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]).

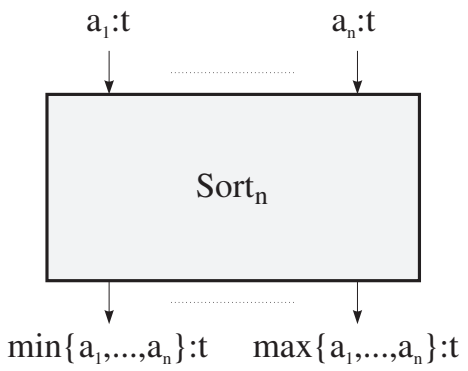

Figure 10.1: Sorting circuit for $n = 2^k$ elements of type t

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

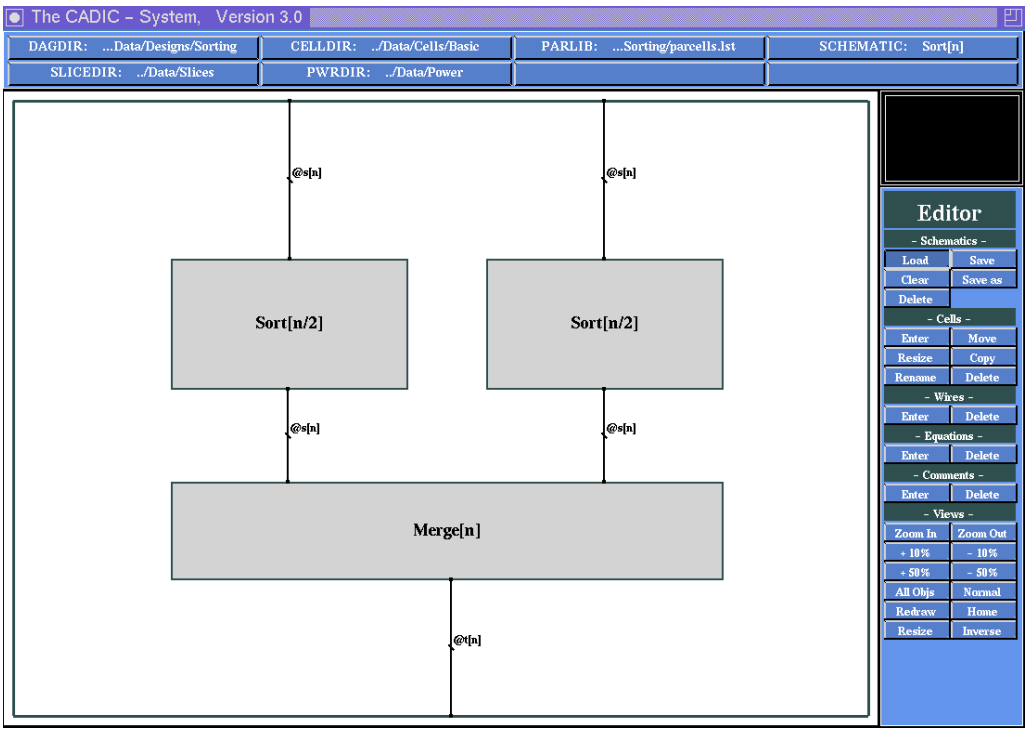

Figure 10.2: Recursive definition of the sorting of $n > 1$ elements

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

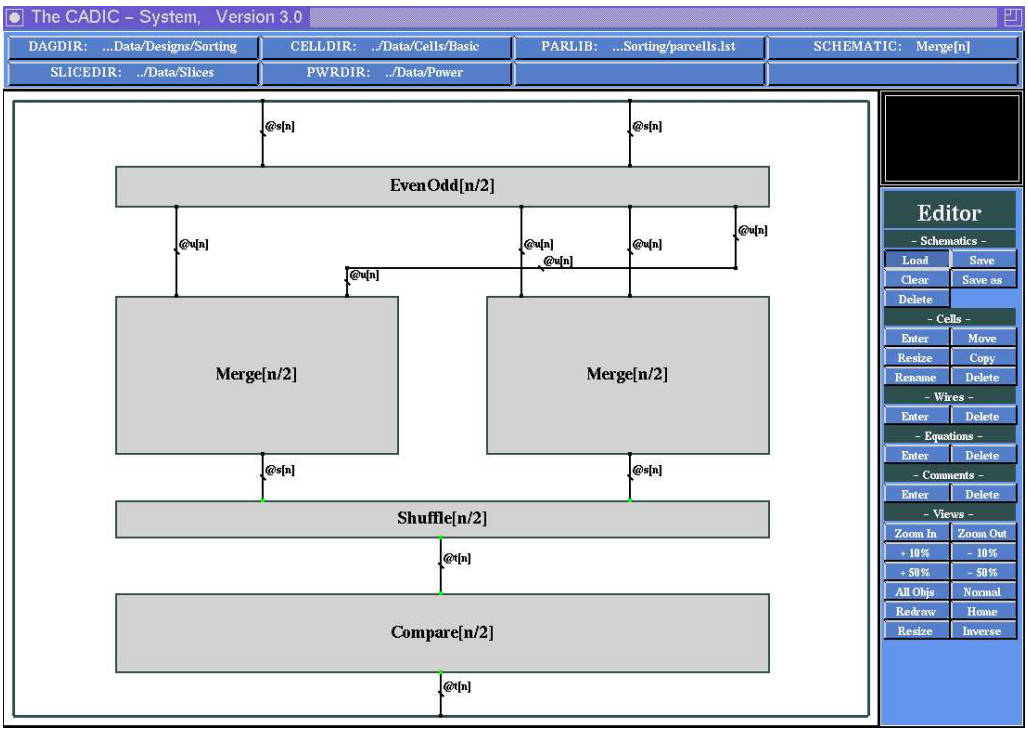

Figure 10.3: Recursive specification for the merging subcircuit `Merge[n]`

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

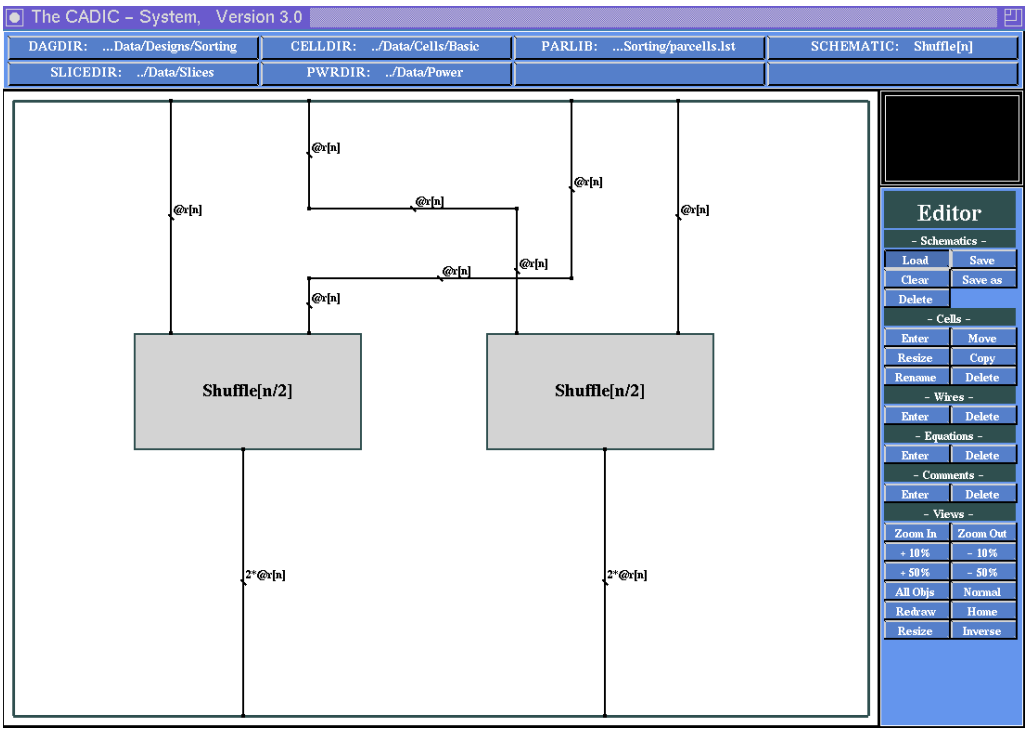

Figure 10.4: Recursive specification of the shuffling operation for two sequences of n elements

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.

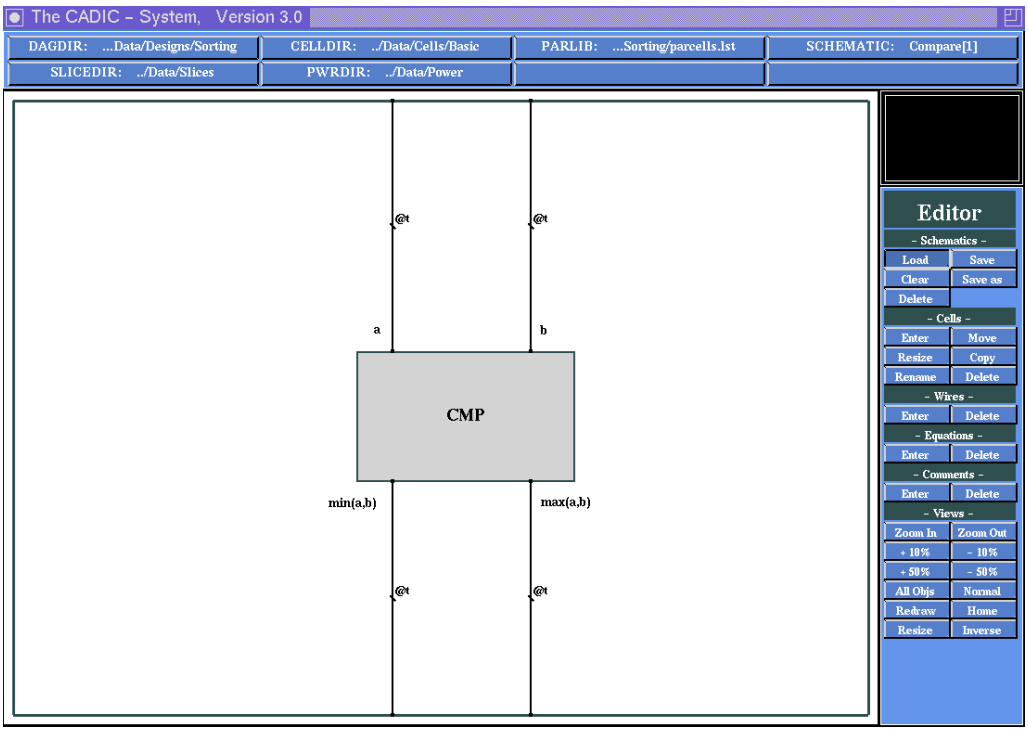

Figure 10.5: Single element of the array of compare components

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.

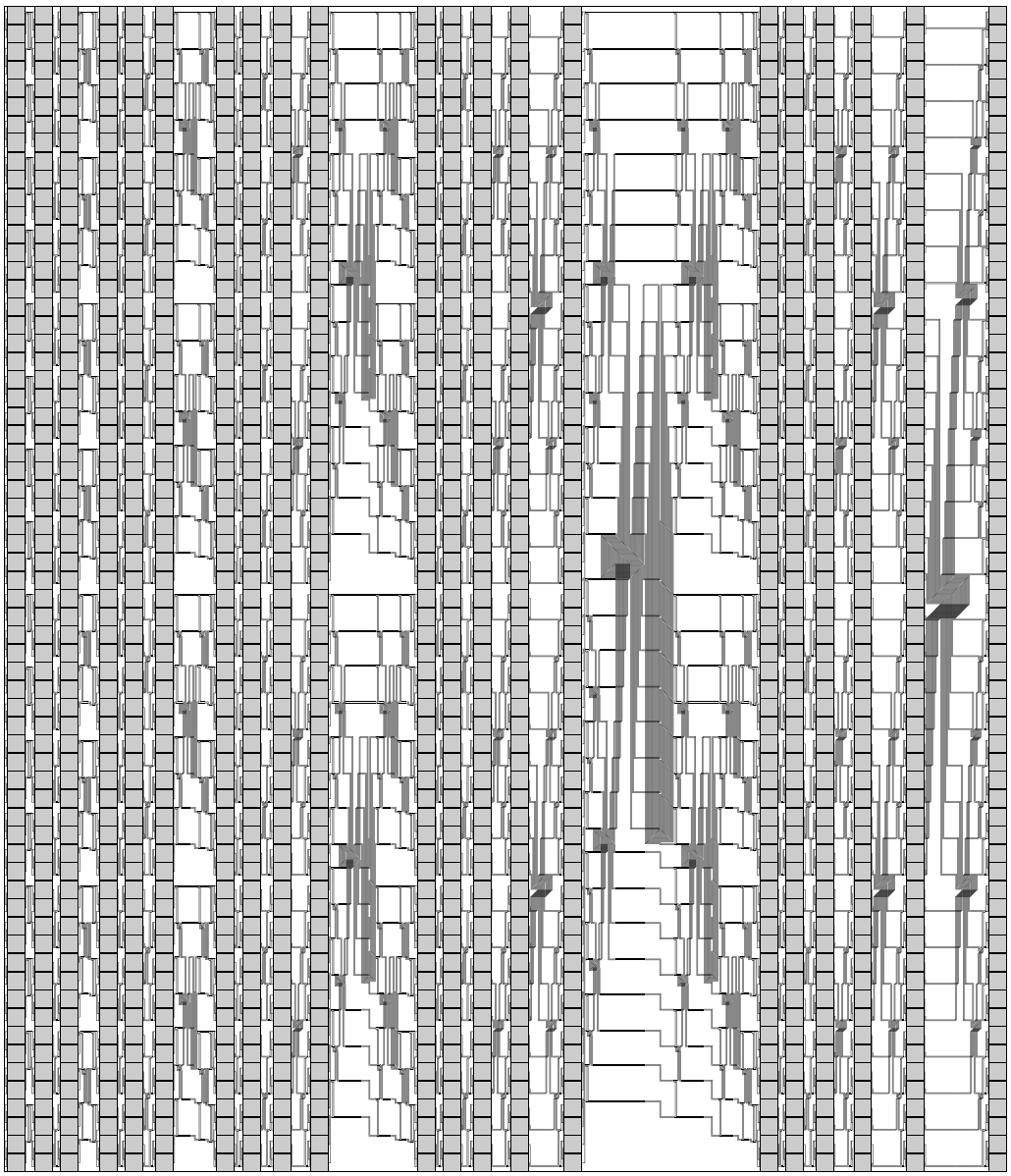

Figure 10.6: Layout for the sorting network for 64 elements of single bit values

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]).

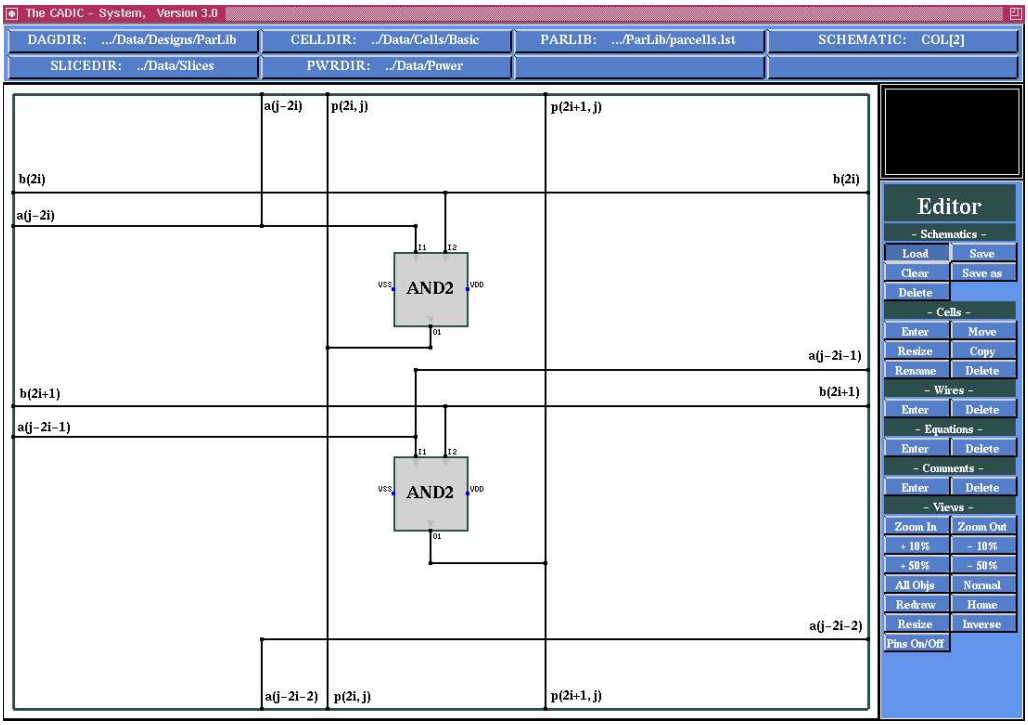

Figure 10.7: Computation of the partial products

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.

$$\begin{pmatrix} 0 & 0 & \cdots & 0 & 0 & a_{n-1}b_0 & a_{n-2}b_0 & \cdots & a_1 b_0 & a_0 b_0 \\ 0 & 0 & \cdots & 0 & a_{n-1}b_1 & a_{n-2}b_1 & a_{n-3}b_1 & \cdots & a_0 b_1 & 0 \\ \vdots & & & & \vdots & \vdots & & & & \vdots \\ 0 & a_{n-1}b_{n-1} & \cdots & a_2 b_{n-1} & a_1 b_{n-1} & a_0 b_{n-1} & 0 & \cdots & 0 & 0 \end{pmatrix}$$

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.

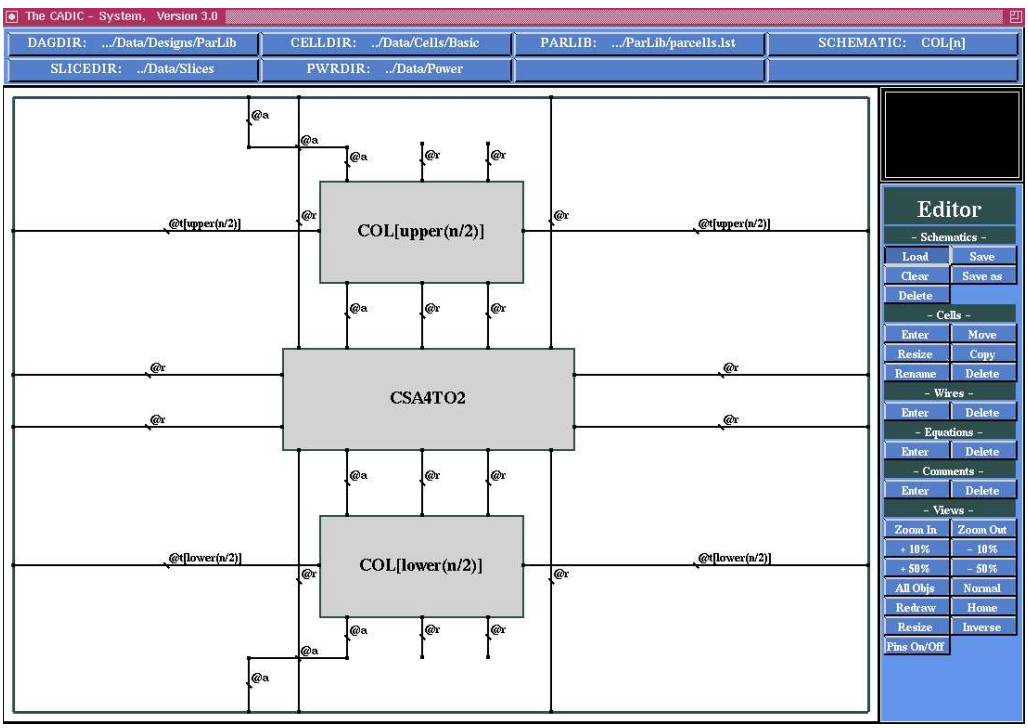

Figure 10.8: Recursive specification of one column of the partial multiplier

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

$$p^k_{4i} + p^k_{4i+1} + p^k_{4i+2} + p^k_{4i+3} = p^{k+1}_{2i} + p^{k+1}_{2i+1}$$

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.

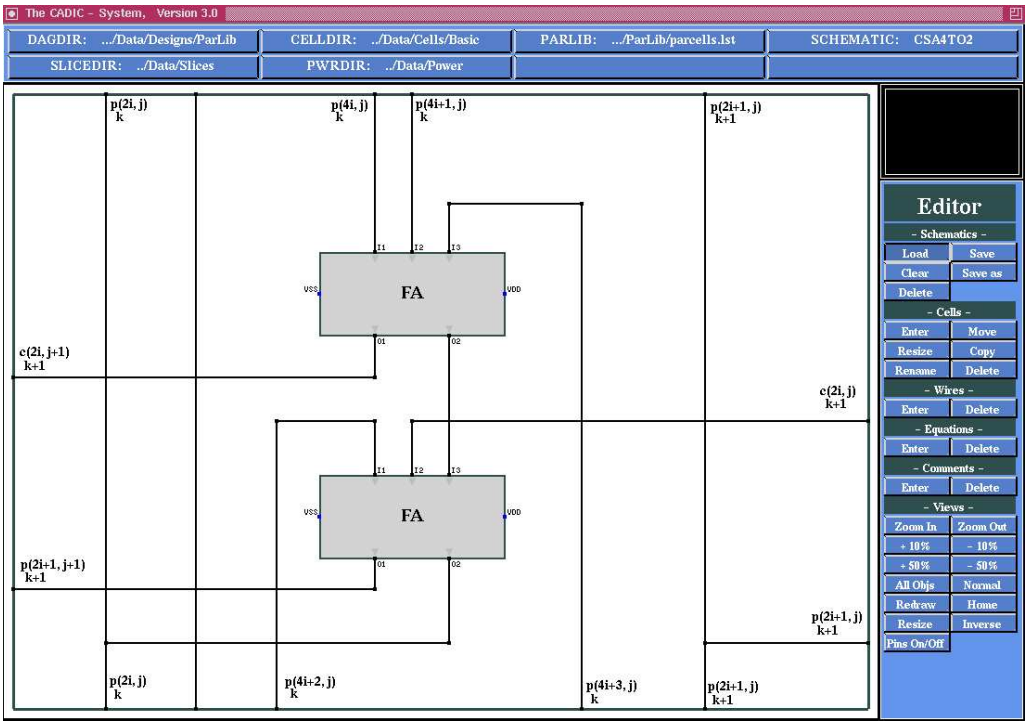

Figure 10.9: Recursive specification of one column of the partial multiplier

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

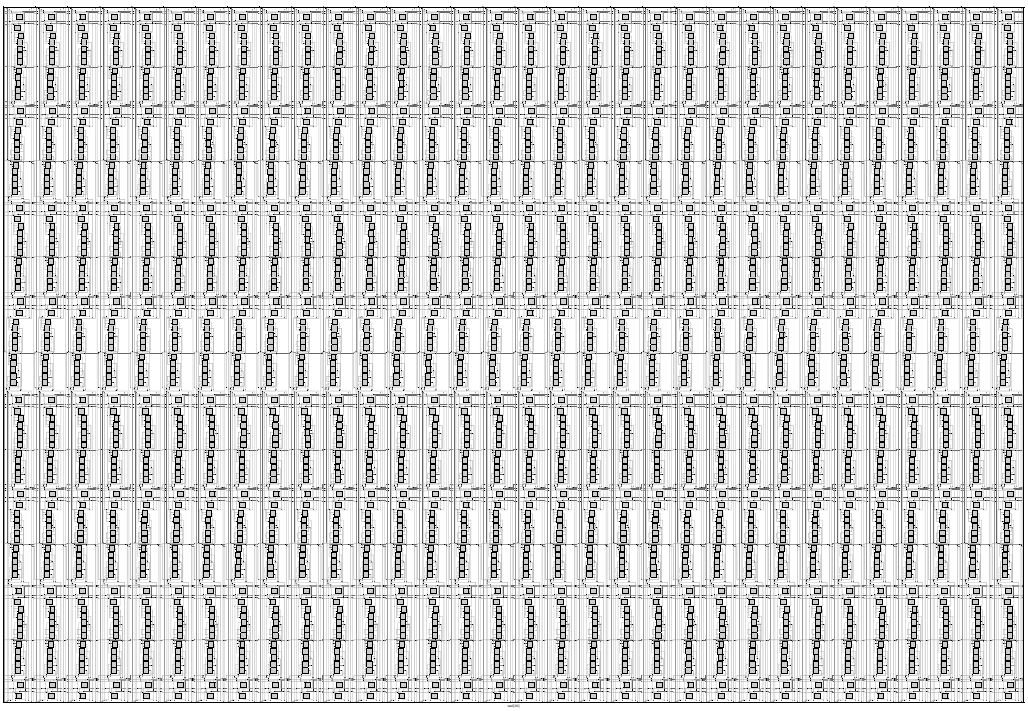

Figure 10.10: Layout for the 16 bit integer multiplier

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:

$$x = [1.x_1 \ldots x_{n-1}] \quad \text{with value}$$
$$\|x\| = 1 + \sum_{i=1}^{n-1} x_i \cdot 2^{-i}$$
$$y = [1.y_1 \ldots y_{n-1}] \quad \text{with value}$$
$$\|y\| = 1 + \sum_{i=1}^{n-1} y_i \cdot 2^{-i}$$

Precondition: $\|x\| \geq \|y\|$

Outputs

`Divider[n]` has n bits as outputs numbered $z_1, \ldots, z_n$. These are interpreted as follows:

$$z = [1.z_1 \ldots z_{n-1}] \quad \text{with value}$$
$$\|z\| = 1 + \sum_{i=1}^{n} z_i \cdot 2^{-i}$$

$\|z\|$ 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).

$$\text{If} \quad A = [a_0.a_1 \ldots a_{n-1}]_{SD2}; \quad a_i \in \{\bar{1}, 0, 1\}$$
$$\text{then} \quad \|A\|_{SD2} = \sum_{i=0}^{n-1} a_i \cdot 2^{-i}$$

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 $[q_1 \ldots q_n]_{SD2}$ into a binary number $[z_1 \ldots z_n]$ such that
   $$\|1.q_1 \ldots q_n\|_{SD2} = \|1.z_1 \ldots z_n\|$$
   (Performed in the Red2BinFast–component)

10.3.3 Graphical Specification of the Divider

Top Level: `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.

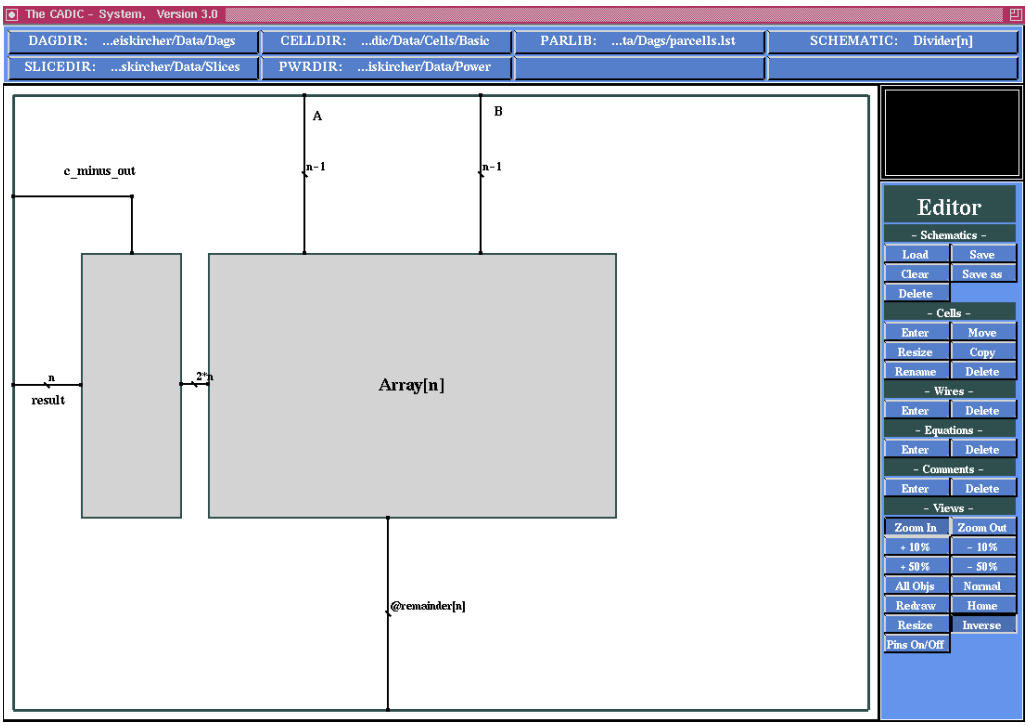

Figure 10.11: The top level of `Divider[n]`

`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

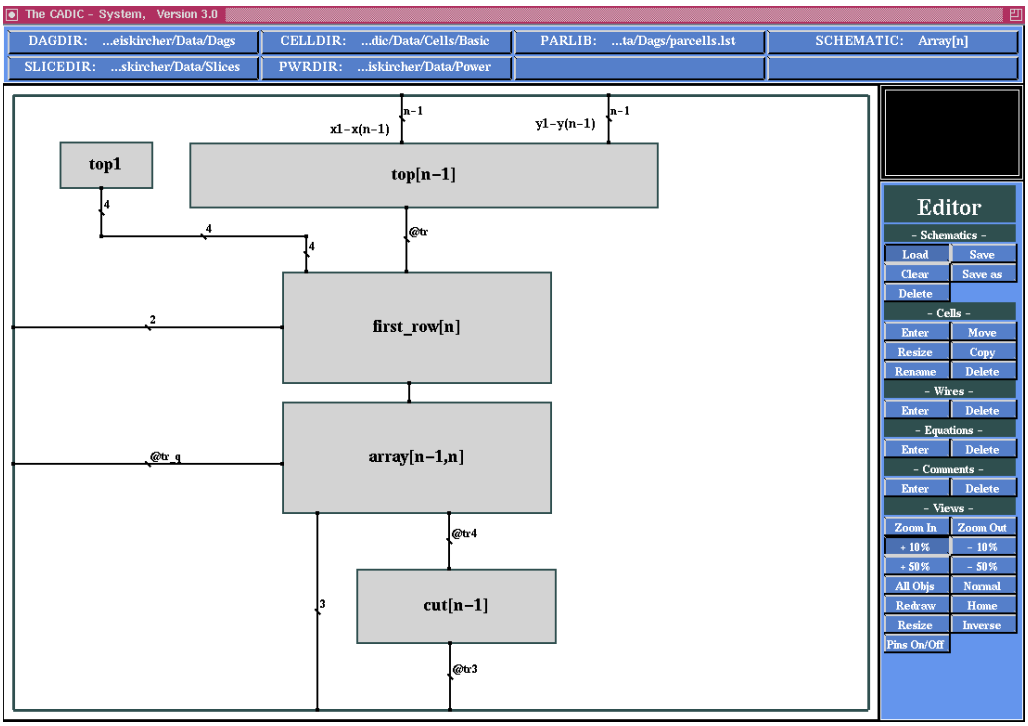

Figure 10.12: `Array[n]`

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]`

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

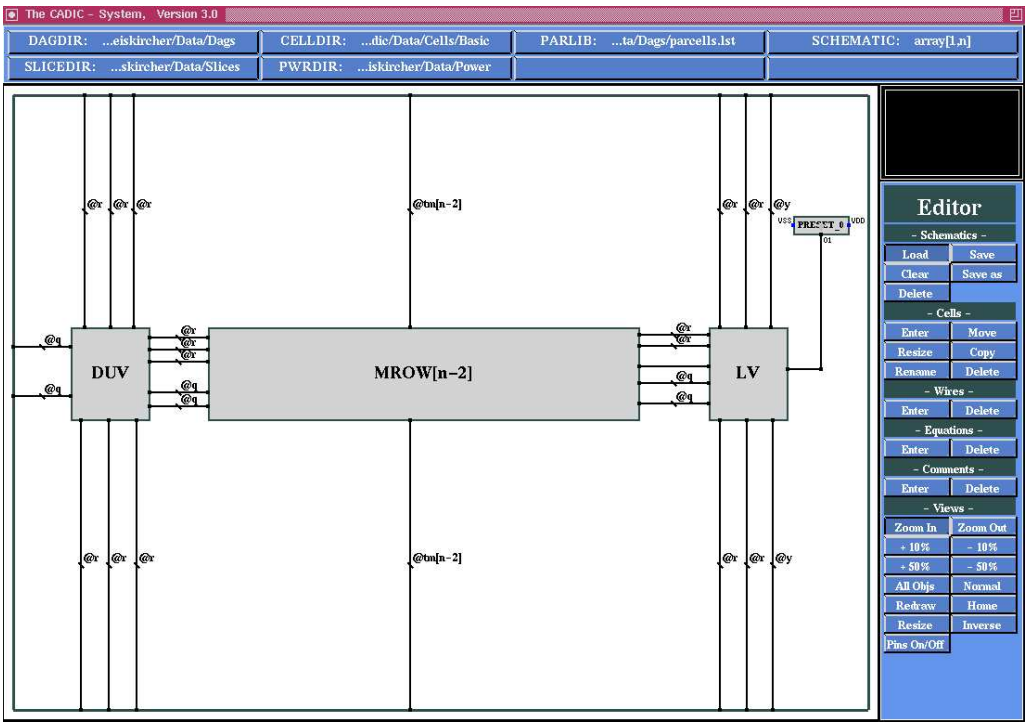

Figure 10.13: One row of the division array

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`

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

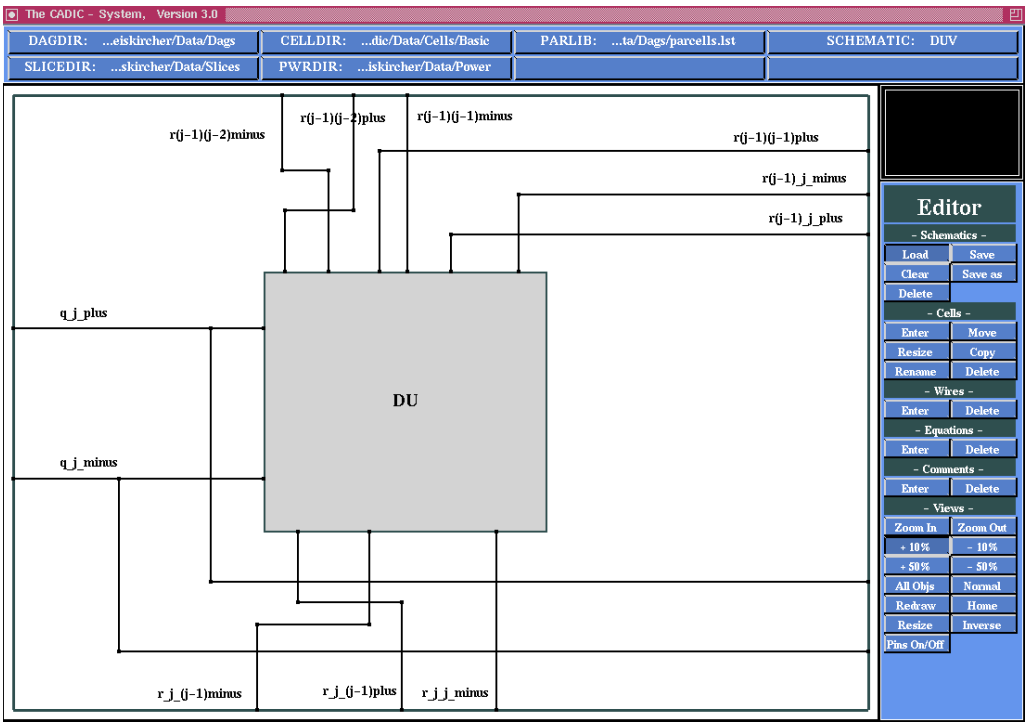

Figure 10.14: First part of each row DUV

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 $r^j_{j-1}{}^-$ and the minus–bit of the remainder–digit r^j_j are calculated according to the following table:

r^{j-1}_{j-2}	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_{j-1} and b is the bit for $r^j_j{}^-$. The last three columns stand for the different values of the digit r^{j-1}_j.

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

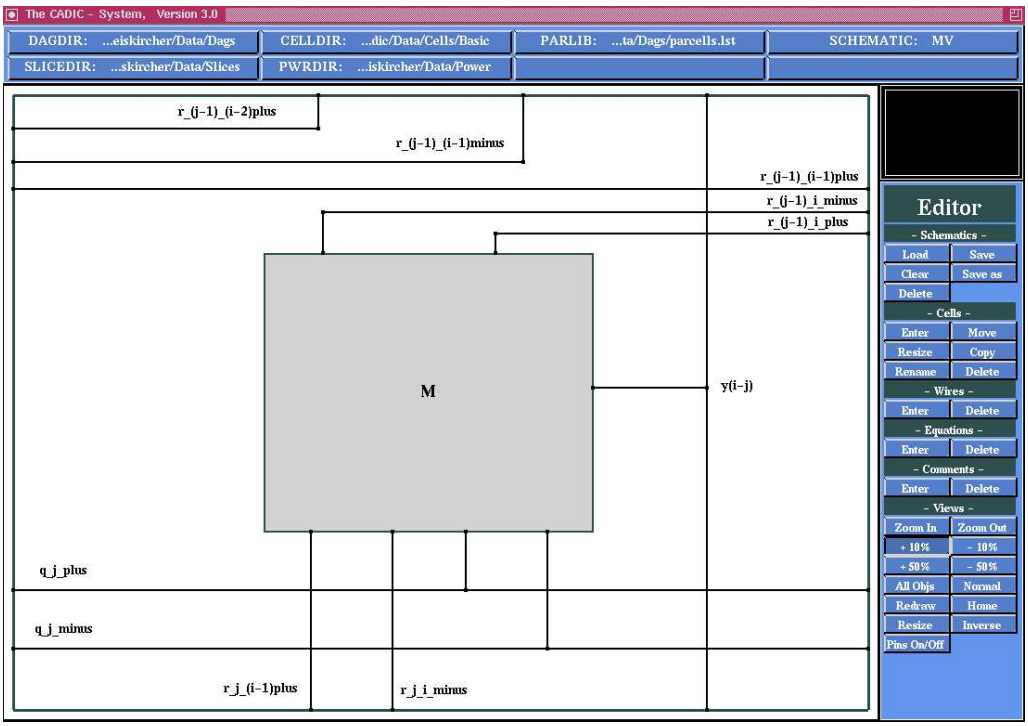

Figure 10.15: Elements in the middle of each row MV

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:

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.

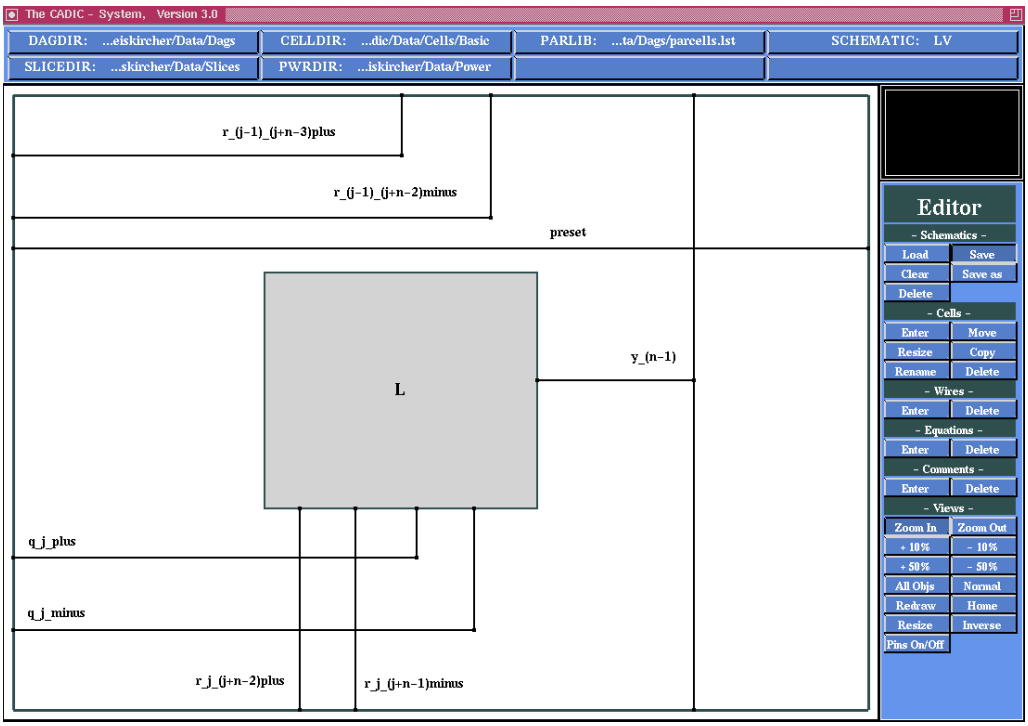

Figure 10.16: The last element in each row LV

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+n-1}^{j} which corresponds to the output–bit ${r_{j+n-2}^{j}}^{+}$ in LV. The second component is the redundant binary digit r_{j+n-1}^{j}. The output bit ${r_{j+n-1}^{j}}^{-}$ of LV is 1 if $r_{j+n-1}^{j} = \bar{1}$ and 0 otherwise.

`Red2BinFast[n]`

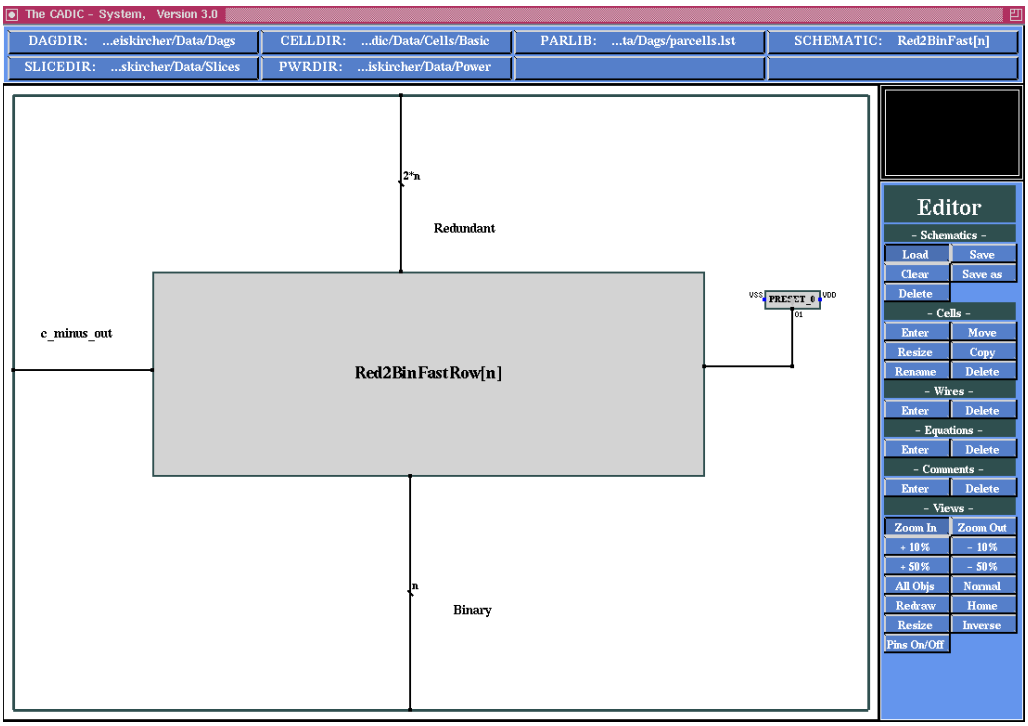

Figure 10.17: Reduction from the redundant representation to standard binary representation `Red2BinFast[n]`

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]`

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:

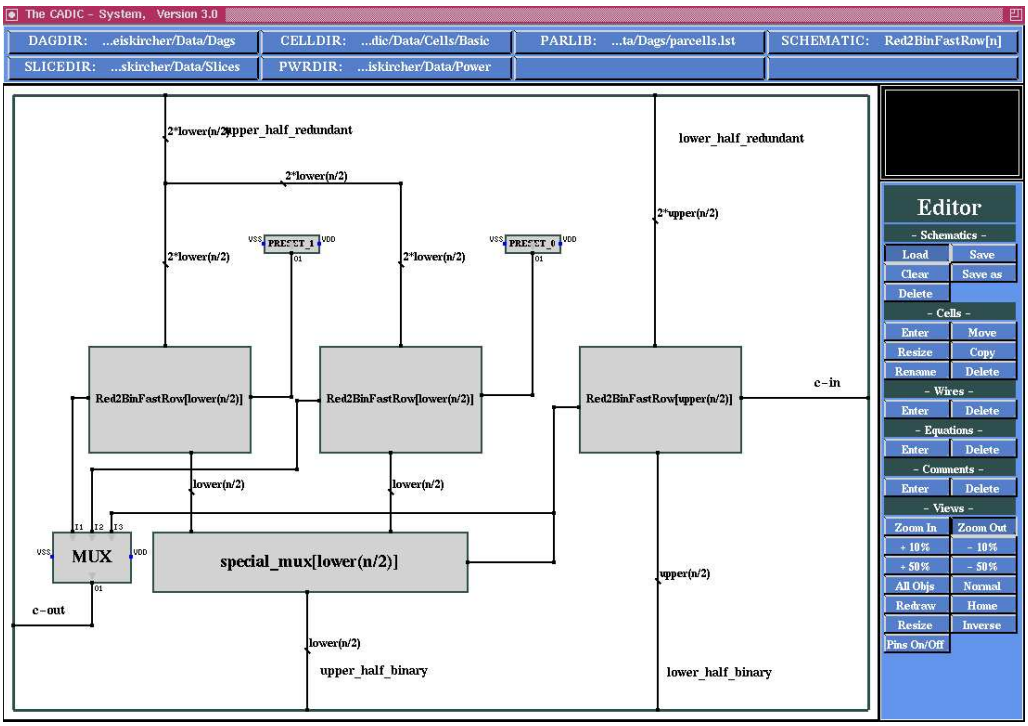

Figure 10.18: Recursive specification for `Red2BinFastRow[n]`

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 sub-circuit for the reduction of the result from the redundant to the binary representation.

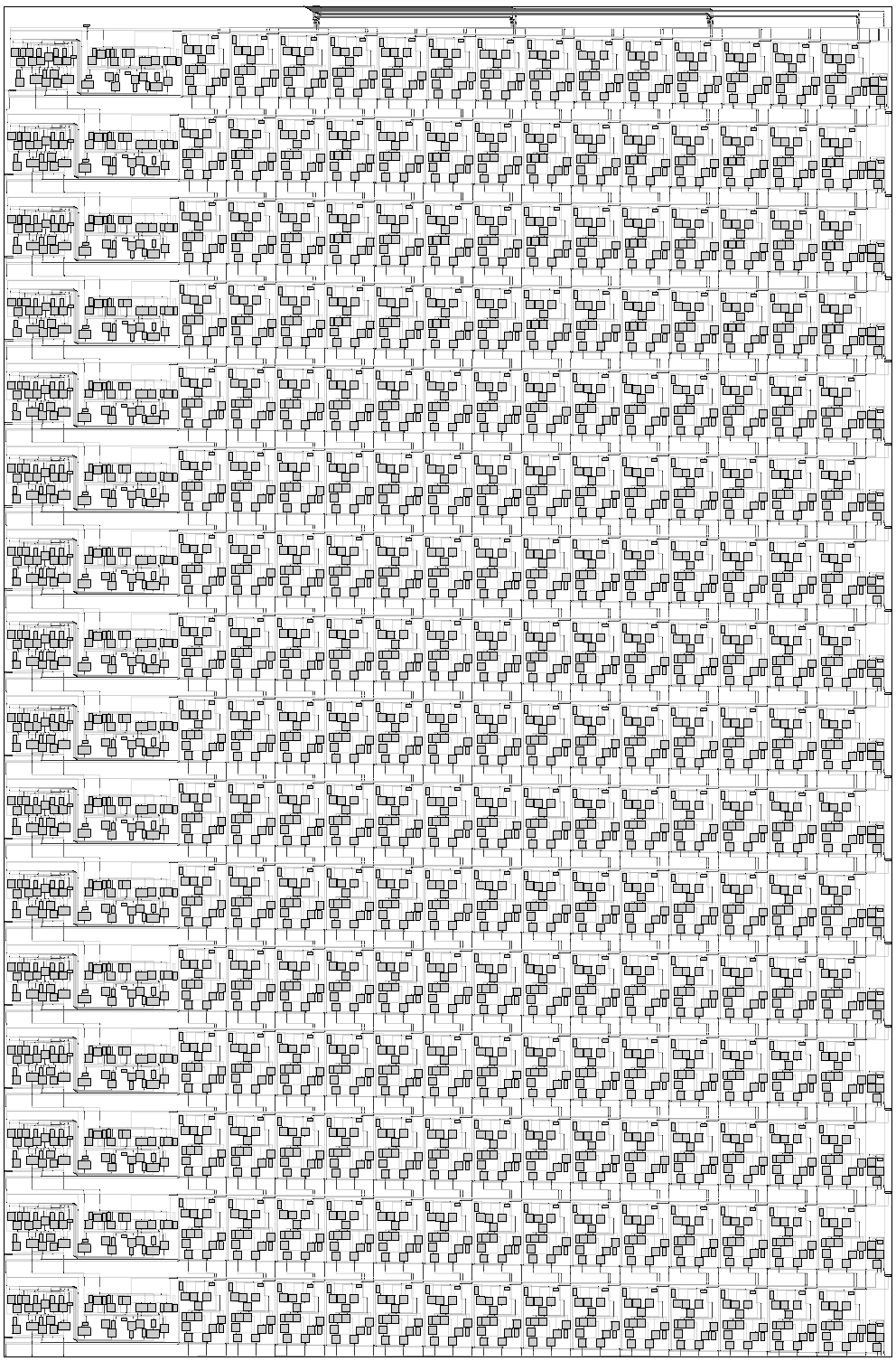

Figure 10.19: Layout for the 16 bit divider `Divider[16]`