the function `Delete` from the `-Wires-` submenu you can erase wires from the current schematic. After activating this function you are in the wire selection mode. If you move the pointer into the workarea, the wire nearest to the pointer position is highlighted (it changes its colour from yellow to cyan). Now you can delete this wire by pressing the left mouse button.

After the deletion of the wire you remain in wire selection mode in order to delete more wires. This mode and the delete wire function can be aborted by pressing the right mouse button within the workarea.

Note: the function is automatically terminated, when you have deleted the last wire in the schematic. That is why you can not activate this function, if there is no wire at all in your schematic.

12.5 Comments

12.5.1 Enter Comments

With the function `Enter` from the `-Comments-` submenu you can insert comment text into your schematic. If you activate this function, the input window will popup with the following prompt:

```
Please enter Comment Text:<        >
```

Now you can type a comment text which may contain every printable character. The maximum length of the text may not exceed 80 characters. If you have to enter more text, you must split it in several comment lines.

If you confirm your input by pressing the return key, the text will appear within the workarea. It is now movable with the pointer and can be placed at any position in your schematic (inside cells or macros, across wires etc.). Move the text to the desired position and drop it there by pressing the left mouse button. You can abort the placement operation by pressing the right mouse button (the comment text is deleted in this case).

After that the input window will popup again and you can type in another comment text. If you do not want to place more comment texts you can abort this function by pressing the return key to the empty input window.

12.5.2 Delete Comments

With the function `Delete` from the `-Comments-` submenu you can erase placed comment text from your schematic. If you activate this function you are in comment selection mode. If you move the pointer into the workarea, the comment text nearest to the pointer position is highlighted (its colour changes from white to cyan).

Now you can delete this comment by pressing the left mouse button. After the deletion of a comment you return to comment selection mode in order to delete more comments. You can abort this mode and the delete function by pressing the right mouse button within the workarea.

Note: the function is automatically terminated, if you have deleted the last comment in the schematic. Therefore you can not activate this function, if there is no comment at all in your schematic.

12.6 Equations

12.6.1 Enter Equations

With the function `Enter` from the `-Equations-` submenu you can insert additional equations about variables which are used for the description of
↝ 12.4.1
the width of bundles of wires . In some cases the equations automatically derived by the system are not sufficient to compute a unique solution. In section 5 we describe, how the system generates equations about the wire variables.

Entering an equation is very similar to entering a comment text. If you activate this function, the input window is popped up with the following prompt:

```
Please enter New Equation:<       >
```

and you can type in an equation, for example

```
@a[n] = @b[n]
@s[n]+@t[n] = 2*@t[n-1]
@t[n] = 10
```

The syntactical structure of both sides of the equation is given by the dia-
↝ 12.4.1
gram in figure 12.16 .

If your input is correct, the equation will appear in the workarea and is movable with the pointer. Just as comment text you can place it anywhere in your schematic and fix its position by pressing the left mouse button. Note that equations are drawn in yellow, whereas comment text is displayed in white.

If you typed in a syntactically incorrect equation the system will respond with an appropriate error message in the message window:

```
!!!Error!!!  Equation ...  is syntactically incorrect
```

Then the input window will popup again in order to correct your input. If you do not want to enter another equation, you can abort the function by pressing the return key to the empty input window.

12.6.2 Delete Equations

With the function `Delete` from the `-Equations-` submenu you can erase placed equations from your schematic. If you activate the function, you are in equation selection mode. If you move the pointer into the workarea, the equation nearest to the pointer is highlighted (it changes its colour from yellow to cyan). Note that you can only select equations, i.e. comment text will not be highlighted.

The highlighted equation can now be deleted by pressing the left mouse button. After that you remain in equation selection mode in order to delete more equations. You can abort this mode and the delete equation function by pressing the right mouse button within the workarea.

Note: the function is automatically terminated, when you have deleted the last equation in your schematic. That is why you can not activate it, if there is no equation at all in your schematic.

12.7 Views

12.7.1 Zooming

With the functions `Zoom In` and `Zoom Out` from the `-Views-` submenu you can select a viewport and change the graphical representation of your schematic.

If you activate the function `Zoom In` you are in viewport selection mode. Now move the pointer into the workarea and select a start point for a rectangular viewport. This need not be the upper left corner of the viewport, according to the selection of the second point it will be interpreted appropri-

ately. After fixing the start point by pressing the left mouse button, move the pointer to the position of the opposite corner of the desired viewport. You notice a rubberbanding frame following the motion of the pointer. This frame indicates the size of the selected viewport and implies the factors by which the schematic will be scaled.

If you confirm the selection of the opposite corner by pressing the left mouse button, the schematic will be scaled and redrawn. The upper left corner of your selected viewport will be moved into the upper left corner of the workarea. You remain in viewport selection mode for further zooming operations. Each zooming operation is pushed onto a zooming stack, so that we can return to any previously selected viewport. The currently active viewport will be shown in the control window as a small yellow rectangle, representing the position and the size of the viewport relative to the size of the whole schematic.

The selection of the corners of a viewport can be aborted by pressing the right mouse button. If you do this during the selection of the first corner, the zooming function is terminated and you return to menu selection mode. If you press the right mouse button during the selection of the opposite corner, the start point is cancelled and you can select a new first point of the viewport.

The inverse function `Zoom Out` can be used to restore the viewport on the top of the zooming stack. This function has no effect, if the zooming stack is empty. If you want to restore the original size of the schematic, you can directly call the function `Normal` which clears the zooming stack in one step.

12.7.2 Scaling

With the function `+10%`, `-10%`, `+50%` and `-50%` from the `-Views-` submenu you can change the size of the schematic without moving its upper left corner to a new position. The scaling factors are 1.1, 0.9, 1.5 and 0.5 respectively.

After calling these functions for several times you can return to the original size of the schematic with the help of the function `Normal`.

12.8 Miscellaneous

With the function `All Objs` from the `-Views-` submenu you get a representation of your schematic which will intuitively show the relation to the underlying mathematical calculus. The graphical represenation resembles an equation, where the left side is the macro representation of the current schematic and the right side is the schematic itself. This denotes that the macro is refined (refinement operator is indicated by an arrow) by the drawing of the schematic.

We do not recommend to use this representation during the input of the schematic, because the frame is scaled down, so that there is not enough space for comfortable input operations. You can return from this representation to the normal mode, if you call the functions `Normal` and `Home`. The call of `Normal` will restore the original size of the schematic and `Home` will move its upper left corner into the upper left corner of the workarea.

The function `Redraw` can be used to refresh the drawing of the schematic within the workarea. This function is useful, if there remains some dirt from entering and deleting objects as macros, wires or comments. Some functions automatically do a redraw operation, so that you do not have to call it very often.

The function `Inverse` changes the background colour of the workarea from black to white. It also changes the colours of macros, wires, comments and equations appropriately. This is useful, if you want to make screendumps from the graphical editor as we did it during the design example in section 4.8.

To terminate the editor session you have to terminate any currently active editor function first. In most cases this can be done by pressing the right mouse button at most twice within the workarea (see also the description of the appropriate function). After that you are in menu selection mode and you can finish your editor session by pressing the right mouse button within the menuline. Now the editor menu will be closed and you return to the main menu of the CADIC system. From there you can select another

tool or you can terminate the whole system call by selecting the entry `Exit` in the main menu.

Bibliography

[Bar78] M. Barbacci et.al. The Symbolic Manipulation of Computer Descriptions. Technical report, Dept. of Computer Science, Carnegie–Mellon University, 1978.

[Biw94] M. Biwersi. μsic – ein kleiner Silicon-Compiler. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, FRG, 1994.

[Bur94] Th. Burch. *Eine graphische Arbeitsumgebung für den parametrisierten Entwurf integrierter Schaltkreise.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, 1994.

[CAD92] CADENCE. *Cadence Design System Manual*, 1992.

[CF86] E. Clarke and Y. Feng. ESCHER – A Geometrical Layout System for Recursively Defined Circuits. In *Proceedings of the 23rd Design Automation Conference*, pages 649–653, June 1986.

[CNR87] H.A. Choi, K. Nakajima, and C.S. Rim. Complexity Results for vertex–deletion Graph Bipartization and Via Minimization Problems. In *Proceedings of the 25th Annual Allerton Conference on Computing, Communication and Control*, September 1987.

[Com87] EDIF Steering Committee. *EDIF Electronic Design Interchange Format Version 2 0 0.* Electronic Industries Association, Washington D.C., May 1987.

[DBR+88] P. J. Drongowski, J. R. Bammi, R. Ramaswamy, S. Iyengar, and T. H. Wang. A Graphical Hardware Design Language. In *Proceedings of the 25th Design Automation Conference*, pages 108–115, 1988.

[Fet95] Th. Fettig. Lokale Plazierung in CADIC. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1995. to appear.

[Gen90] GenRad. *HILO Reference Manual*, 1990.

[GL85] S. M. German and K. J. Lieberherr. Zeus: A Language for Expressing Algorithms in Hardware. *IEEE Transactions on Computer Aided Design*, pages 55–65, Februar 1985.

[Gra92] B. Grande. Verfahren zur hierarchischen Schichtzuweisung und ihre Implementierung in CADIC. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1992.

[HMZ91] G. Hotz, P. Molitor, and W. Zimmer. On the Construction of Very Large Integer Multipliers. In *Proceedings of EURO ASIC 91*, pages 266–269, May 1991.

[Hot65] G. Hotz. Eine Algebraisierung des Syntheseproblems für Schaltkreise. *EIK Journal of Information Processing and Cybernetics*, 1:185–205,209–231, 1965.

[Hot74] G. Hotz. *Schaltkreistheorie.* de Gruyter Lehrbuch. Walter de Gruyter Verlag, 1974.

[KCS88] Y.S. Kuo, T.C. Chern, and W. Shih. Fast Algorithm for Optimal Layer Assignment. In *Proceedings of the 25th Design Automation Conference (DAC88)*, pages 554–559, June 1988.

[KMO89] R. Kolla, P. Molitor, and H. G. Osthof. *Einführung in den VLSI–Entwurf.* Leitfäden und Monographien der Informatik. B.G. Teubner Verlag, Stuttgart, 1989.

[Knu73] D.E. Knuth. *Sorting and Searching, The Art of Computer Programming.* Addison–Wesley, 1973.

[Kol86] R. Kolla. *Spezifikation und Expansion logisch topologischer Netze*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, 1986.

[Lim82] W. Y.-P. Lim. HISDL – A Structure Description Language. *Comm. ACM*, Vol. 25:823–830, November 1982.

[LSU89] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1989. 300 Seiten.

[LV83] W.K. Luk and J. Vuillemin. Recursive Implementation of Optimal Time VLSI Integer Multipliers. In *Proceedings of IFIP Congress 83*, pages 155–168, 1983.

[MN89] K. Mehlhorn and S. Näher. LEDA a Library of Efficient Data Types and Algorithms. Technical Report TR–A 04/1989, FB 10, Universität des Saarlandes, 1989.

[Mol86] P. Molitor. *Über die Bikategorie der logisch topologischen Netze und ihre Semantik*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, FRG, 1986.

[Mol87] P. Molitor. On the Contact Minimization Problem. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS87)*, pages 420–431, February 1987.

[Mol93] P. Molitor. A Hierarchy Preserving Hierarchical Bottom-up 2-Layer Wiring Algorithm with respect to Via Minimization. *INTEGRATION, the VLSI Journal*, 15:73–95, 1993.

[Sch92] J. Schnabel. Generierung der Stromversorgung in CADIC. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1992.

[Sch96] Ch. Scholl. *Logiksynthese unter Ausnutzung funktionaler Eigenschaften*. PhD thesis, Fachbereich Informatik, Universität des

Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, FRG, 1996. to appear.

[Sha82] M. Shadad et.al. VHSIC Hardware Description Language. *IEEE Transactions on Computer Aided Design*, Vol. 18, Februar 1982.

[Skl60] J. Sklansky. Conditional-sum addition logic. *IRE-EC*, 9:226–231, 1960.

[SSC82] J. M. Siskind, J. R. Southard, and K. W Crouch. Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions. In *Proc. Conf. on Advanced Research in VLSI*, pages 28–40, Januar 1982.

[SYS90] TANGENT SYSTEMS. *Tangate Reference Manual*, 1990.

[Wal64] C.S. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Computers*, 13:14–17, 1964.

[Wan95] G. Wannemacher. Generierung eines symbolischen Layouts in CADIC. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, FRG, 1995. to appear.

[Wir82] N. Wirth. Hades: A Notation for the Description of Hardware. Technical report, ETH Zentrum Zurich, August 1982.

Index