SPAR: A SCHEMATIC PLACE AND ROUTE SYSTEM

by

Stephen T. Frezza

B.S. in E.E., University of Pittsburgh, 1985

Submitted to the Graduate Faculty
of the Graduate School of Engineering
in partial fulfillment of
the requirement for the degree of
Master of Science
in
Electrical Engineering

University of Pittsburgh
1991

COMMITTEE SIGNATURE PAGE

Steven P. Levitan, Ph.D.	
Advisor	Signature

ACKNOWLEDGMENTS

I would especially like to thank Steve Levitan for his guidance and patience throughout the development of this system.

Also, I would like to thank my future wife Marge, for her continuous prayers, patience and support.

ABSTRACT

Signature_____

SPAR: A SCHEMATIC PLACE AND ROUTE SYSTEM

Stephen T. Frezza, M.S.

University of Pittsburgh

This thesis presents an approach to the automatic generation of schematic diagrams from circuit netlists. The algorithms which make up the system are based on two overriding principles of schematics readability: *Functional Identification* and *Traceability*. The generation process is broken into five phases: partitioning the netlist, placement of components on the page, global routing, local routing, and addition of system terminals. All phases of the generation process use a novel 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 novel 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.

DESCRIPTORS

Placement
Routing
Schematic Diagrams

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION

SPAR (Schematics Placement and Routing) is a combined algorithmic and heuristic approach to the generation of schematic diagrams. The system automatically creates schematic logic diagrams from textual descriptions of digital systems. SPAR derives the connectivity among the modules in the netlist, and uses this information to form non-fixed-size partitions of functionally related modules. The modules within these partitions are placed separately in gridless tile spaces, then merged using special space management techniques. Global routing of the schematic is accomplished using a breadth-first 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 decisions made. Local routing information is explicitly used to place system terminals on the page.

1.1 Contributions

This thesis presents a unique theoretical approach, and several new techniques for the automatic generation of schematic diagrams. The first contribution presented is that we focus schematic generation on broader goals than has been done in the past, specifically *Functional Identification* and *Traceability*. Other contributions include our approach to space management of the diagram, our use of congestion information to determine the global route and our techniques for optimizing the local route for traceability.

The theoretical underpinnings of our system are somewhat broader in scope than earlier work. What is important about our use of *Functional Identification* and *Traceability* is how we use them to guide our algorithms. These concepts are introduced in Section 1.3 and are placed in the context of earlier work in Section 2.2.

The space management technique we employ manages the diagram space directly, mapping placement and routing structures to the diagram so they can be queried and manipulated. The value of these measures is introduced in Section 2.3, and outlined

in Section 3.2. The specific algorithms employed are included in all five sections in Chapter 4.

Our routing is divided into two stages, global and local. Our global routing algorithm contributes a novel approach to routing, as it employs congestion metrics that discover the route that will be least congested, that is, easiest to trace. These notions are described in Sections 2.3, and expanded in Sections 3.5 and 4.3.

The local router employs a new constraint propagation technique, which provides a solution to the crossover problem. This problem is described in Section 2.2. Our solution is described generally in Section 3.6 and in detail in Section 4.3.

1.2 The Schematics Generation Problem

Schematic diagrams were once the principle design medium for digital circuits, providing a clear, graphical means of specifying circuits at different levels of complexity. They are powerful enough to encapsulate everything from board-level to transistor-level designs. A schematic consists of a set of *modules* that are linked to *nets* by their *terminals*. There is a wide range of graphical (iconic) representations for the modules, and lines are used to represent the connections among the terminals of each net. Schematic capture tools are common on most Computer Aided Design (CAD) systems, primarily as an input medium to the system. 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.

Unfortunately, textual description languages have a tendency to be somewhat obscure [1]* lacking the depth of insight provided by a graphical image. Hence there is a need for a schematic generated from a textual hardware description. Even for designs where schematic capture is the primary input medium, automated optimization processes can be used to modify the underlying structure of a design, invalidating the input schematics. Designers need to graphically document the results of these processes, to reverse engineer the design. This graphical documentation serves to verify the transformations that were applied, observe the results of the optimization

*Parenthetical references placed superior to the line of text refer to the bibliography.

processes [2], and is used to track a design through an iterative design process. The schematic generator must be able to extract features from the input description that are implicit, producing an orderly, easy-to-read (traceable) diagram. Unfortunately, the rules governing the production of schematics with these qualities are difficult to quantify, and often require special-case and exception handling facilities [3].

1.2.1 Problem Statement

The goal of this thesis is to produce an algorithmic system that generates useful schematic diagrams from circuit netlist descriptions. In particular, we solve the general schematics problem, where standard logic gates and block figures may be mixed [4]. Our generation system is expected to:

- Operate without relying on hierarchical information in the source netlist
- Produce schematics that organize the modules on the diagram by function
- Create diagrams that can be traced
- Demonstrate an efficient and flexible use of diagram space

In order to meet this goal, we first must identify the desireable characteristics of schematic diagrams. Second, we must identify useful measures of these characteristics, and third we must incorporate these measures into the generation algorithms.

1.3 Characteristics Desirable in Generated Schematics

For a diagram to be useful as feedback to the designer or as documentation [5], the diagram must be able to communicate "the flow of information, time, or material without reference to a physical layout" [3]. The goal is for the schematic to communicate information about the operation of the design. The positioning of the design modules only has value where it helps to impart this information. To the human designer, the schematic is meant to be understood - both for the recognition of the design and the intuition of the design engineer. Therefore, the utility of any ASG system must be judged by how well its output is understood by the designer. We use the concepts of *traceability* and *functional identification* to describe how well the schematic is understood.

Traceability is the ease with which one can follow signals from their logical sources to their destinations, and centers on having paths that are not complex and do not run through densely-populated areas. *Functional identification* centers around the location of modules that are functionally interconnected, and having distinct groups of modules appropriately separated. Traceability and functional identification are two essential characteristics of an understandable schematic, hence the goal of an ASG system is to produce schematics with that have these two required characteristics.

Ideally, one must be able to identify the functionality represented by the schematic [6, 7], and the traceability of the nets within the schematic. [8]. Functional Identification stems from the readers ability to recognize common structures. Figure 1 shows two common examples: a single gate enabling several others, and the identification of two cross-coupled gates as a flip-flop.

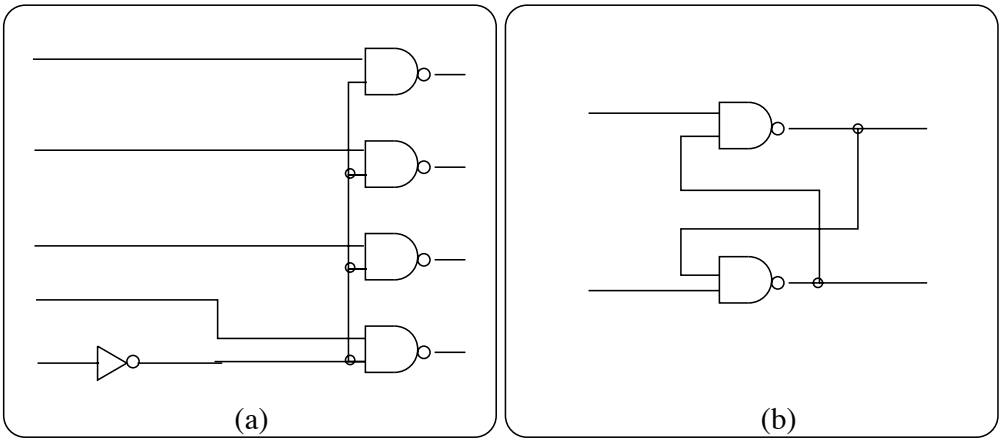


Figure 1 Common examples of *Functional Identification* in schematics

Functional identification and traceability are interdependent, since a given placement can not simply be optimized toward the patterned recognition of structure, or toward the traceability of the given structure. For example, using a simplistic notion of traceability, one could place gates in order from source to destination, following the inheritance of the main signal path. Figure 2a illustrates the results of this view, by applying a simple left-right signal inheritance scheme where the main signal path is easily traced. The circuit illustrated has several cross-coupled modules which are emphasized in Figure 2b. Figure 2b employs functional groupings, but is not as strict with the left-to-right signal inheritance and yields a schematic that is much clearer

than that in Figure 2a. This example illustrates that the two goals of functional identification and traceability are tightly linked and cannot be separated with simple optimization criteria.

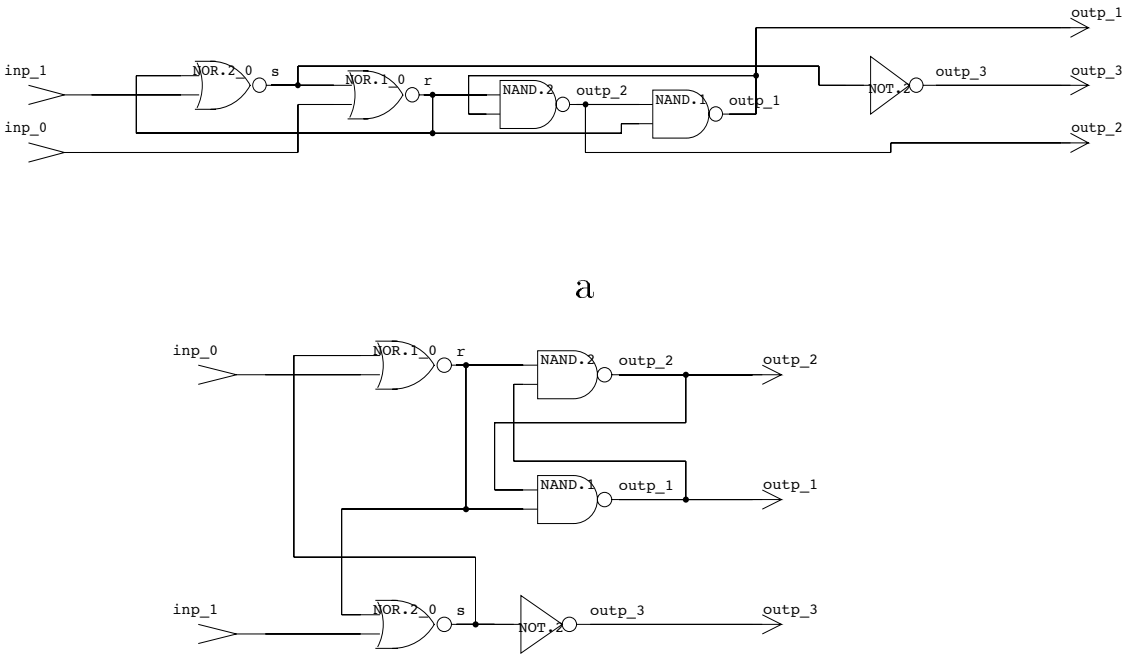


Figure 2 Traceability and functional identification tradeoffs.

Both the identification of structures and the traceability of schematics have "rigid standards" of aesthetic expectations [8, 9]. In general, it is easy for one to say what is *bad* about a schematic, such as that in Figure 2a. Here the placement is such that a single path is easily traced. This obscures the functional blocks within the diagram and the traceability of the other paths. On the other hand, it is more difficult to say what is *good* about it.

To summarize, an ASG must transform netlists into a physical arrangement of icons in an identifiable way, and route them clearly. Furthermore, this arrangement must be traceable, indicating that the route selected for each signal must interact

with the module placement in such a way as to make each net as traceable as possible. Since groups of modules related by function often have several nets interacting in a restricted area, these two goals of tight functional groupings and traceable signals are often conflicting, and form the core of what makes the generation of good schematic diagrams both interesting and difficult.

1.4 Design Requirements

The function of an ASG system is to use the information embedded in a netlist to generate a schematic diagram. Because the design environment that we operate in [10] employs optimization tools, we expect the ASG system not to rely on any hierarchical information within the netlist, but to use *flattened* netlists (those without hierarchical information) to generate the schematic. We expect the system to identify the functionality of the design, and thus discover the structure embedded in the design. We also expect the system to arrange the icons representing the modules and the routes in such a way as to make the diagram traceable.

As mentioned, ASG systems are intended to work with designs created using description languages. A small example of the hardware description language VHDL [11] is provided in Figure 3. This is a hierarchical description of a D-Latch, using structural components. The D-Latch design contains two RS flip-flops, whose VHDL description is presented in Figure 4. As these flip-flops have been designed into the latch, it is expected that they will also be recognized in the schematic for the latch.

This description is compiled into a netlist, which is used to create the schematic. What the netlist contains is the *namespace* of the design, which is used to label the diagram,and it contains the connectivity of the design. This connectivity information is embedded within the module declarations, as each declaration lists the signals that connect to the module. This is highlighted in the inset box in Figure 5.

A general ASG system is expected to work with flattened netlist descriptions, like the one shown in Figure 5. This netlist is *flat* in that it contains no hierarchical information. The modules of the two flip-flops are included with the other modules of the latch, only their names are different, thus the hierarchy originally designed into the VHDL description is not available for the ASG to use in developing the

diagram. In order to meet the expectation of representing the flip-flops, the structure they represent must be discovered through their connectivity, as this is the only information available to the ASG. This discovery of structure within the design is the key to functional identification.

Specification for the D-Latch:

```
entity d_ff is
        port (d, c: in bit; q1, qbar1: out bit);
end d_ff;
architecture d_struct of d_ff is
        signal dbar, cbar: bit;
        signal rbar0, sbar0, q0, qbar0: bit;
        signal rbar1, sbar1: bit;
        label rs_0, rs_1;
        component rs_ff
                port (rbar, sbar: in bit; q, qbar: out bit);
        end component;
        for rs_0, rs_1: rs_ff
                use entity rs_ff(rs_struct);
begin
        dbar <= NOT d;
        rbar0 <= dbar NAND c;
        sbar0 <= d NAND c;
        cbar <= not c;
        rs_0: rs_ff port map (rbar0, sbar0, q0, qbar0);
        rbar1 <= qbar0 NAND cbar;
        sbar1 <= q0 NAND cbar;
        rs_1: rs_ff port map (rbar1, sbar1, q1, qbar1);
end d_struct;
```

Figure 3 Hierarchical VHDL design of a simple D-latch

Specification for an RS Flip-Flop:

```
entity rs_ff is
        PORT (rbar, sbar: in bit; q, qbar: out bit);
end rs_ff;

architecture rs_struct of rs_ff is
begin
        q <= sbar NAND qbar;
        qbar <= rbar NAND q;
end rs_struct;
```

Figure 4 Source VHDL code for the RS flip-flop used to specify the D-latch

Figure 6 shows the diagram generated by SPAR for the D-Latch. The two RS flip-flops are easily recognized, as they have been placed in an appropriate way. The partitioning and placement algorithms combine to center the two RS flip-flops that are part of the diagram, and can be distinguished by the cross-coupling of their signals. These modules that constitute the flip-flops are labeled *rs_0/NAND.1, rs_0/NAND.2* and *rs_1/NAND.1, rs_1/NAND.2*, as specified in Figure 5. The signal flow in the design is also appropriate, flowing from left to right from signal inputs to outputs in a the straightforward manner.

Although it is a small example, the D-latch of Figure 6 exemplifies the task that an ASG system must be good at performing. The schematic generated typifies the identification of function, and the traceability of the signals used.

In this thesis I first present the motivation for automatic schematic generation (ASG) in the problem description of Section 1.2. This chapter describes the view of schematic generation that underlies SPAR, and develops the characteristics that are desirable in a generated schematic. Using these desirable characteristics as a point of focus, Chapter 2 describes previous ASG work, and how SPAR differs from these systems. In the context of this earlier work, I describe many of the common issues in ASG that are addressed in SPAR. Chapter 3 presents a brief overview of

the system operation, and concludes with an outline of the major tasks involved. Chapter 4 follows with a more in-depth look at the mechanics of each of the major tasks, Partitioning, Placement, Global Routing, Local Routing, and the Insertion of System Terminals. Further examples are presented, accompanied by an analysis of the examples presented. My concluding remarks follow, accompanied by a brief discussion of future work.

Netlist information for the D-Latch:

```
MX d_ff.d_struct
{
  MP {d IN;c IN;q1 OUT;qbar1 OUT;}
  FS {cbar dbar q0 qbar0 rbar0 rbar1 sbar0 sbar1}
  FG NOT.1 {NOT 1 d 1 dbar }
  FG NAND.3 {NAND 2 dbar c 1 rbar0 }
  FG NAND.4 {NAND 2 d c 1 sbar0 }
  FG NOT.2 {NOT 1 c 1 cbar }
  FG NAND.5 {NAND 2 qbar0 cbar 1 rbar1 }
  FG NAND.6 {NAND 2 q0 cbar 1 sbar1 }
  FG rs_1/NAND.1 {NAND 2 sbar1 qbar1 1 q1 }
  FG rs_1/NAND.2 {NAND 2 rbar1 q1 1 qbar1 }
  FG rs_0/NAND.1 {NAND 2 sbar0 qbar0 1 q0 }
  FG rs_0/NAND.2 {NAND 2 rbar0 q0 1 qbar0 }
}
```

Example module definition:

FG NAND.5 {NAND 2 qbar0 cbar 1 rbar1 }

is a NAND module having two inputs, nets *qbar0* and *cbar* and one output, *rbar1*.

Figure 5 Netlist used to create the D-latch

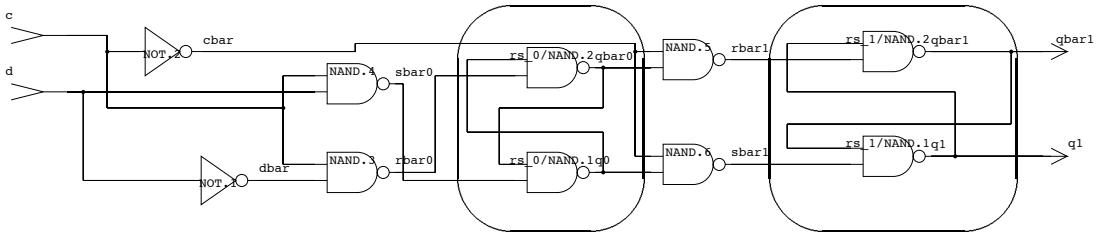


Figure 6 Completed D Latch

2.0 PREVIOUS WORK

This chapter presents a review of previous ASG work, a discussion of the common problems and issues developed in the various ASG systems presented, and the motivation and evolution of our approach.

2.1 Background

There have been numerous ASG systems, aimed at generating schematics from netlists. Some deal with the general schematics problem, while others focus only on specific forms of the problem, such as logic diagrams [5, 6, 12], or flow graphs [13, 14]. The goal has been to reproduce what draftsmen do, and at some level, model the rules that these experts use. There are two basic types of models: algorithmic and expert system. We first detail various algorithmic models, and then go on to describe various expert system approaches.

Most previous ASG approaches are algorithmic in nature, using a combination of specialized heuristics and algorithms. Some of the more illustrative papers include R. J. Brennan, who presents [9] a good heuristic approach to routing. May [15, 16] describes an application of graph-minimization techniques with some impressive placement results. *VISION* [12] and its later modification [6] while focusing on the gate placement problem, have produced some excellent results by combining connectivity information with clustering and simulated annealing techniques. Other researchers [7, 14, 17] use partitioning to simplify the problem. The work described in [5] was one of the first to try to combine placement and routing, by using iterative improvement in an algorithmic approach.

Like our system, the algorithmic approaches mentioned take the view that placement and routing can be dealt with as separate issues, and that the subtle interactions between placement and routing can be effectively managed without directly modeling

the a draftsman's rules, such as in an expert system. The interaction between placement and routing is an important issue, one that has been used to defend the use of expert rules [3], and the associated cost of knowledge acquisition and performance.

Placement and routing interactions break down into two categories, obvious and subtle. Obvious interaction effects are based on the overall placement of the modules, that is, having two cross-coupled gates above each other yields the desired effect of having the connecting routes cross between the two (Figure 1). Subtle interactions are those that affect how the route can be run, and generally affect the quality of the route. Figure 7 contains an example showing the complexities of subtle interactions. This example shows how the placement of modules must anticipate how the route will occur, and illustrates the fact that the separation of the placement and routing tasks must be done carefully.

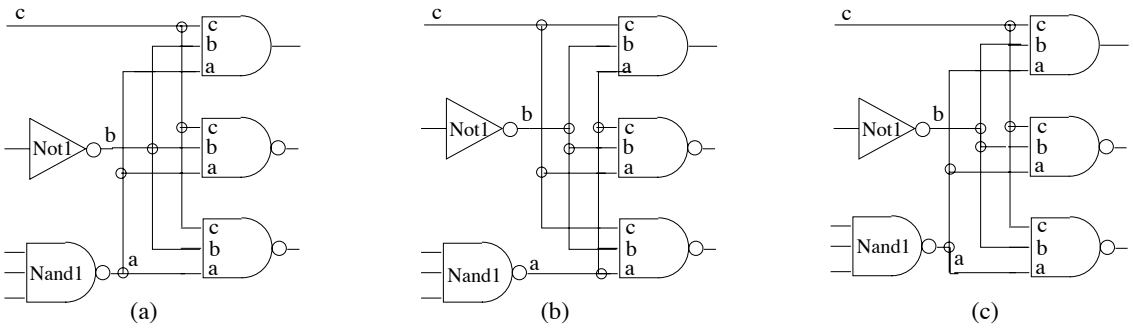


Figure 7 Placement effects routing quality and selection

The subtleties of the interaction between placement and routing can be seen in Figure 7, which shows three placements for the same diagram. Here progressive, minor changes in the locations of modules *Nand1* and *Not1* have been introduced, which force the routes for nets a, b, and c to take particular arrangements. In Figure 7a, nets a, b, and c can be assigned to any of the three columns between the two sets of modules. In Figure 7b, net c must be placed to the right of net b. Finally, Figure 7c shows a situation where the entire route is constrained, that is net a must be to the left of net b, which in turn must be placed to the left of net c. The changes in the placement of modules *Nand1* and *Not1* shown are very small, yet they limit the ability to form the best route. These subtle interactions point out that placement and routing are closely interrelated, and any means of separating the two

must anticipate this interaction.

Another school of thought argues that algorithmic approach cannot deal easily with the multiple objectives and pattern recognition issues [3] that a draftsman can keep in mind. They claim that the artistic nature of the schematic generation task is difficult to capture using conventional programming techniques [13]. These researchers have applied Artificial Intelligence(AI) techniques [3, 8, 13] to the ASG problem, often with a large investment in knowledge acquisition. The expert system work described in [3] is most notable for addressing the issues that make AI appealing, those being the complex rules that humans use to distinguish less-legible routes from more-legible ones. In describing their Knowledge-Based System (KBS) approach, they also provide a good treatment of the interactions that placement and routing have on each other. Typically, these systems separate their rules into general placement and routing categories, and further, the routing rules in a KBS system can also be used to alter the placement. This is not common in an algorithmic approach.

The importance of separating placement from routing to the generation problem is crucial. In point-of-fact, the assumption that the problem can be divided into separate placement and routing phases underlies many of the KBS approaches and all of the algorithmic ones.

Dividing the placement problem into interesting subproblems enables the set of placement heuristics to do a good job. By focusing placement algorithms onto a limited number of tightly-connected modules, the heuristics employed can focus on the proper connection of modules on a local level, yielding good placement results for the subproblem. For instance, *HALS* [13] uses expert rules to form clusters of modules into partitions, with the object of minimizing the complexity of the resulting task. *AUTODRAFT* [8] partitions depend on the hierarchy embedded in the design description. Both *Pablo* [7, 17] and *GEMS* [14] divide the schematic into fixed-sized blocks before any modules are placed. The common element to these partitioning ideas is that they reduce the complexity of the resulting subproblem, with a consequent increase in speed, and they allow the heuristics employed to be arranged to operate on problems of limited size and scope.

2.2 Common Issues

Not only have various algorithmic and KBS techniques been employed, but also various criteria have been put forward as measures for the quality of schematics. These evolved from "ad hoc" rules of both how draftsmen create schematics and how designers use them. Like the algorithms used, these rules can broken up into criteria for good placement vs. criteria for good routing, as summarized in Figure 8.

Diagrams must...

- Be made visually simple, but functionally meaningful. (6)
- Be grouped in functional groups. (3, 7, 12)
- Minimize clutter. (14)
- Utilize and minimize the drawing area. (3)
- Be arranged such that most signal flow is uni-directional. (5, 7)
- Maintain and demonstrate the direction of flow. (3)
- Have short connections between logic units. (5, 7, 15)

Paths should...

- Use small numbers of line branching nodes and intersections. (3, 5, 15, 16)
- Exhibit low net-crossing complexity. (6, 15, 16)
- Not be too long, or have too many bends. (5)
- Minimize the number of sharp turns. (3, 15)
- Have a minimum number of straight line segments. (9)
- Minimize the number of segments, crossovers, and total path length. (7, 9, 15)

Figure 8 Standards of schematic quality

This gross summary of schematic aesthetics in Figure 8 is important in both what it says, and what it does not say: The engineering community has "rigid standards" (9) for schematic generation, but there is no clear consensus as to how to implement these standards. We take the view that although all of these criteria are useful, they only incidentally address the needs of the users of the generated schematics mentioned earlier.

The clear goals of the *user* of the schematic are traceability and identification. The guidelines summarized in Figure 8 only approximate these two concepts. Where many of the previously discussed systems fail is that they address these issues at a level that is too low to provide overall guidance, or too rigid to arrange distinguishable groups of modules. We believe that one of the reasons for these limitations is the highly constrained ways in which these previous systems manage the physical layout of the page. We address these issues in our approach to space management, which is one of the more novel aspects of our work.

The weakness of other systems in their use of space is apparent in the reliance upon a fixed coordinate (grid) system, a feature common to many ASG systems. The convention commonly used is to arrange a grid such that each icon or *via* occupies one grid location. 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]. This simplifies the computational tractability of the problem, especially when using techniques such as simulated annealing [6] or rule-based systems. Unfortunately, the reliance upon a grid system imposes restrictions on the use of space that human drafters do not have. Grids also make it more difficult to cleanly model the idea that *locality* implies *relation*. The functional relationship of a given set of modules is transmitted in two ways: The nearness among related modules, and their distance from unrelated ones. Although a grid-like view of the world may take this into account, the size and orientation of such spacing is constrained by the scale of the grid. Varying the amount of space between modules or groups of modules is difficult to handle.

A fixed coordinate system also has difficulty in modeling how odd-sized shapes are to be placed on the page, as in Figure 9(a). In the general schematic problem *icons are not of fixed sizes*. This is seen in at least two important cases: The first is when representing non-standard gates, as occurs when including hierarchy within a design. The second case occurs within the limitation of using standard logic gates when the space required for the terminals is larger that the icon, *e.g.*, when gates have a large fan-in as in Figure 9(b). These difficulties can be alleviated by using a finer coordinate system, and by using space-holders to indicate that a particular module occupies more than one location. Unfortunately, as the scale decreases, so

does the computational advantages of the grid system, and the ease in which special cases can be handled, such as the inclusion of space holders.

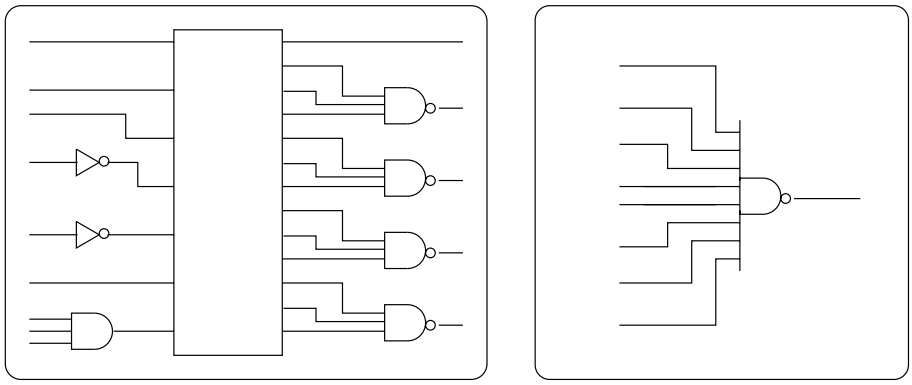


Figure 9 Modules that do not lend themselves to grid arrangements

The use of a fixed coordinate system also simplifies the arrangement of modules into rows and columns, a feature common to hand-drawn schematics. It is desirable to indicate signal inheritance by arranging items in columns and rows, but this does not imply that the entire diagram needs to be restricted to column and rows, or that the same columns and rows should apply of the entire page. Rather, it is often desirable to use space between groups of modules to indicate their grouping.

Other work has been done without the limitations of grids, most notably *Pablo*, *HALS*, and *GEMS*. It is interesting to note that all three of these systems are designed for the more general schematic problem where irregular gate sizes/shapes are expected. The *HALS* and *GEMS* systems are restricted to Register-Transfer (RT) level schematics, and are geared toward a specific set of RT devices utilized by the MIMOLA Software System [13, 14]. All three of the systems use restricted fixed sized partitions, and are thus inflexible in their ability to discover the functional groupings of modules that are desirable.

Pablo, *HALS*, and *GEMS* all develop *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, and are surrounded by a bounding box which defines the space that the modules are placed in. All three systems are only able to abut strings based on their bounding boxes [17]. For simple strings, this is quite useful, but

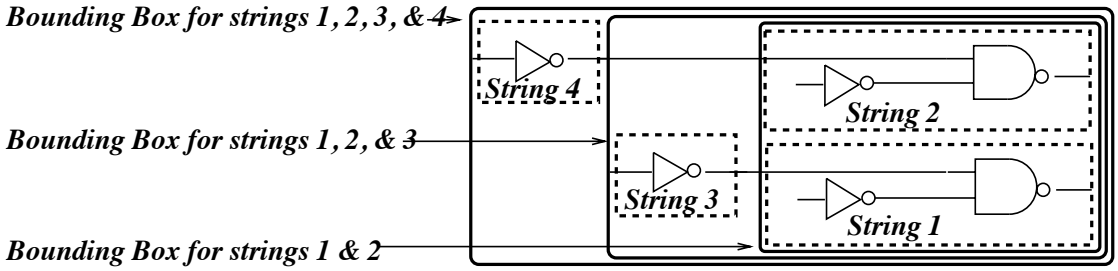


Figure 10 Early attempts at space management

leads to the space problem shown in Figure 10. As strings can only be abutted and not overlapped, empty space appears in the diagram that otherwise cannot be used. This severely limits the problems to which this technique can be well-applied, as the partitioning/string placement schemes must create groupings who's interactions will not produce the empty-space problem. By using our space management techniques, we are able to solve these problems and gear SPAR to the general schematics problem. These techniques are described in more detail in Sections 3.2 and 4.5.

Another area where space usage must be dealt with effectively is in the traceability of a schematic. All of the qualities which have been used to define good schematics (Figure 8) are true, but they mask the dependence of traceability on the nets' proximity to other nets and modules. Proximity of nets, or rather the proximity of the features of a net hinder the traceability of that net. By features, we refer specifically to corners and T-structures in the lines that make up the route. Wherever many of these features occur in a small area, the lines that pass through this area are more difficult to trace. Here we observe that this is an area phenomenon, and that as the number of corners in a specific area (the corner density) goes up, the traceability of the lines through that area goes down. Figure 11 shows two regions with the same number of nets crossing them. In the two figures, the crossover and density counts are reversed. We maintain that the region with the high density count is more difficult to trace than the region with the high crossover count. This indicates that the goal of traceability is better measured via the density of corners rather than the number of crossovers contained.

The rules listed for crossover, bend and connection counts are important measures

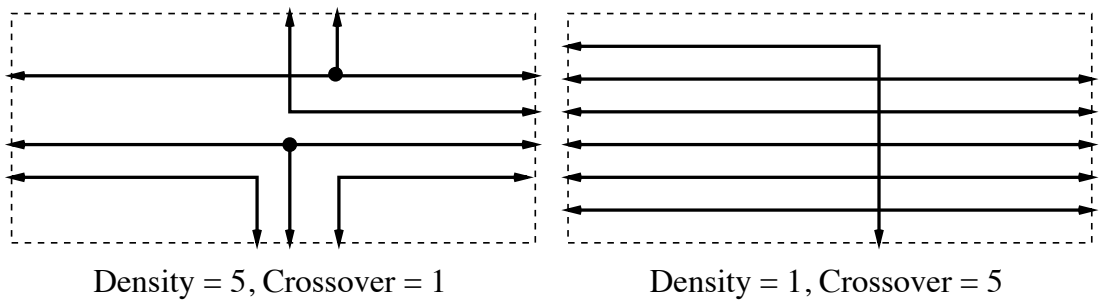


Figure 11 Density vs. crossover count as a measure of traceability

of route complexity, but unfortunately 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. While just a few such bends in a small area can make the given net much more difficult to follow, as in Figure 11. We refer to areas that have a high corner density as being *congested*, and hence the *congestion* of a given area is an inverse measure of how traceable the lines crossing that area are.

All of the previous ASG systems apply rules for schematic routing which simply minimize the number of crossovers, or apply specific ordering rules that model what human designers do. We claim that although good, this is insufficient in that the traceability of a net is not always dependent on crossovers, but rather on the congestion of the areas through which its lines pass. The minimum crossing solution is not necessarily the most traceable solution.

2.3 Our Evolution

As a starting point for our work, we have chosen to use the encompassing goals of *identification* and *traceability* to develop a consistent methodology to drive both our algorithm development and the evaluation of our results, rather than the more common list of ad-hoc rules. We focus on these issues by combining successful features from previous work, along with a unique approach to space management.

Our solution to the space management problem described is the use of *virtual space* as a map, which can be queried for information. This gives several advantages over more traditional methods. To create a useful placement, the netlist is partitioned and

placed on virtual pages. The pages are then merged together to form the complete placement. Having an active representation of these virtual pages is a great advantage in merging them together. By directly manipulating the virtual space, irregular shapes can be fitted, much like jigsaw-puzzle pieces. This aids in placement, as it permits diagram fragments to be merged cleanly, and the space between them to be managed appropriately. It also aids in routing, as it enables congestion to be tracked.

We implement these space management ideas with corner stitching techniques [18]. These techniques provide a clean, tractable method for operating with an irregular, dynamic, two–dimensional surface which makes it ideal for "growing" a schematic diagram. For placement, a virtual page consists of a set of two–dimensional rectangular tiles linked at their corners, and is referred to as a *tile space*. Using such tile spaces, we are able to build individual functional groups (partitions) on virtual pages, merge, route, and later add to them. The utility and operation of this technique is explained further in Section 3.2. The important features of tile space use is in the merging of partitions described in Section 4.2 and the evaluation of proposed nets for congestion described in Section 4.3. This space management technique underlies all of the placement and routing activities within the system.

Aside from the application of corner–stitched tile–spaces to the space management problems of ASG, we also have developed incremental placement and routing techniques, whereby a set of modules is placed, routed, and then the route is used to guide the placement of further modules. We have applied this incremental placement to the system terminals of the diagrams, which is described in Section 4.5.

Another area where our space management approach has a big advantage is in the control of the routing process: By mapping proposed routes directly to the space in which they are run, we enable the calculation of the congestion of the route. We do this by locating the areas in which routes should be run by using an iterative search technique on proposed routes within two 90°-rotated tile spaces. Given a map of the areas that the route uses, we then apply congestion–sensitive metrics. This is described at length in Section 4.3. We also use these tile spaces to locate the corners of the route to which we apply the ordering relation that is at the center of our local router. These local routing details are explained in Section 4.4.

The global router implemented in SPAR is geared toward dealing with congestion as an important factor in traceability. It is not necessarily the number of runs and corners, but rather the *location* of the runs and corners that reduces the nets traceability. We implement congestion as a simple count of the number of corners and joints in a given area. Other routers aim at making the route simple, by minimizing crossovers, turns, etc. [7, 9, 15]. Our view is that good routes are not made simply by minimizing the number of corners and crossovers, but rather by controlling the location of the corners and crossovers, hence reducing the congestion of the figure.

Some parallel can be drawn between the modified line–expansion router utilized in *Eureka* [7, 17] and our global router, especially when congestion metrics are not employed. We gain a significant advantage over the router used in *Eureka* in that we map the route to a tile space, from which our route is queried for information used to accurately place other modules. The similarity exists in that during the local routing process, the global route selected is expanded to fit among the given obstacles (modules). Once this is accomplished, essentially the line expansion [19] for the best route has occurred, only it has been mapped directly to the tile space so that the ordering rules can be applied locally to produce the most traceable set of routes.

The division between the global router and the local router is an important one, and one not made by all researchers. We assert that there is a fundamental difference in the need to locate *where* corners occur (global routing) and *how* those corners are placed *with respect to each other* (local routing).

Our emphasis on congestion and space mechanics is not meant to ignore the importance of line crossings to the quality of the completed route. Even in areas that have a little congestion, improper line crossings can detract from the quality of the route. Figure 12 shows several instances where line crossing clearly detracts from the traceability of the lines. Three good combinations are shown on the left, juxtaposed with three bad combinations. The relative corner placements in Figure 12a are much better than those in 12b. There is a further complication to the problem of line positioning in addressing line–crossing issues, that being the linked nature of corners. Whenever a corner c is fixed, this necessarily fixes the lines connected to that corner. Fixing c restricts where the other corners linked to c can be placed, and although

c may have an optimal placement, the restrictions that this placement has on the corners linked to it may make their placement sub-optimal.

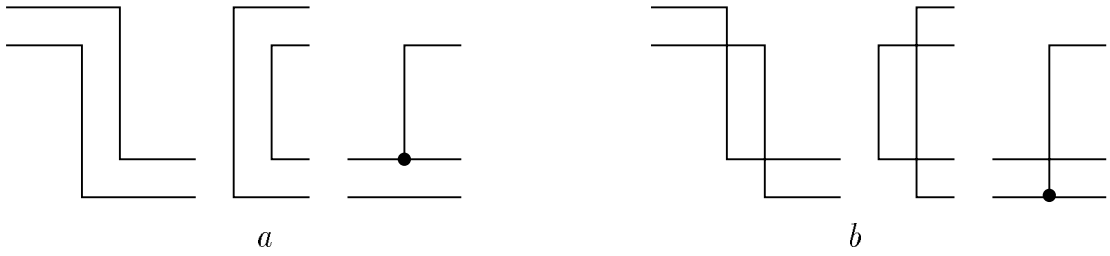


Figure 12 Good and bad line crossing solutions.

The line–positioning problem is certainly not a new problem, as the discussions presented in much of the previous work [3, 7, 9, 15] clearly illustrate the importance of line positioning to the readability of the schematic. We have found that line positioning does not need to be addressed in the global router, as line crossing problems are either fixed by the placement, or limited by the congestion metrics used in creating the global route.

Returning to the idea that net traceability is a local issue, the local router minimizes the route complexity given the positional constraints fixed by the global router. Here we extend locality to the placement of lines in a given area so as to best resolve the line–crossing issues there. The corners in a given area should be spread out so as to minimize the corner–density effects and to maximize the readability. The lines linked to these corners also need to be judiciously placed relative to each other, so as to best resolve the line–crossing issues.

We present a novel constraint-propagation approach in how we address line-crossing issues in our local router. We claim that by doing a good job in the most congested area, the propagation of effects can be limited, thus we evaluate (constrain) the most congested area first, and allow these constraints to propagate. Each area is evaluated by imposing an ordering relation to the corners in the given area. We place lines by applying our *Rules of Easy Reading* to the lines in a given area. These are illustrated in Figure 13.

As indicated in Figure 13, horizontal areas are evaluated to determine the ordering

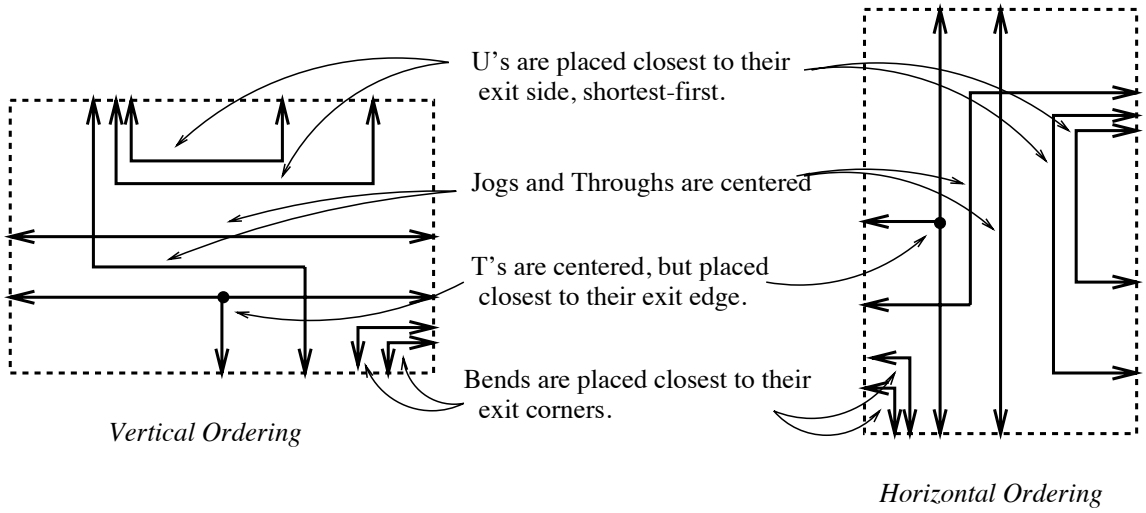


Figure 13 *Ordering Rules for Easy Reading:* Relative placement of segments

for the vertical lines contained. Similarly vertical areas are evaluated to determine the ordering of horizontal lines (Figure 13b). These rules insure that unnecessary line-crossings are avoided.

All of the above ideas are embodied in our Schematic Place And Route (SPAR) system which is presented in the next chapter.

3.0 SPAR OVERVIEW

This chapter presents an overview of the SPAR system. It introduces how we deal with the broad tasks of space management, partitioning, module placement, global and local routing. The details of the implementation of each of these ideas are explained in Chapter 4.

3.1 Separation of Placement and Routing tasks

SPAR is an algorithmic and heuristic approach to ASG that begins with the statement that the placement and routing tasks are interdependent, but can be separated. Given the nature of placement, in that a poor placement can make traceable routing impossible, SPAR falls into the category of systems that determine the module placement based on connectivity issues. That is, we try to discover a partitioning and a placement which exhibits a reasonable representation of the functional nature of the netlist. We assume that a well-partitioned diagram will be well-placed, and that by observing the signal inheritance that exists within these individual partitions, a good route will be achieved. This assumption is quite reasonable, as it emphasizes the placement of modules to meet the first of the two generation requirements - Functional Identification.

3.2 Space Management

SPAR explicitly manages the placement and routing space. This is accomplished by using sets of cornerstitched tile spaces, each with a specific purpose. The placement of all internal modules (not system terminals) occurs on a separate set of tile spaces, each partition being placed in its own tile space. These "pages" are then merged into one tile space. The relative placement information is then used to transfer all of the internal modules to the tile space that is used for routing. The advantage of

the transfer is that a bounding box can be established for the placement, and used to limit the area within which the route is expected to occur.

Routing is done by using two superimposed, 90°-rotated tile-spaces so that routes can compete with each other for space as they are generated. The use of two tile spaces is to take advantage of one of the peculiarities of the cornerstitching algorithm, that being that 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 can be obtained quickly for any arbitrary point.

We also use tile spaces to locate modules and nets. The tiles themselves contain pointers to the modules and nets located therein, which are used to guide the algorithms. This is especially important for the incremental placement of the system terminals, where the system terminals are added to an already-developed tile space that contains routes and modules. As each new module is inserted, the affected tiles are checked to see if the information contained is still valid.

3.3 Partitioning the Problem

We form useful partitions by discovering and highlighting internal structures within the design in order to achieve functional identification among the constituent modules. Important relationships are inheritance, global feedback and cross-coupling. Inheritance as applied to schematics is the chain of nets that link modules, such that if a module's output is connected to a second module's input, the second is said to inherit the first. Cross coupling is where one set of modules both inherit and are inherited by another set of modules. Each has an output that feeds an input of the other set. Global feedback is a distinction from cross-coupling in that the output of a whole set of modules connects to an input of the same set of modules.

We have found that effective partitioning can be achieved by using the connectivity information given in the netlist. These partitions will have an impact on the placement and routing process, so modules are placed within a partition such that the grouping will encapsulate local feedback and leave global feedback for the inter-partition algorithms.

We accomplish this by using clustering techniques [20] based on examining the connectivity between two given sets of modules. A connectivity matrix is built from the modules comprising the diagram, and modified as clusters of tightly connected modules are developed. Clusters continue to be built until a proposed merger of clusters fails to produce a good partition.

3.4 Module Placement

Given a specific partitioning, the placement of individual partitions is handled separately. The collection of modules within a partition is used to form a virtual page, and the modules contained are placed without reference to connections outside the partition. Given that these pages will be placed independently, the inter-partition placement issues are addressed later. Within a partition, modules are formed into strings, and it is in the string formation that the "rigid standards" of Section 2.2 begin to be applied systematically. For placement of modules within a partition, it is important that the selection and arrangements of modules be made with consideration of their *local* connections - in length, density, and need for bends. Within the partition, distances are assumed to be short, so many connections can be dealt with cleanly.

Our placement algorithm for modules within partitions is a series of improvements on that used in *Pablo*. On a given virtual page, sets of strings are developed that maintain the input→output left→right flow of information. These strings are merged together until all modules in the partition have been added to the page.

We improve upon *Pablo*'s string formation algorithm, in that we look specifically for complex strings. Strings of modules are seen more as rightward-fanning trees, rather than simple strings. This permits the proper identification of important features such as local feedback among modules. Another important difference is our use of signal position information within strings to merge strings together. By the use of the tile space, there is little need to worry that these strings take on complex shapes, as we are able to fit modules into any oddly-arranged spaces that might occur, as

shown in Figure 10. The heuristics that govern the merger of strings use the connectivity of the unplaced string, and the locations to which the string is attached, and use this to position the new string in it's partition.

Once the modules within the partitions have been placed on individual pages, these pages are merged to form a complete placement. This arrangement must keep the number of long connections low, so as to avoid cluttering the diagram with long routes that do little to aid in function identification. To accomplish this, partitions are merged together using a set of heuristics similar to those used to merge strings within the partitions. These pages are merged one-by-one until all modules saving system terminals are placed.

System terminals are handled separately, as there are very specific expectations as to where they should be placed. Input terminals are expected to be on the left, and outputs on the right, all arranged in columns. Furthermore, the terminals are expected to be aligned wherever possible on their source or left-most destination terminal. This is accomplished by first placing the modules central to the schematic, and routing them. This routing information is maintained in the tile spaces used for routing, and is queried to guide the placement of the system terminals. These terminals are then incrementally added to the routing spaces, and the route is completed. The details of this are explained in Section 4.5.

3.5 Global Routing

Routing in SPAR is broken into two phases: global routing, where the general location of runs and bends in the nets is discovered, and local routing, where these overlapping areas are seen as combinations of constraints that need to be satisfied. The global router determines the number and location of bends in the broad sense, whereas the local router places the corners and makes connections among them. These two techniques aim at maximizing the readability of the resulting diagram.

Our global router performs a heuristic search by making multi-point node expansions using cost metrics based on the *complexity of the route* (*i.e.* the number of bends) and the *congestion of the route* (meaning the number of other nets and

corners that are grouped in areas that the net passes through). This parallel breath-first search aims at finding the lowest complexity/congestion value for each net. As congestion is a function of the other nets in the circuit, not the one in question, congestion can only be evaluated after a set of global routes has been proposed. Once a route exists, measures for congestion can be evaluated, and the nets are relaid using the existing congestion information. The actual mechanics are described in Section 4.3.

A global route is therefore only a rough plan of where the given nets will run or bend, and is not an exact representation of the route. Rather, the tiles in which the routes go are only important in that they locate the number of and general location of the bends in the route. The process of specifically locating the corners is left to the local routing routine.

3.6 Local Routing

Our local router is based on a novel constraint propagation technique, and is arranged to place the corners that will make up the route for a given net. Once all of the nets in a diagram have been placed in the tile space, the global routes for these nets consist of lists of tiles in which the given nets run. The tiles of the global route define the minimum areas in which the lines and corners that make up the local route can be placed. Thus the global route is used to constrain the local route. These constraints are known as *ranges*, as a constraining range may extend over several tiles. Several nets may use the same routing tile so the legal ranges of these nets will necessarily overlap each other. This overlap must be resolved before the routes can be positioned, and is the major task of the local router.

Given there is finite number of ranges that make up a net, and that a finite set of nets will have been routed through any given area, we reduce the local routing problem to a constraint satisfaction problem. The problem is to find the best solution for the corner locations of each net in each tile, starting with the constraints of the global route. Within this solution, the line-crossing problem mentioned in Section 2.2 must be addressed. In every case there exists an ordering that will produce a good

resolution to the corner positioning problem which in turn will yield a good schematic route.

The tiles of the global route are used to determine the initial ranges that define where corners will fall. The local router expands these ranges from those determined from the global route by expanding into all adjacent tiles that do not obstruct the run. The complete range in which any given corner can be located may extend over several adjacent tiles.

Aside from simply choosing the positions of corners for each of the nets, there also arises the issue of how to communicate the dependency of one corner-placement decision on another. For example, the placement of corner X will affect the location (or be affected by the location) of the corners connected to X. Inherent to this is an ordering problem of which corner should be placed first. Choosing one position will limit the options available to the other corners that are as yet unplaced. This is further complicated in that corners occur in distant tiles. Yet as one corner is fixed, the change must be propagated to all other corners of the same net that are in connected either horizontally or vertically.

We solve these problems by using the congestion information obtained in the global routing phase. We use a greedy algorithm that evaluates the corner positions of all of the nets in the worst tile first. This evaluation is done separately for horizontal and vertical segments, and involves creating a list of ranges that use a given tile. These ranges are then separated so as to best conform with our *Rules of Easy Reading* mentioned in Section 2.2. Each separation restricts the range in which a corner may fall. As corner positions are restricted, all connected corners are also restricted. In this way, as more tiles are evaluated, the restrictions in location grow and are propagated to other, as yet not-evaluated tiles. This continues until all corners have unique ranges for their position. The most congested tiles are evaluated first, so that the most congested (least traceable) set of corners will be most readable.

The advantage to this approach is that it orders the positioning problem cleanly, so that only a small number of positioning decisions need be made in each area. Since the typical congested area starts with several nets that have overlapping constraints, by locating these nets first and communicating these decisions to the linked corners,

further tiles become more constrained. As the algorithm progresses, the tiles become progressively easier to solve. In essence what this technique does is to identify the important ranges to be resolved, and then delay the evalutation of non-critical (congested) ranges until more critical ranges have been fixed.

Having presented an overview of the key ideas in the algorithms used in SPAR, we are now ready to discuss the details of their implementation. These details are the subject of the next chapter.

4.0 SPAR DETAILS

This chapter gives an in–depth discussion of the algorithms used in SPAR, divided into the five major tasks: *Partitioning, Placement, Global Routing, Local Routing,* and *Incremental Placement and Routing of System Terminals.* To place these actions in context, Figure 14 presents an overview of the algorithm used in SPAR:

- Group the modules into functionally related partitions. (Partitioning)
  - Form clusters of modules that are functionally related.
  - Separate system terminals from interior modules.
- Place the Modules in a recognizable way:
  - Place each partition on its own virtual page. (Intra-Partition Placement)
  - Merge virtual pages into a single page. (Inter-Partition Placement)
- Determine the tiles where nets should be allowed to overlap: (Global Routing)
  - Search the routing space for a good solution independent of congestion.
  - Refine the routes, accounting for the congestion present.
- Locate and fix the corners of all the nets: (Local Routing)
  - Develop acceptable ranges where net corners can be placed.
  - Resolve and fix the location of corners into unique positions.
- Place the system terminals on the edge of the page: (Incremental Placement)
  - For each net, find what sections of the route map to the page edge.
  - For each net that does not map, find a nearby unobstructed path.
  - Complete the global routing.
  - Locate and fix the corners for all of the nets (Local Routing).

Figure 14 Outline of SPAR

The details of how our space management techniques are used are included within the discussions of each relavent task. All tasks except *Partitioning* rely on the space

management technique we employ. Following this discussion of the algorithms employed, we present illustratve examples in Chapter 5.

4.1 Partitioning - Discovering Functional Relationships

We begin the ASG process by partitioning a netlist using connectivity clustering techniques. Modules are grouped into clusters, which are checked against partitioning criteria. At each iteration of the algorithm, two clusters are merged based on an evaluation of the *connection matrix*, which organizes the connection counts among the clusters. This merger is conditional. Whenever the two clusters that are being merged would form a cluster that does not make an acceptable partition, one of the two is saved as a partition. The second cluster remains in the matrix for further processing. The process continues, with either a merger occurring, or a cluster being saved as a partition until all clusters are part of a partition.

This process starts by assigning each module to an individual cluster, and creating a connection matrix similar to that used in *VISION*. The connection matrix entries represent the number of connections among the current set of clusters: The off-diagonal slots contain a count of the number of connections between the two indexed clusters. The diagonals are used to maintain the total number of connections that emanate from the given cluster.

The entire connection matrix is evaluated for *cluster values*, that is a number that rates how well-connected any two clusters are. This value is a normalized count of the number of connections that would emanate from the cluster resulting from merging two clusters. Given any two connection matrix (CM) entries i and j, the cluster value is defined to be:

$$Cluster\ Value_{m,n} = \frac{CM[m,m] + CM[n,n] - CM[m,n] - CM[n,m]}{CM[m,m] + CM[n,n]} \qquad (4\text{-}1)$$

A cluster value is computed for each pair of clusters in the matrix. The two clusters i,j that give the smallest cluster value are selected for merger. This merger

causes the connection counts in the matrix to reflect the sum of connections external to the cluster. This reduces the number of clusters remaining by one. Thus one row and one column are removed from the matrix at each iteration of the clustering algorithm.

As each merger occurs, a pair of clusters forms a single, larger cluster. These mergers yield the bottom-up creation of a binary cluster tree, from which desirable partitions are pruned. After each merger, the new cluster is checked to see if it is acceptable as a partition, and the process continues. The clustering process continues until there is only one cluster remaining for merger, that is all modules have either been pruned as partitions, or merged with the last cluster, which becomes a partition. Figure 15 shows the initial connection matrix for the SN7474.

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

Figure 15 Initial connection matrix for the SN7474

Figure 15 lists one column for each module in the diagram, for a total of twelve: four system input modules, two system output modules, and six interior modules. The column and row entries count the number of connections between any two clusters. For example, column i and row j contains the count for the number of connections between the ith cluster and the jth cluster, which in the figure corresponds

to modules *nand4* and *nand5*. Figure 16 shows the schematic from the TTL Data Book [21] for reference. The dotted lines indicate the clusters represented in the matrix. Along the borders of each box, a mark has been made where the route exits the cluster, with a number indicating the number of connections that are made to the cluster via that route. For each cluster pictured, these sum to the number on the corresponding diagonal of the matrix.

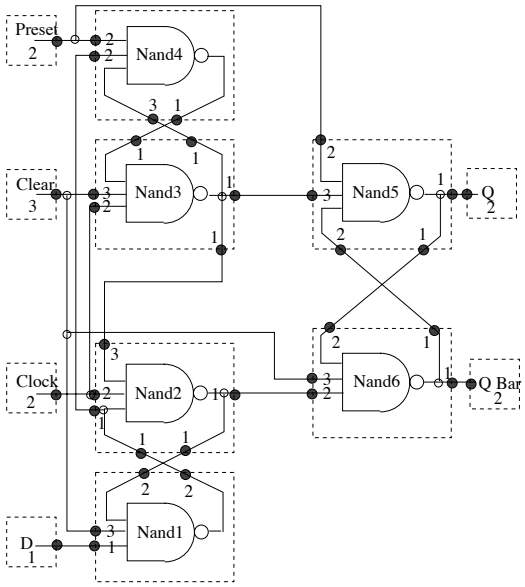


Figure 16 Initial clusters for the SN7474

The cluster values for each pair of clusters are calculated, and the pair of modules with the lowest cluster value would be clustered together. Clusters i and j yield the lowest cluster value:

$$Cluster\ Value_{i,j} = \frac{CM[i,i]+CM[j,j]-CM[i,j]-CM[j,i]}{CM[i,i]+CM[j,j]} \quad (4\text{-}2)$$

$$= \frac{8+9-2-2}{8+9} = \frac{13}{17} \cong 0.765 \quad (4\text{-}3)$$

In this case the clusters to be merged correspond to modules *nand4* and *nand5*. The result of this clustering is that row and column j of Figure 15 are merged with

row and column i. The results of this merger are shown in row and column j of Figure 17. The clusters corresponding to the modified matrix is shown in Figure 18.

	clr	set	clk	d	q	qbr	1	2	3	4+56		
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 17 Connection matrix for the SN7474 after one merger

At the next iteration of the clustering algorithm, the clusters that form the lowest cluster value correspond to rows i and j of Figure 17. These correspond to the modules *nand3* and *nand4 + nand5*. The cluster value calculation for these two clusters is:

$$Cluster\ Value \quad = \quad \frac{9+13-3-3}{9+13} = \frac{16}{22} \cong 0.727 \qquad \text{(4-4)}$$

The clustering process proceeds as described until all clusters have been merged or pruned from the matrix to form partitions. There are several strategies that can be used for this pruning of partitions. Three partitioning strategies have been implemented: fixed size, Rent-like, and slope-based. The strategy employed in creating the diagram is a user-supplied argument.

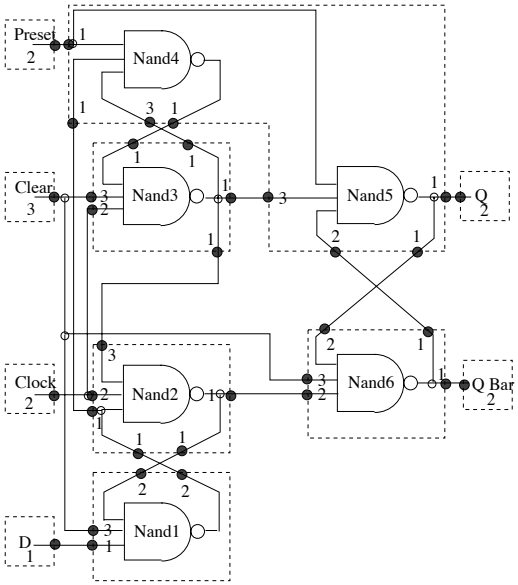


Figure 18 Clusters for the SN7474 after one merger

Given two clusters $cluster_i$ and $cluster_j$ where m_i is the number of modules contained in $cluster_i$ and c_i is the number of connections external to $cluster_i$, Size-based partitioning is qualified by specifying some size limit S:

$$if\ \ (m_{i+j} > S)\ \ then\ \ partition = \begin{cases} cluster_i & m_i > m_j \\ cluster_j & \text{otherwise} \end{cases} \tag{4-5}$$

Rent's rule [22] can be applied, where partitions are formed if the ratio of external connections to the number of modules exceeds a specifiedratio limit R:

$$if\ \ (\frac{c_{i+j}}{m_{i+j}} \leq R)\ \ then\ \ partition = \begin{cases} cluster_i & \frac{c_i}{m_i} \leq \frac{c_j}{m_j} \\ cluster_j & \text{otherwise} \end{cases} \tag{4-6}$$

Slope-based partitoning forms a partition if the number of connections in the proposed partition is less than the largest of the sum of the connections in the child partitions. This method requires no qualifiers, and is defined by:

$$if\ \ (c_{i+j} < MAX(c_i, c_j))\ \ then\ \ partition = \begin{cases} cluster_i & m_i > m_j \\ cluster_j & \text{otherwise} \end{cases} \tag{4-7}$$

The clustering shown in Figures 15 and 17 was pruned using slope–based partitioning. This led to the six central modules of the SN7474 to be divided into

two partitions, as shown in Figure 19. This example illustrates how highly interconnected modules are appropriately separated by combining connectivity clustering with slope-based partitioning.

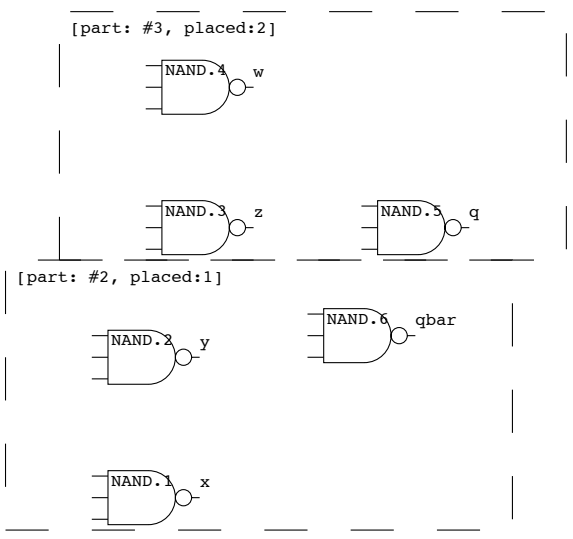


Figure 19 Clustering results for the SN7474

Specifically, slope-based partitioning builds small groups of modules that are strongly interconnected. Larger groups of modules that are interconnected will not be merged, as typically more external connections are added than are collapsed from the merger of the two clusters. In this way, sets of highly-interconnected clusters are saved, leaving strong links by which the partitions can be placed with respect to each other. The slope referred to here is the change in the number of external connections between iterations of the clustering procedure. As clusters grow this slope is at first positive, but levels out as larger clusters are merged.

Once all modules have been assigned to partitions, the partitions are placed on their own virtual pages as will be described in Section 4.2. Figure 20 shows the partitioning and placement for the SN54S151 multiplexer. This schematic has several groups of tightly-interconnected modules: the five-input AND gates that are arranged in two partitions that form a column, and the eight-input OR which is connected to both of the AND partitions. The AND groupings are formed as adding each

gate reduces the number of external connections, because the gates are so tightly connected. Once the larger partitions are broken off, the remaining modules are more connected to the central partition than they are to each other, an thus will not form partitions.

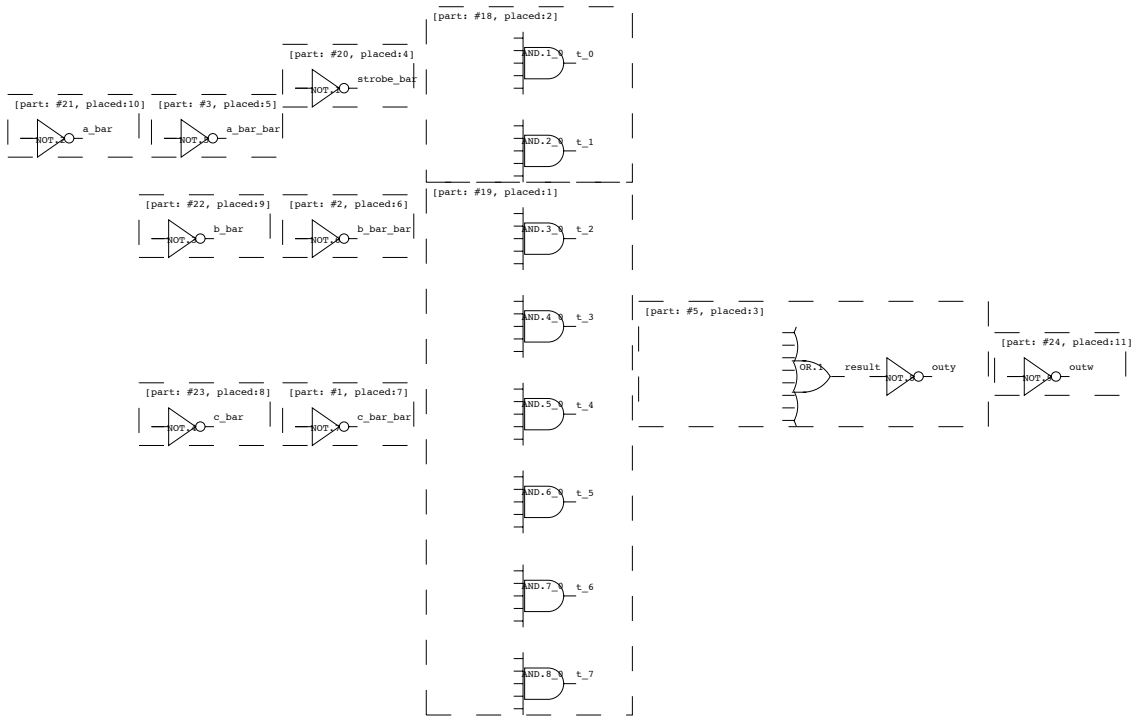


Figure 20 Slope-based partitioning and placement for the SN54S151 multiplexer

4.2 Placement of Modules

The placement process focuses on two tasks: Placing each of the partitions developed on its own virtual page and then merging these pages to form the completed placement. Each of these tasks is carried out with respect to the routing considerations mentioned: Local issues dominate the placement within a partition. Global issues dominate the relative placement of these partitions.

4.2.1 Forming Identifiable Blocks

Given a set of modules grouped into partitions, each partition is placed independently from the rest of the design on its own virtual page. This reduces the placement problem, and focuses the algorithms on a small, independent subset of the design. The placement is accomplished by building strings [7], each of which is placed in its own tile. These strings are formed and their tiles are added to the virtual page until all modules have been placed.

A string is a list of modules formed by tracing the out→in relationships among the unplaced modules within a partition. These modules are pin-aligned; each module to the right has its input pin aligned on the output pin of the module to its left (the module that is driving it). If cross-coupling is detected within a string, a complex string is formed, whereby the string branches at the detected cross-coupling. At the branch point, two substrings are formed from the yet unplaced modules of the partition. Three such branches are depicted in Figure 21 in signals *q* and *qbar*, *r* and *rbar*, and *p* and *bar*.

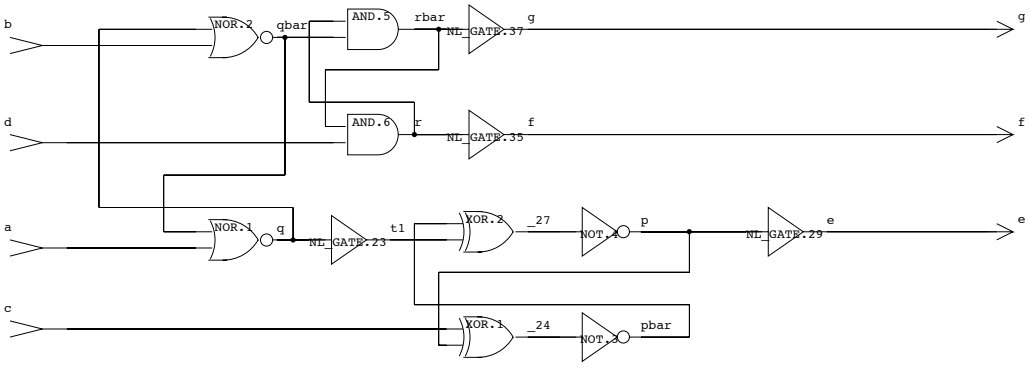


Figure 21 A Complex String

The string formation process continues until there are no more gates that meet the output→input relationship that defines a string. Space is added around the string to insure routeability, and defines the dimensions of the tile in which it is placed.

Once a string is assembled, its tile is added to the page associated with that partition. Adding the first tile to a page is simple, as there are no existing tiles for it to be placed against. The procedure for adding other tiles is more complicated, as the tiles need to fit into the available free space, and be aligned so that the routing among the modules contained is reasonable. We use the angled insertion technique depectied in Figure 22 to accomplish this.

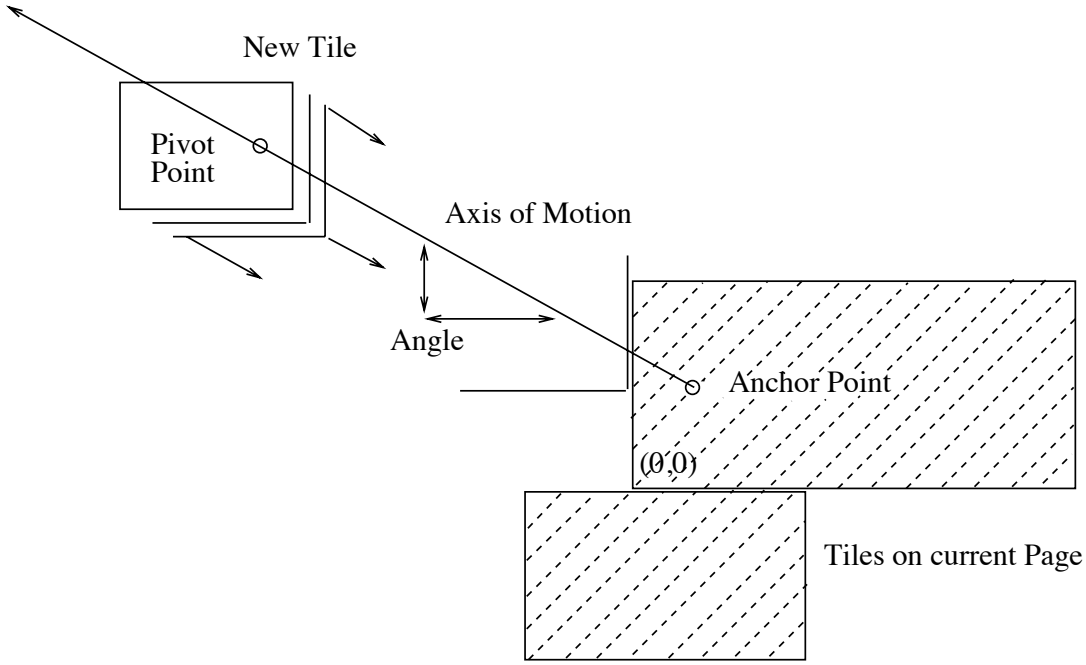


Figure 22 Tile Placement Example

For the tile being inserted, a count of the most dominant form of connection is made (input→input, output→output, input→output, and output→input). This count is used to determine which side of the existing tiles the new tile should be placed against. In→in relationships will cause the new tile to be placed above the existing tiles, out→out below, in→out place tiles to the right, and out→in to the left. The two dominant counts form the slope that determines the *axis of motion*, along which the new tile is inserted. This axis is a line that is defined by this slope, and an *anchor point* determined by the terminals contained in the old tiles which connect to the terminals contained in the new tile. A *pivot point* is established within the new tile, and governs how the new tile moves along the axis, as illustrated in Figure 22.

The anchor and pivot points are determined by the side that is selected for insertion. For example, if the new tile is to be placed to the left of the placed tiles, then the right-most terminal that connects to the placed tiles is used as the pivot point, and the left-most, connected terminal in the placed tiles is used as the anchor point. Newton's method [23] is used to place the new tile along the axis of motion such that the new tile (containing the pivot point) is placed as close as legally possible to the anchor point.

The process of assembling strings, fitting them onto tiles, and inserting the tiles on the virtual page continues until all modules within the partition are placed. Figure 23a shows the string definitions for a single virtual page of an 8-bit ripple-carry adder. The route for this partition is included in Figure 23b to illustrate the connectivity among the strings.

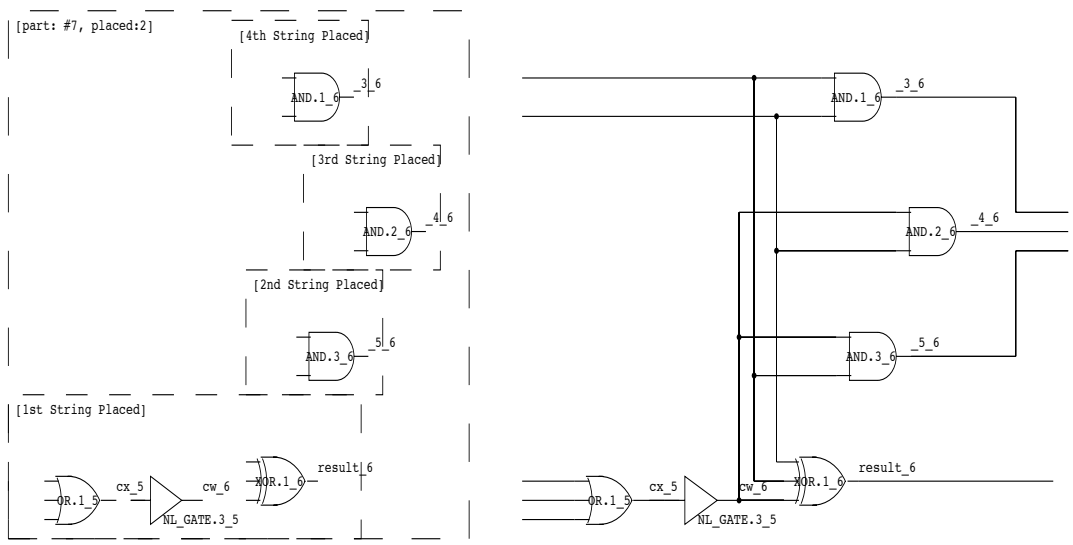


Figure 23 String placement for a partition of a Ripple-Carry Adder(a) and its final route (b)

4.2.2 Merging Blocks

Once all modules have been placed on virtual pages, the task remains to merge these pages and form the complete placement for the schematic. This is done by

ordering the partitions, and merging them together one at a time. 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 connection. These succeeding partitions are placed about the finished partitions in the same manner as tiles were placed on the virtual pages.

The connections between the page being added and the already-placed pages are weighed, and the most dominant are used to determine the relative positioning of the new page. This heuristic is tuned to allow groups of modules at the same inheritance level to be placed parallel with each other. For example, if two groups of modules share many of the same inputs, then the new page is placed above where the connected modules have been placed. In the page merging procedure, groups of tiles merged with the tiles representing the already-placed modules in the same way that string tiles are added to the individual virtual pages. This is a simple extension of the angled insertion procedure (Figure 22) to groups of tiles, rather that a single tile. The usefulness of this insertion technique can be seen in the placement example for the function generator [24] depicted in Figure 24. Here the bounding boxes of partitions are provided, and it is easily seen how groups of modules overlap these boundaries, forming a tighter placement than would be possible if we were restricted to the bounding boxed of the partitions.

The placement technique described insures that the completed placement anticipates the routeability of the diagram. It groups modules based on their connectivity, using these groupings to align successive modules on related terminals, thus avoiding jogs. This technique also handles important special cases, such as local feedback, common inputs, and common outputs, arranging the modules so that routes have sufficient space to complete. Special cases are dealt with at the string formation and placement level, which is tailored to highlight routing channels and cross–coupled gates.

The inter-partition placement is arranged to address more global features, as is illustrated in the 4-bit ripple-carry adder of Figure 25. Here, sized-based are formed and merged together, and combine to clearly show the overall flow of information within the adder. The combination of the partitioning and the placement

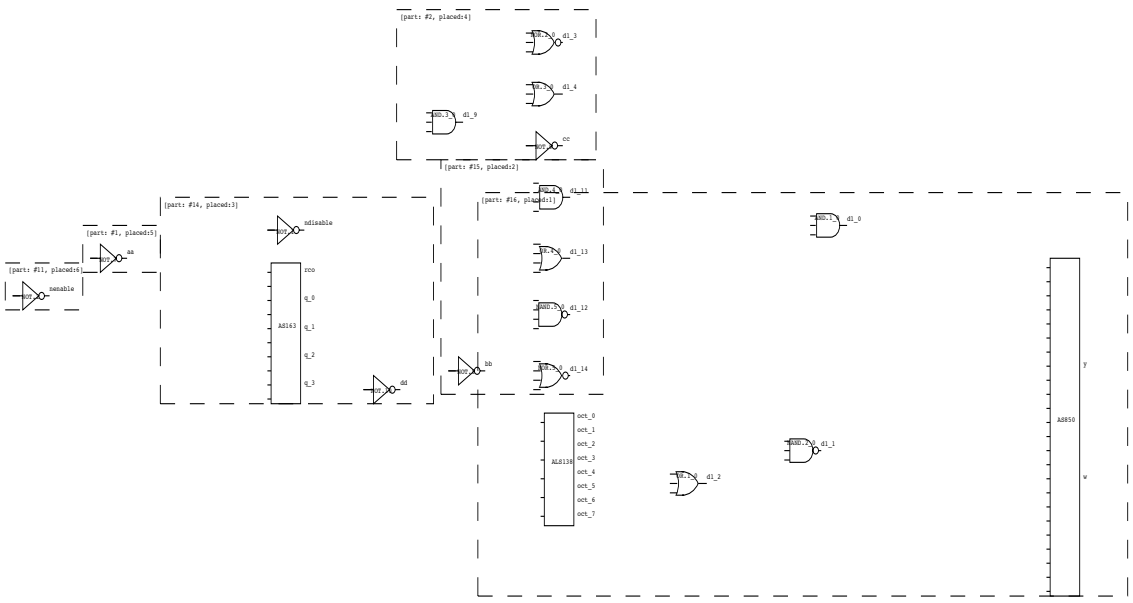


Figure 24 Placement for a function generator, showing partition overlap

rules achieve this, and combine to insure that the overall signal flow is maintained. The completed schematic is included in Figure 26 for reference.

The global routing algorithms that connect the placed modules are described in the two sections that follow. The overall paths that the each net will use is determined, and then the specific locations of corners is determined. All this is done so as to maximize the traceability of the resulting diagram during local routing.

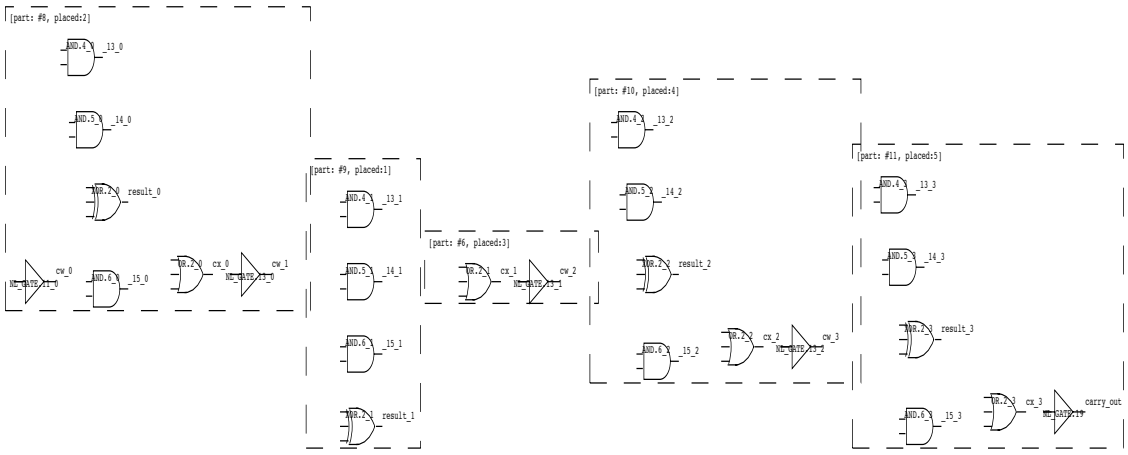


Figure 25 Partitioning and placement for a 4-bit Ripple-Carry Adder

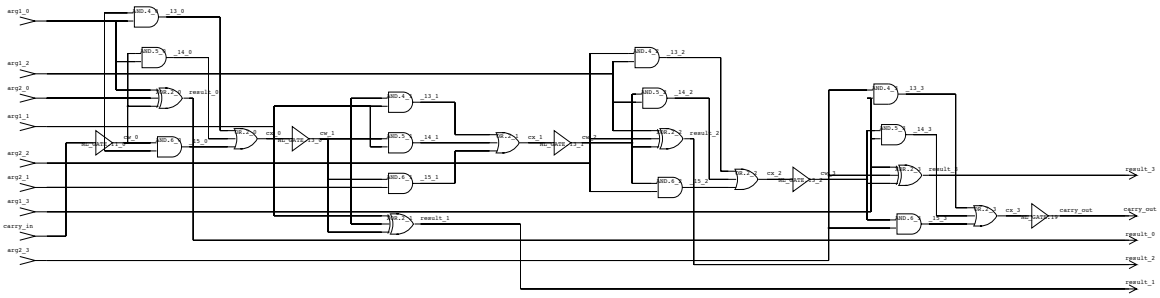


Figure 26 Completed 4-bit Ripple-Carry Adder

4.3 Global Routing

In global schematic routing the most basic requirement is that all nets complete. Once this requirement has been met, our primary interest is in the location of the corners of the route within the routing space, and the accompanying aesthetic requirements. We address these in the search method and cost metrics used to discover the routes.

The discovery of the global route is broken into two passes: the first pass aims at finding a good approximation to the best routes for each net, and the second selectively rips up and reroutes the diagram taking congestion into consideration. The mechanics of each operate in the same way, saving that the cost metrics are different. In the first pass, the router operates on all nets in parallel, with no congestion costs included. In the second pass, each net is evaluated separately. The worst net being evaluated first, with congestion considerations included.

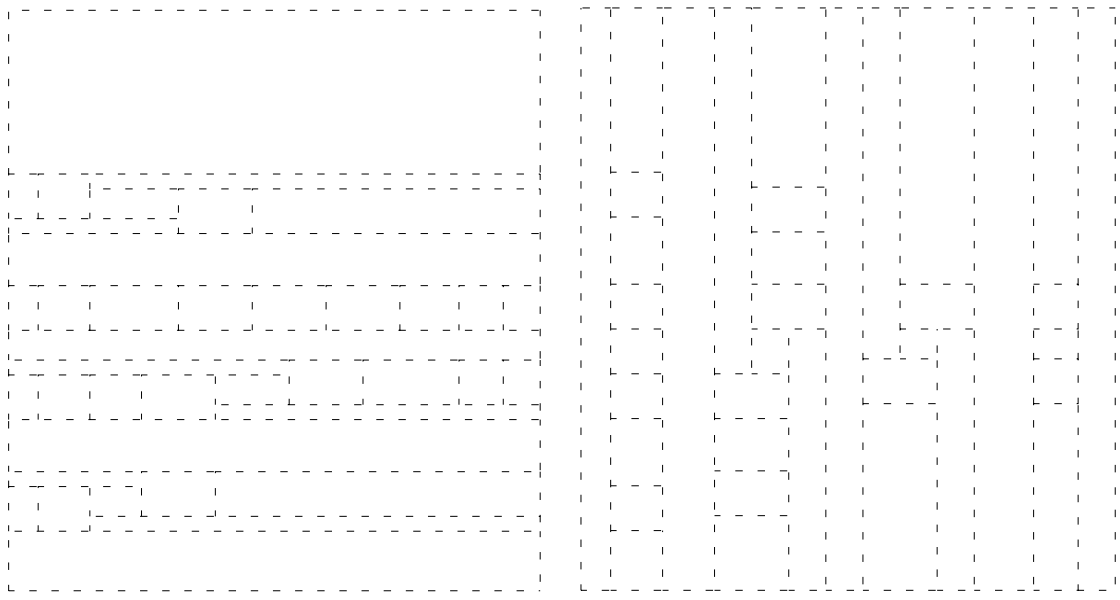
Two passes are needed, as congestion information cannot be known until all nets have been laid. The first-pass global routes are used to calculate congestion, so that the overall route can be optimized for traceability. The two facts that a) no congestion values are available until a first set of routes has been made, and b) the relative 'goodness' of a route will change once there is congestion to consider, combine to imply that the first 'best' route may not be the final route. Rather than recalculate routes after the congestion is known, all possible paths are maintained as the search progresses. Therefore, 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. In this way we avoid re-running the (time-consuming) search process.

4.3.1 The Search Process

The search process is actually a series of searches, one for each terminal of all nets. A partial route (*path*) is constructed until the path for one terminal connects to another path of the same net. A path that connects is said to be *complete*. The collection of paths associated with a given terminal of a net is referred to as an

expansion. At each move, 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. The search takes place in parallel, in that each expansion (there is one expansion for each terminal of each net) will make only one move until all expansions have completed.

Each move in the search consists of a single tile from the routing space being added to an existing path. This space is composed of two, superimposed 90°-rotated tile spaces, which contain the modules as obstacles to the route. Figure 27 shows the two routing spaces for the SN7474, as parallel horizontally-oriented and vertically-oriented spaces. Figure 27a contains the horizontal space, and shows how the tiles are arranged in long horizontal strips. Similarly, the vertical space in Figure 27b is arranged in long vertical strips. Our global route is determined by performing a breadth-first search within these two tile spaces. For reference, the SN7474 is used as an example throughout Sections 4.3 and 4.4 for continuity.



a *b*

Figure 27 Horizontal(a) and vertical(b) tile spaces for the SN7474

A global route is an ordered list of tiles that maps a trail from the source terminal to the destination terminal. A route typically begins as a single horizontally-oriented (h-)tile that is adjacent to a terminal on a module. As the search progresses, a vertically-oriented (v-)tile that overlaps the previous h-tile is added to the list. Another h-tile will be added, and so on until a valid destination is found.

Each search step proceeds by selecting an expansion and choosing the lowest-cost path yet seen. The tile on the end of a path is used to select a set of perpendicularly-oriented tiles which are expanded into, thus creating more paths. An expansion only occurs if the tile selected has not been seen before to avoid cyclic behavior. Figure 28 shows one expansion of net for 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 and one vertical. The last horizontal tile on the path is being expanded into the vertical plane, and can expand into the four vertical tiles 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.

This example shows an important aspect of path expansion, that tiles are only visited once by a particular expansion. Figure 28 shows a long horizontal tile that crosses five vertical tiles, only four of which are used for expansion. Tile 2 is already on the path being expanded. Were the router permitted to expand into tiles that have already been visited, then the path that expanded would contain the same tile twice, and would continue to find the tiles again and again. This problem is a cycle problem, and is avoided by only permitting an expansion to visit a particular tile once.

When a tile of a path is going to be checked for expansion, it is first evaluated to see if it completes a connection to another expansion that is part of the same net. If this expansion contacts another, yet-incomplete expansion of the same net, it may terminate. All of its paths will be merged with those of the still-active terminal. The expansion process continues until a termination leaves only the contacted terminal active, which is then terminated. This ensures that all terminals on a net are connected. The contacted expansion is merged with the active one to ensure that the paths for the terminated expansion can be used to complete other active expansions.

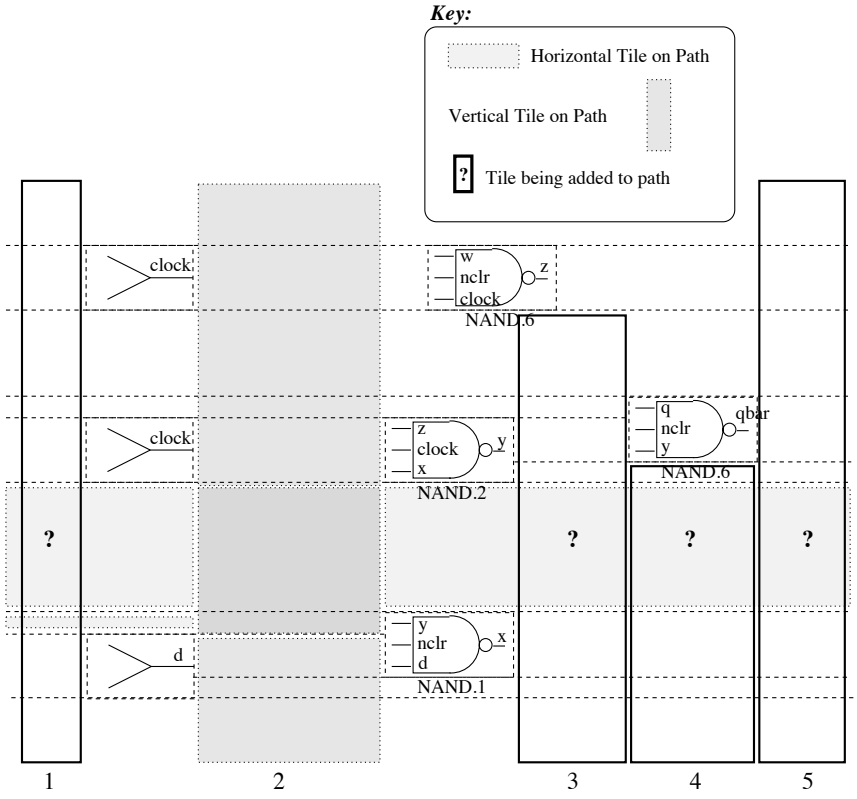


Figure 28 Path Expansion for one path of net y for the SN7474.

When all expansions on a net are terminated, then the net is completed. When all nets are complete, then the global router is complete. The following four steps summarize the actions taken at each search step:

1 *Select the least-expensive path for the given expansion.* This is the path that will be expanded. Of most importance is the last tile on the path, which is called the *current tile.*

2 *Check the current tile to see if this path should terminate.* Termination occurs when a terminal expansion (set of possible paths to the given point) contacts another active expansion.

3 *Use the current tile to expand into all connected, free, perpendicular tiles.* For each valid tile selected, a list of perpendicular tiles is collected, and for every tile not yet visited, the current path is copied, with new tile added to it. This

'not-visited' criterion is to avoid creating cycles, which would cause duplicate paths to compete for expansion.

4 *Determine the cost of each of these new paths.* The active paths for a given expansion are ordered. Therefore, whenever this expansion is checked, its least-cost path can be quickly found.

4.3.2 The Cost Estimation Metric

The key to the success of the search process is the nature of the cost estimate used. There are two key factors to consider: The first is that the cost estimate models the relative importance of the various qualities of the finished route, such as low congestion and few turns. The second is that a low cost estimate of the distance-to-go is provided. This insures that all paths are comparable, and that once a completed path is discovered to be the lowest-cost path, then there are no better paths between the contacted terminals.

The path cost estimate is made up of several parts: a length cost, a corner cost, and a congestion cost. Here the length cost is a weighted measure of the path length, the corner cost a weighted measure of the corner count, and the congestion cost is a weighted measure of the usage of the areas in which the corners fall.

$$Path\ Cost\ Estimate = Congestion\ Cost + Corner\ Count \times cw + Path\ Length \times lw \tag{4-8}$$

where:

$$Path\ Length = \overbrace{\sum_{i=1}^{no.of tiles} \overline{distance(tile_i, tile_{i+1})}}^{Avg.\ Known\ Path\ Length} + \overbrace{\overline{[|termX - \bar{x}| + |termY - \bar{y}| \times f]}}^{Est.\ Length\ to\ Nearest\ Terminal} \tag{4-9}$$

Where $\bar{x}$ and $\bar{y}$ are the position of the last known corner in the path, and $termX$ and $termY$ are the locations of the terminal nearest to ($\bar{x}$, $\bar{y}$).

An illustration of the cost calculation is provided in Figure 29. The dashed line in the figure represents the estimated length to the nearest terminal, the solid line

represents the average known path length. $(\bar{x}, \bar{y})$ occurs where these two lines meet, and $(termX, termY)$ is the output terminal of module *NAND.2*. The corner count is three, and the congestion cost would be dependent upon the existing routes for the other nets.

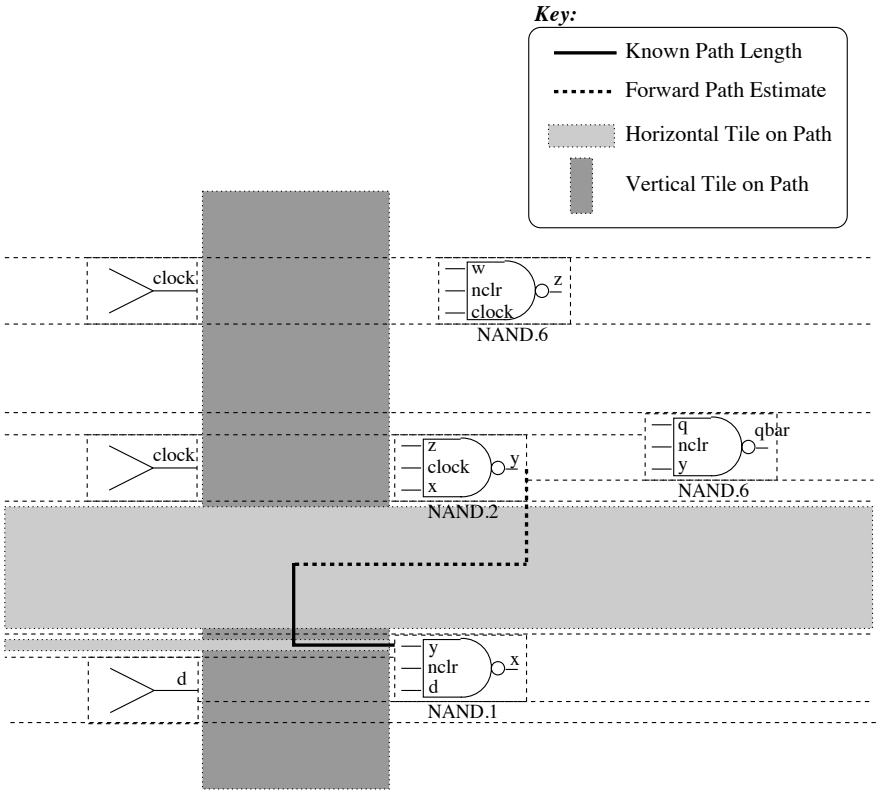


Figure 29 Path Cost Estimation for one path of net y.

As shown above, he length estimate consists of both a known length, given by the average length of the path from source to current terminus, and an estimated length, which is the rectilinear distance from the current terminus to the nearest possible terminal. The average distance is used in an effort to keep the cost estimates for all paths comparable, even though some routes will tun more tightly around obstacles than others. Figure 29 has these two lengths highlighted where the known length is marked as solid and the estimated length as dashed.

The completion of the first search process leaves a first–pass at the route: It should be a reasonable approximation to the best route, in that the paths selected have been optimized for the number of corners and the length of the route. Figure 30 shows the

completed global route for net y of the SN7474. The shaded regions indicate those tiles selected as the best path.

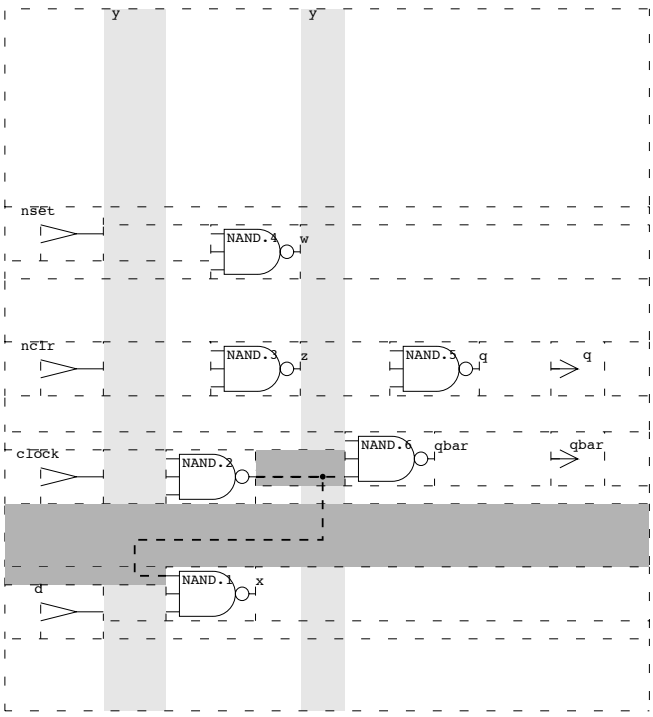


Figure 30 Global Route for net y showing the tiles selected

Once the first-pass paths for all nets have been selected, the path costs for all nets (both those completed and those still incomplete) are recalculated to include congestion. The overall congestion associated with the first-pass path for each net is evaluated and used to order the nets for the iterative refinement stage.

4.3.3 Using Iterative Refinement to Solve Congestion Problems

Iterative refinement consists of ordering the nets by their congestion values, and working from worst net to the best, ripping up the route for that net and rerouting it. This consists of taking the first-pass paths that make up the route, and adding them back to the list of more expensive or incomplete paths associated with the expansions for that particular net. As all of the routing information has been retained, it is simple to recost the paths contained in the net's expansions. The search process is restarted

once all of these paths have been reordered. Depending on the congestion problems, the old route might well be reselected, but this is by no means assured.

As mentioned, congestion is a measure of the density of use. Common sense dictates that if a small tile has several corners in it, then it will be more difficult to trace the nets that pass through. If the tile has more nets than there is space for, then the congestion is critical, and must be avoided. Correspondingly, our cost metric models these rules in that it adds to the cost of a particular path based on the number of nets that actually pass through each tile on the path. Normally, the congestion cost is a weighted ratio of the number of tracks used, over the number of tracks available. The exception to this rule is when this ratio equals one. This occurs when there are as many nets using the tile as can be fit. Such a tile is deemed *overused*, and warrants special treatment. The cost for overused tiles increases geometrically, so that any path using such a tile is unlikely to be a low-cost path, and likely to be ripped up if it is part of a first-pass path. To summarize, the congestion cost is defined to be:

$$Congestion\,Cost = \begin{cases} \frac{Tracks\ Used}{Tracks\ Available} \times w & \text{if } Tracks\ Used < Tracks\ Available \\ 2w \times \frac{Tracks\ Used}{Tracks\ Available} + c & \text{otherwise} \end{cases} \tag{4-10}$$

where w is a constant weight which scales the congestion values to the other values in the path cost estimate. A constant c is used to insure that the geometric progression for tile overusage is sufficiently steep to prevent overused paths from being considered.

It is useful to note that because the net being refined has been ripped up, the congestion values for the paths of the net are completely determined by the *other* finished paths. The expansions for the ripped up net will continue to search for the lowest cost path until one is found, thus adjusting the ripped-up net for the congestion of the other nets. All nets are ripped up and rerouted only once, since we have discovered that the reroute process tends to converge very quickly. Typically, a second iteration using congestion will discover the same route as the first iteration.

The value of re–routing the nets using congestion factors can be seen in the two examples of Figure 31. Figure 31a shows the completed SN7474 flip-flop without congestion considerations, and Figure 31b 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. For reference, the diagram of Figure 31b is compared with the schematic presented in the *TTL Data Book*[21] in Figure 43.

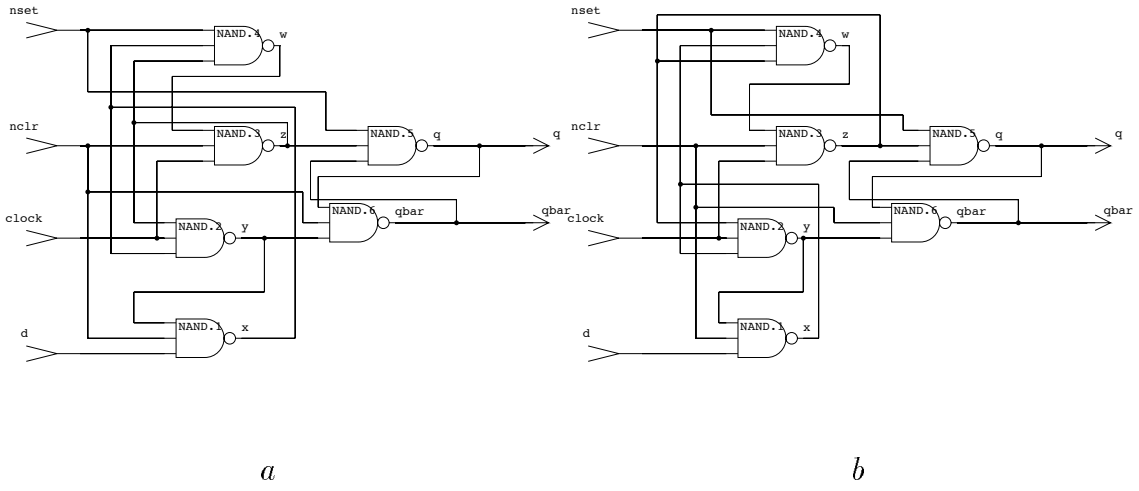


Figure 31 Routing for the SN7474 with congestion turned off (a) and on (b)

4.4 Local Routing

Given that the global router has produced a sequence of tiles marking the areas in which the lines are to be run, the work that remains is to position the corners of these lines. These corner assignments are chosen so as to address the crossover problem mentioned in Section 2.2, and to limit the effects of congestion of corners on the traceability of the routes. We arrange these assignments to follow the *Rules of Easy Reading* depicted in Figure 13.

The local routing process has two phases: a setup phase, where the setup algorithms are applied to each net independently, and a separation phase where separation algorithms are applied to all tiles containing net fragments, starting with the most congested tile first. The setup algorithms map the global route into corners linked by common ranges. As many nets may share a tile, these ranges are typically overlapped. The separation algorithms restrict these overlapped ranges to unique, non-overlapping values, and then to fixed points, which completely define the routes.

4.4.1 Mapping the global route into linked corners

The setup 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 consist of *ranges*, where a range is a continuous set of points that match 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, and are used to link attached corners. 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.

This mapping of tiles into ranges is complicated by the fact that the global route for a multi-terminal net consists of several finished paths, and wherever two paths meet, there will be a tile that is duplicated. Duplicate tiles create duplicate range structures with the same limits. It is important that these ranges are located and collapsed. The fact that there are duplicate tiles in the global route for a net is not a problem, as this is how the global router represents connections. The termination process in the global router insures that all of the terminals connect, and in doing

so, sections of these paths overlap. Where these overlaps occur, corners need to be placed, but the links among the corners of the net must be carefully maintained.

The results of the corner creation, linking, and collapsing process are illustrated in Figure 32. Here the global routing tiles for a three-terminal net are presented with a representation of the local route under construction. The net will have two floating corners, linked to the terminals of the nets. These two floating corners are linked by three ranges, two of which are vertically-oriented, and one of which is horizontal. *Ranges 1* and *3* initially take on the values $[y3..y4]$ and $[y1..y2]$, but as they are linked to terminals, their ranges are fixed to these y positions. *Range 2* will take on the range $[x1..x2]$, as fixed by the vertical tile in the net's global route.

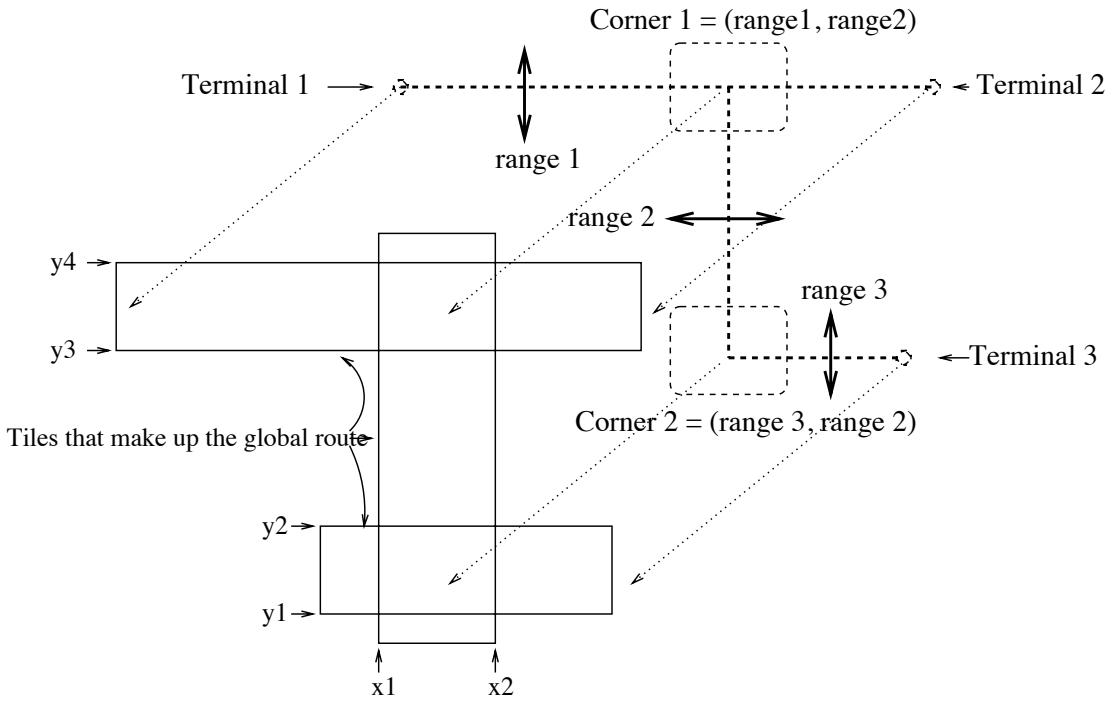


Figure 32 Mapping a global route to a set of corners.

The corners of a route are linked so that the x and y fields of the corners all point to a single range that defines their values. In this way, any restriction made to one corner will immediately be reflected in the linked corners. For example, and change to *Range 2* of Figure 32 will affect *Corner 1* and *Corner 2* equally. It is therefore important that the set of ranges used in the corners of a net not be repeated. Otherwise neither the

expansion process that adds to the ranges, nor the separation process that restricts the ranges will not work properly. All modifications that relax/restrict the initial range are communicated to all corners equally. Figure 33 shows several corners with various range/link combinations.

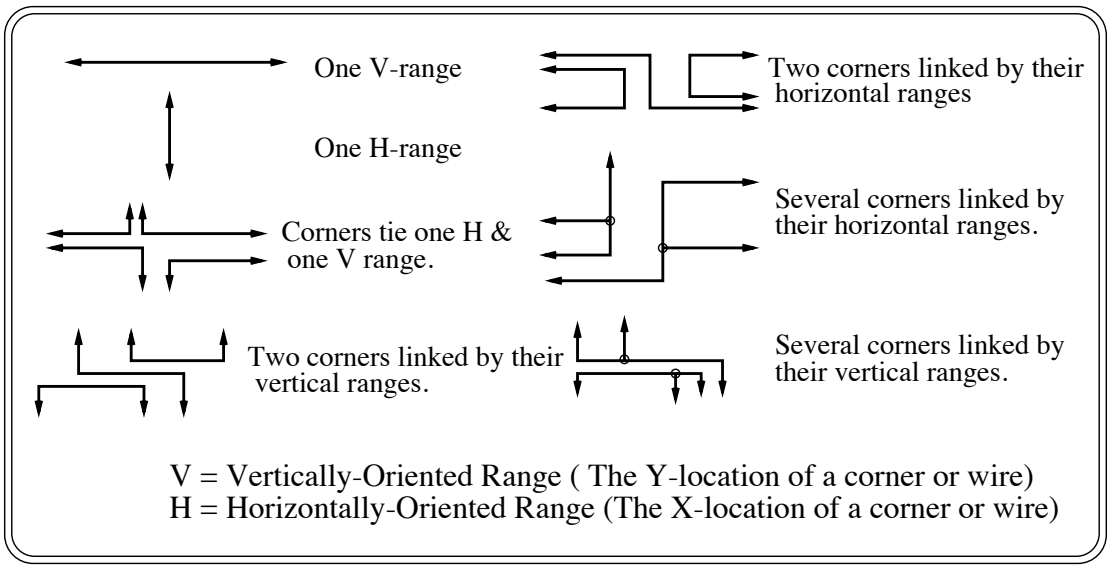


Figure 33 Examples of range/link combinations

In order to place the corners relative to each other, it is necessary to determine the widest set of legal positions that exists. This insures that the separation algorithms can utilize as much of the routing space as is practical. The need to expand the global route stems from the fact that tile spaces can often become fragmented [18], and the ranges developed from the global route represents only a minimal set of ranges in which the lines can be run. Fragmentation is both an advantage in simplifying the process of linking all the corners in a net, and a disadvantage in easily representing the full space in which a route can be run. Since tiles are designed to run as horizontally (vertically) as possible, the placement of obstacles can sliver the space, leaving a larger range in which the wire can be run, depending on the directions the bends in the routes taken and their proximity to obstacles. Thus the initial ranges developed need to be expanded into the available tiles.

This can be seen in Figure 34, which presents an example of a tile found by expansion: The smaller of the three vertical (light grey) tiles was not originally a

part of the global route for net y, but the right-most free corner of net y could be placed in it without obstruction. The x-range for this corner is expanded to include the new tile, as can be seen by the route provided. For reference, the original global route for net y is shown in Figure 30 in Section 4.3.

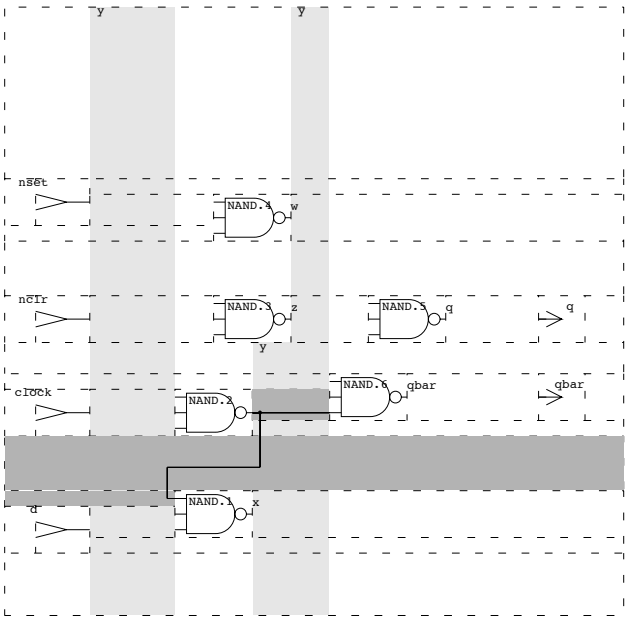


Figure 34 The tiles expanded to for net y of the SN7474.

The expansion into nearby tiles is determined by looking at the maximum area in which it can fall, based on the location of the corners to which it is attached. For a given corner$X$, the corners to which X is attached are used to find an area-of-interest. This area is searched for obstacles, and all non-obstructed tiles are added to the global route for this net. Correspondingly, these tiles are used to determine the new values that the appropriate range of corner X will take on. As new tiles are used to expand the range that a corner may be placed in, the corner is added to the tile structure for later reference.

There is usually no need to modify the corner and range structures that represent a net, as these new tiles only add to the size of the ranges that link the corners. Each net must be checked to see if the route has expanded in such a way that a particular

corner is now redundant, *i.e.* the route can be collapsed. This is not common, but a route with redundant corners is not aesthetically pleasing. This check for redundant corners completes the setup process. Each net has now been converted to a set of corners, linked by a set of ranges that define the positions where the corners are to be placed. As nets often share the same tiles in the global route, so too do many of these ranges share overlapping values. The object of the separation process is to best resolve this overlap to create a traceable set of lines.

4.4.2 Separating Overlapped Ranges

The task of the separation algorithms is to determine the final values that all ranges are to take on. Insuring that routes take on legal, traceable values implies that the ranges may not be left overlapping, and should be selected so as to place corners relative to each other in an appropriate way. This process is applied to each tile in the routing space, starting with the most congested tile, and centers on the judicious positioning of overlapped ranges, as shown in Figure 35.

The grey regions in Figure 35 show the ranges where the lines for the nets will be run in, and the hashed area between them is a small section of overlap. The goal of the separation process is to remove this overlap, which in this example would be to restrict *range3* and *range4* to unique values. The important point in choosing how to separate these ranges is how to do so in such a way as to yield the most readable results. Figure 35b shows two solutions to the overlap of *range3* and *range4*, as indicated by the solid and dashed lines. The separation that results in the solid lines being run is the more desirable of the two, since this eliminates and unnecessary crossover.

Figure 35 is a simple case of range separation, as there are only two ranges that overlap, *range3* and range4. The line segments located by these two vertical ranges overlap and need to be separated. The aesthetics of Figure 35b dictate that the solid-line solution is better. Given this desired separation, *range4*(net$B$) should be placed below *range3*)net A) indicating that both must fall within the hashed region.

The separation problem described is not a small one, in that for every corner there are two ranges to be separated, and there are typically two to three corners for *every*

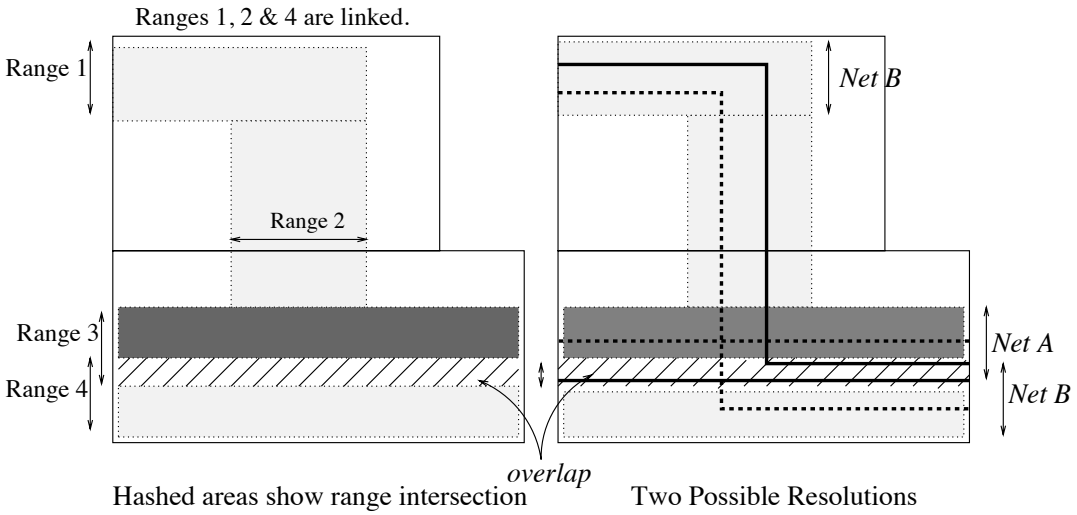


Figure 35 Overlapping ranges(a) and possible resolutions(b)

terminal in the diagram. Even in our smallest example presented, the SN7474, has 10 nets totalling to 61 corners, yielding 122 ranges that might need to be compared. A simplistic solution would be to compare all ranges to each other. This would require 61 ranges to be operated on simultaneously, for a minimum of $2 \times 61 \times 61 = 7442$ range comparisons to insure that all overlap among the ranges is removed. Rather than making this large number of comparisons, we again take advantage of the tile space data structures to localize the number of corners that must be compared.

The use of location information to solve the range overlap problem is based on the use of the tile space. Vertical ranges specify the y location of at least two corners, and thus define where a horizontal line will fall. Due to the long, horizontal nature of the tiles in the horizontal plane of the routing space, any corners containing vertical ranges that overlap will be located in the same tile. Similarly, horizontal ranges define the x location of two or more corners, and those corners that may overlap in the x dimension will be found in the same vertical tile. Thus the horizontal ranges that may overlap are found in the vertical tiles. The case holds true for vertical ranges and the horizontal tiles, which greatly simplifies the separation process. The usefulness of this is seen in the solution to the SN7474, where using the routing space to partition the separation problem reduces the largest number of ranges operated on at one time to eight.

Given the dependence of one corner position on another, there arises an ordering problem as to which corner should be fixed first. As mentioned, the most-congested tile is selected first for separation, and then the next most congested and so on, until all tiles containing corners have been separated. In this way, the most congested tiles are given preference for having the cleanest route.

The range separation operations performed on each 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. A *dependency graph* is 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. The utility of these steps can be seen in Figure 35b, as they result in the more aesthetically pleasing (solid line) routes being chosen over that the less pleasing (dotted line) ones.

Ranges within a tile are ordered by examining one corner for each range, classifying it, and then using the classification along with location and connection information to make the ordering comparisons. Corners are classified as jogs, bends, T's, or U's, depending on whether the other corners to which they are linked fall inside or outside of the tile being examined. This ordering follows the *Rules of Easy Reading* introduced in Section 2.3. These rules are summarized here for reference:

- Bends are placed in order, nearest to their exit corners
- U's are placed nearest to their exit edge, the shortest placed nearest to the edge.
- T's are centered, but nearest to their exit edge
- Jogs are centered in the tile

Figure 36 illustrates an application of these classification and ordering rules, and depicts how the tile space is used to partition the range separation problem. The example here continues with the example of the SN7474 begun in Section 4.3.

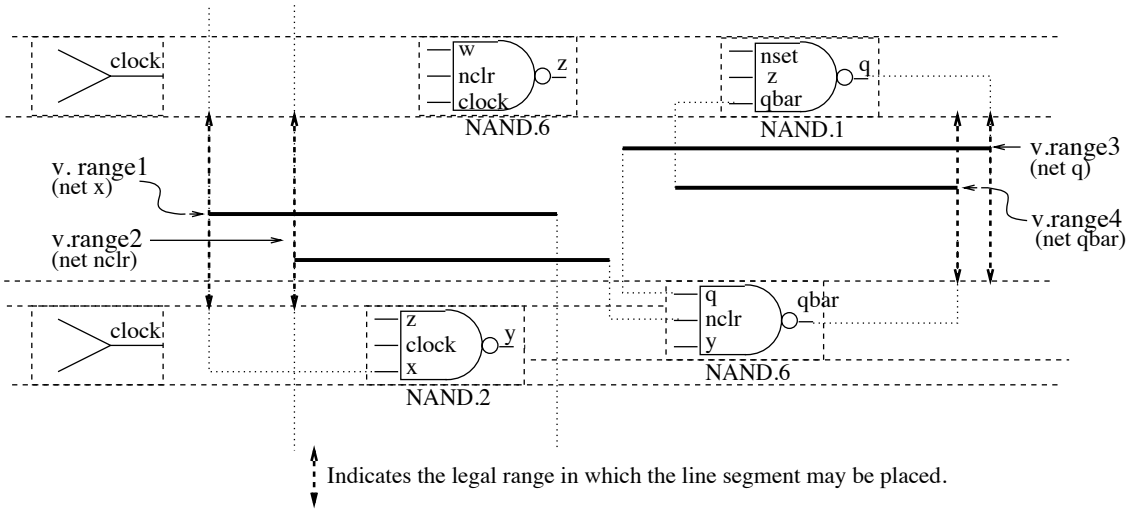


Figure 36 Separation of vertical ranges showing range ordering.

The horizontal line segments of Figure 36 illustrate the vertical ranges selected for ordering in a single horizontal tile. Each segment has a unique range determines the legal y-values that position the corners of the segments. There are four segments, contained (one for each net), and therefore four vertical ranges to be ordered. Two ranges(*v.range3* and *v.range4*) are classified as a jogs, as both corners that use the vertical range exit on opposite sides of the tile. The ranges *v.range1* and *v.range2* are classified as U's, as they have two corners that exit the tile on the same side. From this classification, and the arrangement of the corners, the range ordering is determined.

In this example, *v.range2* is placed lowest, as it is the shortest U, and exits the bottom of the tile. Similarly, *v.range1* is the next lowest, it also being classified as a U. The two jogs are more interesting, as their vertical ordering is somewhat arbitrary. As net *q* exits the tile furthest to the left, its range is placed above that of net *qbar*.

Once the given set of corners has been ordered, the dependency graphs that indicate which ranges need to be compared must be formed. For the corner-placement problem, a dependency graph is a set of ranges that must be separated.

A range z is said to be dependent on another range x ($z \Rightarrow x$) if the x should be placed before z, and the line segment associated with the x overlaps any portion of the line segment associated with z. This relationship is transitive, as if the line segment

of x overlaps y which overlaps z, then z is still dependent on x. Given that $\Rightarrow$ denotes the dependency relation described, x is said to be a parent of the graph $\{x, y, z\}$ if $z \Rightarrow y \Rightarrow x$, and there are no ranges in the tile that x depends on. Figure 37 illustrates this situation:

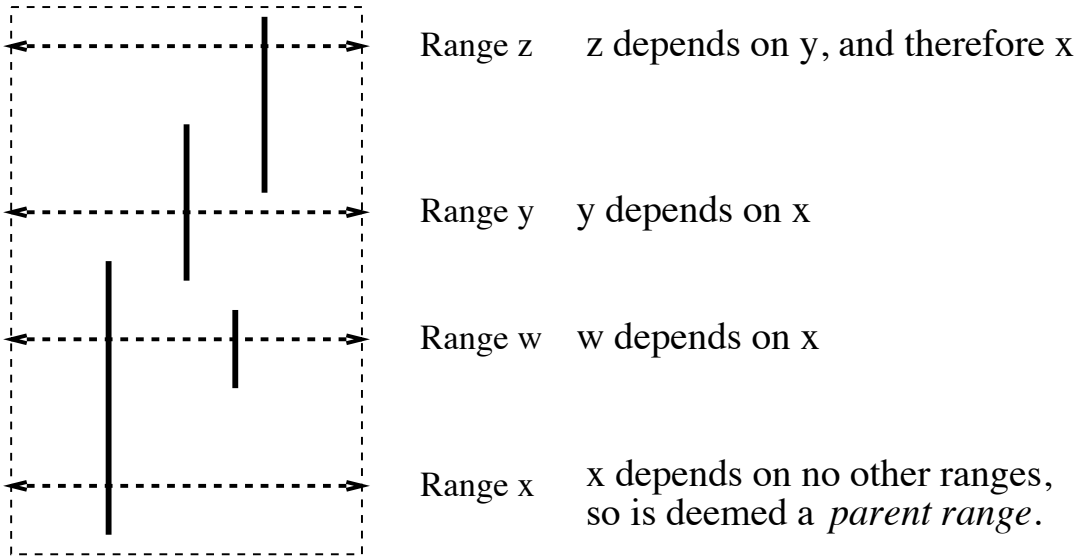


Figure 37 Examples of dependency relations among ranges

It is clearer to see the implications of a range that is *not* dependent on another. Such ranges do not need to be compared for overlap resolution, as the line segments positioned by these ranges can share the same column or row. This can be seen in Figure 37, where range$w$ is *not* dependent on either range y or z. Range w does not need to have any overlap removed from these two ranges, and thus they can have the same range values.. By paying attention to what ranges need to be separated, the use of dependency graphs in the separation process ensures that the maximum available space is allocated to each range.

The purpose of the graph is to divide the set of ranges into subsets containing groups of ranges that cannot overlap. The important implication is that all ranges that are *not* part of a particular graph may overlap with all of the ranges in the graph. This implies that ranges may occur in more than one dependency graph.

These dependency graphs are formed by applying a combination of marking, and recursive tree-traversal techniques. First, the dependence of each range on the others

is formed by comparing the length of the segment associated with each range to the segment length of all ranges that succeed it. This *dependency list* notes *all* of the ranges dependent on the range in question. The dependence list is used to find the dependency order, which lists dependent ranges by their relative ordering. This ordering is used to test each range as a parent of a set of graphs.

The set of graphs is formed by starting with a candidate parent range, and recursively traversing its dependencies in order. At each level of recursion, the next dependent range is added to a temporary list. Once a range has been added to a list with the same parent, it is marked as having been *seen*. If the lowest range is reached, and any of the ranges have not been seen before, then the temporary list is returned as a graph. Otherwise, when the bottom is reached (and *all* of the ranges listed have been seen), no list is returned, and the recursion unwinds until another branch of a dependency list can be tried. When all branches have been tried, there are no more graphs to be found.

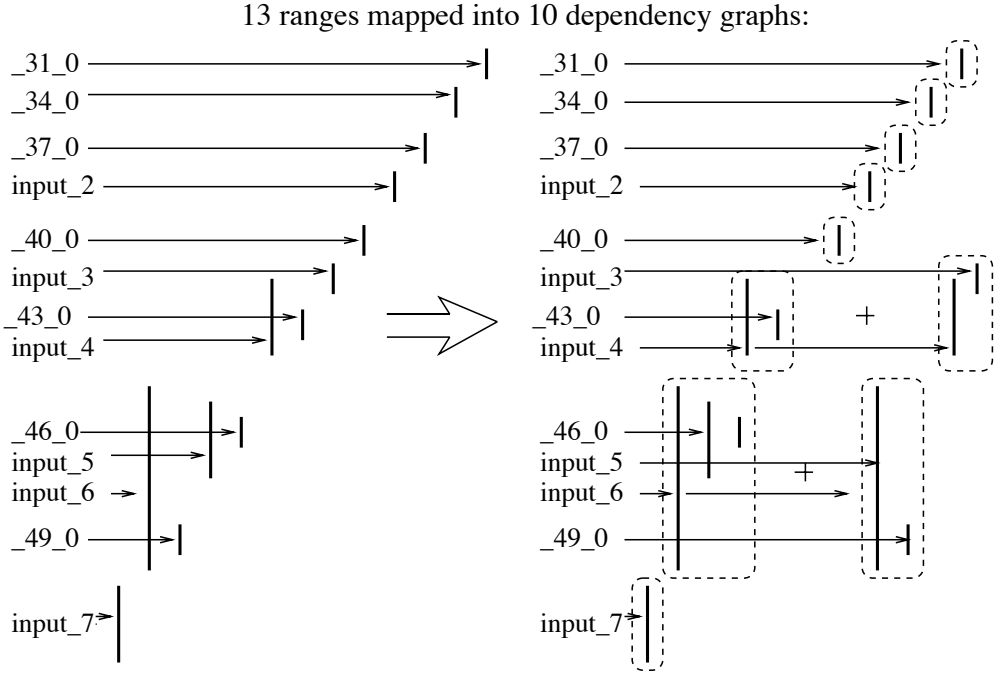


Figure 38 Dependency graph for a tile of an 8-input OR gate

Figure 38 depicts the segment lengths associated with each horizontal range found in a vertical tile of an 8-input OR gate(Figure 41). These segments are listed first

by their range order, and then in their inclusion in dependency graphs. The thirteen ranges found in the tile map (on the left of Figure 38) to 10 dependency graphs shown on the right. There are two instances where ranges appear in more than one graph: The range associated with net *input_6* is utilized twice as it has three ranges that overlap with it, but one of these (net *_49_0*) does not overlap with the other two. The range for net *input_4* is similar in that it overlaps two ranges, but neither of these overlap each other.

The last step of the separation process is to use the dependency graphs formed to remove the overlap from among the ranges being compared. This evaluation distributes the the available space among the ranges of a dependency graph so that none will overlap. The longest dependency graph in the tile is evaluated first, and then the next longest and so on, so that the any repeated ranges are adjusted to match the tightest constraints first. The order of the ranges within the dependency graph matches the ordering done in the first step of the separation process. This ordering is crucial, as the space available is distributed in this same order.

For each graph, the range that the entire graph has to fit in is determined by the minimum and maximum of the ranges at the ends of the graph. The space between these two points is divided evenly among all of the ranges of the graph, such that each receives a slot, and care is taken to insure that the values assigned to the slot are legal restrictions on the range values.

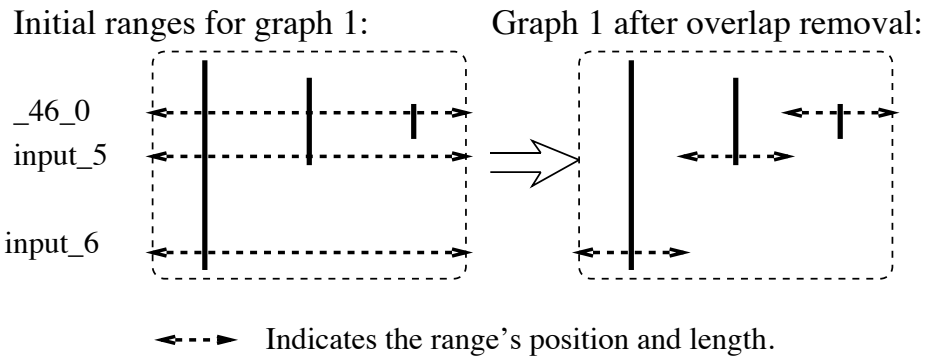


Figure 39 Position assignments for a dependency graph

The overlap removal for the first two dependency graphs depicted in Figure 38 are shown in Figure 39 and 40. First, the three ranges in *graph1* are separated, splitting

the space available into thirds, as shown in Figure 39. The range for net *input_6* is repeated in *graph2*, so this same range is retained when the overlap in *graph2* is removed. The next graph evaluated is pictured in Figure 40:

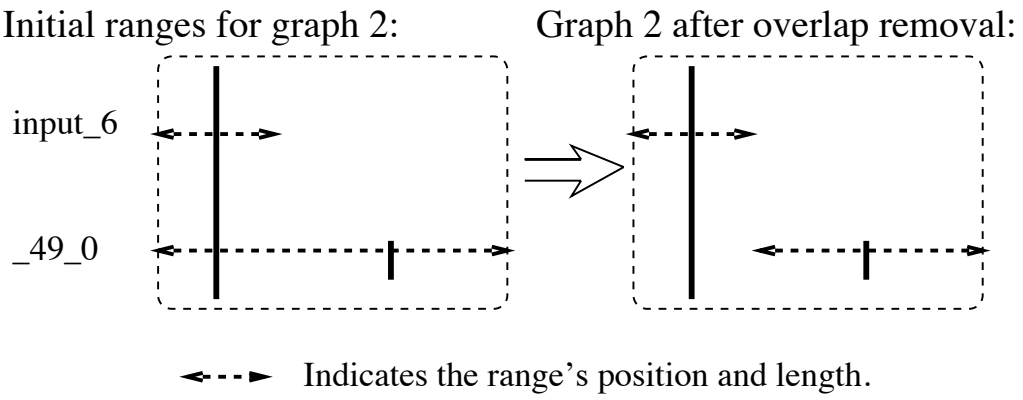


Figure 40 Position assignments for a linked dependency graph

As a result of this previous restriction, the range for net *_49_0* will take up the remaining space, as it is the only other range in *graph2*. The final positions for these nets can be seen in the completed diagram shown in Figure 41.

The separation process for an individual tile completes when all dependency graphs for the tile have had any overlap removed. The result of applying this separation process to all tiles results in a complete removal of all overlap among the ranges that are used to locate the corners of the route within the diagram. The final task remaining to the local router is to center these ranges to the middle of their available ranges. This provides the maximum spacing between lines, so as to further enhance the traceability of the diagram.

4.5 Incremental Placement

The system terminals for a schematic represent a special sub-problem for ASG systems, in that they have a different set of rules governing their placement. System terminals are typically placed in columns on each side of the page. What makes terminal placement interesting is that their placement is completely dependent on the placement of the internal portion of the Figure. In fact, it is not simply the placement

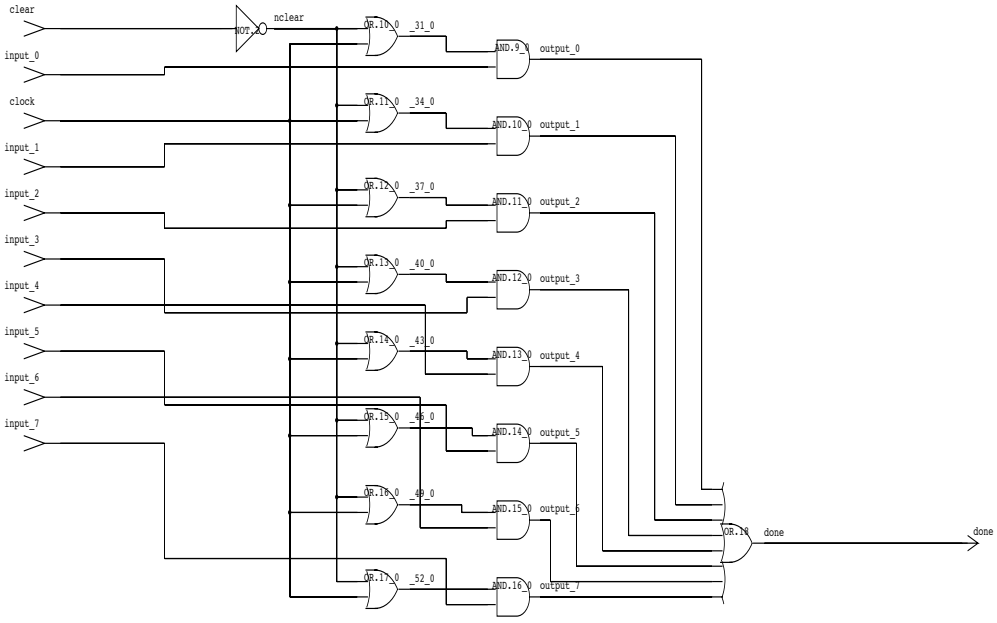


Figure 41 The completed 8-input OR Gate

of the internal modules that is important, but rather the routing of the modules. A system terminal represents either an input or an output to the schematic, and will connect to some net that has already been developed in interior of the schematic. The system terminal must connect to its net with as little congestion as possible.

Noting the dependency of system terminal locations on the interior routes, we have developed a completely incremental method of placement, where the system terminal locations are determined based on the routes for the nets to which they connect. This involves the placement and routing of the internal modules, the use of this information to guide the insertion of the system terminals into the existing routing space, followed by the completion of the routing tasks.

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

4.5.1 Use of Routing Information to Place System Terminals

Given that the central portion of a diagram has been routed, there remains the task of using this information to guide the placement of the system terminals. Since these terminals are placed in either an input column, or an output column, the x positions of the terminals is trivial. The difficulty is to place the module icons corresponding to these terminals (referred to as *systerm* modules) in such a way as to fit in the column and be traceable.

For an input systerm, traceability has most to do with getting the signal to its primary (leftmost) destination while introducing as few bends as possible to the net. This is complicated by the fact that these terminals may overlap if they are aligned directly behind their destinations, and 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 former can be seen in the placement of the input systerm *data2c* in Figure 42. Systerm *strobe1g* avoids both problems, as its destination is both blocked by the placement of gate *NOT.1*, and even were it not blocked, there would be an overlap problem with systerm *data1c*. It is interesting to note that in the placement of systerms *data2c* and *strobe2g*, it is arbitrary as to which terminal is aligned, and which is not, as there are no other factors to guide their placement.

Figure 42 also points out one of the more interesting tasks of this alignment problem, that being in how to tell that a particular destination point (or range) is *not* obstructed. This can be seen in the placement of systerm module *strobe2g*. The path to the destination terminal (one of the terminals on gate *NOR.2*) does not have to run to a point on the edge of the central portion of the diagram as do the lines connecting terminals *selectb*, *data1c*, and *selecta*. Rather, the line for *strobe2g* must pass by gate *NOT.1* to reach its destination. The fact that this is an interior module, and that the path to it is unobstructed must be discovered, as this is not known *a priori*. The completed schematic for the demultiplexer shows both of the desired charateristics for a generated schematic. It is both functionally-identifiable and traceable.

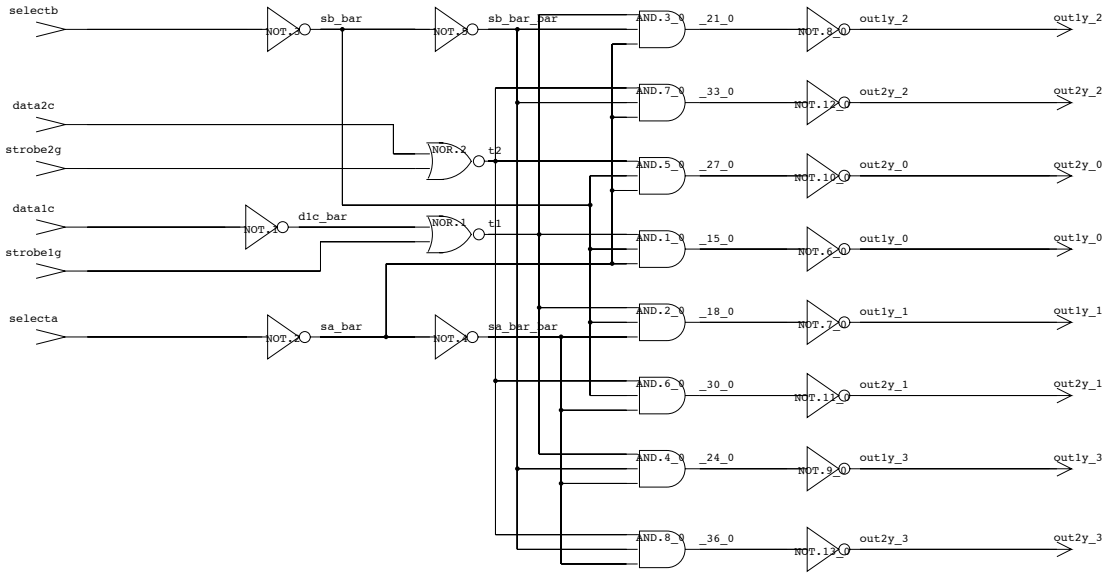


Figure 42 Completed SN54155 demultiplexer

4.5.2 Maintaining Information During Insertion

The second difficult issue for incremental placement and routing is information management. Determining where routes are to be run is expensive, so recomputation of this information should be avoided where possible. The question is how to tell when recomputation is necessary, and how to modify existing information to avoid starting from scratch.

As the insertion of system terminals is on the edge of the diagram, one can assume that their insertion will not invalidate the interior placement and routing information. Aspects of this information, such as the dimensions of tiles within paths (and therefore their relative cost) will change. These changes to the routing costs and information are made while the insertions to the routing space are being made, thus maintaining the existing path data.

This data maintenance is implemented through careful extensions to the corner-stitching algorithms [18]. These extensions modify how the contents of tiles are dealt with, and are called during the tile creation process: immediately following tile mergers, splits, deletions and creation. Specifically, whenever a tile is split, the data

contained is queried to see what paths in the old (large) tile make connections using the new tile, and which make connections using the (reduced) old tile. This process may result in the deletion or duplication of paths, but all will need to have their cost estimates updated.

Once all of the system terminals have been inserted into the routing space, the cost estimates for all paths of all nets are recalculated. The system terminals are added to the nets, so that they may be routed to the existing terminals without difficulty. The last step in schematic generation process in SPAR is to complete the routes for the nets containing system terminals. This entails creating new expansions for the system terminals, 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 routing space is intact, this is a simple process of starting the search process of the global router again. This is followed by local routing, completing the diagram.

Now that we have presented the details of the algorithms that comprise SPAR, it is appropriate to evaluate how well SPAR meets the stated goals of schematic generation: traceability and functional identification. This is the subject of the next chapter.

5.0 EXAMPLES

This chapter presents completed schematics which illustrate key aspects of the algorithms presented in Chapter 4. Performance data is then presented for all examples presented throughout this thesis.

5.1 SPAR-Generated Schematics

Figure 43a shows the completed schematic for an edge-triggered SN7474 D flip-flop. This example was used extensively to illustrate Sections 4.1 and 4.3. This example centers on six highly-interconnected modules that show two levels of cross-coupling, as well as simple cross-coupling. The SN7474 has three sets of cross-coupled NAND gates, two sets of which both feed the third set. The routing is also tricky, as there are five nets that share the same column (nets *nset, nclr, clock, d* and x), and due to the placement, potentially a sixth (net w). None the less, all of these nets are easily traced. The figure from the TTL Data Book [21] for this part is shown in Figure 43b, so that a hand-drawn result might be viewed for reference.

The placement very closely matches that of the hand-drawn figure, and the routing, although not as efficient in its use of column space, still yields a traceable result. The hand-drawn schematic also uses 45° lines to connect cross-coupled gates, a feature that aides in the recognition of cross-coupled modules, but not currently employed in SPAR.

In Figure 44, a simple eight-bit shift register is presented to illustrate the ability of SPAR to handle non-standard icons. The eight flip-flops have multiple outputs, and a common clock entering from the top of the flip-flop icons. This is a simple, functionally-identifiable, traceable schematic. We note that the difference in spacing among the flip-flops of Figure 44 is due to the fact that not all flip-flops are placed in the same partition. Intra- and inter-partition spacings are variable.

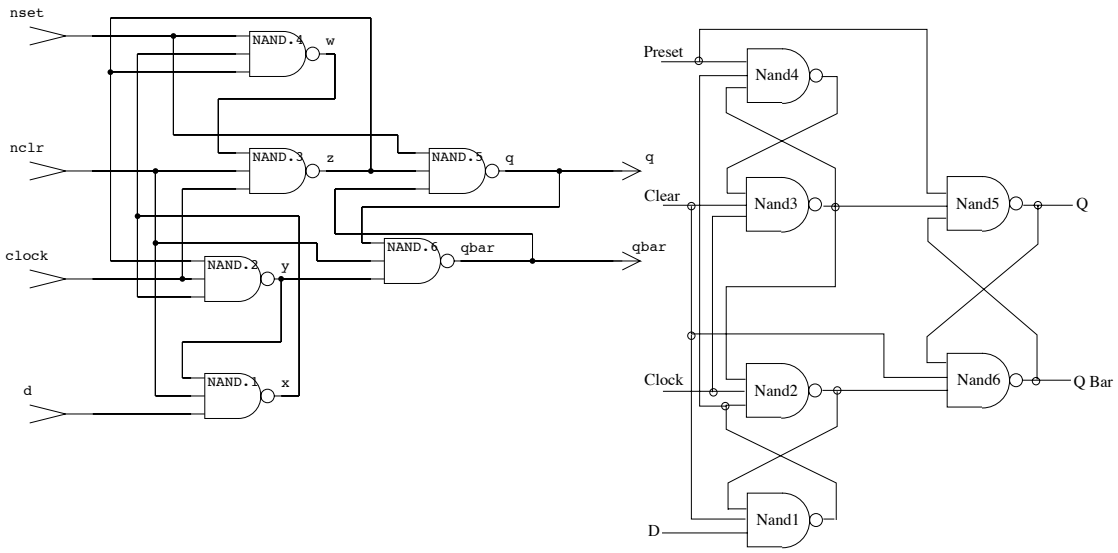


Figure 43 Completed SN7474 (a) versus TTL Data Book(b)

This figure also illustrates that our placement techniques are effective in dealing with icons which use common inputs. Another, larger example of this can be seen in the placement of a four-bit multiplexer, the SN54S151. The schematic for this is shown in Figure 45. This diagram shows how several modules are used as common inputs to the eight central modules of the figure. The placement for this multiplexer is good, as the layout and the route of the eight input signals clearly shows their being multiplexed into a single output pair. This is a clear example of functional identification in the generation of a schematic. The five-input AND modules central to the schematic are arranged in a column, with sufficient space provided for the routing channel that separates these modules from the inverter modules behind them.

The routing in Figure 45 was done without the use of the congestion metrics. Without the use of congestion metrics, the global router created a single routing channel in which to run most of the lines to the eight AND modules. This illustrates the ability of SPAR to create a congested routing channel as are common to many

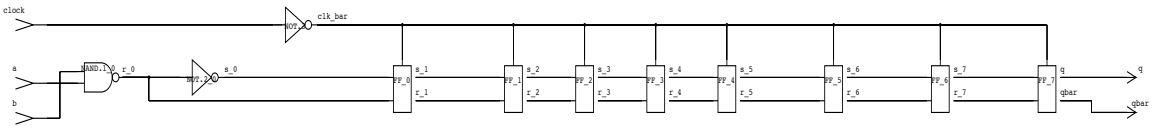


Figure 44 8-bit Shift Register

generated schematics.

The congestion metrics employed in SPAR's global router are designed to create routes that avoid the overuse of individual channels. The effects of employing rip-up and reroute on the multiplexer are shown in Figure 46. This second version of the SN54S151 illustrates the effects of congestion–based routing when applied to a congested channels. The nets that can be pulled away from the channel are, and are arranged outside where possible, leading to the branching effects. This can be seen in the placement of nets *a_bar, b_bar and c_bar.*

The branching effects shown in Figure 46 may not be entirely desirable, as most designers are often accustomed to tracing nets in a congested column. For this reason, the use of congestion routing is an option, and routing can be optionally terminated after completing only the first pass at global routing.

The multiplexer example does not illustrate the true effectiveness of congestion-based routing, as the route with the congested channel may be the more desirable of the two, depending on the needs of the designer, as multiplexers are commonly drawn with a congested channel behind the column of AND modules. The true usefulness of the congestion metrics can be seen more clearly by examining the two versions of a 16–function generator, as shown in Figures 47 and 48.

The results of applying the congestion metric to the function generator are shown in Figure 48, and clearly show how the overall density of the corners is reduced, when

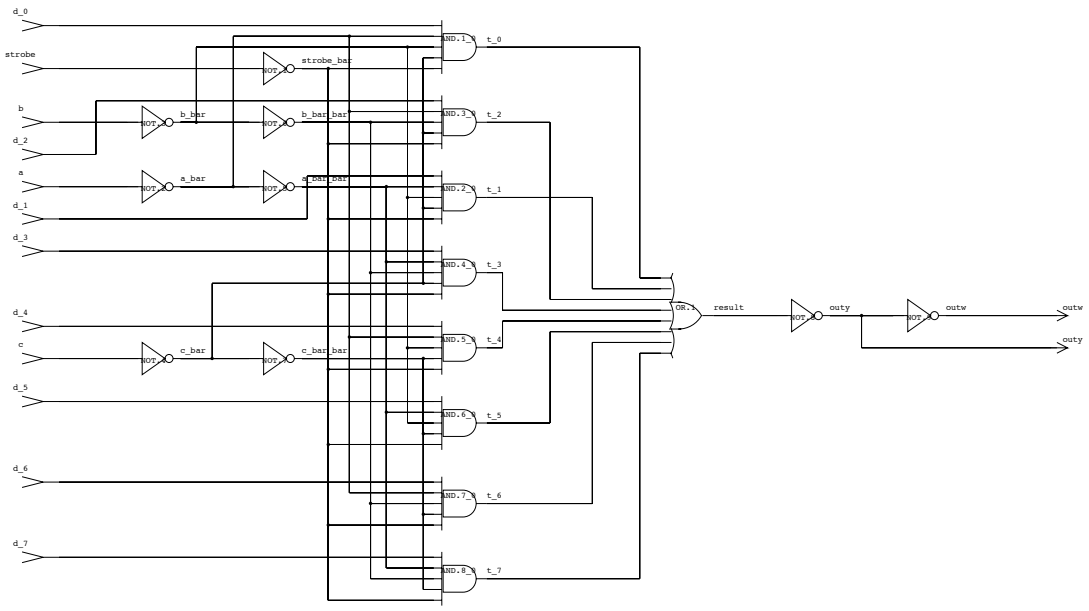


Figure 45 Completed SN54S151 multiplexer without congestion considerations

compared to the schematic without congestion (Figure 47). Figure 48 is easier to trace, and shows how the router tends to move routes away from congested channels.

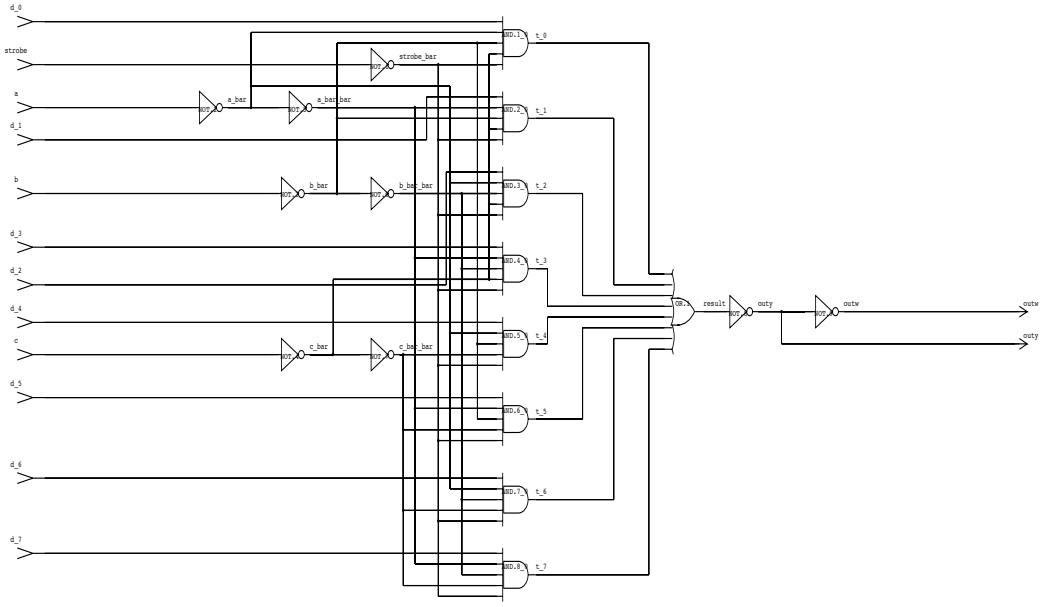


Figure 46 SN54S151 multiplexer, making congestion considerations

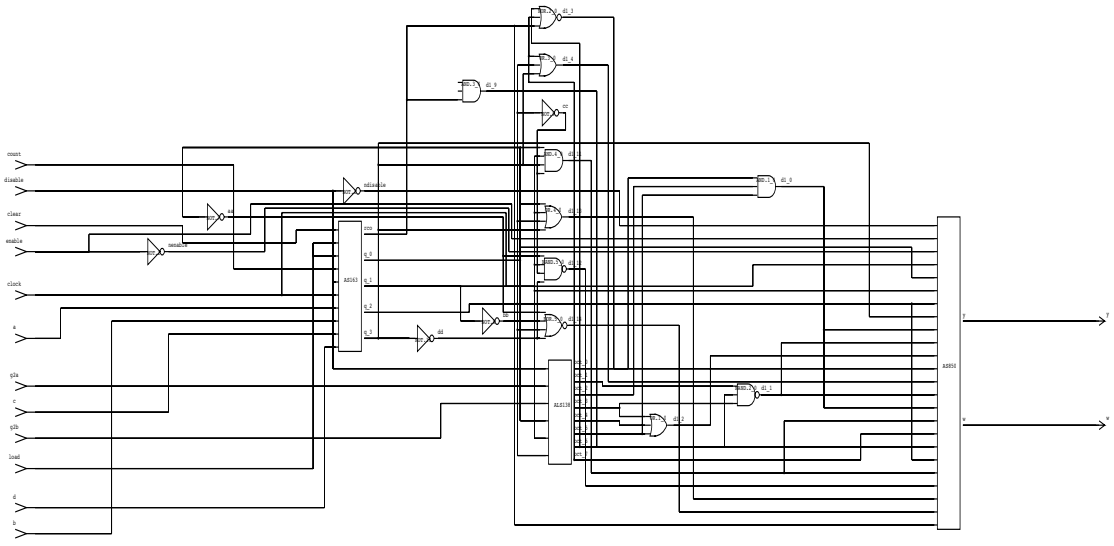


Figure 47 Function generator, without congestion considerations

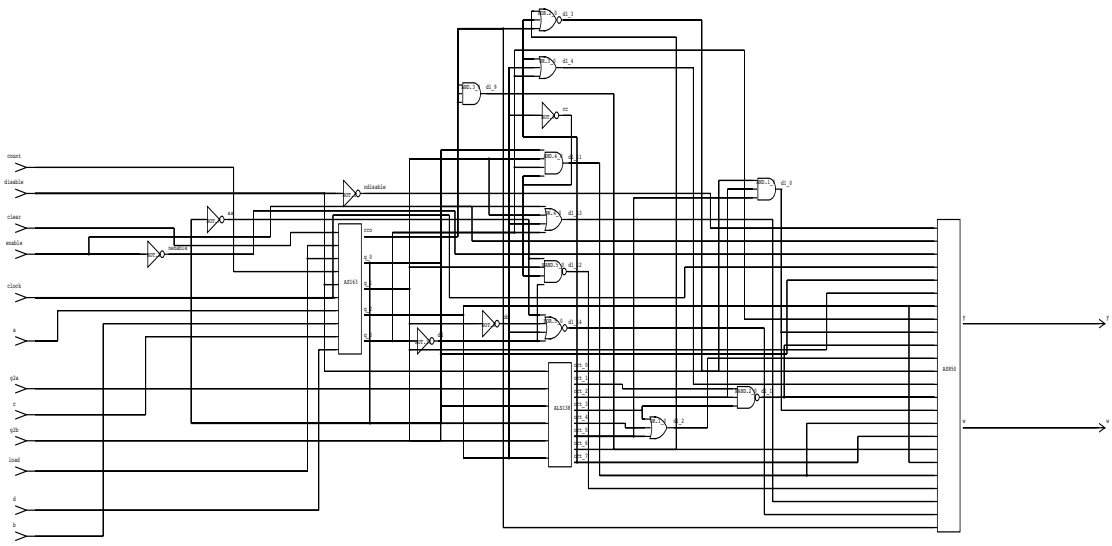


Figure 48 Function generator, making congestion considerations

5.2 Run-Time estimates

To give a rough estimate of the usefulness of this system for a designer, it is important to get the magnitude of the time needed to perform schematics generation using SPAR. The "wall-clock" time to produce the figures in this thesis are listed in table 1:

Name	Modules	Nets	Terminals	Runtime	Figure
FnGen with ripup	33	60	125	24 sec	47
FnGen without ripup	33	60	125	16 sec	48
Ripple-carry adder	39	37	92	7 sec	26
SN54S151 with ripup	32	33	89	3 sec	45
SN54S151 without ripup	32	33	89	8 sec	46
SN54155	37	32	78	6 sec	42
8-input OR gate	29	31	70	7 sec	41
SN5491A	16	25	52	1 sec	44
Complex String	19	20	37	2 sec	21
D-latch	14	15	32	2 sec	6
SN7474	12	13	30	2 sec	43

Table 1 Complexity and runtimes for examples presented

The number of modules, nets and terminals for each example are listed to give an indication of their relative complexity. The figures are listed in order of their terminal complexity, although this is only a rough measure of figure complexity. The SN5491A shift register is an excellent example, as it has 52 terminals, but due to its very regular composure, runs in less time than figures with fewer terminals. The times listed in table 1 are wall clock seconds, run on an unloaded Sun Sparc Station II.

Given these basic runtimes, it is useful to examine where time is spent in creating the schematics. Table 2 depicts the runtimes of the various tasks of the program, broken down into the categories of placement (which includes the parsing of the netlist), global routing, local routing, and the insertion and routing for the system terminals. As many of these times were smaller than the time increment available to

measure, they are marked with as "< 1" to indicate that they took some time less than one second.

Name	Placement	Global Route	Local Route	Systerm Insertion	Total	Figure Number
FnGen w/ ripup	1	10	< 1	13	24	47
FnGen w/out ripup	1	7	< 1	8	16	48
SN54S151 w/ ripup	< 1	3	< 1	5	8	46
SN54S151 w/out ripup	< 1	1	< 1	2	3	45
Ripple-carry adder	1	2	< 1	4	7	26
8-input OR gate	1	2	< 1	4	7	41
SN54155	1	2	< 1	3	6	42
SN7474	< 1	1	< 1	1	2	43
Complex String	< 1	1	< 1	1	2	21
D-latch	< 1	~ 1	< 1	~ 1	2	6
SN5491A	< 1	< 1	< 1	< 1	1	44

Table 2 Runtimes for SPAR tasks

What table 2 points out is that the time–consuming operations in SPAR are the discovery of the global route, and the incremental place-and-route techniques for the insertion of system terminals. Also, the increased cost of ripping up and rerouting the lines for congestion considerations is indicated, especially for the Function Generator (FnGen) and the SN54S151 Multiplexer.

5.3 Placement Quality

The placement of the modules in a schematic is critically important to the quality of the resulting diagram. We acknowledge this importance by employing three different (size-based, Rent's-rule-based, and slope-based) partitioning methods for the placement of modules. These methods were described in Section 4.1.

To illustrate the effectiveness of these methods, we compare several schematics generated from a single design, a four-bit, ripple-carry adder. The Ripple-Carry

Adder has a specific structure to be discovered from the connections among the modules. There are four essential structures, one for each bit of the addition. Based on the criteria of functional identification, the success of the partitioning and placement is judged by how identifiable these structures are. Unfortunately, identification of structure is subjective, and difficult to quantify for use in comparing the relative quality ('goodness') of generated schematics.

We have developed an approximate method for comparing the relative 'goodness' of schematics based on the observation that that well-structure schematics tend to have signals flowing from left to right and (though not as strongly) from top to bottom. This can be measured roughly by observing that input terminals should appear to the right of, and below their source terminals. There are exceptions to this rule, most notably in the case of gates that have feedback. We define two the terms *xfoul* and *yfoul* to describe failures to follow basic signal inheritance notions.

We define an xfoul to be whenever an input terminal is placed to the left of its source (output) terminal. A yfoul is observed whenever an input terminal is placed above its source terminal. Neither foul is counted if the modules containing the two terminals in question are cross-coupled. Table 3 presents a summary of these fouls to the schematics generated by applying the three different partitioning techniques to a 4-bit Ripple-Carry Adder. Diagrams showing the partitions formed and the relative placement for each example are listed in Appendix A.

Table 3 lists the goodness results for the application of all three partitioning styles (size-based, Rent's-rule, and slope-based) to the four-bit RC adder. The qualifiers used for partitioning are listed with a count of the xfouls, yfouls, the dimensions of the diagram, and the area of the finished diagram. The qualifier s denotes the size (in numbers of modules) used for sized-based partitioning, and the qualifier r denotes the ratio (in numbers of connections per-module contained) used in the Rent's-rule partitioning. Slope-based partitioning requires no arguments.

The use of these fouls is only a rough measure of how functionally identifiable the resulting diagram is. In fact, it is easier to categorically state that a diagram having a large foul count (especially xfouls) is *un*recognizable, than it is to say that having

Partition Rule	Partition Qualifier	Xfouls	Yfouls	Total Fouls	Dimensions	Total Area	Figure Number
Size-based	s = 2	10	3	13	256×141	36096	49
Size-based	s = 4	9	4	13	296×99	29304	50
Size-based	s = 6	11	6	17	270×106	28620	51
Size-based	s = 8	0	2	2	376×112	42112	52
Size-based	s = 12	3	4	7	377×106	39962	53
Size-based	s = 15	0	2	2	387×92	35604	54
Size-based	s = 20	0	2	2	348×118	41064	55
Size-based	s = 30	0	4	4	349×124	43276	56
Rent's rule	r = 1.5	6	6	12	362×98	35476	57
Rent's rule	r = 2.0	0	5	5	455×104	47320	58
Rent's rule	r = 2.5	0	6	6	441×114	50274	59
Rent's rule	r = 3.0	6	4	10	344×122	41968	60
Rent's rule	r = 3.5	2	5	7	398×106	42188	61
Slope-based	–	1	6	7	398×100	39800	62

Table 3 Effectiveness of partitioning and placement for a 4-bit RC Adder

a low number of these fouls indicates that the generated diagram is functionally-recognizable.

The most readable results are those that have no xfouls. Six of the fourteen sample runs presented show no xfouls, and by examining the schematics produced (Figures 52, 54, 55, 56, 58, and 59), they all clearly show identifiable characteristics for the sequenced bit operations of the adder.

Table 3 also shows the dimensions and total area for the schematics, which are presented for reference. These dimensions are useful for comparison, but do not give a clear indication as to which schematic is better. Our observations are that the smallest schematic is not always the best schematic.

The partitioning rules and goodness results for all of the major examples presented in this thesis are listed in table 4. From the results in this table, we can see that SPAR did an effective job on the examples presented, as only three designes listed have any

xfouls. Of the three with errors, the Complex String example and the SN7474 are of particular interest in that their schematics have xfouls, but are very well placed. This is due to the fact that both examples have rather complicated cross-coupling among their modules.

Name	Partition Rule	Partition Qualifier	Xfouls	Yfouls	Total Fouls	Figure Number
Function Gen	size-based	s = 5	6	6	12	48
4bit-adder0	size-based	s = 8	0	2	2	26
SN54S151	slope-based	–	0	5	5	46
SN54155	slope-based	–	0	0	0	42
8-input OR	slope-based	–	0	0	0	41
SN5491A	slope-based	–	0	1	1	44
Complex String	size-based	s = 24	2	0	2	21
D-Latch	size-based	s = 4	0	1	1	6
SN7474	slope-based	–	3	0	3	43

Table 4 Effectiveness of partitioning and placement for thesis examples

The examples presented in this chapter show that SPAR does a very good job for placing and routing small schematics. The most notable failure is in the complicated Function Generator example of Figures 47 and 48. We discuss the successes and failures of SPAR in the next chapter.

6.0 CONCLUSIONS AND FUTURE WORK

6.1 Summary

In this thesis, we have presented a new set of algorithms that generate general schematic diagrams from circuit netlists, and have demonstrated how these are created to be useful to the circuit designer for both functional identification and traceability. The approach is novel in that its emphasis lies in manipulating the diagram's space to meet the aesthetic requirements, rather than the use of fixed columns and rows. 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 results presented in this thesis demonstrate that an algorithmic approach to ASG is both feasible, and can yield good results, 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. As SPAR has been developed as a testing platform for the generation algorithms described, the emphasis has not been on the generation of large, industrial schematics, and the system would require work to be applicable in such a setting.

The most notable results of this work have been in two areas: the two-dimensional space-management techniques employed throughout SPAR, and the constraint–propagation techniques used in the local router. There are other area-phenomena to which the space-management techniques could be applied, as well as further developments of constraint-propagation to be explored.

SPAR's weakest point is that the partitioning scheme is not effective for the discovery of functionally-identifiable schematics where the schematics show depth in their structure. The combination of partitioning and placement are very effective for

small examples, but they are not effective for the discovery of combined or deeply-embedded structures. For example, combining the multiplexer and demultiplexers of Figures 46 and 42 into one schematic does not yield the most desirable results. This can be seen in the examples in Appendix B, which illustrate the *best* results that SPAR could produce for the combined schematic. There are several ways to improve on these weaknesses, most notably in the development of a new partitioning scheme.

6.2 Future Work

There are numerous areas where SPAR could be improved. The four most notable areas are in the partitioning methods employed, extensions to the incremental algorithms, extensions to the constraint propagation (delayed evaluation) techniques, and the addition of specialized routing features that have not been employed.

6.2.1 Improved Partitioning

As mentioned, SPAR's partitioning methods are not very effective at handling compound diagrams, such as those depicted in Appendix B. New methods for partitioning the diagram should be employed that can discover such deeply embedded hierarchy to enlarge the class of problems to which SPAR can be used. To accompany this, a facility to have SPAR automatically choose the best partitioning scheme would be very useful, so that the user would not need to try several methods, if the initial results are inadequate.

6.2.2 Incremental Placement

The incremental placement and routing scheme presented should be extended, so that merger of virtual pages of modules could be driven by the routing information provided. By dividing the larger problem into these subproblems, not only the placement but the routing can be determined at the local level. This routing information can then be used to determine how modules can be best added to the growing schematic. This is applicable in two instances: at the lowest level, such as the placement of two cross-coupled modules (Figure 1b), and at a broader level in the

placement of two independent structures(Figure 65). Cross-coupled gates should be routed specially, so that they can be identified, and more complex structures should be formed separately so their independent functions can be identified.

The difficulties with this in-depth form of incremental placement and routing include the efficient transfer of routing information between virtual pages, and the details of route maintenance given that modules can be freely inserted, possibly blocking existing routes.

6.2.3 Delayed Evaluation

Another area for future development is to allow routing decisions to affect the final placement of modules within the diagram. The current system is very inflexible for providing routing space. That is, if the placement routine does not provide ample space, then the what space is provided will become congested, forcing the router to use other tiles. This creates jogs that would otherwise be unnecessary. This is one of the greater advantages that a KBS approach has over an algorithmic approach, as implemented in SPAR. This can be seen in some of the more circuitous routes of Figures 47 and 48.

This spacing problem could be solved by extending the delayed evaluation of corner positions to that of module positions. Delayed evaluation is when the dependencies of one position on another are recorded, and then all of the dependencies are managed to determine an overall optimal solution. This would provide a much more direct link between the placement and the routing of modules, as the placement would provide an initial placement along with limits within which adjustments could be made. Such a notion would introduce another ordering problem, in that the order in which module positions (and thus routing positions) would be fixed would need to be determined.

The effectiveness of delayed evaluation and a solution to a similar ordering problem is presented in the constraint-propagation technique use in SPAR's local router. A set of restrictions (ranges) within which each corner must fall is determined, then these restrictions are compared to produce and optimal crossover solution. The ranges are then evaluated, and reduced to the central point of the range. This same process

could be applied to module positioning, so that the most central module(s) would be fixed, and then the next most connected modules based on the routes that surround them. This would be particularly effective if the dependencies of one module/route on another are tracked, therefore a change in one module or route would propagate to all related modules and routes without affecting the decisions already made.

6.2.4 Specialized Routing Features

There are several other features that should be added to the routing techniques described. These are the addition of bus structures, the use of 45° lines, and the reordering of pins within modules. Busses are a common feature on schematics that are not addressed in SPAR. This addition would extend the usefulness of this generator to schematics that use large bus-structures, as are common in most RT-level schematics. The use of 45° lines would greatly aid in the recognition of cross-coupled flip-flops. Pin reordering should be addressed because most standard gate-level icons, such as many of the icons presented (NAND, NOR, etc.), have no ordering to the pins of the modules. The ordering of such pins should be dictated by the routing of the figure. Such a feature would greatly enhance the routing produced, as the present corner ordering is dependent on the pin, which at present is arbitrary.

APPENDIX A

APPENDIX A

The following fourteen diagrams show the various placements for the four-bit RC Adder as listed in table 3 in Section 5.3. These diagrams show different degrees of success in the creation of functionally-identifiable schematics for the adder. Most notable are Figures 52, 54, 55, 56, 58, and 59, which all exhibit the bit-structure inherent to the RC Adder.

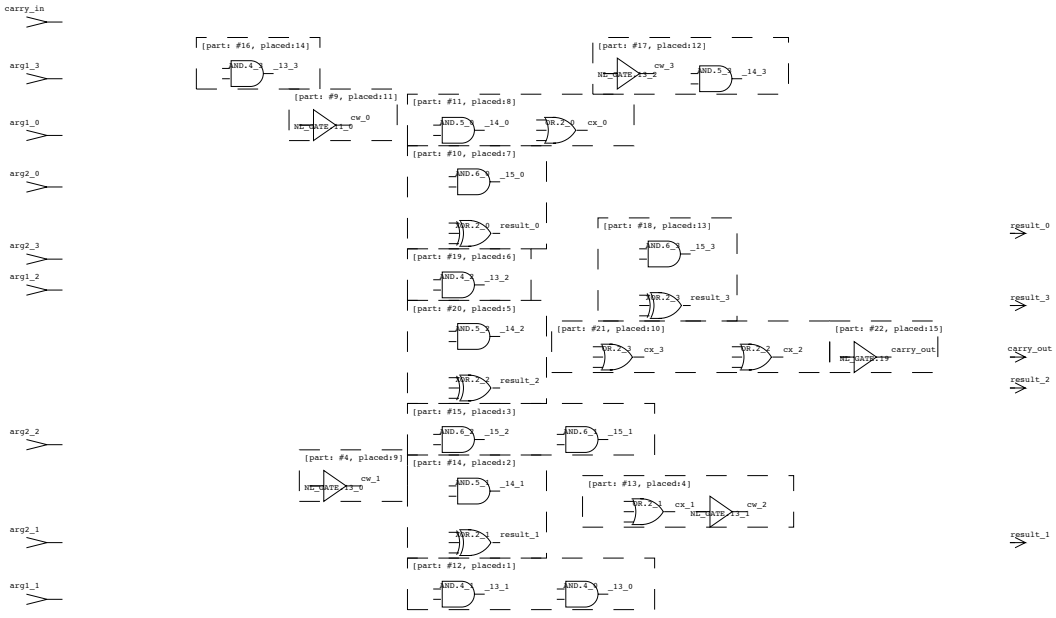


Figure 49 Size-based partitioning, size = 2, xfouls = 10, yfouls = 3

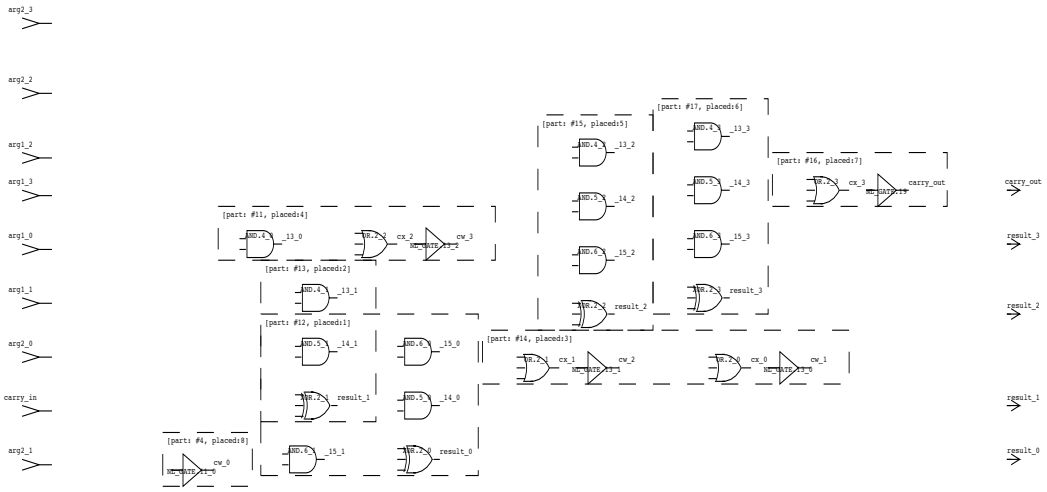


Figure 50 Size-based partitioning, size = 4, xfouls = 9, yfouls = 4

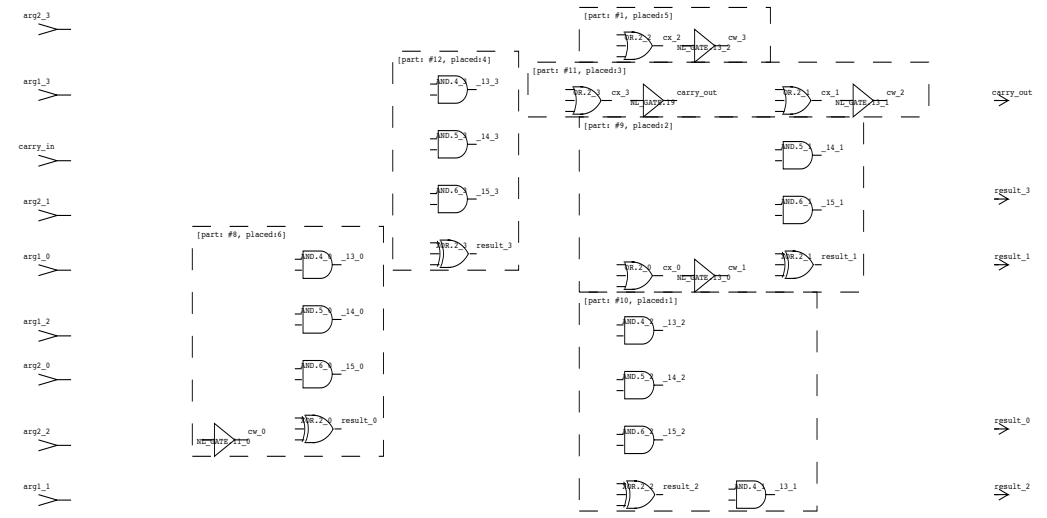


Figure 51 Size-based partitioning, size = 6, xfouls = 11, yfouls = 6

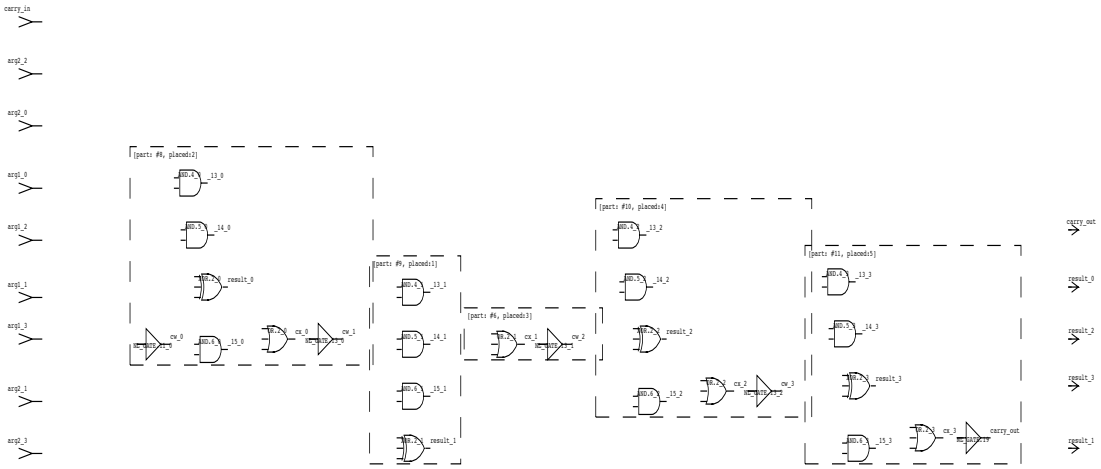


Figure 52 Size-based partitioning, size = 8, xfouls = 0, yfouls = 2

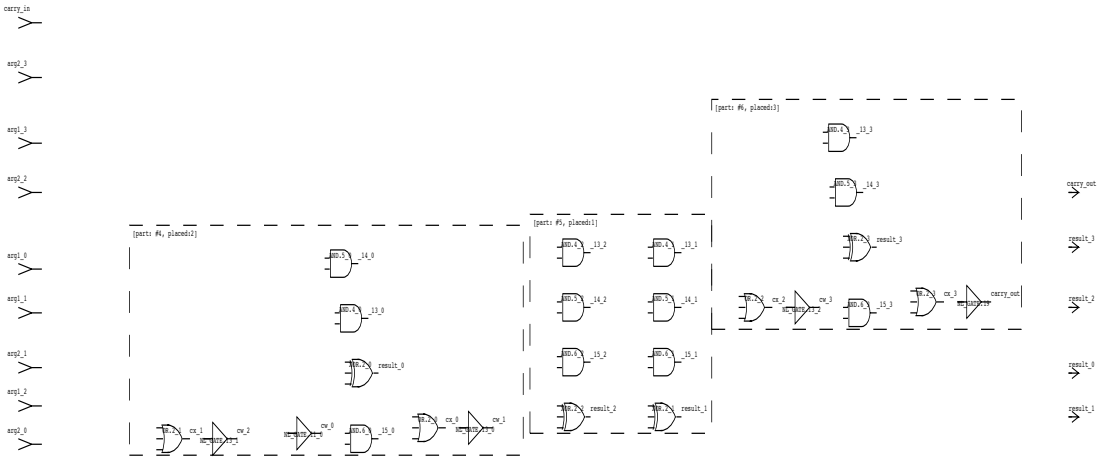


Figure 53 Size-based partitioning, size = 12, xfouls = 3, yfouls = 4

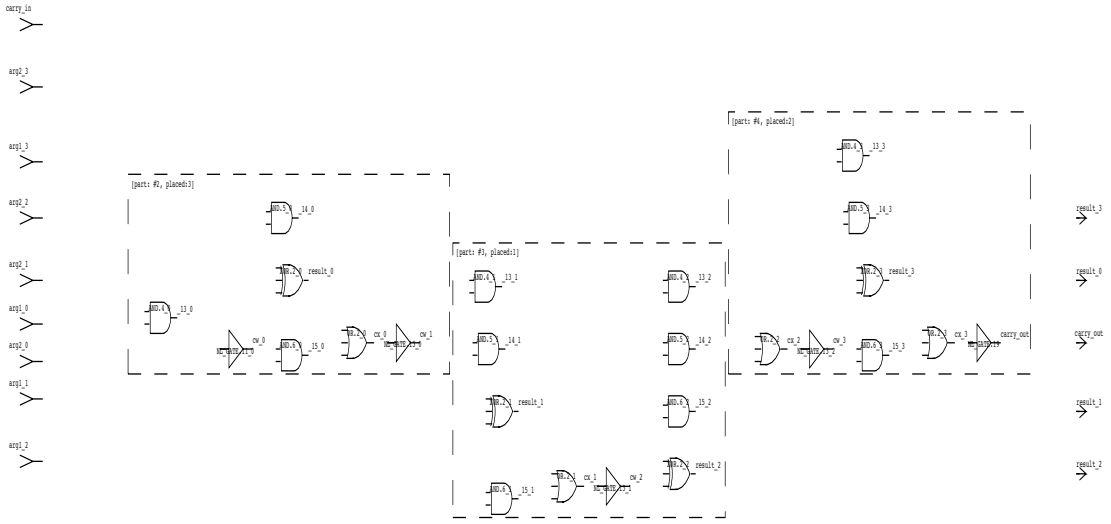


Figure 54 Size-based partitioning, size = 15, xfouls = 0, yfouls = 2

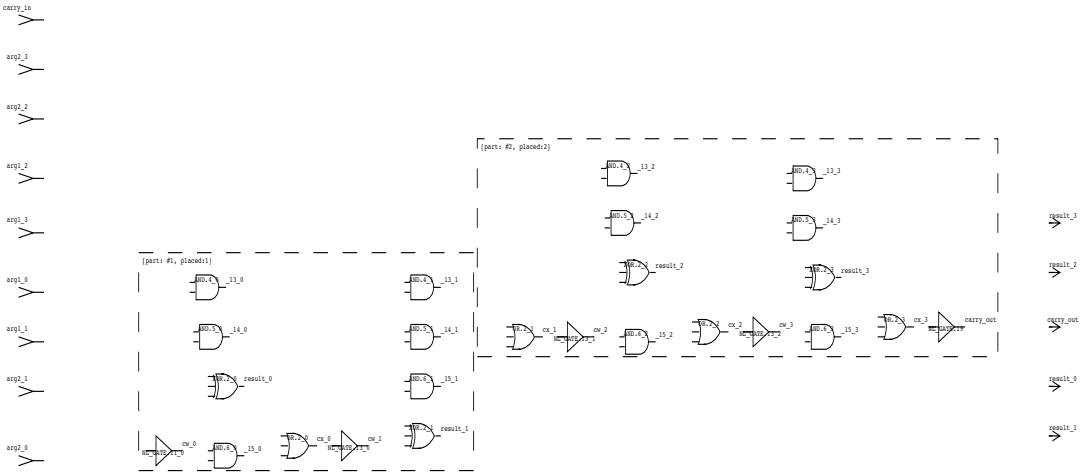


Figure 55 Size-based partitioning, size = 20, xfouls = 0, yfouls = 2

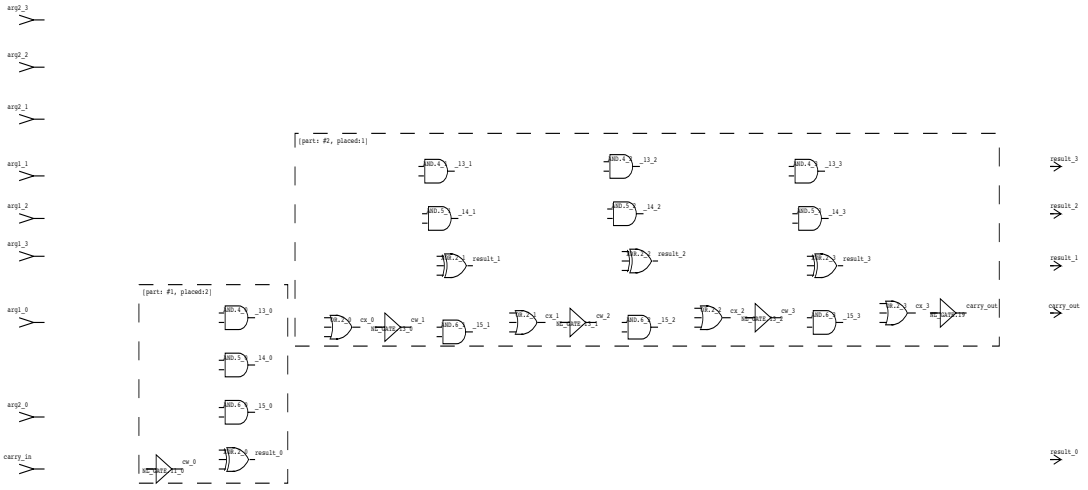


Figure 56 Size-based partitioning, size = 30, xfouls = 0, yfouls = 4

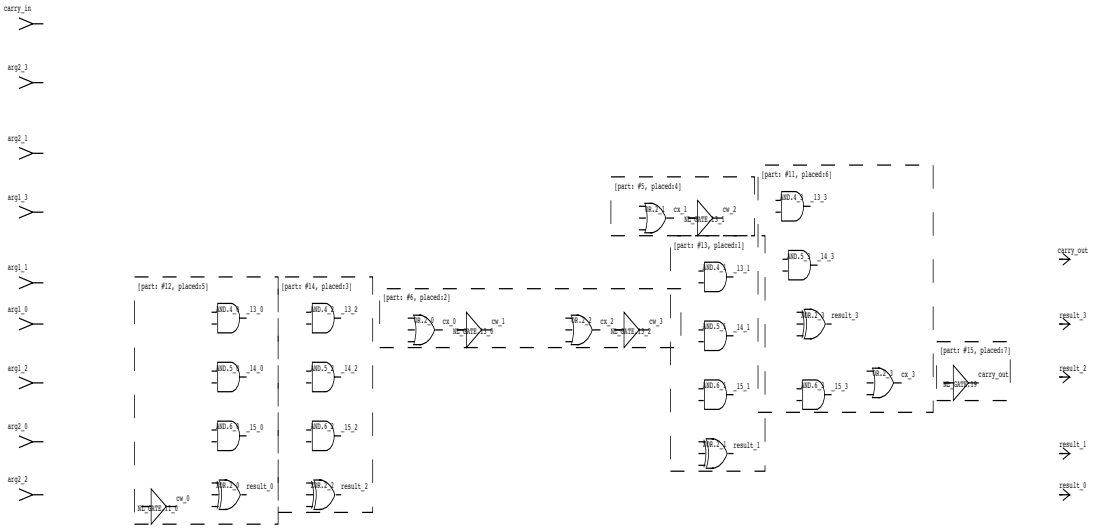


Figure 57 Rent's Rule-based partitioning, ratio = 1.5, xfouls = 6, yfouls = 6

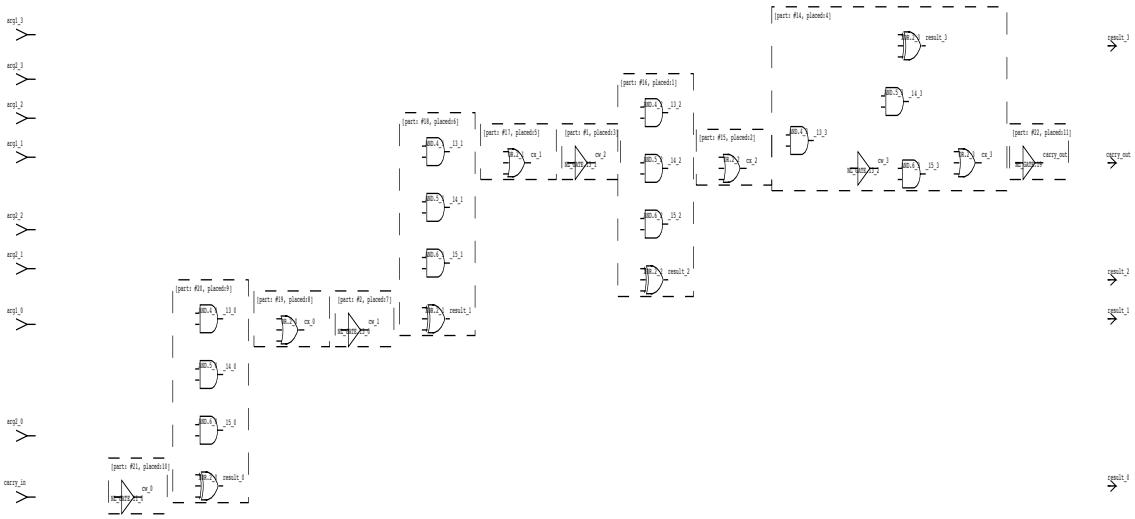


Figure 58 Rent's Rule-based partitioning, ratio = 2.0, xfouls = 0, yfouls = 5

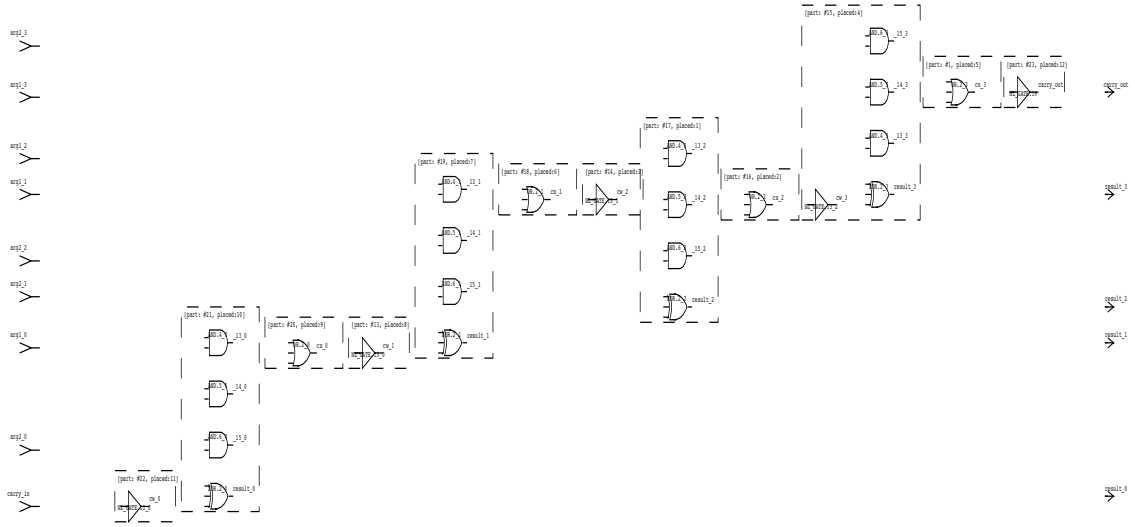


Figure 59 Rent's Rule-based partitioning, ratio = 2.5, xfouls = 0, yfouls = 6

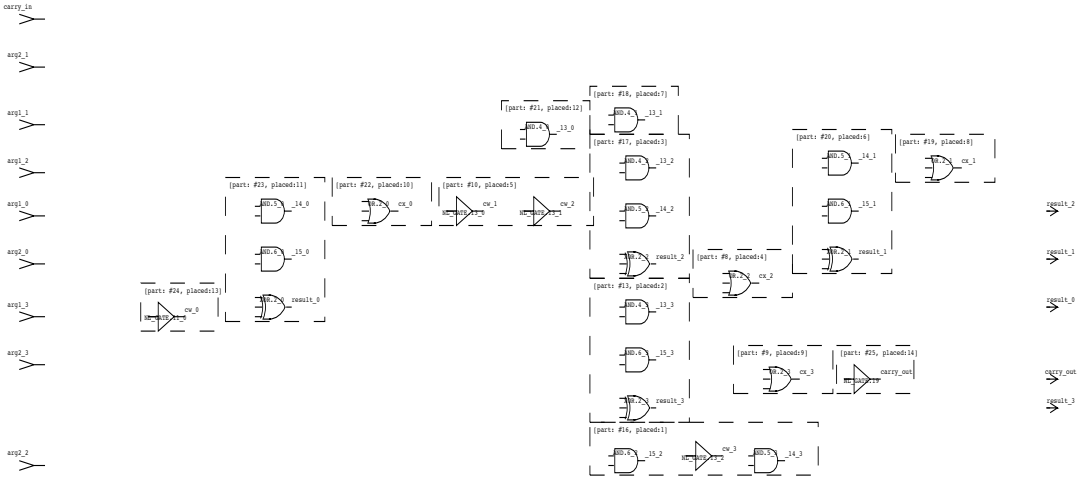


Figure 60 Rent's Rule-based partitioning, ratio = 3.0, xfouls = 6, yfouls = 4

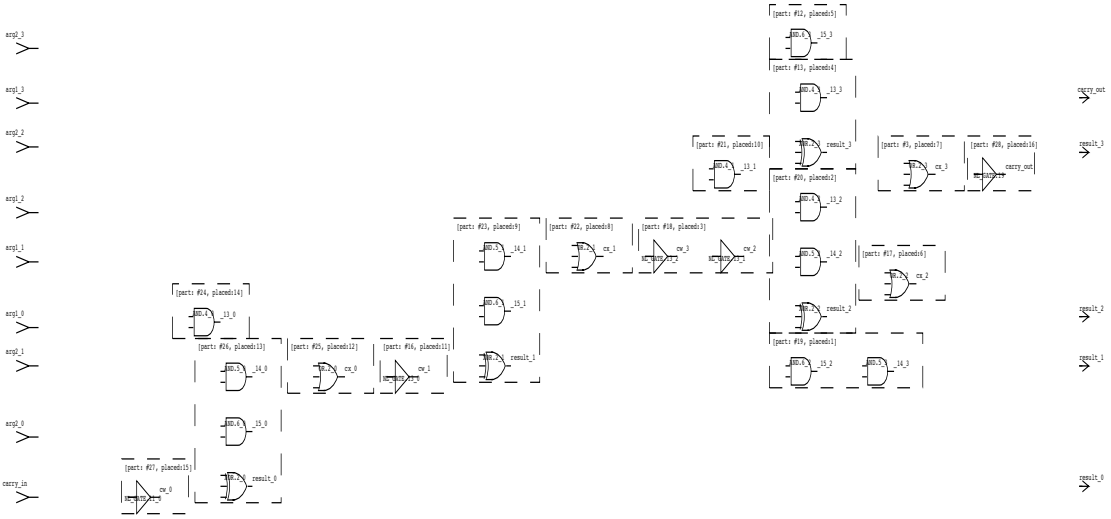


Figure 61 Rent's Rule-based partitioning, ratio = 3.5, xfouls = 6, yfouls = 4

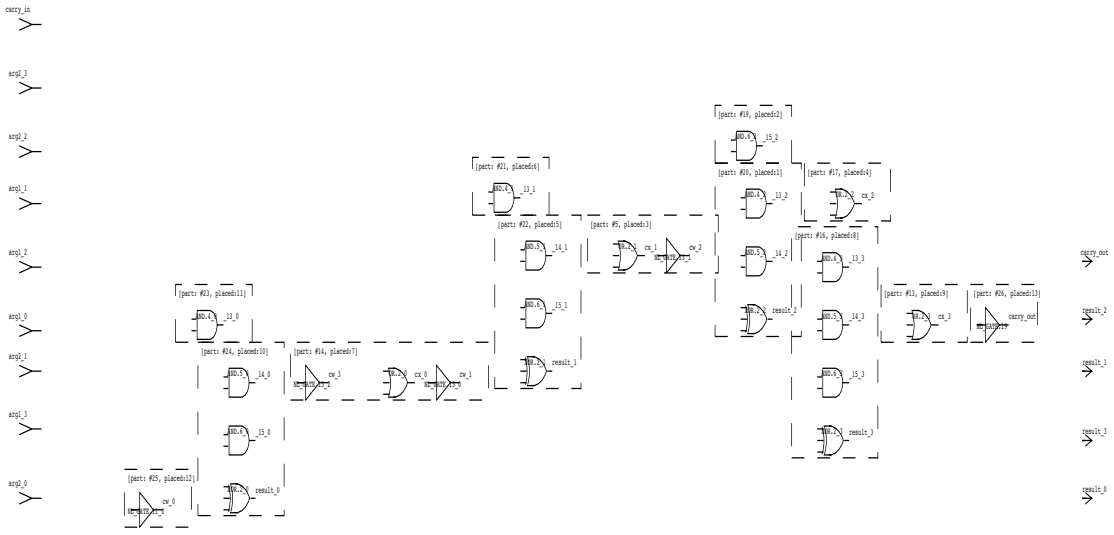


Figure 62 Slope-based partitioning, xfouls = 1, yfouls = 6

APPENDIX B

APPENDIX B

The following figures show the best placement and routing results for a combined schematic. The two schematics combined are those for the SN54155 multiplexer (Figure 46) and the SN54S151 demultiplexer (Figure 42). These clearly show SPAR's weakness in its ability to effectively handle embedded hierarchy within a schematic.

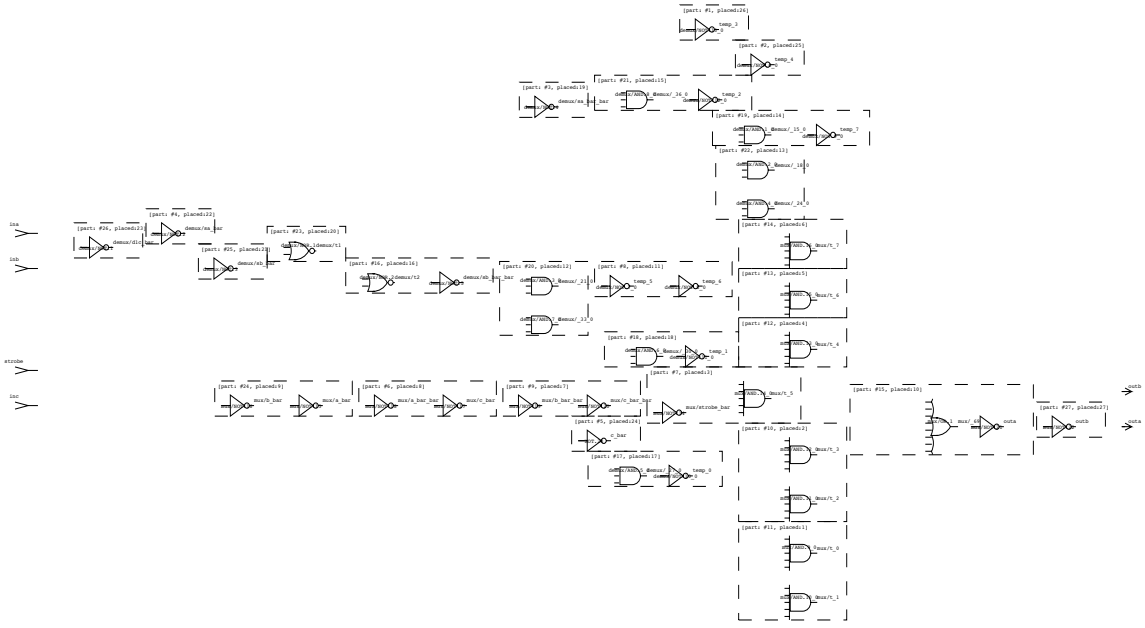


Figure 63 Best size-based partitioning, xfouls = 4, yfouls = 8

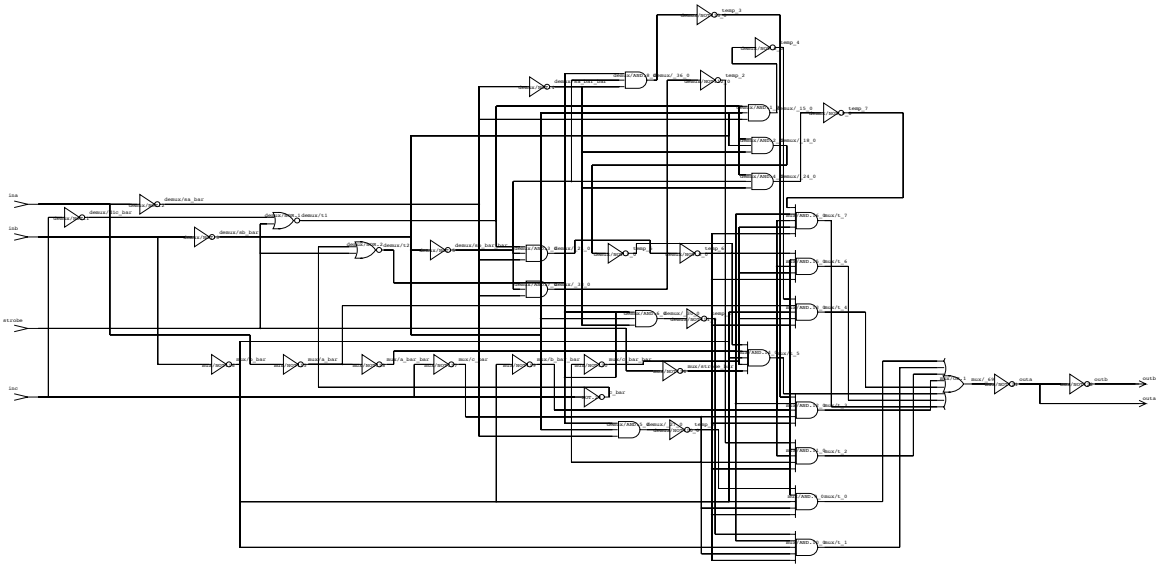


Figure 64 Best size-based partitioning, routed

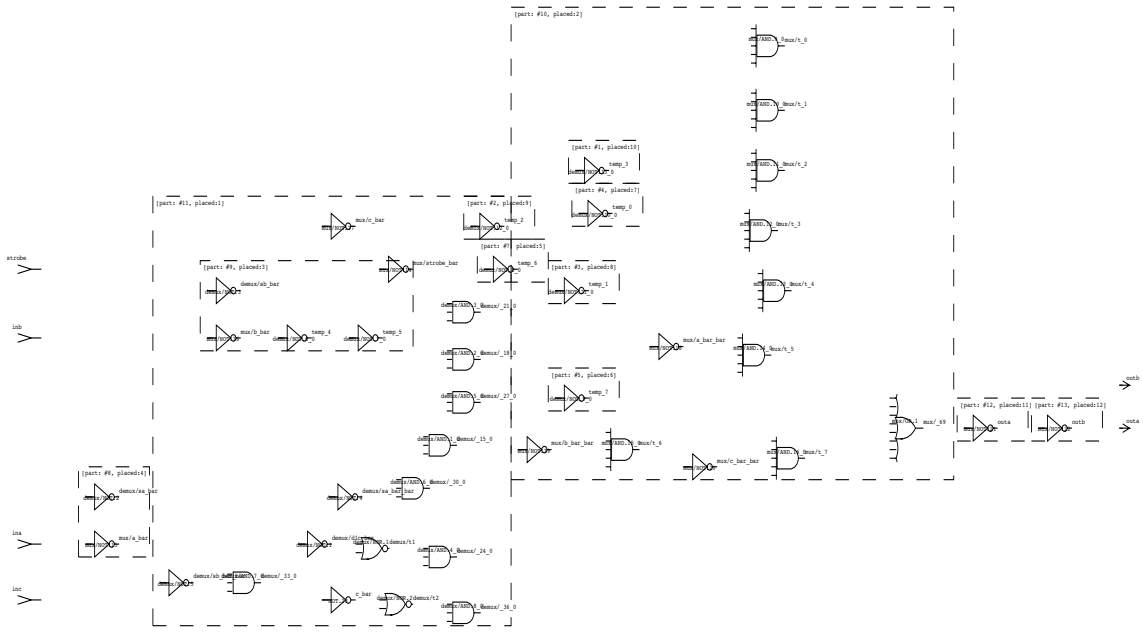


Figure 65 Best Rent's rule-based partitioning, xfouls = 6, yfouls = 12

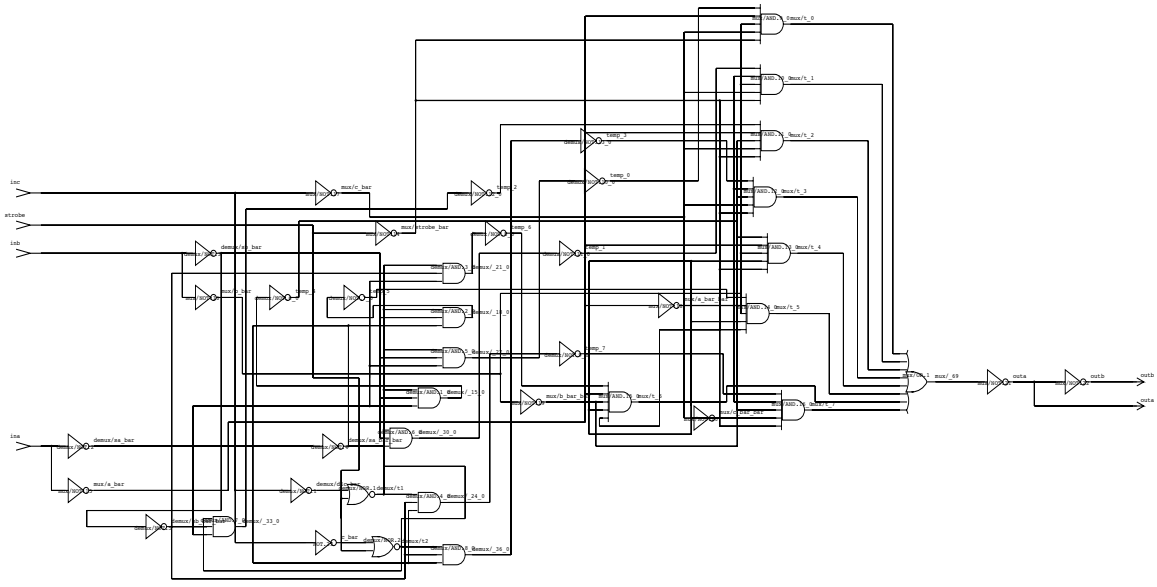


Figure 66 Best Rent's rule-based partitioning, routed

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Ebeling, Carl and Wu, Zhanbing. "WireLisp: Combining Graphics and Procedures in a Circuit Specification Language". *IEEE International Conf. on Comp. Aided Design*, pp. 322–325, November 1989.

[2] Levitan, S.P., Martello, A.R., Owens, R.M., and Irwin, M.J. *Proc. 9th Intl. Symp. on Comp. Hardware Description Languages*, pp. 331–346. Elsevier Science Publishers, B.V., June 1989. "Using VHDL as a Language for Synthesis of CMOS VLSI Circuits".

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

[4] Baer, Charles J. and Ottaway, John R. *Electrical and Elctronics Drawing*, chapter 5, pp. 141–150. McGraw-Hill, fourth edition, 1980. "Flow Diagrams and Logic Diagrams".

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

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

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

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

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

[10] Irwin, M.J., Owens, R., and Levitan, S.P. "Selected Topics in Architecture, Logic and Physical Synthesis". Technical Report TR-CE-88-004, Department of Electrical Engineering, University of Pittsburgh, Pittsburgh, Pennsylvania, November 1988.

[11] IEEE Press, New York, NY. *IEEE Standard VHDL Language Reference Manual*, 1988. IEEE Std 1076-1987.

[12] 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.

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

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

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

[16] May, M. "Computer-Generated Multi-Row Schematics". *IEEE Transactions on Comp. Aided Design*, Vol. 17 No. 1, pp. 25–29, January 1985.

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

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

[19] Hightower, David W. "A Solution to the Line-Routing Problems on the Continuous Plane". *6th Design Automation Workshop Proc.*, pp. 11–29, 1969.

[20] Romesburg, H. Charles. *Cluster Analysis for Researchers*. Lifetime Learning Publications, 1984.

[21] Texas Instruments, Dallas, Texas. *The TTL Data Book for Design Engineers*, first edition, 1983.

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

[23] Borse, Garold L. *Fortran 77 and Numerical Methods for Engineers*. PWS-KENT, Boston, Massachusetts, 1985.

[24] Texas Instruments, Dallas, Texas. *LSI Logic Data Book*, 1986.