SPAR: A Schematic Place and Route System

Stephen T. Frezza and Steven P. Levitan

Abstract

This paper presents an approach to the automatic generation of schematic diagrams from circuit descriptions. The heuristics which make up the system are based on two principles of schematics readability: *Functional Identification* and *Traceability*. SPAR's generation process is broken into five distinct phases: partitioning the netlist, placement of components on the page, global routing, local routing, and the addition of I/O modules. All phases of the generation process use a two dimensional space management technique based on virtual tile spaces. The global router is guided by a cost function consisting of both congestion and wirelength estimates. The local router uses a constraint–propagation technique to optimize the traceability of lines through congested areas. The data structures and algorithms used allow the system to support incremental additions to the schematic without complete regeneration. We describe a technique for evaluating the quality of schematic drawings, and apply this to our results.

This paper appeared in the July, 1993 issue of *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* (12:7) pp. 956-973.

1 Introduction

Schematic diagrams are one of the most comprehensive design representations for human designers [1]. They were once the principle design medium for digital circuits, providing a clear, graphical means of specifying circuits at different levels of complexity. However, the move toward using textual descriptions to generate complex designs has shifted the use of the schematic diagram from an input medium to an output, or display medium. These textual description languages have a tendency to be somewhat obscure [2], often lacking the depth of insight provided by a graphical image. Hence there is a need for schematics generated from textual hardware descriptions.

A useful schematic diagram communicates "the flow of information, time, or material without reference to a physical layout" [3]. Therefore, the utility of any Automatic Schematic Generation (ASG) system must be judged by how well its output is understood by the designer. Put another way, "Schematic drawing...must be considered in human aspects" [1]. To meet the needs of the human, the schematic must readily convey the logical flow of information and the relationship among the components of the circuit. We use the concepts of *functional identification* and route *traceability* [4] to describe how well the schematic meets these needs.

Functional identification occurs when the designer sees functionally interconnected modules located in proximity to each other with separation between distinct groups of modules. Two common examples of identifiable structures would be a single gate enabling several others as a fan-out structure, and two cross-coupled gates as a flip-flop. *Traceability* is the ease with which the designer can follow signals from their logical sources to their destinations, and requires paths that are simple and do not pass through congested regions of the diagram.

With these goals in mind, we have implemented SPAR (Schematics Placement and Routing) as a heuristic approach to the generation of schematic diagrams. The system automatically creates schematic diagrams from textual descriptions of digital systems. SPAR derives the connectivity among the modules in the netlist, and uses this information to form partitions of functionally related modules. The modules within these partitions are placed separately in gridless spaces, then merged using space management techniques. Global routing of the schematic is accomplished using an A* [5] search, which is refined using a congestion metric. Local routing is accomplished by applying ordering rules to separate nets in a specific area, and applying a constraint propagation technique to communicate the re-ordering decisions made. Additionally, local routing information is explicitly used to place the Input/Output (I/O) modules on the page.

Figure 1 shows a diagram generated by SPAR for a simple master-slave D-Latch. Embedded within this design are two RS-flip-flops which are circled in the figure. These are easily recognized, as they have been placed in a functionally identifiable way. The signals flow in the design from left to right, from signal inputs to outputs in an easily traceable manner. Although small, the D-Latch example of Figure 1 exemplifies the task that an ASG system must perform well. The schematic generated typifies the identification of function, and the traceability of the signals used.

In this paper we first present the motivation for automatic schematic generation in the problem description of Section 1. This section describes our view of schematic generation which underlies SPAR, and develops the characteristics that are desirable in a generated schematic. Using these

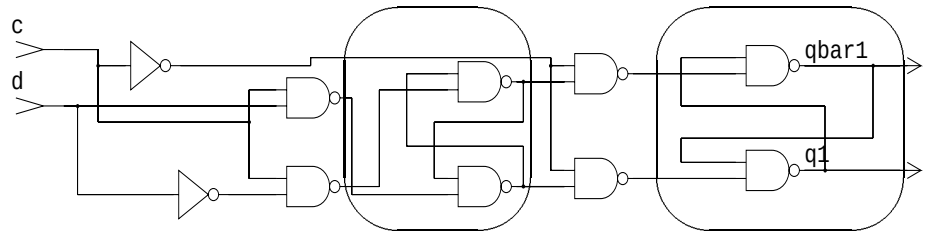


Figure 1: Completed D Latch, size-based partitioning

desirable characteristics as a point of focus, Section 2 describes previous ASG work, and how SPAR differs from these systems. In this context, we describe many of the common issues in ASG that are addressed in SPAR. Section 3 presents an overview of the system operation, and outlines the major tasks involved. This includes the mechanics of each of the major tasks: partitioning, placement, global routing, local routing, and the insertion of I/O modules. Section 4 presents an analysis of our results. Our concluding remarks follow in Section 5, accompanied by a brief discussion of future work.

2 Background

The difficulty in automatic schematic generation is in producing useful diagrams, those that meet the aesthetic and often conflicting expectations that humans make of the diagram. Researchers who have proposed solutions for ASG have developed their systems around sets of rules that capture portions of these expectations. Like the algorithms used, these rules can broken up into criteria for good placement vs. criteria for good routing, as summarized here:

Diagrams must...

- Be regular, symmetrical, functionally readable. [1]
- Be grouped in functional groups. [3, 6, 7]
- Be arranged such that most signal flow is uni-directional. [7, 8]
- Maintain and demonstrate the direction of flow. [3, 9]
- Have short connections between logic units. [7, 8, 10]

Paths should...

- Use small numbers of line branching nodes and intersections. [3, 8, 10, 11]
- Exhibit low net-crossing complexity. [10, 11, 12]
- Not be too long, or have too many bends. [8]
- Minimize the number of sharp turns. [3, 10]
- Minimize the number of segments, crossovers, and total path length. [7, 10, 13, 14]

This gross summary of schematic aesthetics shows that the engineering community clearly has "rigid standards" [14] for schematic generation. However, the large variations in the methods used in previous work show that there is no clear consensus on how to implement these standards. We take the view that although all of the above criteria are useful, they only incidentally address the overall goals of ASG systems. These goals are to readily convey the logical flow of information and the relationship among the components of the circuit by producing traceable diagrams with identifiable structures.

2.1 Traceability

In addressing traceability issues, most ASG systems have emphasized the elimination of crossovers from the diagram. This has been accomplished by heuristic routers [14, 15, 9], modified channel routers [1, 4, 10, 13], modified line-expansion routers [7] or with knowledge-based routers [3, 16, 17, 18]. Some systems also place modules so as to minimize crossovers explicitly, such as those described in [10, 11] and the simulated-annealing technique described in [12].

Unfortunately, crossovers are not the only hindrance to the traceability of a diagram. The proximity of nets, or rather the proximity of the features of nets, also hinder the traceability of the diagram. We observe that this *congestion of net features* is an area phenomenon. For example, as the corner density (number of corners or bends in a specific area) goes up, the traceability of the lines through that area goes down. As an example, Figure 2 shows two regions with the same number of nets crossing them. In the two figures, the crossover and density counts are reversed. Clearly, the wires in the region with the high corner count are more difficult to trace than those in the region with the high crossover count. This indicates that the goal of traceability is better measured by the density of corners rather than simply the number of crossovers.

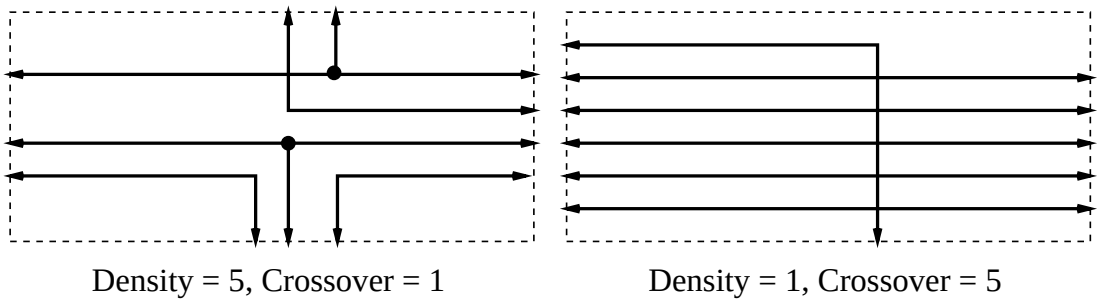


Figure 2: Density vs. crossover count as a measure of traceability

The rules for crossover, bend and connection counts listed at the beginning of this section completely ignore the density issue, as they have no tie to the areas in which they occur. Many such crossovers or bends, when spread over the diagram, do not strongly hinder the traceability of the net. What we observe from Figure 2 is that the minimum crossing solution is not necessarily the most traceable solution. Thus there are two traceability issues to address: congestion *and* crossover.

While the discussions presented in much of the previous work [3, 7, 10, 12, 14] clearly illustrate

the importance of crossover to the readability of the schematic, one of our contributions to ASG research is in addressing the congestion issues that are also present in schematic routing. In SPAR, we address the congestion problem with an iterative-refinement technique in the global router; we address the crossover problem using a constraint-propagation technique in the local router.

2.2 Functional Identification

The functional relationship of a given set of modules is transmitted both the nearness among related modules, as well as their distance from unrelated ones. This relationship is typically addressed at two levels: A logical level, where related modules are grouped together, and a physical level, where this relationship among the modules is mapped to their placement on the page.

The *logical* part of addressing functional relationships is accomplished with partitioning. A benefit of partitioning the design into logical groups of modules is that the placement of each partition can be considered separately, thus reducing the complexity of the placement task. Three different partitioning techniques appear in the literature.

The first technique, as typified by *HAL* [16] is to use expert rules to form clusters of modules into partitions, with the object of minimizing the complexity of the resulting task. A second technique, as used in *AUTODRAFT* [4], partitions depend on the hierarchy embedded in the design description. Finally, *GEMS* [17] divides the schematic into fixed-sized blocks before any modules are placed, while *Pablo* [7] adds modules to partitions until the number of modules contained, or the number of connections external to the partition exceeds their maximum limits [19].

The *physical* means of addressing functional relationships is to create an appropriate placement by varying the amount of space between modules or groups of modules. These spacing issues can be difficult to handle, and are referred to collectively as the Space Management Problem. Most ASG systems manage the diagram space by relying upon a fixed coordinate (grid) system [1, 3, 4, 6, 12, 8, 10, 11, 15, 13, 9].

The grid convention commonly used has each icon or via occupying one location or grid point. The grid scale and module locations are then adjusted to improve the aesthetics of the completed diagram. This has computational advantages in that it limits the reasoning process to a finite set of possibilities [3]. However, grids make it difficult to model the idea that *locality* implies *relation*. This is because the grid provides no means for adding spacing around a particularly identifiable group of modules if they are not all in the same column or row. Grids are also not flexible enough to handle odd sized modules, such as when icons are not of fixed sizes, or when the space required for the terminals is larger than the icon, *e.g.*, when gates have a large fan-in.

Other work has been done without the limitations of grids, most notably *Pablo*, *HAL*, and *GEMS*. These systems manage space by the construction of *strings*, which are lists of modules linked by an output→input relationship among their terminals. A string is composed of one or more of such modules surrounded by a bounding box which defines the space that the modules are placed in. As these systems are only able to abut strings based on their bounding boxes [19], empty space appears in the diagram that otherwise cannot be used. In SPAR we eliminate the

restrictions imposed by the grid and bounding box methods of placement with a gridless space-management technique. In addition, we have implemented a modified version of *Pablo*'s string placement technique. Together, these techniques allow SPAR to use space more effectively than previous systems.

The two methods of addressing functional relationships (logical and physical) are key to meeting the goal functional identification. Together with the previously stated goal of traceability, they constitute our notion of the ASG problem. Given this formulation, and what we have learned from the successes and failures of previous work, we now go on to describe the details of the SPAR system.

3 SPAR Approach

SPAR is a heuristic approach to ASG which uses connectivity information to determine the partitioning, which in turn drives the placement of the modules. Placement is followed by global and local routing. Recognizing that a poor placement can make traceable routing impossible, we try to discover a partitioning and a placement which exhibits a reasonable representation of the functional nature of the netlist. Our method is based on the fact that a well-partitioned diagram will be well-placed, and that by observing the signal inheritance within these individual partitions, a good route will be achieved.

SPAR divides the ASG problem into five major tasks: *Partitioning, Placement, Global Routing, Local Routing,* and *Placement and Routing of I/O Modules.* These tasks are performed in order, where the global routing is adjusted to reflect addition of the I/O modules to the diagram. Local routing is repeated, as new corners have been introduced that may influence the local route.

To place these actions in context, we first present an overview of the space management technique we employ, followed by discussions of the five major tasks performed by the algorithm. The details of each of the algorithms are presented as pseudo-code at the end of each section. Following this discussion, Section 4 presents a discussion of the evaluation of SPAR-generated schematics.

3.1 Space Management

Because routing congestion is an area phenomena, and placement is permitted for arbitrarily-sized icons, space management is a critical ASG issue. Our solution to these problems is to use a corner-stitched *tile space* [20] to maintain the schematic data, and also as a map which can be queried for this information. A tile space consists of a set of two–dimensional rectangular tiles linked at their corners, and in our implementation may contain either modules or net information.

We use tilespaces for both placement and routing, although in different ways. During placement, a tile space is created and assigned to each partition, denoted a *virtual page*. Using such tile spaces, we are able to build individual functional groups for each partition. These virtual pages, containing the locally-placed modules, are then merged to form the diagram space. For routing we use two

90°-rotated tilespaces to directly locate the modules and nets, which permits the tile spaces to be queried for congestion information and other features. This is especially important for the incremental placement of the I/O modules, where the I/O modules are added to an existing tilespace that contains routes and modules.

Standard cornerstitching uses tiles of two varieties - tiles that contain a color, and tiles that do not (empty tiles). Tiles of the same color can be merged, and empty tiles can be merged with empty tiles. Additionally, empty tiles extend horizontally as far as possible. Tile insertion involves locating the area where the new tile will be added, and proceeds only if the entire area contains empty tile(s). The existing tile that contains the top of this area is split horizontally at the point where the top edge of the new tile will fall. The lower of the tiles created is *selected*, and then *split* vertically twice: where the left and right edges of the new tile fall. The tiles that remain outside the new tile are then conditionally *merged* with the tiles above them. The side tiles are only merged with the tiles above them if they have the same horizontal start and end point. This select, split and merge process continues until the bottom edge of the new tile is reached, the bottom edge being handled identically to the top edge. As each split or merger occurs, the links among the tiles are updated.

In our use, tiles containing modules can never be split or merged, but tiles containing routes can. SPAR tiles contain pointers to the structures they contain, such as modules, the list of nets that use the tile, or the structure containing all of the partial routes that use the tile. The details of these structures and their use are described in the relevant sections that follow. Whenever a merge or split operation is required, functions are called to manipulate the structures contained. These modifications to the merge and split operations are described in Section 3.6.

The details of how we use our space management techniques are included within the discussions of each relevant task. The important features of tile space use are in the merging of partitions described in Section 3.3, the evaluation of proposed nets for congestion described in Section 3.4, and the ability to add to the diagram and determine column blockage as described in Section 3.6.2.

3.2 Partitioning - Discovering Functional Relationships

The ASG process begins by forming *partitions* of modules from the input netlist using clustering techniques [21]. The goal of clustering is to group modules together such that the internal structures within the design are highlighted. These clusters are used to form the partitions that drive the placement and routing process. Partitions contain groups of modules that exhibit local feedback within the partition. Global feedback is addressed by the inter-partition algorithms.

Three strategies have been implemented for the formation of partitions: size-based, Rent-like, and slope-based. The strategy employed in creating the diagram is a user-supplied argument.

The first strategy, size-based partitioning, is a technique where partitions are acceptable as long as either the number of modules within the partition is less than some specified size limit, **max-partition-size**, or the number of connections external to the partition is less than some specified size limit, **max-partition-conn**. The second strategy, Rent's rule [22] is used as a model for

Rent-like partitioning. Here partitions are acceptable as long as the ratio of external connections to the number of modules is less than a specified ratio limit, **partition-ratio**. The third strategy, slope-based partitioning, is a technique where a partition is accepted as long as the number of connections in the proposed partition is less than the largest of the sum of the connections in the child partitions. These rules are implemented in the function **Valid-partition()**.

The clustering algorithm uses a connection matrix to maintain the information about the connectivity of clusters of modules. Figure 3 shows the connection matrix for the SN7474 example (Figure 8) after one merger between two clusters. Specifically the clusters containing the single modules *nand4* and *nand5* have just been merged, and this cluster is about to be merged with the cluster containing *nand3*. An overview of the entire clustering algorithm is presented in Section 3.2.1.

	clr	set	clk	d	q	qbr	1	2	3	4+5	6	
nclr	3	0	0	0	0	0	1	0	1	0	1	
nset	0	2	0	0	0	0	0	0	0	2	0	
clock	0	0	2	0	0	0	0	1	1	0	0	
d	0	0	0	1	0	0	1	0	0	0	0	
q	0	0	0	0	2	0	0	0	0	1	1	
qbar	0	0	0	0	0	2	0	0	0	1	1	
nand1	1	0	0	1	0	0	8	2	1	1	2	
nand2	0	0	1	0	0	0	2	9	2	3	1	
nand3	1	0	1	0	0	0	1	2	9	3	1	i
nand5+nand4	0	2	0	0	1	1	1	3	3	13	2	j
nand6	1	0	0	0	1	1	2	1	1	2	9	
									i	j		

Figure 3: Connection matrix for the SN7474 after the merger of the clusters containing *nand4* and *nand5*

Once the modules have been partitioned into smaller groups based on their connectivity, the task remains to use these partitions to place the modules on the page. The placement algorithm is discussed in the following section.

3.2.1 Partitioning Algorithms

Definitions:

clusters : list of clusters, initialized with modules; Each cluster corresponds to one row and column in the connection matrix M.

M : The connection matrix; $M[i,j]$ corresponds to the number of connections between the modules contained in $cluster_i$ and those in $cluster_j$. $M[i,i]$ corresponds to the number of connections external to the modules in $cluster_i$.

partition: the list of partitions to return.

Build-clusters()

```
{ /* Sequentially merge clusters until all have been clipped
     to form partitions. Return the partition list created. */
  do while |clusters| > 1 {
     /* This loop picks one pair of clusters, and either merges
        them or splits one off as a partition. */
     find i, j s.t. ∀i ∀j, i ≠ j, M[i,j] ≠ 0, the cluster value
              ( (M[i,i]+M[j,j]-M[i,j]-M[j,i]) / (M[i,i]+M[j,j]) ) is minimum;

     /* Merge or Clip one cluster from the list: */
     if (Valid-partition(cluster_i, cluster_j))
          Merge column/row j into column/row i of M
               such that the column/row i information correctly
               represents the connections to/from the merged
               cluster;
          Merge cluster_j into cluster_i;
     else if (|cluster_j| ≥ |cluster_i|)
          Clipcolumn/row j from M such that M reflects
               the connections with cluster_j removed;
          Save cluster_j as a partition;
     else
          Clipcolumn/row i from M such that M reflects
               the connections with cluster_i removed;
          Save cluster_i as a partition.
  }
  add cluster_0 to partition;
  return(partition);
}
```

Valid-partition($cluster_i, cluster_j$)
{ /* Return TRUE if the combination of $cluster_1$
and $cluster_2$ yields an acceptable partition. */
$connection\,count \longleftarrow M[i,i] + M[j,j] - M[j,i] - M[i,j]$;
$module\,count \longleftarrow |cluster_i| + |cluster_j|$;

case (partition-type) {
(size-based) :
if ($module\,count \leq$**max-partition-size**) *and*
($connection\,count \leq$**max-partition-conn**)
return(TRUE);
(Rent-like) :
if ($\frac{connection\,count}{module\,count} \geq$**partition-ratio**)
return(TRUE);
(Slope-based) :
if ($connection\,count \geq \mathrm{MAX}(M[i,i], M[j,j])$) *and*
($connection\,count \leq$ **max-partition-conn**)
return(TRUE);
otherwise : return(FALSE);
}
}

3.3 Placement of Modules

Our placement algorithm for modules, within partitions, is a series of improvements on that used in *Pablo* [7]. A *string* is a list of modules formed by tracing the out→in relationships among the unplaced modules within a partition. Each partition is dealt with separately as its own virtual page, and later these pages are merged. The placement on a virtual page is formed by building strings of modules, with each string placed in its own tile. These strings are formed, placed on tiles, and their tiles are added to the virtual page until all modules in the partition have been placed. Later, the tiles that make up the virtual page are merged with those of other pages to produce the final placement.

When forming and placing a string of modules, SPAR looks specifically for strings with cross-coupling, called *complex* strings. Complex strings are formed recursively by forming substrings at each cross-coupled point. In Figure 4, three such substrings are shown. The main string starts with module *NOR.2* and ends with module *NOT.3*. Substring 1 also begins with *NOR.2* but ends with *BUF.1*. Substring 2 begins with *AND.5* and ends with *BUF.2*. Finally, substring 3 begins with *NOT.4* and ends with *BUF.4*. This string generation permits the proper identification of important features such as local feedback among modules. Throughout the placement process, each tile contains a single complex string, which is maintained as a linked list of modules. The locations assigned to each module are all within the boundaries of the tile used, and are changed whenever the tile is moved.

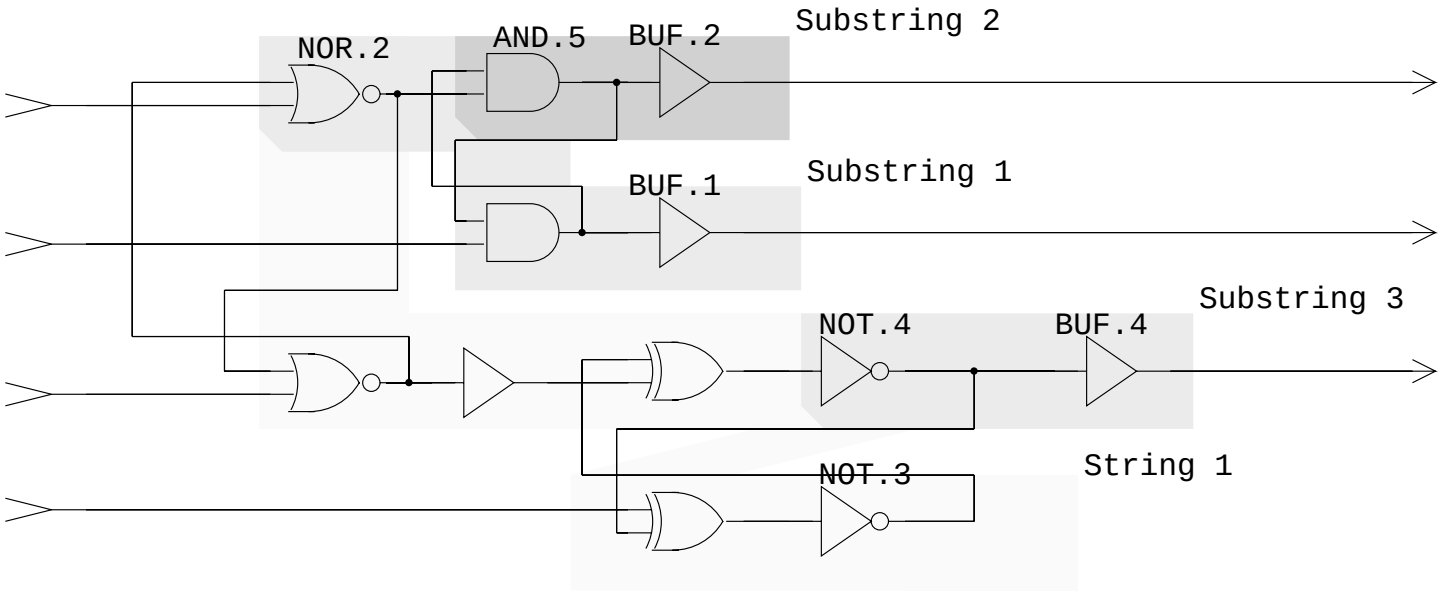


Figure 4: A complex string, consisting of one main string and three substrings

An important difference between SPAR and other systems is our use of signal position information to insert new tiles (and thus the strings) into the virtual page. The heuristics that govern the insertion of tiles use the connectivity of the unplaced string and the locations at which the string is attached to position the new string as a tile on the virtual page. This technique is described in **Find-dominant-connection()** and **Angled-insertion()**, and is illustrated in Figure 5.

For the tile being inserted, counts (n_{ct}) for each of the connection types (in→in, out→out, in→out, and out→in) are made. The largest count is used to determine which side of the existing tiles the new tile should be placed against. As illustrated in Figure 5, the two largest counts form the slope that determines the *axis of motion*, along which the new tile is inserted. The terminals contained in the placed tiles that connect to the new tile determine the *anchor point*, which fixes the axis of motion in the plane of the placed modules. A *pivot point* is established within the new tile. Newton's method is used to govern how the new tile moves along the axis toward the anchor point.

Once all partitions have been placed on their own virtual pages, the task remains to merge these pages to form the complete placement for the schematic. This is done by ordering the partitions, and merging them together one at a time (See **Merge-partition()** in Section 3.3.1). First the partitions are ranked based on their connectivity to other partitions, using a connection matrix like that used in the clustering process. The most highly interconnected partition is placed on the page, followed by the remaining partitions in descending order of connections. These succeeding partitions are placed about the finished partitions in the same manner as tiles were placed on the

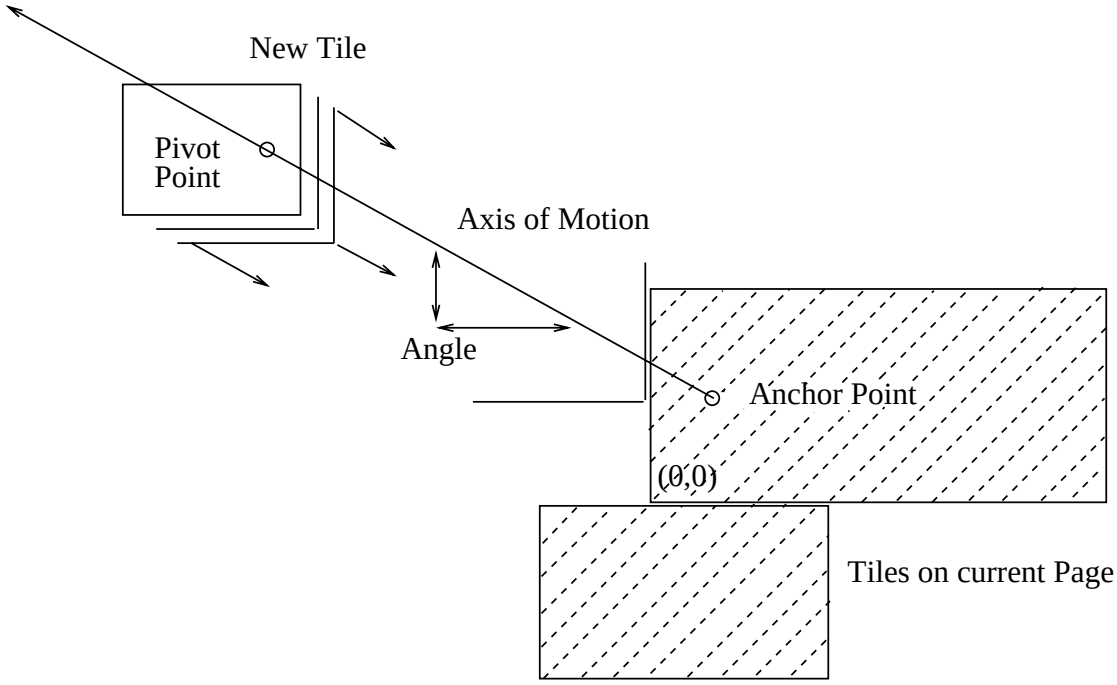


Figure 5: Tile Placement Example

virtual pages.

The inter-partition placement addresses the global features of the circuit which are not considered by the string placement algorithm. This is illustrated in the 4-bit ripple-carry adder of Figure 6. Here, a Rent's-rule partitioning was used, and the partition placement clearly shows the overall flow of information within the adder. The combination of the partitioning and the inter-partition placement rules combine to insure that the overall signal flow is maintained.

Once placement is completed, the modules are transferred to a separate routing tile space to prepare for the routing process, discussed in the next sections.

3.3.1 Placement Algorithms

Definitions:

white-space : Incremental amount of space to add for each terminal.

virtual-page$_p$: The tile-space containing the placed modules from $partition_p$.

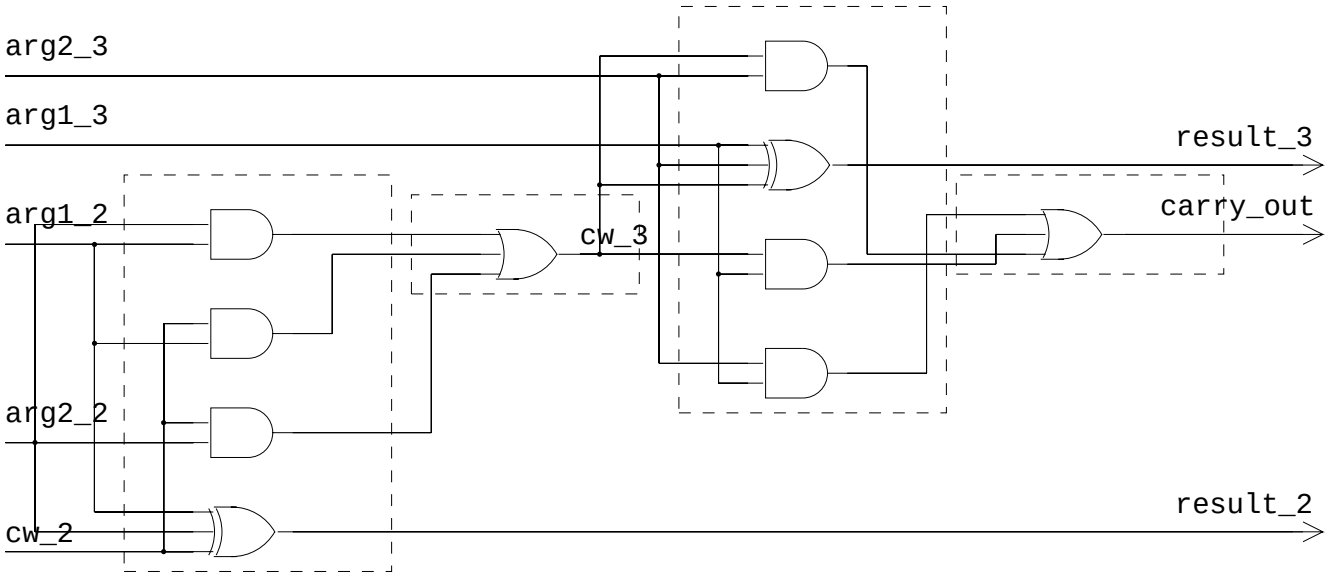


Figure 6: Last two bits of a 4-bit Ripple-Carry Adder showing partition placement (Rent-like partitioning)

Placement()
{
  /* Using the partitions developed during clustering,
     place all of the modules. (Note that system terminals are dealt with separately.) */
  /* Intra-Partition Placement: Construct and place the
     strings in each partition: */
  for p from 1 to number-of-partitions do {
    /* Locate the modules for the longest string in
       $partition_p$; Create a tile for this string, which
       is placed on the virtual page for $partition_p$. */
    $string \longleftarrow$ Form the set of longest paths in $partition_p$
      by doing a recursive depth-first search of the
      unplaced modules in p using each module in
      p as the head. Choose the longest, breaking

```
            ties by selecting the path with the highest
            out degree to unplaced modules in p

        Place the first module (string_1) at the origin,
            with an x offset of white-space × # terminals
            on the left side of m, and a y offset of white-
            space × # terminals on bottom of m;

        /* Place the remaining modules in string, using
           cornerstitching insertion [20] to place the string
           on p's virtual page. */
        Place-string(string, p);

        /* Find the longest string of unplaced modules
           remaining in partition_p; Create a tile for
           this string, and merge it onto the virtual page
           for partition_p. */
        while unplaced modules in partition_p do {
          string ⟵ Longest path of unplaced modules
                   in partition_p
          Place-string(string);
          Find-dominant-connections
              (string, p, anchor, pivot, angle);
          Angled-insertion
              (string, anchor point, pivot, angle, virtual-page_p);
        }
    }

    /* Inter-Partition Placement: Order and merge all
       partitions into one: */
    Order partitions from highest to lowest connectivity;
    Move most highly-connected partition to virtual-page_0
       (partition_0) directly;
    for p in descending connectivity order do {
       Merge-partition(p, virtual-page_0) };

    /* Map these module locations to the two new
       90°-rotated tile spaces for routing: (See Section 3.4)*/
    ∀ m ∉{system-terminals} do {
       Create a tile t_m matching the size of the icon
          that corresponds to m;
       Insert t_m into the routing space;
    }
}
```

```
Merge-partition(p, virtual-page_0)
{
  /* Determine the best side for insertion, and insert
     partition_p into virtual-page_0 */
  string ⟵ all modules in partition_p;
  Find-dominant-connections
       (string, 0, anchor, pivot, angle);
  Angled-insertion
       (string, anchor point, pivot, angle, virtual-page_0);
}

Place-string(string, p)
{
  /* From the position of the first module in string,
     recursively place all subsequent modules. */
  bot-mod, top-mod, cc-mod : Modules who's positions
       are key for constructing complex strings.
  last-m : The (just placed) module preceding m in string.

  for each unplaced module m in string do {
      Perform an iterative-deepening Best-First Search,
        looking for the deepest module in string (bot-mod)
        that is cross-coupled with m;
      Save the current m as cc-mod, and note the module
        top-mod that precedes bot-mod in string;

      Rotate m so its connection to the next module is
        to the right.
      if (m = bot-mod) {
        Construct a substring extending from top-mod
          from the unplaced modules in partition_p, recursively
          calling Place-string to place this new string and
          determine its y offset;
        Position m underneath cc-mod, using the offset
          returned by the recursive Place-string call, the
          number of terminals on top of m, and cc-space to
          calculate how far down to place m;
      }
      else /* Place modules to the right of m: */{
        Position m to the right of the last-m, aligning
          on the terminal that connects them;
        Allow space between m and last-m based on
          the # connections from last-m to m;
      }
      Update the bounding box for the string;
```

}
return the y offset based on the bounding-box info;
}

Find-dominant-connections
(*string*, *p*, *anchor*, *pivot*, *angle*)
{
/* Calculate from among the placed terminals in
p that connect to modules in *string* the best
point from which to extend an insertion axis. */
$\forall$ connected terminals t, t' s.t.
$t \in m$ in *string*, $t' \in m'$ placed in $partition_p$
for each connection type $ct \in${ out→out, out→in,
in→out, in→in } do {
$n_{ct} = \sum$ number of connections of type ct;
$(x, y)_{ct} = \sum(x, y)$ position of t_{ct};
$(x', y')_{ct} = \sum(x, y)$ position of t'_{ct};
}
Use the largest number of connections n_{ct} to
determine which side of $partition_p$ the *string*
will be inserted from; (where out→out
connections $\Longrightarrow$ bottom, out→in $\Longrightarrow$ left,
in→out $\Longrightarrow$ right, in→in $\Longrightarrow$ top) */
If there are no ties:
$pivot \longleftarrow \frac{(x,y)_{ct}}{n_{ct}}$;
$anchor \longleftarrow \frac{(x',y')_{ct}}{n_{ct}}$;
angle ⟵ slope based on the ratio of the
largest two connection counts;
otherwise tiebreaking cases are used;
}

Angled-insertion
(*string*, *anchor point*, *pivot*, *angle*, $virtual\text{-}page_p$)
{
Use Newton's method to choose points along the
line defined by *anchor* and *angle* to place the
pivot point of the tile containing *string*; (See Figure 5)
}

3.4 Global Routing

Routing in SPAR is broken into two phases: global routing, where the general location of runs and corners in the nets is calculated, and local routing, where these locations are manipulated as combinations of constraints that need to be satisfied. The global router determines the number and location of corners in the broad sense, and the local router places these corners in relation to each other, fixing their position. The combination of these two techniques aims at maximizing the readability of the resulting diagram.

The global route is not only required to connect all points of all nets, but also to yield an overall traceable result. We do this by performing a heuristic search that makes multi-point node expansions using a cost metric based on both the length and the congestion of the route. We map proposed routes directly into the routing tile space, thus enabling the calculation of the congestion of the routes.

The routing tile space is actually composed of two superimposed, 90°-rotated tile spaces, which contain the modules as obstacles to the route. The unobstructed tiles are linked into *paths* to form the global route for each net. The use of two tile spaces takes advantage of a feature of the tile space data structure: free tiles, those without obstructing contents, extend as far in the horizontal (or vertical) direction as possible. By using two 90°-rotated spaces, the unobstructed horizontal or vertical area containing any arbitrary point can be obtained quickly. Each of these tiles contains a structure with space for a hash table for partial route information, a pointer for a module, a list of nets that use this tile in their final global route, a list of corners that are located in this tile, and a congestion value.

Global routing is accomplished in two phases. The first phase of the search discovers the best routes without congestion considerations, and these routes are used to approximate the congestion of the routes. In the second phase, we continue the search, incrementally updating the costs for the paths discovered to include congestion information.

Rather than recalculate routes after the congestion is known, all partial paths are maintained in hash tables as the search progresses. In this way a change in congestion of a particular tile will only require the cost for the paths that pass through it to be recomputed, rather than the search having to begin again from scratch. This technique permits us to avoid re-running the time-consuming search process.

For the search itself we use an A* [5] best-first heuristic search algorithm. This algorithm uses a cost metric composed of both a cost of the path so far, and a low estimate of the cost of completion for that path. The use of a low estimate guarantees that the search will result in an optimal solution.

3.4.1 The Search Process

The search process is actually a parallel set of searches, one for each terminal of each net. For each terminal, a partial route (*path*) is constructed until the path for that terminal connects to

another path of the same net. A path that connects is said to be *completed*. The collection of paths associated with a given terminal of a net is referred to as an *expansion*. At each move, for each expansion, the best path seen is selected, and the last tile on that path is expanded, creating several new paths. When the best path selected is complete, then the search for that expansion is terminated. These actions are detailed in **Make-next-move()** in Section 3.4.4.

Figure 7 shows the expansion of one path for a terminal in the SN7474 example. The example depicts an expansion for a path for net *y* of the expansion associated with the input terminal of module *NAND.1*. This path contains three tiles, two horizontal (*b* and *c*) and one vertical (*1*). The last horizontal tile on the path (tile *c*) is being expanded into the vertical plane, and can expand into the four vertical tiles (*2, 3, 4,* and *5*) marked by *?*. Thus the one path will have a vertical tile added to it, and three copies of the path will be made, with each of those having a different vertical tile added to it. The resulting paths to be considered again for expansion will be *b-1-c-2*, *b-1-c-3*, *b-1-c-4*, and *b-1-c-5*.

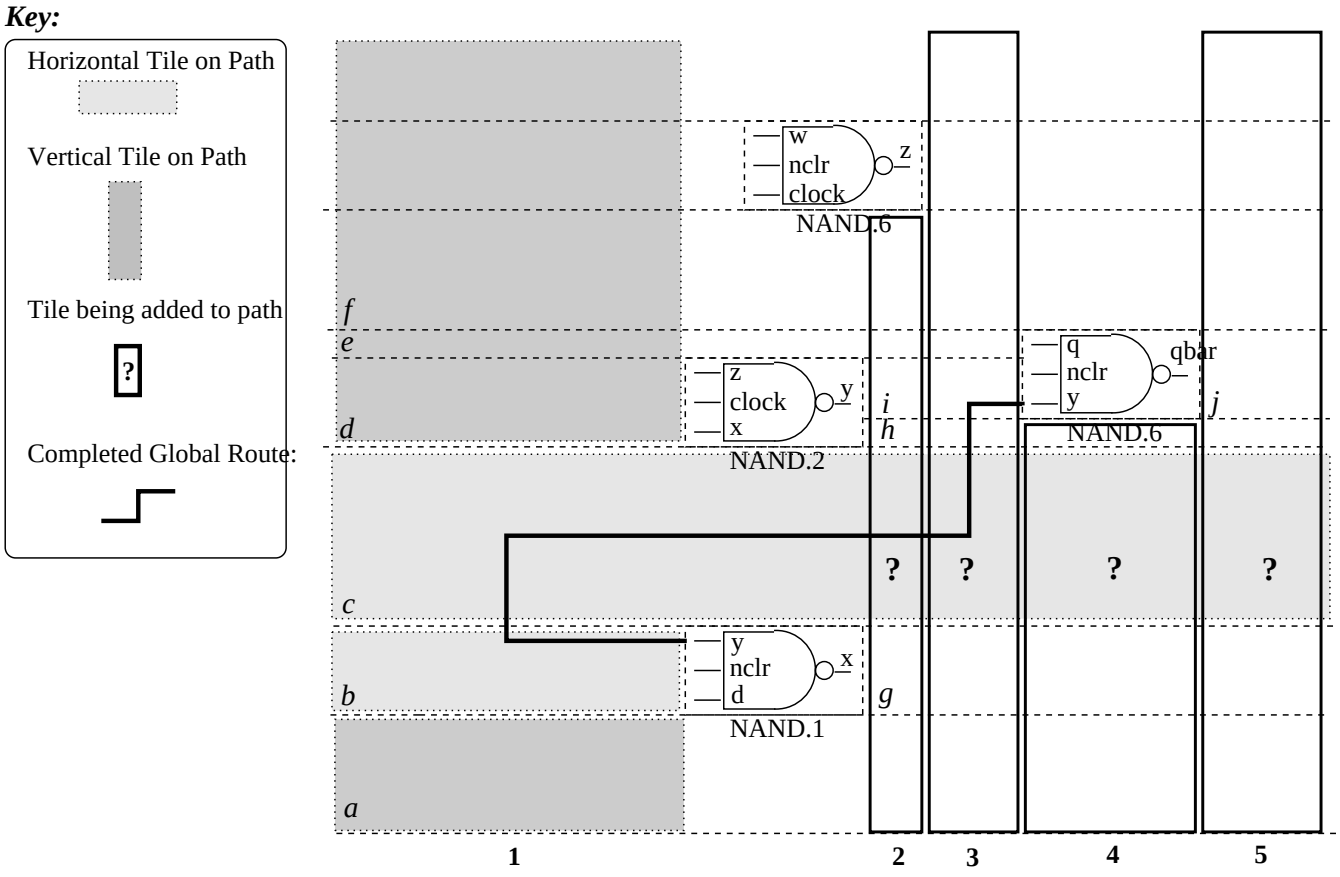


Figure 7: Path Expansion for one path of net *y* for the SN7474.

3.4.2 The Cost Estimation Metric

The key to the success of the search process is the nature of the cost estimate used. The cost estimate reflects the relative importance of the qualities of the finished route, such as low congestion and few turns. The estimate is structured so that all costs are directly comparable. An under-estimate of the distance-to-go for all partial routes insures that the A* qualities of the search are maintained.

The completion of the first phase of the search creates a first–pass at the route for all nets: It is a reasonable approximation to the best routes, in that the paths selected have been optimized for the number of corners and the length of the routes. The dark line in Figure 7 shows the completed global route for net *y* of the SN7474. This route consists of tiles *b, 1, c, 3,* and *i*.

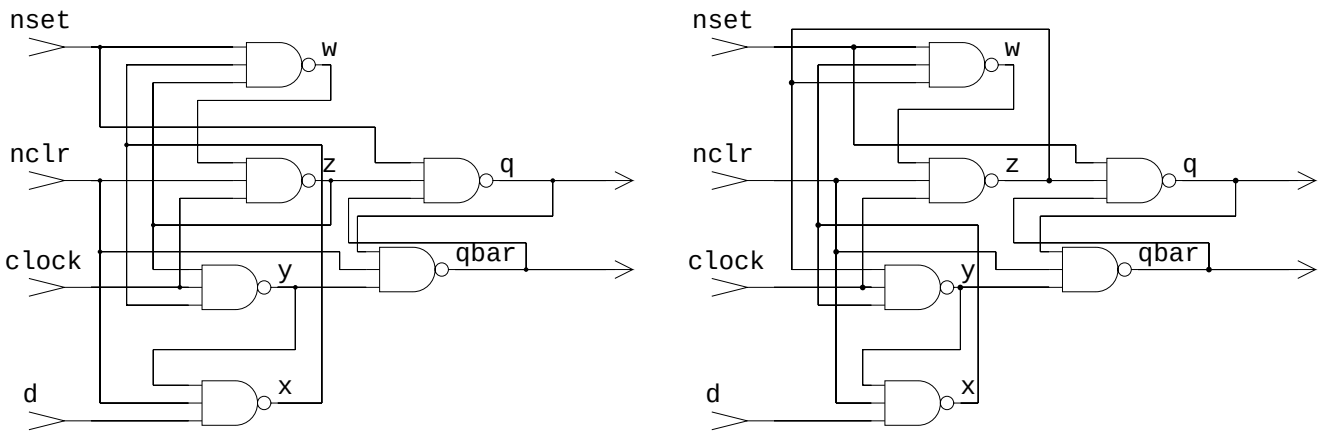
3.4.3 Using Iterative Refinement to Solve Congestion Problems

Once the first–pass paths for all nets have been selected, the path costs for all nets are recalculated to include congestion. We re-cost both the completed paths, and those partial paths that were in competition but did not complete. The congestion for each tile is calculated based only on the completed nets, and this congestion cost is added to all paths of all nets that go through that tile.

We resolve congestion by using an iterative refinement technique. The technique consists of ordering the nets by their congestion values, and working from the highest-cost net to the lowest, ripping up the route and rerouting it using congestion considerations. As all partial paths are maintained, the expense of this operation is in recosting the partial paths. Depending on the congestion problems, the old route might well be reselected, but this is not assured.

The value of re-routing the nets using congestion factors can be seen in the two examples of Figure 8. Figure 8a shows the completed SN7474 flip-flop without congestion considerations, and Figure 8b shows the same schematic using congestion. The difference between the two figures is the positions of nets *nset* and *x*. The right-hand figure is clearly easier to trace. A more general example is presented in Figure 9 that illustrates the use of the congestion metrics. The parallel binary adder-subtractor unit depicted in this figure shows a clear and even distribution of the net corners and connections.

Once the set of lowest-cost paths for all nets has been discovered, the task remains to refine these into a traceable set of routes. As the global route only determines the tiles in which to run the nets, the local router must locate all corners for each net, given these sets of tiles.



a *b*

Figure 8: Routing for the SN7474 with congestion turned off (a) and on (b) (Slope-based partitioning)

3.4.4 Global Routing Algorithms

Definitions:

path: An ordered list of tiles in alternating horz/vert sequence. Each path corresponds to one possible global route for a piece of the net to which it belongs.

expansion: The data structure associated with each terminal that contains the set of paths expanded for this terminal. The paths are contained in a hash table, and are ordered based on their cost. An expansion is said to *terminate* whenever the best path for the expansion contacts any path of another expansion belonging to the same net.

forward-est: Constant (0.80) that insures that the forward distance estimate is always less than the actual forward distance.

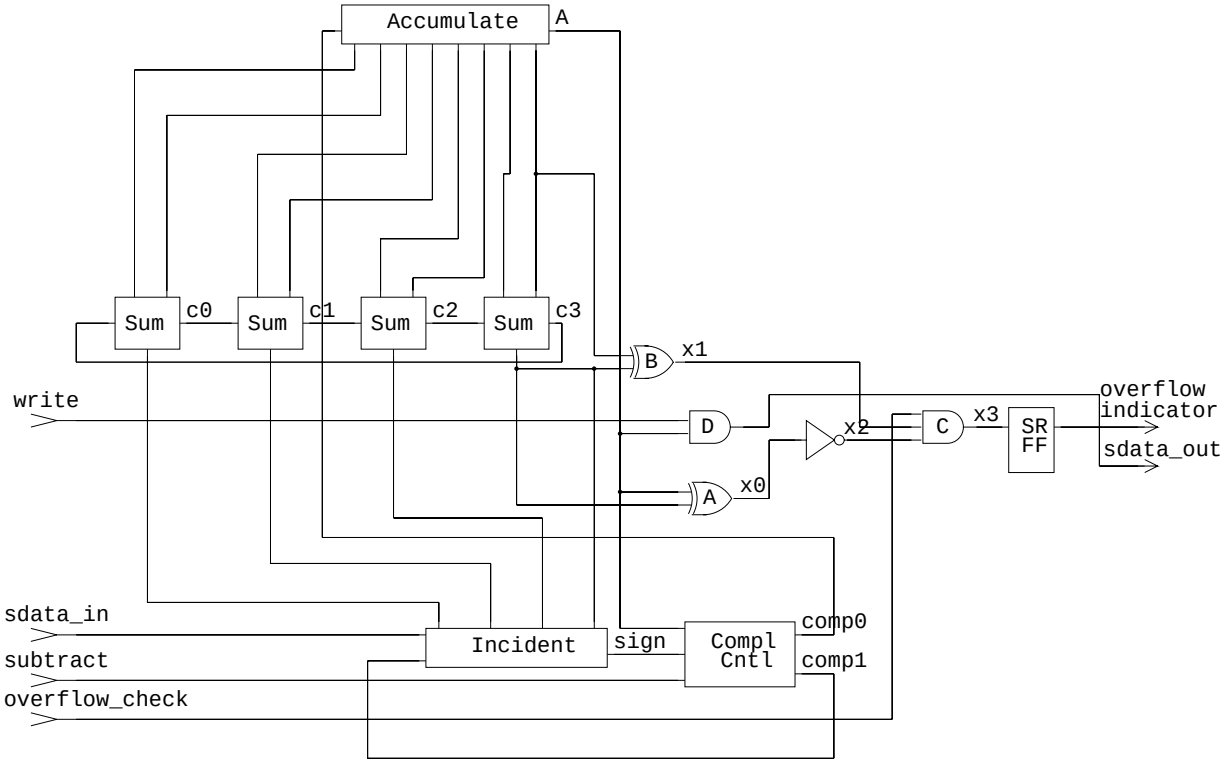


Figure 9: Parallel binary adder-subtractor unit, using size-based partitioning

Global-routing(*nets*)
{
  /* Develop global routes that connect all terminals
     of the *nets* */
  $\forall$ terminals $t \in$ nets $n \in$ *nets* {
    Create an expansion node e, expanding into the
      nearest horizontal tile;
    Add all non-completed expansions to the
      active-expn-list;
  }

  /* Perform an A* search to find the minimal paths */
  while ($|$*active-expn-list*$| > 0$) do {
    $\forall$ $e \in$ *active-expn-list*, **Make-next-move**(e); }

```
  /* Generate congestion costs for the initial routes: */
  ∀ paths p ∈ e ∈ n ∈ nets Path-cost(p);
  Return all expansions e to the active-expn-list;

  /*Iteratively improve the route until all nets have
    been relaid once. */
  for each net n in nets do {
      ∀ e ∈ n, Make-next-move(e);
      ∀ paths p ∈ e ∈ n ∈ nets Path-cost(p);
  }
}


Make-next-move(e)
{
  /* See if we can contact a path on the same net: */
  expn-path ←— lowest-cost path of all paths in e;
  Find the set of expansions E' s.t. e ≠ e',
        all e' ∈ E' are on the same net as e, and
        the last tile on expn-path contacts any path p
        ∈ e' ∈ E' (may be ∅);
  if (E = ∅ {
    expn-tile ←— last tile on expn-path.
    ∀ tiles t s.t. t ∉ expn-path, t ∉ same plane as
                  expn-tile and t intersects expn-tile do {
      Create a copy of expn-path, adding t to the
            end of the new-path;
      Path-cost(new-path)
      Install new-path into the hash table of cost-ordered
            lists contained in e;
    }
  }
  else /* There is at least one possible completion */ {
    e completes - remove e from the active-expn-list;
    Foreach e' ∈ E' ∈active-expn-list do {
        if(lowest-cost path in e' connects to expn-tile) or
          (e' is the last active expansion for net n)
          /* e' completes: */
          Remove e' from the active-expn-list;
    }
  }
}


Path-cost(p)
```

{
/* Compute the cost for path p, based on the contents of the tiles that comprise it. This cost has three portions: a *Congestion*, an estimate of the manhattan length of the path discovered so far(*Known*), and a forward (under)estimate of the distance to the nearest terminal. (*Forward*) */

($\bar{x}$,$\bar{y}$) ⟵ position of the last known corner used in the *Known-length* calculation;
(*termX*,*termY*) ⟵ location of the terminal nearest to ($\bar{x}$, $\bar{y}$) on the same net as p;

Congestion ⟵ $\sum_{n=0}^{|p|}$(2 × no. of corners in tile$_n$) + (no. of nets crossing in tile$_n$);
Known ⟵ $\sum_{n=0}^{|p|-1}$ avg. distance tile$_n$ to tile$_{n+1}$;
Forward ⟵ [$|termX - \bar{x}| + |termY - \bar{y}|$] × **forward-est**;

return(*Congestion* + *Known* + *Forward*);
}

3.5 Local Routing

We use a constraint-propagation approach for local routing. This allows us to directly address the aesthetic placement of corners to maximize traceability of the resulting schematic. Our approach arranges corners of routes such that all corners of the same net at the same x or y position share the same global variable for the value of that position. Such corners are said to be *linked*. Thus a single change to the value of a given position is automatically propagated to all affected corners. This allows us to evaluate one tile at a time, as the changes made to each net in that tile will automatically be propagated to the relevant corners of the net in other tiles.

The solution to traceability at this level is found by imposing an ordering relation on the corners contained in each tile. We place lines by applying our *Rules of Easy Reading* (Figure 10) to the corners in a given tile. The horizontal tiles as typified by Figure 10a are evaluated to determine the ordering for the vertical lines they contain. Similarly vertical tiles are evaluated to determine the ordering of horizontal lines as typified by Figure 10b. These rules insure that unnecessary line-crossings are avoided.

3.5.1 Mapping the global route into corners

The local routing process begins by first mapping the global routes of each net into a linked sequence of corners, where the actual (x,y) positions of the corners are not known. Rather, these positions

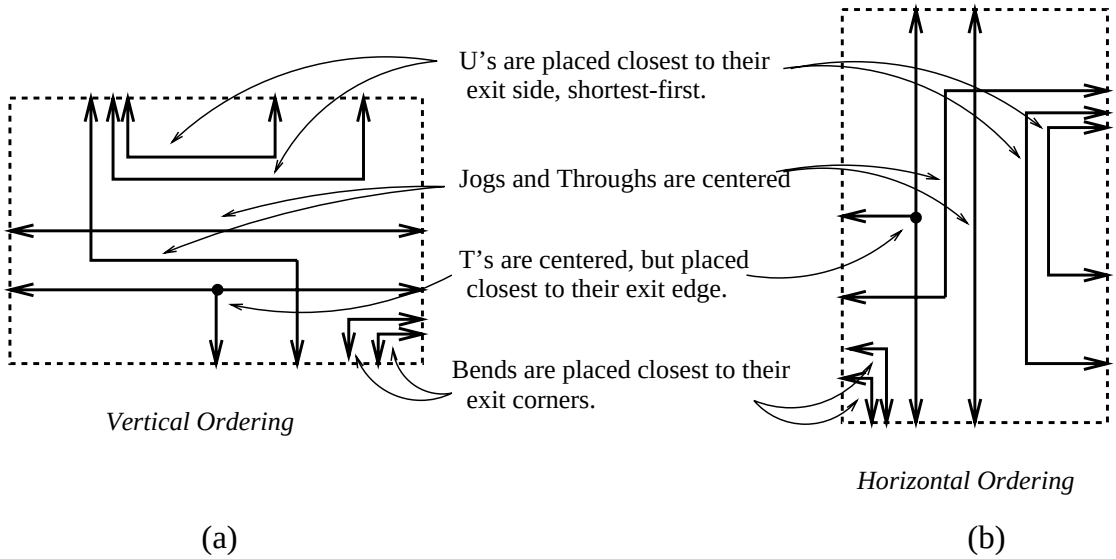


Figure 10: *Ordering Rules for Easy Reading*: The relative placement of segments

consist of *ranges*, where a range is based on the limits determined by the paths used in the global route. These ranges define the x and y positions where the corners may be placed. Corners are linked by sharing the same range to record their x- or y-position. In this way, any restriction made to one corner will immediately be reflected in the linked corners. For example, any change to *Range 2* of Figure 11 will affect both *Corner 1* and *Corner 2*. The minimum and maximum values for these ranges are initially set by the widths of the tiles in the global route, and are created as the global route is mapped into corners.

The results of the corner creation and linking process are illustrated in Figure 11. Here the global routing tiles for a three-terminal net are presented with a representation of the local route under construction. This net has two floating corners, *Corner 1* and *Corner 2* linked to the three terminals which make up the net. These two floating corners are linked by three ranges, two of which are vertically-oriented, and one of which is horizontal. *Range 1* and *Range 3* initially take on the values $[y3..y4]$ and $[y1..y2]$, but as they are linked to terminals, their ranges are further restricted to the y positions of the terminals. *Range 2* will take on the range $[x1..x2]$, as determined by the width of the vertical tile in the net's global route. This range will be refined in subsequent operations.

There is a need to augment the set of tiles for each net beyond those found in the global route. This is to insure that the separation algorithms discussed below can utilize as much of the routing space as is practical. Since tiles are designed to run as horizontally (vertically) as possible, the placement of obstacles can 'sliver' [20] the space, thus the ranges developed from the global route represent only a minimal set of ranges in which the lines can be run. Depending on the directions that the bends in the routes take and their proximity to obstacles (placed modules), the initial ranges developed may need to be expanded into the nearby tiles. This is outlined in **Augment-ranges()** included in Section 3.4.4.

Once all ranges have been augmented to include their maximum reachable area, the ranges for

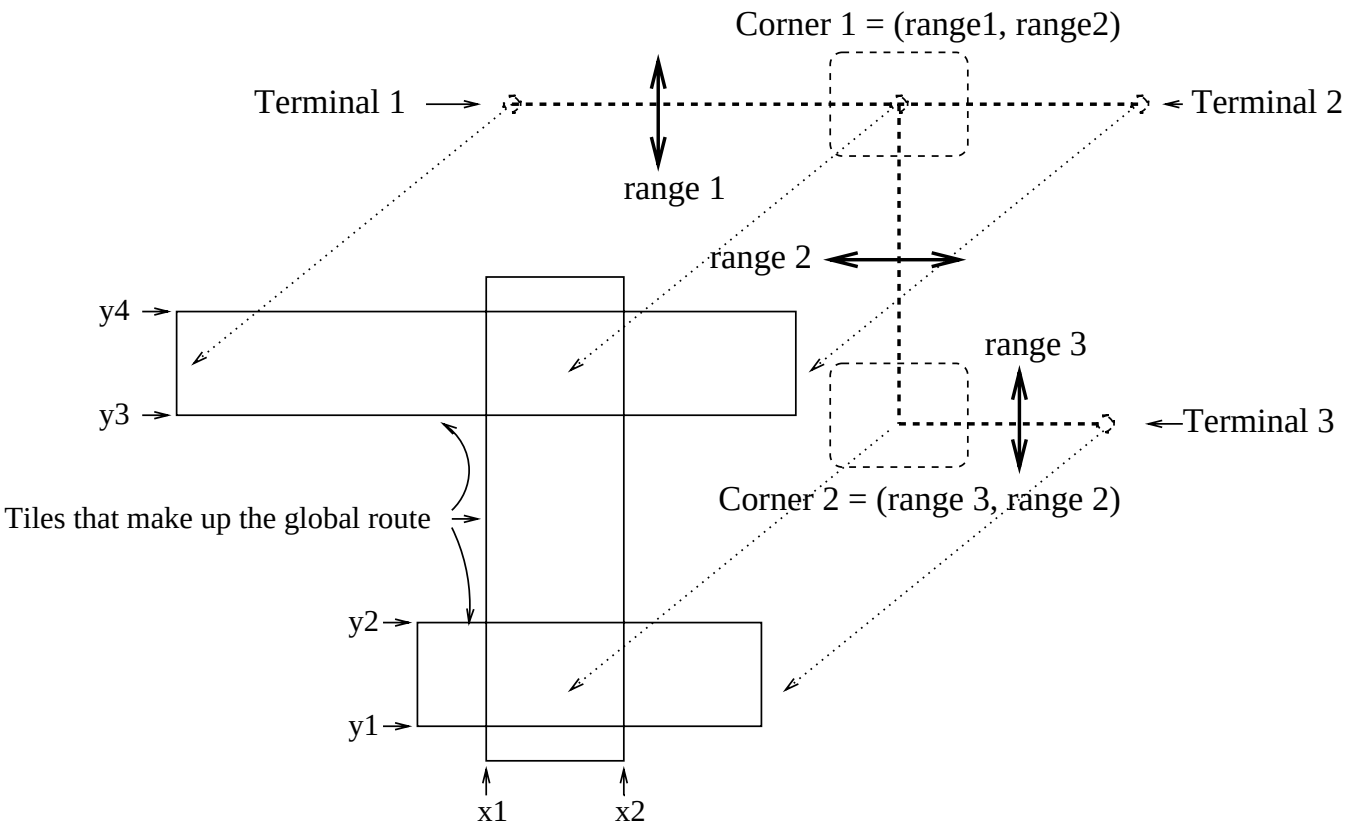


Figure 11: Mapping a global route to a set of corners.

each corner need to be examined to center the corners in their range, and to insure that no wire overlaps another.

3.5.2 Separating Overlapped Ranges

The range separation process performed on each tile, starting with the most congested tile, breaks down into three steps, *range-ordering*, *dependency-graph creation*, and *space distribution*. *Range ordering* is a mapping of aesthetic rules to the given ranges such that the desired relative placement of each range is known. *Dependency graph creation* is the creation of a list of the interfering ranges, sequenced by the given range-ordering. *Space distribution* is the mapping of a given ordered list of ranges to the overall space which they can occupy such that each range takes on a unique (non-overlapping) value. These three actions comprise the bulk of the local routing work, and are outlined in **Order-and-separate-overlapping-ranges()** and **Form-dependency-graphs()** described in Section 3.5.3.

Ranges within a tile are ordered by first classifying the range as either a jog, bend, T, or U as shown in Figure 10. This classification, along with the location and connection information, is used to order the range relative (top→bottom/left→right) to the other ranges in the tile.

We now use the left→right/top→bottom ordering of the ranges to develop the necessary dependency graphs. These graphs divide the set of ranges into subsets, containing groups of ranges that must not overlap. Any given tile can contain multiple graphs, and ranges may occur in more than one dependency graph. A range z is said to be dependent on another range x ($z \Rightarrow x$) if the ordering process places range x before z, and the line segment associated with x overlaps any portion of the line segment associated with z. To resolve the overlap problem, ranges need only be compared to those they are dependent upon.

Each dependency graph is used to restrict the ranges contained in the graph to values that do not overlap and yet maintain the ranges' left→right/top→bottom ordering. The dependency graphs are solved for overlap from longest graph to shortest graph. In this way, ranges that appear in multiple graphs are adjusted to match the tightest constraints first.

Due to the propagation of the corner range values, the process of solving for the corner positions within one tile affect the solutions for the corners in the neighboring tiles. This is because the ranges that define the x- and y-positions of the corners are shared among linked corners. As each tile is processed to resolve the overlap for the corners contained, we reduce the range available to each corner. When the process completes, all ranges are known to be non-overlapping, and can be easily resolved to a point by centering the range on its legal value. This provides the maximum spacing between lines, so as to further enhance the traceability of the diagram. The details of these algorithms are included in Section 3.5.3.

The local router completes when all corners have been fixed to individual points. This corner information can then be used in conjunction with the original placement to optimize the positions of the the remaining I/O modules on the diagram.

3.5.3 Local Routing Algorithms

Local-route(*nets*)
{
  /* Create the corner and range structures from
     the best paths for each expansion of each net:*/
  ∀ $n \in nets$ do {
     Map the global routes for n into constrained
          corners, tie to the tiles that comprise global route;
     Link the corners in the tiles where the paths for
          the different terminals in n overlap;
     /* Expand the range constraints in n to their
        maximum values: */
     Augment-ranges(n);
     Remove any duplicate corners and ranges;
  }
  /* Now that all nets have been mapped to corners
     and to the two routing tilespaces, Resolve the
     range constraints to create a completed route. */

```
  Tiles-used ←– All tiles containing one or more
                corners, sorted by number of corners
                in decreasing order;
  Order-and-separate-overlapping-ranges(Tiles-used);
  ∀ ranges r ∀ tiles t :
    Restrict each range to the average of its min and
    max values;
}

Augment-ranges(n)
{
  /* Augment the range constraints for all corners
     in net n to their maximum values. */
     ∀ tiles t used by net n do {
       /* Look to see if there are other adjacent tiles
          that can be used. If so, then modify the corner
          limits, and do the book keeping for the corner
          structures. */

       For some corner c used by n in t do {
          /* Define the x- and y-extent over which
             this corner may be placed: */
          if vertical tile(t) then
            y-extent is defined by the corners c', c''
                 linked to c with the maximum
                 and minimum y values;
            x-extent is defined by the max/min x
                 values of the corners outside t to
                 which c' and c'' are linked;
          else /* For horizontal t: */
            x-extent is defined by the corners c', c''
                      linked to c with the maximum and
                      minimum x values;
            y-extent is defined by the max/min
                      y values of the corners outside t to
                      which c' and c'' are linked;

          ∀ tiles t' within the area defined by
                    x-extent × y-extent s.t. ∃ no tiles
                    containing modules between t and t' do {
            Add c to t';
            if (t, t' are vertical tiles) Expand the
              x-range of c to include the x-dimension of t';
            else /* t, t' horizontal tiles */
              Expand the y-range of c to include the
```

```
                y-dimension of t′;
            }
        }
    }
}
```

Order-and-separate-overlapping-ranges(T)
{
  /* Separate the ranges in each tile in T. */
  Sort T in order of decreasing congestion;
  $\forall$ tiles $t \in T$ do {
    $\forall$ nets n with corners in t,
      find the set of ranges R =
        $\{r_n |$ r_n has orientation perpendicular to tile t *and*
            r_n represents the segment of the route
            completely contained in t $\}$
    $\forall$ $r_n \in R$ do {
      Characterize r_n by the side(s) where n
        exits t (left, right, up, down);
      Characterize r_n by the net's configuration
        in t (U, T or Z) (See Figure 10);
    }
    Order R based on the configurations and their relative
      placement, as per the *Rules of Easy Reading*
      of Figure 10;
      /* For horizontal t, this is a top→bottom
        ordering; For vertical t, a left→right ordering. */
    dependency-lists ⟵ **Form-dependency-graphs**(R);

    /* Starting with the largest dependency list,
       evenly distribute the available space among the
       ranges in each dependency list */
    $\forall$ *r-list* $\in$ *dependency-lists* in order of decreasing size
    do {
      space-avail ⟵ Difference between min point
                      of $r\text{-}list_{first}$ and the max point of $rlist_{last}$;
      slot-width ⟵ $\frac{space\text{-}avail}{|r\text{-}list|}$;
      Walkthrough *r-list* sequentially limiting
          each range encountered to the *slot-width*. This
          is done in such a way as to distribute the available
          space evenly among the ranges in *r-list*.
    }
  }
}

Form-dependency-graphs(*ordered-ranges*)
{
  /* Create a list of ordered range lists to maintain
     the left→right ordering of *ordered-ranges*,
     but contain only ranges who's *extents* overlap.

```
    The extent of a range is the min and max points over
    which that the range has effect, eg. a vertical
    tile contains (x-)ranges that must be separated. Each
    x-range extends from some minimum y to some maximum y
    point. Wherever y-extents overlap, there is a dependency. */
  ∀ ranges r in ordered-ranges do {
    ∀ ranges r' in ordered-ranges s.t. r' is
              after r in ordered-ranges do {
      if (extent(r) overlaps extent(r'))
        mark r' as dependent on r;
    }
    Construct a new ordered range list r-list for each
      unique path through the dependency graph created;
  }
  return(Set of r-lists discovered );
}
```

3.6 Incremental Placement of I/O Modules

The modules for Input/Output terminals for a schematic represent a special sub-problem for ASG systems, in that they have a different set of rules governing their placement. I/O modules are placed in columns on each side of the page. What makes this sub-problem interesting is that it is dependent on the placement and routing of the internal portion of the figure. Further, the I/O module must connect to its net with as little congestion as possible. Therefore, the routing information for the placed modules is used to guide the insertion of the I/O modules into the existing routing space. This is followed by the routing of the placed I/O modules.

The major obstacle to any incremental placement and routing technique is how to change a schematic and still maintain the partial schematic through the changes. It is important that all valid information is retained, while all invalidated information is dropped. The complicated aspect of this task is the maintenance of the routing information.

3.6.1 Use of Routing Information to Place I/O Modules

For an input I/O module, traceability is primarily dependent on its path to its leftmost destination introducing as few bends as possible to the net. Therefore, the best place to locate an I/O module is to align the module on a corner or terminal point for its net. However, this placement rule is too simple, since it may cause module icons to overlap. This placement is further complicated by the fact that some destinations may be completely blocked from the terminal column. Blockage implies that the destination cannot be reached without introducing one or more corners into the diagram.

The overlap problem can be seen in the placement of the input I/O module *data2c* in Figure 12.

Both blockage and overlap complicate the placement of I/O module *strobe1g*. Its destination is blocked by the placement of gate *NOT.1*. Even when gate *NOT.1* is ignored, lining *strobe1g* on its destination results in an overlap problem with I/O module *data1c*. We note that in the placement of I/O modules *data2c* and *strobe2g*, either module could be aligned, as there are no other factors to guide their placement.

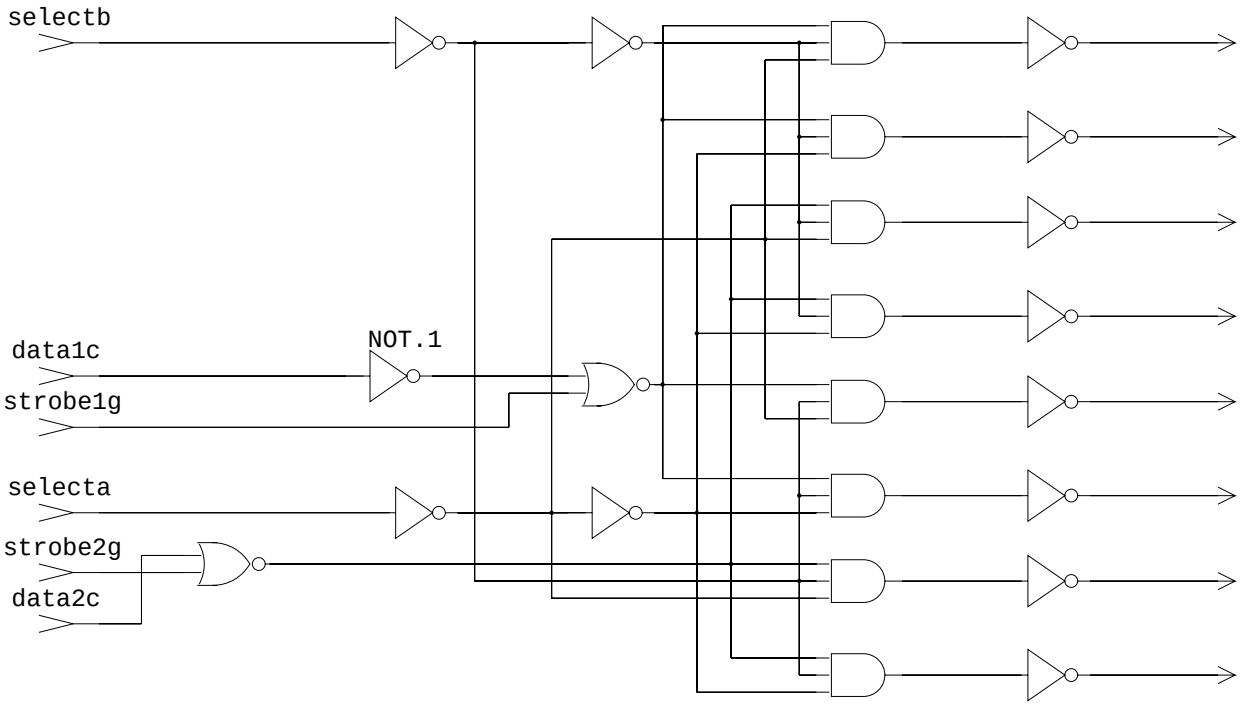


Figure 12: Completed SN54155 demultiplexer using slope-based partitioning

One of the more interesting aspects of this alignment problem is how to tell that a particular destination point (or range) is *not* obstructed. Without evaluating the diagram, it is not easily known which connections to existing modules are obstructed. These obstructions must normally be discovered by examining the positions of all objects on the page. This is another place where the tile space is useful, as it reflects the ability of one point to directly reach another point. Since modules define their own tiles, if both points lay within the same routing tile, then there are no obstructing modules between the two points.

3.6.2 Maintaining Information During Insertion

As mentioned above, the difficult issue for incremental placement and routing is information management. Determining where global routes are to be run is expensive, so recomputation of this information should be avoided wherever possible. On the other hand, local routing can be done quickly, and is recomputed after all insertions have been made.

As part of the normal cornerstitching insertion, whenever the tile for a new I/O module is inserted into the diagram, some of the existing tiles are split, sometimes repeatedly. As mentioned in Section 3.4 tiles contain the global route information. This information must be updated whenever a split or merger of tiles occurs, which happens repeatedly during the tile insertion process. At worst this involves duplicating all paths for all nets contained within the tile (ie. duplicating the partial-path hash table), but more often many of the would-be-duplicated paths no longer exist, and should be deleted. This turns out to be an easy test, as a tile in the middle of any path must overlap with both tiles to which it connects. Even though there are two tilespaces being altered as the tile for the I/O module is inserted, if all routes are maintained at each insertion, valid routing information is retained.

This data maintenance is implemented through additions to the cornerstitching algorithms to allow a content-dependent function to be called at each tile split or merger. Specifically, whenever a tile is split, it is reduced in size, and its corners are stitched to a new tile which fills the empty space. The global routing data contained in the paths that use the original tile is queried to see which paths now make connections using the new tile, and which still make connections using the (reduced) original tile. The data structures for each tile and each path are updated appropriately.

Once all of the I/O modules have been inserted into the routing space, the cost estimates for all paths of all nets are recalculated. The terminals connected to the I/O modules are added to the nets, so that they may be routed to the existing terminals without difficulty. We complete the routes for these nets by creating new expansions for the I/O modules, and allowing these to compete with the existing paths for the interior expansions. This ensures that the new routes discover the best connections to the interior of the diagram. As the information within the routing space is valid, this is a simple process of starting the search process of the global router again. This is followed by local routing, completing the diagram.

This section presented the details of the algorithms that comprise SPAR. In the in following section we evaluate how well SPAR meets the stated goals of schematic generation: traceability and functional identification.

4 Judging the Results

The difficulty in developing schematic diagrams is answering the question "Did we do a good job?". The quality of schematics is based on their usefulness, which is in turn subject to aesthetic constraints. As Smith stated: "The criterion of 'goodness' depends mostly on ease of comprehension" [9]. Unfortunately, this criterion is subjective, unlike other placement and routing problems in VLSI CAD where the evaluation criteria used are based on the physical properties of the circuit.

The subjectivity of the quality of a schematic is difficult to avoid, but it can be limited. We model the quality of the schematic by quantifying the *badness* of the diagram in such a way as to be able to make relative comparisons between two diagrams for the same circuit. As we have developed our system around the two central themes of *functional identification* and *traceability*, we also evaluate our work by measures of these qualities.

To model the badness of a schematic, we simplify *functional identification* to a model where we expect the most tightly-connected modules to be placed near each other, and for the direction of the signal-flow to be correct. The measure we use for this we call the *Manhattan Pairwise Sum*(MPS). We simplify *traceability* to two measures of congestion: the number of corners within the diagram, and the average density of the corners on the page. The average congestion density gives a measure of how cluttered the diagram is, and the overall congestion count gives a measure of the general complexity of the routes.

The MPS relies upon the calculation of a Distance Matrix, which parallels the Connectivity Matrix of Section 3 with each entry containing a weighted sum of the Manhattan distances between each pair of modules in the design. The MPS is a sum over all module pairs, where the pairwise value is formed by dividing the appropriate Connection Matrix entry by the corresponding Distance Matrix entry. By combining the MPS value with measures of the traceability of the design, we form an accurate relative scaling for different schematics representing the same design. The total number of corners, the average density of their distribution throughout the diagram, and the MPS measurement are each scaled and added to form a *badness value* which express the relative quality of a particular diagram. Equations 1 and 2 describe these relations in detail. The definitions used to develop the *Badness Value* are listed in Table 1 for reference.

$$ManhattanPairwiseSum = \sum_{i=0}^{N} \sum_{j=i+1}^{N} \frac{Connectivity_{i,j}}{SignalFlow\,(M_i, M_j) \times [\; Xsep\,(M_i, M_j) \;+\; Ysep\,(M_i, M_j)\;]} \tag{1}$$

$$BadnessValue = \frac{4}{MPS} \;+\; 12 \times Avg.CongestionDensity \;+\; \frac{CongestionCount}{4 \times No.ofTerminals} \tag{2}$$

Symbol	Reference	Notes
M_i	Module$_i$	i is the row/column index of Connectivity Matrix
$Connectivity_{i,j}$	Connectivity Matrix entry	Number of connections between modules M_i and M_j
N	modules in the design	Determines the size of Distance and Connectivity matrices.
$Xpos\,(M_i)$, $Ypos\,(M_i)$	X-, Y-Axis position of M_i	
$SignalFlow\,(M_i, M_j)$	positive if signal flow is left→right, negative if signal flow is right→left.	= -1 if $(Xpos\,(M_i) < Xpos\,(M_j))$ **and** (M_i does not drive M_j) = 1 otherwise
$Xsep\,(M_i, M_j)$, $Ysep\,(M_i, M_j)$	Separation between M_i and M_j	Center-center distance along respective axis
$CongestionCount$	total number of corners	
$No.ofTerminals$	total number of terminals	Counted among all modules in the design
$Avg.CongestionDensity$	avg. number of corners per-unit-area	Counted among all tiles containing one ore more corners

Table 1: Definitions used to develop the *Badness Value*

The badness value is important in that it allows diagrams to be compared without requiring a time-consuming human analysis. This value gives a good ordering to the schematics, and more particularly gives a number to the relative quality of any two schematics.

The results of a comparative study for one design are presented in Table 2. These results were generated by varying the qualifiers to the three partitioning schemes discussed in Section 3. This study shows the relative badness values for twelve variations of an SN54155 Demultiplexer. To correlate the badness value to an image, the best diagram is shown in figure 12.

Since the badness measure is a weighted sum of each of the MPS, average congestion density and the congestion count, simply having a lower congestion count does not necessarily make for a 'better' schematic. This can be seen in the *Rent's Rule, ratio=2.0* and the *Rent's rule, ratio=2.5* examples, where the overall congestion and the average congestion density (respectively) are smaller than those for the diagrams that have better overall badness values.

In general, the slope-based partitioning worked best [23], and in many instances the Rent's rule-based partitioning was also effective, especially using the ratio of 2.5 connections-per-module. We discovered that sized-based partitioning is only effective if the proper partition size is known beforehand. Clearly, judging the quality of a schematic is not simple. We have found that none of the metrics alone gave a good indication of schematic quality. However, we did find that the composite badness value does match our subjective evaluation for a large number of examples.

Besides evaluating the quality of the generated schematics, it useful to know how fast the system runs. The "wall-clock" time to produce the figures in this paper are listed in table 3. The number

SN54155 Demultiplexer

No. of Terminals = 78

Partition Type	Qualifier	Manhattan Pair-Wise Sum	Average Congestion Density$\times 10^{-2}$	Overall Congestion Sum	Badness Value	Badness Differential	Figure No.
Slope	–	3.33	2.47	148	1.97	0	12
Size	size=2	3.43	2.49	168	2.00	0.03	–
Rent	ratio=3.0	3.20	2.56	208	2.22	0.25	–
Rent	ratio=2.0	3.14	3.08	201	2.29	0.32	–
Rent	ratio=3.5	3.20	2.85	225	2.31	0.34	–
Rent	ratio=2.5	2.92	2.39	212	2.33	0.36	–
Size	size=4	2.43	2.10	269	2.76	0.79	–
Size	size=6	2.32	2.75	337	3.14	1.17	–
Size	size=20	2.16	3.86	401	3.60	1.63	–
Size	size=30	2.17	4.14	394	3.61	1.64	–
Size	size=15	1.91	3.64	410	3.85	1.88	–

Table 2: Badness Value Calculations for 12 Demultiplexer Schematics

of modules, nets and terminals for each example are listed to give an indication of their relative complexity. We note that the SN5491A shift register has 52 terminals, but due to its very regular composure, runs in less time than figures with fewer terminals. The times listed in table 3 are wall clock seconds, run on an unloaded Sun Sparc Station II.

Name	Modules	Nets	Terminals	Runtime	Figure	Badness
Ripple-carry adder w	34	34	82	7.94 sec	6	2.08
Ripple-carry adder w/o	34	34	82	4.15 sec	–	2.11
SN54S151 w ripup	32	33	89	7.02 sec	–	2.92
SN54S151 w/o ripup	32	33	89	3.65 sec	–	1.90
SN54155 w	37	32	78	4.57 sec	–	2.02
SN54155 w/o	37	32	78	2.70 sec	12	1.97
Parallel add/sub w	19	35	65	3.42 sec	9	5.59
Parallel add/sub w/o	19	35	65	1.89 sec	–	5.61
SN5491A w	16	25	52	1.42 sec	–	2.86
Complex String w	19	20	37	1.41 sec	4	8.82
Edge-trig. D-latch w	14	15	32	1.43 sec	1	6.96
SN7474 w	12	13	30	1.34 sec	8b	11.69
SN7474 w/o	12	13	30	0.95 sec	8a	11.67

Table 3: Complexity and runtimes for comon examples presented

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new set of algorithms that generate schematic diagrams from circuit netlists. The algorithms have been designed to generate schematics that are useful to the circuit designer in terms of both *functional identification*, the grouping of related modules, and *traceability*, the ease with which routes can be followed.

The approach is novel in the use of a two dimensional tile space for module placement, congestion evaluation during routing, and the incremental placement of I/O modules. Another new technique presented is the use of constraint propagation in the local router to link connected corners throughout the diagram. The success of SPAR can be summarized as follows:

- Creates schematics from flattened netlists
- Produces schematics that organize the modules by function
- Creates diagrams that can be traced
- Uses page space in an efficient and flexible manner

The most interesting results of this work have been in two areas: the successful application of two-dimensional space-management techniques employed throughout SPAR, and the success of the constraint-propagation techniques used in the local router. Most notable is the application of the space management techniques to allow for incremental placement and routing. There are other area-phenomena to which the space-management techniques could be applied, as well as further developments of geometric constraint-propagation to be explored. To continue this work for the development of a more robust ASG system, we see three necessary extensions:

- Extend the incremental placement and routing scheme to apply to the merger of strings and virtual pages. This would permit routing information to help determine the placement of modules in a growing schematic.
- Extend the delayed evaluation of corner positions so that module placement is dependent on final corner positions. This would provide a much more direct link between the placement and the routing of modules, as the module positions would provide an initial placement along with limits within which adjustments could be made.
- Extend the router to handle bus structures, 45° angle corners, and improve crossover elimination by reordering of pins within modules.

The results presented in this paper demonstrate that a heuristic approach to ASG is both feasible and effective, especially for smaller schematics. Our experience in developing SPAR shows that schematic generation is not a trivial problem, and requires special-purpose algorithms and data-structures to yield good results.

6 References

[1] Jehng, Y.S., Chen, L.G., and Parng, T.M.. "ASG: Automatic Schematic Generator". *Integrated VLSI Journal*, Vol. 11 No. 1, pp. 11–27, March 1991.

[2] Ebeling, C. and Wu, Z.. "WireLisp: Combining Graphics and Procedures in a Circuit Specification Language". *IEEE Int. Conf. on Comp. Aided Design*, pp. 322–325, Nov. 1989.

[3] Swinkels, G.M. and Hafer, L.. "Schematic Generation with an Expert System". *IEEE Trans. on Comp. Aided Design*, Vol. 9 No. 12, pp. 1289–1306, Dec. 1990.

[4] Majewski, M., Krull, F., Fuhrman, T., and Ainslie, P. "AUTODRAFT: Automatic Synthesis of Circuit Schematics". *IEEE Int. Conf. on Comp. Aided Design*, pp. 435–438, Nov. 1986.

[5] Nilsson, N.J. *Principles of Artificial Intelligence*. Tioga Publishing, Palo Alto, CA, 1980.

[6] Chun, R.K., Chang, K., and McNamee, L.P. "VISION: VHDL Induced Schematic Imaging On Net-lists". *24th Design Automation Conf.*, pp. 436–442, 1987.

[7] Stok, L. and Koster, G.J.P. "From Network to Artwork". *26th Design Automation Conf.*, pp. 686–689, 1989.

[8] Arya, A., Kumar, A., Swaminathan, V., and Misra, A. "Automatic Generation of Digital System Schematic Diagrams". *22nd Design Automation Conf.*, pp. 388–395, 1985.

[9] Smith, J.A. *Automated Generation of Logic Diagrams*. PhD thesis, Dept. of Comp. Science, Univ. of Waterloo, Waterloo, Ontario, 1975.

[10] May, M., Iwainsky, A., and Mennecke, P. "Placement and Routing for Logic Schematics". *Comp. Aided Design*, Vol. 15 No. 3, pp. 89–101, May 1983.

[11] May, M. "Computer-Generated Multi-Row Schematics". *Comp. Aided Design*, Vol. 17 No. 1, pp. 25–29, Jan. 1985.

[12] Lee, T.D. and McNamee, L.P. "Structure Optimization in Logic Schematic Generation". *IEEE Int. Conf. on Comp. Aided Design*, pp. 330–333, 1989.

[13] Brady, H.N. Jr. *Automatic Generation of Schematic Diagrams*. PhD thesis, Dept. of Elec. Engr., Univ. of Texas, Austin, TX, 1982.

[14] Brennan, R.J. "An Algorithm for Automatic Line Routing on Schematic Diagrams". *12th Design Automation Conf.*, pp. 324–330, 1975.

[15] Kumar, A., Arya, A., Swaminathan, V., and Misra, A. "Automatic Generation of Digital System Schematic Diagrams". *IEEE Design and Test*, pp. 58–65, Feb. 1986.

[16] Ahlstrom, M.L., Hadden, G.D., and Stroick, G.R. "HAL: A Heuristic Approach to Schematic Generation". *IEEE Int. Conf. on Comp. Aided Design*, pp. 83–86, 1983.

[17] Venkataraman, V.V. and Wilcox, C.D. "GEMS: An Automatic Layout Tool for MIMOLA Schematics". *23rd Design Automation Conf.*, pp. 131–137, 1986.

[18] Baruah, S.K., Jerath, M.R., Sundaresan, S., Banerjee, S., Kumar, S., Kumar, A., and Bhatt, P.C.P. "A Blackboard Architecture to Support Generation of Schematics for Design Automation". *Proc. IFIP TC 10 / WG 10.2 Working Conf. on CAD Systems Using AI Techniques*, pp. 51–58, June 1989. North Holland Press, G. Odawara Ed.

[19] Koster, G.J.P. and Stok, L. "From Network to Artwork: Automatic Schematic Diagram Generation". Technical Report EUT 89-E-219, Eindhoven Univ. of Tech., Eindhoven, Netherlands, April 1989.

[20] Ousterhout, J.K. "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools". *IEEE Trans. on Comp. Aided Design*, Vol. CAD-3 No. 1, pp. 87–100, 1984.

[21] Romesburg, H.C. *Cluster Analysis for Researchers*. Lifetime Learning Publications, 1984.

[22] Landman, B.S. and Russo, R.L. "On a Pin Versus Block Relationship for Partitions of Logic Graphs". *IEEE Trans. on Comp.*, C-20, pp. 1469–1479, 1971.

[23] Frezza, S.T. "SPAR: A Schematic Place And Route System". Technical Report TR-CE-91-02, Dept. of Elec. Engr., Univ. of Pittsburgh, Pittsburgh, PA, April 1991.

Footnotes

1. Manuscript received _____ . Revised _____ .

2. Affiliation of Authors:
S. T. Frezza is with the Department of Electrical Engineering, University of Pittsburgh, Pittsburgh, PA 15261.

S. P. Levitan is with the Department of Electrical Engineering, University of Pittsburgh, Pittsburgh, PA 15261.

3. Acknowledgment of Financial Support:
This work was supported, in part, by a Software Capitalization Grant from the National Science Foundation, MIP-9101656.

Technical Biographies

Stephen T. Frezza is a Ph. D. student in the Department of Electrical Engineering at the University of Pittsburgh. Frezza graduated *cum laude* in 1985 from the University of Pittsburgh with a B.S. in Electrical Engineering. He worked for two years with Lutron Electronics and Westinghouse Electric as a Project Engineer. He received his M. S. in Computer Engineering from the University of Pittsburgh in 1991. His research interests include Design Environments, Design and Requirement Representations, Schematic Generation and Intelligent Systems. He is a student member of IEEE-CS DATC and ACM SIGDA.

Steven P. Levitan is the Wellington C. Carl Associate Professor of Electrical Engineering at the University of Pittsburgh. He received the B. S. degree from Case Western Reserve University (1972) and his M. S. (1979) and Ph. D. (1984), both in Computer Science, from the University of Massachusetts, Amherst. He worked for Xylogic Systems designing hardware for computerized text processing systems and for Digital Equipment Corporation on the Silicon Synthesis project. He was an Assistant Professor from 1984 to 1986 in the Electrical and Computer Engineering Department at the University of Massachusetts. In 1987 he joined the Electrical Engineering faculty at the University of Pittsburgh. Dr. Levitan's research interests include computer aided design for VLSI, parallel computer architecture, parallel algorithm design, and VLSI design. He is a member of the IEEE-CS, ACM, SPIE, and OSA.