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.

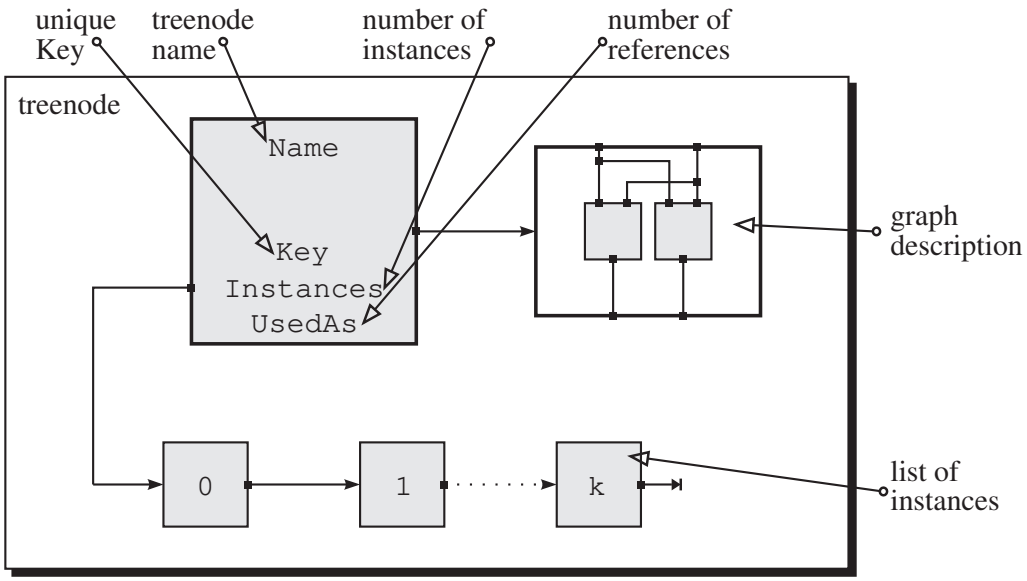


Figure 5.1: Basic structure *treenode* of the hierarchical representation

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

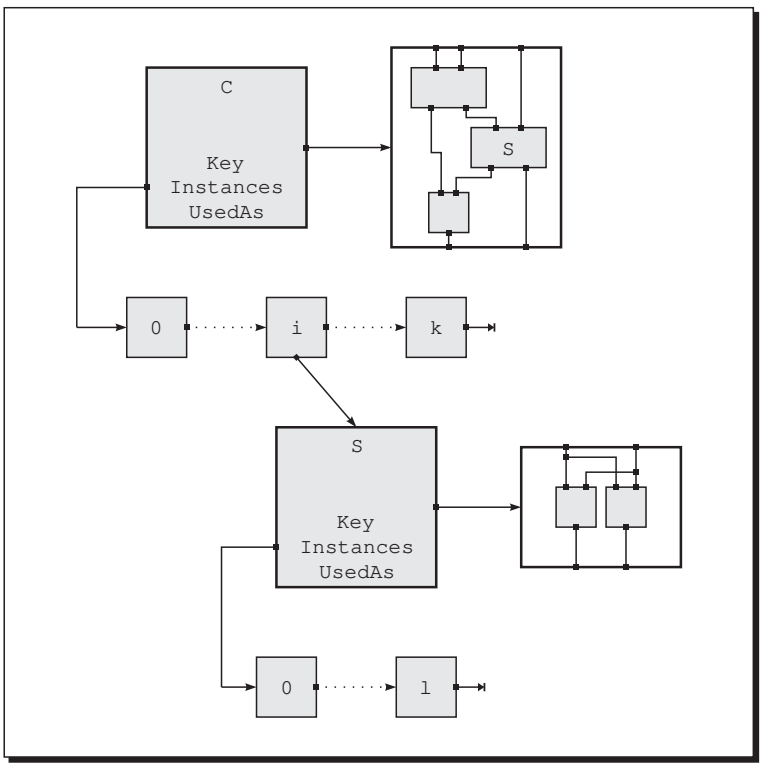


Figure 5.2: Transition from an instance of a treenode at level i to the corresponding treenode at level $i+1$

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.

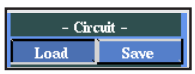
- push all instances of the current element, i.e. all its successors in the hierarchy, together with their parameter list onto the stack.
- insert the just created treenode into a hash table, such that we can check that is not creted more than once.

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.

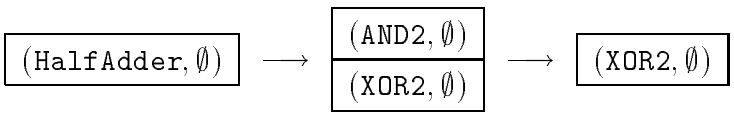


Figure 5.3: The stack during the loading of `HalfAdder`

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.

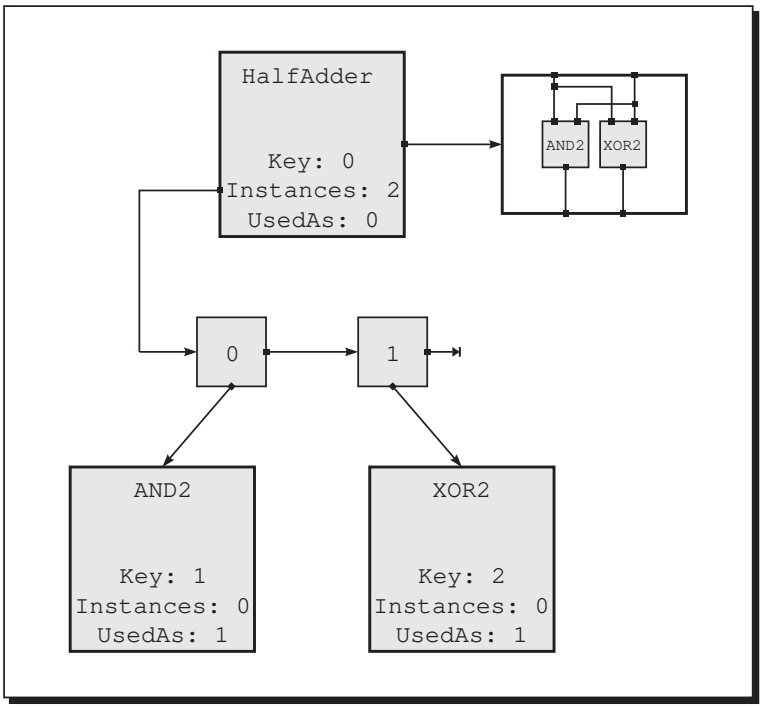


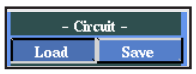
Figure 5.4: DAG structure for `HalfAdder`

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 (`FullAdder`, $\emptyset$). This entry is replaced by its instances, i.e. by two elements (`HalfAdder`, $\emptyset$) and one element (`OR2`, $\emptyset$). After that the treenode for `HalfAdder` is created and the first pair (`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

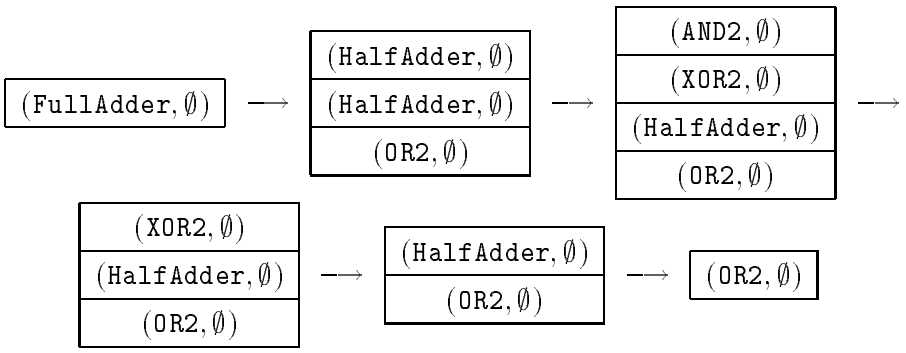


Figure 5.5: The stack during the loading of `FullAdder`

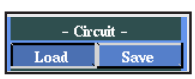
of the circuit hierarchy is given by `Circuit "FullAdder" loaded, Total Data Size: 14698 Bytes`. After the setup of the pointers from the instances to the corresponding treenodes the DAG for `FullAdder` has the form shown in figure 5.6.

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

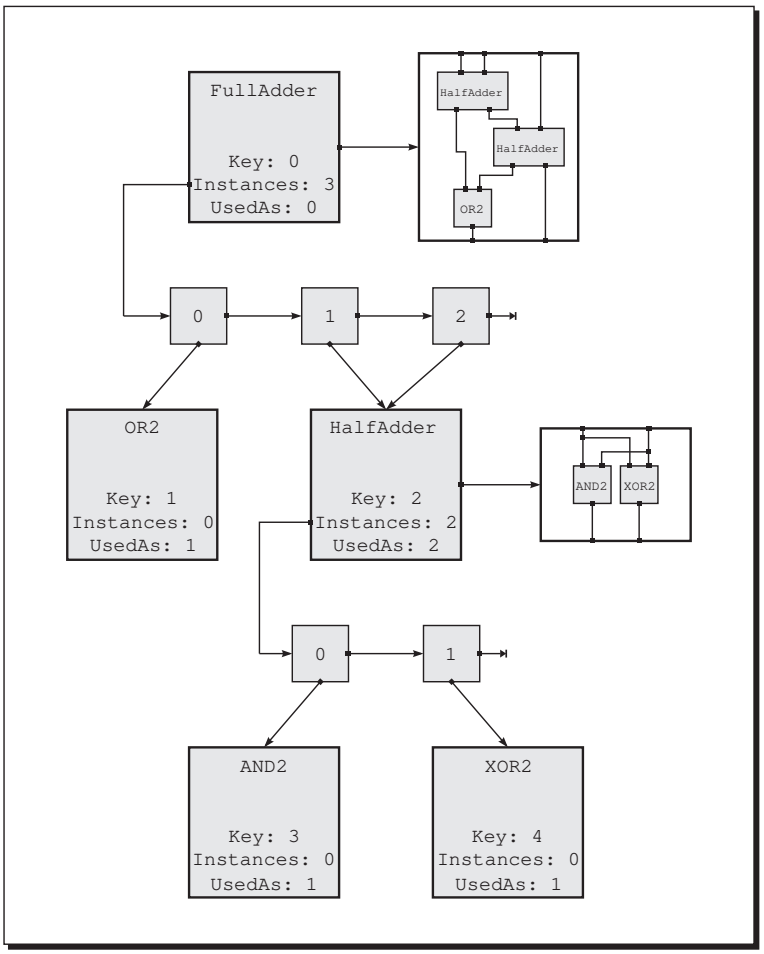


Figure 5.6: DAG structure for `FullAdder`

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).

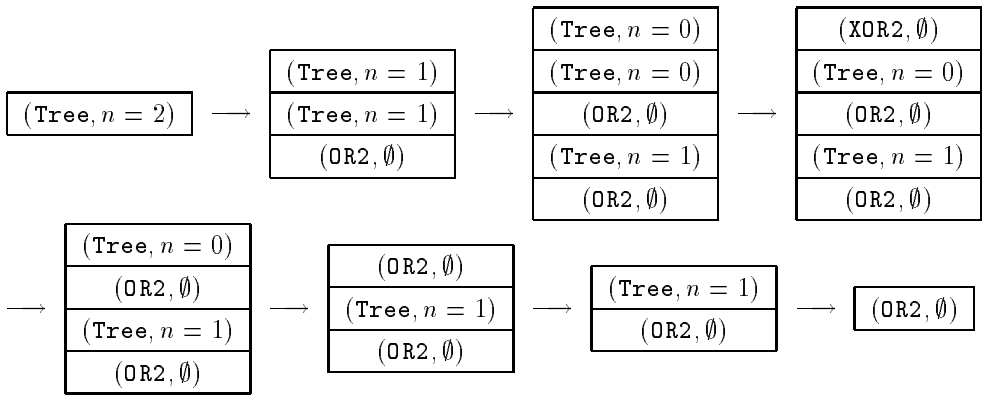


Figure 5.7: The stack during the loading of `Tree[2]`

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.

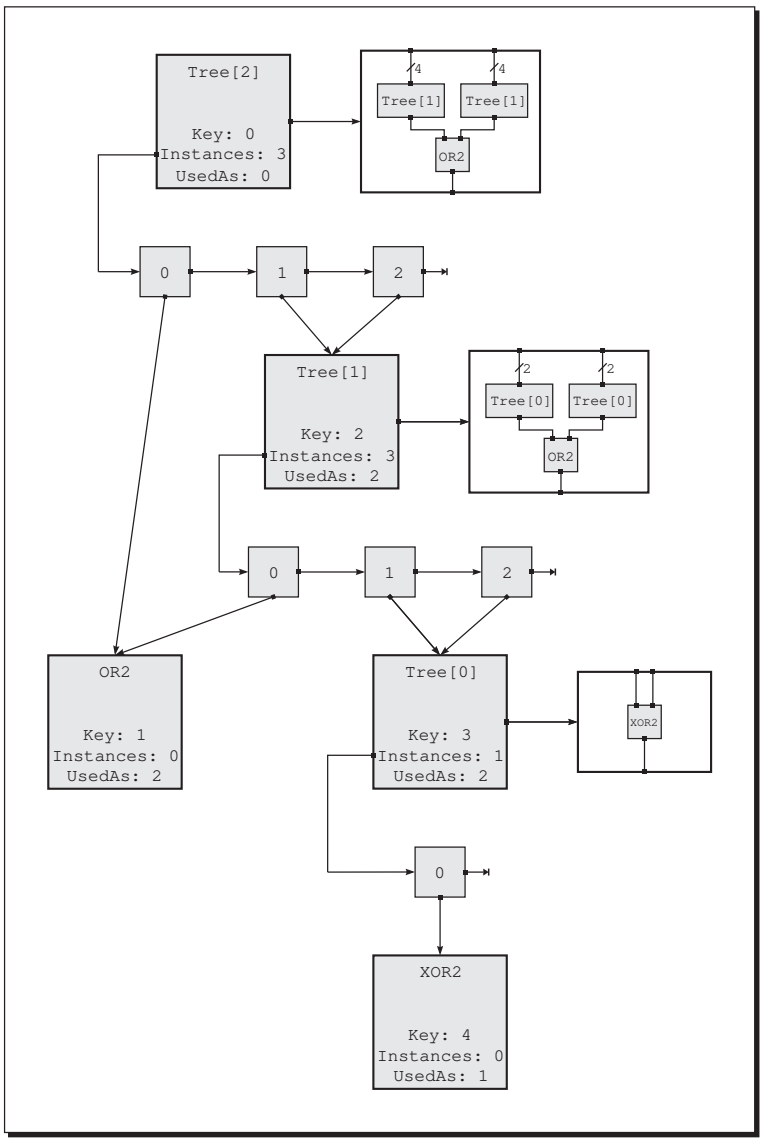


Figure 5.8: DAG structure for `Tree[2]`

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.

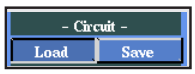
In the example of the comparison tree the hierarchical description is already smaller than the flat representation of the circuit. Figure 5.8 shows, that we need five treenodes for the hierarchical description of `Tree[2]` which also represents seven basic gates. If we would build the DAG for the next 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

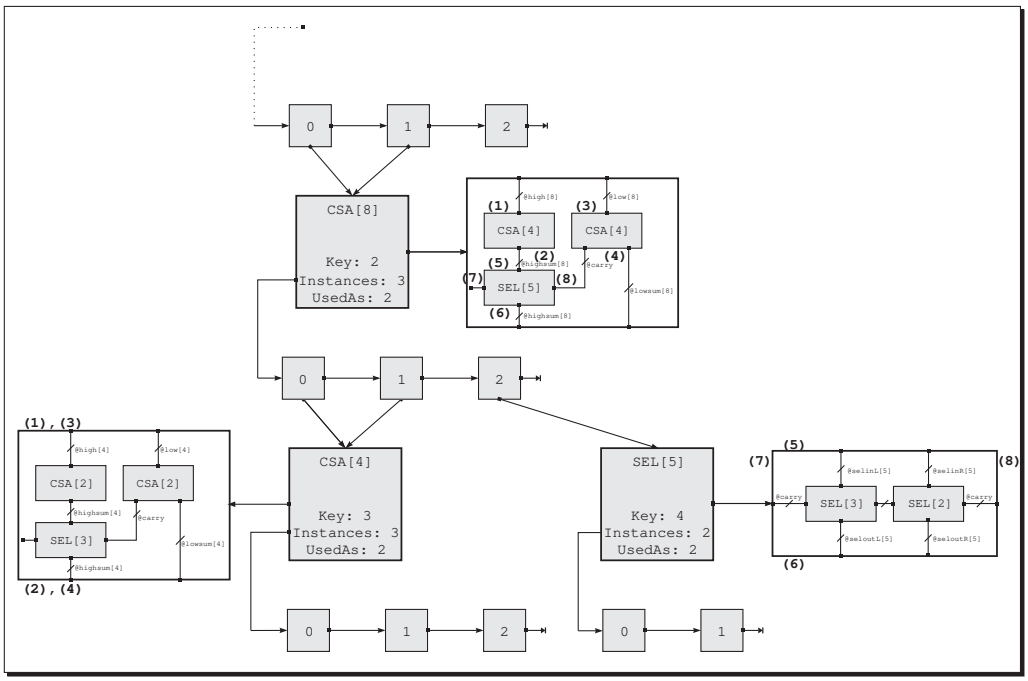


Figure 5.9: Deriving equations about the wire variables

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

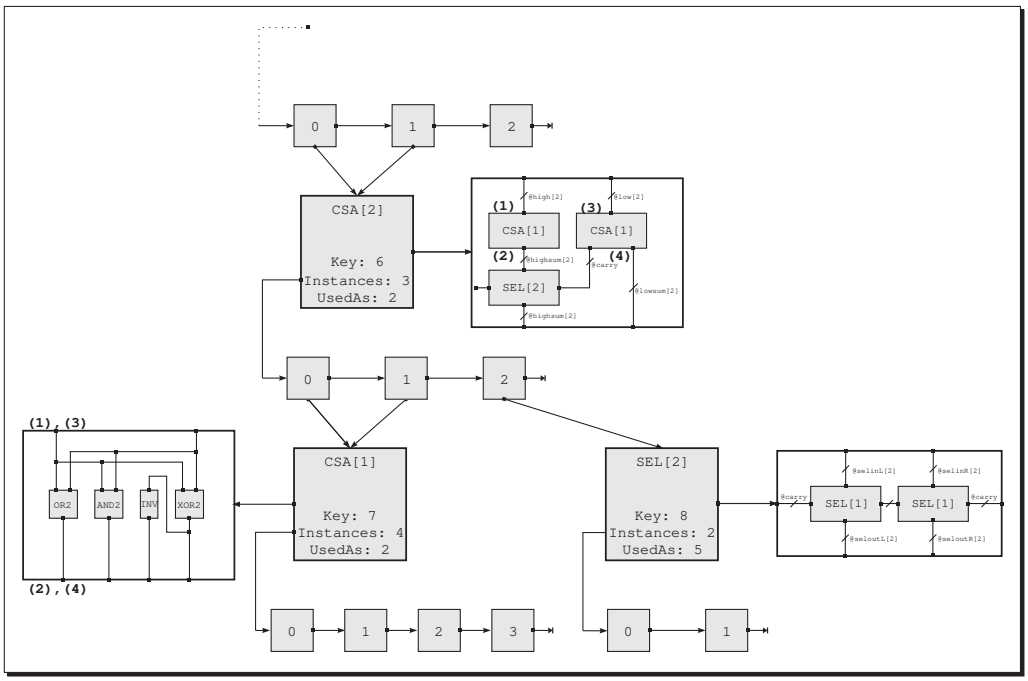


Figure 5.10: Equations for the wire variables at the lowest hierarchy level

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

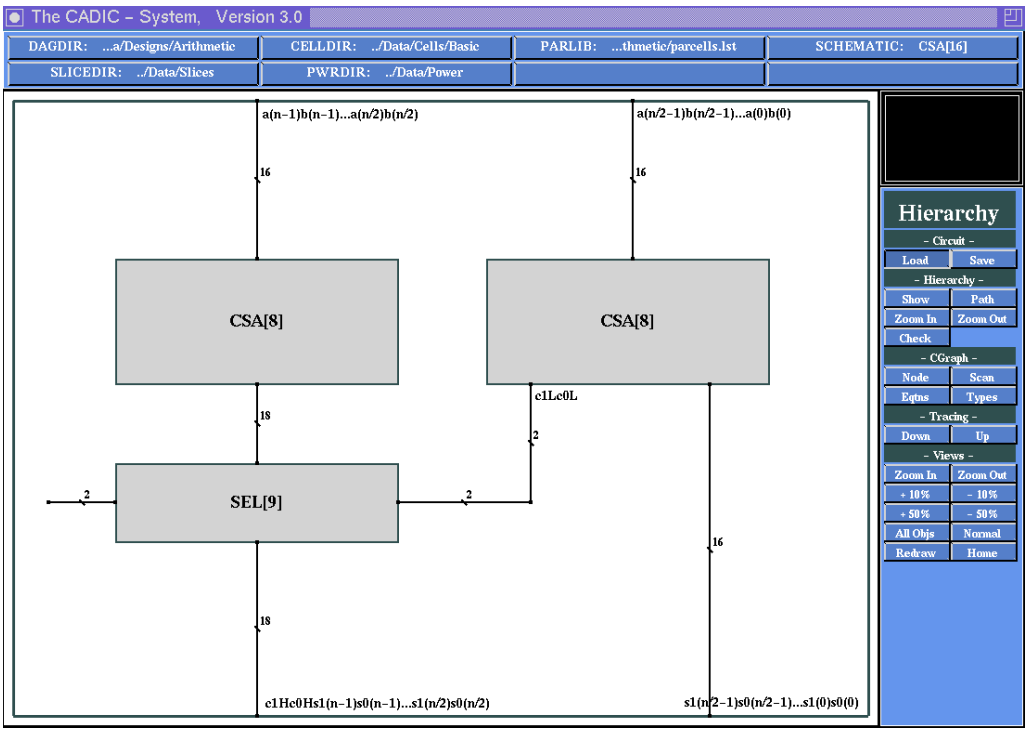


Figure 5.11: Evaluated wire variables at the top level of `CSA[16]`

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

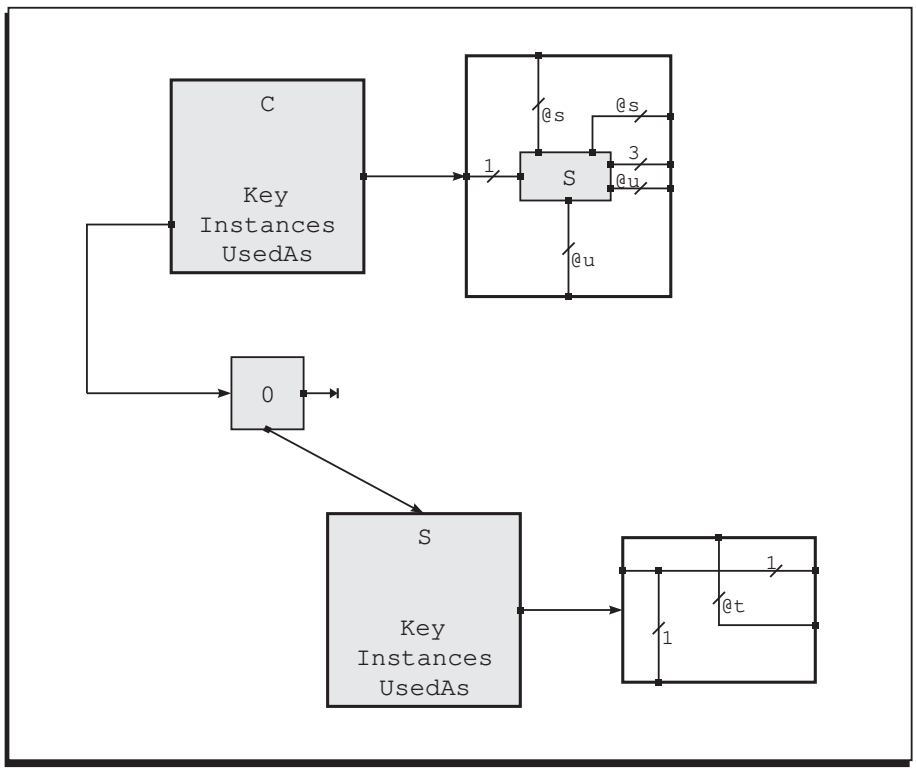
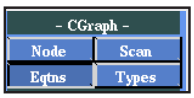


Figure 5.12: Specification with illegal solution for the system of linear equations over its wire variables

```
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

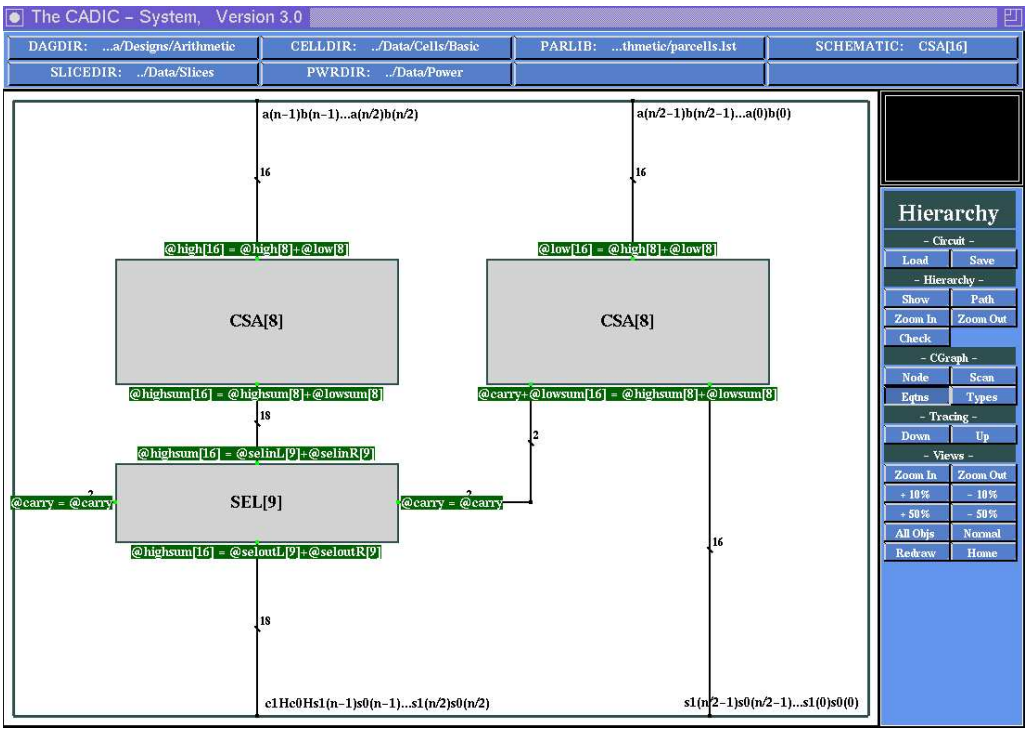


Figure 5.13: Display of the equations about the wire variables

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

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

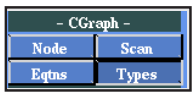
`@highsum[16]` = `@highsum[8]` + `@lowsum[8]`

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.

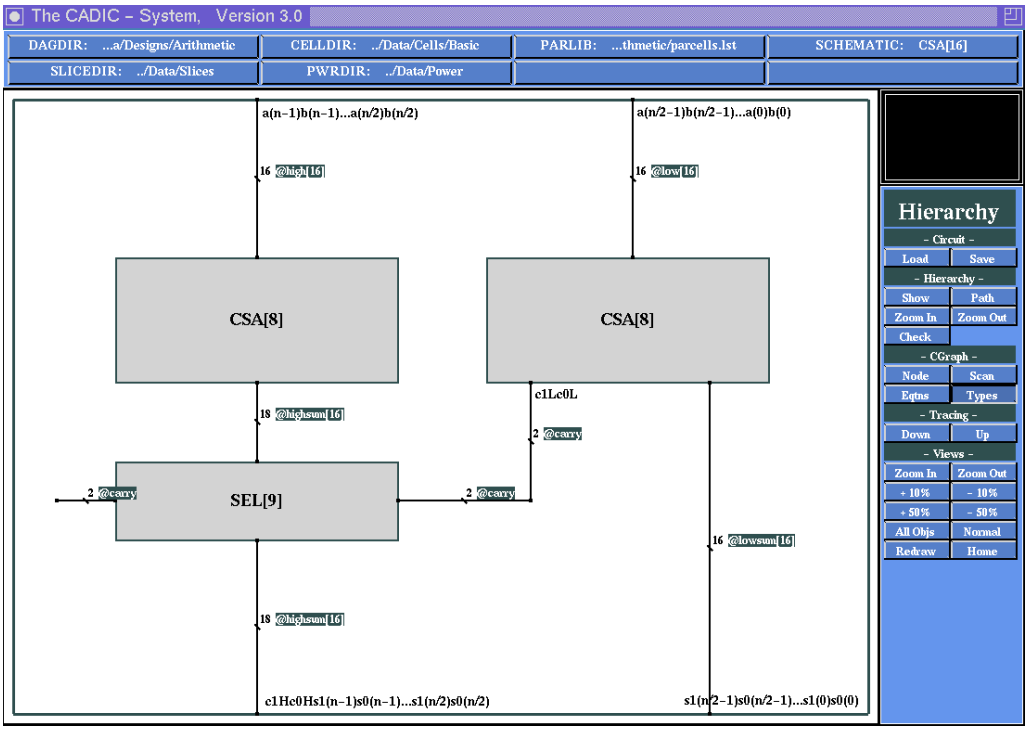


Figure 5.14: Display of the names of wire variables

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.

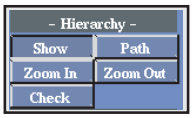
  An example for such a generic circuit is an odd–even-mergesort network which can be specified independently of the type of elements it should sort. The type (e.g. boolean, 32 bit integer, etc.) of the elements is simply represented by an appropriate wire variable `@t`. To configure the sorting network 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.

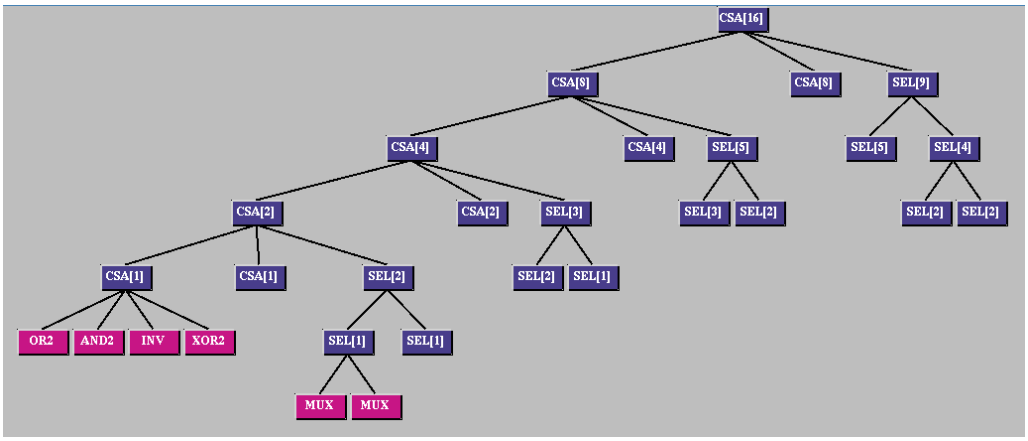


Figure 5.15: Visualization of the DAG structure for `CSA[16]`

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.

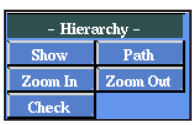
Changing the Visible Area

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.

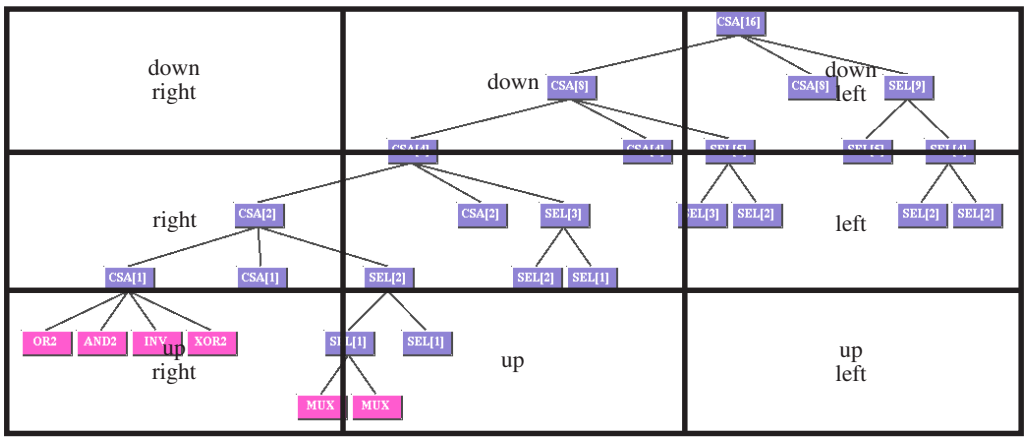


Figure 5.16: Grid of the hierarchy window for scrolling the visible area

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.

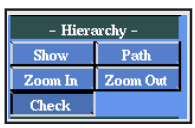
Because the number of external connectors at each border of an instance which may be a basic cell, macro or parameterized macro, must be equal to the number of its internal connectors, this is a powerful check to find out whether the recursive specification is free of errors.

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:
```

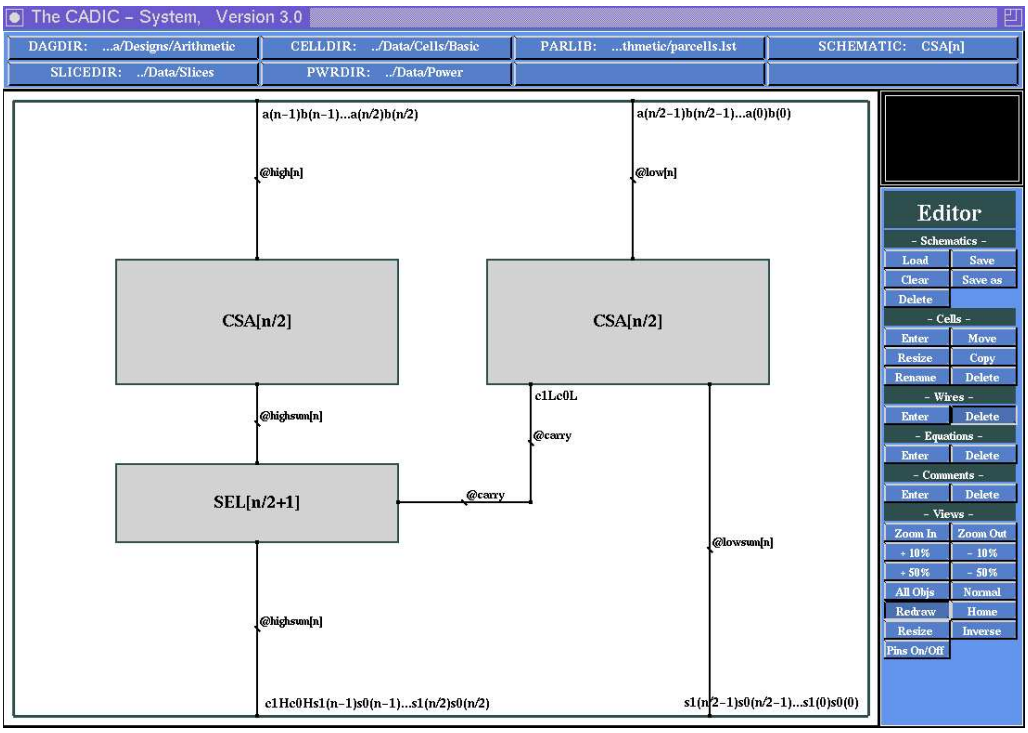


Figure 5.17: An error in the schematic `CSA[n]` at the western border of the instance `SEL[n/2+1]`

```
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.

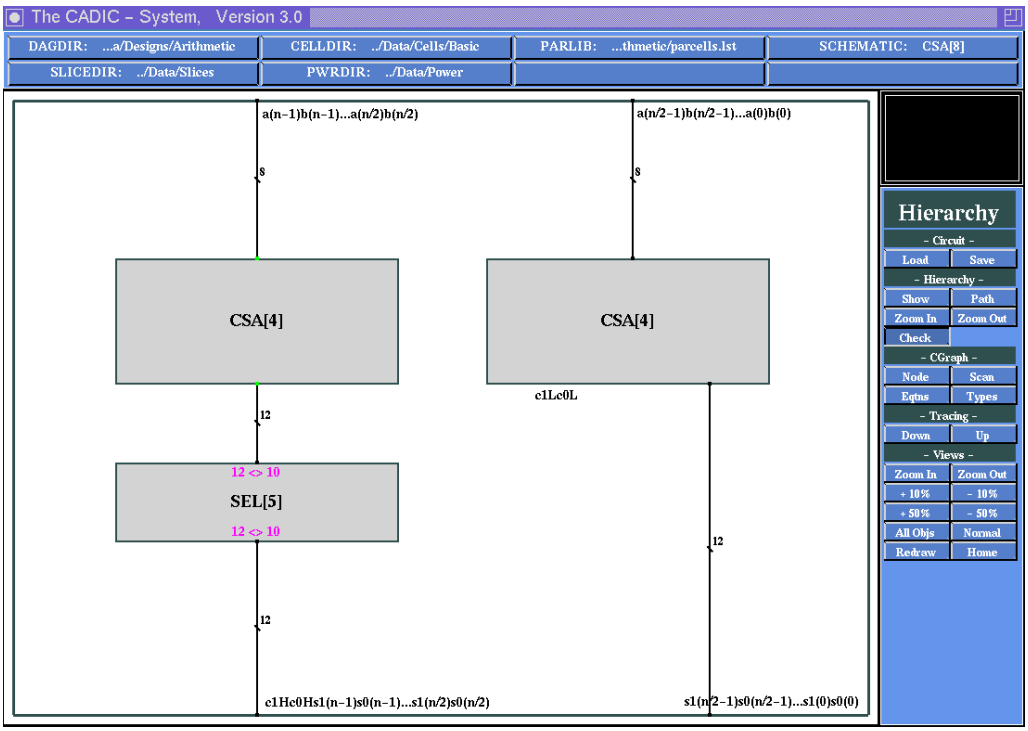


Figure 5.18: Visualization of specification errors at the hierarchy level `CSA[8]`

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.

⇝ 5.6

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.

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).

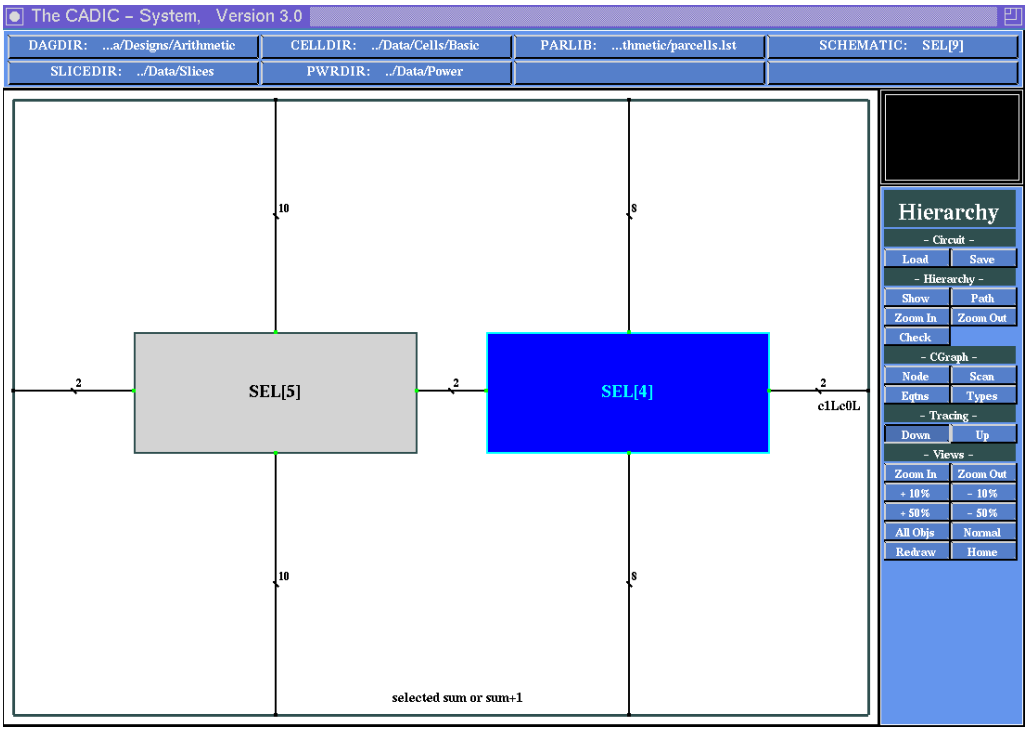


Figure 5.19: Tracing down from `CSA[16]` into `SEL[9]`

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.

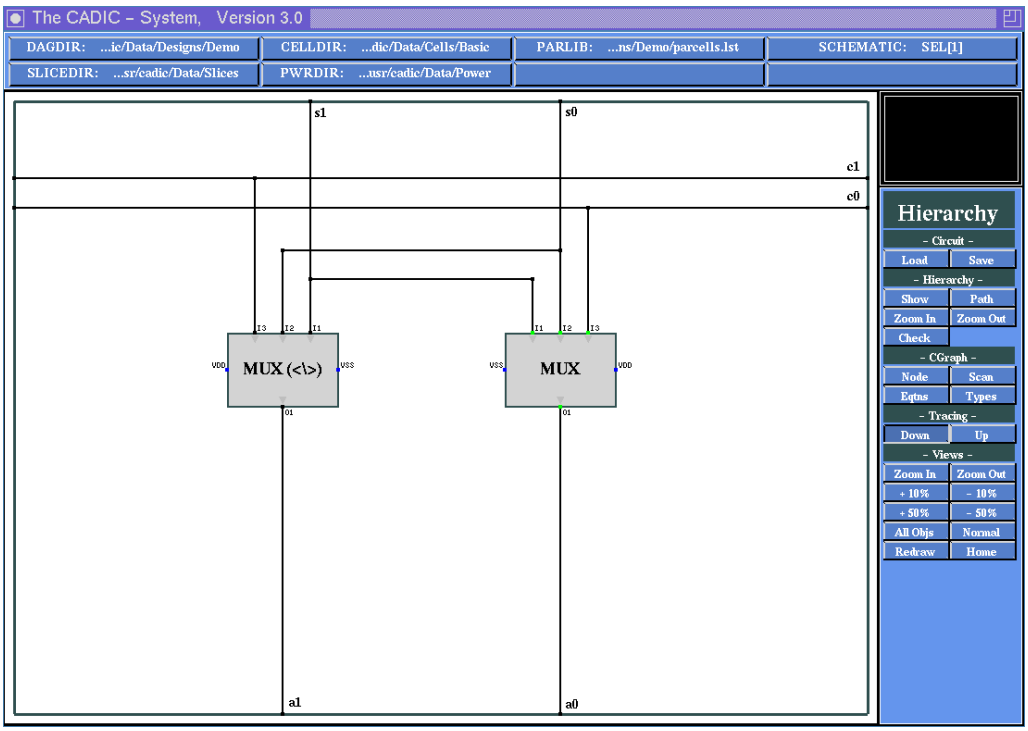


Figure 5.20: Tracing down from SEL[2] into SEL[1] and arriving at the lowest hierarchy level

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.

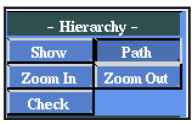
Tracing Up

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

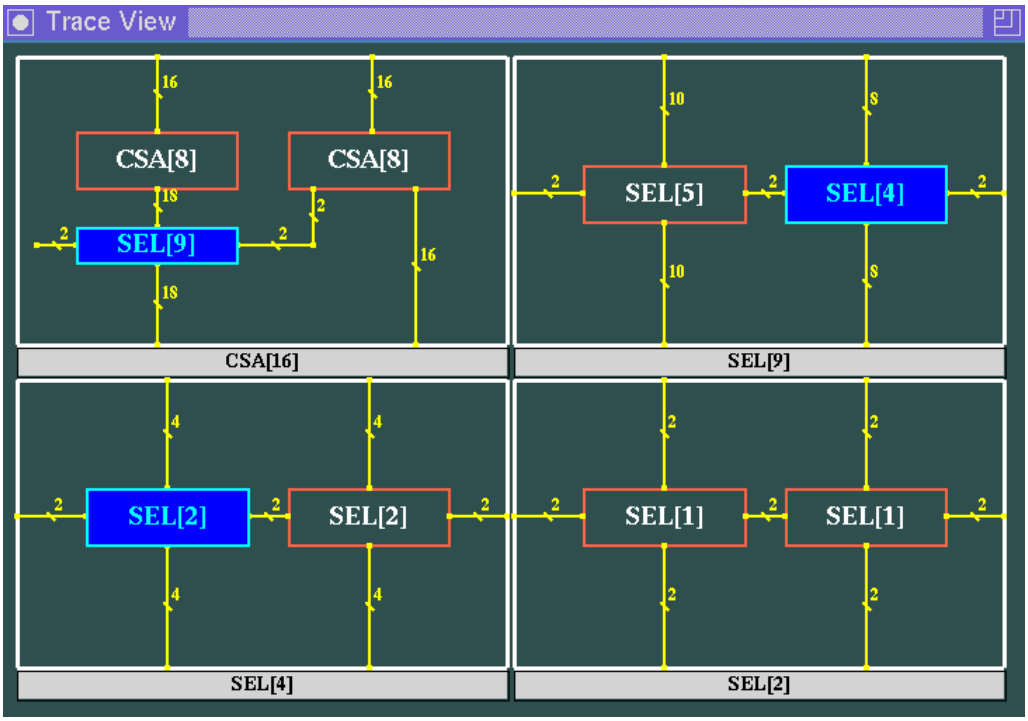


Figure 5.21: Trace view window with path from `CSA[4]`, then `SEL[3]` to `SEL[1]`

`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.

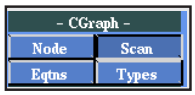
The tracing operations are essential for the work with CADIC . They allow you to inspect the hierarchy of a design in every desired way. In combination with the visualization of design results they are a powerful method for analyzing a circuit. We will use the tracing functions with nearly every integrated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

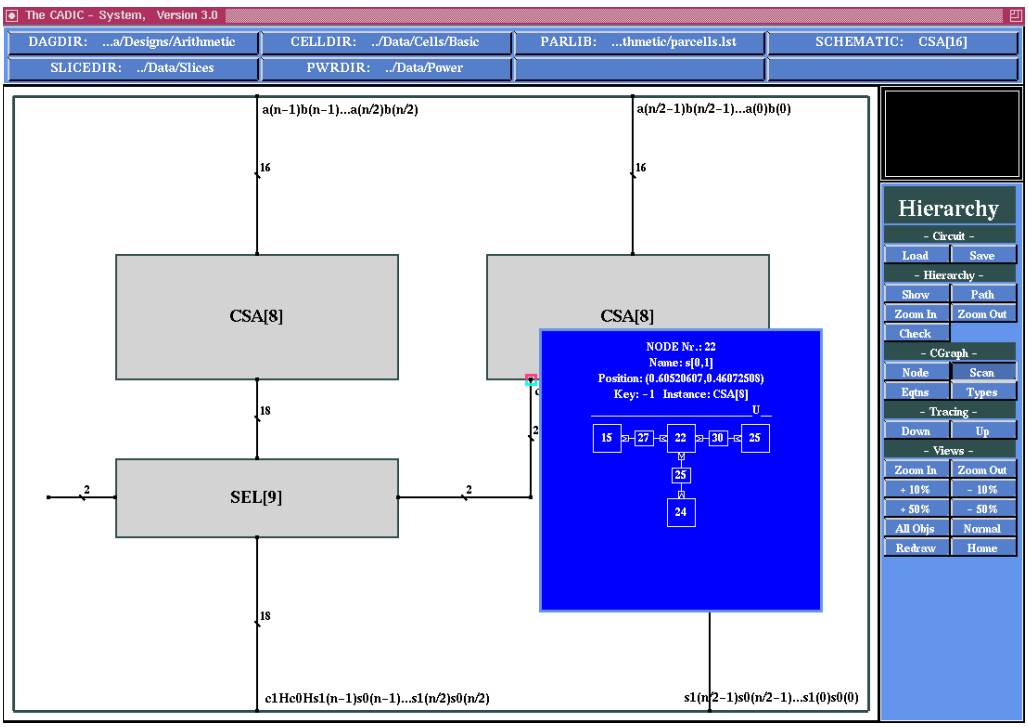


Figure 5.22: Cgraph window with node informations

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.

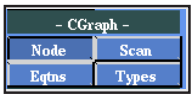
In our example, the node is the first pin at the southern border of the right instance `CSA[8]` in the schematic `CSA[16]`. Its internal number is `22`. The name is `s[0,1]` since the connected wire is a bundled wire of width 2. Its position coordinate is `(0.60520607, 0.46072508)`. The instance to which 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

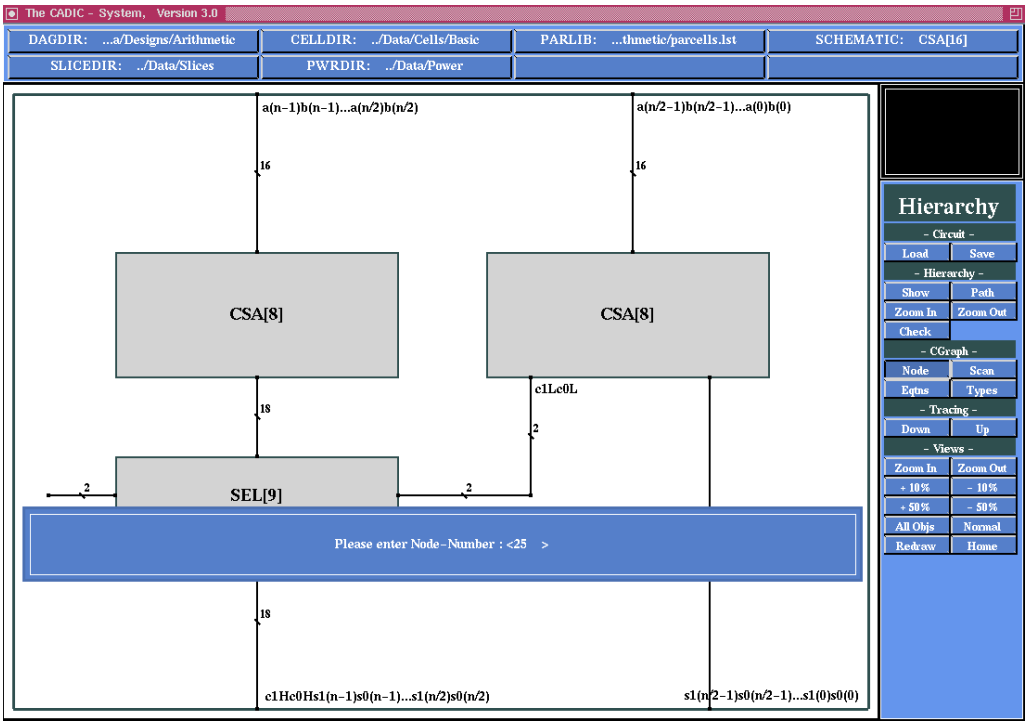


Figure 5.23: Input window with the prompt for showing the information of a specified node

⇝ 5.5

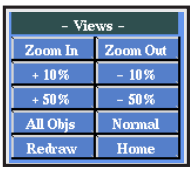
errors with the help of the check functions from section 5.5. You should always remember, for which set of parameter values your specification works fine. The specification level of the graphical editor provides a lot of functions, such that you also can manage more complicated recursions (not only reductions 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`.

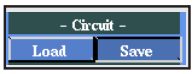
Miscellaneous

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`.