Answer

a two layer swicthbox router

Verschelling, A.J.F.

Award date:
1991

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section (ES)

Answer :

A Two Layer Switchbox Router

by A.J.F. Verschelling

Master Thesis
Report on graduation work
from September 1990 to May 1991
by order of professor dr. -ing. J.A.G. Jess
and coached by ir. E.P. Huijbregts

Abstract

In this report a new two layer switchbox router is presented: Answer. To attain the goal of every switchbox router, namely connecting all necessary connections, a new routing algorithm is developed. First all nets are described as a set of intervals and the routing algorithm then makes the route by assigning the intervals to a suitable track. To improve the results obtained by the routing algorithm some modificationsteps are introduced. In these modifications intervals or vias of already completed nets are pushed away to create more routing space for the remaining connections, and a Lee router is used as a last option to complete the routing.

For all switchbox benchmarks presented in literature more than 90% of the necessary connections are made by Answer. Running times always are smaller than or equal to previous presented results.

Contents

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

An important technique in VLSI design is detailed routing. Detailed routing is usually done by two different routers : a channel router and a switchbox router. Both routers deal with a rectangular region, where a channel router allowes terminals to appear only on two parallel boundaries, while a switchbox router may have its terminals on all four sides. This report describes a new switchbox router : Answer.

Several two layer switchbox routers have been recently developed. Joobbani and Siewiorek [1] have developed Weaver, a succesful knowledge-based-expert-system router. Weaver, uses a 100% routability expert. Due to the many experts and several hundred rules the router is very complex and running time is very long. Therefore, it is not very efficient to use. Shin and Sangiovanni-Vincentelli [2] developed a router based on incremental routing modifications: Mighty. The modification steps they use were already presented by Dees and Karger [3]. The results they obtained were good and the running time was short, but one of the examples introducing the modificationsteps made us realize that the same results would be obtained using a more simple approach, based on channel routing techniques.

Beaver [4] is the name of a switchbox router developed by Cohoon and Heck, and it is the fastest router we have heared of so far. It uses three routing styles, a corner router, a line sweep router and a thread router, and the assignment of wires to layers is delayed as long as possible. A recently presented switchbox router is Carioca [5] developed by Dubois, Puissochet and Tagant. In this router the nets are devided in subnets before the routing is started. The order in which the subnets are routed is forced by a set of rules, among them some rules which have already been presented in [6]. One feature is important to notice : Carioca can handle terminals that are not on a grid on one set of parallel boundaries. The results of Carioca are good but running times are significantly longer than with Mighty or Beaver.

Although all these switchbox routers mentioned above have good results, we have developed a new switchbox router : Answer. The main reason was that, while studying the other routers, we felt that it must be possible to use a more simple and straightforward approach, obtaining the same results.

Chapter 2
The Routing Algorithm

The ultimate goal of a switchbox router is to interconnect a 100% of the necessary electrical connections. For the switchbox router described in this report a new routing algorithm has been developed in which a straight forward approach is used to gain speed and to avoid blockage. The nets are first divided into intervals of which the length and position depends on the position of the terminals on the boundaries. The algorithm then tries to complete the connections by assigning each interval to a suitable track and updating the orthogonal intervals to make the connection to the terminals.

2.1 Definitions and Problem Statement

A switchbox is a rectangular region with terminals on all four sides. A switchbox can therefore formally be described as a region $R = (Xdim*Ydim) = \{0,1,..,Xdim-1\}*\{0,1,..,Ydim-1\}$ where *Xdim* and *Ydim* are positive integers. The zeroth and *Xdim-1* columns are respectively the left and right boundaries of the switchbox. Similary, the zeroth and *Ydim-1* rows are respectively the top and bottom boundaries of the switchbox. This means that the grid point with coordinates (0, 0) is the left upmost corner of the switchbox.

To make sure that the terms used throughout this report will be well understood some definitions are given. We have tried to follow, as much as possible, the terminology used in previously presented papers.

- Routing area : The switchbox without its boundaries. Therefore, the routing area is the set of grid points $\{(x,y) \mid x \in [1,Xdim-2], y \in [1,Ydim-2]\}$.
- Row : $R_i = \{(x,y) \mid x \in [0,Xdim-1] \wedge y = i\}$ with $0 \le i \le Ydim-1$
- Column : $C_i = \{(x,y) \mid x = i \wedge y \in [0,Ydim-1]\}$ with $0 \le i \le Xdim-1$
- Track : A row or column of the switchbox.
- Begin point : Grid point (x_b, y_b) marking the begin of an interval.
- End point : Grid point (x_e, y_e) marking the end of an interval. Notice that for all intervals $x_e > x_b \wedge y_e > y_b$.
- Horizontal interval : The set of grid points $\{(x,y) \mid x \in [x_b, x_e] \wedge y = y_b = y_e\}$
- Vertical interval : The set of grid points $\{(x,y) \mid x = x_b = x_e \wedge y \in [y_b, y_e]\}$ There are two types of intervals :
    i. Free Interval : An interval with begin and end point both within the routing area.

ii. Boundary Interval : An interval with either begin or end point positioned on a boundary.

- Net : A set of intervals to be connected.
- Via : An interconnection of two intervals on two different layers belonging to the same net. A Via is placed at the gridpoint the intervals have in common.
- Constraints Graph *CG(V,E)* : A graph introducing a precedence relation between the nets on two parallel boundaries. Each net is represented by a node $v \in V$, and there is a directed edge from node v to node w if in a column c there is a terminal of v above a terminal of w. There are two Constraint Graphs for the switchbox.

  i. Top-bottom-CG : The constraints graph made over the top and bottom boundaries.

  ii. Left-right-CG : The constraints graph made over the left and right boundaries.

- Level of a net : The level of the net in the constraints graph. This is the topological place of the net in the CG. For a net i its level is denoted by $\mathcal{L}(i)$.
- Predecessor : Net i is a predecessor of net j if there is an edge in the CG from node i to node j, eg $\mathcal{L}(i)<\mathcal{L}(j)$.
- Successor : Net i is a successor of net j if there is an edge in the CG from node j to node i, eg $\mathcal{L}(i)>\mathcal{L}(j)$.
- Maxlevel : The highest level appearing in the CG.

The problem that has to be solved is : Connect all terminals of each net in the routing area such that each grid point is used by at most one net. The primary objective is to complete as much nets as possible, using an algorithm based on some straight forward routing ideas, such that the solution is similar to one that would be made if it were human work. The secondary objective is to complete the remaining connections by using some intelligent modification steps.

In the research done to develop such a switchbox router we have introduced some constraints :

- The switchbox consists of two layers.
- There are no blockages allowed either on the boundaries or within the routing region.
- The routing algorithm only uses the preferred layer to make a connection. Therefore, only the modification steps can force an interval to change layer.
- The preferred layer for vertical (horizontal) intervals is layer 0 (1). This means that a connection can exit a terminal in top or bottom only in layer 0. Similarly, a connection can exit a terminal in left or right only in layer 1.
- There are no cycles allowed in either one of the Constraint Graphs.

There are two points to be mentioned on these constraints. First, blockages can be introduced in a rather simple way, which will be discussed later on. Second, if a cycle appears in one of the constraint graphs, we can't find a precedence relation for

the nets on these boundaries. Therefore the program will stop as soon as a cycle is detected.

2.2 The Generation of Intervals

As mentioned in the Introduction, it was one of the examples of Mighty's modification steps [2] that made us realize that this kind of problem could be well prevented by using a channel routing like approach with its VCG [7] , because the VCG gives us the information which leeds to the same solution (fig. 2.1).

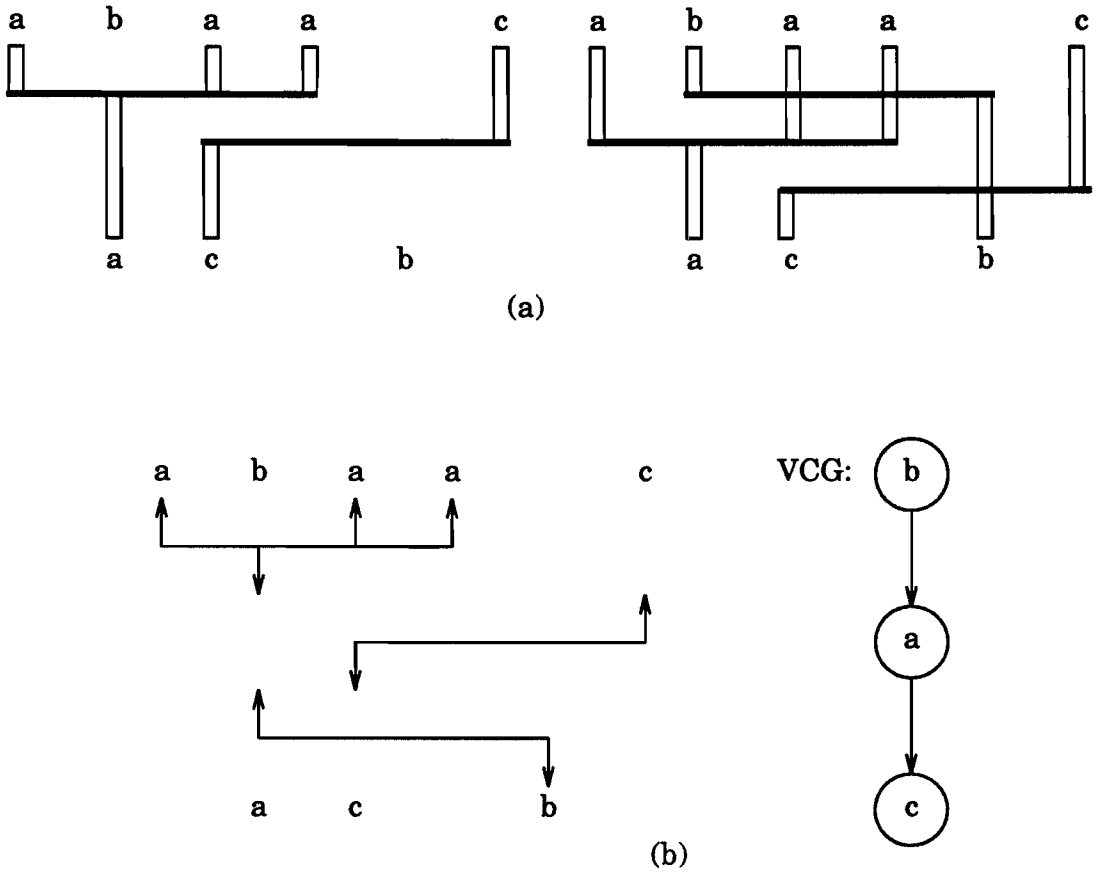


Figure 2.1. Example of Mighty's unit-push (a) and the channel routing approach (b)

Out of this the idea was born that perhaps the routing could be made by simply defining the right interval for each net, and searching for the right permutation over all intervals. However, we can't choose the intervals in the same way as it is done with channel routing (i.e. from the leftmost till the rightmost terminal of the net). The main reason for this is, of course, the fact that the terminals are on all four sides of the switchbox. Considering only the terminals on two parallel boundaries and then choosing the interval from the leftmost till the rightmost terminal doesn't seem a good idea either.

While developing the algorithm we wanted to make use of heuristics which would make a route such as also would be made when the routing would be human work. One of the routing patterns that always will be made by man work is a Corner

route. Another pattern which will always be choosen the same, is when there are two terminals on the same boundary or on two parallel boundaries. In this case we will always try to make a connection using only three intervals : One parallel with the boundaries from terminal position to terminal position, and two orthogonal with the boundaries making the connection with the free interval (fig. 2.2).

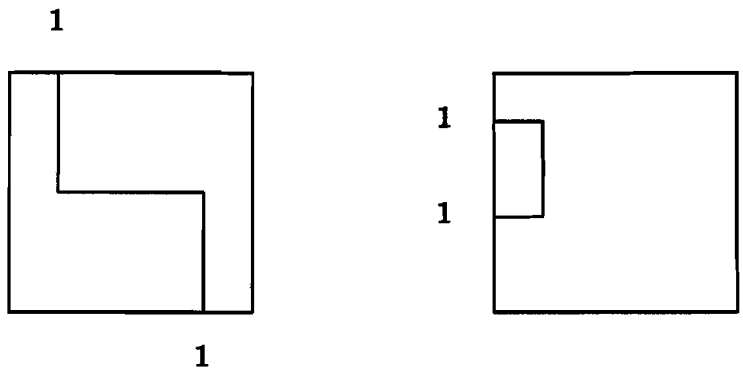


Figure 2.2. Connecting two terminals using a minimum number of intervals

Therefore, the definition of the intervals chosen is such that most Corner routes are made directely, and for the terminals on two parallel boundaries there is a free interval defined from each terminal to the closest terminal of the same net, and two boundary intervals at the terminal positions. This interval definition is done seperately for horizontal and vertical intervals.

For horizontal intervals this means that there are free intervals from terminal to terminal positioned on the top and/or bottom boundaries, and boundary intervals at the terminal positions at the left and right boundaries. Let us look at an example: Consider a switchbox $R = (Xdim*Ydim) = (6*6)$ with its terminals given per boundary (fig. 2.3a).

i. $Top \Rightarrow \{ (1,0),(3,0) \}$

ii. $Bottom \Rightarrow \{ (4,6) \}$

iii. $Left \Rightarrow \{ (0,2),(0,4) \}$

iv. $Right \Rightarrow \{ (6,1) \}$

Thus for the horizontal intervals there is a free interval to be defined $\{ (x,y) \mid x \in [1,3] \}$ and a free interval $\{ (x,y) \mid x \in [3,4] \}$ but we yet don't know in which track this interval will be placed and therefore is y undefined. There are also some boudary intervals to be defined. For the terminals in the left boundary we have two intervals $\{ (x,y) \mid y = 2 \}$ and $\{ (x,y) \mid y = 4 \}$. The begin and endpoints of these intervals have to be chosen, such that the smallest corners will be routed immediately. This means that -in this example- there is an interval $\{ (x,y) \mid x \in [0,1] \wedge y = 2 \}$ and an interval $\{ (x,y) \mid x \in [0,x_{und}] \wedge y = 4 \}$ where x_{und} means that we haven't defined that position yet. Of course, this is the same as $\{ (x,y) \mid (0,4) \}$ while only the begin point has been defined. For the terminal in right the same definition is used which means that there is an interval

$\{(x,y) \mid x \in [4,6] \wedge y = 1\}$.

Thus, there are two points to be mentioned on this definition : First, not every possible corner is always routed due to the algorithm but only the smallest one's, and second, the corner route is not made by some corner routing algorithm but it's simply a feature of the interval generation.

The complete interval definition is given in table 2.1 and figure 2.3b, where an arrow means that beginpoint, endpoint or track hasn't been defined for that interval yet.

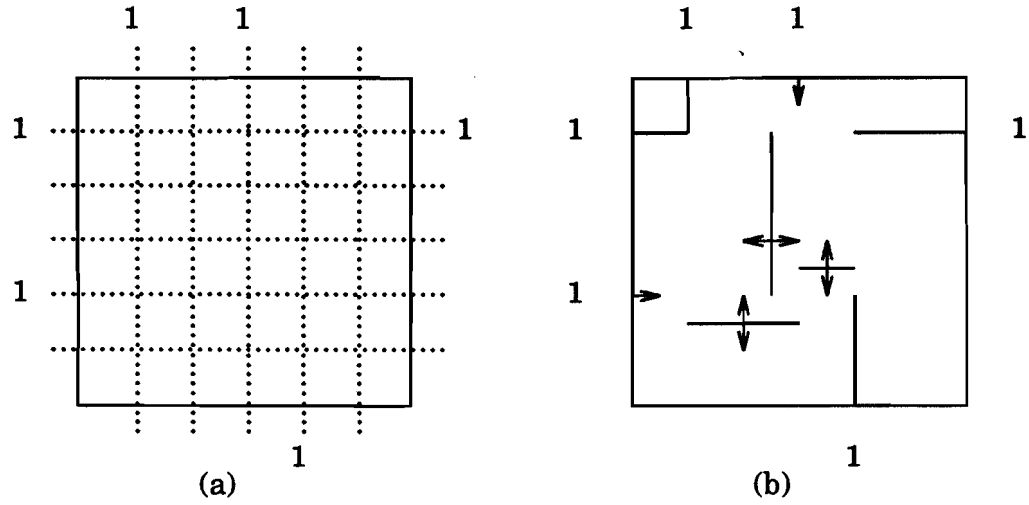


Figure 2.3. A switchbox example with its intervals

Table 2.1. Interval definition for figure 2.3

Horizontal intervals			
Interval	Beginpoint	Endpoint	Track
boundary	x = 0	x = 1	y = 1
boundary	x = 0	x = undefined	y = 4
free	x = 1	x = 3	y = undefined
free	x = 3	x = 4	y = undefined
boundary	x = 4	x = 6	y = 1

Vertical intervals			
Interval	Beginpoint	Endpoint	Track
boundary	y = 0	y = 1	x = 1
boundary	y = 0	y = undefined	x = 3
free	y = 1	y = 4	x = undefined
boundary	y = 4	y = 6	x = 4

An example of the interval definition for a well known switchbox, Burstein's difficult switchbox, is given in figure. 2.4. In this figure all boundaryintervals which have defined begin and endpoints are given. We can see that all nets consisting of only one corner route are already routed due to our interval definition. For all other nets

the smallest corners are already routed, due to this definition.

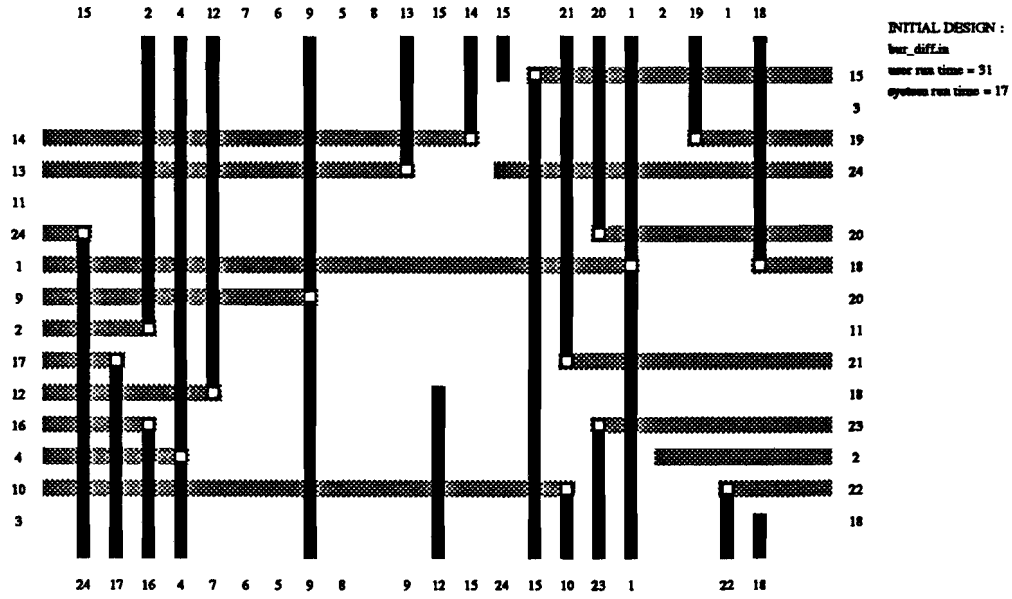
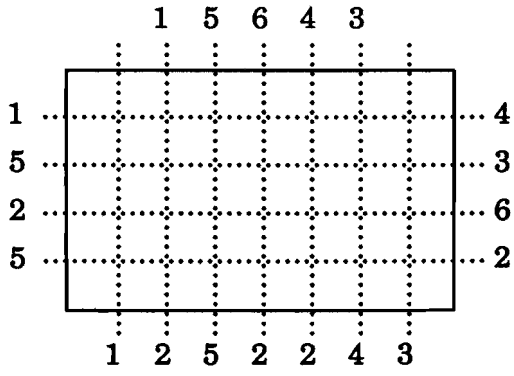


Figure 2.4. The results of the interval definition on Burstein's difficult switchbox

2.3 The Constraint Graph and its application

The constraint graph, as mentioned in paragraph 2.1, gives a precedence relation between the nets of the switchbox. To obtain this precedence relation, the CG in our switchbox router is defined in a similar way as the Vertical Constraint Graph in channel routing [7]. For the top_bottom_cg there is a precedence relation between net h and net i if they are present in the same column c. If, in column c , the terminal of net h is in the top boundary and the terminal of net i is in the bottom boundary, net h is a predecessor of net i, and thus net i is a successor of net h.

In figure 2.5 a switchbox example, with its two cg's is given.



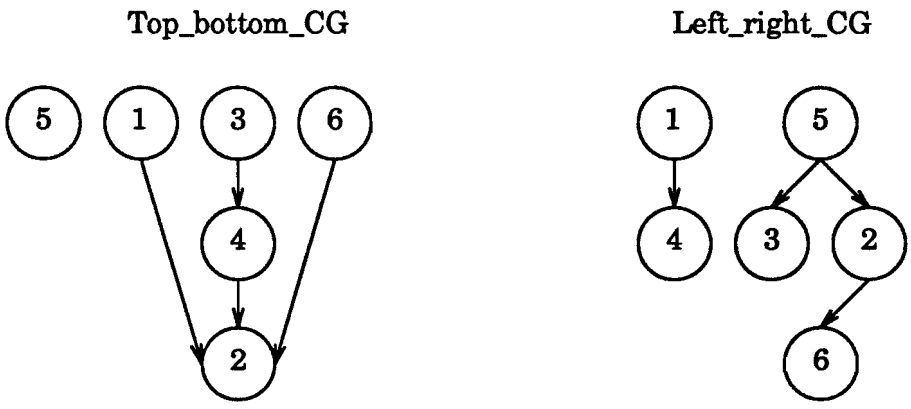


Figure 2.5. A switchbox example with its CG's

The Constraint Graph in this switchbox router will now be used to :

i. Find a netorder to make the route.

ii. Find a set of allowed tracks for each interval

2.3.1 Finding a netorder

One of the problems we have to cope with, is the question of which net to start with. In recent papers on switchbox routing some heuristics to answer this question have been introduced. For example, the simplest nets first [4] (i.e. straight lines and corners) or by checking if a possible shortest connection is feasible [2].

Common to the interval definition, we have two netorders : one for the horizontal intervals and one for the vertical intervals. According to the CG, every net is given a level which is common to the place of the net in the CG. Nets who have no predecessors in the CG have level zero. Nets who don't appear in the top_bottom_cg because they have no terminals on either top or bottom have also level zero. Similar, nets who don"t appear in the left_right_cg have level zero. For the nets in figure 2.5, this means that in the top_bottom_cg nets *1, 3 5* and *6* have level *0*. Net *4* has level *1* and net *2* has level *2*. In the left_right_cg nets *1* and *5* have level *0*, nets *2, 3* and *4* have level *1* and net *6* has level *2*. The nets will be ordered in increasing level. First all nets of level zero, the all nets of level one an so on. For nets with the same level, the priority is simply given to the net with the lowest netnumber. The netorderings for the example will then be :

i. *Horizontal_netorder = [1, 3, 5, 6, 4, 2]*

ii. *Vertical_netorder = [1, 5, 2, 3, 4, 6]*

These netorderings will be used when routing the switchbox.

2.3.2 Finding a set of allowed tracks

In the previous paragraphs we have generated the intervals and a netordering for the routing algorithm. In this routing algorithm we have to assign each free interval to a track. However, not every track in the routing area is a suitable track. Therefore, we define a set of allowed tracks for an interval before we start searching for a suitable track. This set of allowed tracks can be found using the cg, because the set of allowed tracks is the set of tracks inbetween the tracks occupied by nets with

a lower level and those with a higher level.

The set of nets occupied by nets with a lower level will be easy to obtain because these nets have all been routed before the current net. This means that we can simply check on overlap problems with the nets of a lower level, by starting out from track 1 as a lower bound for the intervals of the current net.

For the set of tracks occupied by the nets with a higher level, only the tracks used by the boundary intervals are known. From these boundary intervals, only the ones that have overlap with the current interval need to be taken into account. Therefore the Maxtrack for the current interval is given by the lowest track of all boundary intervals of the nets with a higher level which have an overlap with the current interval. In the algorithm this overlap is tested by $x_{j,b} \leq x_{i,e} \wedge x_{j,e} \geq x_{i,b}$, where i is the current net and $j \neq i$.

But, not all nets with a higher level will have boundary intervals. For example, if a net has only two terminals, one in top and one in bottom, than there is a horizontal interval but there are no boundary intervals who have already been assigned to a track and therefore we can't destilate a Maxtrack for the current interval. In this case the level of the nets is used. Maxtrack is initialy set to (Ydim - 2), i.e. the highest possible track of the switchbox. The routing space (i.e. the number of tracks) needed for the nets with a higher level after we have routed the current net, is equal to difference between the Maxlevel and the level of the current net. The right value for Maxtrack is now given by (Ydim - 2) minus this difference in level. The algorithm, finding the maxtrack of an interval, is given below.

Algorithm 2.1. Find_maxtrack()

```
void find_maxtrack(i,(x_b,y_b),(x_e,y_e) )
{
  Maxtrack = Ydim - 2;

  for ( all j ≠ i | L(j) > L(i) )
    for ( all intervals of j | track(j)  ≠  Undefined )
      if(x_{j,b}≤x_{i,e}∧x_{j,e}≥x_{i,b})
        Maxtrack = MIN ( ∀ track(j) )

  if ( Maxtrack == (Ydim - 2) && L(i)  ≠  Maxlevel ){
    Levelshift = Maxlevel - L(i);
    Maxtrack -= Levelshift;
  }
}
```

Algorithm 2.2. Make_route()

```
void make_route( i )
{
  for ( all free intervals of i ) {
    Maxtrack = find_maxtrack(i,(x_b,y_b),(x_e,y_e) );

    for ( 1 ≤ t ≤ Ydim-2 )
      if ( interval already contained by i in t )
        track(i) = t;
        break;

    for ( 1 ≤ t ≤ Maxtrack ) {
      if ( t already used by i ){
        if ( no_overlap() && no_via_problem() ){
          weight = Xdim;
          put_in_tracklist( track, weight );
        }
      }
      else                    /* track not used before by net i */
        if ( no_overlap() && no_via_problem() ){
          weight = x_e-x_b + 1;
          put_in_tracklist( track, weight );
        }
    }
    track of interval = assign_track();
    update_ver_intervals();
  }
}
```

2.4 The main algorithm

The main algorithm which makes the actual route, named make_route, is given in algorithm 2.2. It is given for the routing of the horizontal intervals. The routing algorithm of this switchbox router searches a suitable track for each free interval and makes the connection to the terminals with update_(ver_)intervals(). Update_intervals() thus is a procedure which extends the boundary intervals. This means that the begin or end positions of the boundary intervals, placed at a track equal to the begin or end position of the routed free interval, are updated such that they make a connection with the free interval.

The second for-loop is the real routing algorithm. The allowed set of tracks is given by $1 \leq t \leq$ Maxtrack, where the tracks occupied by the nets with a lower level are automaticaly skipped, due to the tests no_overlap() and no_via_problem().

No_overlap returns FALSE if assigning the interval to a certain track will cause a shortcut.

No_via_problem() returns FALSE if the other layer is not free at one of the terminal positions (begin or end point), i.e. no connection from the interval to the terminal can be made. These two tests will find all tracks occupied by the nets with a lower level.

For example, consider two nets *h* and *i*. Net *h* has level zero and net *i* is a successor of net *h*, an has therefore level one. This also means that there is a terminal of net *h* in top and a terminal of net *i* in bottom in a column *c*. Net *h* will be routed first, say in track 1. Routing net *i* will find an overlap in track 1 and skip it. If net *h* is routed in track 2, track 1 may still be empty. For the routing of net *i* again we start in track 1 and no overlap will be found. But, in column *c* net *h* is connected to it's terminal in top, due to update_intervals(). Therefore, the other layer will not be empty, at the begin or end position of net *i* (i.e. column *c*) This results in a via problem and track 1 is skipped. Track 2 will also be skipped due to an overlap problem. Another possibility would be searching for a Mintrack common to Maxtrack. This could, of course, be done by a reverse procedure of find_maxtrack().

Now we must find the most suitable track out of the set of allowed tracks. First we define a weight for the current interval. While calculating this weight two different cases are considered :

i. The track is already used by the same net, which means that there are more intervals of the same net in this track, or there is a terminal of the net on this trackposition. In this case the weight of the interval is set equal to the length of the track (Xdim or Ydim) to make sure that this track has precedence over the other tracks.

ii. The track is not used before by the same net. In this case the weigth of the interval is set equal to its length.

Each track is then also given a weight which is the number of already occupied gridpoints of that track plus the weight of the current interval. If a track may not be used because it doesn't belong to the set of allowed tracks, or due to an overlap or via problem, its weigth is set zero. In assign_track() the interval is now assigned to

the track with maximum weight. If all tracks have weight zero, the interval will not be assigned to any track. This means that for the interval the set of allowed tracks is empty. The interval then still has an undefined track which means that we did not complete the routing.

The first for-loop in the algorithm is not only working on the set of allowed tracks but on the complete set of tracks of the switchbox. In this loop we are testing if an interval is contained in the already existing route of the same net. In figure 2.3 is a switchbox where such a situation can occur is shown In this switchbox first the horizontal free intervals are routed and the vertical boundary intervals are updated. Thus the interval $\{(x,y) \mid x \in [1,3]\}$ is first routed and we find track 1 (i.e. for y == 1) already used by the same net. Due to the boundary intervals in track y == 1, the weight for this track will be the maximum weight over all tracks and therefore the interval will be assigned to track 1. The interval is now given by $\{(x,y) \mid x \in [1,3] \wedge y \equiv 1\}$. Now we must update the vertical intervals which means that the vertical boundary intervals placed in a track equal to the begin or end position of the horizontal free interval have to be updated. Thus the intervals $\{(x,y) \mid x \equiv 1 \wedge y \in [0,1]\}$ and $\{(x,y) \mid x \equiv 3 \wedge y \in [0,y_{und}]\}$ have to be updated. The first interval doesn't need updating, because y_e is already equal to the track of the horizontal free interval. For the second interval y_e (which is y_{und}) is set equal to the track of the free interval and thus $y_e = 1$. The next horizontal free interval $\{(x,y) \mid x \in [3,4]\}$ will now be routed, and for the same reasons track 1 will be chosen. The interval is thus given by $\{(x,y) \mid x \in [3,4] \wedge y \equiv 1\}$. The only interval to update is the interval $\{(x,y) \mid x \equiv 4 \wedge y \in [4,6]\}$. For this interval y_b is not equal to the track of the free interval, so $y_b = 1$. The routing of the horizontal intervals is now completed. Now the vertical free intervals are routed. The only vertical free interval $\{(x,y) \mid y \in [1,4]\}$ is already contained in the route due to the last vertical interval update. Therefore, the interval is assigned to this track (i.e x == 4), so the interval is given by $\{(x,y) \mid x \equiv 4 \wedge y \in [1,4]\}$. The horizontal boundary intervals are updated which means that for the interval $\{(x,y) \mid x \in [0,x_{und}\} \wedge y \equiv 4\}$ $x_e = 4$. In figure 2.6 the switchbox with the intervals and the completed routing is given.

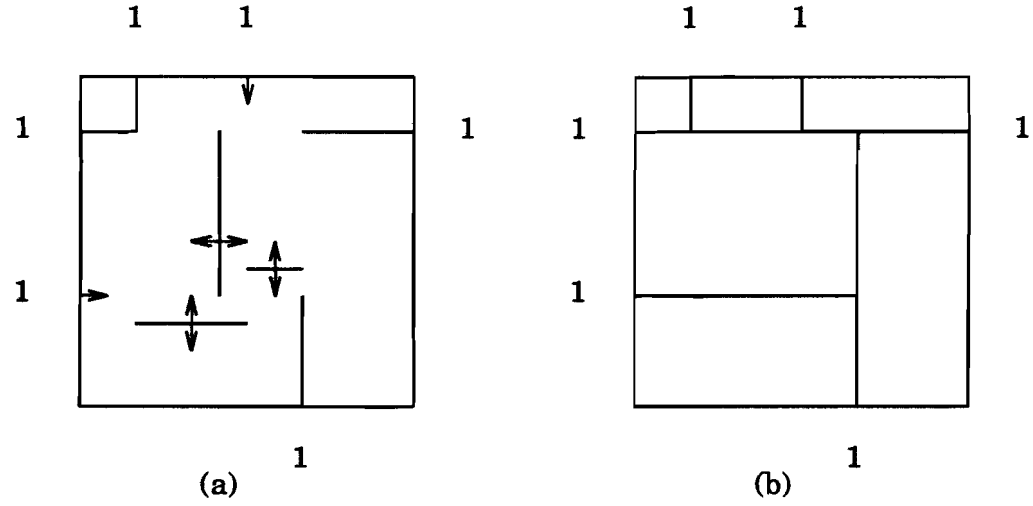


Figure 2.6. A switchbox example with interval definition (a) and routing solution (b)

Using this routing algorithm a solution is found according to the first objective in the problem statement namely, making a route which would also be made when routing would be human work. By giving a higher weight to the tracks which have already been used by the same net, we are sure that several intervals of the same net will be routed in one track. By assigning the interval to the track with the highest weight, we are sure that the tracks will be filled as much as possible and there will be as much as possible routing space left in the other tracks for the remaining nets.

The results of this routing algorithm on three benchmarks, Burstein's difficult switchbox, the pedagogical switchbox and the modified dense switchbox, are given in figures 2.7, 2.8 and 2.9. The results are hopeful, because most of the connections are made and the cpu times, which are given in units of 1/60 of a second (!), are very short. One thing is important to notice: although all terminals are positioned on the grid, the routing algorithm does not use the grid to make the route but it only handles the data stored in the lists of intervals. The data structure used to handle this information will be discussed in chapter 4.

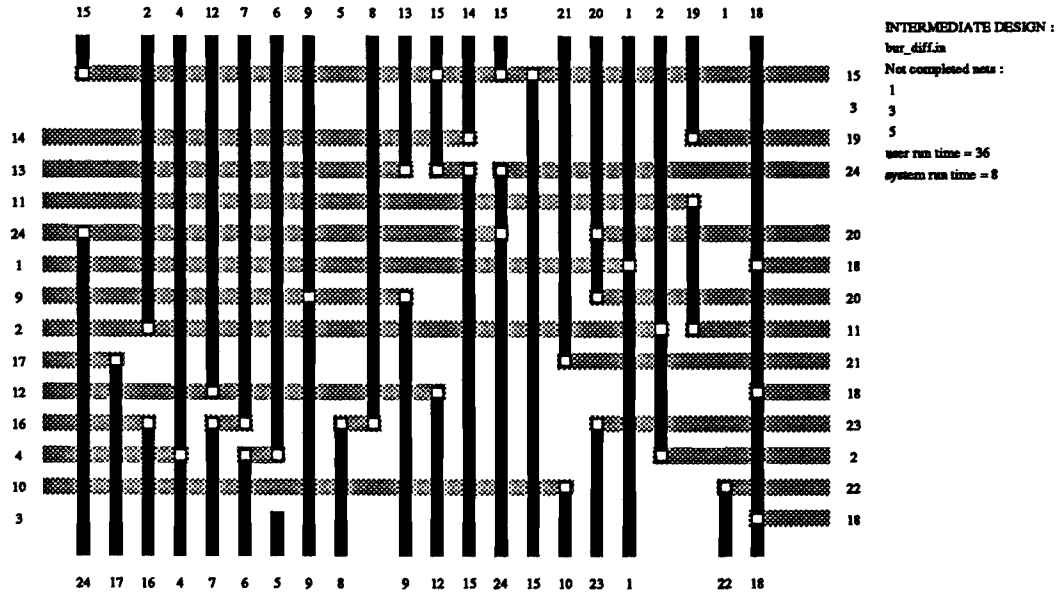


Figure 2.7. Results of the routing on Bursteins difficult switchbox

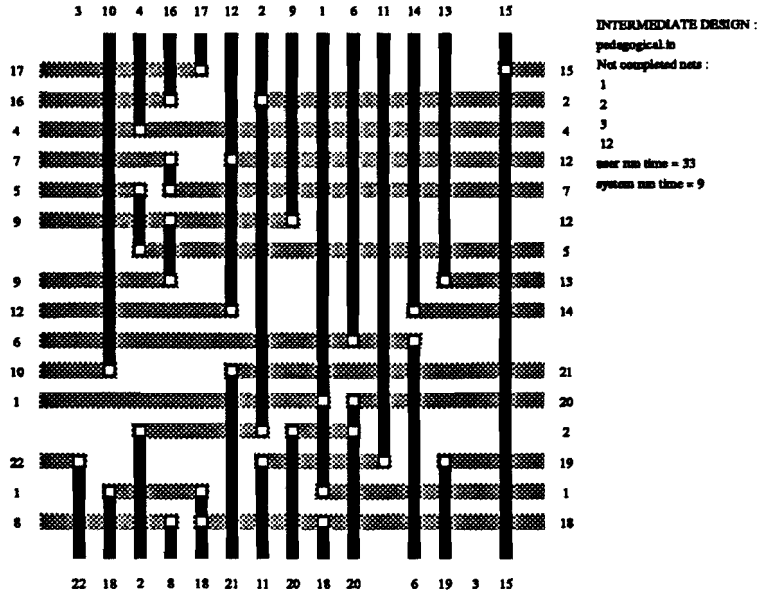


Figure 2.8. Results of the routing on the pedagogical switchbox

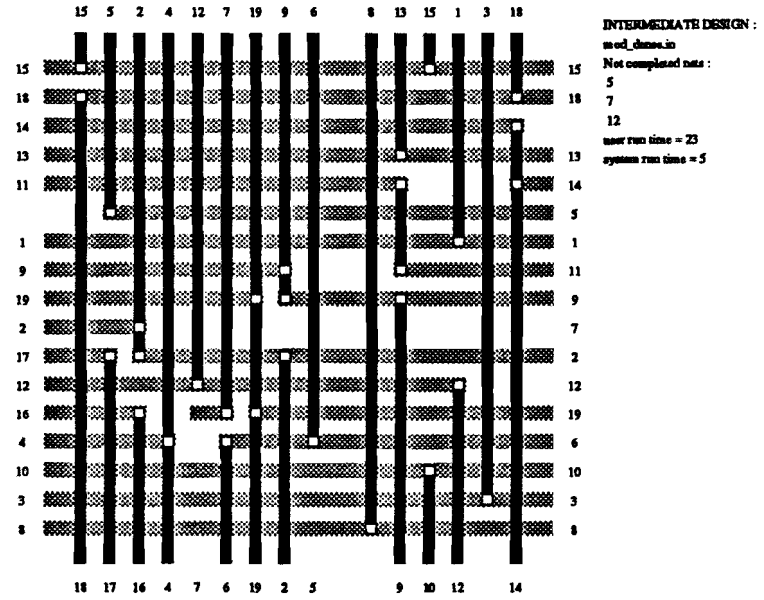


Figure 2.9. Results of the routing on the modified dense switchbox

2.5 Introducing blockages

In the switchbox constraints (paragraph 2.1) we said that no blockages were allowed but they could be introduced later. Introducing blockages is important because it makes that the switchbox router is applicable to a more general kind of detailed routing problems.

In this switchbox router we can describe a blockage as a set of intervals with fixed begin- and endpoint, track and layer. When there are N nets we give these

intervals of the blockage netid *N+1*. This makes that we can describe every possible shape of blockage. Introducing these blockages we have to consider two different situations :

i. The blockages are completely within the routing area or the blockages are on the boundaries and the terminals are still only on the boundaries.

ii. The blockages may be on the boundaries and the terminals that were on those positions are propagated into the routing area.

In the first case we only have to generate a set of intervals out of each blockage. Appending these intervals to the lists of intervals of all nets will make that, due to no_overlap, a track with a blockage on it will be skipped during the routing.

In the second case the idea is the same but the implementation is somewhat more difficult. Because a blockage on a boundary may then also mean that not all terminals are in the same row or column anymore. This means that irregular shaped switchboxes, L-shaped channels etc. are just rectangular switchboxes with blockages on the boundaries. For such switchboxes however, our interval generation will be somewhat more complex.

Thus, introducing blockages in this switchbox router is possible using the intervaldefinition but untill now we have not implemented this possibility.

Chapter 3
Modifying the Route

In the previous chapter an algorithm for switchbox routing has been introduced. The results are hopeful, and due to the straight forward approach, the runtime is very short. The examples (fig. 2.7, 2.8 and 2.9) illustrating the results show us that we did not interconnect a 100% of the necessary electrical connections. We can try to get a better solution by appending more heuristics to the routing algorithm, or we can modify the route already made. The latter option is chosen. This chapter will describe what kind of problems were left after the routing was done, and what kind of modifications we introduced to solve them.

3.1 Updating the switchbox problem

Before we can start modifying the route, we have to know which part of our initial problem we have solved and which part not. In other words, we need to know which nets are finished and which nets are not. This testing whether a net is finished or not is done by simply starting out a dfs routine at one of the terminals and checking if all other terminals of the same net are reached.

The data contained in the intervallists and in the grid will be updated. For the finished nets, this means that we make sure that no parts of the route are described in more than one interval. For the unfinished nets, this means that dangling ends are removed from the grid, and the intervals, which were part of the dangling end, are given their initial values again.

Using the knowledge on which the routing algorithm was based, the problem statement is reduced to :

i. There is a free interval which hasn't been assigned to a track yet (nets 3 and 5 in example 2.7). This may occur when the set of allowed tracks for the interval was found to be empty.

ii. There is a boundary interval which doesn't make a connection with a free interval (nets 1 and 12 in example 2.8). This may occur when the set of allowed tracks was not empty, but there was an overlap problem in the track of the boundary interval.

The modificationsteps that will be described in this chapter have all been implemented for horizontal as well as vertical intervals. To give one general description a new interval definition will be given.

Definition 3.1 An interval I is a fourtuple $I = \{T,L,B,E\}$ where T denotes the track, L the layer, B the beginpoint and E the endpoint of the interval. An interval of net i will be given by $I_i = \{T_i, L_i, B_i, E_i\}$. For the interval L_i means that the interval is placed in the preferred layer and $\bar{L}_i$ means that the interval is placed in

the nonpreferred layer.

3.2 Modify_route

Modify_route() is the algorithm introducing the modification steps needed to solve the problems that are left after the routing phase. In the previous paragraph the problem statement has been updated leaving two kinds of problems.

In the first case we need to create more routing space by assigning intervals of already finished nets to another track. This is done by pushing the intervals of those already finished nets away (intervalshift).

In the second case again we need to create more routing space and try to extend the boundary interval towards the free interval, or try to make a connection towards the nearest terminal of the same net on the same boundary (intervalextend).

To create more routing space, we can try to minimize the number of via's. This via minimization will create more routing space in the nonpreferred layer and will also avoid blockages. We can use this routing space in a last attempt, trying to complete the routing also using the nonpreferred layer for a free interval. The algorithm modify_route() is thus given as follows.

Algorithm 3.1. Modify_route()

```
void modify_route()
{
    intervalshift();
    intervalextend();
    via_minimization();
    last_attempt();
  }
}
```

3.2.1 Intervalshift

For all unfinished nets, again we first consider the free intervals which haven't been assigned to a track yet. Such an interval, for example, exists for net 5 in figure 2.7. This net has an interval $\{ (x,y) \mid x \in [6,8] \wedge y = y_{und} \}$. In the switchbox, every track is occupied or there will be a via problem. However, if we would assign the free intervals of nets 6 and 7 to a track closer to the top boundary and the free interval of net 8 closer to the bottom boundary, then there would be enough routing space for net 5. The questions that remain are : How general is this approach, and how do we find the blocking nets?

Due to our routing algorithm using the CG, we can easily obtain the blocking nets. Because the problem was that the set of allowed tracks for the interval seemed to be empty, the only solution is to enlarge this set. Enlarging this set means also making smaller the set of not allowed tracks. In other words, we have to make smaller the set of tracks occupied by the nets with a lower level and the set of tracks occupied by the nets with a higher level. This means that the nets to be pushed away are the direct predecessors and direct successors of the net in the CG. The

algorithm for intervalshift is given below.

Algorithm 3.2. Intervalshift()

```
void intervalshift()
{
  for ( all free intervals of an unfinished net i)
    if ( track == undefined )
      for ( all direct predecessors of i)
        if ( ! up( i,pred. ))
          for ( all direct successors of i)
            down ( i,suc. );
}
```

First we try to push up the direct predecessors. This means that for every free interval I of these direct predecessors we try to assign the interval to a track t with $t = T_{pred}-1$. Pushing up the predecessor is done in up(). In this procedure up() we first try to push up the predecessor in the preferred layer, by calling tryup(), and then we try to route the free interval of net i in try_(hor_)route. Try_(hor_)route tries to assign a free interval of net i to a track in a similar way as make_route() (described in the previous chapter). This means that try_(hor_)route will assign an interval to a track only in the preferred layer. If this isn't possible, try_(hor_)route will return FALSE. When try_(hor_)route returns FALSE, this means that we did not create enough routing space, and we try to push up the predecessor in the non preferred layer.

The algorithm for up() is given below.

Algorithm 3.3. Up()

```
int up( i,pred )
{
  tryup( pred., preferred layer );
  if ( !try_hor_route(i) ) {
    tryup( pred., nonpreferred layer );
    return( try_hor_route(i) );
  }
  return( TRUE );
}
```

If we still can't make the route for net i we try to push down the sucessors of net i. The procedure down() trying to push down these successors is similar to the procedure up(). For the vertical intervals, of course, there is a similar approach with the procedures left() for predecessors and right() for successors.

Tryup() is the procedure that is realy pushing up an interval. If an interval of a net is pushed up it is also marked to prevent oscillations during the modification. Every unfinished net is also marked. Tryup() will push an interval under two conditions :

i. The track and layer to which we want to push the interval have to be empty.

ii. The interval has to be modifyable.

Definition 3.2 An interval of net j is modifyable if :

i. It is a free interval.

ii. Net j is not marked.

iii. $L(j)<L(i)$ for tryup and tryleft and $L(j)>L(i)$ for trydown and tryright, where i is the unfinished net.

When an interval is modifyable there might still be a problem because the track to which we want to shift the interval may not be empty. In the preferred layer this often means that there is another horizontal interval placed in that track. If there is only one net causing this blockage we try to push this blockage first. In the nonpreferred layer such a blockage might be caused by a vertical interval taking control of that track. The blockage is not modifyable in that case. It might also be caused by another horizontal interval which was already assigned to the nonpreferred layer before. The blockage might be modifyable and will be pushed away when possible. The blockage on the nonpreferred layer might also be a via. In this case the via is tried to push by one grid point. This via pushing is done while testing if the grid is empty for the interval at the desired position.

When pushing up a predecessor or a blocking net is succesfull, the net is tried to be pushed up as far as possible by repeating the procedure for the same net. The algorithm for tryup() is given below.

Algorithm 3.4. Tryup()

```
void tryup(j,L) /* j is a predecessor or a blocking net */
{
    Mark( j );
    for ( all free intervals of j )
      if ( grid empty for I = {Tj-1,Lj,Bj,Ej} )
        Tj = Tj-1;
        tryup(j,L);
      }
      else
        if ( there is a single blocking net k )
          if ( !Marked(k) && L(k) < L(j) )
            tryup(k,L);
}
```

3.2.2 Intervalextend

After we have tried to route the remaining free intervals there may still be boundary intervals b with $B_b = und$ or $E_b = und$. In other words there are still some terminals not making any connection with any other part of the route (for example, nets 1 and 12 in figure 2.8). To make a connection with the remainig part of the route, we have to extend these boundary intervals towards a part of the route of the same net. The question is how, or in which direction, the extension should be made.

When there is a terminal of i on a boundary, and we have such a boundary interval b with $B_{b,i} = und$ or $E_{b,i} = und$, then we can have two different cases. The first case is that there is a free interval f with $B_{f,i} = T_{f,i}$ or $E_{f,i} = T_{f,i}$. In this case the interval has to be extended towards the free interval. However, no routing space exists and therefore we need to create more routing space by pushing the blocking nets away. This pushing away of the blocking nets is similar to the pushing in intervalshift().

The second case is when there is no $B_{b,i} = T_{f,i}$ or $E_{b,i} = T_{f,i}$ or when the blocking nets could not be pushed away. In this case we have to extend the interval towards a part of the route in another track. We then look for the nearest terminal of the same net on the same boundary. If there is such a terminal, a new interval I: $\{T = und, L, B = T_{term1}, E = T_{term2}\}$ is appended to the interval list, and this interval will be routed using try_(hor_)route(). If this routing is succesful, the boundary interval with undefined begin or endpoint will be connected to the free interval by update_(ver_)intervals(). The algorithm for intervalextend is given below.

Algorithm 3.5. Intervalextend

```
void interval_extend(i)
{
  succeed = FALSE;

  if ( x_b == Und || x_e == Und ){
    if ( grid empty for I = {T_i,L_i,B_i,E_i} )
      succeed = extend interval();
    else
      if ( there is a single blocking net k ) {
        tryup( k );
        if ( grid empty for I = {T_i-1,L_i,B_i,E_i} )
          succeed = extend interval();
        else
          trydown( k );
          if ( grid empty for I = {T_i+1,L_i,B_i,E_i} )
            succeed = extend interval();
      }

    if ( !succeed ){
      find_nearest_terminal();
      define new interval;
      try_route( new interval );
    }
  }
}
```

3.2.3 Via_minimization

In the previous modification steps the most important feature was creating more routing space. Removing blockages is also a method to create routing space. So, if we are able to minimize the number of vias we are also minimizing the number of blockages, because a via uses two layers where an interval uses only one layer.

Minimizing the number of vias can be done by assigning some intervals to another layer. What we need to know is for which intervals a change of layer will result in a decreasing number of vias. Of course, the interval needs to be a free interval because, due to our switchbox constraints, a boundary interval cannot change layer. For the free intervals we only try to change layer when the interval is still in the preferred layer. This constraint is made because a free interval may already have changed layer, due to the modifications, and a change of layer will then only cause an increasing number of vias. To be sure that changing the layer will cause a decreasing number of vias, free intervals connected to the parallel boundaries only, may change layer. So, for example, when there is a horizontal free interval next to a horizontal boundary interval of the same net, the free interval may not change layer because there will still be a via needed to keep the connection with the boundary interval.

The algorithm for via minimization is given below.

Algorithm 3.6. Via_minimization

```
void via_minimization()
{
  for ( all free intervals of net i in the preferred layer )
    if ( gridpoint(B_i-1, T_i) ∉i && gridpoint(E_i+1, T_i) ∉i )
      if ( grid empty for I = {T_i-1, L̄_i, B_i, E_i} )
        L_i = L̄_i;
}
```

3.2.4 Last_attempt

In last_attempt() again we try to route the free intervals I which still have $T = und$, making use of the routing space created by the via minimization. We try to route the interval in both layers. When trying to route the interval in the nonpreferred layer we try to remove blocking nets. A blocking net again will only be removed when it is modifyable according to definition 3.2.

Fig. 3.1 shows an example of a switchbox where last_attempt has been used to

complete the routing.

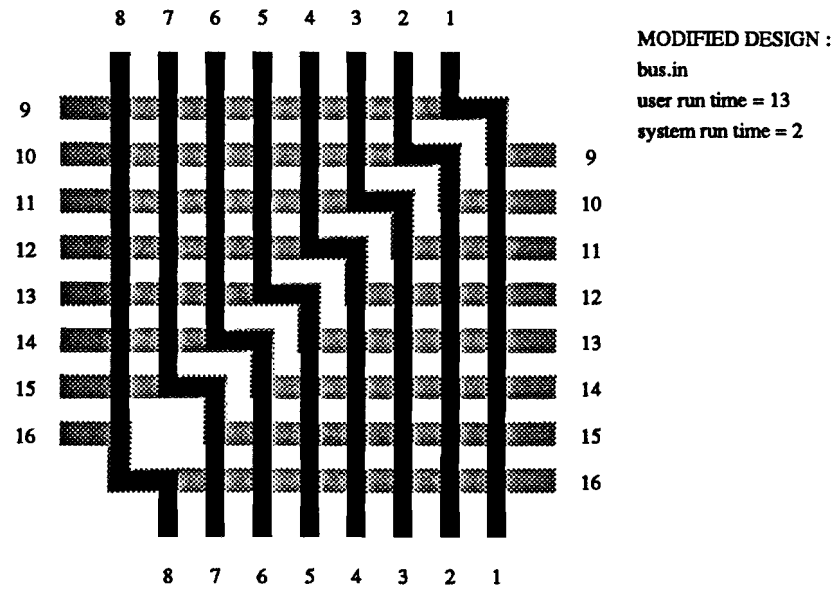


Figure 3.1. A switchbox example of two crossing busses with its solution

The results of Burstein's difficult switchbox and the pedagogical switchbox, using the modificationsteps described in this chapter, are given in figure 3.2 and 3.3. In figure 3.2 we see how intervalshift has created routing space to finish the routing of net 5. The modificationsteps however, didn't create enough routing space for net 3 so we could not complete the entire switchbox. In figure 3.3 we see how intervalextend introduced a new interval to complete the routing of net 12 but again we could not complete the entire switchbox.

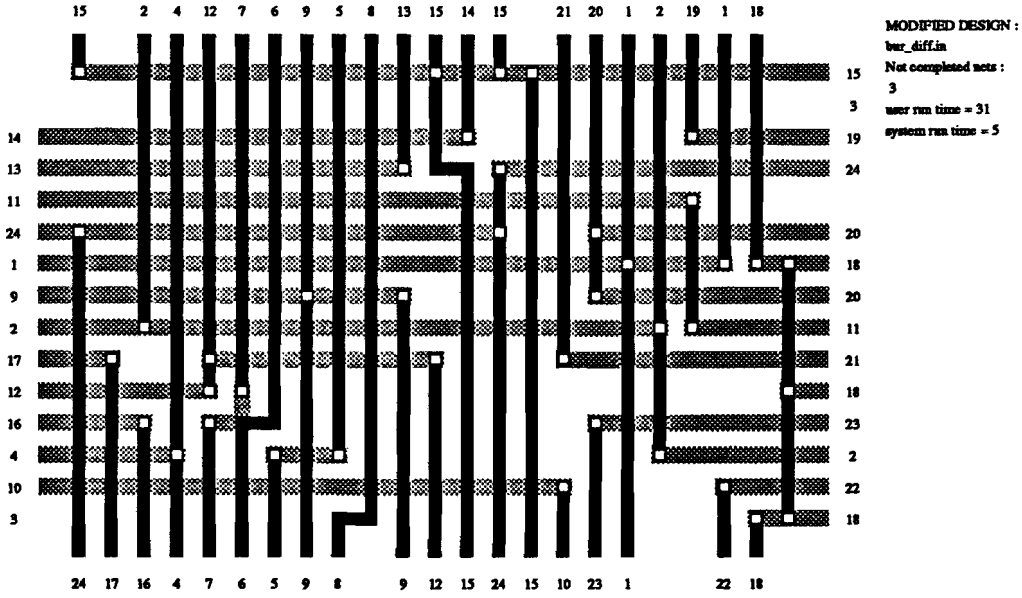


Figure 3.2. The solution on Burstein's difficult switchbox after modify_route()

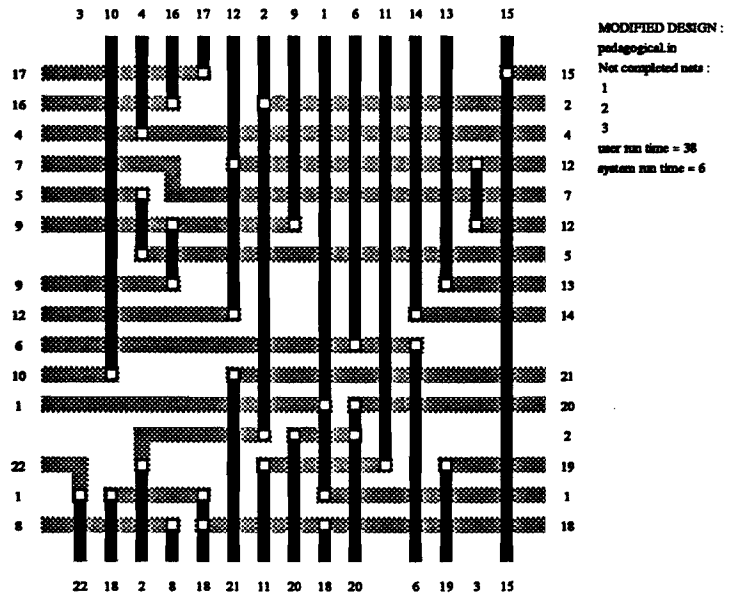


Figure 3.3. The solution on the pedagogical switchbox after modify_route()

3.3 Lee routing

In figure 3.4 the results obtained on the modified dense switchbox are given. An unfinished net was net 5 (fig. 2.9). The problem for this net was a boundary interval I with $B = und$. Although the modificationsteps have been introduced, this net is unfinished because the blocking nets are not modifyable and there is no other terminal of net 5 on the same boundary.

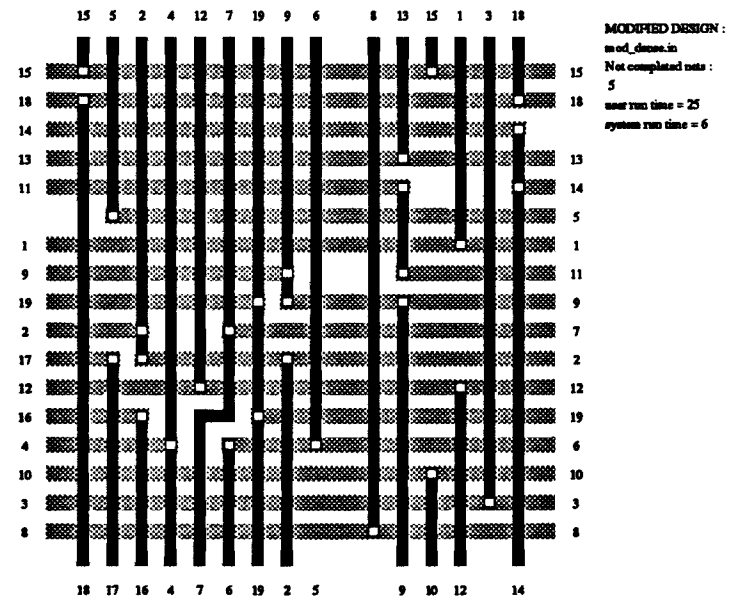


Figure 3.4. The solution on the modified dense switchbox after modify_route()

In figures 3.2 and 3.3 there were also some unfinished nets, for example net 3 in figure 3.2. For this net 3 there is a vertical free interval $\{(x,y) \mid y \in [2,15]\}$ which cannot be routed. Whatever modification we use, this interval is so large that there will never be enough routing space. Working forth on the same basics we perhaps could split the interval and try to route the two parts independently. But, where do we split? In the middle of the interval? Or should we split the interval in more than two parts? And, if splitting once doesn't find a solution, do we split again? Many of these questions can be asked for, and no good answer will be found.

A more general approach to this problem is splitting the interval in intervals of unit length, and applying the routing algorithm on these intervals. However, making a route in steps of unit length is almost the same as making a route using some kind of modified Lee router [8,9]. Therefore, we have chosen to append a modified Lee router to our switchbox router.

Lee's algorithm for path finding was one of the first algorithms used for switchbox routing. Because it is a rather slow algorithm, many modifications of the algorithm exist, such as starting out from two terminals instead of one. The line search algorithms and minimum Detour algorithm [9] are also based on Lee's algorithm. The Lee router in this switchbox router is also a modification of the original algorithm.

The Lee router developed for this switchbox router is starting out from all terminals of the net. A path is searched and, as soon as a connection between two terminals is found, the dangling ends of that path are removed. All grid points belonging to the connection just found are appended to the list of startpositions. From these startpositions, and the already existing search trees, the algorithm is continued until the next connection is found. Again, only the dangling ends of the path just found are removed and so on untill all terminal are connected, or no other connection can be found anymore.

With this Lee router we can now try to complete the remaining connections. The result for the modified dense switchbox is given in figure 3.5.

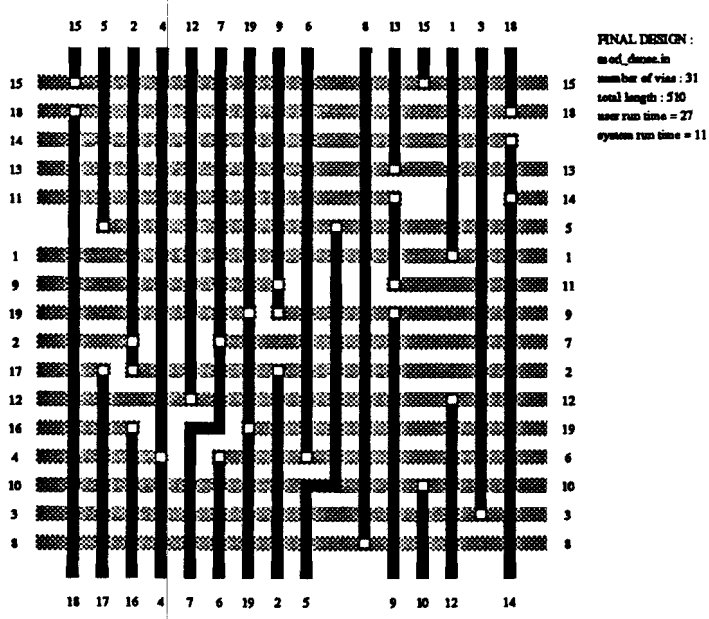


Figure 3.5. The final solution on the modified dense switchbox

The switchbox router is now complete and for the example shown above we are able to realize all connections. The results on some other benchmarks will be discussed in chapter 5.

3.3.1 Introducing Rip-up and Reroute

The example of figure 3.5 shows a switchbox for which the routing has been completed. However, there are also switchboxes for which we can not complete the routing. The results obtained on Burstein's difficult switchbox, after we have tried to route the remaining nets using the Lee router, are given in figure 3.6. This figure shows that the Lee router also failed in routing net 3.

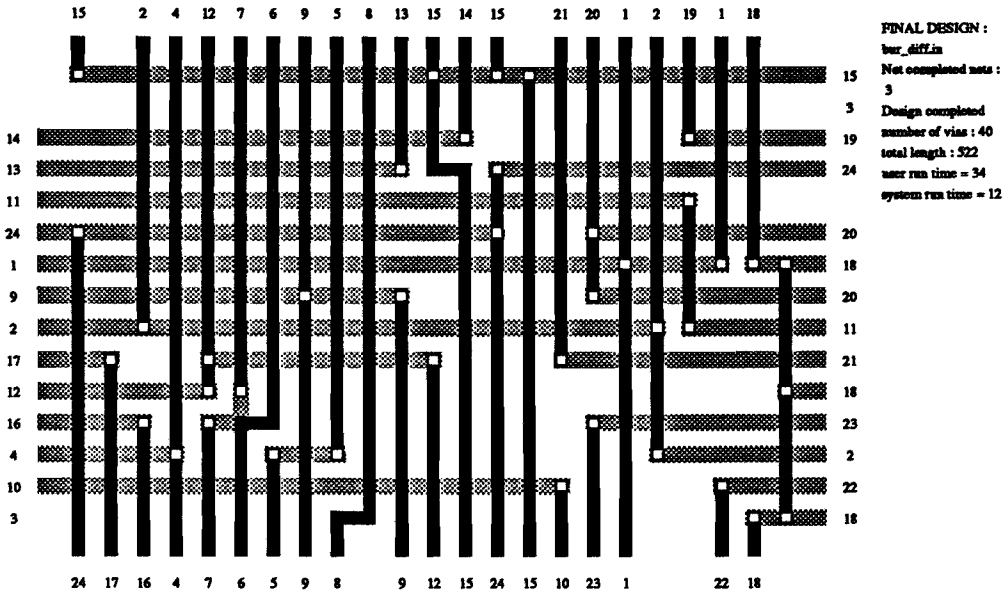


Figure 3.6. The final solution on Burstein's difficult switchbox

The main reason for failure of the Lee router is the solution for net 15 as it is found by Answer. Net 15 uses almost two complete columns (13 and 15) where only one would be enough. This solution has been found due to the interval definition. Because there are terminals of net 15 in column 12, 13, 14 and 15 there are three intervals of unit length (I : $\{x \in [12,13]\}$, I : $\{x \in [13,14]\}$ and I : $\{x \in [14,15]\}$). The routing algorithm applied on these three intervals results in the solution shown. Therefore, the only way to complete the routing for net 3 is by removing part of the route of net 15, and route it again after net 3 has been routed.

Removing already made connections and routing them again later on is called rip-up and reroute. This rip-up and reroute can help to remove blocking nets where the other modificationsteps fail. In Mighty [2] rip-up and reroute (named strong modification) is also used. In our switchbox router we could also introduce rip-up and reroute to remove a blocking connection, using the Lee router to route the unfinished and removed nets again. But there are some questions. How do we find the nets to rip-up? And should we remove the complete net or only part of it?

An answer to the first question has been given by Ohtsuki [9] but the definition and examples do not agree with each other. For the second question we have not

Chapter 4
Data structures

When we are routing a switchbox, we are working with great amounts of data containing information on the grid, the CG etc. Especially during the main routing algorithm, when only the lists of intervals are used and not the grid, this data handling is very important. Therefore, in this chapter the data structures, used to store the information, will be discussed. There are two data structures which are important to discuss, namely the lists of intervals and the CG.

4.1 The list of intervals

Nest are defined as a set of intervals to be connected. We have defined two structures, one for the nets and one for the intervals. The structure Net contains a pointer to the first and last interval. The structure Interval describes each intervals by a begin and end point, a track, a layer and a pointer to a next interval.

To describe all intervals there are two arrays, Hor_netparts for the horizontal and Ver_netparts for the vertical intervals. If there are N nets, Hor_netparts contains N elements of the structure Net. For each net i its horizontal intervals are found in the ith element of Hor_netparts. Similar, Ver_netparts contains N elements of the structure net and for each net i its vertical intervals are found in the ith element of Ver_netparts.

The definition of the structures and the list of intervals is given in definition 4.1 and a graphical overview is given in figure 4.1. In this figure Netparts can be either Hor_netparts or Ver_netparts.

Definition 4.1

```
typedef struct net NET;
typedef struct interval INTERVAL;

struct net {
        INTERVAL  *first, *last;
};

struct interval {
        int   track;
        int   l;
        int   begin, end;
        INTERVAL   *next;
};

Hor_netparts = (NET *) calloc ((unsigned)Net_dim, sizeof(NET));
Ver_netparts = (NET *) calloc ((unsigned)Net_dim, sizeof(NET));
```

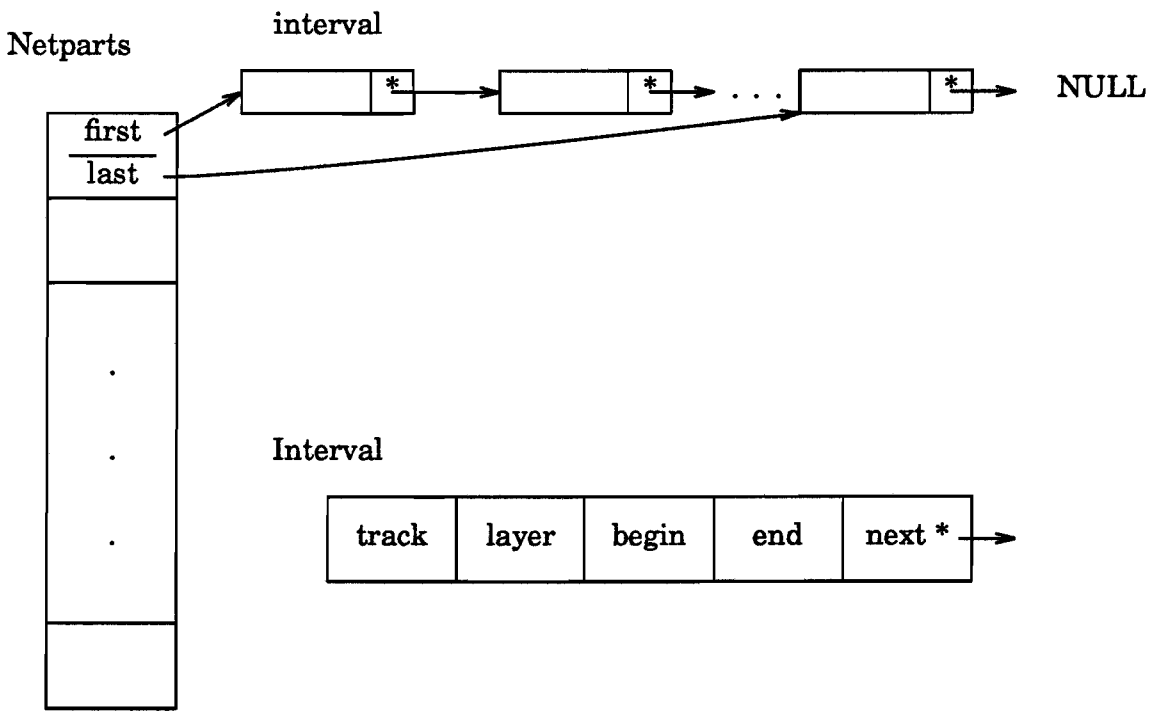


Figure 4.1. Graphical overview of the data structure for intervals

4.2 The constraint graph

In the CG each node corresponds with a net and has a certain level. For such a node the only important information, besides its level, are the netids of its direct successors and direct predecessors. To store these data there are two structures defined : A structure Node containing the level of the node and two pointers, a

pointer to the first direct predecessor or direct successors and a pointer to the last direct predecessor or successor. A structure Suc containing the netid of the direct pred- or successor and a pointer to the next direct pre- or successor. Four arrays, named nodelists, are used to store the data of the CG. Two nodelists containing the data of the predecessors, one for the top_bottom_cg and one for the left_right_cg, and two nodelists containing the data of the successors, again one for the top_bottom_cg and one for the left_right_cg. If there are N nets, these nodelists contain N elements of the structure Node. The pre- or successors of net i are found in the ith element of the nodelists.

The definition of the structures is given in definition 4.2 and a graphical overview is given in figure 4.2. Nodelist may be either hor/ver_sucnodelist or hor/ver_prenodelist.

Definition 4.2

```
typedef struct node NODE;
typedef struct suc SUC;

struct node {
  int level;
  SUC *first, *last;
};

struct suc {
  int netid;
  SUC *next;
};

Hor_sucnodelist = (NODE *) calloc ((unsigned)Net_dim, sizeof(NODE));
Ver_sucnodelist = (NODE *) calloc ((unsigned)Net_dim, sizeof(NODE));
Hor_prenodelist = (NODE *) calloc ((unsigned)Net_dim, sizeof(NODE));
Ver_prenodelist = (NODE *) calloc ((unsigned)Net_dim, sizeof(NODE));
```

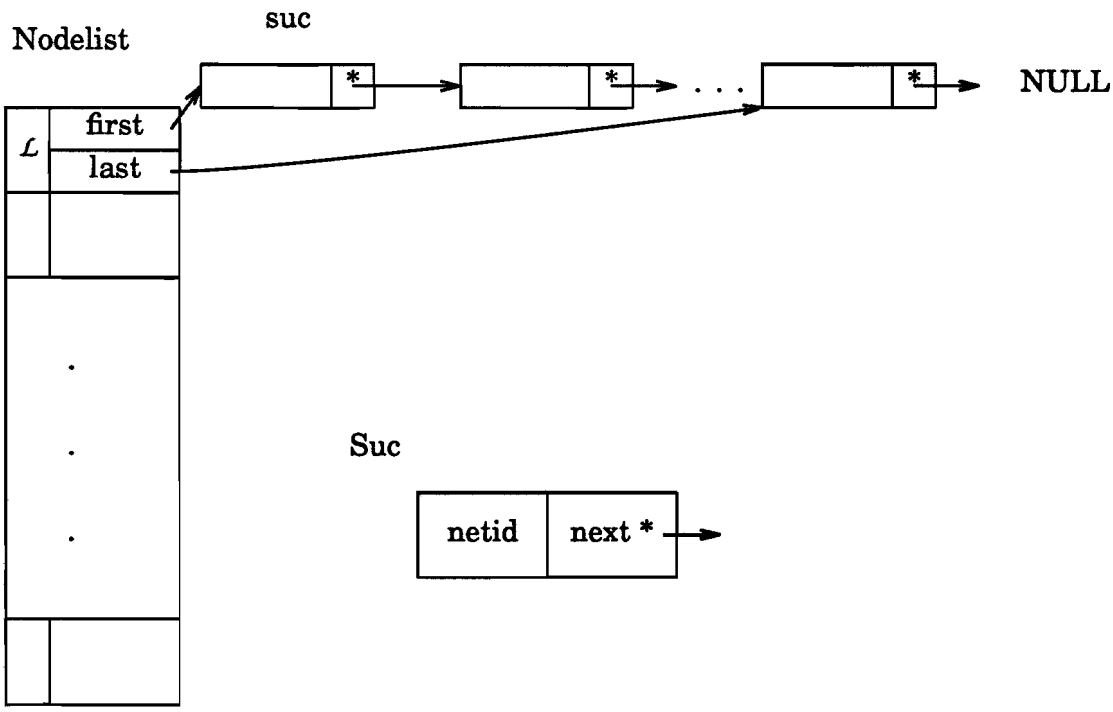


Figure 4.2. Graphical overview of the data structure for the CG

Chapter 5
Results

In this chapter the results obtained on several switchbox examples will be discussed, and a comparison with some other well-known switchbox routers will be given. In the tables, "Answer" is the name of our router.

To compare the cpu times of all switchbox routers, table 5.1 lists the machines on which the routers have run. In this table GSR [10] stands for Luk's greedy switchbox router; Weaver [1] is the name of Joobbani and Siewiorek's router; Beaver [4] is the switchbox router of Cohoon and Heck; Mighty [2] is the name of Shin's and Sangiovanni-Vincentelli's router; Carioca [5] is a switchbox router developed by Dubois, Puissochet and Tagant, and Packer and Cracker [11] are two routers developed by Gerez.

Table 5.1. Machine comparison for some switchbox routers

Routername	Machine
GSR	?
Weaver	VAX 11/785
Beaver	Sun3
Mighty	VAX 11/785
Carioca	Sun3/160
Packer	Apollo DN 4000
Cracker	Apollo DN 4000
Answer	HP 9000

The cpu times given in the following tables will all be in seconds.

The results of our switchbox router on some well-known benchmarks, and the comparison with the routers mentioned above, will now be given in graphics and tables. In fig. 5.1 Answers solution to Burstein's difficult switchbox is given. In chapter 3 we have already seen that Answer did not complete the routing for net 3 and that the only solution would be introducing rip-up and reroute. The results in comparison to some other routers is given in table 5.2. The * in the row for Answer denotes that Answer did not complete the routing for the switchbox.

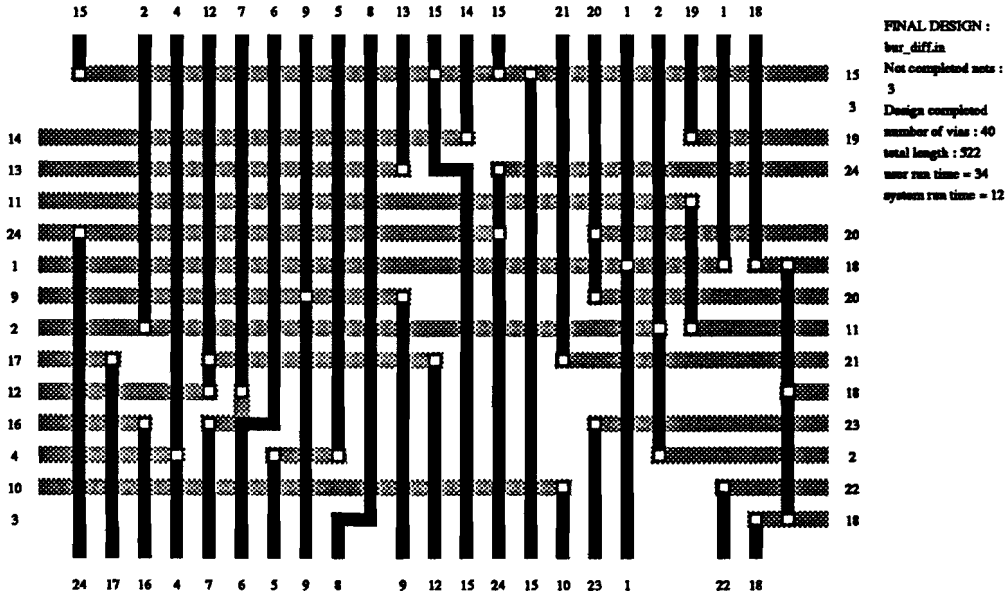


Figure 5.1. Answer's solution for Burstein's difficult switchbox

Table 5.2. Router comparison for Burstein's difficult switchbox

Routername	#vias	length	cpu time
GSR	58	577	15
Weaver	41	531	1390
Beaver	35	547	1
Mighty	39	541	4
Carioca	43	535	54
Packer	45	546	56
Cracker	45	542	210
Answer *	40	522	0.8

In fig. 5.2 the results of Answer for the modified dense switchbox are given. In table 5.3 a comparison with other routers is made. Note that Answer used two more vias. This is a result of the type of via minimization we have chosen. From the figure we see that for both net 2 and net 9 there is a via which can easily be removed, and there's no real diffrence between Answers solution and the others.

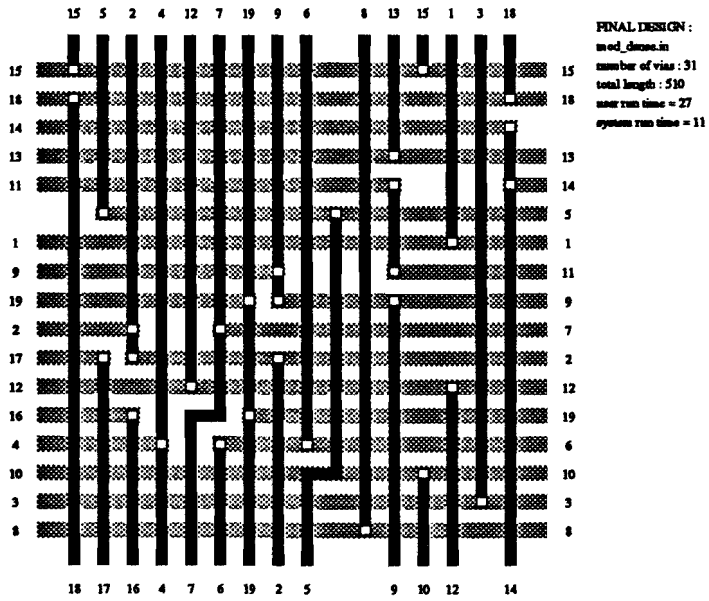


Figure 5.2. Answer's solution for the modified dense switchbox

Table 5.3. Router comparison for the modified dense switchbox

Routername	#vias	length	cpu time
Weaver	29	510	924
Beaver	29	510	1
Mighty	29	510	?
Carioca	29	510	65
Packer	29	510	36
Cracker	29	510	40
Answer	31	510	0.6

In fig. 5.3 the results of Answer on the terminal intensive switchbox are given. Again Answer did not complete the routing. A solution for this switchbox can be found when several nets are ripped-up and rerouted. However, there's no good strategie to find these nets. In table 5.4 Answer is marked with a * to denote that the routing was not completed.

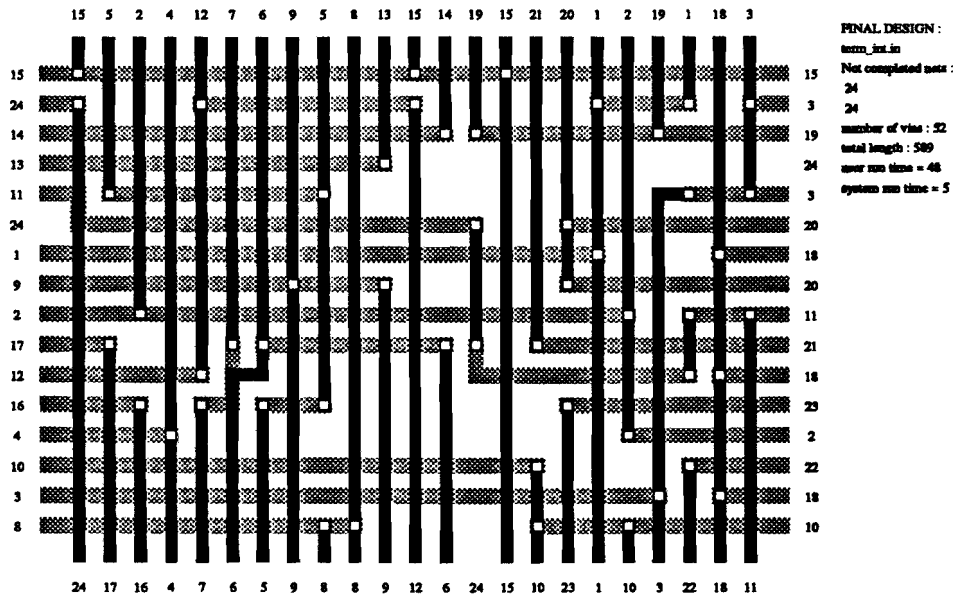


Figure 5.3. Answer's solution for the terminal intensive switchbox

Table 5.4. Router comparison for the terminal intensive switchbox

Routername	#vias	length	cpu time
GSR	68	632	?
Weaver	49	615	1874
Beaver	46	632	1
Mighty	50	629	?
Packer	50	626	210
Cracker	49	620	490
Answer *	52	589	0.9

In fig. 5.4 the results of Answer for a simple switchbox, the sample switchbox, are given. This switchbox was routed by Answer using only the original routing algorithm. The difference in number of vias for this example is due to the fact that some switchbox routers allow the terminals to be on either layer (Weaver and Beaver).

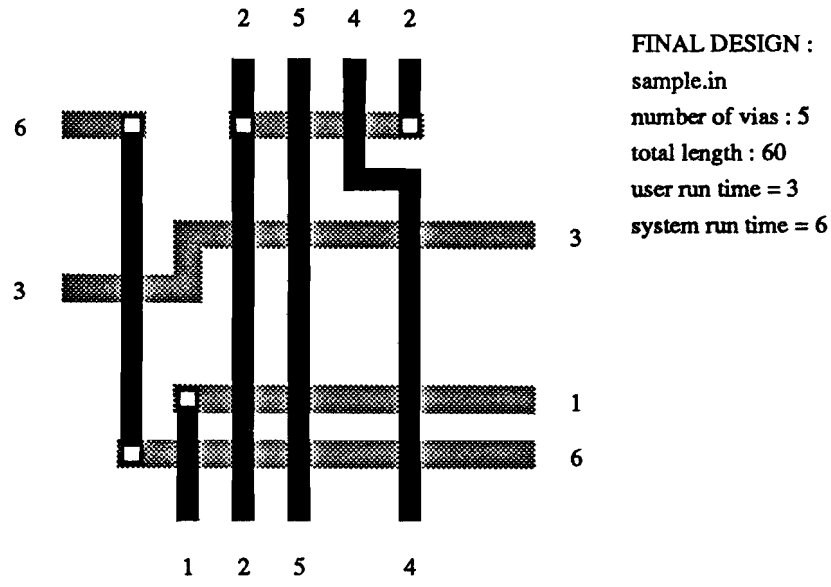


Figure 5.4. Answer's solution for the sample switchbox

Table 5.5. Router comparison for the sample switchbox

Routername	#vias	length	cpu time
Weaver	4	60	73
Beaver	3	60	1
Mighty	5	60	?
Answer	5	60	0.1

As a final overview of the results obtained by Answer, a table is given in which various benchmarks are compared. The results obtained on these benchmarks are also given in figures 5.5 to 5.10.

Table 5.6. Router comparison for various benchmarks

Example name	#vias	length	cpu time	unfinished nets
Bur_diff	40	522	0.8	3
Mod_dense	31	510	0.6	-
Term_int	52	589	0.9	24
Sample	5	60	0.1	-
Aug_dense	33	529	0.5	-
Bur_more_diff	38	510	0.6	3
Comp_1	38	467	0.7	3,11
Mod_diff	34	428	0.9	3,21
Pedagogical	36	318	0.7	1,2,3
Random	49	647	1	2

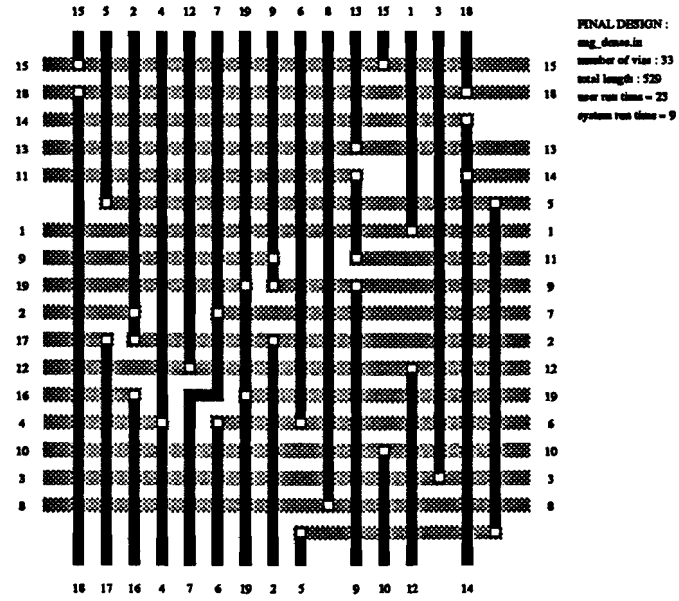


Figure 5.5. Answer's solution for the augmented dense switchbox

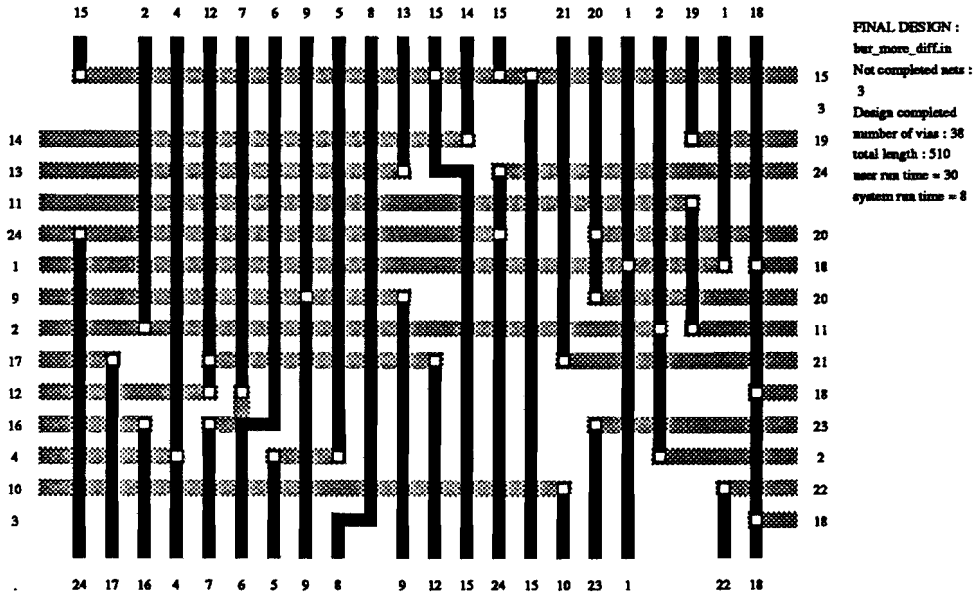


Figure 5.6. Answer's solution for Burstein's more difficult switchbox

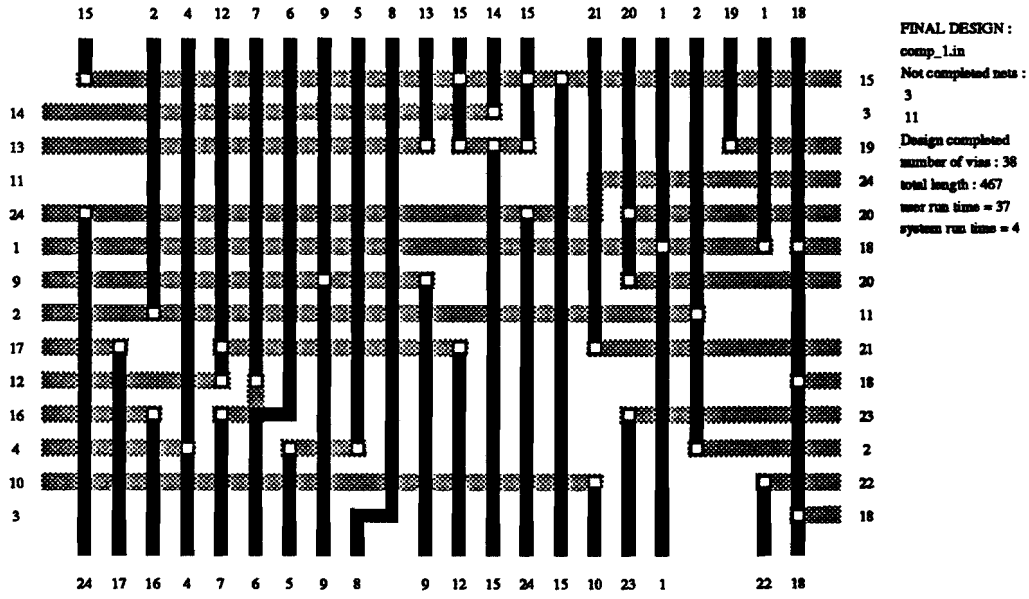


Figure 5.7. Answer's solution for the comp_1 switchbox

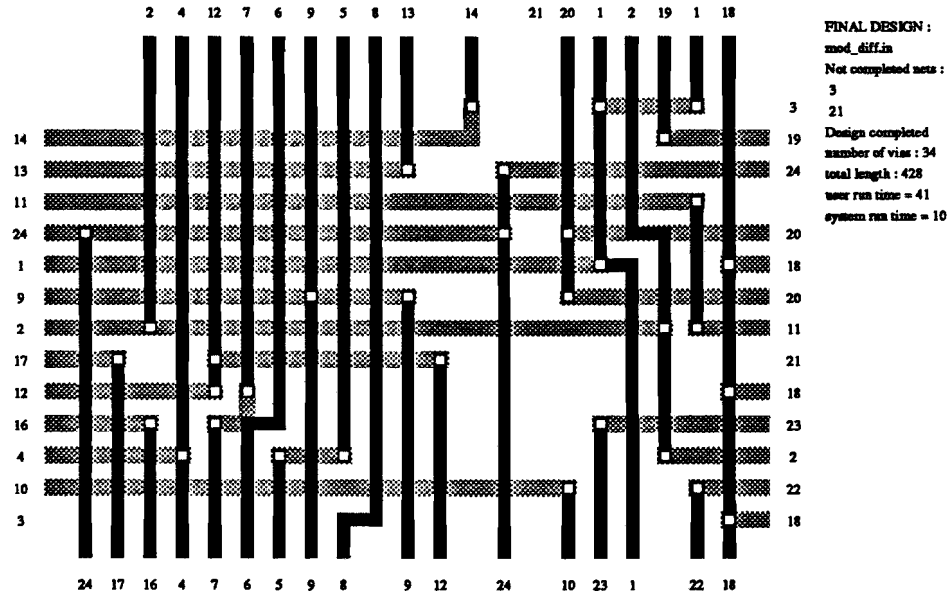


Figure 5.8. Answer's solution for the modified difficult switchbox

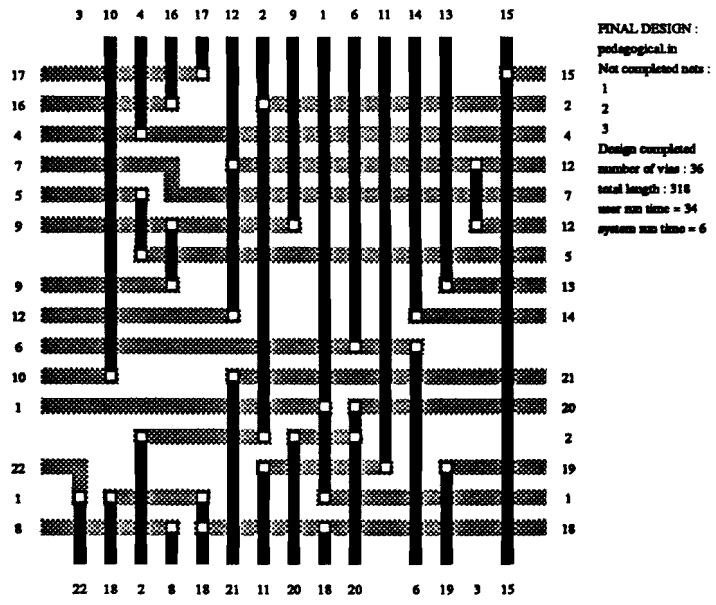


Figure 5.9. Answer's solution for the pedagogical switchbox

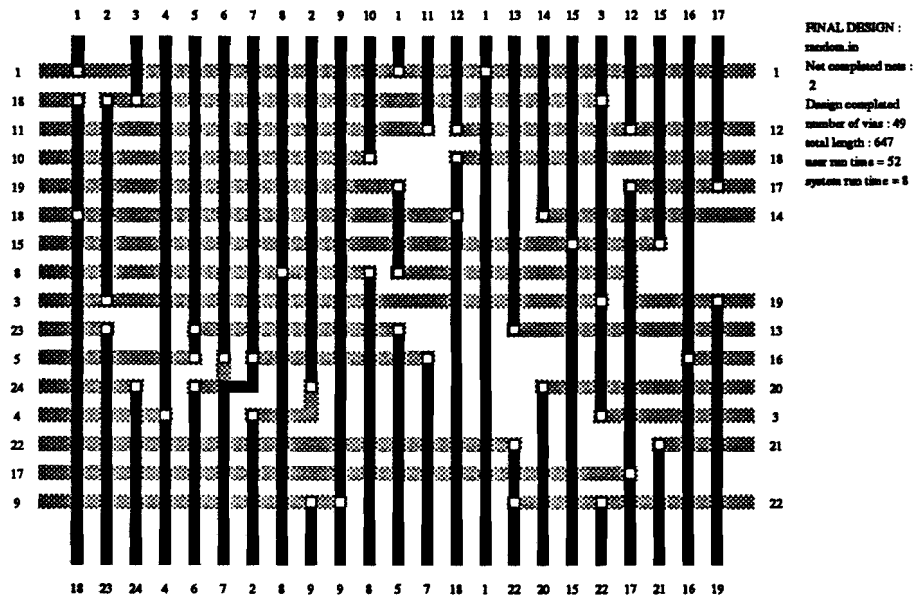


Figure 5.10. Answer's solution for the random switchbox

Chapter 6
Conclusions

A new routing algorithm based on a straightforward approach has been developed and some modificationsteps to improve the solution have been introduced.

The results obtained by this switchbox router are hopeful because most of the necessary connections are made but the router sometimes fails to interconnect a 100%. The greatest advantage of this switchbox router is that it is very fast. To improve the results even more, rip_up and reroute can be introduced. A more general approach to detailed routing can also be obtained by this router by introducing blockages.

Fields of interest in further research on this matter should therefore be:

- Introducing blockages.
- Introducing rip_up and reroute

Other fields of interest to improve this router can be:

- More than two layers.
- Floating terminals.

References

[1] R. Joobbani and D.P. Siewiorek, "WEAVER: A knowledge based routing expert," in *IEEE Design & Test*, vol. 3, pp. 12-33, Feb. 1986.

[2] Hyunchul Shin and Alberto Sangiovanni-Vincentelli, "A Detailed Router Based on Incremental Routing Modifications: Mighty," in *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 942-955, November 1987.

[3] William A. Dees, Jr. and Patrick G. Karger, "Automatic Rip-up and Reroute Techniques," in *Proceedings 19th Design Automation Conference*, pp. 432-439, 1982.

[4] James P. Cohoon and Patrick L. Heck, "BEAVER: A Computational-Geometry-Based Tool for Switchbox Routing," in *IEEE Transactions on Computer-Aided Design*, vol. CAD-7, pp. 684-697, June 1988.

[5] Pierre Francois Dubois, Alain Puissochet, and Anne Marie Tagant, "A General and Flexible Switchbox Router: CARIOCA," in *IEEE Transactions on Computer-Aided Design*, vol. CAD-9, pp. 1307-1317, December 1990.

[6] Malgorzata Marek-Sadowska, "Two-dimensional router for double layer layout," in *Proceedings 22th Design Automation Conference*, pp. 117-123, 1985.

[7] M. Burstein, "Channel Routing," in *Advances in CAD for VLSI, Elsevier Sience Publishers*, vol. 4 Layout Design and Verification, pp. 132-167, 1986.

[8] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 346-365, 1961.

[9] Tatsuo Ohtsuki, "Maze-Running and Line-Search Algorithms," in *Advances in CAD for VLSI, Elsevier Sience Publishers*, vol. 4 Layout Design and Verification, pp. 99-131, 1986.

[10] W.K. Luk, "A greedy switchbox router," in *Integration, the VLSI journal*, vol. 3, pp. 129-149, 1985.

[11] Sabih Habib Gerez, "Local wire routing by stepwise reshaping," *Ph.D. thesis, University of Twente, Faculty of Electrical Engineering*, Oct. 1989.