LAYOUT FROM A TOPOLOGICAL DESCRIPTION

Martine Denise Francoise Schlag

July 1986
CSD-860039

UNIVERSITY OF CALIFORNIA

Los Angeles

Layout from a Topological Description

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Martine Denise Francoise Schlag

1986

© Copyright by

Martine Denise Francoise Schlag

1986

To Sebastian

TABLE OF CONTENTS

Chapter 1 Introduction 1

1.1 Integrated Circuits 2
1.2 Design Systems for Integrated Circuits 7
1.3 Proposed Method: Layout from a Topological Description 14

Chapter 2 Planar Topology 21

2.1 Defining and Representing Planar Topology 22
2.2 Circuits with Planar Topology 28
2.3 Operations on Planar Circuits 35
2.4 Representation of Layouts by Planar Circuits 24
2.5 Unique Representation of Planar Topology 46
2.6 Removing Unclean Divides 58
2.7 Removing Glues 67
2.8 Best Planar Topology 92

Chapter 3 Mapping FP to Planar Circuits 100

3.1 FP and its Salient Features 100
3.2 Describing Circuits in FP 103
3.3 The Structural Implications of FP 105
3.4 Pruning Planar Circuits 112
3.5 The Planar Circuit of an FP expression 120
3.6 Sequential Circuits 138
3.7 Implementation of ω 153
3.8 Planar Topology of FP Expressions 156

Chapter 4 Mapping Planar Topology to Layouts 169

4.1 Layouts of Planar Circuits 170
4.2 The Complexity of Mapping Planar Circuits to Layouts 173
4.3 Packing Planar Circuits into Cross-sections 183
4.4 Mapping Sequences of Cross-sections to Abstract Layouts 196
4.5 Transforming Abstract Layouts into Layouts 211

Chapter 5 Examples 219

5.1 Decoders 220
5.2 Carry-Save Array Multiplier 225
5.3 Carry Chain Adder 232
5.4 Tally 240
5.5 FFT 244
5.6 Inner Product 250
5.7 Memory 254

Chapter 6 Improving Planar Topology 261

6.1 Measuring Planar Circuits 261
6.2 Transforming Planar Circuits 271
6.3 The Complexity of Pin Alignment in Planar Circuits 274
6.4 Using Computation Trees to Improve Planar Circuits 284

Chapter 7 Conclusion 300

7.1 Summary 300
7.2 Future Research 304

References 308

Appendix: Description of FP 312

LIST OF DEFINITIONS

2.1 Embedding of a graph 22
2.2 Planar Realization of a graph 23
2.3 Homeomorphism 23
2.4 Orientation preserving hoemorphism 23
2.5 Topological Transformation 23
2.6 Topologically Equivalent Planar Realizations 23
2.10 Net-connectivity 29
2.11 Circuit-connectivity 29
2.12 Planar Circuit 32
2.13 Trivial node of a Planar Circuit 34
2.14 Trivial self-loop of a Planar Circuit 34
2.15 Isomorphism of Planar Circuits 34
2.16 Homeomorphic Planar Circuits 40
2.18 Planar Topology of a Layout 46
2.19 Indivisible R-node 47
2.20 Maximal R-node 47
2.21 Maximal, Indivisible, Planar Circuit 47
2.40 Tangling 77
2.41 Orderly Divide 78
2.42 Orderly Refolding 78

3.1 Weak Planar Circuit 112
3.2 Dead-end Pin 113
3.3 Marking of a Weak Planar Circuit 115
3.4 Partial order of Markings of a Weak Planar Circuit 115
3.5 Markable pin 115
3.6 Maximal Marking 115
3.8 Pruning of a Marked Weak Planar Circuit 117
3.9 Output Marking of a Weak Planar Circuit 117
3.10 Directable Weak Planar Circuit 118
3.12 Marking of an FP object, $\mu(x)$ 133
3.18 Flattened node of a Computation Tree 160
3.19 Routing Tree 161
3.20 Normal Form of a Computation Tree 161
3.21 Pseudo-Normal Form of a ! Computation Tree 164
3.22 Fork-Routing Tree 165
3.23 Fork-Routing Tree 165
3.24 Pseudo-Normal Form of a Seq Computation Tree 165
3.25 Normal Form of a $\&^T$ Computation Tree 166

6.1 Bipartite *R*-node 264
6.2 Transitively Orientable Graph 265
6.3 Cost of an *R*-node 269
6.4 Legally Derived Planar Circuit 273
6.5 Spine, Vertebrae 291

ACKNOWLEDGEMENTS

I am in debt to both of my co-chairs, Professors Ercegovac and Greibach, for their advice, encouragement and support during my graduate studies. I really appreciated their help and hope that I can be as good an advisor. I would like to thank Professors Carlyle, Baker and Stiny for their participation and comments.

The members of the FP group, Jose Arabe, Dorab Patel, T. M. Ravi, and Paul Tu as well as the members of the loyal opposition, Miquel Huguet and Pak Chan, deserve thanks for letting me bend their ears on countless occasions and for providing invaluable feedback. The denizens of Boelter Hall 3680 provided a lively and stimulating environment which I will miss.

Last but not least, I would like to thank my parents without whom this work would not have been possible, and Pak Chan for his black bean chicken, picky comments, and friendship.

This research was supported in part by an NSF Graduate Fellowship, an ARCO Doctoral Fellowship, the ONR Contract N00014-83-K-0493 "Specification and Design Methodologies for High-Speed Fault Tolerant Array Algorithms and Structures for VLSI," and the State of California MICRO-ROCKWELL Grant "A High-Level Language Approach to Custom Chip Layout Design."

ABSTRACT OF THE DISSERTATION

Layout from a Topological Description

by

Martine Denise Francoise Schlag

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1986

Professor Sheila A. Greibach, Co-Chair

Professor Milos D. Ercegovac, Co-Chair

The interconnection topology of a circuit does not, in general, correspond to a planar graph. However by encompassing the routing of a circuit in the specification, it is possible to obtain a planar characterization of the topology of a circuit. The planar topology of a circuit is formally defined and the use of specifications with planar topology for the layout of integrated circuits is examined. An applicative language(FP) is used to obtain circuit specifications with planar topology. The planar topology arises naturally out of the constructs used to specify the behavior of the circuit. An efficient mapping from planar topology to geometry is implemented. The problem of transforming the planar topology to minimize the interconnection complexity is addressed by exploiting the structural information of the specification as opposed to using only the planar topology.

VITA

August 4, 1957	Born, Uccle, Belgium
1978	B.A. Mathematics, University of California, Los Angeles
1982	M.S. Computer Science, University of California, Los Angeles

PUBLICATIONS

M. Schlag, Y. Z. Liao and C. K. Wong, "An Algorithm for Optimal Two-Dimensional Compaction of VLSI Layouts," Integration, the VLSI Journal, Vol. 1 No. 2&3 (Oct. 1983) 179-209.

M. D. F. Schlag, L. S. Woo and C. K. Wong, "Maximizing Pin Alignment by Pin Permutations," Integration, the VLSI Journal, Vol. 2 1984 pgs. 279-307 .

M. Schlag, F. Luccio, P. Maestrini, D. T. Lee and C. K. Wong, "A Visibility Problem in VLSI Layout Compaction," **Advances in Computing Research, Vol. II, VLSI Theory** 1984.

M. D. F. Schlag, E. J. Yoffa, P. S. Hauge, and C. K. Wong, "A Method for Improving Cascode-Switch Macro Wirability," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-4 No.2 pgs. 150-155. April 1985.

D. Patel, M. Schlag, and M. Ercegovac, "vFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms," **Functional Programming Languages and Computer Architecture Conference, Nancy, France, September 1985. Lecture Notes in Computer Science**, ed. J. Jouannaud pgs. 238-255, Springer-Verlag Berlin 1985.

Y. F. Wu, P. Widmayer, M. D. F. Schlag, and C. K. Wong, "Rectilinear Shortest Paths and Minimum Spanning Trees in the Presence of Rectilinear Obstacles," to appear in IEEE Transactions on Computers.

CHAPTER 1

Introduction

The time and effort required to design an integrated circuit is the most significant factor in its cost. The complete geometrical specification of an integrated circuit is time consuming, error-prone and inefficient since parts must often be redesigned due to uninformed decisions made at early stages. Computer scientists have attempted to reduce this cost by providing 'silicon compilers' which produce an integrated circuit from a behavioral specification as conventional compilers produce machine code from a programming language [Joha79, Gajs85]. This concept although attractive, results in integrated circuits with lower performance and less efficient use of area. Unlike the generation of machine code in which the location of code has little bearing on its performance, the geometrical properties of an integrated circuit can affect its performance. The efficient use of area impacts performance by permitting larger systems to be implemented as integrated circuits. The generation of integrated circuits from behavioral descriptions without considering nor providing any of the geometry of the design, seems to compromise performance and efficient use of area.

There is a level of geometry which is flexible enough to encompass both abstract representations of a design as well as its final layout, 'planar topology.' In this thesis, the use of behavioral specifications with planar topology for integrated circuits is examined and a system generating layouts of integrated circuits is implemented. A functional language, FP [Back78], in which behavioral

specifications imply the planar topology of a circuit, is used as a specification language for integrated circuits. Since only the planar topology of the specification is fixed, its geometry is flexible and adapts, as the specification is refined from a high level behavioral description to one comprising only circuit devices. This is contrary to the notion of a silicon compiler, since designers are responsible for the geometric implications of their specifications. However, since graphical feedback at varying levels of abstraction can be provided efficiently, the geometric consequences of the specification can be assessed during the synthesis of the design rather than after its completion.

The use of a behavioral language as a specification of a circuit and its planar topology will be described in the last section of this chapter. Before introducing this approach, integrated circuits and the design process are described as well as the state of the art in design tools and the problems associated with them.

1.1 Integrated Circuits

An integrated circuit is formed by overlapping various conducting materials on different levels with insulating layers in between. In the Metal-Oxide-Semiconductor (MOS) technology, the conducting materials consist of diffusion, polysilicon and metal. A transistor is formed by bridging two diffusion regions with polysilicon, providing the basic switching element for the implementation of digital circuits. Wires to interconnect these elements are formed by paths in layers. A wire may change layer with a contact hole, which is a hole cut out of the insulating layers enabling the conducting layers to come into contact. These wires are not ideal conductors since the electrical properties of the different materials introduce

resistance and capacitance. This provides a mechanism for realizing these devices. However it also results in 'parasitic effects,' unintended effects which affect the behavior of the circuit.

An integrated circuit design consists of the masks used in the fabrication process to generate the patterns of material on each layer. The masks are described by a set of shapes in the plane, of different colors, which indicate the presence or absence of certain layers within the region occupied by shape. This formulation of the circuit is referred to as artwork or fixed geometry. Laying out an integrated circuit consists of generating the necessary shapes to implement the circuit elements and embedding them in a planar region according to design rules of the technology being employed. Imperfections in the patterning of the materials on the wafer occur during the fabrication process. The design rules provide a degree of tolerance to the limitations of the fabrication process, insuring that the artwork has a reasonable chance of being realized by the fabrication process. Although design rules depend on the particular technology, they generally consist of minimum width requirements (which can be different for each layer), overlapping requirements and spacing constraints between elements (which may depend on the types of elements involved).

Advances in fabrication technology have made it possible to reduce the minimum width and spacing requirements, thus accommodating larger systems as integrated circuits. As a result the complexity of generating and analyzing these designs has become a the major difficulty in the exploitation of Very Large Scale Integration. The use of computers in the form of design tools has become essential in managing the complexity of designing a VLSI circuit. The design process entails

going from a function describing the behavior of the circuit to an arrangement of colored polygons in the plane (artwork). To use these tools the designer must provide a description of the circuit at a level suitable for the particular tool. The descriptions can be classified into the four levels given below. The first is the highest level and the last is the lowest level, at which an integrated circuit can be specified.

Functional description

A functional description characterizes the behavior of a circuit. Examples of functional descriptions are algorithms, temporal logic equations or logic diagrams. The functional descriptions can be at various levels of abstraction and are not required to provide information to realize their behavior with circuit elements (transistors, resistors, capacitors). The primitives of the functional description must be realized by circuit elements and these must be combined in a manner which realizes the behavior of the functional description.

Circuit description

A circuit description is given in terms of transistors, resistors, capacitors and their interconnection. It differs from a functional description in that its behavior is the result of the time dependent interaction of its components forming electrical circuits.

Stick Diagram/Symbolic Layout

A symbolic layout is an abstraction of fixed geometry in which symbols corresponding to circuit elements, are embedded in the plane. These symbols are expanded into the corresponding artwork for the circuit element and

positioned to meet the design rules. The relative positions between symbols are preserved in the expansion and positioning of the artwork.

Fixed Geometry

This is the actual artwork. It contains the exact coordinates and dimensions of the shapes of each layer.

These four levels of descriptions are illustrated in Figures 1.1, 1.2, 1.3 and 1.4 with a Nand gate implemented in *nmos*.

Functional description

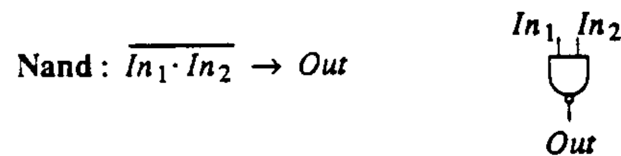


Figure 1.1 Functional Description of a Nand gate.

Circuit description

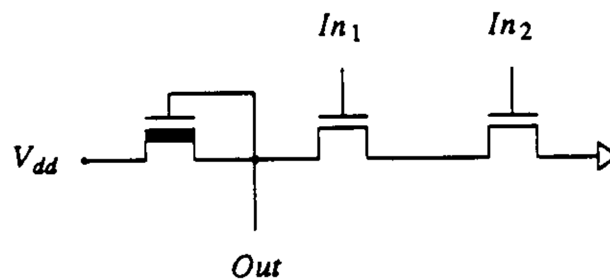


Figure 1.2 Circuit Description of a Nand gate.

Stick Diagram/Symbolic Layout

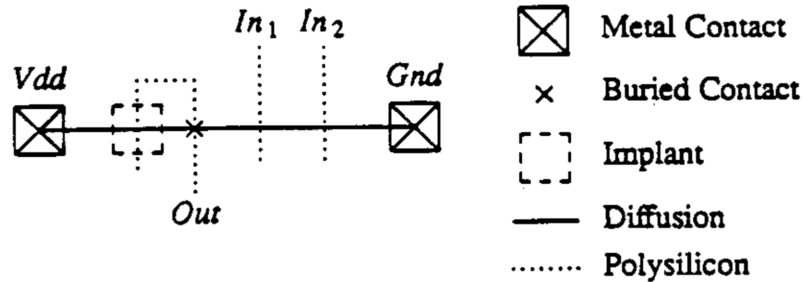


Figure 1.3 Stick Diagram of a Nand gate.

Fixed Geometry

Figure 1.4 Layout of a Nand gate.

The last level is the desired final format. Unfortunately, many of the characteristics needed to accurately estimate the feasibility and performance of a design are geometric and only manifest themselves in the lower two levels. As a result, the design process usually requires several iterations, in which the specification at a higher level must be modified to meet constraints at a lower level.

We classify systems of computer-aided design tools for integrated circuits according to the level of specification at which the tools take over, that is, become responsible for the correspondence of a description to its descriptions at the lower levels. Verifying that a design does in fact meet its higher level specifications is an essential aspect of the design process because of the time and cost of the design

cycle. Thus the higher the level at which the system takes over and insures 'correctness' with respect to the specification, the better. However, as we shall see in the next section producing layouts of integrated circuits from high level specifications with no geometry is computationally intensive and may require compromising performance and efficient use of area.

1.2 Design Systems for Integrated Circuits

Design tools can be separated into two classes according to the level of specification which they can accept. Low level tools operate on fixed geometry or symbolic layouts while high level tools accept circuit or functional descriptions.

Low Level Specifications

Specifying a circuit in terms of fixed geometry entails the use of a graphics editor with which elements can be placed and positioned using keyboard commands and/or a mouse. There is usually a library available from which predesigned elements can be retrieved and placed in the current design. Some systems such as Magic [Oust83] provide additional layout aids such as 'plowing,' and some form of 'automatic routing.' The advantage of this type of system is that it gives the designer complete control over the final layout. However it suffers from the following drawbacks.

1. The objects handled have fixed shapes and sizes. They are not flexible in adapting to the different interconnection, size and shape requirements of various environments. Much of the work involved in making the pieces fit is left to the designer. The same circuit element with a slightly different shape or with different terminal positions is dealt with as a different object.

2. Since in this type of system the designer is dealing with shapes, the correspondence between these shapes and the intended design must be verified with the use of error detection tools such as design rule and electrical rule checkers. Design rule and electrical rule checkers only check the 'syntax' of the layout and the error reports they generate must be scrutinized.

3. The quality of the layout is a function of the skill of the designer and the time and effort afforded to it. As the size of circuits increases, the complexity and time involved will exceed the capacities of human designers.

4. The complexity of simulating a low-level specification is prohibitive and requires the use of extractors to abstract the circuit from the artwork, in essence reconstructing the specification at a higher level. Extraction is not error free and cannot abstract functional specifications which must be provided by the designer.

The second problem can be dealt with as in Magic, by providing an incremental design rule checker which operates in parallel with the designer during an editing session, checking each change in the layout for design rule violations as it is made. It is interesting to note that Magic has been made more efficient than its predecessors by exploiting the fact that the set of shapes being manipulated is not arbitrary, but corresponds to a circuit.

The first two problems associated with specifying layouts by fixed geometry, are alleviated by using symbolic layout specifications [Will78, Roge86]. With a graphics editor, a designer positions symbols which are subsequently 'expanded' by the system. Expanding a symbolic layout entails replacing symbols by their

corresponding fixed geometry and positioning them with respect to the other expanded objects. Since the design rules are used as guides to position the objects, the layout should automatically satisfied the design rules. This type of system can benefit from 'delayed binding' [Lipt82]; the entire layout is described before any part of the specification is transformed into a lower level of specification. This allows the transformation to be more sophisticated, using global information to determine the dimensions and exact positions of elements.

Once the elements have been expanded (their sizes and shapes have been determined), positions are obtained by compaction. The goal of compaction is to pack the artwork as tightly as possible to reduce the size of the layout. The size of the layout is usually measured as the area of the smallest bounding box containing it. Compaction is performed by satisfying a set of inequalities which represent minimum distance and relative position constraints among pairs The success of current compaction algorithms depends largely on the initial positioning of elements; shifting an element upward or downward slightly before packing it horizontally against other elements may produce a better result. Unfortunately two-dimensional compaction in which elements are permitted this type of movement has been shown to be NP-hard [Schl83]. Compaction usually consists of a sequence of one-dimensional compactions, alternating between vertical and horizontal compaction.

Programming languages can also be used to specify fixed geometry or a symbolic layout which would otherwise be generated with a graphics editor. Using a procedural language such as ALI [Lipt82] or CHISEL [Karp83] the layout is described by calls to procedures which generate the element, connect it, and position it with respect to other elements the procedure is told about or generates itself. In

ALI the designer must specify relative position constraints explicitly. There is no guarantee that the constraints specified by the designer are consistent. In CHISEL, the objects have fixed geometry and limited routing is provided. The advantage of a language over the graphics editor is in its ability to handle parameterized descriptions of circuits, generating elements repetitively, based on parameters computed within the language. It is a powerful tool to automate the specification of a symbolic layout; however the designer is still forced to think in terms of the relative positions and dimensions of elements and their interconnection. Behavior must still be inferred.

Symbolic layout tools do not resolve the third and fourth difficulties which are the result of describing layouts by structure alone. The design time and inefficiency involved in producing an integrated circuit by specifying it with structure, as well as the complexity of verifying the design still remain. These problems have motivated the interest in automatic layout tools and silicon compilers, tools which start from a higher level specification.

High Level Design Tools

A system of tools is considered high-level when it generates the masks itself based on a functional or circuit level description. These tools consist of automatic layout tools and silicon compilers. Automatic layout tools take a description of the circuit in terms of modules and nets and generate masks while silicon compilers generate circuit elements and interconnections from a behavioral specification. The behavioral specification can be in terms of switching logic expressions and/or a register-transfer language. A silicon compiler may in fact use an automatic layout system once it has determined the circuits elements and their interconnection.

Automatic place and route systems for custom layout accept as input a set of circuit elements (modules) and their interconnections (signal nets). Modules are geometric objects (usually rectangles) with labeled points on their boundaries (pins). Nets are lists of pins which are to be connected. These can be generated either by a silicon compiler or provided by the designer. The problem of layout is then the task of embedding the hypergraph corresponding to the modules and nets into the grid graph. Minimizing the area of the layout, even for planar graphs, is NP-hard [Dole81]. Asymptotic lower and upper bounds are derived in [Leis80, Vali81] for families of graphs, however these results do not offer practical approaches for efficient layouts of irregular sized objects nor do they consider wire length.

Automatic layout is generally divided into two steps, placement and routing. Two popular schemes for tackling placement are partitioning and clustering. Partitioning is a divide-and-conquer strategy in which the circuit is divided into two roughly equal parts with as few connections as possible between the two parts. The two parts are then placed separately and then glued together. In clustering each circuit element starts in a cluster consisting only of itself. Clusters with high affinity for each other are placed together forming larger clusters. Once the placement is obtained, a router is called on to interconnect the modules. Routing is divided into two phases. In the first phase, wires are assigned to channels (areas left between the modules); they are routed relative to the modules without fixing exact coordinates or considering each other. An attempt is made to avoid congesting any particular channel. Once this is done, each channel is routed, one-by-one taking into account the routing in adjoining channels.

Unfortunately, most of the optimization problems associated with routing and placement have been shown to be NP-hard [Sahn80, Dona80], leaving only heuristic algorithms as candidates for these jobs, making the tools slow and/or the results inefficient and awkward. The problem arises because placement and routing are tightly coupled. It is awkward to deal separately with routing and placement. By considering layout to be the task of embedding of an arbitrary hypergraph in the grid graph, small changes in the hypergraph may trigger large changes in the layout. Because of the complexity in optimizing and balancing opposing criteria, obtaining layouts by performing small random operations has been proposed. The scheme known as Simulated Annealing [Kirk83], involves selecting at random, simple operations whose effect on the layout can easily be computed and performing them if they meet certain criteria. If only operations which improved the layout were allowed, the algorithm would become blocked, unable to make any moves, possibly in a sub-optimal local minimum. To avoid this problem, some operations which increase the cost of the layout below a certain value are permitted. A set of configurations is explored by first allowing moves which increase the cost within a large range and then slowly decreasing this range, simulating the annealing process and hopefully resulting in a near optimal layout. It has been observed [Kirk84] that higher values and thus more dramatic moves are required for layouts of connection graphs of actual circuits as opposed to randomly generated graphs. This is attributed to the hierarchical structure of circuits.

To facilitate the layout of circuits, structured design techniques such as gate array, master slice, or standard cell layouts are used. In gate array and master slice, the layout task consists of mapping circuit elements into predefined and positioned structures. The placement algorithm attempts to minimize the routing needed to

interconnect these elements. These methods suffer in terms of performance because of wire lengths and inefficient use of area. In standard cell layout, the design must be constructed of 'cells' satisfying specified geometric and electrical rules. Because of the homogeneity of the cells, an efficient placement can be obtained by arranging the cells into row and columns; however they still must be routed. These techniques avoid some of the optimization problems of custom layout by sacrificing some of the geometric freedom available in VLSI. However the remaining optimization problems of structured design techniques are no easier than those of custom layout.

To avoid the difficulty of routing and placement, languages for specifying both behavior and structure have been proposed. These tools which are often referred to as silicon compilers can be divided into two categories, those which map behavioral components into pre-conceived physical components and layouts, and those which process behavioral specifications augmented with structural information.

1. Circuits corresponding to a set of architectural types such as registers or Arithmetic Logic Units are generated for behavioral constructs in the language. Logic expressions can be mapped into Programmable Logic Arrays. These circuits are then mapped to a layout using an automatic place and route system geared to the particular architectural types. The placement algorithm contains rules governing the placement of particular types. Examples of such a systems are MacPitts [Ance83], and CAPRI [Ance83]. In CAPRI which the behavioral specification is obtained from a register transfer language, IRENE. This style of design relies heavily on past experience for good templates for each architectural type. It is not suitable

for experimenting with new algorithms and interconnection patterns since the geometry of the layout is more the result of the tool than of the specification.

2. Relative positions of elements and interconnections are specified within cell descriptions. Examples of such languages are ZEUS [Lieb83] and Bristle Blocks [Joha79]. In Bristle Blocks the layout is constructed of stretchable cells which are connected by cell abutment. ZEUS is a procedural language similar to ALI in which behavioral information is provided as well as relative positions. Although these languages offer a behavioral description, the designer is still forced to think in terms of size and shapes and connections. These languages are in essence an augmentation of low-level specification languages such as ALI and CHISEL with behavioral information.

In the next section we will propose a design methodology based on a functional language. The idea is not to impose geometry on a behavioral specification, but rather exploit the geometry which is a consequence of the behavioral specification. This is achieved by using a functional language, FP [Back78], in which behavioral specifications of circuits imply the topological organization of the components and their interconnection in the plane. This is the level of geometry which can follow the design through the different design stages as it is refined.

1.3 Proposed Method: Layout from a Topological Description

The layout methods described in the last section are either Combinatorial or Constructive. Since most of the problems in placement and routing are probably intractable, Combinatorial Methods rely on heuristics such as partitioning,

clustering, quadratic assignment and simulated annealing. The computational requirements, varying quality of results, and the instability of these methods make them unattractive for layout design.

In a Constructive Method, the layout is assembled by the designer using either graphical tool or a description language. Examples of these include graphics editors, symbolic layout tools and layout languages such as ALI. The symbolic layout tools and languages provide more flexibility than dealing directly with fixed geometry, and the languages have the ability to capture patterns and symmetries in the design. Unfortunately two-dimensional compaction is also probably intractable and so the exact positions of objects in a graphically specified design determine the final result. This is less of a problem with languages, since relative positions can be as incomplete or complete as desired. However, connections are treated as any other elements so the wiring is inflexible. Layout languages such as ALI provide the means to capture the patterns and symmetries of the circuit and its layout. This is a first step in incorporating the geometric implications of a circuit's behavior into its layout.

Many researchers have proposed applicative (functional) languages as hardware description languages, precisely because they provide at once both a behavioral and structural description of a circuit [Laht81, John84, Shee84, Pate85]. However a behavioral specification in the functional language, FP, provides not only a circuit description but geometric information which should be exploited to obtain its layout. The patterns and symmetries of the placement and routing can be captured directly from the behavioral description. 'Abstract layouts' or layouts of circuits will be obtained from the FP behavioral specifications. In the following

chapters the level of geometry which is implied by an FP specification, 'planar topology' and the mapping of the FP specifications to layouts will be explored. 'Abstract layouts' and an actual layout produced by an implementation of this mapping will be presented. The following is a brief overview of the contents of the following chapters.

We will define the level of description of a layout offered by FP as the 'planar topology' of a circuit. This is formally defined and discussed in Chapter 2.

We develop the mapping from FP expressions to the planar topology of a circuit in Chapter 3. We also incorporate new constructs defined in [Pate85] to describe sequential circuits into this mapping. The mapping is implemented in such a way as to preserve the hierarchical representation of the circuit afforded by the constructs of the FP expression.

In Chapter 4 we discuss the technique used for transforming the planar topology of a circuit described in FP to a layout. The problem of minimizing the area of a the layout of a circuit with a fixed planar topology is shown to be NP-hard. A method for constructing the layout which exploits the hierarchical representation of a circuit afforded by the FP specification is implemented. This implementation generates both 'abstract layouts' which are similar to symbolic layouts and artwork.

In Chapter 5 we give several examples of FP specifications of circuits and their corresponding 'abstract layouts' and/or layout.

The problem of altering the planar topology of a circuit to reduce its wiring

complexity is discussed in Chapter 6. We show that it is NP-complete and provide a method based on the hierarchical structure afforded by an FP specification.

As mentioned, the behavioral specification will provide the circuit and its topological organization in the plane. Symbolic layouts tools and languages also provide this level of description of the layout. The difference is that they do so by providing an initial embedding or relative positions of objects which influences the final layout as discussed. They also make no distinction between connections and circuit devices, resulting in a loss of flexibility. In our case, the topological organization in the plane will be the result of constructs which provide the planar organization of sub-circuits and the routing between sub-circuits in the plane. This is possible because the constructs in FP provide both the behavioral and structural relations between sub-circuits. The final layout will be influenced by the constructs used to describe the circuit. However, the algebraic nature of FP will permit us to perform transformations on the specification which alter the constructs and the final layout while preserving its 'planar topology.' By the 'planar topology' of a circuit, we mean the set of layouts which can be transformed into one another by moving circuit elements around in the plane and stretching connections without ever breaking any connections or lifting wires or circuit elements out of the plane. These transformations are operations which two-dimensional compaction would be expected to perform, and in addition, local reorganization of the wires.

We begin in Chapter 2 by formalizing this notion of the 'planar topology' of a circuit. We first discuss in Section 2.1 how to represent the planar topology of the embedding of a graph using a result of Edmonds. Since circuits do not in general

correspond to planar graphs, in Section 2.2 we capture the layout of a circuit as a graph in which special nodes are used to hide the routing. This graph and its embedding which we call a 'planar circuit' is said to represent the layout of the circuit. Unfortunately this representation is not unique, and in addition to characterize the 'planar topology' of the layout, we must consider operations involving local reorganization of the wiring. These operations on planar circuits which form a group are given in Section 2.3. In Section 2.4, we define what it means for a planar circuit to represent a layout and show that any two planar circuits representing the same layout can be transformed into one another using these operations. We argue that the set of layouts which can be obtained from one another by moving circuit elements around and stretching connections as described above, are represented by planar circuits which are also equivalent modulo these operations. We can then define the 'planar topology' of a layout to be the set of layouts whose planar circuits are equivalent modulo these operations (homeomorphic). In the remainder of the chapter we provide a normal form for planar circuits and show that under certain assumptions about the cost measure and layout procedure, this normal form is optimal within a class of homeomorphic planar circuits. This normal form which is called a maximal-indivisible planar circuit is defined in Section 2.5. Sections 2.6 and 2.7 are devoted to showing the uniqueness (modulo one operation) of the normal form within each class of homeomorphic planar circuits. The uniqueness result can be found at the end of Section 2.7. In Section 2.8 the process of transforming a planar circuit into a homeomorphic maximal-indivisible planar circuit and the optimality of this planar circuit with respect to other homeomorphic planar circuits is addressed.

In Chapter 3, we map behavioral specifications written in FP, to planar circuits. In Section 3.1 we briefly discuss FP and the properties which make it attractive as a specification language. The full description of FP can be found in the Appendix. Sections 3.2 and 3.3 discuss the correspondence between FP expressions and circuits, and the limitations of describing circuits in FP. Before developing the mapping to planar circuits in Section 3.5, we discuss the pruning of planar circuits to remove unnecessary structure, structure which cannot influence the behavior of the circuit. After giving the mapping from FP expressions to planar circuits in Section 3.5, we consider the mapping of the constructs necessary for describing synchronous sequential circuits in Section 3.6. Section 3.7 describes the implementation of the mapping which preserves the hierarchical representation of the planar circuit afforded by FP's combining forms. In Section 3.8 operations are applied to this hierarchical representation to transform the planar circuit into a homeomorphic maximal-indivisible planar circuit.

In Chapter 4, the problem of mapping a planar circuit to a layout is discussed. We show that as in the case of two-dimensional compaction, finding the layout of minimal area is probably intractable. We provide a method for synthesizing the layout of using the hierarchical description afforded by the FP specification. Thus the behavioral description is exploited to provide this layout. We obtain an 'abstract layout' for FP expressions in which wires have zero width and circuit elements are boxes whose dimensions are in abstract units. We show how to transform these 'abstract layouts' into actual layouts, artwork.

Several examples of specifications and their 'abstract layouts' are given in Chapter 5 to illustrate the features of using FP to describe circuits as well as the

resulting layouts. The ability of the system to provide graphical feedback through 'abstract layouts' at varying levels of abstraction during the synthesis of the design is illustrated in these examples.

Topological cost measures for planar circuits are discussed in Section 6.1 of Chapter 6 and the operations which should be considered are discussed in Section 6.2. We show the intractability of optimizing planar circuits considering only small operations in Section 6.3. In Section 6.4 we present a method for performing these operations on planar circuits generated from FP expressions.

By using FP we have provided a system in which the 'planar topology' of the layout is the direct result of its behavioral description. We have not avoided the intractable problems of layout by specifying the 'planar topology' of the circuit, however we have exploited the behavioral description to obtain reasonable layouts quickly. As a result, the quality of the layouts obtained from FP expressions depend on the specification. This is mitigated to some extent by the operations performed in Section 3.8 and Section 6.4. However, the efficiency in producing graphical feedback from a behavioral description allows the designer to evaluate design decisions and experiment with alternatives during the synthesis. The flexibility of specifying only the 'planar topology' of the layout, allows changes to be easily accommodated. In addition, the algebraic nature of FP provides the mechanism for transforming the specification to improve its layout while preserving its behavior. The automatic application of transformations to improve the layout is a topic for future research which depends on a well defined mapping from FP to layouts.

CHAPTER 2

Planar Topology

In this chapter, the 'planar topology' of a circuit is defined. We will begin by defining the planar topology of a graph and then extend this to circuits. In Section 2.1, a combinatorial representation of the planar topology of an embedding of a graph is obtained using a result of Edmonds. Since circuits do not in general correspond to planar graphs, in Section 2.2 the layout of a circuit is captured as a graph with special nodes to hide the routing. This graph and its embedding, a 'planar circuit,' represents the planar topology of a layout. Unfortunately this representation is not unique, and in addition to characterize the 'planar topology' of the layout, we must consider operations involving local reorganization of the wiring. These operations on planar circuits which form a group, are given in Section 2.3. In Section 2.4, we discuss how a planar circuit represents a layout and show that any two planar circuits representing the same layout can be transformed into one another using these operations. The set of layouts which can be obtained from one another by moving circuit elements around and stretching connections, have representations as planar circuits which are equivalent modulo these operations. The 'planar topology' of a layout can thus be defined as the set of layouts whose planar circuits are equivalent modulo these operations (homeomorphic). In the remainder of the chapter we provide a normal form for planar circuits and show that under certain assumptions about the cost measure and layout procedure, this normal form is optimal within a class of homeomorphic planar circuits. This normal form which is called a

maximal-indivisible planar circuit is defined in Section 2.5. Sections 2.6 and 2.7 are devoted to showing the uniqueness (modulo one operation) of the normal form within each class of homeomorphic planar circuits. The uniqueness result can be found at the end of Section 2.7. In Section 2.8 the process of transforming a planar circuit into a homeomorphic maximal-indivisible planar circuit and the optimality of this planar circuit with respect to other homeomorphic planar circuits is addressed.

2.1 Defining and Representing Planar Topology

The planar realization of the graph is, in simple terms, a drawing of the graph on a surface so that no edges cross and each edge coincides only with the two vertices it joins. More formally,

Definition 2.1

An *embedding* of a graph $G=(V,E)$ in a space X, is a pair, (f_V,f_E) where $f_V:V\rightarrow X$ is a mapping of the vertices of G into points of X and a mapping of the edges into paths in X, $f_E:E\rightarrow([0,1]\rightarrow X)$ associates with each edge of G a homeomorphic image of the unit interval, $f_E(e)=e(t)$ satisfying the following conditions.

a. Vertices are mapped to distinct points.

   If $f_V(v) = f_V(u)$ then $v = u$.

b. Edges do not intersect.

   For any $0 < t$, $s < 1$ if $f_E(e)(t)=f_E(e')(s)$ then $e=e'$ and $t=s$.

c. Edges intersect only the two vertices they join at their ends.

   $f_E(e)(t)=f_V(v)$ if and only if $e=(u,w)$, and $(v,t)=(u,0)$ or $(v,t)=(w,1)$.

Definition 2.2

A *planar realization* of a graph is an embedding of the graph in the plane, R^2. A *window* of the embedding is a maximal set of points in the plane which can be pairwise connected by a path not intersecting the embedding of the graph. The *external window* is the window containing the point at infinity (arbitrarily distant points from the graph).

Not all graphs have planar realizations; those that do are *planar* graphs. Thus when we speak of the planar topology of a graph or an embedding of a graph in the plane it is understood to be a planar graph. By the planar topology of a graph, we mean a class of planar realizations which are equivalent modulo the operations of sliding vertices and edges and stretching edges, i.e. transformations which preserve the structural integrity of the graph without ever picking it up, out of the plane. A more formal definition is provided below.

Definition 2.3

A *homeomorphism, h,* of a space X onto a space Y is a continuous bijective mapping whose inverse is also continuous.

Definition 2.4

A homeomorphism, h, of an orientable surface with an orientable surface is *orientation-preserving* if for each closed curve C, traversing C in the clockwise direction corresponds to the traversal of $h(C)$ in the clockwise direction as well.

Definition 2.5

A *topological transformation* of an orientable surface is an orientation-preserving homeomorphism of the surface with itself.

Definition 2.6

Two planar realizations of a graph G, $\Psi = (f_V, f_E)$ and $\Psi' = (f'_V, f'_E)$ are *topologically equivalent* if there exists a topological transformation of plane, h, such that $h(f_V(v)) = f'_V(v)$ for each vertex v and for each edge e, if $f_E(e) = e(t)$ and $f'_E(e) = e'(t)$ then $h(e(t)) = e'(t)$.

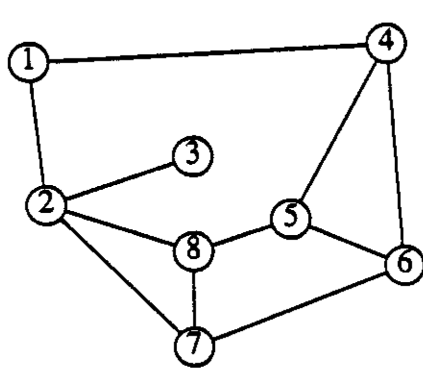
It is clear that this defines an equivalence relation since topological transformations include the identity mapping, are invertible, and composable. Thus specifying a particular planar realization of a graph G is one way of specifying a planar topology for G. However this definition does not provide a practical method for determining if two embeddings are equivalent or for generating equivalent embeddings. A combinatorial method for specifying a planar topology of a graph is offered by a theorem of Edmonds.

Theorem 2.7 [Edmo60]

For any connected graph with an arbitrarily specified cyclic ordering of the edges to each vertex, there exists a topologically unique embedding in an oriented closed surface so that the clockwise edge orderings around each vertex are as specified and so that the complement of the graph in the surface is a set of discs.

This theorem proved in [Youn63], guarantees that all embeddings of a graph in the sphere are topologically equivalent if they have the same clockwise ordering of edges around each vertex. The plane is not a closed surface but it is homeomorphic to the sphere minus the north pole, the point that gets mapped to infinity under stereographic projection. Thus we can place the topologically equivalent embeddings of a graph in the plane into a one to one correspondence with pairs consisting of a class of topologically equivalent embeddings in the sphere

along with a specified disc of the embedding from which a point will be removed. This yields the more useful specification of a planar topology for a connected graph comprised of, the clockwise ordering of edges around each vertex and the traversal of its external window in the clockwise direction. By traversal of the external window we mean the following. Select a vertex, v, which is on the external window. Select any edge $e = (v,u)$, which has the external window on its left when looking away from v towards u, (the external window may be on the right as well). Traverse e to reach the u. At u select the edge, e', immediately following e in the clockwise ordering; e' may be the same as e if u is of degree one. Proceed until you return to the original vertex, v, and would next select the initial edge traversed, e. The sequence of vertices visited corresponds to the external window traversed in clockwise order. Figure 2.1 contains an embedding of a planar graph, and the specification of this embedding by the clockwise ordering of edges around each vertex and the clockwise traversal of its exterior window.



Clockwise Ordering of Edges

1	(1,4) (1,2)
2	(2,3) (2,8) (2,7) (2,1)
3	(3,2)
4	(4,6) (4,5) (4,1)
5	(5,8) (5,4) (5,6)
6	(6,7) (6,5) (6,4)
7	(7,2) (7,8) (7,6)
8	(8,2) (8,5) (8,7)

External Window : 1 , 4 , 6 , 7 , 2

Figure 2.1 The specification of the planar topology of a graph.

Before showing that planar topology can be specified in this manner, consider how this definition differs from the usual graph theoretic one. The graph

theoretic definition of equivalent planar embeddings of a planar nonseparable graph requires a one to one correspondence between the windows of the embeddings, (i.e. topologically equivalent embeddings in the sphere without orientation). Our definition makes a distinction in the choice of the exterior window and orientation, and allows us to consider separable graphs as well.

Theorem 2.8 Two planar realizations of a connected graph $G=(V,E)$ have the same clockwise cyclical ordering of edges around each vertex and the same clockwise traversal of their external windows if and only if they are topologically equivalent.

Proof: If two planar realizations are topologically equivalent then it can be shown that they have the same clockwise ordering of edges by noting that the topological transformation of the plane which takes one embedding to the other must preserve a closed curve around a vertex as well as its orientation. Clearly the traversal of the external window must also be preserved along with its orientation. Now suppose two planar realizations of G have the same cyclical ordering of edges around each vertex and clockwise traversal of the external window. Consider their images under stereographic projection into the sphere. The clockwise ordering of edges at each vertex must also be the same in the sphere (although they are reversed with respect to the plane). By Edmond's theorem there is topological transformation h, which takes one embedding into the other in the sphere. Assume for the moment that this homeomorphism preserves the north pole of the sphere. If we take the composition of this homeomorphism with stereographic projection from the plane to the sphere first and then back onto the plane afterwards, we obtain the topological transformation of the plane which topological equivalence requires. Now suppose this homeomorphism does not preserve the north pole. Since both embeddings have

the same cycle and orientation along the external face, the north pole lies within the same disc of the two embeddings in the sphere. In particular their boundaries and orientation are preserved by h. Then there is another homeomorphism, h' which preserves the boundary of this disc in the second embedding while mapping h(north pole) into the north pole; it can be extended to a topological transformation of the sphere by defining it to be the identity on the rest of the sphere. The required topological transformation can be obtained as before using the composition of h' and h in the place of h.

□

This definition offers a much more practical method for representing planar topology since it suffices to specify a cyclical ordering of edges around each vertex and its exterior face. It can be further simplified in cases where there is a vertex of degree one on the exterior window.

Corollary 2.9 If two planar realizations of a graph G have the same clockwise cyclical ordering of edges around each vertex and there is a vertex of degree one which appears on the external window of both, then these two planar realizations are topologically equivalent.

Proof: Simply begin the traversal of the external window described above at the vertex of degree one. There is only one choice for the first edge and each subsequent edge selected must be the same for both embeddings since the vertices have the same clockwise cyclical ordering in both embeddings. Thus by the last theorem the embeddings are topologically equivalent since they have the same clockwise traversal of their external windows.

□

In order to encompass graphs which are not connected, we restrict the planar realizations of a graph to those in which each component has a vertex adjacent to the component of the plane containing infinity. A component is never embedded inside a window of another. Clearly to extend the previous results it suffices to have the same clockwise traversal of external windows for each component or to specify a vertex of degree one on the external window for each component. However, we will only be considering connected graphs.

2.2 Circuits with Planar Topology

In the last section, we discussed how to represent an embedding of a planar graph. In this section we discuss how the layout of a circuit can be represented by the embedding of a particular planar graph which will not only represent the interconnection topology, but the actual routing of the interconnections as well. For our purposes a circuit consists of the following.

1. A set of modules with specified points on their boundaries called pins. These modules may sometimes be referred to as components or boxes.

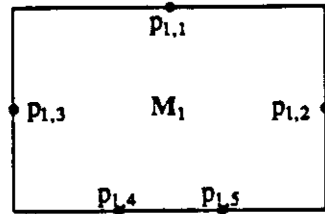
2. A set of input/output pins which will be required to lie on the external boundary of the layout.

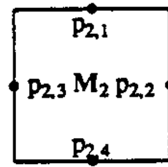
3. A set of net-lists which partitions the set of all module pins and input/output pins into disjoint sets of pins which are to be interconnected.

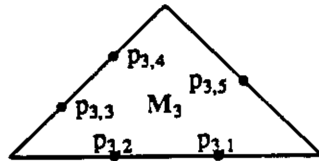
Figure 2.2 contains a circuit and Figure 2.3, a layout for this circuit.

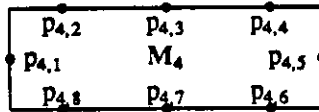
We make a distinction between two types of connectivity in a circuit, connectivity by nets alone and connectivity through modules as well.

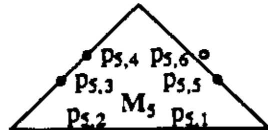
Modules :











Input/Output Pins :

Net lists :

$p_{1,1}, p_{4,3}, p_{5,5}$

$p_{1,2}, p_{1,4}, p_{5,6}$

$p_{1,3}, p_{5,3}, p_{2,3}$

$p_{1,5}, p_{4,2}$

$p_{2,1}, p_{5,2}$

$p_{2,2}, p_{5,1}, io_2, io_6$

$p_{3,1}, p_{4,5}$

$p_{3,2}, p_{4,7}$

$p_{2,4}, io_5$

$p_{3,3}, p_{4,6}$

$p_{3,4}, p_{4,1}, io_3$

$p_{3,5}, p_{4,8}, io_4$

$p_{4,4}, p_{5,4}, io_1$

Figure 2.2

Definition 2.10

Two pins are *net-connected* if they belong to the same net-list.

Definition 2.11

Two pins, p and q are *circuit-connected* if there exists a sequence of pins, $p_1 \cdots p_n$ such that $p_1 = p, p_n = q$, and for each $1 \le i < n$, p_i and p_{i+1} belong to the same net-list or module.

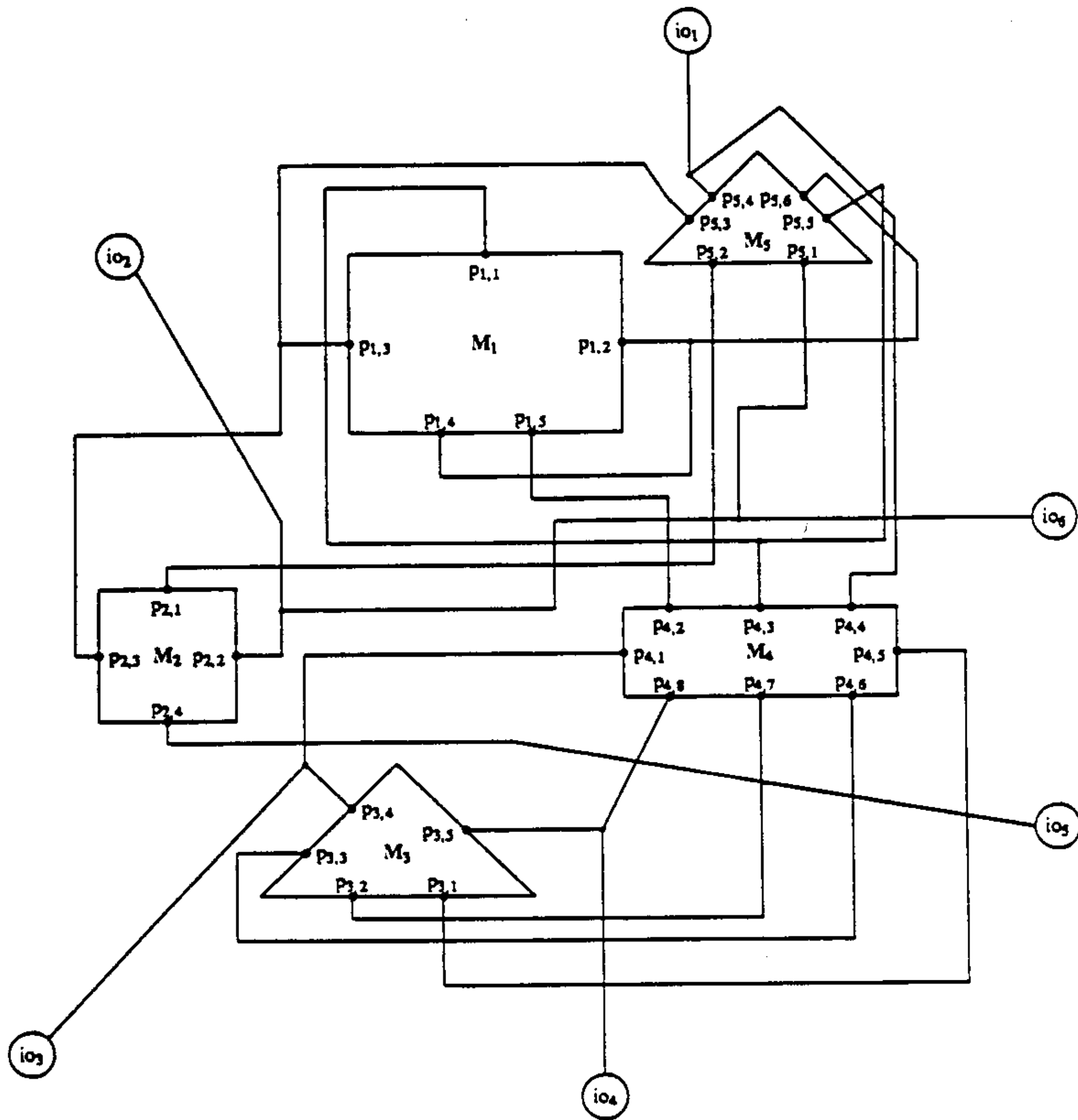


Figure 2.3

The layout of a circuit is an arrangement of the modules and the input/output pins of the circuit in the plane so that no two overlap, along with an implementation of each net-list as a tree whose leaves are the pins which it must interconnect. The

net-lists do not otherwise intersect the modules or input/output pins. The nets are allowed to cross each other and themselves, but a distinction is made as to whether a point is a branching or a crossing even if only one net is involved. The input/output pins must be on the exterior; they must be accessible from the exterior by a path not crossing any of the nets. In Section 2.4 we will also add the restriction that any connected set of points in the plane such that two or more nets lie on each point is contractible to a point after removing the interiors of the components and I/O pins. This will be required in order to cover any such set of points by a region whose boundary is a simple closed curve.

In general, circuits do not correspond to planar graphs (or hypergraphs) and even if they do, it may desirable to lay them out in a non-planar fashion. To specify the planar topology of a circuit with a planar graph, crossing and branching points must be made explicit by incorporating nodes to represent them in the graph. These nodes will be called R-nodes to distinguish them from the nodes corresponding to the modules, B-nodes. Suppose we have a layout of a circuit; we would like to define the planar topology of this circuit as the set of layouts of this circuit which can be obtained from each other by the operations of sliding the modules and input/output pins in the plane as well as stretching the nets and reshaping the nets in a manner which always maintains the net-connectivity. To accomplish this, we define a *planar circuit* to be an embedding of a planar graph which hides the crossing and branch points of the nets of the layout inside R-nodes. Using the results of the last section, the planar topology of the embedding can be specified by a sequence of pins for each B and R-node, corresponding to the clockwise cyclical ordering of edges around this node. In Section 2.4 we show how to *represent* the planar topology of the layout of a circuit as a planar circuit. Operations on these

planar circuits will be introduced in the next section, to simulate the actions of nets sliding around and being reshaped as well as to provide an equivalence among the planar circuits which represent the same layout. The planar topology of a layout will then be defined as the equivalence class of planar circuits under these operations which contains a planar circuit *representing* the layout. We begin by defining planar circuits.

Definition 2.12

A *planar circuit* is a five-tuple $A = (P, IO, B, R, W)$ where P is a set of pins, IO is a sequence of pins, and B and R are sets of B and R-nodes respectively. Each B and R-node is a sequence of pins, and W is a pairing of the elements of P such that,

a. Each pin appears exactly once in IO, B or R.

b. The set of pins of each R-node, $u = \{u_i \mid i=1,\ldots,m_u\}$ is partitioned into disjoint subsets of size at least two, $P_u = \{P_{u,j} \mid j=1,\ldots,c_u\}$ such that $|P_{u,j}| > 1$ for $1 \le j \le c_u$. These partitions correspond to sets of pins which are connected within the R-node.

c. The *plane graph* of A, $G_A = (IO \cup B \cup R, E)$ where

$$E = \{e_p = (u,v) \mid \text{for each } (u_i, v_j) \text{ in } W\} \cup \{(io_i, io_{i+1}) \mid 1 \le i < n\} \cup \{(io_n, io_1)\}$$

is connected and can be embedded in the plane such that the clockwise cyclical order of edges around each node agrees with the sequence of pins of the corresponding B or R-node, and the clockwise traversal of the exterior window corresponds to IO.

d. The net-connectivity graph, $G_N = (P \cup (\bigcup_{u,j} P_{u,j}), W \cup W')$, where W'

contains an edge from each pin of an R-node to the partition of the R-node which contains the pin, $\bigcup_{u \in R} \{(u_i, P_{u,j}) \mid u_i \in P_{u,j}\}$, is acyclic.

e. Every pin is circuit-connected to at least one IO pin. That is, if we add edges between pins belonging to the same element of B to the net connectivity graph,

$$\bigcup_{b \in B} \{(b_i, b_j) \mid \text{for each } i \neq j \text{ where } b = b_1 \cdots b_m\} ,$$

then each pin is connected to at least one element of IO.

Figure 2.4 contains a layout of a circuit and a planar circuit representing it. R-nodes will be represented by circles or ellipses while B-nodes are represented by boxes.

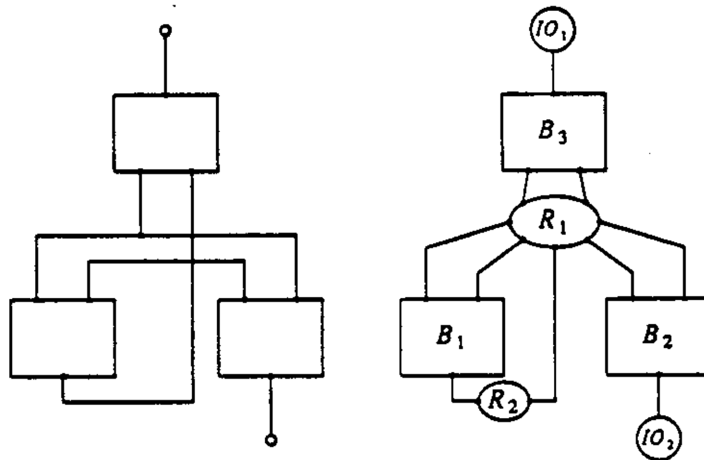


Figure 2.4 A layout and a planar circuit representing it.

According to the last section, the planar topology of the embedding of the plane graph is specified uniquely by the cyclic ordering of the elements of B and R and the traversal of the exterior window. The last three conditions are referred to as

the *embedding, acyclic nets* and *reachability* conditions. Note that the reachability condition implies the existence of at least one element in IO if P is non-empty and the connectivity of the plane graph, since all IO pins are connected by edges in the plane graph. Pins in IO are referred to as IO-nodes or IO-pins. In the sequence of pins of a B-node or a R-node, since only the cyclic order is important, for ease of notation all subscripts are understood to be modulo the number of pins in the sequence.

Definition 2.13

A *trivial node* of a planar circuit is an R-node with two pins.

By definition the two pins of a trivial R-node must be in the same partition of the R-node and hence, are connected inside the R-node. In Figure 2.4 the lower R-node is a trivial R-node. Nodes can have self-loops. A self-loop of a node forms a closed curve with its node which divides the plane into two connected components. Since the graph of a planar circuit is connected and its embedding in the plane is topologically unique, the interior of a self-loop can be defined as the component of the plane which does not contain the point at infinity. Thus we can refer to the interior of a self-loop of a node.

Definition 2.14

A *trivial self-loop* of an R-node is a self-loop whose interior does not contain any other pins.

Definition 2.15

An *isomorphism* of planar circuits $A=(P,IO,B,R,W)$ and $A'=(P',IO',B',R',W')$ is a bijection $f_P:P \rightarrow P'$ such that if we extend f_P to sequences of pins, by defining $f_P((a_1, \ldots, a_n))$ to be $(f_P(a_1), \ldots, f_P(a_n))$,

then

a. $f_P(IO) = IO'$.

b. $u \in B,R$ if and only if $f_P(u) \in B',R'$ respectively.

c. $p,q \in P$ are in the same partition of an element of R if and only if $f_P(p)$ and $f_P(q)$ are in the same partition of an element of R'.

d. $(p,q) \in W$ if and only if $(f_P(p),f_P(q)) \in W'$.

An isomorphism of planar circuits is simply a renaming of the pins; it represents the trivial form of equivalence. Isomorphic planar circuits will be referred to as being the same.

2.3 Operations on Planar Circuits

Seven kinds of operations on planar circuits will be defined. Operations are defined for a specific planar circuit and operate on a specific set of R-nodes of that planar circuit. In the following sections, we will show that there is a unique 'maximal' planar circuit in each equivalence class modulo the last of these operations, refoldings.

Merges

Any two R-nodes which are connected by one or more wires can be merged along those wires. Figure 2.5 contains an example of a merge. A merge of $u = u_1, \ldots, u_n$ and $v = v_1, \ldots, v_m$ is specified by $M(u,v,i,j,h)$ where $1 \le i \le n$, $1 \le j \le m$, $1 \le h \le min(n,m)$ and $(u_{i+k}, v_{j+h-k}) \in W$ for $1 \le k \le h$. The result of

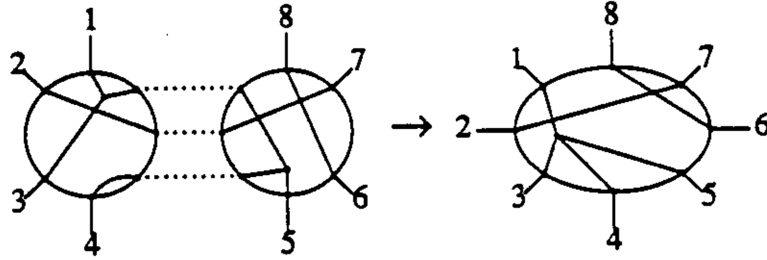


Figure 2.5 A merge of two R-nodes.

the merge is a new planar circuit which is obtained by removing u and v and adding the node z,

$$z = \begin{cases} v_{j+h+1},\ldots,v_{j-1},u_{i+h+1},\ldots,u_{i-1} & \text{for } h<min(n,m) \\ v_{j+h+1},\ldots,v_{j-1} & \text{for } h=n<m \\ u_{i+h+1},\ldots,u_{i-1} & \text{for } h=m<n \end{cases}$$

as well as removing the wires, (u_{i+k},v_{j+h-k}) for $0\leq k\leq h$. These wires are said to be *subsumed* by z since the connectivity they represent will be represented internally, by the partitions of z. The partitions of z are obtained from those of u and v by merging the partitions containing u_{i+k} and v_{j+h-k} for each $1\leq k\leq h$ and then removing these pins. Note that the case $h=n=m$ has been omitted; this case will never occur since by assumption the planar circuit's plane graph is connected and has at least one IO-node, if P is non-empty.

$M(u,v,i,j,h)$ takes (P,IO,B,R,W) to (P',IO,B,R',W') where

$$P' = P - \{u_{i+k} \mid 1\leq k\leq h\} - \{v_{j+h-k} \mid 1\leq k\leq h\}$$

$$R' = R - \{u,v\} \cup \{z\}$$

$$W' = W - \{(u_{i+k},v_{j+h-k}) \mid 1\leq k\leq h\}$$

A merge is *complete* if the set of wires it subsumes, is a maximal set of wires: if neither (u_{i-1},v_{j+h+1}) nor (v_{j-1},u_{i+h+1}) are elements of W. The significant characteristic of a complete merge is that it does not produce any new trivial self-loops.

Clean Divides

Cleanly dividing an R-node results in two new R-nodes with no wires between them such that the net-connectivity is maintained. This operation is only possible when the sequence of pins of an R-node can be divided so that no partition is represent on both sides of the division. Determining whether an R-node can be cleanly divided will be discussed in a Section 2.8. Figure 2.6 contains an example of a clean divide.

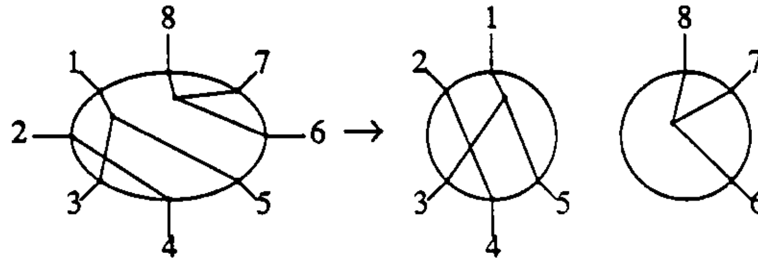


Figure 2.6 A clean divide of an R-node.

Formally suppose there is an R-node, $z = z_1, \ldots, z_m$ such that there exists $1 \le j \le m$ and $1 < h < m-1$ such that no partition of z contains a pin from both $z_j, \ldots, z_{j+h}$ and $z_{j+h+1}, \ldots, z_{j-1}$. Then $C(z,j,h)$ takes (P,IO,B,R,W) to (P,IO,B,R',W) where

$$R' = R \cup \{u,v\} - \{z\}$$

and $u = z_j, \ldots, z_{j+h}$ and $v = z_{j+h+1}, \ldots, z_{j-1}$. Note that the reachability condition of planar circuits insures that the clean divide will not disconnect the plane graph of the planar circuit. This condition insures that each pin is circuit-connected to an IO-node and since all of the IO-nodes are connected in the plane graph, the plane graph

cannot become disconnected since the net-connectivity of the planar circuit is maintained.

Glues

Gluing two R-nodes is similar to merging but no wires are subsumed. It is the only operation whose applicability depends not only on the nodes it is applied to, but the planar circuit they sit in. In order to glue two R-nodes, they must be on the same window of the plane graph of the planar circuit. Since the only nodes on the exterior are IO-nodes there are never any glues along the external window. A glue is more easily defined as the inverse of a clean divide.

$G(u,v,i,k)$ takes (P,IO,B,R,W) to (P,IO,B,R',W) if there exists C(z,j,h) which takes (P,IO,B,R',W) to (P,IO,B,R,W) such that $z = z_1,..,z_m$ and and $u = u_i, \ldots, u_{i-1} = z_j, \ldots, z_{j+h}$ and $v = v_k, \ldots, v_{k-1} = z_{j+h+1}, \ldots, z_{j-1}$.

Unclean Divides

Unlike clean divides, an unclean divide must create some new wires between the two nodes it creates in order to maintain the net-connectivity of the planar circuit. It is more easily defined as the inverse of a merge.

An unclean divide $U(z,k,l)$ takes (P,IO,B,R,W) to (P',IO,B,R',W') if there exists $M(u,v,i,j,h)$ which takes (P',IO,B,R',W') to (P,IO,B,R,W) where

$$z = z_1, \ldots, z_m$$

and

$$z_{k+1}, \ldots, z_{k+l} = v_{j+h+1}, \ldots, v_{j-1} \text{ for } 0 < l \leq m$$

$$z_{k+l+1}, \ldots, z_k = u_{i+h+1}, \ldots, u_{i-1} \text{ for } 0 =< l < m$$

Insertions

A trivial node can be inserted along any wire of the planar circuit. This operation will be used to remove trivial self-loops, by inserting a trivial node on the self-loop and then merging this trivial node with the node to which the self-loop belonged. Figure 2.7 contains an example of an insertion.

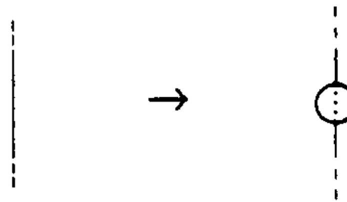


Figure 2.7 The insertion of a trivial R-node.

If $w = (p,q) \in W$ then $I(w)$ takes (P,IO,B,R,W) to (P',IO,B,R',W'),

$$P' = P \cup \{p',q'\}$$

$$R' = R \cup \{u\}$$

where u is a new trivial node with pins p' and q' and,

$$W = W \cup \{(p,p'),(q,q')\} - \{(p,q)\}$$

No nodes are introduced on the external face of the plane graph since the edges which connect the IO-nodes in the plane graph are not wires.

Removals

Any trivial node can be removed, leaving behind a wire. If z is a trivial node $R(z)$ takes (P,IO,B,R,W) to (P',IO,B,R',W') if there exists $I(w)$ taking (P',IO,B,R',W') to (P,IO,B,R,W).

Refoldings

A refolding of a node is in fact two operations, a divide (clean or unclean) of a node followed by a merge of the two nodes created by the divide. If the divide is clean then the original node must have had some self-loops since otherwise no merge could take place. A refolding can be the identity operation if the merge takes place along exactly those wires created by the divide. If the divide is unclean then the inverse of a refolding is a refolding; otherwise it is an unclean divide followed by a glue. The reason for considering refoldings as single operations rather than as the composite of two operations will be explained in Section 2.5.

It is clear that these seven operations preserve the net-connectivity of the circuit represented by the planar circuit and do not introduce any new nets. Applying one of these operations to a planar circuit uniquely defines a planar circuit. If A is a planar circuit and s is an operation on A, then (A,s) is the planar circuit obtained by applying s to A. If we have a sequence of operations $S = s_1, \ldots, s_n$ and planar circuits $A_1, \ldots, A_{n+1}$ such that $A_{i+1} = (A_i, s_i)$ for $1 \le i \le n$, then we use (A_1, S) to represent A_{n+1}. The operations defined above characterize the notion of topological equivalence among planar circuits. They form a group since the inverse of an operation is also an operation (or two in the case of refoldings).

Definition 2.16

Two planar circuits, A and A', are *homeomorphic* if there exists a sequence of operations S such that A is isomorphic to (A', S).

Homeomorphic planar circuits form equivalence classes.

2.4 Representation of Layouts by Planar Circuits

This section defines the relation between layouts and planar circuits. We define what it means for a planar circuit to *cover* a layout and show that any two planar circuits which *cover* the same layout are homeomorphic. We also discuss how the operations on planar circuits correspond to changes in the layout.

We define the layout of a circuit as follows. The layout of a circuit is an assignment of coordinates in the plane to the modules and an implementation of the nets as trees such that,

1. Modules do not touch or overlap.

2. I/O pins do not touch nor overlap with each other and the modules.

3. Each net coincides with the modules and the I/O pins in exactly the set of points which correspond to the pins in its net-list.

4. Each connected set of points onto which more than one net has been mapped is contractible to a point in the space obtained by removing the interiors of the modules from the plane.

Suppose we have a layout of a circuit. We can *cover* that layout with a planar circuit as follows. Assign nodes of type IO and B to the elements representing the inputs/outputs and modules. Select a set of disjoint simple closed curves which do not intersect any of the inputs/outputs or modules and such that each crossing and branching point of the nets is inside one of these curves. These will be the R-nodes. One way of doing this is by drawing a sufficiently small circle around each crossing and/or branching; in the case where nets cross at more than one point the circles are

joined together. This particular representation is called the 'primitive representation' since the R-nodes are as small as possible. Each connection of a net to an input, output and module is assigned a pin as well as each intersection of a net with the curve of an R-node. The pins of the R and B-nodes are ordered by clockwise traversal of the curves and modules respectively. Clearly the intersection of the nets with the curves and their exteriors correspond to disjoint simple paths whose endpoints are the pins. We assign a wire to each such path by giving its pins. The partitions of the R-nodes are based on the connectivity of the nets only within the R-node; that is, two pins belonging to the same net but not joined by a branch inside an R-node are not in the same partition. The planar circuit obtained in this manner is said to *cover* the layout. It is always possible to obtain such a covering; the fourth requirement of a layout guarantees that the curves corresponding to R-nodes can be found. Figure 2.8 contains a layout and its primitive representation.

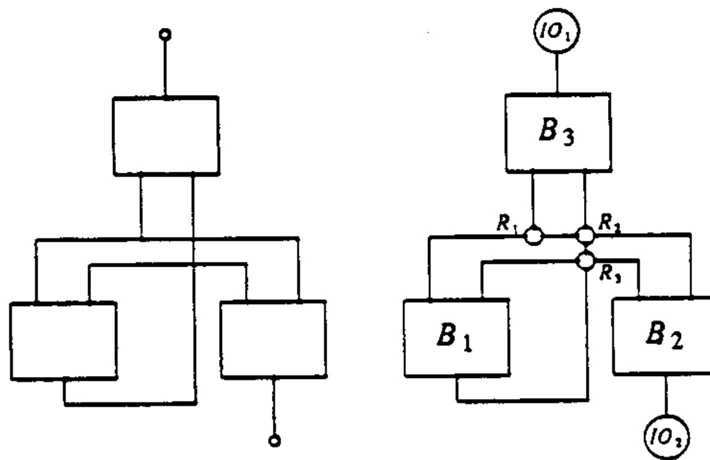


Figure 2.8 A layout of a circuit and its primitive representation.

There is more than one planar circuit which covers a layout. Figures 2.4 and 2.8 contain two different planar circuits covering the same layout. However, it can

be shown that if two planar circuits cover the same layout then they are homeomorphic. This is achieved by showing that any representation of a layout by a planar circuit can be transformed into the primitive representation by applying the operations described in the last section.

Proposition 2.17 Two planar circuits which cover the same layout are homeomorphic.

Proof: Suppose $A_p = (P_p, IO_p, B_p, R_p, W_p)$ is the primitive representation and $A = (P, IO, B, R, W)$ is any other representation of the same layout. Clearly there is a one-to-one correspondence between elements of IO_p and IO, and, B_p and B; we must show how we can transform the R-nodes of A into those of A_p by operations. A node of R, u corresponds to a simple closed curve. If there are no crossing or branching points of the layout inside this curve then A_p does not have any R-nodes corresponding to u. By repeatedly performing clean divides on u until only trivial nodes remain and then removing them we can transform A into a planar circuit which covers the layout in the same manner except without the R-node, u. Repeating this for each R-node of A which does not contain any crossings or branchings we obtain a planar circuit, A' which is homeomorphic to A such that each of its R-nodes contains at least one branching and crossing. Since homeomorphism is an equivalence relation it suffices to show that A' and A_p are homeomorphic.

Since A_p is the primitive representation, we can assume that each R-node of A' contains the R-nodes of A which cover the crossing and branch points inside it. Suppose an R-node, u, of A' contains exactly one R-node of A_p, u_p. There are no crossing or branch points in the difference of their interiors. If u_p has a trivial self-loop which is interior to u then it can be removed by performing an insertion on it

and merging it into u_p. Note that if u_p has any self-loops which are interior to u then at least one must be trivial. By removing these trivial self-loops, u_p will eventually have no self-loops which are interior to u. In this case, the sequence of pins around u and u_p correspond to the same wires; u and u_p are the same up to relabeling of the pins.

Suppose now that the R-node of A', u, contains more than one R-node of A_p. We can reduce this case to the previous case as follows. Take any two of these R-nodes of A_p which share a window inside u and glue them together. This is the same as drawing a curve around them which is still inside u and does not entirely contain any path. If there is no such pair which can be glued together, then there are one or more wires separating them. In this case insert trivial nodes on these wires, glue these trivial nodes together and then glue this node to both of the R-nodes. The number of R-nodes is reduced by one and hence, by repeatedly performing these glues, we will end up with one R-node inside of u. By repeating this process for each R-node of A' we will eventually transform A_p into A'. It follows that A', A_p and A are homeomorphic. Thus any two planar circuits covering the same layout are homeomorphic to the primitive representation and hence to each other.

□

The operations which are defined on planar circuits not only allow us to associate the planar circuits which cover the same layout, but also provide the means to model the structural changes in the layout which occur as modules and nets are moved around and stretched while still remaining in the plane. We examine how structural changes in the layout can be simulated by operations on the planar circuits which cover them.

A wire meets another wire. We introduce trivial nodes on each wire and then glue the nodes together.

Figure 2.10 Two wires meet.

A wire meets a branch/crossing. In this case we introduce a trivial node and then again glue the two nodes together.

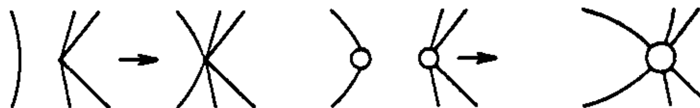


Figure 2.11 A wire meets a branch/crossing.

A self-loop disappears. This is accomplished in the representation by introducing a trivial node on the loop and then merging with it.

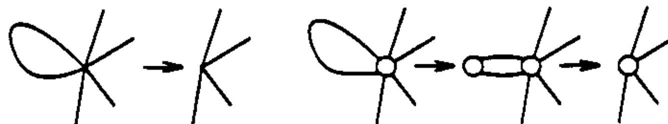


Figure 2.12 A self-loop disappears.

Several branch/crossings meet. This can be accomplished by merges. Clearly the order of the merges is unimportant since this operation can be hidden within an R-node; this is equivalent to changing the representation of the layout.

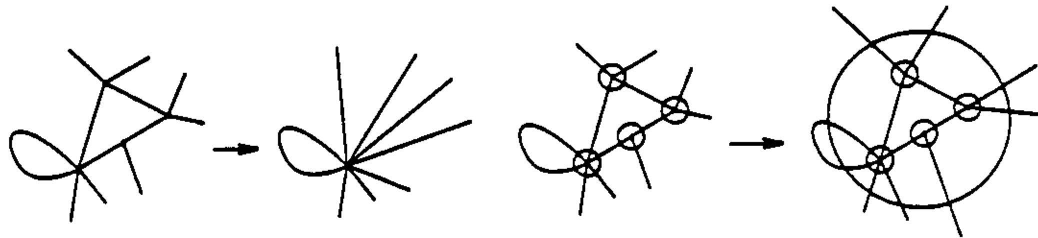


Figure 2.13 Several branch/crossings meet.

Since the changes to the layout allowed by the planar movement of the modules and nets can be simulated by operations on the planar circuits covering the layout, the planar topology of the layout will be defined as follows.

Definition 2.18

The *planar topology* of a layout is the class of homeomorphic planar circuits which contains its coverings.

2.5 Unique Representation of Planar Topology

As we have seen the representation of a layout as a planar circuit is not unique. To select the best layout with a given planar topology, we would need to consider all planar circuits in its class of homeomorphic planar circuits. The number of homeomorphic planar circuits is in general too large to make this practical. However if we limit the planar circuits by requiring their R-nodes to have certain properties, we can show that the planar circuits within each class of homeomorphic planar circuits possessing these properties are isomorphic modulo refoldings. This gives us a one-to-one correspondence between the R-nodes of these planar circuits. In Section 2.8 we will show that a planar circuit possessing these properties is optimal within its class of planar circuits given some assumptions about the layout

procedure. These properties provide a normal form for planar circuits.

Definition 2.19

An R-node is *indivisible* if it cannot be cleanly divided even after refoldings.

Definition 2.20

An R-node is *maximal* if it does not have a wire connected to another R-node, is not trivial and does not have any trivial self-loops.

Definition 2.21

A planar circuit is *maximal, indivisible, maximal-indivisible* if all its R-nodes are maximal, indivisible, maximal and indivisible respectively.

We will show that any two homeomorphic maximal-indivisible planar circuits can be transformed into each other by refoldings. The proof will consist of showing that in transforming a planar circuit into a maximal-indivisible one, unclean divides and glues are not necessary and the remaining operations can be rearranged in a particular order. In this section we show for each sequence of operations without glues and unclean divides there is another sequence resulting in the same planar circuit in which all merges occur before any clean divide. In Section 2.6 we will show we can obtain an equivalent sequence without the last unclean divide. We will do the same for glues in Section 2.7. We will have shown that a planar circuit which is homeomorphic to a maximal-indivisible planar circuit, can be transformed to it by a sequence in a particular form (Theorem 2.48). From this result, we will show that two homeomorphic maximal-indivisible planar circuits can be transformed to one another using only refoldings (Theorem 2.49).

The uniqueness can only be up to refoldings since an *R*-node can never completely surround a *B*-node of the planar circuit; if it were to do so then the information as to relative position of the wires internal to the R-node with respect to *B*-node would be lost. Figure 2.13 contains a layout which can be covered by either the planar circuit in Figure 2.14 or Figure 2.15.

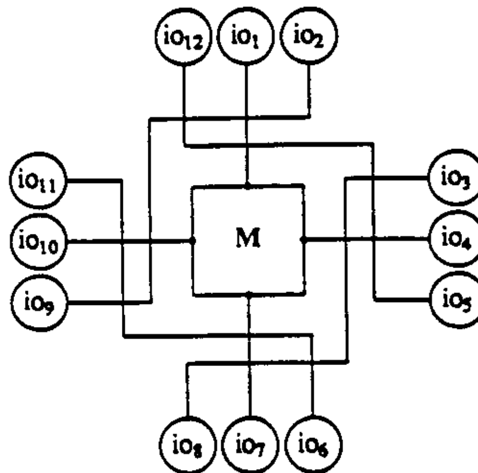


Figure 2.13

The following notation will be used. We associate with each operation, s, a symbol $t(s)$, according to its type. The symbols are,

M for a merge	*I* for an insertion of a trivial node
G for a glue	*R* for a removal of a trivial node
C for a clean divide	*Re* for a refolding of a node
U for an unclean divide	*Rec* for a refolding of a node whose divide is clean

For any sequence of operations, $S = \{s_i\}_{i=1}^{n}$ let $t(S) = t(s_1) \cdots t(s_n)$. If L is a subset

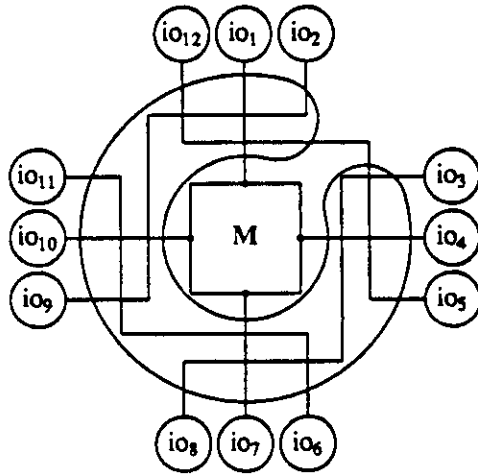


Figure 2.14

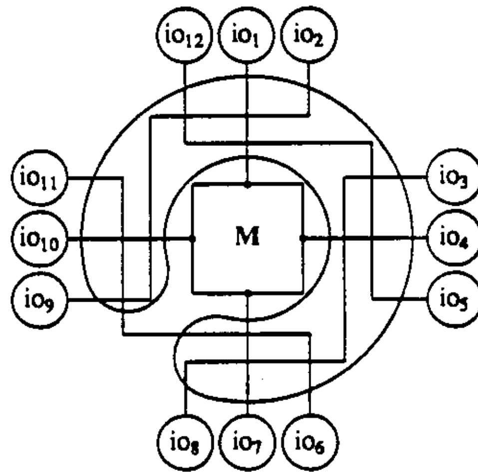


Figure 2.15

of Σ^* where Σ is the alphabet of symbols $\{M, G, C, U, I, R, Re, Rec\}$, then a sequence $S = \{S_i\}_{i=1}^{n} \in L$, if $t(S) \in L$. If S and S' are two sequences, the sequence of operations consisting of S followed by S' is the concatenation of S with S' and is denoted by SS'.

As discussed, the uniqueness of maximal-indivisible planar circuits will be proven by showing that a sequence leading to a maximal-indivisible planar circuit can be altered to remove the last operation which is a glue or unclean divide. This will be shown in the next two sections. It is necessary first to show that a sequence consisting of the other types of operations can be rearranged so that all merges occur before clean divides and refoldings.

Lemma 2.22: A refolding operation which produces a trivial node can be replaced by a sequence of insertions and merges to obtain the same planar circuit.

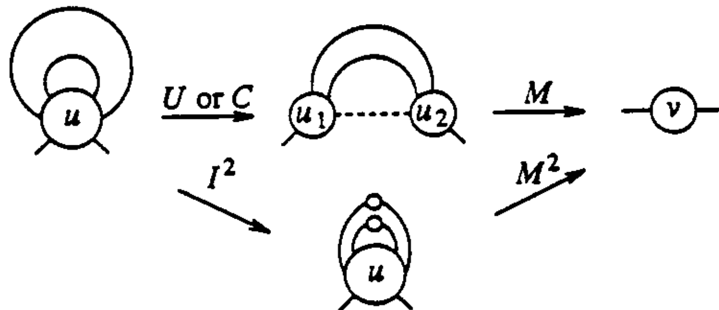


Figure 2.17 Replacing a refolding into a trivial node by Insertion and Merges

Proof: Suppose we have a refolding which is applied to a node u of a planar circuit and results in a trivial node v. This refolding consists of a divide followed by a merge. Suppose u_1 and u_2 are the two nodes produced by the divide. The upper portion of Figure 2.17 depicts this situation. The dashed line between u_1 and u_2 exists if the divide of u is unclean. Since merging u_1 and u_2 produces a trivial node, these two nodes must both have all but one pin connected to each other. The wires subsumed in the merge must have existed as trivial self-loops of u or have been created by the divide. Thus u must have exactly two pins connected to other node(s) and possibly some trivial self-loops. If u is trivial then the refolding is the identity

operation and can be omitted. Otherwise the pins of u consist exactly of the two that will belong to v and pins belonging to trivial self-loops. By removing these self-loops, we can also obtain v. This can be accomplished by inserting trivial nodes along these self-loops and then merging them with u. In Figure 2.17 u has two loops and hence two insertions and two merges are required.

□

We can thus assume with out loss of generality that refoldings do not produce trivial nodes. In the following lemma we show that a clean divide can either be moved past the next merge in the sequence or combines with it to form a clean refolding, if there are only insertions and removals between the clean divide and the merge.

Lemma 2.23: Given (A,S) with $S \in C(I+R)^*M$, there exists $S' \in (I+R+M)^*(\lambda+C+Rec)(I+R)^*$, such that $(A,S) \equiv (A,S')$.

Proof: We establish this lemma by induction on the length of S, n.

Basis: (n=2). In this case, s_1 is the clean divide of node z into nodes u and v and s_2 is a merge. There are three cases to consider depending on whether the nodes generated by the s_1 are part of the merge, s_2. If s_2 is a merge of two other nodes q and p then clearly $(A,s_1s_2) \equiv (A,s_2s_1)$ so $S' = s_2s_1$ satisfies the lemma. Suppose that s_2 merges u with a node, p, which is not v to get z'. In this case merging z and p along the same pins and then cleanly dividing the result into z' and v also takes A into (A,S) and satisfies the lemma. The last case to consider is if s_2 merges u and v. This can only happen if u and v have wires in common which means they were originally self-loops of z. In this case s_1s_2 constitute a clean refolding of z. If the result of s_2 is a trivial node, then we replace s_1s_2 by insertions and merges as described in the previous lemma.

Induction: ($n>2$). Suppose we have shown the result for all sequences of length less than $n>2$. Since a trivial node has only one partition, it cannot be cleanly divided. Let s_k be the first insertion of a trivial node. Then $k>1$ and if s_k inserts a trivial node on wire w which is not generated by any of the removals of trivial nodes $s_2 \cdots s_{k-1}$ then s_k can be moved ahead of s_1 and the induction hypothesis can be applied to $s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n$ to obtain S'. $s_k S'$ is then the required sequence. Suppose on the other hand, that there is a removal of a trivial node, s_j, which generates w by removing the node x. In this case we can drop both s_j and s_k since removing and then inserting a trivial node along the same wire produces the same planar circuit. We obtain a sequence of length $n-2$ to which the induction hypothesis can be applied.

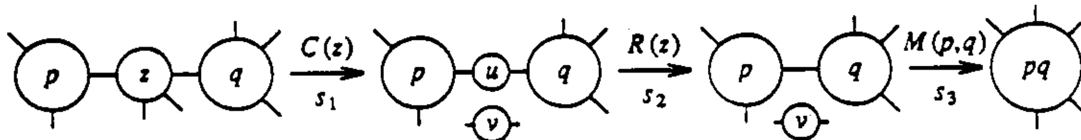


Figure 2.18

This leaves the case in which there are no insertions in S, $S \in C(R)^*M$. If one of the removals is of a trivial node not created by s_1 then it can be moved to the front of the sequence and the induction hypothesis can be applied to the subsequence starting at the second operation to get the desired sequence. So we assume that the at most two removals between s_1 and s_n are of nodes created by s_1. Consider swapping these removals with s_n. If this can be done, then we can again apply the induction hypothesis to a shorter sequence. The only obstacle to this swap is if the removal creates one of the wires subsumed in the merge s_n. So the situation, as illustrated in Figure 2.18, is reduced to the following: s_1 cleanly divides z into two nodes u and v

of which at least one is trivial, say u. The removal of node u generates a wire which is subsumed in the merge of p and q. It is also possible that node v is trivial, possibly gets removed and possibly generates a wire subsumed by the merge, although this is not the case depicted in Figure 2.18. We can achieve the same effect by merging p with z and merging the result with q and then cleanly dividing this node to obtain v if necessary; we obtain a sequence satisfying the lemma.

□

We now show that a refolding can be moved past insertions and removals followed by a merge.

Lemma 2.24 Given (A,S) with $S \in Re(I+R)^*M$ there exists $S' \in (I+R+M)^*(Re+\lambda)(I+R)^*$ such that $(A,S) \equiv (A,S')$.

Proof: The proof proceeds by induction on the length of the sequence.

Basis: $(n=2)$. s_1 is the refolding of node z which is achieved by dividing z into nodes u and v and then merging them along a different set of wires then the one possibly created by the divide, resulting in z'. s_2 is a merge. If s_2 is a merge of two nodes, q and p, neither of which is z, then the sequence $(A,s_1s_2) = (A,s_2s_1)$ and $S' = s_2s_1$ satisfies the lemma. So assume s_2 merges z' with another node p to get z''. If the pins along which z' is merged with p are adjacent pins either belonging only to u or only to v, then we can substitute a merge of z with p along these pins followed by a refolding, for S. If this is not the case the situation is somewhat more difficult: the refolding consists of a divide $s_{1,1}$ of z into u and v followed by a merge $s_{1,2}$ of u and v such that before this merge some of the pins along which z' and p are merged are from u and some are from v or they are not adjacent. We have one of the two situations depicted in Figure 2.19. The dashed lines indicate that in the first case u

may or may not have wires connected to p. If so then these are the wires along which it is merged to p.

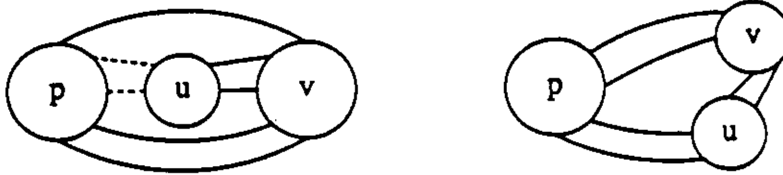


Figure 2.19

Suppose first that these dashed lines do in fact exist as wires. Then in both cases u and p are connected by wires which will be subsumed by the merge of p and z'. In either case, merging p and u along the subset of the subsumed wires that they share and then merging the result with v along the remaining wires to be subsumed results in the same node as $s_1 s_2$ would produce. Thus we have replaced $s_1 s_2$ by $s_{1,1} M(u,p,...) M(up,v,...)$. Now consider swapping the first two operations in this sequence. Clearly the same thing can be accomplished by first merging z with p and then dividing it into up and v. We thus obtain $M(z,p,...) U(up,v) M(up,v...) \in MRe$ which satisfies the lemma. Note that if the divide of the refolding was originally clean then it will remain so. If the new refolding results in a trivial node we replace it by insertions and merges.

If the dashed lines do not exist, then u shares all of its wires with v. The refolding of z must either be the identity or removes trivial self-loops of z. In the former case we can drop the refolding while in the latter case we replace it by insertions and merges.

Induction: ($n > 2$). Suppose we have shown the result for all sequences of length less than n. If s_k for $1 < i < n$ is the removal of a trivial node then either it can be moved ahead of s_1 or it cancels some s_j for $1 < j < k$ which is the insertion of a trivial node.

In the former case, the induction hypothesis can be used on $s_1 s_3 \cdots s_n$ to get a sequence S' such that $s_k S'$ satisfies the lemma. In the latter case applying induction to the sequence obtained by omitting s_k and s_j gives a sequence which also satisfies the lemma. It is not possible for a refolding to result in a trivial node by assumption. Suppose now $S \in Re(I)^*M$; there are no removals. If any insertion is of a trivial node on a wire not created by s_1 then it can be moved to the front of the sequence and the induction hypothesis can be applied to the subsequence starting at the second operation yielding the desired sequence. Similarly if the insertion does not create a node involved in the merge, s_n, then it can be moved to the end of the sequence and induction applied to the subsequence preceding it. Suppose the insertion is involved in the merge. Merging a node with a trivial node along one wire is equivalent to removing it and hence we can simply omit the insertion and merge to obtain the desired sequence. The remaining case consists of a refolding which creates a trivial self-loop on which a trivial node is inserted and then merged along both of its wires to remove this trivial self-loop. In this case the divide of the refolding can be altered so that the trivial self-loop is not created and then the insertion and merge can be omitted.

□

By applying the previous two lemmas we can now move a merge ahead of all clean divides and refoldings.

Corollary 2.25: Given (A,S) with $S \in (Re+C+I+R)^*M$, there exists $S' \in (I+R+M)^*(Re+C+I+R)^*$, having no more refoldings and clean divides than S such that $(A,S) \equiv (A,S')$.

Proof: This is by induction on the number of refoldings and clean divides in the

sequence. If there are none, the original sequence satisfies the corollary. So assume the lemma is true for sequences with less than ($k>0$) of these two types of operations. Consider the last such operation. If it is a clean divide apply Lemma 2.23 and if it is a refolding apply Lemma 2.24. A sequence in $(Re+C+I+R)^*(I+R+M)^*(Re+C+\lambda)(I+R)^*$ is obtained such that the first component has one less refolding or clean divide. We can then apply the induction hypothesis repeatedly to move the merges one by one ahead of the refoldings and clean divides. This is possible since application of the induction hypothesis does not increase the number of refoldings and clean divides.

□

Corollary 2.26 Given (A,S) with $S \in Re(I+R+M)^*$ there exists $S' \in (I+R+M)^*(Re+\lambda)(I+R)^*$ such that $(A,S) \equiv (A,S')$.

Proof: The argument is the same as for the previous corollary with the added observation that when there are no clean divides in the original sequence, none are ever introduced in the process of transforming the sequence, and hence the final sequence can not contain any.

□

The goal of this section is the following.

Corollary 2.27: Given (A,S) with $S \in (Re+I+R+M+C)^*$, there exists $S' \in (I+R+M)^*(Re+I+R+C)^*$ such that $(A,S) \equiv (A,S')$.

Proof: This corollary is obtained by induction on the number of merges. If there are no merges then the corollary is satisfied by the original sequence, so assume that it is true for sequences with less than $n>0$ merges. Suppose the sequence has n merges, the first of which is s_c. Apply the previous corollary to $s_1 \cdots s_c$ to get $s'_1 \cdots s'_m$ in

which each clean divide and refolding occurs after all the merges. Suppose s'_h is the last merge. Then we can apply induction to $s'_{h+1} \cdots s'_m s_{c+1} \cdots s_n$ since it has $n-1$ merges. The corollary is then satisfied by appending this new sequence to $s'_1 \cdots s'_h$.

□

Before tackling the glues and unclean divides, we show that insertions are not necessary at the end of a sequence leading to a maximal-indivisible planar circuit.

Lemma 2.28 Given (A,S) which is maximal-indivisible with $S \in I(C+Re+R)^*$ there exists a proper subsequence of S, S', such that $S' \in (C+R+Re)^*$ such that $(A,S) \equiv (A,S')$

Proof: Consider what happens to the trivial node, u, created by $s_1 \in I$. u has two pins, u_1 and u_2 which are connected by wires, either to two other nodes or to one node (if u was inserted on a self-loop). Let us call a node, belonging to one of the planar circuits defined by the sequence of operations, a *descendant* of u if it is created by an operation performed on u or one of its descendants. Clean divides, refoldings and removals of trivial nodes are all operations which are applied to individual nodes and whose applicability depends only on the properties of the node and in the case of refoldings, the existence of its self-loops. Suppose we consider the result of all of these operations performed on u and its descendants. The operations performed on the other nodes can all be carried out independently of the operations on descendants of u with one exception, a refolding whose merge is along a wire which is created by removing a descendant of u. In this case, u and its descendants must have been removed leaving only the wire behind. Until such a refolding is encountered, operations on descendants of u are independent of operations on other nodes. We can consider them as occurring within a region which

originally contained only u and out of which only the two wires connected to u emerge.

If there is such a refolding in S, then all of the operations on u and its descendants occur before this refolding and result in one wire; the last operation on a descendant of u is a removal of a trivial node along this wire. In this case we can omit s_1 and all of the operations on u and its descendants. If there is no such refolding, then consider the final planar circuit. There are exactly two wires (or maybe two ends of the same wire) emerging from the region and these two wires are connected inside the region. Thus there can be at most one node inside the region since it must be maximal and the graph is connected. It has exactly two pins which connect it to other nodes outside the region. Any other pins would belong to self-loops and these would have to be trivial self-loops. Since the node is maximal it cannot have trivial self-loops and so it must have only two pins. But since a trivial node is not maximal there can be no node. Thus again, the last operation performed on a descendant of u must be a removal of a trivial node leaving behind the wire emerging from the region. By omitting the insertion and all operations on descendants of u we obtain a subsequence of S which results in the same maximal-indivisible planar circuit.

□

2.6 Removing Unclean Divides

In this section we show how a sequence resulting in a maximal-indivisible planar circuit which consists of an unclean divide followed only by merges, clean divides, insertions, removals and refoldings, can be replaced by a sequence with no glues nor unclean divides. An unclean divide which a creates trivial node can be

replaced by an insertion of a trivial node if only one new pin is created, or a refolding followed by an insertion if the trivial node contains two new pins. We will assume that an unclean divide of a node generates two non-trivial nodes and will show that this remains the case as the sequence is altered.

We first show that an unclean divide can be moved past a merge if there are only insertions and removals in between. The unclean divide may disappear or combine with the merge to form a refolding.

Lemma 2.29 Given (A,S) with $S \in U(I+R)^*M$ there exists $S' \in (I+R)^*(Re+MU+U+M^++\lambda)(I+R)^*$ such that $(A,S)\equiv(A,S')$.

Proof: The proof proceeds by induction on the length of the sequence n.

Basis: $(n=2)$. In this case s_1 is the unclean divide of node z into nodes u and v and s_2 is a merge. If s_2 is a merge of two nodes neither of which is u or v, then the sequence $(A,s_1s_2)=(A,s_2s_1)$ so $S'=s_2s_1$ satisfies the claim. Suppose that s_2 merges u with a node $p\neq v$ to get z'. In this case it is clear that merging z and p along the same pins and then dividing the result into z' and v also takes A into (A,S) and satisfies the claim. Figure 2.19 illustrates this case. s_1 is the unclean divide $U(z)$ and s_2 is the merge $M(u,p)$. Neither z' nor v can be trivial nodes so this new unclean divide satisfies the requirements. The last case to consider occurs if s_2 merges u and v. This either results in the original node z in which case we can omit both s_1 and s_2 from the sequence, or s_1s_2 constitute a refolding which also satisfies the lemma. If this refolding creates a trivial node then replace it by insertions and merges.

Induction: $(n>2)$. Suppose we have shown the result for all sequences of length less than n. If s_k, $1<k<n$ is the removal of a trivial node then either it can be moved

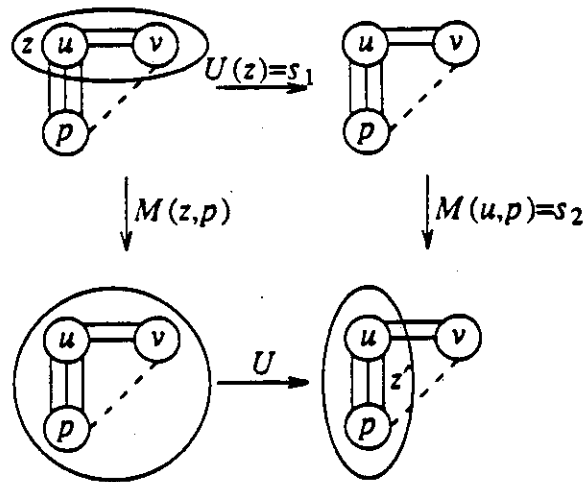


Figure 2.19

ahead of s_1 or it cancels some s_j which is the insertion of a trivial node, for $1<j<k$. In the former case, the induction hypothesis can be used on $s_1 \cdots s_{k-1}s_{k+1} \cdots s_n$ applied to (A,s_k) to get a sequence S' such that s_kS' satisfies the lemma. In the latter case applying induction to the subsequence obtained by omitting s_k and s_j gives a sequence which also satisfies the lemma.

Suppose now that there are no removals; $S \in U(I)^*M$. If there is an insertion on a wire not created by s_1 then it can be moved to the front of the sequence and the induction hypothesis can be applied to the subsequence starting at the second operation to get the desired sequence. Similarly if the insertion does not create a node involved in the merge s_n, then it can be moved to the end of the sequence and induction applied to the subsequence without it. So we assume that the at most two insertions between s_1 and s_n are on wire(s) created by s_1 and are involved in the merge s_n. If there are two of these insertions, then the two trivial nodes are merged and the two insertions and merge can be replaced by a single insertion since the result of the merge is also a trivial node along the same wire. If there is one

insertion, then the trivial node produced must be merged with either u or v; the insertion and merge cancel each other and can be dropped to obtain the desired sequence.

□

By applying this lemma repeatedly, an unclean divide can be moved past a sequence of merges, insertions and removals.

Corollary 2.30 Given (A,S) with $S \in U(I+R+M)^*$ there exists $S' \in (I+R+M)^*(Re+U+\lambda)(I+R)^*$ such that $(A,S)\equiv(A,S')$.

Proof: The proof is by induction on the number of merges in S. If S has no merges, then it satisfies the requirements. So assume the corollary has been shown for sequences with less than $n>0$ merges and $S=s_1 \cdots s_m$ has n merges. Suppose s_k is the first merge in the sequence. Applying Lemma 2.29 to $s_1 \cdots s_k$ gives a sequence $S'=s'_1 \cdots s'_{m'} \in (I+R)^*(Re+MU+U+M^++\lambda)(I+R)^*$. If S' has no unclean divides or refoldings in it, then $S's_{k+1} \cdots s_m$ satisfies the corollary. Otherwise suppose s'_h is an unclean divide. Note that in this case $S' \in (I+R)^*(MU+U)(I+R)^*$. Thus $s'_h \cdots s'_{m'}s_{k+1} \cdots s_m$ is in $U(I+R+M)^*$ and has $n-1$ merges. Applying the inductive hypothesis to this sequence gives S'' and then $s'_1 \cdots s'_{h-1}S''$ satisfies the corollary. If s'_h is a refolding then applying Corollary 2.26 to $s'_h \cdots s'_{m'}s_{k+1} \cdots s_m$ also gives a sequence S'' such that $s'_1 \cdots s'_{h-1}S''$ satisfies the corollary.

□

We now show that we can remove the last unclean divide if there are no merges or glues succeeding it and the sequence results in a maximal-indivisible planar circuit.

Lemma 2.31 Given (A,S) which is a maximal-indivisible planar circuit with $S \in (U+\lambda)(C+I+Re+R)^*$ then either $S \in (R+C+Re)^*$ or there exists

$S' \in (R+C+Re)^*$ such that $(A,S)\equiv(A,S')$ and such that the length of S' is less than the length of S and has no more refoldings than S.

Proof: If $S \in (C+I+Re+R)^*$ then by applying Lemma 2.28 repeatedly, we can remove operations to obtain a subsequence of S, $S' \in (R+C+Re)^*$ such that $(A,S)\equiv(A,S')$. So we need only consider the case where $S \in U(C+I+Re+R)^*$. To facilitate the proof we relax the requirement that an unclean divide not create trivial nodes. By pushing the unclean divide to the end of the sequence it may be transformed so that it generates a trivial node. Since the final sequence will not have any unclean divides, relaxing this assumption has no effect on other results. No result which depended on this assumption is used in the proof.

The proof is by induction on n, the length of S. If $n=1$ then we are done since an unclean divide would create two non-maximal nodes. Suppose we have shown the lemma for all sequences of length less than $n>1$. If S contains an insertion of a trivial node then apply the Lemma 2.28 to the subsequence beginning with the last insertion and obtain a proper subsequence which can replace it in S. Applying induction to this new shorter sequence gives us the desired sequence. So assume $S \in U(C+Re+R)^*$ and hence s_1 uncleanly divides z into nodes u and v and consider s_2. If it is an operation on a node other than u or v then it can be swapped with s_1 and induction can be applied on $s_1 s_3 \cdots s_n$ to obtain the desired sequence. Otherwise there are three cases to consider, corresponding to whether s_2 is a clean divide, a refolding or a removal of a trivial node.

The case in which s_2 is a clean divide is considered first. Suppose s_2 cleanly divides u into u_1 and u_2 such that u_2 contains all the new pins created by the unclean divide. Figure 2.20 illustrates this case. s_1 is the unclean divide $U(z)$ and

s_2 is the clean divide, $C(u)$ of u which results in u_1 and u_2. The sequence $s_1 s_2$ can be replaced by a clean divide of z into u_1 and z' followed by an unclean divide of z' into u_2 and v. This is possible since u_1 cannot be surrounded by wires created by the unclean divide s_1. Applying induction to the subsequence starting with the unclean divide gives the desired sequence.

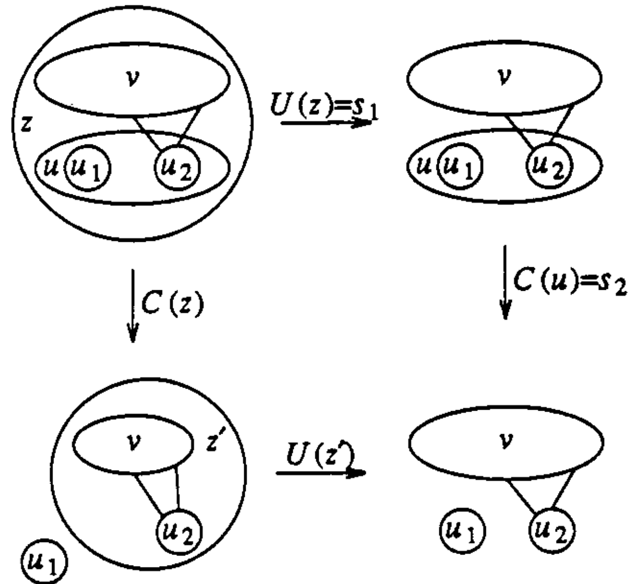


Figure 2.20

Suppose now that the clean divide, s_2, divides u so that both u_1 and u_2 have some of the pins which were created by the unclean divide s_1 as in Figure 2.21. s_1 is the unclean divide $U(z)$ and s_2 is the clean divide, $C(u)$ of u which results in u_1 and u_2. In this case we can replace $s_1 s_2$ by two unclean divides, of z into u_1 and z' and then of z' into u_2 and v. The order of the unclean divide must be determined if one of the nodes is surrounded by the other two as is the case in Figure 2.21 if the dashed line exists. We have $s'_1 s'_2 s_3 \cdots s_n$ taking A to (A,S) in $U^2(C+Re+R)^*$. Applying induction on $s'_2 s_3 \cdots s_n$ gives S' of length less than $n-1$ taking (A,s'_1) to (A,S) in $(C+Re+R)^*$. Then applying induction to $s'_1 S'$ gives $S'' \in (C+Re+R)^*$ of length less

than $n-1$ taking A to (A,S) which satisfies the lemma.

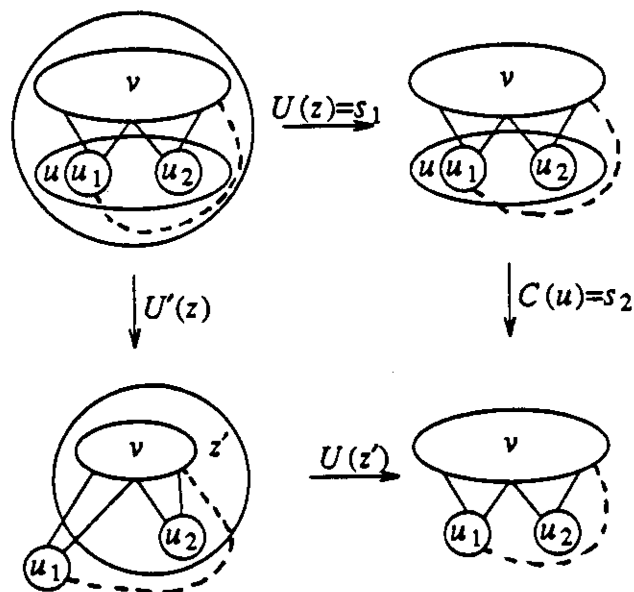


Figure 2.21

Suppose now that s_2 is a refolding of u which divides it into u_1 and u_2 and then merges u_1 and u_2 to obtain u'. This is the sequence of operations corresponding to the topmost and rightmost operations in Figure 2.22. Ignoring the merge for the moment, the two divides in this path, divide z into u_1, u_2 and v. The same result can be achieved by first dividing z into u_1 and z' and then uncleanly dividing z' into u_2 and v. Care must be taken to select u_1 so that it is not surrounded by the other two nodes. This sequence corresponds to the operations performed by moving downward, then right and then downward again in Figure 2.22. Now consider the merge of u_1 and u_2. As argued previously, we can exchange the unclean divide of z' and this merge with a merge of z' and u_1 to form z'' and then an unclean divide of z' into u' and v. This corresponds to the leftmost and bottommost sequence of operations in Figure 2.22. Note that the two leftmost operations form a refolding of z into z''. We thus obtain a refolding followed by an unclean divide which can replace

$s_1 s_2$. We apply induction on the sequence starting with the unclean divide to get the sequence required by the lemma.

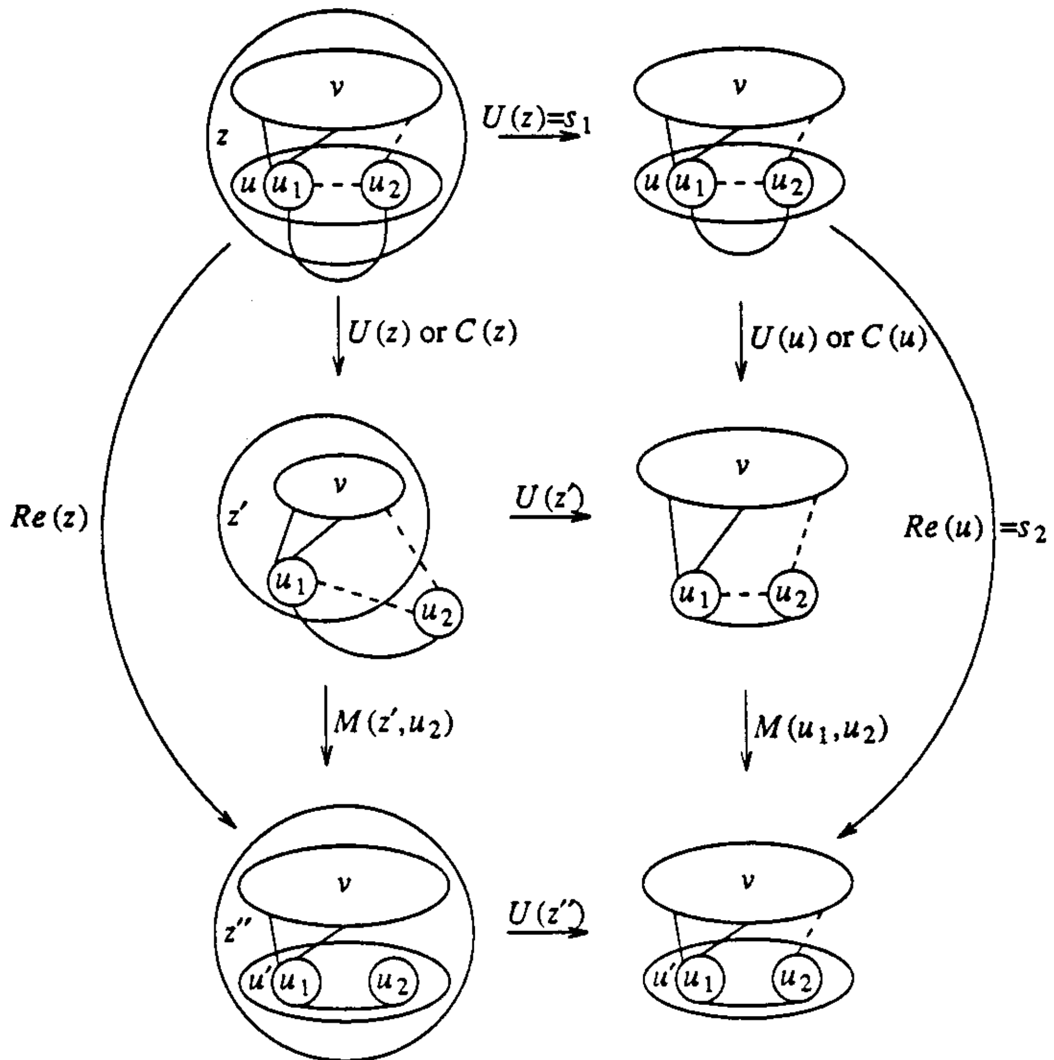


Figure 2.22

The remaining case to consider is when s_2 is the removal of u. In this case s_1 creates the trivial node u. There are two ways in which an unclean divide can create a trivial node, either by creating a trivial node on an existing wire or by generating two new pins and two new wires. In the first case, we can replace the divide by an

insertion and apply Lemma 2.28 to obtain the final sequence. The other case is more complicated since the unclean divide is in fact creating a new trivial self-loop and inserting a node on it. Thus the effect of $s_1 s_2$ is of adding a trivial self-loop to z. Consider this trivial self-loop. It will remain a trivial self-loop as long as there is no clean divide or refolding whose divide cuts between the two pins of the loop. Since the final planar circuit is maximal one of these two events must occur. In the latter case, $s_1 s_2$ can be dropped from the sequence and the missing loop can be created with the divide of the refolding to obtain a sequence satisfying the lemma. In the former case, we can also drop $s_1 s_2$; but in this case to create the loop we must replace the clean divide by an unclean divide. However we can then apply induction to the sequence starting with the unclean divide to get a sequence satisfying the lemma as well.

□

By combining Lemma 2.31 and Corollaries 2.27 and 2.30, the last unclean divide of a sequence leading to a maximal-indivisible planar circuit can be removed.

Corollary 2.32 Given (A,S) which is a maximal-indivisible planar circuit with $S \in U(C+I+Re+R+M)^*$ then there exists $S' \in (I+R+M)^*(R+C+Re)^*$ such that $(A,S) \equiv (A,S')$.

Proof: Using Corollary 2.27, we first transform S into $S' \in U(I+R+M)^*(Re+I+R+C)^*$ such that $(A,S) \equiv (A,S')$. Then apply Corollary 2.30 to obtain $S'' \in (I+R+M)^*(Re+U+\lambda)(Re+I+R+C)^*$ such that $(A,S) \equiv (A,S'')$. If $S'' \in (I+R+M)^*(R+C+Re)^*$ then S'' satisfies the corollary. Otherwise by applying Lemma 2.31 to the subsequence starting with the first unclean divide or insertion after the merges, gives $S''' \in (I+R+M)^*(R+C+Re)^*$ such that $(A,S) \equiv (A,S')$.

□

Repeated application of this corollary gives the desired result of this section, allowing us to remove the last unclean divide from a sequence leading to a maximal-indivisible planar circuit.

Corollary 2.33 Given (A,S) which is a maximal-indivisible planar circuit with $S \in (U+C+I+Re+R+M)^*$ then there exists $S' \in (I+R+M)^*(R+C+Re)^*$ such that $(A,S) \equiv (A,S')$.

2.7 Removing Glues

Our final task is to get rid of the glues. We will first show that a glue can be moved past merges and then clean divides. We then face the difficult task of moving it past refoldings. To accomplish this we need to separate refoldings into differents types of refoldings. Intuitively, the result of a glue is a node which is divisible and should cancel with a clean divide. However, refoldings of the glued node may tangle up the self-loops of the node so badly that it is no longer possible to divide the node with just one clean divide. We will introduce a special kind of refolding, a *tangling* whose only effect is to permute the order of a set of adjacent self-loops of a node. We will then be able to obtain a sequence in which only tangles, clean divides and removals follow the glues. We can then show that the effects of the tangles cancel, permitting the glued node to be cleanly divided.

To move a glue past a sequence of merges, clean divides, insertions and removals, we first rearrange the sequence so that all merges occur before any clean divides and refoldings. This is necessary since moving a glue past a clean divide may result in two glues (of smaller size) but moving a glue past a merge may

increase its size. We first move a glue past a merge.

Lemma 2.34: Given (A,S) where $S \in G(I+R+M)^*$, there exists $S' = s'_1 \cdots s'_m$, such that $(A,S) \equiv (A,S')$ and $S' \in (I+R+M)^*(G+\lambda)$.

Proof: The proof proceeds by induction on n, the length of the sequence. If $(n=1)$, then $S \in G$ and $S = S'$ satisfies the lemma so assume that the lemma is true for all sequences of length less than $n>1$. Suppose s_1 is $G(u,v,i,j)$ which creates the node z. Consider s_2. If it is the removal of a trivial node, then it can be exchanged with s_1 since z is not trivial. Applying the induction hypothesis to $s_1 s_3 \cdots s_m$ gives a sequence $s'_1 \cdots s'_m$ such that $s_2 s'_1 \cdots s'_m$ satisfies the lemma. If s_2 is the insertion of a trivial node, then the same is true since a glue does not remove any wires. The last case to consider is when s_2 is a merge. If this merge does not involve z, then again we can exchange s_1 and s_2, and apply induction to obtain a sequence satisfying the lemma. So assume s_2 is the merge, $M(z,p,i,j,h)$. All of z's pins were originally u's and v's, $z = u_1, \ldots, u_r, v_1, \ldots, v_t$. Suppose first that the wires which are subsumed by the merge, include only pins which were originally part of u. This case is depicted in Figure 2.23 where s_1 is the glue of u and v, $G(u,v)$ producing z, and s_2 is the merge of z with p, $M(p,z)$. In this case we can replace $s_1 s_2$ by the merge $M(u,p,a,j,h)$ which results in node z' and a glue $G(z',v,b,j)$. Applying induction to $G(z',v,b,j)s_3 \cdots s_n$ gives the sequence $s'_1 \cdots s'_m$ and then $M(u,p,a,j,h)s'_1 \cdots s'_m$ satisfies the lemma.

The same argument can be made if $z_i, \ldots, z_{i+h}$ includes only pins which were originally v's. If the pins involved in the merge were originally both from u and v as in Figure 2.24, then we can replace $s_1 s_2$ by two merges as follows. We first merge p with which ever of the nodes, u or v does not have pin z_i and then we merge

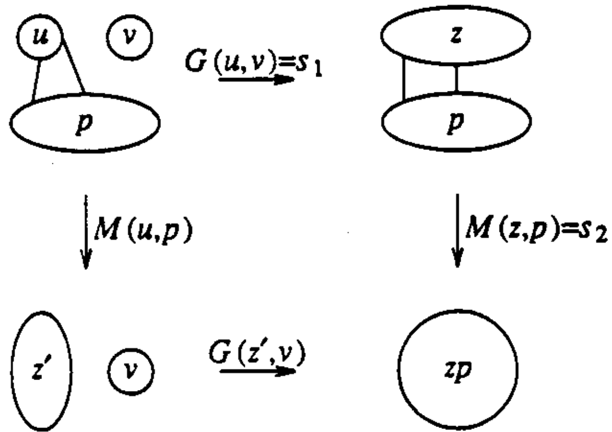


Figure 2.23

the result with the other. In this way we can handle the case in which one of the two nodes is surrounded by p and the other as is the case for v in Figure 2.24. We obtain a sequence without a glue which satisfies the lemma.

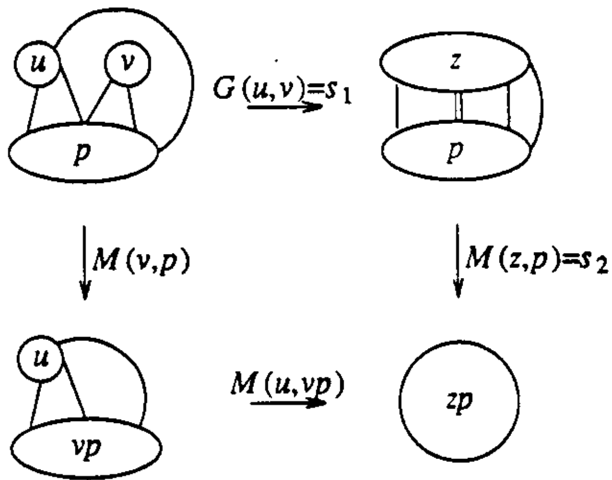


Figure 2.24

□

Corollary 2.35 Given (A,S) with $S=s_1 \cdots s_n \in G(I+R)^*$, then $(A,s_2 \cdots s_n s_1) \equiv (A,S)$.

Proof: This is observed by noting that in the proof of Lemma 2.34, s_1 was merely

swapped with removals or insertions of trivial nodes to get $s'_1 \cdots s'_m$.

□

To get a glue past a clean divide we will need to perform induction on the size of the glue as well as the length of the sequence, since moving a glue past a clean divide may require replacing the glue by two glues. We define the size of a glue as, $|G(u,v,i,j)| = m + n$ where m and n are the number of pins in u and v respectively.

Lemma 2.36 Given (A,S) with $S \in G(I+R+C)^*$, there exists a sequence of operations, $S' \in (I+R+C)^* G^*$, such that $(A,S') \equiv (A,S)$.

Proof: This lemma is proved by induction on the size of the glue. The minimal size of a glue is four since the minimal size of an R-node is two. Assume $s_1 = G(u,v,i,j)$ which creates the node z.

Basis: Suppose $|s_1| = 4$. The argument in this case proceeds by induction on n, the length of the sequence. If $n=1$ then we are done since s_1 itself satisfies the lemma. Assume we have shown this case for sequences of length less than $n>1$. If s_2 is the removal or insertion of a trivial node, then we can swap it with s_1 by the last corollary and apply induction. Suppose s_2 is a clean divide. If s_2 does not divide z, then again we can swap it and apply induction, so assume s_2 divides z. Since $|s_1| = 4$, u and v are both trivial. Since s_2 is a clean divide, it divides z along its partitions and since the partitions of z correspond to u and v, s_2 divides z back into u and v. Thus $(A, s_3 \cdots s_n) \equiv (A,S)$ and $s_3 \cdots s_n$ satisfies the lemma.

Induction: Assume we have shown the lemma to be true when the size of the glue is less than L and that $|s_1| = L > 4$. Again we proceed by induction on n the length of the sequence. The lemma is trivially satisfied for $n=1$ so assume it is true for sequences of length less than $n>1$. If s_2 is the insertion or removal of the trivial

node we can apply induction on n to get the desired sequence after swapping s_1 and s_2. Similarly if s_2 is the clean divide of any other R-node than z. So suppose s_2 divides z into p and q. If p and q are u and v then clearly we can drop $s_1 s_2$ and $s_3 \cdots s_n$ satisfies the lemma, so assume not. Since z is obtained from a glue of u and v, and z can be cleanly divided into p and q, its sequence of pins consists of u's and v's

$$z = u_1, \ldots, u_{m_u}, v_1, \ldots, v_{m_v}$$

and also consists of the pins which will become p's and q's,

$$z = p_1, \ldots, p_{m_p}, q_1, \ldots, q_{m_q},$$

with $m_u + m_v = m_p + m_q$. Suppose that v properly contains all the pins in q. This also means that p properly contains all the pins in u. This case is illustrated in Figure 2.25. We we can replace $s_1 s_2$ by $s'_1 s'_2$ where s'_1 is a clean divide of v into q and p', the intersection of v and p, and s'_2 is the glue of u and p' resulting in p. $|s'_2| < L$ so we can apply induction to $s'_2 s_3 \cdots s_n$ and to obtain $S' \in (I+R+C)^* G^*$ and then $s'_1 S'$ satisfies the lemma.

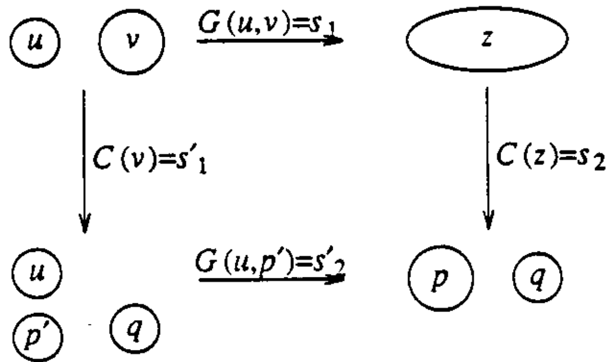


Figure 2.25

The remaining case occurs when none of the nodes are contained in any other; that

is, z is partitioned into four non-empty sets of pins by pairwise intersecting u and v with p and q as in Figure 2.26. In this case, we replace $s_1 s_2$ by first cleanly dividing u and v into their p and q components and then gluing these components together. We have $(A,S) \Longrightarrow (A,S')$ where $S' = s'_1 s'_2 s'_3 s'_4 s_3 \cdots s_n$, s'_1, s'_2 are clean divides and s'_3, s'_4 are glues.

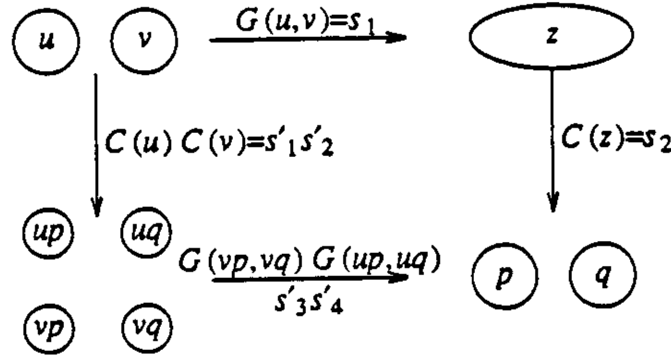


Figure 2.26

Since $|s'_3|+|s'_4| = |s_1| = L$, we can apply the induction hypothesis to $((A,s'_1 s'_2 s'_3), s'_4 s_3 \cdots s_n)$ to get $S'' \in (I+R+C)^*(G)^*$ where s''_c is the first glue (if there is no glue let c be the length of $S'' + 1$). Then apply the induction hypothesis to $((A,s'_1 s'_2), s'_3 s''_1 \cdots s''_{c-1})$ to get $S''' \in (I+R+C)^*(G)^*$. The sequence $s'_1 s'_2 S''' s''_c \cdots s''_{m''}$ satisfies the lemma.

□

In order to move the glues past refoldings we need to examine the refoldings more carefully. A refolding is *complete* if its merge is complete. The symbol Re_+ will be used to represent the complete refoldings.

Lemma 2.37 Given (A,S) which is maximal-indivisible with $S \in (R+C+Re)^*$, there exists $S' \in (R+C+Re_+)^*$ such that $(A,S) \equiv (A,S')$ with the length of S' not greater than the length of S.

Proof: The proof proceeds by induction on the length of S, n. If $n=1$ then clearly $S \in (R+C+Re_+)^*$ since an incomplete merge would leave a non-maximal node: a node with trivial self-loops. So assume that the lemma holds for all sequences of length less than $n>1$. We can apply induction to the sequence starting with s_2 to obtain $S' \in (R+C+Re_+)^*$ such that $(A,S) \equiv (A,s_1S')$ and the length of S' is at most $n-1$. If $s_1 \in (R+C+Re_+)^*$ then s_1S' satisfies the lemma, so suppose $s_1 \in Re - Re_+$. s_1's merge is an incomplete merge which results in a node z with one or more trivial self-loops. Consider the sequence s_1S'. If the operations in S' do not involve z, then we can place s_1 at the end of the sequence and this leads to a contradiction. z is not trivial so s'_1 can not be its removal. We shall show how we can alter s_1S' so that it is not longer and the first operation is a complete refolding.

Consider what happens to the trivial self-loops created by s_1. They must disappear before the end of the sequence. They are clearly not affected by a removal of a trivial node, a clean divide which does not cut across them or a refolding whose divide does not cut across them. Hence there must be a refolding or clean divide in S' which cuts across these self-loops.

Consider the operation which cuts across the innermost self-loop. If it is a refolding, we replace this refolding's division so that it creates these loops and modify s_1 so that it subsumes these loops in its merge. Note that the existence or absence of these loops does not affect the other operations in the sequence, other than possibly affecting the completeness of some refoldings. If this is the case we apply the induction hypothesis to restore the completeness of these refoldings without lengthening the sequence. If the operation is a clean divide then we can also drop the loop by replacing the clean divide by an unclean one which generates the

self-loops. Note that this unclean divide does not create a trivial node. In this case we apply Lemma 2.31 to the subsequence beginning with the unclean divide, to get shorter sequence to which we can apply induction. By repeating this process we can obtain a sequence in which s_1's merge becomes complete.

□

Lemma 2.38 Given (A,S) which is maximal-indivisible with $S \in (Re+C)^* R^*$ there exists $S' \in Re^* C^* R^*$ such that $(A,S) \equiv (A,S')$ and $|S| = |S'|$.

Proof: The proof proceeds by induction on the length of the sequence, n. If $n=1$ then S itself satisfies the requirement. So assume the lemma is true for sequences of length less than $n>1$. By applying the lemma to $s_2 \cdots s_n$ we can assume that $S \in (Re+C)Re^* C^* R^*$. If S is not in $Re^* C^* R^*$ then s_1 must be a clean divide and s_2 a refolding. So assume node z is cleanly divided into u and v by s_1 and then u is refolded by s_2 into u' as in Figure 2.27. The refolding divides u into p and q and then merges them along the solid line. Since z is cleanly divided into u and v, its sequence of pins comprises those of u and v, $z = u_1 ....u_m v_1 \cdots v_k$. Assume without loss of generality that pin u_m will become part of q, thus the pins of u which will become part of p are $u_j \cdots u_h$ for $h<m$. We will replace $s_1 s_2$ as follows. First perform a divide of z into p and $z' = y_1 .. y_l u_{h+1} \cdots u_m v_1 \cdots v_k u_1 \cdots u_{j-1}$ or $z' = y_1 .. y_l u_{h+1} \cdots u_m v_1 \cdots v_k$ if $j=1$ where $y_1 \cdots y_l$ are the new pins created by possible created by the divide of u. Now perform a merge of p and z' along the same pins as the merge of s_2. At this point we have $z'' = u'_1 \cdots u'_g v_1 \cdots v_k$ which can be cleanly divided into u' and v. A sequence of length n in $ReCRe^* C^* R^*$ is obtained. To obtain the required sequence, it suffices to apply induction to the subsequence starting with second operation.

□

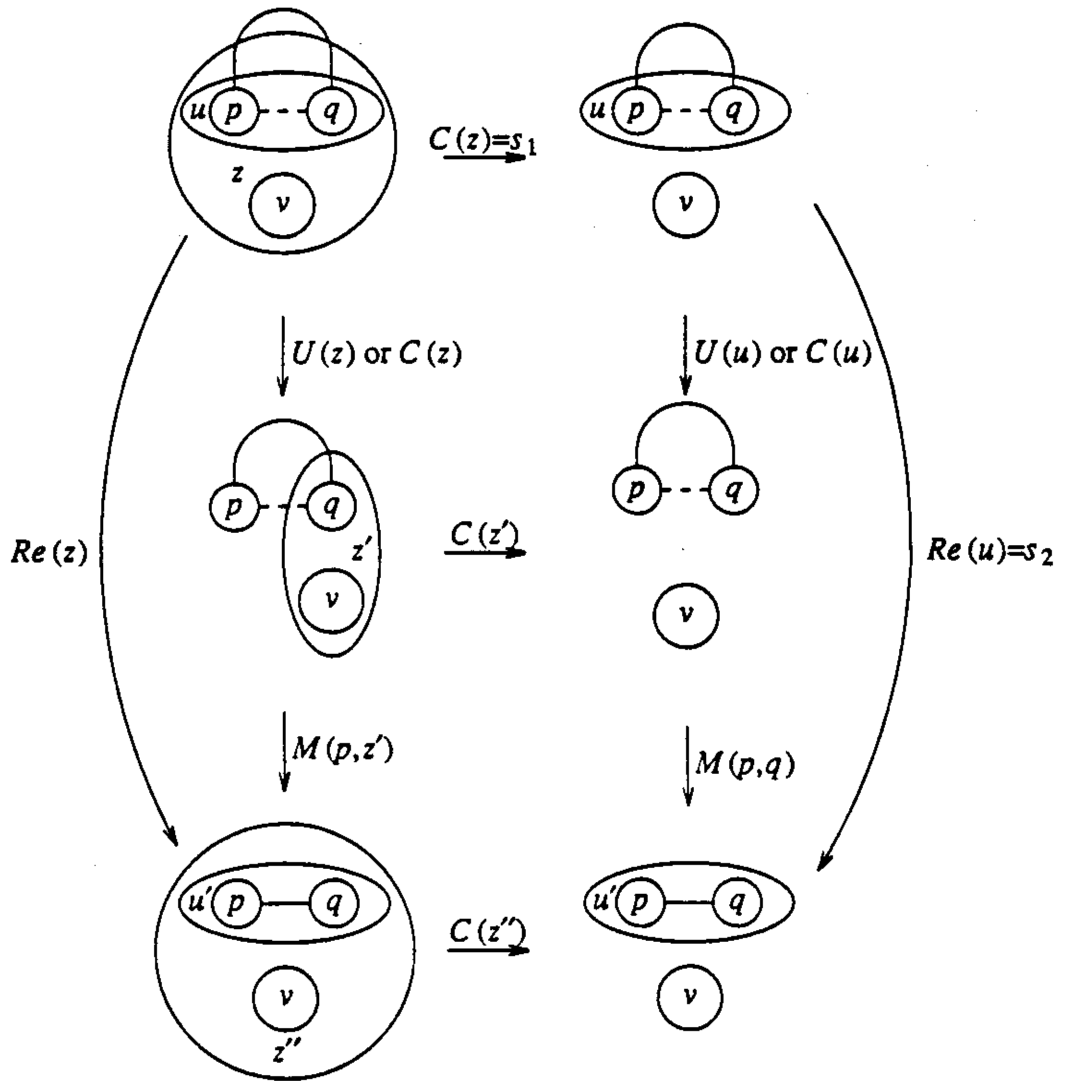


Figure 2.27

These last two lemmas can be combined to organize a sequence of refoldings, clean divides and removals so that all refoldings are complete and occur before any clean divides and removals, when the sequence leads to a maximal-indivisible planar circuit.

Corollary 2.39 Given (A,S) which is maximal-indivisible with $S \in (Re+C+R)^*$,

there

exists $S' \in (M+I)^* Re_+^* C^* R^*$ such that $(A,S) \equiv (A,S')$.

Proof: Consider a removal which occurs in the sequence before a refolding or clean divide. If there are no refoldings whose merge depends on the wire created by the removal, then we can move it to the end of the sequence since it can jump over these refoldings, any clean divide and other removals. So assume this is not the case. Since the removal is needed in order to perform a refolding, the trivial node sits on a self-loop of the node to be refolded. In this case we replace the removal by a merge. By repeating this for each removal, a sequence in $(M+Re+C)^* R^*$ is obtained. The merges can be moved to the front of the sequence using Lemmas 2.23 and 2.24. In either case no removals are introduced so we obtain a sequence in $(I+M)^*(I+Re+C)^* R^*$. Applying Lemma 2.28 to the part of the sequence in $(I+Re+C)^* R^*$ we obtain a subsequence in $(Re+C)^* R^*$ to which Lemma 2.38 can be applied. We now have replaced S by a sequence in $(I+M)^* Re^* C^* R^*$. We need only make the refoldings complete. In the proof of Lemma 2.37, the process of transforming the refoldings into complete refoldings involves transforming either another refolding or a clean divide into an unclean one. In Lemma 2.31, an unclean divide followed by clean divides and removals was replaced by only clean divides and removals. Removals can always moved behind clean divides. Thus the process of transforming the refoldings into complete ones will maintain the refoldings ahead of the clean divides followed by the removals.

□

The last and most difficult step in removing a glue from a sequence leading to a maximal-indivisible planar circuit, is to move it past refoldings. We will show that a

glue can be moved past a set of refoldings, leaving behind refoldings of the original nodes followed by unclean divides, gluings and a special type of refolding which serves to entangle the self-loops. We cannot move the glues past all of the refoldings since the unclean divide of a refolding can be arbitrary, generating and interleaving partitions unnecessarily. However, we can show that these refoldings must cancel each other out and that the gluings and clean divides must also cancel each other out if the final planar circuit is maximal-indivisible. This special type of refolding is called a *tangling*.

Definition 2.40

A *tangling* is a complete refolding whose effect is only to permute the order of a set of adjacent non-trivial self-loops.

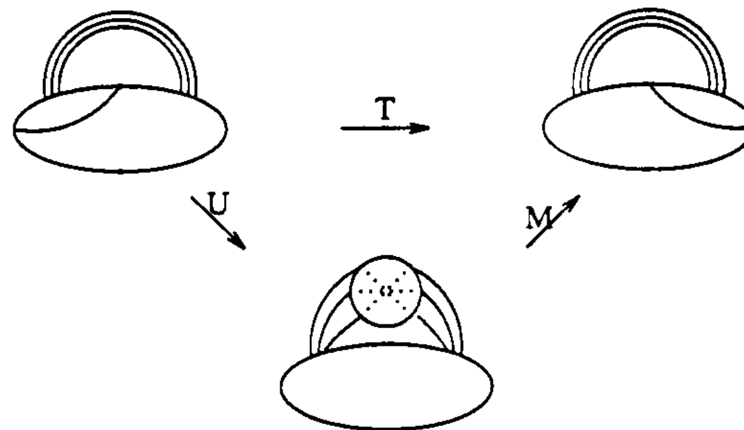


Figure 2.28

Figure 2.28 contains a tangling. Since a tangling can only affect a set of adjacent self-loops of a node; it does not affect any pins which are connected to other nodes and it does not create nor remove partitions. The symbol T will be used to represent tanglings. Before we can attack glues we need to prove a few things about tanglings

and refoldings.

Definition 2.41

A divide of a node u is *orderly*, if any clean divide that was applicable to u can be performed by two clean divides on the two nodes created by the divide.

In an orderly unclean divide, the new pins along the cut do not entangle any partition. That is if the node could be separated cleanly, then the same will be true after the unclean divide. All clean divides are orderly.

Definition 2.42

An *orderly refolding* is a refolding whose divide is orderly.

Re_o will be used to represent orderly refoldings. Re_{o+} represents an orderly complete refolding. We will decompose arbitrary complete refoldings into orderly complete refoldings followed by tanglings. The size of the tangling is the number of self-loops it tangles.

Lemma 2.43 Given (A,S) with $S \in TRe_{o+}^*$, there exists $S' \in Re_{o+}^*T^*$ such that $(A,S)\equiv(A,S')$.

Proof: We will show that we can move a tangle past an orderly complete refolding, obtaining one or two orderly complete refoldings followed by at most two tangles such that the combined sizes of the new tangle(s) is no more than the size of the original tangle. The lemma will then follow by induction on the number refoldings and the size of the tanglings.

If the tangling and refolding are of different nodes we can interchange them; we need only consider the case where the refolding is applied to the node resulting from the tangling. Suppose we have a tangling s_1 of a node u resulting in u' followed by a refolding s_2 of u' resulting in u''. Since a tangling does not introduce new or remove any partitions along the self-loops it is clear that the unclean divide of the refolding can not depend on any partitions added or removed by the tangling. The argument can be divided into four cases according to the location of the tangled self-loops on u' with respect to the divide and merge of the refolding.

Suppose first that the tangled self-loops remain on only one of the nodes of the divide. Then we can perform the tangling on only that node after the divide instead of before. Likewise, we can also perform the tangling after the merge. We can thus perform the same tangling after the refolding. In the second case, the divide cuts across all of the self-loops. If the refolding merges along the loops involved in the tangling, then we can skip the tangling all together since the effect of the tangling is destroyed by the merge of the refolding. If the merge is along some other set of self-loops, the order of the tangled self-loops is irrelevant to the refolding; the tangle can be performed after the refolding. Again in this case the refolding and the tangling can be swapped.

The remaining two cases occur when the divide of the refolding, s_2 cuts across only a portion of the tangled self-loops. In this case, the merge may introduce pins in between the self-loops making it impossible to perform the tangling after the refolding. In this case we first transform the refolding so that it cuts across all of the tangled self-loops. This is achieved as in Figure 2.29, by moving the divide over until it cuts all of the self-loops as indicated by the two dotted lines. This means that

the divide is now certainly unclean and creates wires for these self-loops as shown in Figure 2.29. The difference between these last two cases is in the loops subsumed by the merge of the refolding. The merge of s_2 either merges along the tangled set of loops or along some other set of loops as depicted by the two results of the merge $M(u'_1, u'_2)$ in Figure 2.29.

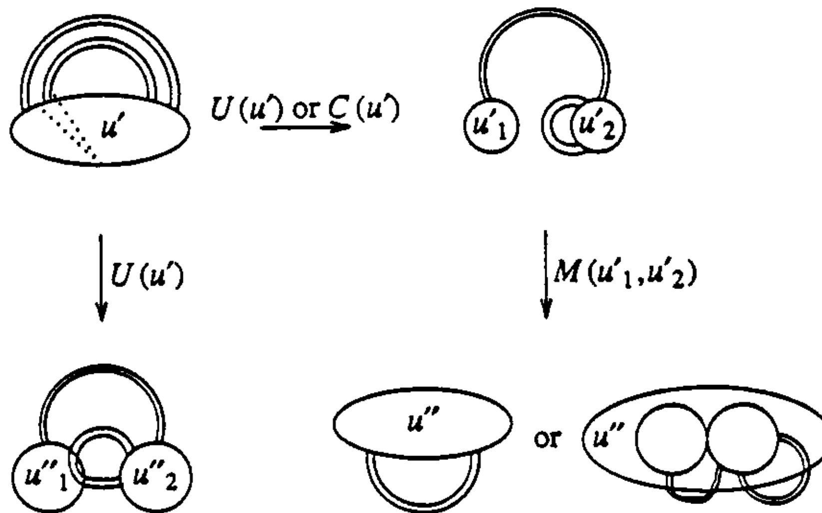


Figure 2.29

Consider first the case in which the merge of s_2 was along the tangled self-loops that it cut. Since the refolding was complete, its merge was along all of the loops cut by the original divide. We simply change the merge so that it now also subsumes the additional loops cut by the new divide. This is depicted by the merge $M(u''_1, u''_2)$ in Figure 2.30 The new wires created by the new divide are left behind to take their place. This new merge along with the new divide form a refolding which is complete and orderly. Now consider the tangling. In this case the tangling only affects the order of the self-loops. The tangling of the self-loops which are subsumed by the merge is unnecessary since they will disappear and there order is irrelevant to the merge. The tangling on them can be omitted, but we must apply it to

the new self-loops which remain after the merge to take their place. Thus we can perform the unclean divide and refolding on u and then apply a tangling on these self-loops. The divide of u can be made orderly since the order chosen for the new wires is unimportant; the tangling will alter this order to put the self-loops in their final required order. The completeness of the merge follows from the completeness of the original merge. We obtain a complete orderly refolding of u followed by a tangling of smaller size. This corresponds to the left arrow and bottom two arrows in Figure 2.30.

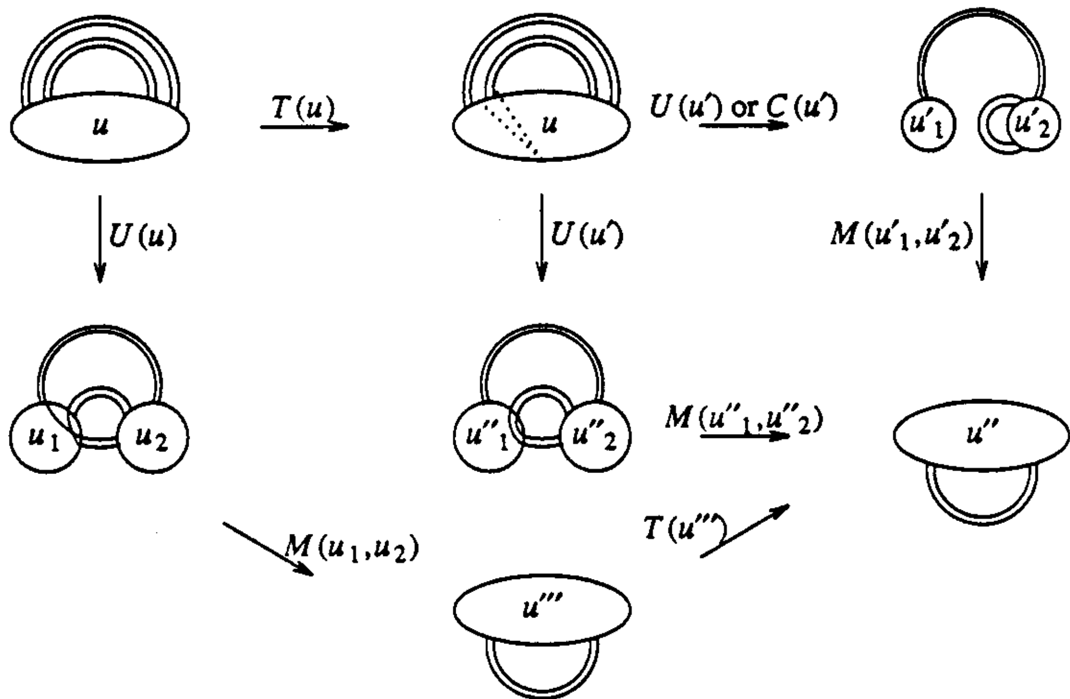


Figure 2.30

The final case occurs when the merge is not along the tangled self-loops. This is more complicated as shown in Figure 2.31. Essentially we perform the same merge on u''_1 and u''_2 as $M(u'_1, u'_2)$. Unfortunately, this does not result in u''. We

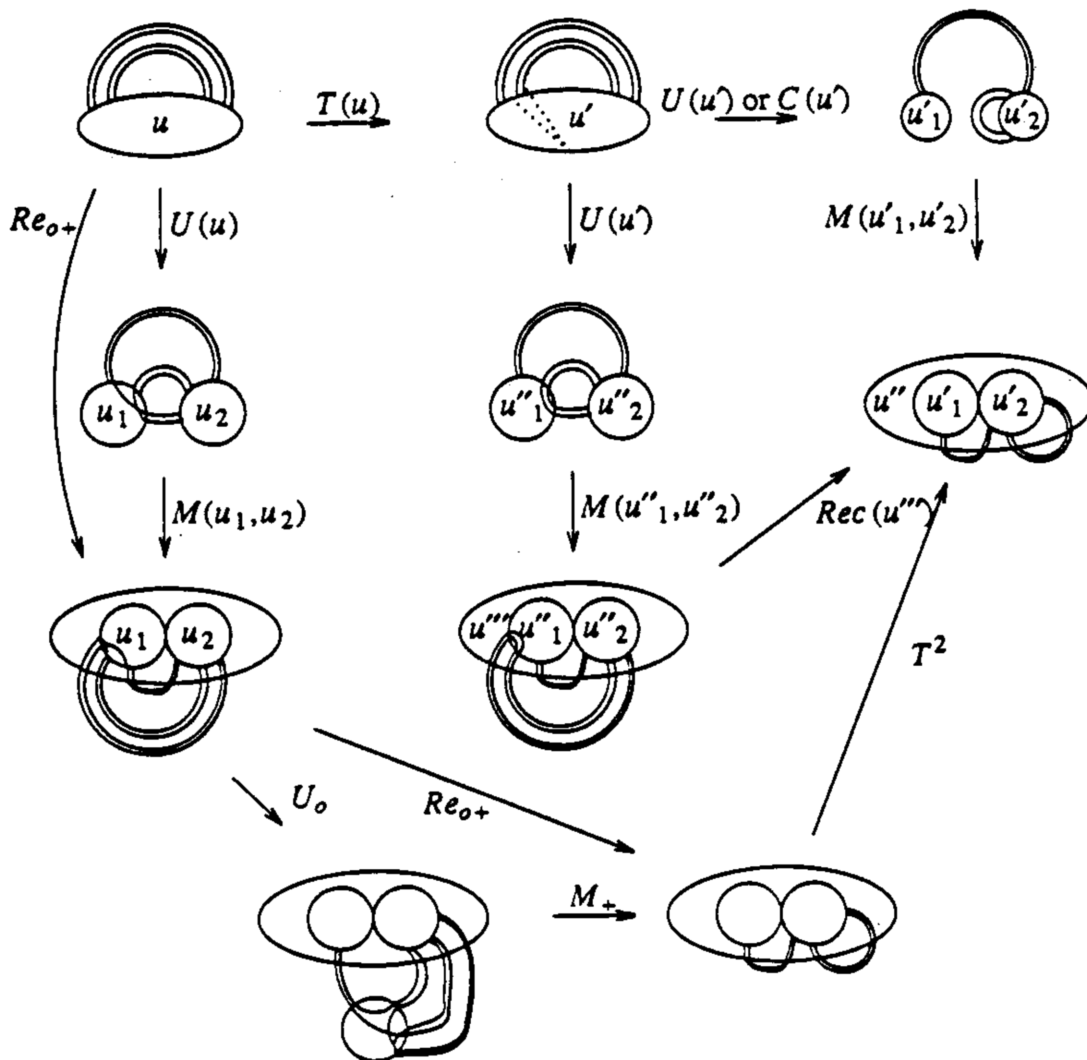


Figure 2.31

need to free the self-loops that have been hooked into u''_1. This is accomplished with a clean divide followed by a complete merge which combine to form a refolding. Now consider the tangling. We can perform the same unclean divide and merge on u as on u'. The only difference is in the order of the self-loops. We then perform a refolding similar to the one performed on u''' to get u'' with the exception that the divide might not be clean since some of the wires which are not hooked may be

entangled with those that are. We thus perform an orderly unclean divide followed by a complete merge to unhook and separate these wires. Finally we may need to perform at most two tangles on these two sets of self-loops in order to arrange them in their final order in u''. Note that this is only necessary if the original refolding was of size greater than two; the sum of the sizes of these two tanglings is the size of the original. Figure 2.31 depicts the case in which the merge of s_2 was along a set of self-loops above u'_1 and u'_2. The other case is symmetric and can be dealt with in the same manner.

In each of the four cases, a tangling followed by a complete orderly refolding has been replaced by a sequence of at most two orderly complete refoldings followed by at most two tanglings such that the sum of the sizes of these tanglings is no greater than the size of the original tangling. Induction on the size of the tangling is used to finish the proof. If the size is two, then the case in which two tanglings are generated can not occur and we can move the tangling past the refoldings leaving behind complete orderly divides until the tangling either disappears or reaches the end of the sequence. Now assume the lemma is true for all tangles of size less than $n>2$. In this case we can move the tangle over the refoldings. Either it will disappear, split into two smaller tangles or end up at the end of the sequence. If it splits into two tangles then the induction hypothesis can be used since the two new tangles will be of smaller size.

□

A sequence of complete refoldings can now be divided into a sequence of complete orderly refoldings followed by tanglings.

Corollary 2.44 Given (A,S) with $S \in Re_+^*$, there exists $S' \in Re_{o+}^* T^*$ such that

$(A,S)\equiv(A,S')$.

Proof: The proof is by induction on the number of unorderly refoldings. If there are none the corollary is satisfied by S. Otherwise take the last one. By reordering the pins created by the divide of the refolding, we can make this divide orderly. Since the refolding is complete, after the merge the new wires created by the divide will form a set of adjacent non-trivial self-loops. We can then apply a tangling to these to put them in the same order as the original refolding had them. Thus we can replace this refolding by a complete orderly refolding followed by a tangling. The previous lemma is applied to move the tangling to the end of the sequence. This procedure can be repeated until a sequence with only orderly complete refoldings followed by tanglings is obtained.

□

The result of a glue is a divisible node, and hence must be cleanly divided in a sequence leading to a maximal-indivisible planar circuit. It would seem that a glue should cancel with a clean divide in such a sequence. However, this can be complicated by intervening tanglings.

Lemma 2.45 Given (A,S) which is maximal-indivisible with $S \in G^*T^*C^*R^*$, there exists $S' \in (I+M+Re+C+R)^*$ such that $(A,S)\equiv(A,S')$.

Proof: A tangling can move past a glue or another tangling which is not applied to the node it produces, hence by regrouping the operations in the sequence, it is sufficient to consider the case in which the glues construct one node, to which all the tanglings are all applied. We will assume that the original nodes can not be cleanly divided since otherwise we could simply add clean divides and glues to the front of the sequence and apply the lemma to the subsequence beginning with the glues,

restoring the clean divides to the result. We can assume that tanglings operate on a maximal set of adjacent self-loops and combine those that operate on the same set into one tangling which permutes the initial order of the set of self-loops into its final order after all of the tanglings. Thus we will assume that the tangles operate on disjoint sets of wires, that is, there is at most one tangling for each set of self-loops.

The proof proceeds by induction on the number of tanglings, so suppose $S \in G^*T^*C^*R^*$ has n tanglings. If there are no tanglings (n=0) then by applying Lemma 2.36 we obtain a sequence in $(I+R+C)^*G^*$. Since each node in (A,S) is indivisible there can be no glues and since each node is maximal each insertion must cancel out some removal. Thus we obtain a sequence in $(C+R)^*$ which satisfies the lemma.

So we assume that a sequence in $(I+M+Re+C+R)^*$ can be obtained for sequences with less than $n>0$ tanglings. The glues all form one node; they can be rearranged so that the nodes are glued together in any order. Since the tanglings are all disjoint, they also can be performed in any order.

Consider a tangling. If the loops it tangles belong to only one of the nodes before the gluings then we could perform the tangling on this node before any of the glues. We can then apply induction on the remaining sequence since it has one fewer tangling. Remember that a tangling is also a refolding so the resulting sequence obtained by tagging the tangle back on the front satisfies the lemma.

Now suppose that the loops involved in the tangling are wires that connect two of the nodes to be glued. In this case we can also simulate the tangling by an unclean divide and a merge before any of the gluings. By applying the induction

hypothesis to the remaining sequence we obtain a sequence in $UM(I+M+Re+C+R)^*$. Applying Corollary 2.32 gives the required sequence.

The next case to consider is when a tangle applies to a set of self-loops which belong to one node on one side but to more than one node on the other end of the loops. Figure 2.32 illustrates how this can happen. In this case we rearrange the glues so that the nodes which are at the other end of the loops are glued first and then we replace the tangling by an unclean divide and a merge which achieves the tangling before the rest of the glues. This can also be achieved by merges along the self-loops and then an unclean divide as in the operations along the left and bottom arrows in Figure 2.32. A sequence in $M^*UG^*T^*C^*R^*$ with one less tangling is obtained. Applying induction and then removing the unclean divide gives the required sequence.

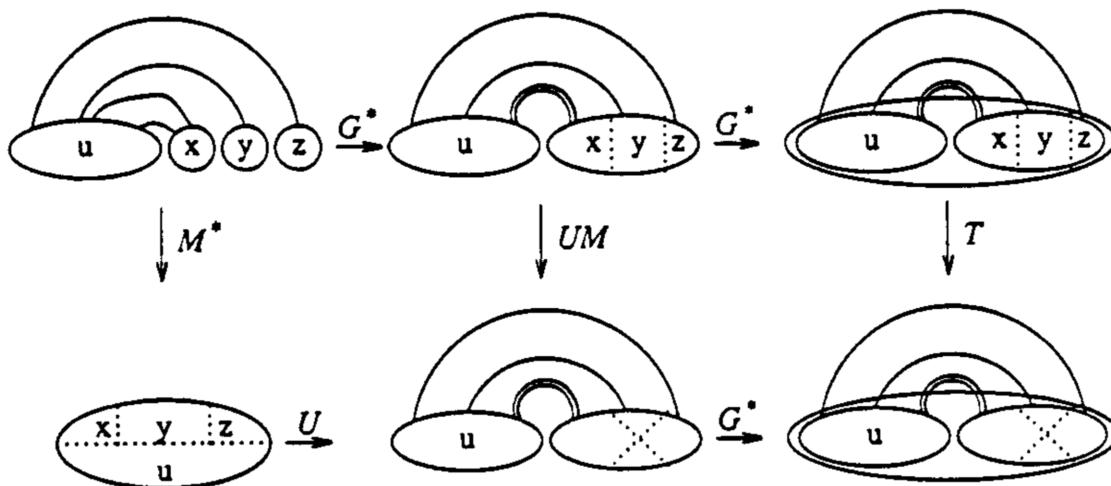


Figure 2.32

The last case to consider is complicated. We will show that it cannot occur. If we have any tangles of the previous types, we already know that we can remove them from the sequence obtaining a new sequence in

$(U+M+I+Re+R+C)^* G^* T^* C^* R^*$ with less tangles. Only the case in which none of the tangles fall in the previous categories needs to be considered. We decompose this remaining type of tangle into an *orthogonal tangle* and the previous kinds of tangles. An *orthogonal tangle* is one which applies to a set of self-loops belonging to nodes $u_1 \cdots u_k$ on one side and nodes $v_1 \cdots v_k$ on the other end such that all loops are between u_i and v_{k-i+1}. In addition, the tangling preserves the order of the self-loops belonging to the same nodes, and we allow $u_k = v_1$.

A tangling can be decomposed into an orthogonal tangle and the previous kinds of tangles. The latter can be performed by operations in $(U+M+I+R+Re+C)^*$ before any of the gluings, so we need only consider the case in which all tangles are orthogonal. We still have at most one of these tanglings applied to each set of self-loops. We will show that orthogonal tangles followed by clean divides and removals cannot lead to a maximal-indivisible planar circuit. This is intuitive, since orthogonal tangles are intertwining the components of divisible nodes.

Consider all of the pins belonging to loops which are tangled and which change position with respect to some other loop as a result of the tangle. Some loops even though they are part of the tangle do not change position with respect to the other loops. Since the nets are acyclic, there must be a node which has a pin connected internally to one of these loops, which is itself not part of such a loop. Suppose u_i is this node and suppose we apply the tangling of u_i first. Thus u_i is involved in the tangle with nodes, $u_1 \cdots u_k$. It is tangled up with one of its neighbors say u_{i+1} and can no longer be cleanly separated from u_{i+1} after the tangle. No other tangle applied can undo the tangling since u_i has a pin which is not involved in any tangle and the only way to untangle u_i and u_{i+1} would be to apply a

tangle to the same wires again. This tangle is depicted in Figure 2.33. The dotted lines indicate the separation between the glued nodes and the dashed lines are the internal connections. The node u_i is in between the leftmost pair of dotted lines. The self-loop of u_i is internally connected to another pin which shares a wire with a pin of another node. The wires connecting the other pins may or may not be self-loops.

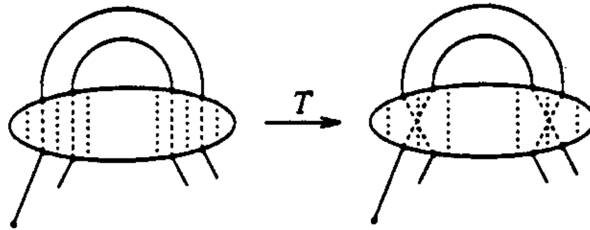


Figure 2.33

After the tangle, u_i and u_{i+1} are part of a non-trivial node u, which cannot be cleanly divided but is divisible, since we could divide it back after untangling the self-loops. Consider what happens to these tangled self-loops at the end of the sequence. These tangled wires of u_i and u_{i+1} cannot be self-loops at the end of the sequence, since a tangling is a refolding and we could cleanly divide the node into u_i and another node after untangling the loops. This is not possible if all nodes of the final planar circuit are indivisible. The wires cannot end up connecting two R-nodes since then this would result in a non-maximal R-node. Hence the only possibility is that the wires end up connected to a B, or IO-node. We will show that this cannot be the case; each wire remains connected, possibly through some other R-nodes, to two R-nodes which cannot become trivial.

Consider the glued nodes at the other end of these tangled wires, they correspond to v_{k-i+1} and v_{k-i}. The first tangle also tangled them. Thus we obtain two non-trivial divisible nodes which cannot be cleanly divided which are connected

by a pair of wires after the first tangle. If there was only one tangle or no other tangle applied to these nodes, we would be done. Thus there must be some tangling applied to $v_1 \cdots v_k$ which untangles these, however it tangles $z_1 \cdots z_k$. Thus u_i and u_{i+1} are still connected to a non-trivial node, z, through v. Repeating the argument for z gives the same result: either no more tangles are applied, or u_i and u_{i+1} become connected to another non-trivial node, possibly through many other nodes. Hence after all the tangles, we will still have $u = u_1 \cdots u_k$, non-trivial, divisible, and connected to another non-trivial divisible node which cannot be cleanly divided. Since this would imply the existence of a non-maximal node, we cannot have any orthogonal tanglings.

□

It remains only to move glues past complete orderly refoldings in order to remove them altogether from a sequence leading to a maximal-indivisible planar circuit.

Lemma 2.46 Given (A,S) with $S \in G^* Re_{o+}{}^*$, there exists $S' \in (U+M+Re_{o+})^* G^*$ such that $(A,S) \equiv (A,S')$.

Proof: This is by induction on the number of refoldings. Obviously if there are none then the original sequence satisfies the lemma. So assume the lemma is true for all sequences with less than $n>0$ refoldings. Consider the first refolding and the composite node resulting from the glues, z. If the refolding is limited to one node then clearly it can be performed before any of the glues. Consider the divide of the refolding. It can be simulated with many divides of the nodes since it is an orderly divide. Then we can glue all of the nodes back together. Now consider the merge. By using Lemma 2.34 we can also move the glues past the merges. In either case we obtain a sequence in $S' \in (U+M+Re)^* G^* Re_{o+}{}^*$ with one fewer refolding after the

gluings. Applying induction gives the result.

□

We now can combine these results to remove the last glue in a sequence without unclean divides, which results in a maximal-indivisible planar circuit.

Lemma 2.47 Given (A,S) which is maximal-indivisible with $S \in G(I+R+M+Re+C)^*$ there exists $S' \in (I+R+M+Re+C)^*$ such that $(A,S) \equiv (A,S')$.

Proof: First rearrange the sequence so that it is in the form $G(I+R+M)^*(I+R+Re+C)^*$ by applying Corollary 2.25 to the subsequence beginning just after the glue. By applying Lemma 2.34 to this sequence we can obtain a sequence in $(I+R+M)^*(G+\lambda)(I+R+Re+C)^*$. If there is no glue we have the required sequence. Otherwise apply the Lemma 2.28 to remove the insertions; the sequence is now in $(I+R+M)^*G(R+Re+C)^*$. By Corollary 2.39 we can further rearrange the subsequence after the glue, so that the sequence is in $(I+R+M)^*G(M+I)^*Re_+^*C^*R^*$. Reapplying Lemma 2.34 gives $(I+R+M)^*(G+\lambda)Re_+^*C^*R^*$. Again if there is no glue we are done otherwise apply Corollary 2.44 to separate the refoldings into orderly complete refoldings followed by tanglings. We now have a sequence in $(I+R+M)^*GRe_{o+}{}^*T^*C^*R^*$. Applying Lemma 2.46 gives a sequence in $(I+R+M)^*(U+M+Re)^*G^*T^*C^*R^*$. And applying Lemma 2.45 gives a sequence in $(U+I+M+Re+C+R)^*$. The remaining problem is to remove the unclean divides. This is accomplished by Corollary 2.33 which gives a sequence in $(I+R+M)^*(R+C+Re)^*$.

□

Combining the results of this section and Section 2.6 gives us the desired format for

sequences resulting in maximal-indivisible planar circuits.

Theorem 2.48 Given (A,S) which is maximal-indivisible there exists $S' \in (I+R+M)^* Re_+^* C^* R^*$ such that $(A,S) \equiv (A,S')$.

Proof: First transform S so that it has no glues or unclean divides. This is accomplished by looking at the last operation which is either a glue or unclean divide. If it is a glue, apply the Corollary 2.47 to remove it. If it is an unclean divide, then Corollary 2.33 can be applied to remove it. So we have a sequence in $(I+R+M+C+Re)^*$. Corollary 2.25 allows us to transform it into a sequence in $(I+R+M)^*(I+R+Re+C)^*$ and from there Lemma 2.28 can be applied to get it into $(I+R+M)^*(R+Re+C)^*$. Corollary 2.39 applied to the second part gives the required sequence in $(I+R+M)^* Re_+^* C^* R^*$.

□

Using the preceding theorem, we can show that homeomorphic maximal-indivisible planar circuits can be transformed into one another by sequences consisting only of refoldings.

Theorem 2.49 If A and A' are maximal-indivisible homeomorphic planar circuits, there exists $S \in Re_+^*$ such that $(A,S) \equiv A'$.

Proof: Since A and A' are homeomorphic, there exists a sequence of operations, S' such that $(A,S') \equiv A'$. Applying the preceding theorem, we can transform S' so that it is in $(I+R+M)^* Re_+^* C^* R^*$. Now consider the first merge in the sequence. A is maximal so none of its R-nodes share wires with other R-nodes. A merge in S' must be with one or between two trivial nodes created by a previous insertion S'. Since A is maximal, none of its nodes have trivial self-loops; any merge involving a trivial

node is a merge along a single wire. Clearly we can drop the merge and the insertions from S'. So assume there are no merges in S' and consider a removal of a trivial node before the refoldings. None of the nodes of A are trivial so it must cancel out some insertion of a trivial node. By dropping these insertions and removals we transform S' so that it is in $I^* Re_+^* C^* R^*$. A trivial node cannot be refolded by a complete refolding to obtain anything but a trivial node. Such a refolding can be dropped from the sequence since it produces an isomorphic planar circuit. Since A' has no trivial nodes any insertions must then cancel out with removals and so these can be dropped from the sequence as well. S' is now in $Re_+^* C^* R^*$. There cannot be a clean divide in S' since this would imply that some node of A was not indivisible. If there is a removal then some node of A was not maximal. Hence we must obtain $S' \in Re_+^*$.

□

In the next section we show how to transform a planar circuit into a homeomorphic planar circuit and discuss the optimality of a maximal-indivisible planar circuit.

2.8 Best Planar Topology

In the previous sections, homeomorphic maximal-indivisible planar circuits were shown to be unique modulo refolding operations. We define this to be the normal form for our planar circuits and discuss how this normal form can be obtained and used; the following questions are addressed:

How can we tell if a planar circuit is maximal-indivisible ?

How do we transform a planar circuit into a maximal-indivisible one ?

Why should a planar circuit be maximal-indivisible ?

To determine whether a planar circuit is maximal-indivisible we must determine if its R-nodes are maximal and indivisible. Determining whether an R-node is maximal consists of determining whether it shares a wire with another R-node, whether it is trivial and whether it has a trivial self-loop. These can all be determined by inspection and in the event that the R-node is not maximal, operation(s) can be applied to the planar circuit either to merge the adjacent R-nodes, to remove trivial R-nodes or to subsume trivial self-loops. Determining whether an R-node is indivisible is more complicated since the definition of divisibility permits the application of refoldings to the R-node before the clean divide is performed. Fortunately, by using results from the previous section we can devise a simpler check for the divisibility of an R-node. The next lemma will show that we need only consider tanglings in determining if an R-node is divisible.

Lemma 2.50 If A is a maximal planar circuit and $S \in Re^*C$ is a sequence which is applicable to A, there exists $S' \in T^*C$ which can be applied to A.

Proof: Suppose we have a planar circuit A and a sequence of operations, $S \in Re^*C$ which can be applied to A. If there are no refoldings, then we are done since S is in T^*C. So assume there is at least one refolding. Let s_i be the first refolding in S whose divide is clean, $s_i = s_{i_1} s_{i_2} \in CM$. Then $S' = s_1 \cdots s_{i-1} s_{i_1}$ is a sequence in Re^*C in which all the refoldings have unclean divides. Hence it suffices to consider only the case in which all refoldings have unclean divides.

Since the inverse of a refolding with an unclean divide is also a refolding, $S^{-1} \in GRe^*$. We can assume that the refoldings are complete since the R-nodes of A are maximal; any trivial loops produced would disappear in a later refolding so we can change the refoldings so that they are complete. By applying Corollary 2.44 we

can divide the refoldings into orderly complete refoldings and tanglings. Lemma 2.46 can then be applied to obtain a sequence $S'' \in (U+M+Re_{o+})GT^*$ such that $(A,SS^{-1}) \equiv (A,SS'') \equiv A$. Note that S''^{-1} is applicable to A and $S''^{-1} \in T^*C(U+M+Re)^*$. The subsequence of S''_{-1} which is in T^*C satisfies the lemma.

□

Thus an R-node is divisible if and only if it can be cleanly divided after applying a sequence in T^*. This means that in determining whether an R-node is divisible, it is sufficient simply to consider any possible permutation of adjacent self-loops.

If an R-node has no trivial self-loops than either it can be cleanly divided or it is indivisible since there is no complete tangling which can be performed other than the identity operation. To check if an R-node, u, is divisible, it is sufficient to check for each pair of adjacent pins u_i, u_{i+1} whether it is possible to cleanly divide the node between these two pins. That is whether there exists two other pins , u_j, u_{j+1} such that no partition of u is represented in both of the subsequences, $u_{i+1}, \ldots, u_j$ and $u_{j+1}, \ldots, u_i$. This can be determined by traversing the sequence of pins starting at u_{i+1} and keeping track of the partitions represented in the subsequence so far. This procedure is then repeated for each pair of pins u_1, u_{i+1} resulting in an algorithm quadratic in the number of pins.

If a node can be cleanly divided such that the cut does not separate adjacent self-loops, then it is clear that the order of the self-loops does not affect this clean divide. Thus we can still use the procedure above for searching for cuts not between pins of adjacent self-loops. In fact, we can extend it to cases where only one side of the cut is between self-loops as follows.

Suppose that we are looking for a cut between u_i and u_{i+1} and we come to pins $u_j, \ldots, u_{j+k}$ which belong to adjacent self-loops. It is sufficient to determine whether there is any order which will complete the partitions represented so far in $u_{i+1}, \ldots, u_{j-1}$. If so we can order the loops so that these occur first in the sequence to obtain a clean divide. The last case to consider is when the cut is between pins of adjacent self-loops on both ends. So suppose we are considering a cut which is between pins $u_i, \ldots, u_{i+m}$ and $u_j, .., u_{j+k}$ and assume first that two sets of pins do not belong to the same self-loops. A cut whose ends fall in these two sets of pins, will divide the rest of the pins into two subsequences, $S_1 = u_{j+k} \cdots u_i$ and $S_2 = u_{i+m} \cdots u_j$. If there is any partition which is represented in both of these sets of pins than no such cut is possible.

First assume that the two sets of pins $u_i \cdots u_{i+m}$ and $u_j \cdots u_{j+k}$ do not belong to the same set of self-loops: they are not connected by self-loops. Since the other ends of the self-loops are all in one of these two sets as well, there order is unimportant; we are free to rearrange $u_i \cdots u_{i+m}$ and $u_j \cdots u_{j+k}$ in whatever order is convenient. In this case a clean divide is possible since we can add the pins $u_i \cdots u_{i+m}$ and $u_j \cdots u_{j+k}$ to either S_1 or S_2 according to which partition they belong to. The remaining pins belong to partitions which consist only of pins from $u_i \cdots u_{i+m}$ and $u_j \cdots u_{j+k}$. These pins can all be added to S_2. Thus we divide $u_i \cdots u_{i+m}$ into two subsequences: the subsequence of pins belonging to partitions of S_1 and the remaining subsequence. We perform a tangle so that $u_i \cdots u_{i+m}$ is ordered with the subsequence of S_1 followed by the other subsequence. We do the same for $u_j \cdots u_{j+k}$ only reversing the order of the subsequences so that the sequence of pins corresponding to S_1 is after the other.

The tanglings that are performed in this case are called *untanglings* since they reorder the pins according to their connections to two disjoint sets of partitions without affecting the relative order of the pins which belong to the same partition. No additional interactions among the partitions are introduced by this type of tangling. The case in which the clean divide cuts through the both ends of a set of self-loops can not occur in a planar circuit; it would disconnect the planar circuit, and imply that the original planar circuit did not satisfy the reachability condition.

Thus to transform a planar circuit into a maximal-indivisible one, we perform operations in the following order, at each stage removing any trivial nodes that are created,

1. Remove trivial nodes.

2. Merge adjacent R-nodes.

3. Remove any trivial self-loops by inserting trivial nodes on them and then merging them into the node.

4. Examine each node with the procedure described above to determine whether it is indivisible and perform the necessary untanglings and clean divides.

The reason for obtaining a maximal-indivisible planar circuit is an issue which is related to the implementation of the procedure which will produce the layout from a planar circuit. If we have a procedure which transforms a planar circuit into a layout how will the particular choice of planar circuit from among homeomorphic ones to which we apply the procedure affect the layout? The underlying assumption is that such a procedure produces a layout which is

represented by, covered by, the given planar circuit. We make some assumptions on the cost functions which measure the layout and the procedures which produce the layout.

Assumption 1:

The cost function over the layout is an additive function of the components, B-nodes, and the implementations of R-nodes. Thus we can measure the cost of the layout obtained from a planar circuit $A=(P,IO,B,R,W)$ by

$$c(A) = \sum_{x \in B-nodes} C(x) + \sum_{u \in R-nodes} c(u)$$

where $C(x)$ is the cost associated with B-node x and $c(u)$ is the cost of implementing R-node u.

Assumption 2:

If u and v are two adjacent R-nodes then $c(u)+c(v) \geq c(M(u,v))$.

Assumption 3:

If z is an R-node which can be cleanly divided into u and v then $c(z) \geq c(u)+c(v)$.

Assumption 4:

If u is an R-node and T is an untangling of u as described above then $c(u) \geq c(T(u))$.

Assumption 5:

If u is a trivial R-node then $c(u)=0$.

At the topological level these are natural assumptions to make since the cost functions measure topological characteristics such as number of crossings required. At the geometrical level these assumptions imply that the procedures which generate

the layout implicitly examine the operations of gluing and uncleanly dividing nodes. Such an assumption is also quite natural since the procedure which transforms topology to geometry cannot be decomposed; it must consider the geometric interactions of neighboring sections of the layout. It is easier for the layout procedure to determine how to "glue" adjacent sections of the layout and to decompose R-nodes by uncleanly dividing them rather than to have to deal with non-maximal or divisible R-nodes. In the later, case the layout procedure would most likely perform operations which are in effect equivalent to performing merges and clean divides. We would like to be able to say that the costs of all homeomorphic maximal-indivisible planar circuits are the same but this requires an additional assumption. The problem is that even though a node is maximal-indivisible, performing tanglings on its self-loops can affect its internal wiring complexity. If we assume that the layout procedure is allowed to order the adjacent self-loops as it sees fit, then the costs of homeomorphic maximal-indivisible planar circuits are the same. Thus we assume that if u is a maximal-indivisible R-node and Re_+ is a complete refolding of it, then

$$c(Re_+(u)) = c(u).$$

Under this assumption it is immediate from Theorem 2.49 that homeomorphic maximal-indivisible planar circuits have the same cost. Thus we can show the following.

Lemma 2.51 If A is a planar circuit and A' is a homeomorphic maximal-indivisible planar circuit and then under the above assumptions $c(A') \le c(A)$.

Proof: The proof is by induction on the number of operations in a sequence taking A to A'. We first transform A into a maximal-indivisible planar circuit A'' by applying

the procedure described in this section. We apply merges, insertions, removals, clean divides and untanglings. By the assumptions, each one of these operations does not increase the cost of the planar circuit it is applied to. Hence $c(A'')\leq c(A)$. By Theorem 2.49, there is a sequence of complete refoldings taking A'' to A'. Since by assumption applying complete refoldings to maximal-indivisible R-nodes does not change their cost, we have $c(A'')=c(A')$ and $c(A')\leq c(A)$.

□

Summary

We have introduced planar circuits and shown that the concept of the 'planar topology' corresponds to an equivalence class of planar circuits under transformations which simulate local reorganization of wires. We have provided a normal form for planar circuits and shown its uniqueness within its equivalence class of planar circuits modulo the refolding operation. Under certain assumptions about the layout procedure we have shown that this normal form is optimal within respect to equivalent planar circuits.

CHAPTER 3

Mapping FP to Planar Circuits

In this chapter, we map behavioral specifications written in a functional language, FP, to planar circuits; a mapping is developed between these applicative programs and planar circuits. In Section 3.1 we briefly discuss FP and the properties which make it attractive as a specification language. Sections 3.2 and 3.3 discuss the correspondence between FP expressions and circuits, and the limitations of describing circuits in FP. Before developing the mapping to planar circuits, we discuss in Section 3.4, the pruning of planar circuits to remove unnecessary structure, that is, structure which cannot influence the behavior of the circuit. After describing the mapping from FP expressions to planar circuits in Section 3.5, we consider the mapping of the constructs necessary for describing synchronous sequential circuits in Section 3.6. Section 3.7 describes the implementation of the mapping which preserves the hierarchical representation of the planar circuit afforded by FP's combining forms. In Section 3.8 operations are applied to this hierarchical representation to transform the planar circuit into a homeomorphic maximal-indivisible planar circuit.

3.1 FP and its Salient Features

The FP language, as described in [Back78], consists of objects, primitive functions, and functional forms. FP objects are atoms (alphanumeric strings) and sequences of objects. Special significance is attached to the atom $\perp$ which is termed

"bottom" or undefined. If an object contains this atom it is said to be equivalent to it. FP functions are mappings of objects to objects. Combining forms map functions or objects to functions. Computations in FP are invoked by the application of a function to an object. Computations (functions) in FP are defined by constructing new functions from existing ones using the combining forms. The appendix contains a formal description of FP, including the lists of primitives and combining forms of the version of FP used in this thesis.

Below is an FP program (function) which computes the exclusive-or of two booleans using the nand operator.

Nand @ &Nand @ [[1,2],[2,3]] @ [1,Nand,2]

The '@' symbol represents the composition of functions from right to left. The '&' symbol represents the **Apply-to-All** combining form which applies a function (in this case the Nand) to each object in its input sequence and collects the outputs in a sequence. The '[','] ' represents the **Construct** combining form which applies each function enclosed between the brackets to the input, and collects the outputs in a sequence. **Nand** is the primitive function which computes the nand of two booleans in an input object, $<b_1,b_2>$. The functions, **1**, **2** and **3** are selector functions. The function **n** for any positive integer, n, selects the nth object in a sequence starting from the left. This program is composition of four functions, **[1,Nand,2]**, **[[1,2],[2,3]]**, **&Nand** and **Nand**. When this program is applied to the input object $<0,1>$, the first of these four functions is applied to it.

[1,Nand,2]: <1,0>

This results in the sequence

< **1**: <1,0>, **Nand**: <1,0>, **2**:<1,0>>.

Each of the functions in the list above are FP primitives which are applied to their

objects resulting in the following object.

<1,1,0>

The next of the four functions is applied to this object,

[[1,2],[2,3]]: <1,1,0>

which gives

< [1,2]: <1,1,0>, [2,3]: <1,1,0>> ,

< < 1: <1,1,0>, 2: <1,1,0>>, < 2: <1,1,0>, 3: <1,1,0>>> ,

<<1,1>,<1,0>>.

We then apply **&Nand** to this object,

&Nand: <<1,1>,<1,0>>

< **Nand**: <1,1>, **Nand**: <1,0>>

obtaining,

<0,1>.

The last function, **Nand**, is applied giving the final output object, 1.

Nand: <0,1>

1

Since there are no variables in FP, a function locates and identifies its arguments by their positions within its input object. This allows the definition of functions to be generic, independent of the size of their arguments. For example, a function which adds two bit vectors of any size can be defined.

Combining forms, such as **Compose** and **Construct**, specify precedences and parallelism among functions. Various algorithmic structures can also be made explicit by the use of a special form, although the same structure could be specified otherwise. The use of these forms allows these algorithmic structures to be

recognized and exploited. The combining forms can be used not only to specify the connectivity of the computation graph but its planar embedding; each combining form implies a planar organization of the computation graph of its subfunctions.

A computation can be viewed as consisting of two types of activities: directing data movement and affecting changes in value. In FP the delineation between these two types of activities is often explicit. This delineation is useful in the extraction of structural information from an FP function.

The only bindings in FP are those of functions to function names and these do not change during the application of a function. These bindings can be assigned by the user to establish a hierarchy within an FP function. This hierarchy can be exploited to make the extraction of structure more efficient. As will be discussed, the extraction process itself must be functional in order to exploit this hierarchy. This correspondence between behavioral and structural hierarchy is also expected to facilitate the simulation of the circuit.

Finally, the algebraic properties of FP offer the possibility of transforming an algorithm by applying algebraic identities to its FP specification. These transformations would affect the structure of a function without altering its input-output behavior and hence could be used to improve the algorithm.

3.2 Describing Circuits in FP

The concept of state does not exist within an FP program. Whatever information is needed for a computation must appear in the input of the function performing the computation. The result of a computation is otherwise independent of its environment. When a function is invoked (applied) it is evaluated and only its

output is retained. There is no other history of the execution of the function. These execution semantics make FP inappropriate for describing circuits in terms of low level circuit elements (i.e., transistors, resistors, capacitors) since the behavior of a circuit is the result of the time-dependent continuous interaction of these types of components. FP is appropriate for describing circuits whose behavior is the result of discrete interactions of elements which themselves have a behavior which can be described functionally (as a mapping of input values to output values). These elements are represented as boxes; the correspondence between these boxes and circuit elements must be established by the designer.

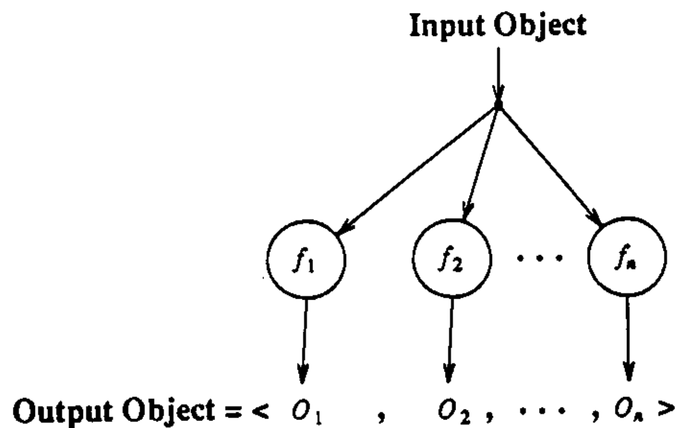
Since there are no states in FP, a sequential circuit could be described by a function which passes its state as an argument back to itself. Unfortunately this mechanism for describing sequential circuits presents the difficult task of determining whether the invocation of a function generates a new circuit or corresponds to an already implemented circuit for that function. In addition, it must be determined whether sequences are mapped into space or time. To avoid this difficult task, it is assumed that each invocation of a function corresponds to a new circuit. Feedback will be the result of the application of a form. Work along these lines has been described in [Shee84, Mesh84, Pate85]. The method we will use [Pate85], describes sequential circuits as the folding of combinational circuits, using the same structure to perform a computation in time rather than space. We will first consider the mapping from FP to combinational circuits and then extend this mapping by folding the circuit.

3.3 The Structural Implications of FP

The type of structure which must be captured from FP functions is 'boxes and wires;' FP functions should be representable as boxes with input and output wires. As illustrated in Figure 2.1, the FP combining forms interconnect and instantiate functions yielding planar graphs with functions as nodes along with their embeddings. The Construct also generates an R-node, but otherwise we can obtain a planar circuit which has B and R-nodes corresponding to FP primitive functions. Note that the conditional form does not appear in this figure; it will be discussed later.

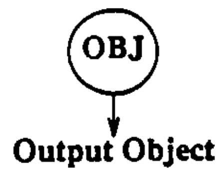
a) Compose $f_1@f_2@...@f_n$

b) Construct $[f_1, f_2, ..., f_n]$

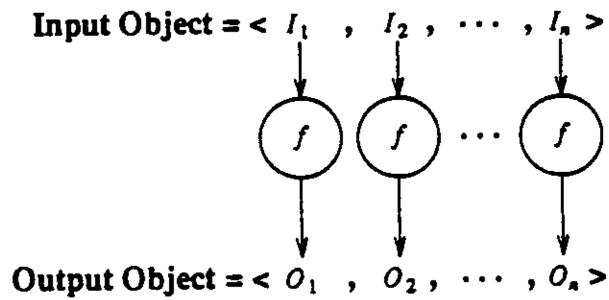


Output Object = $< o_1 , o_2 , \cdots , o_n >$

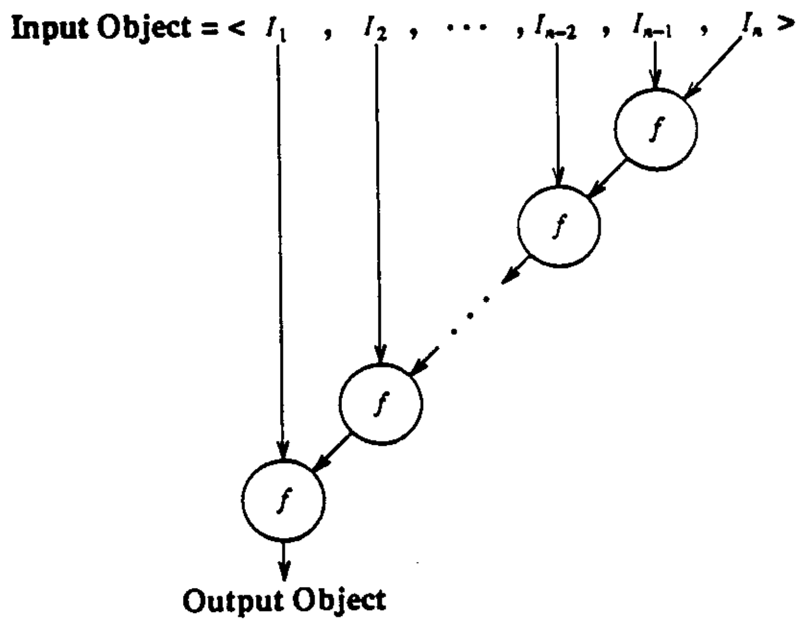
c) Constant %OBJ



d) Apply to All &f



e) Right Insert !f



f) Seq *seq(f)*

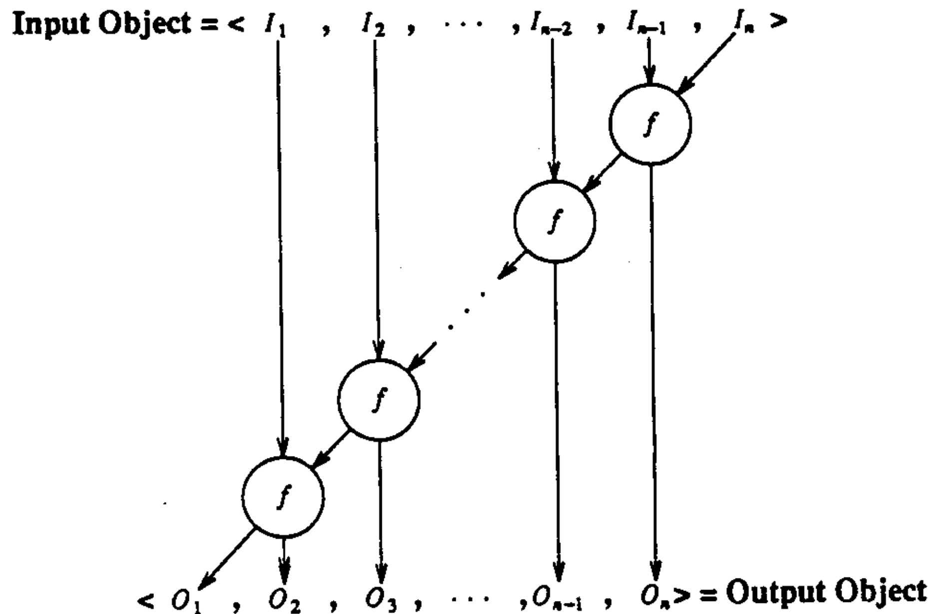


Figure 3.1. The structures of the FP combining forms

For the forms (**Apply-to-All, Right Insert, Seq**), the structure also depends on the object it is invoked with. To implement the connections represented by the arcs of the computation graph, the structure of the objects which will traverse an arc must be known. The amount of "structure" which can be extracted from an FP function without knowing its input is very limited. On the other hand the structure of an FP expression (a function applied to a specific object) can be completely determined. Clearly the structure of a function can not be extracted for each possible input object. Since the inputs to a circuit have some predefined structure, a more reasonable approach is to extract the structure of an FP function for some class of inputs over which it is invariant. A carry-propagate adder can be defined generically

(for any size inputs) in FP, but to obtain its structure the size of the input must be specified.

Two objects are said to be structurally equivalent if one can be obtained from the other by merely applying a substitution of labels. This need not be a consistent substitution; different labels can be substituted for various occurrences of the same label. A symbolic object is sufficient to represent a structural equivalence class of objects. No label is repeated in the symbolic object chosen as the representative, simply as a means of ensuring that only structural information is represented. The following are examples of equivalent symbolic objects.

<<<a , <b>> , c , <d , e>> , f>

<<<This , <is>> , a , <sym , bolic>> , object>

By using symbolic objects, the computation graph of an FP function can be derived. The symbolic output object generated by the FP combining forms can be determined from the symbolic outputs of their sub-functions and each node can be replaced by the structure of its corresponding function until only nodes corresponding to primitives remain. The resulting graph is the computation graph of the function. The replacement of nodes by the structure of their associated functions can be monitored to obtain a hierarchical representation of the computation graph. Even though the nodes corresponding to computational primitives could be drawn as boxes, this graph is still far from a 'layout' since the arcs transport arbitrary objects and the routing primitives are represented as nodes.

The symbolic objects associated with the arcs in the computation graph must be mapped into space (wires) and time. Each atom can be considered as a signal

which is representable on a wire in a unit of time. For other objects, a decision must be made as to whether sequences are mapped into time or space. The simplest decision is to always map into space; every atom of an object gets its own wire. However this may not be possible for some functions (e.g. iota), and may not be desirable for others. Initially we will map every atom to a separate wire.* This decision limits the class of circuits which can be described to the combinational circuits. However we will describe sequential circuits by folding combinational circuits.

The unit of information represented by an atom can be arbitrary; it reflects the level of abstraction desired in the representation of an FP expression. For example in a decoder each atom would most likely be a bit, while in an FFT each atom could represent a complex number. Once the level of representation of the atoms is fixed, the FP primitives of an FP expression can be classified into one of the following two categories.

Computational Primitives

These functions have the potential to generate atoms which are not atoms of the input object and/or their effect is determined by the value of input atoms (such as a comparator).

Routing Primitives

These functions never create new atoms and their effect is independent of the value of their input atoms. They merely rearrange the atoms within an

*There is a problem with <> since it can be considered to be both an atom and a sequence. The latter interpretation is chosen, and should be kept in mind while writing FP functions since otherwise the FP function may not have an "extractable structure."

FP object, possibly leaving some out and replicating others.

Routing Primitives can be executed on symbolic objects. Computational Primitives cannot and must be represented as black boxes; their output is a symbolic object with new labels. Computational primitives whose symbolic output object can not be determined from a symbolic input object can not be used. Note the absence of the function, iota, defined in Backus' original FP which generates the sequence of integers, $<1,2,\ldots,n>$, when it is applied to the positive integer n. Computational primitives are the primitive components of the layout, while routing primitives yield connectivity between intermediate input and output objects. The **Nand** in the example in Section 3.1 would fall into the first category, while the selector functions, **1**, **2** and **3**, would be considered routing primitives. Examples of other FP routing primitives include **trans**, **distl** and **reverse**.

The use of the **Conditional** must be restricted in order to extract the structure of an FP function. Two types of conditionals are permitted.

1. The first type acts as a switch; its output is either that of f or g, depending on the value of p. In this case the value of p depends on the particular atoms in its input object; p can take on different values for equivalent symbolic objects. Whenever this type is used, (p→f;g), f and g must produce structurally equivalent output objects for any symbolic input object they might receive. This type of conditional would yield the structure depicted in Figure 3.2. The function **Switch** should be provided by the designer depending on the logic required to perform the selection. However, a primitive called **Switch** is provided whose behavioral description is given by

the FP function, (1→2;3) and is represented as a box in the layout.

2. The second type of conditional is interpreted as structural control. The predicate must be based purely on structure (e.g. atom, null, =@[length,%3], etc.). The value of the predicate can be determined from the symbolic input object; it is independent of the value of the input atoms. In this case, the structure generated by (p→f;g) applied to an object is the structure of one of the two functions (f or g) applied to the object, depending on the value obtained from applying p to the object.

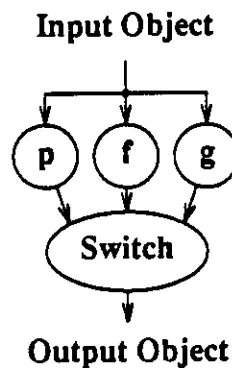


Figure 3.2 Realization of a Non-structural Conditional

Each invocation of a function results in a new implementation; all recursions are completely unfolded. Each recursion must be terminated by a structural predicate, a conditional of the second type, since otherwise we would obtain a circuit with the correct number of recursions for only the particular input values it is exercised on. With these restrictions, only acyclic computation graphs can be described; only combinational circuits can be generated from combinational primitives. The addition of new forms to describe sequential circuits is discussed in a later section.

3.4 Pruning Planar Circuits

In the next sections we will describe the mapping from FP expressions to planar circuits. Unfortunately, the FP routing primitives may generate *dead-end pins*, that is, pins which are not connected to any other pin in an R-node. In addition, the reachability condition, which required every pin to be circuit-connected to an IO-pin, may not be satisfied by the circuits generated from FP expressions, even though their plane graphs may be connected. These dead-end pins will be removed as well as components which are not circuit-connected to any input or output, to obtain a planar circuit. In the process, we can also remove pins which are not needed by the B-node to which they belong. We first address this pruning of planar circuits. We define a weaker form of planar circuit in which the partitions of its R-nodes may now contain only one pin, the plane graph, G_A is not necessarily connected and the reachability condition is omitted.

Definition 3.1

A ***weak planar circuit*** is a five-tuple $A = (P,IO,B,R,W)$ where P is a set of pins, IO is a sequence of pins, and B and R are sets of B and R-nodes respectively. Each B and R-node is a sequence of pins, and W is a pairing of the elements of P such that,

a. Each pin of P appears exactly once in IO, B or R.

b. The set of pins of each R-node, $u = \{u_i \mid i=1,...,m_u\}$ is partitioned into non-empty disjoint subsets, $P_u = \{P_{u,j} \mid j=1,...,c_u\}$ such that $|P_{u,j}| \geq 1$ for $1 \leq j \leq c_u$.

c. The component of the plane graph $G_A = (IO \cup B \cup R, E)$ where

$E = \{e_p = (u,v) \mid \text{for each } (u_i,v_j) \text{ in } W\} \cup \{(io_i,io_{i+1}) \mid 1 \le i < n\} \cup \{(io_n,io_1)\}$,

which contains the *IO*-pins can be embedded in the plane such that the clockwise cyclical order of edges around each node agrees with the sequence of pins of the corresponding *B* or *R*-node, and the clockwise traversal of the exterior window corresponds to *IO*.

d. The net connectivity graph, $G_N = (P \cup (\bigcup_{u,j} P_{u,j}), W \cup W')$, where W' contains an edge from each pin of an *R*-node to the partition of the *R*-node which contains the pin, $(u_i, P_{u,j})$, is acyclic.

Definition 3.2

A *dead-end* pin of a weak planar circuit is a pin of an *R*-node whose partition contains no other pins (has size one).

A weak planar circuit differs from a planar circuit in that dead-pins are allowed, the plane graph may not be connected and the reachability condition is not satisfied. Note that all of the *IO*-pins are in the same component of the plane graph since they are connected together by edges which form a path linking them. Thus other components of the plane graph do not contain any *IO*-pins and can hence not be circuit-connected to any of the *IO*-pins. We will discard these components, since any *B*-node or *R*-node which does not have a pin which is circuit-connected to an *IO*-pin can not influence the behavior of the circuit.

To obtain a planar circuit from a weak planar circuit, we must remove each dead-end pin, the wire attached to it, and the pin at the other end of this wire. This may create another dead-end pin or we may end up removing a pin of a *B*-node. If

this occurs some of the input pins of the B-nodes may no longer be required and we may wish to remove these pins, or the entire B-node if it no longer has any pins which are needed. In order to determine which of the pins of a B-node are needed, we assume that the pins of each B-node can be classified as either inputs or outputs of the B-node, and if a B-node has m inputs and n outputs, then we have a mapping,

$$f_B: \{0,1\}^n \rightarrow \{0,1\}^m,$$

which maps vectors of length n to vectors of length m in which a 1 in the ith position indicates that a pin is not required. We also assume that this mapping is monotonic in the sense that if additional output pins are determined not to be required then no unneeded input pin can become needed. That is,

$$\text{for } \overline{V}, \overline{U} \in \{0,1\}^n, \quad \text{if } \overline{V} < \overline{U} \text{ then } f_B(\overline{V}) \leq f_B(\overline{U}),$$

where "≤" is the partial order defined by component-wise comparisons. This will be the case for the B-nodes of planar circuits resulting from FP expressions since each B-node will corresponds to an FP primitive.

We also strengthen the reachability condition for planar circuits. In the original definition of planar circuits in Section 2.2, no distinction was made as to whether a particular IO-pin was an input or output of the circuit. In addition to the requirements of this definition, we now assume that each IO-pin is labeled as either an input or output pin. We will refer to IO-pins labeled as input and output pins, as I and O-pins respectively. We then replace the reachability condition of planar circuits by the following stronger condition,

Every pin is circuit-connected to at least one O-pin.

Thus we require that each pin have some path to at least one output IO-pin. Note that this reachability requirement has no bearing on our definition of weak planar

circuits since they have no reachability requirements at all. We need to mark the weak planar circuit in order to determine which pins should be removed.

Definition 3.3

A *marking* of a weak planar circuit $A=(P,IO,B,R,W)$, is a mapping of P, $M : P \rightarrow \{0,1\}$, such that for each wire, $(p,q) \in W$, $M(p)=M(q)$.

If $M(p)=1$ then p is said to be *marked*, and otherwise *unmarked*.

A pair (A,M) where M is a marking of A is a *marked weak planar circuit*.

We can define a partial ordering on the markings of a weak planar circuit as follows,

Definition 3.4

If M and M' are markings of a weak planar circuit, A, then $M \geq M'$ if for every pin p of A, $M'(p)=1 \Rightarrow M(p)=1$.

Definition 3.5

An unmarked pin is *markable*, if it belongs to one of the two following groups.

1. The pin belongs to a partition of an R-node in which all but one of the pins are marked. Note that pins of partitions of size one are always markable.

2. The pin is an input pin of a B-node which is not required to compute the currently unmarked output pins of the B-node.

Definition 3.6

A marking of a weak planar circuit which does not have any markable pins, is *maximal*.

If a marked weak planar circuit has a markable pin, then we can derive another marking by marking the pin and the pin at the other end of its wire. This operation can only be carried out a finite number of times since we must eventually run out of pins to mark. Note that the derived marking is strictly greater than the original marking since it has two additional marked pins. By repeatedly applying this operation, we must eventually derive a maximal marking of the weak planar circuit.

Lemma 3.7 If M_1 and M_2 are both maximal markings derived from a marked weak planar (A,M) then $M_1=M_2$.

Proof : The proof consists of the application of the Finite Church-Rosser Theorem [Newm42]. We simply need to show that the process of marking operations is finite and Church-Rosser, that is, if there are two operations which can be applied to the same marking then there are other operations which can be applied to these markings to transform them to the same marking. This is easily shown by observing that a pin which is markable, is either marked or still markable after a marking operation. If M_1 and M_2 are markings of A derived from M by marking p_1 and p_2 respectively, then either $M_1 = M_2$ or we can mark p_2 in M_1 and p_1 in M_2 to obtain in both cases M'. The Church-Rosser Theorem then tells us that for any two markings M_1, M_2, derived from a marking M, there exists a marking M', which can be derived from M_1 and M_2. It follows that if M_1 and M_2 are both maximal, we must have $M_1 = M_2$.

□

If we have a marked weak planar circuit, we can then prune it by removing all of the marked pins.

Definition 3.8

The *pruning* of a marked weak planar circuit (A,M), is the weak planar circuit obtained by omitting marked pins, and then removing any other pins, which cannot reach at least one O-pin. Any nodes and wires whose pins have all been removed are also removed.

We must argue that pruning a marked weak planar circuit does in fact result in a planar circuit. The embedding and acyclic net conditions are inherited from the weak planar circuit; removal of pins, nodes and wires do not affect these properties. The remaining property, the reachability condition holds by construction since we have defined the pruning so that only pins which are connected to some O-pin are retained. If there are no markable pins, there will be no partitions of size one in the pruned weak planar circuit. Hence the pruning of a maximally marked weak planar circuit is a planar circuit.

To obtain a planar circuit from a weak planar circuit A, it is sufficient to compute the maximal marking of (A,M_0) where $M_0(p)=0$ for all p. However we provide the designer with the added feature of being able to discard some of the output pins of the planar circuit. This feature can be exploited to simplify the specification. An example in Chapter 5 in which it will prove useful will be presented. Thus we will consider planar circuits obtained by pruning the maximal marking of the weak planar circuit with an output marking.

Definition 3.9

A marking M of A is an *output marking*, if $M(p)=1$ implies that p is an O-pin or is connected by a wire to an O-pin.

The weak planar circuits generated from FP expressions, will have additional properties which allow the computation of their maximal markings to be decomposed by the combining forms of the FP function. These weak planar circuits will have the property that we can label the pins of R-nodes as inputs and outputs as well as those of the B-nodes such that the following properties.

Definition 3.10

A weak planar circuit is *directable* if the pins of its R-nodes and B-nodes can be labeled as inputs or outputs of the nodes, R_{in}, R_{out}, B_{in} and B_{out} respectively, such that,

1. Wires connect only pins which are labeled R_{in}, B_{in} or O-pins to pins which are labeled R_{out}, B_{out} or I-pins.

2. Each partition of an R-node contains exactly one pin labeled R_{in}.

This leads to the following useful result about the markings of the directable weak planar circuits.

Lemma 3.11 If M_{μ} is an output marking of a directable weak planar circuit A, and M is a marking derived from from (A,M_{μ}), then all markable pins of (A,M) must be input pins of B or R-nodes.

Proof: The markable pins by definition are either input pins of B-nodes or pins belonging to partitions of R-nodes with only one unmarked pin. Thus it is sufficient to show that the last unmarked pin of each partition of an R-node must be an input pin of that R-node. This is established by showing that this property is true of the original marking and is preserved by the marking operation. Consider the original

marking, M_μ. The only pins marked in M_μ are some O-pins and pins connected to these marked O-pins. Since the wires from O-pins to R-node pins can not involve input pins of R-nodes, and each partition of an R-node has at least one input pin, this marking can not result in any partitions where the last unmarked pin is an output pin. Suppose now that the lemma holds for the original marking, $M_0=M_\mu$ and for the markings $M_0, \ldots, M_n$ where for $0 \le i \le n$, M_i is obtained from M_{i-1} by a marking operation. Suppose that M_{n+1} is obtained by marking a markable pin of M_n. The markable pin of M_n, p is by assumption the input of a B or R-node. The pin connected to the other end of its wire, q, must be either the output pin of a B or R-node or an I-pin. In the case where q is the output of a B-node or an I-pin, it is clear that the only pins in M' which can become markable are the inputs of B-node and hence M_{n+1} satisfies the lemma. The case in which q is the output of an R-node requires closer examination. Since p is unmarked, q must also be unmarked. Consider the partition to which q belongs. Suppose the input pin of this partition is marked. It could never have been previously markable since q is unmarked, and hence it must have become marked in some M_i as a result of marking the pin at the other end of its wire. But this pin would be the output of an R or B-node or an I-pin and none of these types of pins are markable pins or correspond to μ. Thus in fact the input pin of q's partition must be unmarked and hence marking q cannot create a markable output pin of an R-node.

□

We can now proceed with the mapping of FP expressions to weak planar circuits using the results of this section to map these weak planar circuits to planar circuits.

3.5 The Planar Circuit of an FP expression

The mapping is divided into two steps. We first map each FP expression to a weak planar circuit and then prune the maximally marked weak planar circuit derived from this weak planar circuit with some output marking to obtain a planar circuit. Remember that an FP expression is an FP function along with an FP object; it will be represented by $f ;x$. Note that this is not the same as $f :x$ which denotes the FP object obtained from the application of f to x. We use the trivial output marking M_0 by default. The mapping is defined by

$$\omega(f ;x, M_\mu) = \gamma(\sigma(f ;x), M_\mu),$$

where σ takes an FP expression to a weak planar circuit and γ maps a marked weak planar circuit to the planar circuit obtained by pruning the maximal marking derived from it. The function σ is defined for each of the FP primitives and then extended to other FP functions by defining it in terms of each combining form. We first describe σ. The weak planar circuits which will be generated will have their sequence of *IO*-pins arranged so that all of the *I*-pins occur before the *O*-pins; the inputs and outputs are not interleaved in the weak planar circuits generated by σ.

If x is an FP object we define the function $\rho(x)$ to be the sequence of atoms that x comprises in order from left to right; $\rho(x)$ is $\varnothing$ if x=<>, $\rho(x)$ is x if x is an atom and otherwise $\rho(x)$ is the flattening of x which is obtained by removing all of the brackets from x except the outermost pair. Remember that <> is not considered to be an atom. If x contains no atoms, then $\rho(x)=\varnothing$.

If $f :x = \perp$ then we define $\sigma(f ;x) = \perp$. Hence in the following we assume that $f :x \neq \perp$.

Primitive FP functions

As described earlier the primitive functions of FP can be classified as either computational or routing once they are applied to a particular object. If f is a primitive FP function then we define $\sigma(f\,;x)$ as follows.

If $f\,;x$ is computational, then $\sigma(f\,;x)=(P,IO,B,\varnothing,W)$ where P, IO, B and W are as follows.

$$P=\{i_1,\ldots,i_n,o_1,\ldots,o_m,x_1,\ldots,x_n,y_1,\ldots,y_m\},$$

$$IO=i_1\cdots i_n o_m\cdots o_1\ ,$$

$$B=\{x_1\cdots x_n y_m\cdots y_1\},$$

and

$$W=\{(i_j,x_j)\mid 1\le j\le n\}\ \dot{\cup}\ \{(o_j,y_j)\mid 1\le j\le m\}.$$

where $|\rho(x)|=n$, $|\rho(f\,:x)|=m$ and $n+m>0$. If $n+m=0$ then $\sigma(f\,:x)=P_{\varnothing}=(\varnothing,\varnothing,\varnothing,\varnothing,\varnothing)$, the empty planar circuit. See Figure 3.3a.

If $f\,;x$ is a routing primitive, we replace x by the symbolic object $symb(x)$ which is obtained by replacing each atom of x by a unique new atom. We let $y=f\,:symb(x)$ which is defined. By definition, since $f\,;x$ is a routing primitive, all of the atoms which occur in y also occur in $symb(x)$.

If $\rho(x)=\varnothing$ then $\rho(y)=\varnothing$ and then $\sigma(f\,:x)=P_{\varnothing}$, the empty planar circuit.

Otherwise $\rho(symb(x))$ is $x_1\cdots x_n$ for $n>0$ and $\rho(y)$ is $y_1\cdots y_m$ for $m\ge 0$. In this case, $\sigma(f\,;x)=(P,IO,\varnothing,R,W)$ where P, IO, R and W are as follows:

$$P = \{i_1, \ldots, i_n, o_1, \ldots, o_m, x_1, \ldots, x_n, y_1, \ldots, y_m\},$$

$$IO = i_1 \cdots i_n o_m \cdots o_1 \ ,$$

$$R = \{x_1 \cdots x_n y_m \cdots y_1\},$$

where the partitions of this R-node consist of one partition for each atom of $symb(x)$ along with all of the atoms of y which correspond to it,

$$\text{for } 1 \le i \le n \ , \ P_i = \{x_i\} \cup \{y_j \mid y_j = x_i\}$$

and

$$W = \{(i_j, x_j) \mid 1 \le j \le n\} \cup \{(o_j, y_j) \mid 1 \le j \le m\}.$$

It is clear that this also forms a planar circuit unless there is an atom of $symb(x)$ which does not appear in y. In this case, the input pin corresponding to this atom forms a partition of size one and $\sigma(f;x)$ is a weak planar circuit. Note that no two atoms of $symb(x)$ are in the same partition. This observation will be used to show the acyclic nets condition of the final weak planar circuit. Figure 3.3b contains the weak planar circuit defined by a computational primitive.

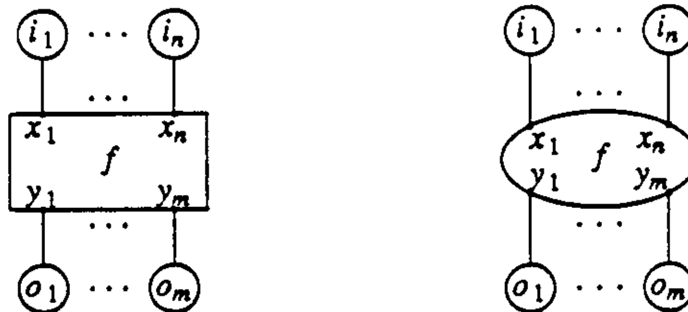


Figure 3.3 a) A computational primitive b) A routing primitive

To extend σ to other FP functions we must define σ in terms of each combining form of FP. The following properties of σ will be established by structural induction.

1. If $f:x$ is defined, then $\sigma(f;x)$ is a weak planar circuit with $IO = i_1 \cdots i_n o_1 \cdots o_m$ where the length of $\rho(x)$ is n and the length of $\rho(f:x)$ is m.

2. $\sigma(f;x) = P_{\varnothing}$ if and only if $\rho(x) = \rho(f:x) = \varnothing$.

3. The sequence of IO-pins is the concatenation of the subsequence consisting of I-pins and the subsequence consisting of O-pins. These subsequences will be referred to as I and O respectively.

4. In the net connectivity graph of $\sigma(f;x)$, G_N, no two I-pins are connected.

These properties clearly hold for the weak planar circuits defined by the FP primitives and we shall show them to be true for the weak planar circuits defined by the combining forms. The following notation is introduced.

Given a set of pins S, let $W(S)$ denote the wires which contain a pin belonging to S,

$$W(S) = \{(x,y) \in W \mid x \in S \text{ or } y \in S\}.$$

Composition $\quad f = f_n @ f_{n-1} @ \cdots @ f_1$

If $n=1$ then we define $\sigma(f;x) = \sigma(f_1;x)$. If $n>1$, let

$$A = \sigma(f_{n-1} @ \cdots @ f_1;x) = (P,IO,B,R,W)$$

and

$$A' = \sigma(f_n;(f_{n-1}@\cdots@f_1:x)) = (P',IO',B',R',W').$$

Both of these planar circuits are defined since $f:x$ is defined. If $\rho(x)=\varnothing$ and $\rho(f:x)=\varnothing$, then let $\sigma(f;x)=P_{\varnothing}$. Otherwise, $\sigma(f;x)=(P'',IO'',B'',R'',W'')$ where

$$P'' = P\cup P' - I' - O$$

$$B'' = B\cup B' \;,\quad R'' = R\cup R' \;,\quad IO'' = I''\cdot O''$$

where $I'' = I$ and $O'' = O'$ and,

$$W'' = W\cup W' - W(O) - W'(I') \cup \{(o_i, i'_{m-i+1}) \mid 1\le i\le m\}$$

where $m = |I'| = |O|$.

Figure 3.4 illustrates how the two planar circuits, A and A' are combined to obtain $\sigma(f;x)$.

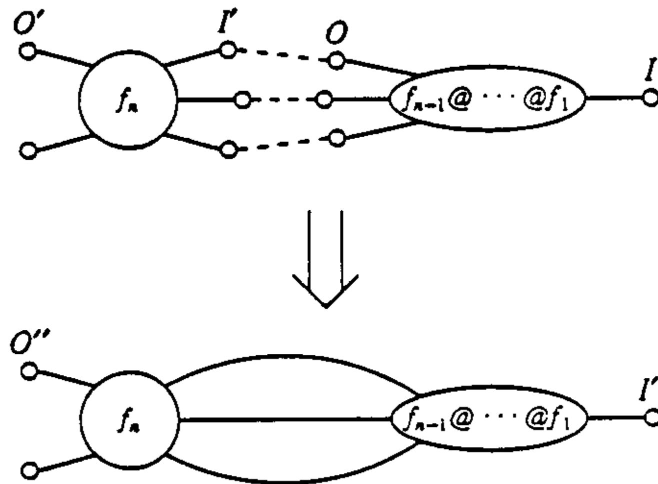


Figure 3.4

Clearly the component of plane graph containing the IO-pins can be embedded respecting the orderings of its B and R-nodes. The acyclicity of its nets is obtained from the observation that both A and A' have acyclic connectivity graphs and hence a cycle in $\sigma(f;x)$'s net connectivity graph can occur only if there are two I-pins of A'

which are connected. This is not the case, by induction. We must also argue that the same property holds for $\sigma(f;x)$, no two I-pins of $\sigma(f;x)$ are connected. If two such I-pins exist then either they are connected in A or there must exist two I-pins of A' which are connected in A'. Neither of these cases is possible by induction.

Construct $f = [f_1, \ldots, f_n]$

$f_i : x$ is defined for each $1 \leq i \leq n$ since $f : x$ is defined. Let $\sigma(f_i;x) = A_i = (P_i, IO_i, B_i, R_i, W_i)$. Suppose $\rho(x) = x_1 \cdots x_m$ and let $i_{j,k}$ be the jth input of A_k, and O_k denote the subsequence of O-pins of A_k. If $\rho(x) = \varnothing$ and $\rho(f:x) = \varnothing$, then let $\sigma(f;x) = P_{\varnothing}$. Otherwise, $\sigma(f;x) = (P, IO, B, R, W)$ where,

$$P = \bigcup_{i=1}^{n} P_i \cup \{x_1, \ldots, x_m\} \cup \{i_1, \ldots, i_m\}$$

$$IO = i_1 \cdots i_m O_n O_{n-1} \cdots O_1$$

$$B = \bigcup_{i=1}^{n} B_i \;, \qquad R = \bigcup_{i=1}^{n} R_i \cup \{u_D\}$$

where $u_D = x_1 \cdots x_m i_{m,n} \cdots i_{1,n} i_{m,n-1} \cdots i_{1,n-1} \cdots i_{2,1} i_{m,1} \cdots i_{1,1}$

and is partitioned into $P_{u_D,j} = \bigcup_{k=1}^{n} \{i_{j,k}\} \cup \{x_j\}$ for $1 \leq j \leq m$.

Finally,

$$W = \bigcup_{i=1}^{n} W_i \cup \{(x_j, i_j) \mid 1 \leq j \leq m\}.$$

Figure 3.5 illustrates the construction of $\sigma(f;x)$ from the planar circuits of the $\sigma(f_i;x)$'s. We create a new R-node, u_D to distribute the object x to each of the planar

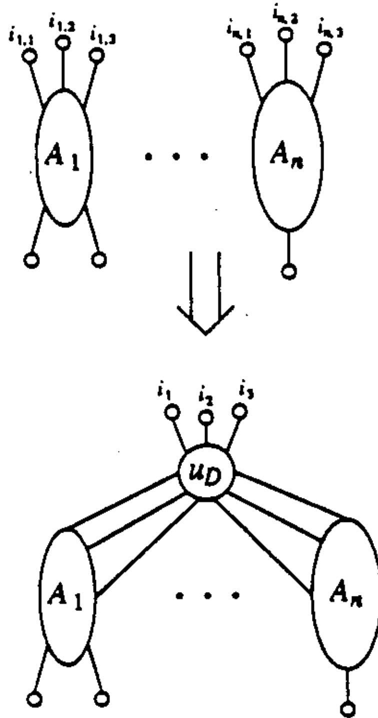


Figure 3.5

circuits defined by the f_i;x's. The original input pins of these planar circuits become part of the R-node u_D and new I-pins are created and connected to the input pins of u_D. The requirements of a weak planar circuit are clearly met in this case as well as the induction properties. No two I-pins can be connected since the A_i's remain disconnected except for the distribution of their inputs through u_D.

Constant $\quad f = \%Object$

The constant combining form is in effect a parameterized FP primitive. This form is treated as a computational primitive. We define $\sigma(\%Object{:}x) = \sigma(f\ {:}x)$ where $f\ {:}x$ is treated as a computational primitive such that $f\ {:}x = Object$. As is the case with computational primitives, $\rho(x) = \varnothing$ and $\rho(\%Object{:}x) = \varnothing$, then

$\sigma(\%Object;x) = P_{\varnothing}$.

Apply-to-All $\&f$

Since $\&f : x \neq \bot$, we must have $x = \langle x_1, \ldots, x_m \rangle$ for $m \geq 0$.

If $m=0$ then $\sigma(\&f;x) = P_{\varnothing}$

If $m>0$, then $f : x_i$ is defined for $1 \leq i \leq m$. Let $A_i = \sigma(f;x_i) = (P_i, IO_i, B_i, R_i, W_i)$. Let I_i and O_i denote the subsequences of I-pins and O-pins of A_i, respectively.

If $\rho(x) = \varnothing$ and $\rho(f : x) = \varnothing$, then let $\sigma(f;x) = P_{\varnothing}$. Otherwise, $\sigma(\&f;x) = (P, IO, B, R, W)$

$$P = \bigcup_{i=1}^{n} P_i \ ,$$

$$IO = I_1 I_2 \cdots I_m O_m O_{m-1} \cdots O_1$$

$$B = \bigcup_{i=1}^{n} B_i \ , \ R = \bigcup_{i=1}^{n} R_i \text{ and } W = \bigcup_{i=1}^{n} W_i$$

Figure 3.6 illustrates the construction of $\sigma(\&f;x)$ from the planar circuits of the $\sigma(f;x_i)$'s.

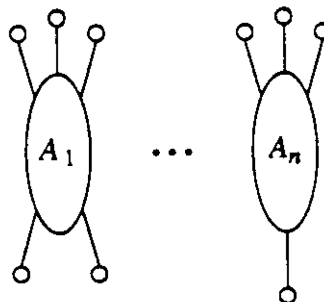


Figure 3.6

The embedding of the IO component of the plane graph as well as the acyclic nets

condition is assured since the weak planar circuits it comprises satisfy these conditions. The property that no two I-pins are connected clearly holds since, no new wires are introduced and two I-pins of the same A_i are not connected by assumption.

Right Insert $!f$

Suppose $x = <x_1, \ldots, x_n>$. If x were an atom $!f : x$ would not be defined.

If $n=0$ then we define $\sigma(!f ; x) = P_{\varnothing}$.

If $n=1$ then we define $\sigma(!f ; x) = \sigma(Id ; x_1)$ where Id is the identity function.

If $n=2$ then we define $\sigma(!f ; x) = \sigma(f ; x)$.

For $n>2$ we define $\sigma(!f ; x)$ as follows. Let

$$A = \sigma(!f ; <x_2, \ldots, x_n>) = (P, IO, B, R, W)$$

and

$$A' = \sigma(f ; <x_1, (!f : <x_2, \ldots, x_n>)) = (P', IO', B', R', W').$$

Both of these planar circuits are defined since $f : x$ is. If $\rho(x) = \varnothing$ and $\rho(f : x) = \varnothing$, then let $\sigma(f ; x) = P_{\varnothing}$. Otherwise let $y = !f : <x_2, \ldots, x_n>$, let I'_1 be the first $|\rho(x_1)|$ inputs of A' and let I'_2 be the other $m = |\rho(y)|$ inputs. Then $\sigma(!f ; x) = (P'', IO'', B'', R'', W'')$ where

$$P'' = P \cup P' - I'_2 - O \ ,$$

$$IO'' = I'_1 IO' \ , \ B'' = B \cup B' \ , \ R'' = R \cup R'$$

and if $I'_2 = i'_1 \cdots i'_m$, $O = o_m \cdots o_1$,

$$W'' = W \cup W' - W(O) - W'(I'_2)$$

$$\cup \{(p_j, p'_k) \mid \exists 1 \le h \le m : (p_j, o_{m-h+1}) \in W \text{ and } (p'_k, i'_h) \in W'\}$$

In Figure 3.7 A and A' are combined to form the planar circuit, $\sigma(!f ; x)$.

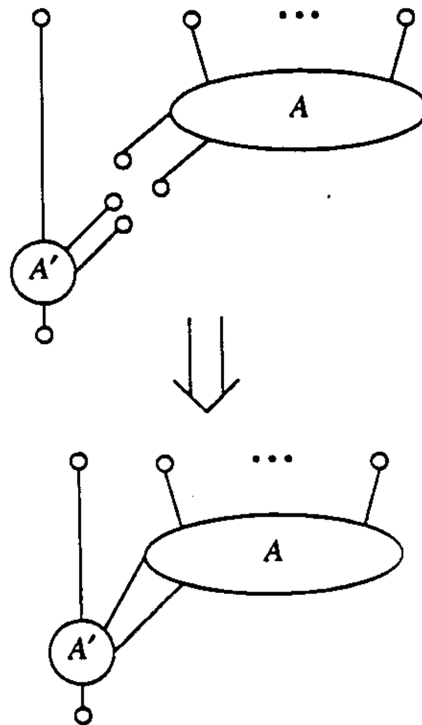


Figure 3.7

We remove the O-pins pins of A and the corresponding I-pins of A' and splice the wires connecting these pins together. The embedding of the IO component of the plane graph of this weak planar circuit is clear. The acyclic nets condition is assured since there is a cycle in the net connectivity graph of $\sigma(!f ; x)$ if and only if there is one in either A or A', or there are two I-pins of A' which are connected. Neither of these cases can occur. The property that two I-pins of $\sigma(!f ; x)$ are not connected is based on the observation that this would require either A or A' to have two connected I-pins.

Seq *seq*(f)

Suppose $x = <x_1, \ldots, x_n>$. If x were an atom $seq(f):x$ would not be defined.

If $n=0$ then we define $\sigma(seq(f);x) = P_{\varnothing}$.

If $n=1$ then we define $\sigma(seq(f);x) = \sigma(Id;x_1)$ where Id is the identity function.

If $n=2$ then we define $\sigma(seq(f);x) = \sigma(f;x)$.

For $n>2$ we define $\sigma(seq(f);x)$ as follows. Let

$$A = \sigma(seq(f);<x_2, \ldots, x_n>) = (P,IO,B,R,W)$$

and

$$A' = \sigma(f;<x_1,y_1>) = (P',IO',B',R',W'),$$

where $y = <y_1, \ldots, y_{n-1}> = seq(f):<x_2, \ldots, x_n>$. Both of these planar circuits are defined since $f:x$ is. If $\rho(x) = \varnothing$ and $\rho(f:x) = \varnothing$, then let $\sigma(f;x) = P_{\varnothing}$. Let I'_1 be the first $|\rho(x_1)|$ I-pins of A', let I'_2 be the other $m = |\rho(y_1)|$ inputs, and let O_1 be the m outputs of A corresponding to y_1 and O_2 the other outputs.

Then $\sigma(seq(f);x) = (P'',IO'',B'',R'',W'')$ where

$$P'' = P \cup P' - I'_2 - O_1 \ ,$$

$$IO'' = I'_1 IO_2 O' \ , \ B'' = B \cup B' \ , \ R'' = R \cup R'$$

and if $I'_2 = i'_1 \cdots i'_m$, $O_1 = o_m \cdots o_1$,

$$W'' = W \cup W' - W(O_1) - W'(I'_2)$$

$$\cup \{(p_j,p'_k) \mid \exists h: 1 \le h \le m \ , (p_j,o_{m-h+1}) \in W \text{ and } (p'_k,i'_h) \in W' \}$$

In Figure 3.8 A and A' are combined to form the planar circuit, $\sigma(seq(f);x)$. This is similar to the **Right Insert** except that we remove only the last m outputs pins of A and the corresponding inputs pins of A' and splice the wires connecting these pins.

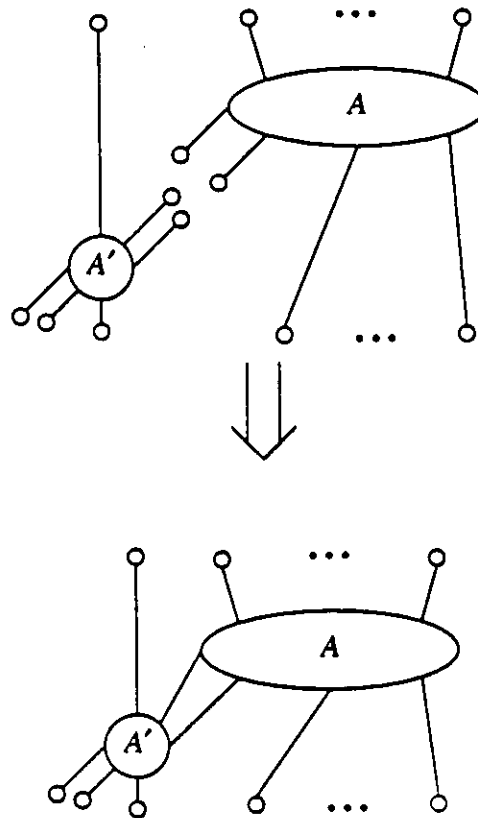


Figure 3.8

The embedding of the IO component of the plane graph, and acyclic nets conditions of this weak planar circuit are assured in the same manner as for the **Right Insert.**

Conditional $(p \rightarrow f;g)$

As discussed in the Section 3.3, we require the predicate of a conditional to depend only on the structure of the input object. Thus if $p:symb(x)$ is undefined then $\sigma(f;x)$ is also undefined. Otherwise $\sigma((p \rightarrow f;g);x) = \sigma(f;x)$ if $p:x = T$, and $\sigma(g:x)$ otherwise. This is the only case in which $f:x$ might be defined and $\sigma(f;x)$ might not.

This completes the definition of σ. As an example Figure 3.9 contains the weak planar circuit of the FP function presented in Section 3.1.

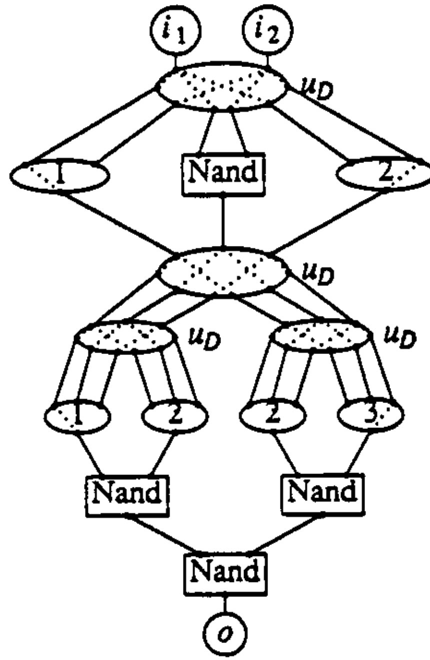


Figure 3.9 The weak planar circuit of **Nand @ &Nand @ [[1,2],[2,3]] @ [1,Nand,2]**

We now consider the computation of γ which prunes the weak planar circuit generated by σ using an output marking. In order, to compute γ we assume that for each B-node we have a mapping,

$$f_B: \{0,1\}^n \rightarrow \{0,1\}^m.$$

Remember that we assumed that f_B was provided with each B-node to indicate which inputs could be discarded given the set of outputs required. We impose the additional requirement that if none of the output pins are required, then none of the input pins are required, $f_B(\mathbf{1}^n)=\mathbf{1}^m$. The B-nodes which the result from the application of *%Object*, do not need their inputs to determine the output; we can always remove their input pins. The other B-nodes correspond to FP primitives, it

suffices to associate such a mapping with each FP primitive. We decompose the computation of ω over the FP expression in much the same manner as σ, obtaining the marking of the weak planar graph of each combining form by first computing the markings of the weak planar graphs of its subfunctions. In fact it will suffice to obtain the markings of the *IO*-pins of the weak planar circuits of the sub-functions in order to compute the marking of the weak planar circuit of a combining form. This gives a more efficient method of computing ω since we can avoid generating the structures of σ which γ would remove.

Definition 3.12

A *marking* of an FP object x, is an FP object, $\mu(x)$, such that if x is an atom then $\mu(x) \in \{1,0\}$ and otherwise $x = <x_1, \ldots, x_m>$ for $m \geq 0$ and $\mu(x) = <\mu(x_1), \ldots, \mu(x_m)>$.

The interpretation of $\mu(x)$ is that it replaces each atom of an FP object by 1 or 0 indicating whether the pin corresponding to the atom is marked or unmarked, respectively. Let $M_{\mu(f:x)}$ denote the output marking in which the *O*-pins corresponding to marked atoms in $\mu(f:x)$ are marked. We will use the notation $\omega(f;x,\mu(f:x))$ instead of $\omega(f;x,M_{\mu(f:x)})$ for the sake of brevity.

The properties of the directable weak planar circuits generated by σ will be exploited to realize the decomposition of ω. As discussed in the Section 3.4, we classify the pins of each *R*-node as inputs and outputs. This can be accomplished since each *R*-node is generated either as a result of an FP routing primitive or as the u_D of a **Construct**. If we have an *R*-node generated by the application of the routing primitive, $f:x$, then we consider the pins corresponding to $symb(x)$ as the inputs and those corresponding to $f:symb(x)$ as the outputs. In the latter case, when the *R*-node corresponds to u_D, we consider the pins which are connected to *I*-pins to be the

inputs and the remaining pins, the original inputs of the weak planar circuits of the subfunctions, to be the outputs. We can then establish the two properties required of directable weak planar circuits for the weak planar circuits generated by σ, by observing that they hold for the FP primitives and are preserved by the combining forms. We can then decompose the computation of the maximal marking of $\sigma(f;x)$ since Lemma 3.11 is guarantees that the output pins of B and R-nodes can only become marked as a result of marking the pin they are connected to by a wire.

We will show that in addition, the maximal markings of the weak planar circuits generated by σ from an arbitrary output marking have the *unmarked reachability* property which requires the existence for each unmarked pin, of a path consisting only of unmarked pins to some unmarked O-pin. This property simplifies the pruning, since after removing the marked pins each pin is still connected to some O-pin and hence no other pins, wires and nodes need be removed to assure the strong reachability of the planar circuit. Note that this property implies that all of the pins of a weak planar circuit are marked if it has no unmarked O-pins.

Composition $f = f_n @ f_{n-1} @ \cdots @ f_1$

If $n=1$, $\sigma(f;x) = \sigma(f_1;x)$ and $\omega(f;x,\mu(f:x)) = \omega(f_1;x,\mu(f_1:x))$. Otherwise $\sigma(f;x)$ is constructed from $A_{n-1} = \sigma(f_{n-1} @ \cdots @ f_1;x)$ and $A_n = \sigma(f_n;(f_{n-1} @ \cdots @ f_1:x))$. The wires in between A_n and A_{n-1} are from the inputs of B or R-nodes or O-pins of A_n to the outputs of B or R-nodes or I-pins of A_{n-1}; by Lemma 3.11 no pin of A_n can become markable as a result of marking a pin of A_{n-1}. Hence we can determine the maximal marking of A_n first, and use the markings of its inputs pins to obtain a marking for $\mu(f_{n-1} @ \cdots @ f_1:x)$ which can be used to mark the output pins of A_{n-1}. By connecting the outputs of $\omega(f_{n-1} @ \cdots @ f_1;x,\mu(f_{n-1} @ \cdots @ f_1:x))$ to the inputs

of $\omega(f_n;(f_{n-1}@ \cdots @f_1:x),\mu(f:x))$ $\omega(f;x,\mu(f:x))$ is obtained. In the case in which A_n had no I-pins and consequently A_{n-1} had no O-pins, all of A_{n-1} becomes marked. If A_n and A_{n-1} both have the property that each unmarked pin is connected to some unmarked O-pin, then so will $\omega(f;x,\mu(f:x))$.

Construct $[f_1, \ldots, f_n]$

In this case $\sigma(f;x)$ is constructed from the $\sigma(f_i;x)$'s by joining their inputs at an R-node, u_D. Since the original pins of the $\sigma(f_i;x)$'s become output pins of u_D, they are never markable. Thus we can determine the markings of the $\sigma(f_i;x)$'s and then use the markings of their input pins to mark u_D. In this case, $f:x = \langle y_1, \ldots, y_n \rangle$ and so we compute $\omega(f;x,\mu(f:x))$ by first computing for $1 \le i \le n$, $\omega(f_i;x,\mu(f_i:y_i))$. The markings of their I-pins, gives us n markings of x, $\mu_i(x)$. These markings are used to mark the node u_D. We mark any input pin of u_D and also the corresponding I-pin if and only if the atom corresponding to it in each $\mu_i(x)$ is marked. If each $\omega(f_i;x,\mu(f_i:y_i))$ has the property that an unmarked pin has an unmarked path to an unmarked O-pin then, clearly $\omega(f;x,\mu(f:x))$ will have this property as well since an unmarked I-pin is connected to an unmarked input pin of u_D which must be connected to an unmarked pin which was an I-pin of some $\omega(f_i;x,\mu(f_i:y_i))$.

Constant *%Object*

The constant combining form results in a planar circuit consisting of a B-node. The output of the constant combining form is the same object regardless of its input object (unless this is $\perp$). Hence we can discard its inputs without affecting its outputs. This B-node has the property that all of its input pins are markable

regardless of the marking of its outputs. Thus the maximally marked weak planar circuit of $(\sigma(\%Object;x),M)$ for any marking has no unmarked I-pins. It trivially satisfies the reachability property.

Apply-to-All $\&f$

If $m=0$ then $\sigma(\&f;x)=P_\varnothing$ and the same is true of $\omega(f;x,\mu(f:x))$. If $m>0$, $\sigma(\&f;x)$ is constructed from the $\sigma(f;x_i)$'s. In this construction no wires or nodes are added. The maximal marking of $(\sigma(\&f;x),\mu(\&f:x))$ corresponds to the maximal markings of the $(\sigma(f;x_i),\mu(f:x_i))$'s. We can construct $\omega(\&f;x,\mu(\&f:x))$ from the $\omega(f;x_i,\mu(f:x_i))$'s in the same manner as we constructed $\sigma(\&f;x)$ from the $\sigma(f;x_i)$'s. If each $\omega(f;x_i,\mu(f:x_i))$ has the reachability property then $\omega(\&f;x,\mu(\&f:x))$ will have it.

Right Insert $!f$

$x=<x_1,\ldots,x_n>$.

For $n=0$ we have $\sigma(!f;x)=P_\varnothing$ so $\omega(!f;x,\mu(!f:x))$ must be $P_\varnothing$ as well.

For $n=1$ we have $\sigma(!f;x)=\sigma(Id;x_1)$ and so $\omega(!f;x,\mu(!f:x))=\omega(Id;x_1,\mu(Id:x_1))$.

For $n=2$ we have $\sigma(!f;x)=\sigma(f;x)$ and so $\omega(!f;x,\mu(!f:x))=\omega(f;x,\mu(f:x))$.

For $n>2$ we $\sigma(!f;x)$ was constructed from $A=\sigma(!f;<x_2,\ldots,x_n>)$ and $A'=\sigma(f;<x_1,(!f:<x_2,\ldots,x_n>))$ by connecting the outputs of A to left inputs of A'. As in the case of **Composition**, we can obtain the maximal marking of A' first since none of the pins of A which could be markable are connected to pins of A'. Hence we obtain $\omega(f;<x_1,(!f:<x_2,\ldots,x_n>),\mu(!f:x))$ and use the markings of its inputs pins as the marking for $y=!f:<x_2,\ldots,x_n>)$. This corresponds to the markings of the

O-pins of $\sigma(!f : \langle x_2, \ldots, x_n \rangle)$. Thus we can construct $\omega(!f; \langle x_1, \ldots, x_n \rangle), \mu(!f : x))$ from $\omega(f; \langle x_1, (!f: \langle x_2, \ldots, x_n \rangle), \mu(!f : x))$ and $\omega(!f; \langle x_2, \ldots, x_n \rangle), \mu(y))$ in the same manner as we constructed $\sigma(!f ; x)$. If both of these planar circuits have the reachability property, then so will $\omega(!f; \langle x_1, \ldots, x_n \rangle), \mu(!f : x))$.

Seq *seq(f)*

The argument for the $\omega(seq(f);x)$ is the same as the argument for $\omega(!f ; x)$.

Conditional $(p \rightarrow f;g)$

If it is defined, $\sigma(p \rightarrow f ; g ; x)$ is either $\sigma(f ; x)$ or $\sigma(g ; x)$. In the former case, $\omega(p \rightarrow f ; g ; x)$ is $\omega(f ; x)$ while in the latter, $\omega(g ; x)$.

It remains to define ω on the FP primitives. We first compute σ and then find the maximal marking derived from $M_{\mu(f : x)}$. By our assumption that $f_B(\mathbf{1}^n)=1^m$ we know that the reachability property is satisfied if $f ; x$ is a computational primitive. It is also satisfied in the case of routing primitives, since the only pins which would not be connected to some O-pin would be in a partition in which all other pins would be marked. Thus we generate $\sigma(f ; x)$, and define $\omega(f ; x, \mu(f : x))$ to be the planar circuit obtained from the maximal marking of $(\sigma(f ; x), M_{\mu(f : x)})$ by removing the marked pins. Note that this may result in $P_{\varnothing}$. The planar circuit generated in this case clearly satisfies the reachability property.

This completes the definition of the mapping from FP expressions to planar circuits which are combinational. Before extending the mapping to sequential circuits in the next section, we present in Figure 3.10 the planar circuit generated

from the FP function presented as an example in Section 3.1. The dashed lines are wires which along with their pins have been removed.

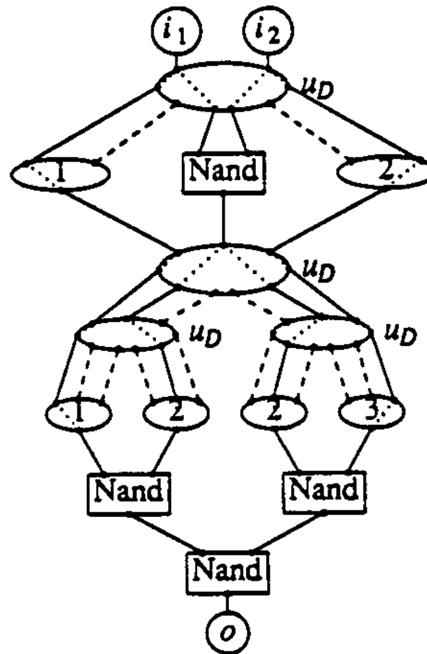


Figure 3.10 The planar circuit of **Nand @ &Nand @** [[**1**,**2**],[**2**,**3**]] **@** [**1**,**Nand**,**2**]

3.6 Sequential Circuits

We have shown how to map FP expressions to planar circuits which are combinational circuits; we can implement FP functions in space. In this section we extend the mapping in order to describe sequential circuits in FP. The extensions are from [Pate85]. Intuitively, using FP to describe sequential circuits does not seem appropriate since there is no concept of state in FP. However if we view the decision of whether to implement a particular computation in space or time as an implementation decision rather than an algorithmic one, FP seems highly appropriate since we can describe a computation without committing ourselves to either a space or time implementation. The description of the computation is

independent of the time/space implementation decision.

We have shown that all FP expressions under restrictions previously described, have a space implementation. In mapping these FP expressions to combinational circuits, we mapped each atom to a separate wire. For sequential circuits, we must map several FP expressions to the same structure. To obtain sequential circuits, we implement the combining forms, **Apply-to-All**, **Right Insert**, and **Seq** in time rather than space. Not all instances of these combining forms can be *folded* from space into time; the weak planar circuits from which they are constructed must be isomorphic. From the construction of σ, it follows that if each computational primitive f has the property that $\sigma(f;x) \equiv \sigma(f;y)$ for structurally equivalent FP objects, x and y, then this property holds for any FP function. This is the case for FP primitives. Thus if x and y are structurally equivalent, $\sigma(f;x) \equiv \sigma(f;y)$. Figures 3.11, 3.12 and 3.13 illustrate the folding of the **Apply-to-All**, **Right Insert**, and **Seq** combining forms, respectively. Note that the connections in these figures may correspond to several wires.

In order to fold these combining forms we must first pass their input objects which are implemented in space, through a converter which implements their sequence of input objects in time rather than space. In the case of **Apply-to-All** and **Seq** we must convert the sequence of output objects back into a space sequence. The new FP primitives, **SOPI** (Serial Out Parallel In), **POSI** (Parallel Out Serial In), illustrated in Figure 3.14 perform these conversions.

We must discuss timing in order to assure the correct operation of the circuit. In FP, the application of a function to an object is an atomic event. With the introduction of **SOPI** and **POSI**, this is no longer the case. This poses a problem in

Apply-to-All &f

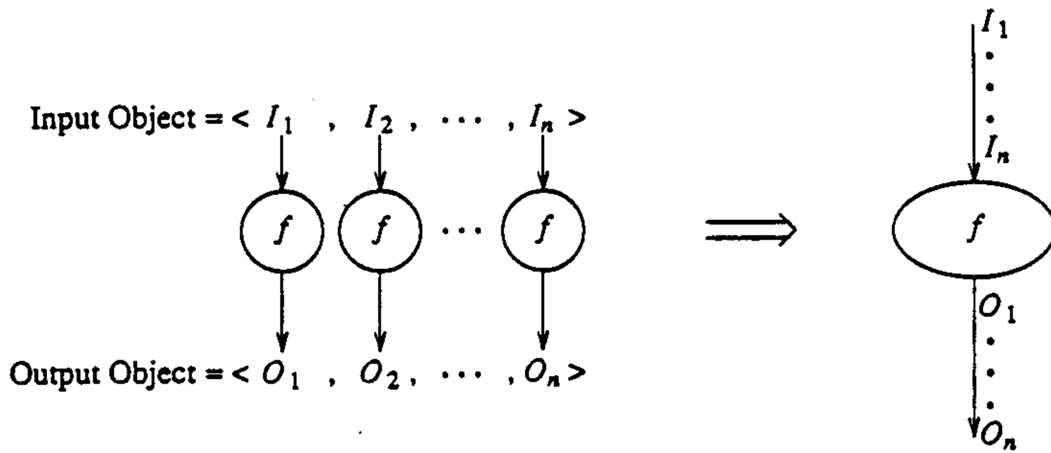


Figure 3.11 The *folding* of **Apply-to-All**

Right Insert !f

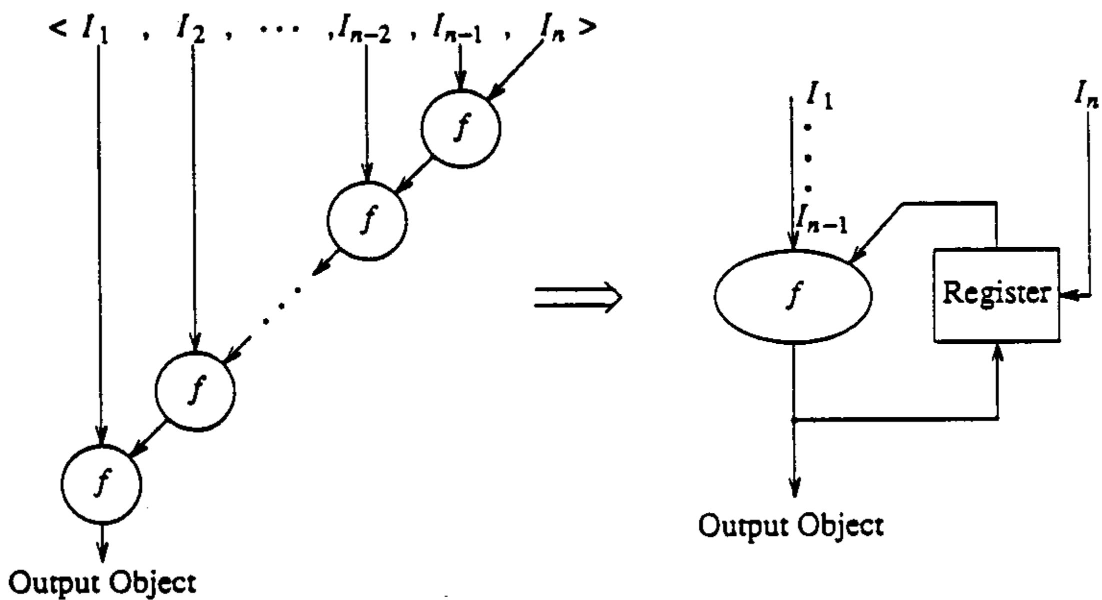


Figure 3.12 The *folding* of **Right Insert**

Seq *seq(f)*

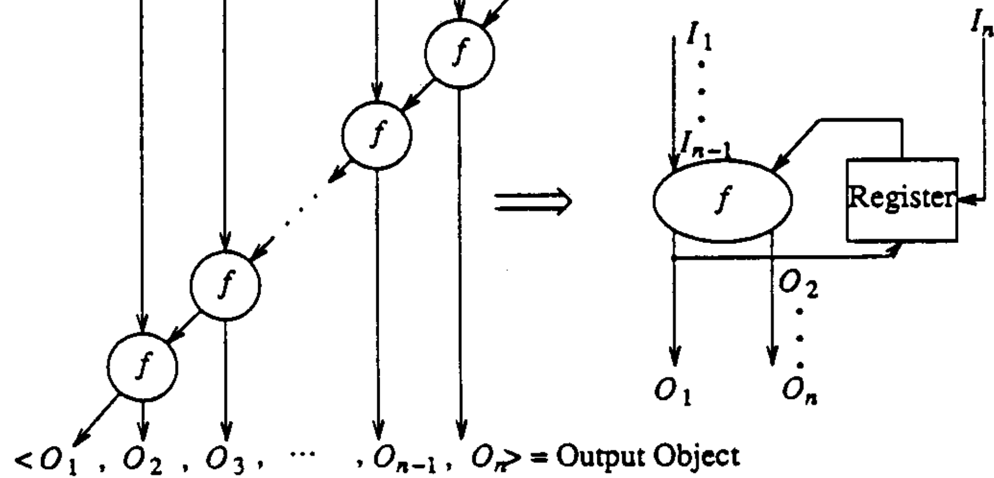


Figure 3.13 The *folding* of Seq

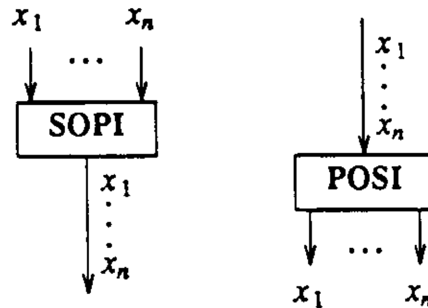


Figure 3.14 The space/time converters SOPI and POSI

synchronizing computations which are occurring in parallel, in separate sub-functions of a **Construct** or an **Apply-to-All** for example. In order to maintain the synchronization, we introduce the delay primitives, D and D^{-1} which delay the signals by one clock pulse and advance the signals by one clock pulse respectively. Each object has a clock pulse associated with it and the output of the application of a function to that clock pulse has the same clock pulse associated with it. In this sense each FP function remains combinational. In order to maintain these timing

semantics, when we introduce a SOPI, POSI pair we also add a D^{-1} to restore the atomicity of the computation. Of course, we can not implement D^{-1}; instead we transform it until it sits at the end of the FP specification where it indicates the number of clock delays required by the function. In the process of pulling the D^{-1}'s out, we insert D's on parallel branches. The folding of combinational implementations is accomplished by a set of transformations. The transformations, introduced in [Pate85], modify the FP specification by introducing new functions and then replacing the combining forms **Apply-to-All**, **Right Insert**, and **Seq** by their sequential equivalents.

$$\&f \equiv D^{-1} @ \text{POSI} @ \&^{T} f @ \text{SOPI}$$

$$!f \equiv D^{-1} @ !^{T} f @ \text{apndr} @ [\text{SOPI} @ \text{tlr}, \text{last}]$$

$$seq(f) \equiv D^{-1} @ \text{apndl} @ [1, \text{POSI} @ \text{tl}] @ seq^{T}(f) @ \text{apndr} @ [\text{SOPI} @ \text{tlr}, \text{last}]$$

The functional behavior of these new primitives is equivalent to the *Id* primitive, and the sequential versions of the combining forms are the same as their combinational versions. These new FP primitives and combining forms only differ under ω, the mapping to planar circuits. The intent is not that these forms be used by the designer, but rather that they result from transformations applied to the FP functions. In this manner we obtain a superset of FP, in which each function can be mapped to FP by replacing these new primitives by *Id* and the combining forms by their original forms.

σ can be easily extended to interpret of the sequential versions of the combining forms, but their interpretation by ω is more complicated. This is because by folding a combining form, we are performing the computations of one function

applied to a set of input objects using the same physical structure. In the space implementation we generated a separate weak planar circuit for each input and marked these separately according to the markings of their respective outputs. For a time implementation we must mark the weak planar circuit of the function applied to only one input, such that it represents the union of the planar circuits that would result from the application of the function to each input object. We provide their mappings under ω directly.

Each of the new FP primitives, **SOPI**, **POSI**, **D** for a one clock pulse delay and $\mathbf{D^{-1}}$ for a one clock pulse inverse delay, is interpreted as *Id* in terms of behavior. However their interpretation in terms of structure is different. They are interpreted by ω as follows.

SOPI

SOPI converts a space sequence to a time sequence. It assumes that its input is of the form, $x = \langle x_1, \ldots, x_n \rangle$ in which each x_i is structurally equivalent. If this is not the case then $\omega(SOPI:x, \mu(SOPI:x))$ is undefined. Since $SOPI:x = x$, we have $\mu(SOPI:x) = \mu_{out}(x) = \langle \mu_{out}(x_1), \ldots, \mu_{out}(x_n) \rangle$. It is also assumed that all of x_i's except for x_1 have all of their atoms marked. A *B*-node will be generated for **SOPI** if there is at least one unmarked atom in $\mu_{out}(x_1)$. $\mu_{in}(x)$ is $\langle \mu_{in}(x_1), \ldots, \mu_{in}(x_n) \rangle$ where each $\mu_{in}(x_i)$ has a marked atom if and only if the corresponding atom in each $\mu_{out}(x_1)$ is also marked. *I* and *O*-pins are generated for each unmarked input and output pin of the *B*-node and connected by wires to these.

POSI

As with **SOPI**, $\omega(POSI;x, \mu(POSI:x))$ is defined only if $x = <x_1, \ldots, x_n>$ in which each x_i is structurally equivalent. **POSI** converts a time implementation of a sequence back into a space implementation.

$$\mu(POSI:x) = \mu_{out}(x) = <\mu_{out}(x_1), \ldots, \mu_{out}(x_n)>.$$

A B-node will be generated for **POSI** if there is at least one unmarked atom in $\mu_{out}(x)$. $\mu_{in}(x)$ is $<\mu_{in}(x_1), \ldots, \mu_{in}(x_n)>$ where each $\mu_{in}(x_1)$ has a marked atom if and only if the corresponding atom in each $\mu_{out}(x_j)$ is also marked and $\mu_{in}(x_i)$ for $1<i\leq n$ has all of its atoms marked. As in the case of **SOPI**, a planar circuit with one B-node labeled **POSI** is generated.

The sequential versions of the combining forms are also dealt with differently by ω. In each case ω will generate one copy of $\sigma(f;x)$ and in the case of **Right Insert** and **Seq** the output will be fed back into some of the inputs through a **Register**. Computing ω in this case is more involved, since a pin should be marked only if each application of the function does not require it. We introduce the notion of products of markings to handle this case. If we have two markings M, M', of a weak planar circuit A, then we define $M \cdot M'$ to be the marking in which $M \cdot M'(p) = M(p) \cdot M'(p)$; a pin is marked if and only if it is marked in both M and M'. In the same manner, if we have structurally equivalent objects $x_1, \ldots, x_n$ and markings of each, $\mu_1, \ldots, \mu_n$, we define the product of these markings to be the marking in which an atom is marked if and only if all of the corresponding atoms of in the markings of $\mu_1, \ldots, \mu_n$ are marked. This gives us the following rather strange identity.

$$M \leq M_1 \cdot M_2 \text{ if and only if } M \leq M_1 \text{ and } M \leq M_2$$

We can then show the following.

Lemma 3.13 If M and M' are maximal markings of a weak planar circuit, A, then $M \cdot M'$ is also a maximal marking.

Proof: If there is a markable pin in $M \cdot M'$ then it must be unmarked in either M or M'; assume it is unmarked in M without loss of generality. If it is markable in $M \cdot M'$ because it is the only unmarked pin in its partition then all other pins in this partition are marked in $M \cdot M'$ and hence they must be marked in M as well. If it is markable because it belongs to a B-node whose output markings indicate that it can be marked, then these output pins of this B-node must also be marked in M. Thus if a pin is markable in $M \cdot M'$ it must be markable in either M or M'. It follows that if both M and M' are maximal markings, have no markable pins, then $M \cdot M'$ must also be maximal.

□

If $\mu(y)$ is a marking of the output pins of A, then let $M^0{}_\mu$ denote the marking in which the output pins of A corresponding to the atoms marked in μ are marked. Let M_μ denote the maximal marking derived from $M^0{}_\mu$. We then have the following.

Lemma 3.14 If μ and μ' are markings of the output pins of a weak planar circuit, A, then $M_\mu \cdot M_{\mu'} = M_{\mu \cdot \mu'}$.

Proof: Clearly $M^0{}_\mu \geq M^0{}_{\mu \cdot \mu'}$. Hence $M_\mu \geq M_{\mu \cdot \mu'}$ since the set of markable pins in $M^0{}_{\mu \cdot \mu'}$ is a subset of the markable pins of $M^0{}_\mu$ and we can maintain this property in the derivations of their respective maximal markings by marking the only pins which are markable in the the former. Hence $M_\mu \geq M_{\mu \cdot \mu'}$ and the same is true for μ', $M_\mu{}' \geq M_{\mu \cdot \mu'}$. We then have $M_\mu \cdot M_{\mu'} \geq M_{\mu \cdot \mu'}$ by the identity.

To show $M_{\mu} \cdot M_{\mu'} \leq M_{\mu \cdot \mu'}$ we consider the derivation of $M_{\mu \cdot \mu'}$. We show that a pin p of A which becomes markable in this derivation must be marked in $M_{\mu} \cdot M_{\mu'}$. We rely on the observation in the preceding lemma that if $M' \leq M$ then any markable pin of M' is either markable or marked in M. If M is maximal then the maximal marking derived from M' cannot be greater than M, since each markable pin must already be marked in M and hence marking it gives a new marking which is no greater than M. We have $M^0_{\mu \cdot \mu'} \leq M^0_{\mu}$ and $M^0_{\mu \cdot \mu'} \leq M^0_{\mu'}$. This gives $M^0_{\mu \cdot \mu'} \leq M_{\mu}$ and $M^0_{\mu \cdot \mu'} \leq M_{\mu'}$ which implies that $M^0_{\mu \cdot \mu'} \leq M_{\mu} \cdot M_{\mu'}$. Using the observation, we then have $M_{\mu \cdot \mu'} \leq M_{\mu} \cdot M_{\mu'}$.

□

Time Apply-to-All $\quad \&^T f$

The assumption made here is that the input $x = <x_1, \ldots, x_n>$ arrives as a time sequence on a set of wires. The x_i's are structurally equivalent. In this case, we implement $f ; x_i$ only once since $\&^T f$ will be accomplished sequentially in time rather than in parallel in space. We generate $\sigma(f ; x_i)$ for some i and then we must determine a marking sufficient for each $f ; x_i$. By the previous lemma we know that

$$\prod_{i=1}^{n} M_{\mu(f : x_i)} = M_{\prod_{i=1}^{n} \mu(f : x_i)}$$

Thus we define $\omega(\&^T f ; x, \mu(x))$ to be $\omega(f ; x_1 ; \mu^*(f : x_1))$ where

$$\mu^*(f : x_1) = \prod_{i=1}^{n} \mu(f : x_i).$$

The choice of x_1 is arbitrary and does not matter since the x_i's are assumed to be structurally equivalent.

Time Right Insert $!^T f$

The assumption in this case is that $x = <x_1, \ldots, x_n>$ for $n > 2$, the x_i's for $1 \leq i < n$ are structurally equivalent and x_n is structurally equivalent to each $y_i = !f :<x_i, \ldots, x_n>$ which is the structure of the intermediate results as well as the final output. The structure generated in this case is essentially that of $\sigma(f ;<x_1, y_2>$. See Figure 3.12. We connect the O-pins of $\sigma(f ;<x_1, y_2>$ to an R-node which duplicates each output. One copy of the outputs is connected to new O-pins and the other copy is connect the inputs of a B-node which corresponds to a register. The output of this register is connected to the I-pins of $\sigma(f ;<x_1, y_2>$ which correspond to y_2. I-pins corresponding to x_n are also connected to this register; they provide the register with its initial value. To calculate $\omega(!f ;x, \mu(!f :x))$ we must first determine the maximal markings $(\sigma(f ;<x_i, y_{i+1}>, \mu(y_i))$. $\mu(y_1) = \mu(!f ;x)$ and $\mu(y_i)$ for $1 < i \leq n$ corresponds to the markings of the inputs in the maximal marking $M_{\mu(y_{i-1})}$. Let $In(M)$ denote the markings of these inputs for a marking M. The desired marking would then be

$$\prod_{i=1}^{n} M_{\mu(y_i)} = M_{\prod_{i=1}^{n} \mu(y_i)}.$$

This would require obtaining each $\mu(y_i)$. A more efficient method of computing this marking is to take the product of each $In(M_\mu)$ with μ and repeat $n-1$ times or until the markings converge, no new inputs become unmarked. We let $\mu_1 = \mu(!f :x)$ and define $\mu_{i+1} = In(M_{\mu_i}) \cdot \mu_i$. We need to show that $M_{\mu_n} = M_{\prod_{i=1}^{n} \mu(y_i)}$. This is accomplished by showing that $\mu_k = \prod_{i=1}^{k} \mu(y_i)$. This is true by definition for $k=1$. If we assume it is true for k then we can show it to be true for $k+1$ as follows. Since

$$\mu_k = \prod_{i=1}^{k} \mu(y_i),$$

$$\mu_{k+1} = In\,(M_{\mu_k}) \cdot \mu_k,$$

$$= In(M_{\prod_{i=1}^{k}\mu(y_i)}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= In\,(\prod_{i=1}^{k} M_{\mu(y_i)}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= \prod_{i=1}^{k} In\,(M_{\mu(y_i)}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= \prod_{i=2}^{k+1} \mu(y_i) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= \prod_{i=1}^{k+1} \mu(y_i).$$

If for $k<n$, $\mu_k = \mu_{k+1}$ then $\mu_k = \mu_n$. Although there must be k for which this occurs since $\mu_k \le \mu_{k-1}$, it may not occur for $k<n$. This will occur when the length of the object to which the sequential version of $!f$ is applied, is not sufficient to completely exercise all of the structure of f. We choose to define the structure of $!f$ in such a manner that will accommodate any number of iterations rather than the number specified by the actual input object. The reasons for this are twofold. First we may wish to remove the **SOPI** preceding the **Right Insert** in the specification in order to operate on a stream of objects. Defining ω in this way allows us to do this. Secondly, in extracting the planar circuit, different sequential portions may require a different size of input in order to exercise them. By defining, ω in this manner, we are not required to use the maximum of the sizes. Thus we define $\omega(!^T f{:}x, \mu(!f:x))$ to be

$\omega(f;x_1;\mu_k)$ where k is such that $\mu_k = \mu_{k+1}$.

Time Seq $\quad seq^T(f)$

The assumption in this case is that $x = \langle x_1, \ldots, x_n \rangle$, the x_i's for $1 \le i < n$ are structurally equivalent, and x_n is structurally equivalent to each y_i where $\langle y_i, z_i, \ldots, z_{n-1} \rangle = seq(f){:}\langle x_i, \ldots, x_n \rangle$ for $1 \le i < n$. In addition, the z_i's are assumed to be structurally equivalent. The structure generated in this case is that of $\sigma(f;\langle x_i, y_{i+1} \rangle$. The construction is similar to that of the sequential version of the **Right Insert** except that the R-node connected to the outputs duplicates only the outputs corresponding to y_i and connects only these to the register. See Figure 3.13. To compute $\omega((seq(f);x, \mu((seq(f):x))$ we must determine the maximal markings of each application of f, $(\sigma(f;\langle x_i, y_{i+1} \rangle, \mu(\langle y_i, z_i \rangle))$. The marking of the outputs of $\sigma(seq(f);x)$ is $\mu(\langle y_1, z_1, \ldots, z_{n-1} \rangle) = \mu((seq(f);x)$ and $\mu(y_{i+1})$ for $1 \le i < n$ is the marking of the inputs corresponding to y_{i+1} in the maximal marking $M_{\mu(\langle y_i, z_i \rangle)}$. Let $In(M)$ denote the markings of the inputs for a marking M. The desired marking would then be

$$\prod_{i=1}^{n-1} M_{\mu(\langle y_i, z_i \rangle)} = M_{\prod_{i=1}^{n} \mu(\langle y_i, z_i \rangle)}.$$

As in the case of the **Right Insert** we compute this marking more efficiently by taking the product of each $In(M_\mu)$ with μ and repeating $n-1$ times or until the markings converge. We let $\mu_1 = \mu(y_1)$ and define $\mu_{i+1} = In(M_{\langle \mu_i, \mu(z_i) \rangle}) \cdot \mu_i$. We need to show that

$$M_{\langle \mu_{n-1}, \prod_{i=1}^{n-1} z_{n-1} \rangle} = M_{\prod_{i=1}^{n-1} \langle \mu(y_i), \mu(z_i) \rangle}.$$

This is accomplished by showing that $\mu_k = \prod_{i=1}^{k} \mu(y_i)$. This is true by definition for

$k=1$. If we assume it is true for k then we can show it to be true for $k+1$ as follows.

Since $\mu_k = \prod_{i=1}^{k} \mu(y_i)$,

$$\mu_{k+1} = In(M_{<\mu_k, \mu(z_k)>}) \cdot \mu_k$$

$$= In(M_{<\prod_{i=1}^{k}\mu(y_i), \mu(z_k)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= In(M_{\prod_{i=1}^{k}<\mu(y_i), \mu(z_k)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= In(\prod_{i=1}^{k} M_{<\mu(y_i), \mu(z_k)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= \prod_{i=1}^{k} In(M_{<\mu(y_i), \mu(z_k)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= In(M_{<\mu(y_k), \mu(z_k)>}) \cdot \prod_{i=1}^{k-1} In(M_{<\mu(y_i), \mu(z_k)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= \mu(y_{k+1}) \cdot \prod_{i=1}^{k-1} In(M_{<\mu(y_i), \mu(z_k)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$= \prod_{i=1}^{k-1} In(M_{<\mu(y_i), \mu(z_k)>}) \cdot \prod_{i=1}^{k+1}\mu(y_i),$$

$$\leq \prod_{i=1}^{k+1}\mu(y_i),$$

To show the other inequality we use the fact that if μ and μ' are markings of the outputs of a weak planar circuit such that $\mu \leq \mu'$, then $M_{\mu} \leq M_{\mu'}$ and so $In(M_{\mu}) \leq In(M_{\mu'})$. Thus since $\mu(z_k) \geq \prod_{i=1}^{k} \mu(z_i)$ we have,

$$\mu_{k+1} = In(M_{<\prod_{i=1}^{k}\mu(y_i),\mu(z_k)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$\geq In(M_{<\prod_{i=1}^{k}\mu(y_i),\prod_{i=1}^{k}\mu(z_i)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$\geq In(M_{\prod_{i=1}^{k}<\mu(y_i),\mu(z_i)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$\geq In(\prod_{i=1}^{k}M_{<\mu(y_i),\mu(z_i)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$\geq \prod_{i=1}^{k}In(M_{<\mu(y_i),\mu(z_i)>}) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$\geq \prod_{i=2}^{k+1}\mu(y_i) \cdot \prod_{i=1}^{k}\mu(y_i),$$

$$\geq \prod_{i=1}^{k+1}\mu(y_i).$$

This gives the desired result, $\mu_{k+1} = \prod_{i=1}^{k}\mu(y_i)$. Unlike the case of the **Right Insert**, $\mu_k = \mu_{k+1}$ for $k<n$ does not imply that $\mu_k = \mu_{k+i}$ for $i>0$. This is because there may be new unmarked atom's in the z_i's for $i>k$ which have not been used yet. We extend the definition of μ_k for $k \geq n$ by defining $\mu_z = \prod_{i=1}^{n-1}\mu(z_i)$, and $\mu_{k+1} = In(M_{<\mu_k,\mu_z>}) \cdot \mu_k$ for $k \geq n-1$. If $\mu_m = \mu_{m+1}$ for $m \geq n$ then clearly $\mu_m = \mu_{m+i}$ for $i \geq 0$. However we can hope to obtain a convergence before n by using μ_z in the definition of μ_k instead of $\mu(z_k)$. As before we let $\bar{\mu}_1 = \mu_1$ and we define $\bar{\mu}_{i+1} = In(M_{<\bar{\mu}_i,\mu_z>})$. Then we can show that $\bar{\mu}_i \leq \mu_i$ and if $\mu_m = \mu_{m+1}$ for $m>n$ then $\bar{\mu}_i \geq \mu_m$ for all i. Clearly $\bar{\mu}_k \leq \mu_k$ for k=1. We can show that it holds for k+1 by

assuming it is true for k.

$$\bar{\mu}_{k+1} = In(M_{<\bar{\mu}_k, \mu_s>}) \cdot \bar{\mu}_k$$

$$\leq In(M_{<\mu_k, \mu_s>}) \cdot \mu_k$$

$$\leq In(M_{<\mu_k, \mu(z_k)>}) \cdot \mu_k$$

$$= \mu_{k+1}$$

The proof of $\bar{\mu}_k \geq \mu_m$ is also realized by induction. Clearly it is true for $k=1$ and if we assume it is true for k,

$$\bar{\mu}_{k+1} = In(M_{<\bar{\mu}_k, \mu_s>}) \cdot \bar{\mu}_k$$

$$\geq In(M_{<\mu_m, \mu_s>}) \cdot \mu_m$$

$$= \mu_{m+1} = \mu_m$$

If $\bar{\mu}_k = \bar{\mu}_{k+1}$ then clearly $\bar{\mu}_k = \bar{\mu}_{k+i}$ for any $i > 0$. Then we have

$$\bar{\mu}_m \leq \mu_m \leq \bar{\mu}_k \equiv \bar{\mu}_m$$

which forces

$$\mu_m = \bar{\mu}_k.$$

We define $\omega(seq(f):x, \mu(seq(f):x))$ to be the planar circuit obtained from the marking $M_{\bar{\mu}_k}$.

This completes the mapping from FP to planar circuits. Examples will be given in Chapter 5.

3.7 Implementation of ω

In the implementation of ω, we retain the structural information provided by the FP combining forms. We instantiate the planar circuits corresponding to the FP primitives and use the combining forms to put together these planar circuits. The planar circuit of an FP expression is specified recursively in terms of the combining forms. We preserve this representation rather than flattening it to a planar circuit to retain the structural information provided by the combining forms. This information will be exploited in mapping the planar circuit to a layout and improving its wiring complexity.

Unfortunately the computation of σ proceeds in the direction of input to output since the FP objects generated by sub-functions of a **Compose, Right Insert** or **Seq** must be determined in order to evaluate the next sub-function, while the computation of markings is performed in the reverse direction. This requires us to retain the intermediate objects in order to perform the marking. We implement ω by first determining the input and output objects of the sub-functions and then their markings. This information is stored as a tree which corresponds to the decomposition of the FP expression in terms of its combining forms.

The decomposition of ω and σ in terms of the combining forms suggests a tree structure in which we identify each combining form with a node whose children are the sub-functions of the combining form. We construct a tree which corresponds to the computation graph of the FP expression. We define $\tau(f:x, \mu(f:x))$ to be a tree in which we store in the root, x the input, $f:x$ the output, $\mu(f:x)$ the marking of the output and $\mu(x)$, the marking of the input implied by $\mu(f:x)$. Figure 3.15 depicts a typical node. The sub-trees and the label attached to the root are determined as

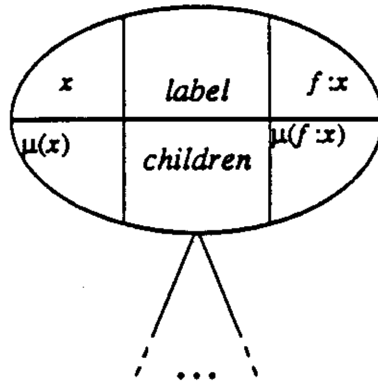


Figure 3.15 A node in the computation tree.

follows.

If f is an FP primitive then $\tau(f ;x, \mu(f :x))$ consists of a single node with no sub-trees which is labeled f.

If $f = f_n @ \cdots @ f_1$ then $\tau(f ;x, \mu(f :x))$ consists of a node labeled '@' with the trees $\tau(f_i ;y_{i-1}, \mu(y_i))$ for $1 \le i \le n$, where $y_i = f_i @ \cdots @ f_1 :x$, in left to right order as its children.

If $f = [f_1, \ldots, f_n]$ then $\tau(f ;x, \mu(f :x))$ consists of a node labeled '[,]' with the trees $\tau(f_i ;x, \mu(f_i :x))$ for $1 \le i \le n$, as its children in left to right order.

If $f = \%Object$ then we treat f as a computational primitive. $\tau(f ;x, \mu(f :x))$ consists of a single node with no sub-trees which is labeled *%Object*.

$\tau(\&f ;x, \mu(\&f :x))$ for $x = \langle x_1, \ldots, x_n \rangle$, consists of a node labeled '&' with the trees $\tau(f ;x_i, \mu(f :x_i))$ for $1 \le i \le n$, as its children in left to right order.

$\tau(!f ;x, \mu(!f :x))$ for $x = \langle x_1, \ldots, x_n \rangle$ and $n > 2$, consists of a node labeled '!' with the trees $\tau(f ;\langle x_i, y_i \rangle, \mu(f :\langle x_i, y_i \rangle))$ as children in left to right order where

$y_i = !f:<x_{i+1}, \ldots, x_n>$ for $1 \le i < n$. If $n=2$ then $\tau(!f;x, \mu(!f:x)) = \tau(f;x, \mu(f:x))$

$\tau(seq(f);x, \mu(seq(f):x))$ for $x = <x_1, \ldots, x_n>$ for $n > 2$ consists of a node labeled 'seq' with the trees $\tau(f;<x_i,y_i>, \mu(f:<x_i,y_i>))$ as children in left to right order where y_i is the first object in the sequence $seq(f):<x_{i+1}, \ldots, x_n>$ for $1 \le i < n$. If $n=2$ then $\tau(seq(f);x, \mu(seq(f):x)) = \tau(f;x, \mu(f:x))$

ω is a recursive procedure implemented on the tree. The B and R-nodes of the planar circuit are the leaves of the computation graph corresponding to computational and routing primitives respectively. In addition an R-node is generated for each **Construct**. Connections between these nodes are specified by the combining forms. An FP expression specifies a particular planar circuit. To obtain a homeomorphic maximal-indivisible planar circuit we need to merge adjacent R-nodes and separate them into indivisible R-nodes. Some of these operations can in fact be performed directly on the computation tree of an FP expression using identities to transform the FP expression, grouping together routing primitives. These will be discussed in the next section. As an example the computation tree of the example given in Section 3.1 is provided in Figure 3.16. The nodes are labeled with only their respective combining form or primitive function.

The specification can consist of a single FP function consisting only of the FP primitives or several functions defined in terms of one another. In this manner, the FP specification can impose a hierarchy on the design corresponding to the functions of the specification. We may wish to preserve the boundaries between these functions. In the implementation of τ when a new function, $f;x$, is encountered, we have two choices. We either leave a node labeled f and then a special link to the tree defined by $f;x$ or we use $\tau(f;x)$ directly. The former case prevents us from

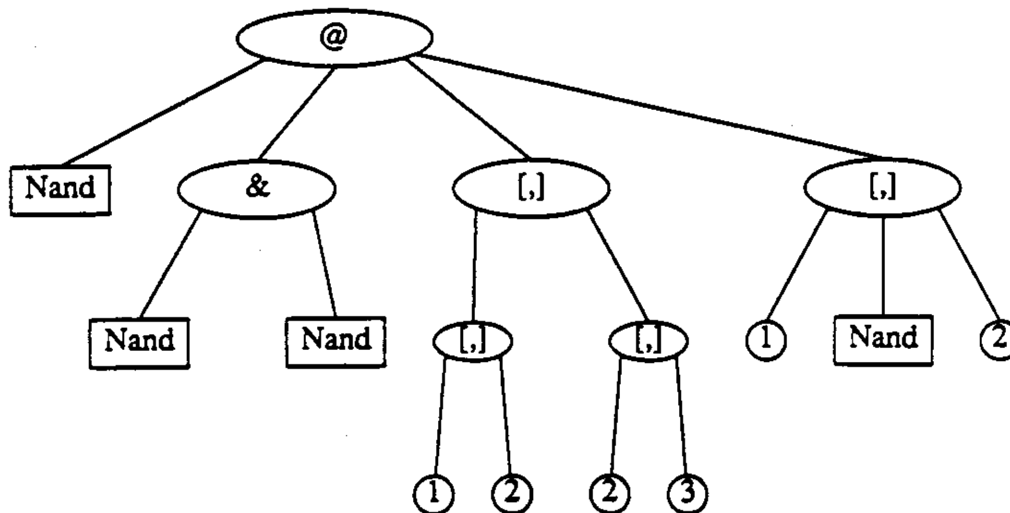


Figure 3.16 The computation tree of **Nand @ &Nand @ [[1,2],[2,3]] @ [1,Nand,2]**

performing the rearrangements described in the next section across the boundaries of this function, however since it preserves the boundary of this function we can choose to represent this function as a component, a *B*-node. The choice of which method is appropriate for each function defined in the specification depends on the intent of the designer. The ability to indicate which method to employ for each of the FP functions defined in the specification is provided. The default is the latter method in which the boundary of the defined function is not retained.

3.8 Planar Topology of FP Expressions

In the previous sections we have mapped FP expressions to planar circuits. In Chapter 4, we will show how to obtain a layout from these planar circuits. If we wish to consider an FP expression as specifying the planar topology of a layout rather than a particular planar circuit, we must consider the class of homeomorphic planar circuits to which the planar circuit belongs. The layout procedure may be

sensitive to the particular planar circuit chosen from a class of homeomorphic planar circuits. As discussed in Section 2.8, if the layout procedure satisfies certain properties, then it is sufficient to consider a maximal-indivisible planar circuit. In this section we show how the Computation Tree of an FP expression can be rearranged to achieve much of the merging of adjacent R-nodes. In general, we will not be able to rearrange it so that only maximal R-nodes are generated, but enough so that the layout procedure can easily consider maximal-indivisible R-nodes.

Each R and B-node of a planar circuit corresponds to an FP primitive with the exception of the R-node, u_D generated for each instance of the combining form, **Construct**. Connections, wires, between these nodes are realized by the **Compose**, **Construct**, **Right Insert** and **Seq** combining forms. We will rearrange the computation tree of an FP expression so that the sub-trees of each node will either consist of only R-nodes or will correspond to a planar circuit in which none of the I or O pins are connected to R-nodes: the inputs and outputs are connected to components or run through. If there are no B-nodes within a tree, then the entire tree can be implemented as one R-node. We achieve the merging of adjacent R-nodes by grouping together adjacent R-nodes within a sub-tree and then implementing the sub-tree as a single R-node. Clean divides are considered during the layout of these R-nodes. The rearrangements are based on FP identities which will be given. We change the definition of ω so that it does not generate an R-node for Id; it simply generates wires connecting inputs to outputs.

We first extract the R-node u_D, generated for each *Construct* combining form. This is achieved by introducing a new combining form called **Projection**. It is denoted by $\{h_1, \ldots, h_n\}$ and is equivalent to $[h_1@1, \ldots, h_n@n]$. It differs from the

Construct combining form in that instead of applying each h_i to the input object, h_i is applied only to x_i where x is assumed to be of the form $<x_1, \ldots, x_n>$. If x is not of this form then $\{h_1, \ldots, h_n\}:x$ is undefined ($\perp$). We then have the following identity for FP functions,

$$[h_1@g_1, \ldots, h_n@g_n] \equiv \{h_1, \ldots, h_n\}@[g_1, \ldots, g_n] \equiv <h_1(g_1(x)), \ldots, h_n(g_n(x))>.$$

σ and ω of $\{h_1, \ldots, h_n\}$ are constructed from σ and ω of the FP expressions $h_i;x_i$ in the same manner as σ and ω of $\&h$ are constructed from the $h;x_i$'s. $\{h_1, \ldots, h_n\}$ is represented in the computation tree as a node labeled '{,}' with the sub-trees corresponding to $h_i;x_i$'s in left to right order as its children.

We accomplish the merging of R-nodes by rearranging the tree of an FP expression to collect adjacent routing nodes within a sub-tree. If a sub-tree does not contain any B-nodes then we can enclose it by itself within a simply connected region of the plane with only its input and output wires emerging from this region. Since there are no B-nodes within this region we can consider this region to form an R-node. This has the same effect as if we had combined all of the sub-tree's R-nodes using merges and glues.

If $[g_1, \ldots, g_n]$ does not contain any B-nodes then we can implement it as one R-node as described. By replacing f by $f@Id$, we can write $[f_1, \ldots, f_n]$ as $\{f_1, \ldots, f_n\}@[Id, \ldots, Id]$. The sub-tree corresponding to $[Id, \ldots, Id]$ can then be implemented as a single R-node which is the same as the R-node u_D generated for $[f_1, \ldots, f_n]$.

Once we isolate the routing node u_D generated for the **Construct** by rewriting the form as above, then adjacent R-nodes in the planar circuit are a result

of only the **Compose**, **Right Insert** and **Seq** combining forms. The operations which are performed on the tree correspond to the following FP identities.

$$f_1@(f_2@f_3) = (f_1@f_2)@f_3 \tag{3.15}$$

The associativity of the **Compose** combining form allows us to group routing functions within one sub-tree. If we have the function, $f_1@f_2@f_3@f_4$ where f_2, f_3 are routing functions and f_1, f_4 are not, then we will rewrite this function as $f_1@(f_2@f_3)@f_4$. This will generate the tree on the right in Figure 3.17 instead of the tree on the left. We can then consider the merging the R-nodes corresponding to f_2 and f_3 by implementing the sub-tree containing both of them as one R-node.

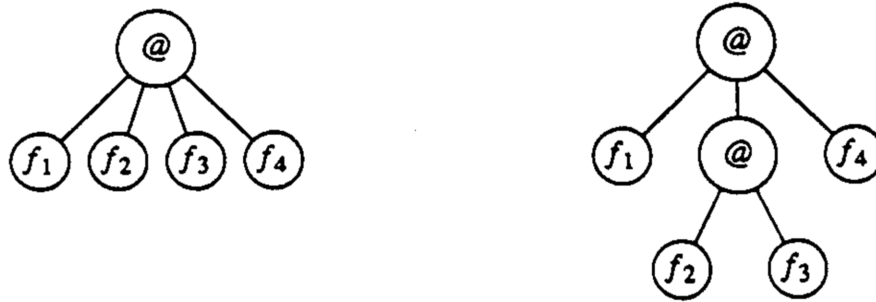


Figure 3.17 Rearranging the tree to group R-nodes together.

$$[h_1@g_1, \ldots, h_n@g_n] \equiv \{h_1, \ldots, h_n\}@[g_1, \ldots, g_n]. \tag{3.16}$$

As mentioned previously this identity will permit us to separate out the R-node u_D of a **Construct**. In practice the g_i's will consist only of routing functions.

$$\{h_1@g_1, \ldots, h_n@g_n\} \equiv \{h_1, \ldots, h_n\}@\{g_1, \ldots, g_n\}. \tag{3.17}$$

Identity 3.17 allows us to pull routing functions out of the end of a '{,}'. Each of these identities rearranges the computation tree of an FP expression. Note that these rearrangements have no effect on the planar circuit generated by ω since the connectivity and the embedding of the planar circuit is preserved. However if we

modify ω so that it generates an R-node for each tree which does not contain any B-nodes, then these rearrangements have effects equivalent to applying a sequence of operations to the planar circuit derived from the original tree. Hence these operations do not alter the planar topology but do bring us closer to a maximal-indivisible representative of this planar topology.

Using the associativity of the **Compose** we can flatten '@' nodes.

Definition 3.18

> A node of a computation tree is *flattened* if it is not a '@' or does not have any children which are '@'s. A computation tree is *flattened* if all of its nodes are.

We can flatten an '@' node of a computation tree by first flattening its children. If it has an '@' node as a child, then we simply replace this child by its list of children in the list of children of the parent '@' node. This operation is depicted in Figure 3.18.

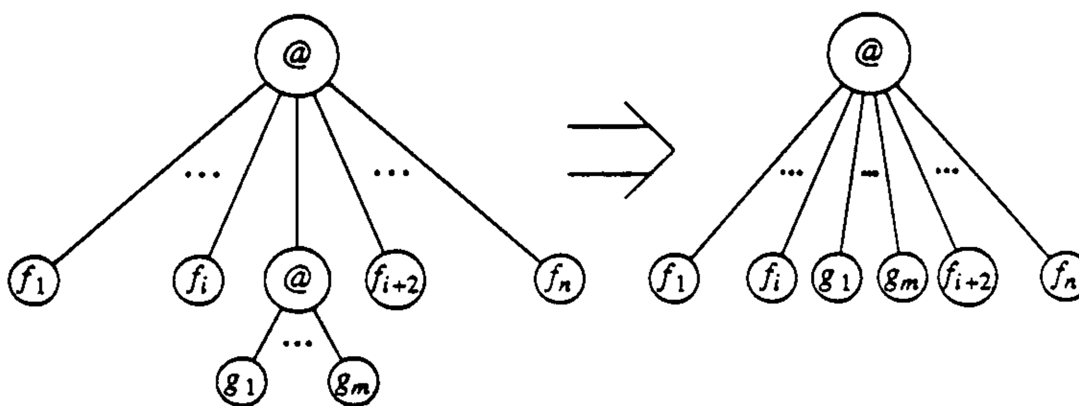


Figure 3.18 Flattening an '@' node.

Applying this procedure to the '@' nodes from the leaves up will flatten a computation tree.

Definition 3.19

A computation tree is a *routing tree* if all of its leaves are R-nodes.

We first restrict ourselves to case in which there are no **Right Inserts** and **Seq** combining forms and incorporate them afterwards. In this case we define a *normal form* for the computation tree of an FP expression as follows. In the notation we associate a computation tree with its FP expression.

Definition 3.20

1. Every computational FP primitive is in *normal form.*

2. $f_1@\cdots@f_n;x$ is in *normal form* if each f_i is either a flattened routing tree or it is in normal form and not an '@' node. In addition, neither f_1 nor f_n is a routing tree and for any $1 \le i < n$ either f_i or f_{i+1} is not a routing tree.

3. $\{f_1, \ldots, f_n\};<x_1, \ldots, x_n>$ is in normal form if each $f_i;x_i$ is Id or is in normal form.

4. $\&f;<x_1, \ldots, x_n>$ is in normal form if each $f;x_i$ is Id or is in normal form.

A computation tree without **Right Inserts** and **Seqs** can be put into normal form by the following recursive procedure. We define a procedure, Π which by applying the identities transforms a computation tree into a computation tree corresponding to $b^R@f^N@f^R$ where b^R and f^R are flattened routing trees and f^N is either Id or is in normal form. If f_N is Id then we require that f^R be Id as well. We will define Π in terms of the leaves of the computation tree and then the combining forms '@', '[,]', '{,}' and '&.' Π does not alter the leaves of the computation tree since each node is

either a routing primitive which is a flattened routing tree or a computational primitive which is already in normal form by definition. We simply set the other functions to *Id*. It remains only to define Π for each of the combining forms.

$f_1 @ \cdots @ f_n$

We apply Π to each of the sub-trees, f_i, to obtain $b^R{}_i @ f^N{}_i @ f^R{}_i$. We create a new list of children for this '@' node by replacing each f_i by the list of children of $\Pi(f_i)$. We remove any *Id*'s from this list. If any of the $f^N{}_i$'s are '@' nodes we replace them by their list of children. We now have a list of children for the '@' node which meets the requirement that each of its children is either a flattened routing tree or is a tree in normal form which is not an '@' node. We still might have adjacent routing trees. To fix this we take each consecutive sequence routing trees, $f_j, \ldots, f_{j+k}$ and replace these children by the flattening of $f_j @ \cdots @ f_{j+k}$. We now have a list of children $f'_1, \ldots, f'_n$ in which there are no adjacent routing trees. If $n=1$ then f'_1 is either a flattened routing tree or it is in normal form and not an '@' node. We add *Id*'s in order to provide the form required by Π. If $n>1$ there is at least one child which is not a routing tree. We let $b^R = f'_1$ if f'_1 is a routing tree and $b^R = Id$ otherwise. We let $f^R = f'_n$ if f'_n is a routing tree and $f^R = Id$ otherwise. Note that this covers the case $n=2$. If $n=2$ then we let f^N be f'_1 or f'_2, which ever is not the routing tree. Finally if $n>2$, we let $f^N = f'_i @, \ldots, @ f'_j$ where f'_i is the first child which is not a routing tree and f'_j is the last child which is not a routing tree. In any case we have

$$b^R @ f^N @ f^R = f'_1 @ \cdots @ f'_n$$

$\{f_1, \ldots, f_n\}$

We apply Π to each f_i to obtain $b^R{}_i @ f^N{}_i @ f^R{}_i$. Using the identity 3.17 we can

replace this tree by

$$\{b^R{}_1, \ldots, b^R{}_n\} @ \{f^N{}_1, \ldots, f^N{}_n\} @ \{f^R{}_1, \ldots, f^R{}_n\}$$

Since each $b^R{}_i$ is a flattened routing tree or Id, $b^R = \{b^R{}_1, \ldots, b^R{}_n\}$ is a flattened routing tree. Similarly for $f^R = \{f^R{}_1, \ldots, f^R{}_n\}$. Since each $f^N{}_i$ is in normal form or is Id, $f^N = \{h_1, \ldots, h_n\}$ is in normal form. Then $b^R @ f^N @ f^R$ is the required function.

$[f_1, \ldots, f_n]$

Again we apply Π to each f_i to obtain $b^R{}_i @ f^N{}_i @ f^R{}_i$. Using the identities 3.16 and 3.17 we can replace this node by

$$\{b^R{}_1, \ldots, b^R{}_n\} @ \{f^N{}_1, \ldots, f^N{}_n\} @ [f^R{}_1, \ldots, f^R{}_n].$$

Since each $b^R{}_i$ is a flattened routing tree or Id, $b^R = \{b^R{}_1, \ldots, b^R{}_n\}$ is a flattened routing tree. Similarly for $f^R = \{f^R{}_1, \ldots, f^R{}_n\}$. Since each $f^N{}_i$ is in normal form or is Id, $f^N = \{h_1, \ldots, h_n\}$ is in normal form. Then $b^R @ f^N @ f^R$ is the required function.

$\&f ; <x_1, \ldots, x_n>$

We deal with & by replacing it using the following identity.

$$\&f : <x_1, \ldots, x_n> = \{f_1, \ldots, f_n\} : <x_1, \ldots, x_n>$$

in which $f_i = f$ for $1 \le i \le n$. We can then deal with it as we would with '{,}'.

Hence we can apply Π to any computation tree without **Right Insert** and **Seq** to transform it into normal form. We perform one last operation on the tree in order to maximize the parallelism. If there is an '@' node which has two adjacent children which are '{,}'s, we combine them using

$$\{h_1, \ldots, h_n\}@\{g_1, \ldots, g_n\} \equiv \{h_1@g_1, \ldots, h_n@g_n\}.$$

Clearly if h_i is in normal form and g_i is in normal form then $h_i@g_i$ can be put in normal form. Then if we change ω so that it generates one R-node for a routing tree, instead of evaluating the FP expression corresponding to the routing tree, two R-nodes will be adjacent if and only if they correspond to nodes f and f' which occur as $f@\{g_1, \ldots, g_n\}@f'$ where one of the g_i's is Id. Thus each maximal R-node corresponds either to some routing sub-tree in the computation tree or to a set of routing sub-trees under a '@' which are separated by nested '{,}'s, one of which has an Id as a sub-tree. Only the latter, can have self-loops and these must correspond to an additional Id within the nested '{,}'s.

Unfortunately this property does not extend readily to trees containing **Right Inserts** and **Seqs**. We could replace each **Right Insert** and **Seq** by appropriate structure consisting of '@'s and '{,}'s to obtain a normal form for the tree which then corresponds to an equivalent Maximal-Indivisible planar circuit. Unfortunately, these transformations would lose the structural information provided by the presence of these forms. If they have only routing nodes in their sub-trees then they can be dealt with as one R-node. Only the case in which they have B-nodes in their sub-trees needs to be considered. We define a less strict form for '@' nodes in which the first function applied is permitted to be a routing tree.

Definition 3.21

$f\,;x$ is in *Pseudo-normal form* if it is in normal form or it is $f_1@ \cdots @f_n$ for $n \geq 2$ which satisfies all of the requirements of normal form except that f_n is a routing tree.

We relaxed the requirement of & that all of its sub-trees correspond to the same

function f by replacing it by '[,]'. We will also relax this requirement for the **Right Insert** and **Seq**. Note that ω and σ do not require that the sub-functions be the same.

Definition 3.22

A tree with '!' as its root is in normal form if each $f_i;<x_i,!f_{i+1}:<x_{i+1},\ldots,x_n>$ is in pseudo-normal form.

To place an '!' tree in normal form we start with the rightmost child, $f_{n-1};<x_{n-1},x_n>$. We apply Π to it to obtain $b^R@f^N@f^R$ or just b^R. We then move b^R to the sub-tree to the left by replacing f of the child just to the left by $f@[1,b^R@2]$ and leaving behind $f^N@f^R$ or Id. We thus proceed through the list of children from right to left until we arrive at the last child. We apply Π to it as well but in this case we move the b^R function outside the **Right Insert**. We obtain then $b^R@f$ where f is a routing tree with an ! as the root in in normal form.

The **Seq** is somewhat more complicated to deal with since it not only might need an R-node to merge inputs, but one to separate its outputs as well. We proceed in somewhat the same manner as for the **Right Insert** except that we leave behind a particular routing node whose purpose is to separate the outputs of each child into y_i and z_i, those that are fed to the child on the left and those that are outputs of the Seq respectively.

Definition 3.23

A *fork routing tree* is a routing tree whose output object is $<y_1,y_2>$ such that both $\rho(y_1)$ and $\rho(y_2)$ are subsequences of $\rho(x)$ where x is the input object.

Definition 3.24

A tree whose root is a Seq is in normal form if each $f_i;<x_i,y_i>$ is in normal form or its is in the form $f_M@f_1@\cdots@f_n@f_R$ where $f_1@\cdots@f_n@f_R$

where is in pseudo-normal form and f_M is a fork routing tree.

We place a 'seq' in normal form by traversing its list of children from right to left. We apply Π to a child $f;<x_i,y_i>$ to obtain $b^R{}_i@f^N{}_i@f^R{}_i$ or just $b^R{}_i$. In either case we replace $b^R{}_i$ by $[b^1{}_i@1,b^2{}_i@2]@b^F{}_i$ where $b^F{}_i$ is a fork routing tree. We then move $b^1{}_i$ to the child to the left if there is one and out of the 'seq' otherwise. $b^2{}_i$ is moved out of the 'seq'. We thus leave behind either $b^F{}_i$ or $b^F{}_i@f^N{}_i@f^R{}_i$. We obtain

$$\{b^1{}_1,b^2{}_1,b^2{}_2,\ldots,b^2{}_{n-1}\}@seq$$

where the 'seq' is in normal form. Note that its children may no longer correspond to the same function.

The sequential versions of **Apply-to-All, Right Insert** and **Seq** will result in planar circuits which are constructed from the planar circuit of the function they are applied to. We define the normal form of these combining forms in terms of the normal form of this function. Unlike the combinational versions of these forms, the planar circuits obtained by applying σ to their sub-functions must be the same. Hence the normal forms of their sub-trees will also be the same. For the sequential version of **Apply-to-All** we require the sub-trees to be in normal form.

Definition 3.25

$\&^T f;x$ is in *normal form* if each $f;x_i$ is in normal form.

Since the normal forms of the children of an '$\&^T$' will be the same we only need to apply Π to one sub-tree. We apply Π to $f;x_1$ to obtain $b^R@f^N@f^R$. We then replace $\&^T f$ by $\&^T b^R@\&^T f^N@\&^T f^R$. We remove any $\&^T Id$'s.

For the sequential versions of **Right Insert** and **Seq**, we apply Π only to the leftmost child. We do not change the other sub-trees since ω will only use the leftmost sub-tree in either case.

The R-nodes in the planar circuit generated from a tree in normal form are not necessarily maximal. The layout procedure might need to decompose maximal R-nodes in order to implement them anyway. A maximal-indivisible planar circuit may in fact have only one R-node. Making R-nodes maximal would result in the loss of their initial decomposition. On the other hand implementing adjacent R-nodes instead of merging them may be inefficient. This can occur if the order of the wires connecting the two R-nodes results in unnecessary crossings or if branchings can be delayed resulting in fewer wires. However we can detect these inefficiencies and alter the R-nodes involved. This will be discussed when we consider the optimization of the planar topology. Hence we compromise by performing the merges described in this section which do not destroy the structure implied by the combining forms.

Summary

In this chapter, we have defined the mapping from FP expressions to planar circuits. In order to define this mapping we introduced weak planar circuits which are planar circuits with some of the connectivity conditions relaxed. In order to obtain a planar circuit we 'pruned' these weak planar circuits. We then defined the mapping from FP expressions to planar circuits by first obtaining a weak planar circuit and then pruning it to obtain a planar circuit. We exploited the fact that the weak planar circuits defined by FP expressions are *directable* to efficiently compute the planar circuit, avoiding the generation of structures which will be pruned. We

extended this mapping to sequential circuits. The implementation of this mapping generates a computation tree which preserves the hierarchical representation of the FP expression in terms of its combining form. The computation tree was rearranged by transformations corresponding to FP identities in an attempt to obtain a planar circuit with maximal R-nodes.

CHAPTER 4

Mapping Planar Topology to Layouts

To obtain a layout or a visual representation of a planar circuit, it must be mapped to geometry; coordinates must be assigned to its elements. Although the space of layouts to be considered in this mapping has been greatly reduced by restricting the layouts to only those which the planar circuit represents, the problem is still not an easy one. In fact, we will show that minimizing the area is NP-hard in Section 4.2. However, the planar circuits which we are mapping to layouts are not arbitrary, they originate from FP expressions and their computation trees provide them with a hierarchical representation in terms of combining forms. We will exploit this structure in order to obtain a layout efficiently. Hence the quality of the layout will depend on how well this hierarchical representation captures the layout. We will obtain 'abstract layouts' for the planar circuits and then transform these into actual layouts. In Section 4.1 we discuss the details of layout and define an 'abstract layout' to be a layout of a circuit on a grid with wires of zero width, and in which all coordinates and dimensions are in grid units. This format will permit graphical feedback of the geometric consequences of decisions early in the synthesis of the design. In Section 4.3, we will first map the 'boxes and wires' of the planar circuit to horizontal cross-sections. In Section 4.4 these cross-sections are stacked vertically and horizontal compaction is performed to resolve constraints between the boundaries of these cross-sections to obtain the 'abstract layout.' Transforming these 'abstract layouts' into actual layouts is discussed in Section 4.5. We first discuss the

details of layout.

4.1 Layouts of Planar Circuits

Suppose we are given a planar circuit and wish to find a layout which is covered by the planar circuit. In Section 2.3 we defined a layout as follows. The layout of a circuit is an assignment of coordinates in the plane to the modules and an implementation of the nets as trees such that the following hold.

1. Modules do not touch or overlap.
2. I/O pins do not touch nor overlap with each other and the modules.
3. Each net coincides with the modules and the I/O pins in exactly the set of points which correspond to the pins in its net-list.
4. Each connected set of points onto which more than one net has been mapped is contractible to a point in the space obtained by removing the interiors of the modules from the plane.

This description is an abstract one designed to capture all layouts at a topological level. In practice, there are limitations on the geometry of layouts. Layouts are constructed in terms of *lambda* units which represent the smallest unit of resolution the fabrication process can reasonably assure. They primarily consist only of rectilinear objects since this greatly reduces the complexity of the design tools needed to handle and operate on the layout without significantly impacting the design. We have placed no limit on the number of nets which can lie on top of each other. However there are actually only a small number of layers on which to

implement the circuit and there are restrictions on which layers can overlap. To complicate matters even further, the layers may have different costs associated with them due to different electrical properties and different resolutions in fabrication. Contacts are used to interconnect layers and these also have a cost associated with them since a contact between two layers precludes the use of any of the layers in between in the vicinity of the contact. All of these factors are dependent on the fabrication technology. A planar circuit is independent of such considerations, while a layout is not. In mapping a planar circuit to a layout the rules and costs of the fabrication technology must be considered.

Many decisions about the layout can not be readily made without geometric information. Hence we provide both an abstract representation of the layout and the means to realize this abstract representation as a layout. This permits the geometry available at the current stage in the design process to be represented without making all of the decisions about the layout. The abstract representation which will be referred to as the *abstract layout* or *sketch*, is also independent of technology. In the abstract representation, we impose a grid on the plane and require all coordinates of the specification of a layout to fall on the grid points. All dimensions are in terms of grid units. The boundaries of components and the path segments of wires are required to fall on the grid lines. *IO*-pins occupy one grid point each. One coordinate is sufficient to specify the position of *IO*-pins, while two coordinates are necessary for the *B*-nodes since the orientation must also be specified.

To complete the layout we must implement the nets so that the resulting layout is covered by the planar circuit. In the abstract layout since we have no notion of layers we limit the crossovers by allowing at most two wires to intersect at a grid

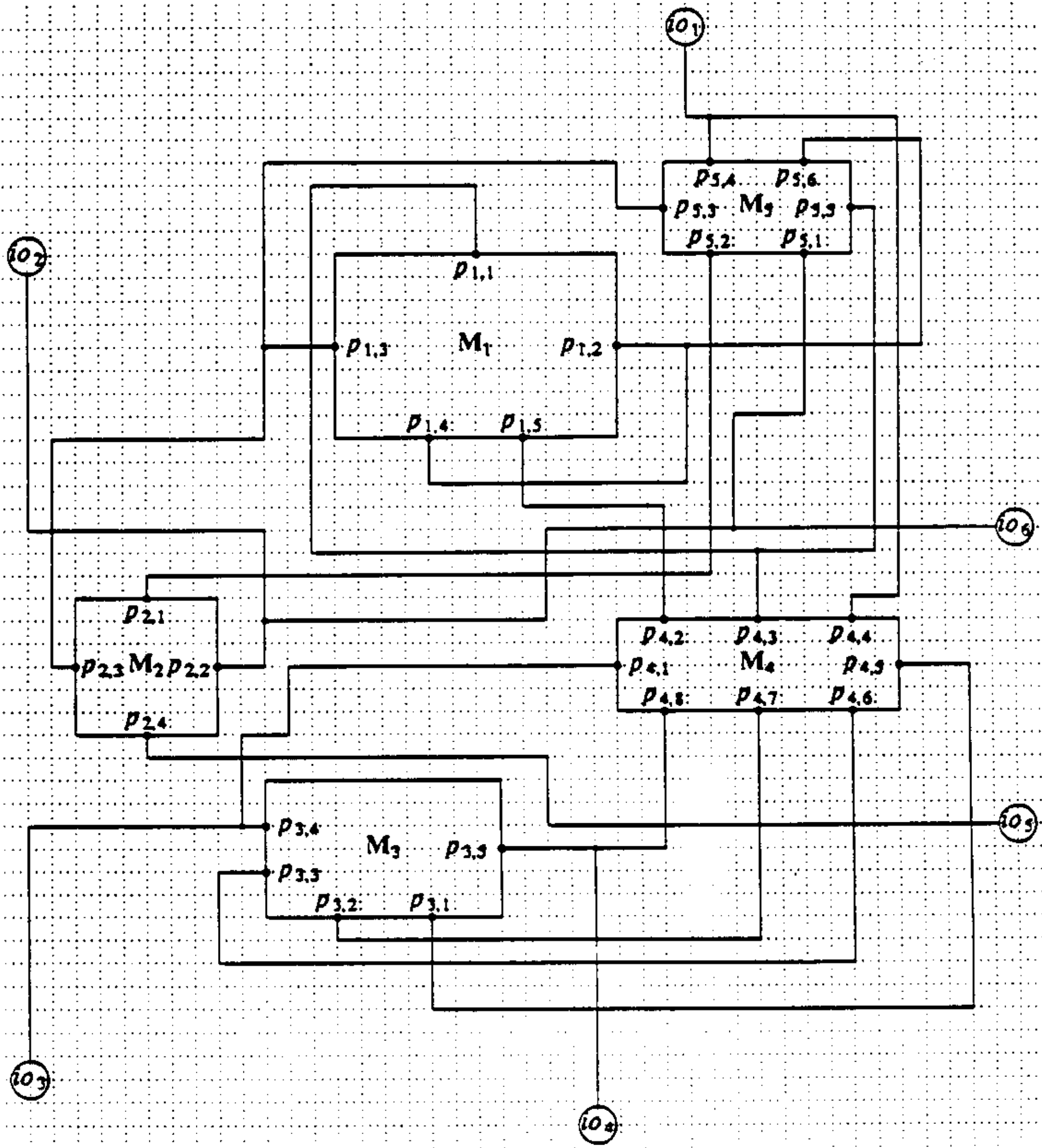


Figure 4.1 The mapping of a layout on the grid

point and requiring this intersection to be between perpendicular wires. To specify implementation of the each net, we specify the location of the branch points of the nets and paths along the grid lines to interconnect the branch points and the pins belonging to the net. In order to realize the sketch as a layout, we will need to

specify the layers on which the path segments are situated and any contacts that are necessary to connect path segments to other path segments and/or pins which are on other layers. The spacing between elements can then be adjusted. The size of the layout is defined to be the area of the smallest rectangle enclosing it. All pins, components and wires must be contained in the rectangle. *IO*-pins may sit on the boundary of this rectangle although this is not required.

Figure 4.1 shows a mapping of the layout of Figure 2.3 in Chapter 2. Notice that the modules are now all rectangles and the wires are either horizontal or vertical.

4.2 The Complexity of Mapping Planar Circuits to Layouts

In this section we show that attempting to find a layout of minimal size for a planar circuit, even at the abstract level and without *R*-nodes, is probably intractable (unless **P=NP**). The problem of finding an optimal layout for a planar circuit can be stated as:

> Given a planar circuit $A=(P,IO,B,R,W)$ and an integer K is there an abstract layout of size K or less which is covered by the planar circuit ?

We assume the description of the abstract layout is given by two coordinates for each *B*-nodes and the coordinates of the path segments and branch points which implement the nets. There is no bound on the size of the description of the abstract layout in this format since the paths can be arbitrarily complicated, venturing arbitrarily far from the components. However, any layout of size K or less will have a description which is bounded by $K \cdot |P|$, since K is the maximum number of grid

points which can be occupied and the maximum number of objects which can be mapped to the same coordinate is bounded by the maximum number of pins in an R-node or 2 if there are no R-nodes. To specify the coordinates of the IO and B-nodes and the branch points of the nets requires at most $O(logK \cdot |P|)$ space. However the size of the description can still be exponential with respect to the size of the input, since we use $O(logK)$ space to represent K, and the abstract layout could have a path which consists of $O(K)$ line segments.

Lemma 4.1 The problem of deciding whether there is a layout of size K of a planar circuit, $A=(P,IO,B,\varnothing,W)$, is NP-hard.

Proof: We reduce the NP-complete problem **3-Partition** [Gare79] to it.

3-Partition

Instance: (A,B,s) where A is a set of $3m$ elements , $B \in Z^+$, and s is a mapping, $s:A \rightarrow Z^+$ which associates a size with each element of A such that $B/4 < s(a) < B/2$ for any $a \in A$ and $\sum_{a \in A} s(a) = mB$.

Question: Is there a partition of A into m disjoint subsets, $A_1, A_2, \ldots, A_m$, such that for any $1 \leq i \leq m$, $\sum_{a \in A_i} s(a) = B$?

Construction: We will assume without loss of generality that m is odd. Otherwise we can add three more elements of size $B - 2 - 2 \cdot \lfloor B/4 \rfloor$, $\lfloor B/4 \rfloor + 1$ and $\lfloor B/4 \rfloor + 1$ to make m odd without affecting the existence of a solution either way. Given an instance of **3-Partition**, (A,B,s) we construct the following planar circuit. $A=(P,IO,B,\varnothing,W)$ where,

$$P = \{u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, i, o\} \cup \bigcup_{i=1}^{3m} \{t_i, d_i, p_i, q_i\},$$

$$IO = \{i, o\}$$

and

$$B = \{b_1, b_2, b_3, b_4\} \cup \{b_{a_i} \mid a_i \in A\}$$

where b_1, b_2, b_3 and b_4 are pictured in Figure 4.2. and b_a is pictured in Figure 4.3.

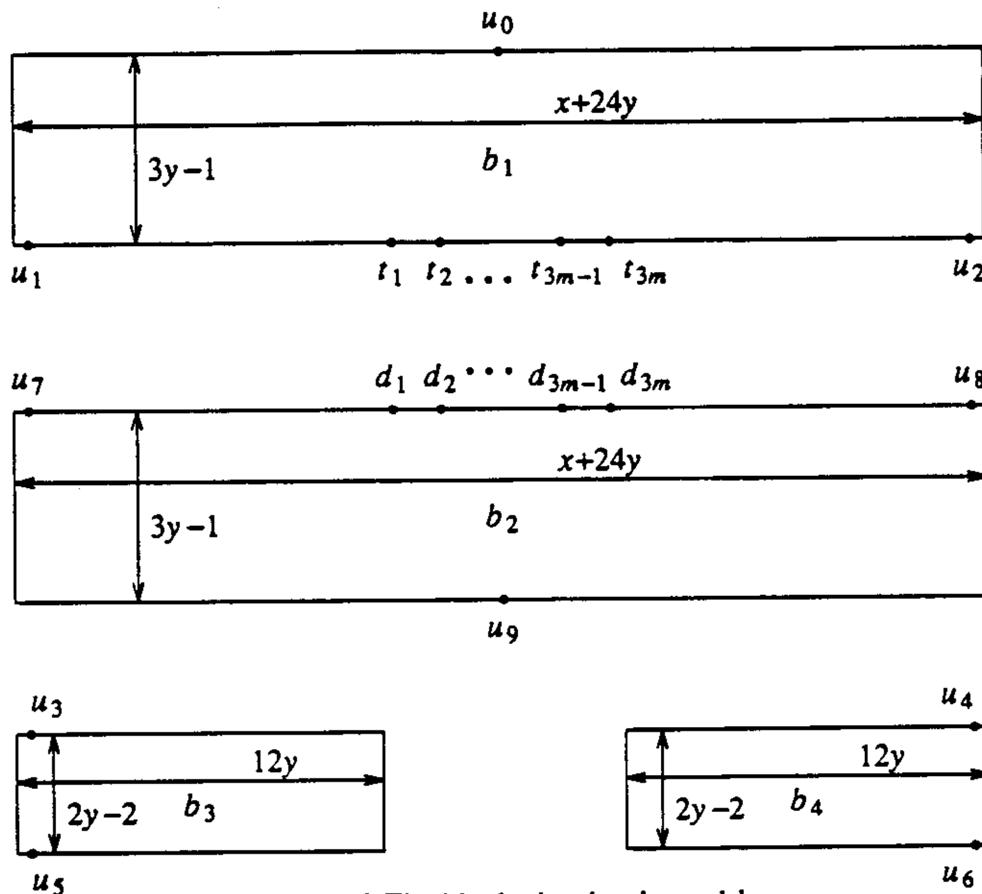


Figure 4.2 The blocks b_1, b_2, b_3 and b_4

W consists of four wires joining b_1, b_2, b_3 and b_4 in a ring, two wires connecting b_1 and b_2 to the two IO pins, and $6m$ wires joining each b_{a_i} to b_1 and b_2,

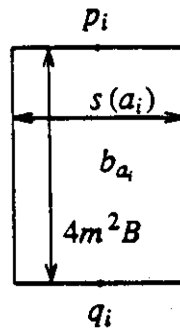


Figure 4.3 The block b_{a_i}

$$W = \{(u_1,u_3),(u_2,u_4),(u_5,u_7),(u_6,u_8)\} \cup \{(u_0,i),(o,u_9)\}$$

$$\cup \{(t_i,p_i) \mid 1 \le i \le 3m\} \cup \{(d_i,q_i) \mid 1 \le i \le 3m\}.$$

Figure 4.4 shows the embedding of this planar circuit.

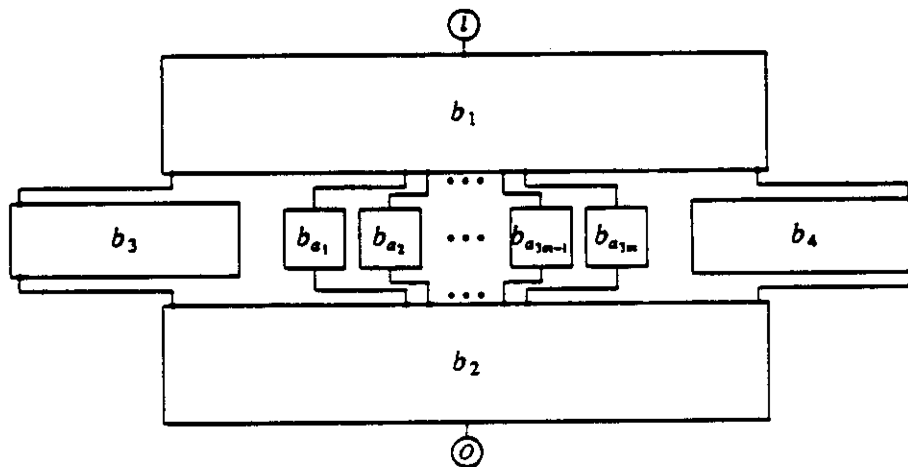


Figure 4.4

To finish the construction we must provide the exact dimensions of all of the B-nodes. The dimensions of b_1, b_2, b_3 and b_4 were given in terms of x and y, so it remains to specify x and y. We let

$$y = \frac{4m^3B+3m^2+4m+1}{2} \quad \text{and} \quad x = B+3m+1.$$

Remember that we assumed m to be odd. To complete the construction we must provide, K, the size of the layout desired,

$$K = (x+24y)(8y) = 8xy+192y^2.$$

In order to prove the lemma we must show that there is a layout of this planar circuit of size K or less if and only if there is a solution to the **3-Partition** problem.

We show first how a partition of A into $A_1, A_2, \ldots, A_m$, such that for any $1 \le i \le m$, $\sum_{a \in A_i} s(a) = B$ yields a layout of this planar circuit of size K or less. We arrange the blocks b_1, b_2, b_3, and b_4, as in Figure 4.5.

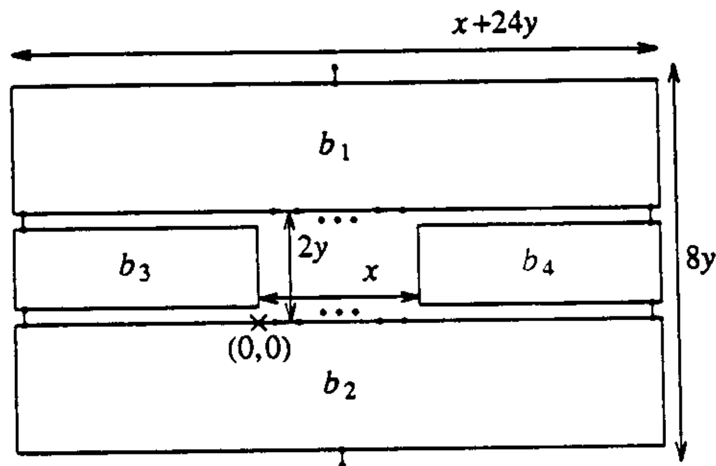


Figure 4.5

This leaves us an area of size x by $2y$ in which to place the remaining components, the b_{a_i}'s. Assume the origin is at the lower left corner of this area as shown by the X in Figure 4.5. The b_{a_i}'s will be place in rows of three corresponding to the partitions. In between the rows of blocks, $3m$ horizontal grid lines will be used to route the wires. We divide up this x by $2y$ box into horizontal strips alternating in height between $3m+1$ and $4m^2B$ as shown in Figure 4.6. The blocks corresponding to A_i will be placed in the ith strip of size $4m^2B$ counting from the bottom with the

orientation in Figure 4.3, that is, with pin p_i on the top edge. The x-coordinates of the b_{a_i}'s are determined as follows. Assume first that each A_j is ordered by increasing index, $A_j = \{a_u, a_v, a_w\}$ such that $u<v<w$. Then three blocks, b_{a_u}, b_{a_v} and b_{a_w} are placed in the jth row of size $4m^2B$ in left to right order leaving u, $v-u$, $w-v$ and $3m-w+1$ space in between. The coordinates of the upper left corners of the blocks are, $(u \, , \, j(4m^2B+3m+1))$, $(v+s(a_u) \, , \, j(4m^2B+3m+1))$ and $(w+s(a_u)+s(a_v) \, , \, j(4m^2B+3m+1))$ respectively.

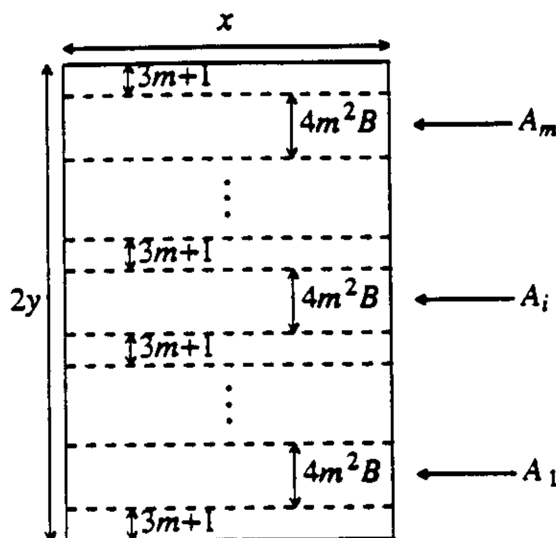


Figure 4.6

We have $v-1$ vertical tracks free to the left of b_{a_u}, $v-u-1$ vertical tracks between b_{a_u} and b_{a_v}, $w-v-1$ vertical tracks between b_{a_v} and b_{a_w}, and $3m-w$ tracks to the right of b_{a_w}. Notice that the number of vertical tracks in between the blocks accommodates exactly the number of blocks between 1 and $u-1$, $u+1$ and $v-1$, $v+1$ and $w-1$, and $w+1$ and $3m$, respectively. We route the wires from top to bottom starting with the leftmost (t_1, p_1) and proceeding with (q_1, d_1) and then routing (t_2, p_2). Each (t_i, p_i) must pass to the right of all blocks b_{a_j} for $j<i$ and to the left of

the remaining. If we always use the leftmost available vertical track in between rows of blocks this will be achieved since exactly the numbers of vertical tracks necessary have been left in between the blocks in each row. When we reach the row containing b_{a_i} we use a horizontal track to connect to p_i and then start the wire (q_i,d_i) from q_i. We use at most one horizontal track in between row of blocks to route this wire. If the wire must travel to the right we use the lowest available track and otherwise the highest. We proceed in this manner routing (t_i,p_i) and (q_i,d_i) for $1 \le i \le 3m$. We can never run out of horizontal tracks since there are $3m$ available in between each row of blocks and there are exactly $3m$ wires routed in between each row of blocks.

We now argue that a layout of size K or less provides a solution to the partition problem. We first show that the arrangement of the four main blocks in Figure 4.5 is the only one which can meet the size constraint. Since the size of b_1 is already $x+24y$ this is a lower bound on one of the dimensions of the layout. Assume without loss of generality that this is the horizontal dimension. We consider the arrangements of the other blocks with respect to b_1. If b_2 is not placed so that its longest dimension is also horizontal then we have an area of at least

$$(x+24y)(3y-1+1+x+24y) = (x+24y)(x+27y) > 8y(x+24y) = K.$$

Thus both b_1 and b_2 must have their longest dimension in the horizontal direction. If either b_3 or b_4 have their longest dimension in the vertical direction, then we have an area of

$$(x+24y)12y > 8y(x+24y) = K.$$

Thus all four blocks must be placed so that their longest dimensions are horizontal. We can thus refer to the horizontal and vertical dimensions of the layout and blocks, as their widths and heights respectively. We next consider the relative positions of

these four blocks. We say that two blocks overlap horizontally (vertically) if their projections onto the x-axis (y-axis) overlap. If two blocks overlap horizontally, then the height of the layout is bounded from below by the sum of the heights of the blocks plus 1. If they don't overlap horizontally, then the width of the layout is bounded by the sum of the widths of the blocks plus one. We first argue that b_1 and b_2 must have some horizontal overlap. Suppose b_1 and b_2 have no horizontal overlap. Then the width of the layout is already $2(x+24y)+1$. Consider the position of b_3 in this case. If it is above either b_1 or b_2 this gives a height of $5y$ and the size bound K is exceeded. So both b_3 and b_4 cannot overlap horizontally with either b_1 or b_2. If b_3 and b_4 have no horizontal overlap, this gives an area of

$$\begin{aligned}(12y+12y+2(x+24y))(3y-1+1) &= 6y(x+36y)\\ &= 6yx+216y^2\\ &> 8yx+192y^2 > K,\end{aligned}$$

since $y > x$. If on the other hand they overlap horizontally, we have

$$(12y+2(x+24y))(2y-2+2y-2+1+1+1) = (8y-1)(x+30y) > 8y(x+24y) = K.$$

Thus b_1 and b_2 must have some horizontal overlap. Again consider b_3. If it does not have any horizontal overlap with either b_1 or b_2, then we have an area of at least

$$(12y+x+24y)(3y-1+3y-1+2) = 6y(x+36y) > K.$$

The same argument holds for b_4, thus both b_3 and b_4 must overlap horizontally with both b_1 and b_2. This gives a height of at least $8y$ which is the maximum since the width is at least $x+24y$. Thus the layout must fit into the box which is $8y$ by $x+24y$. There are essentially two configurations, either b_3 and b_4 are sandwiched in between b_1 and b_2 or they are on top of both. The latter can be eliminated because

additional tracks on the sides of the layout would be needed to connect the u pins and there is no more room for tracks horizontally. The layout in Figure 4.5, is the only possible arrangement of the blocks which meets the size bound K. The remaining blocks must then be in the x by $2y$ rectangle left in the middle by these four blocks. We argue that they must be arranged in rows which correspond to partitions. We show first that each block must be placed so that its pins are on the top and bottom sides. If not then there is a block of width $4m^2B$. But $x = B+3m+1 < 4m^2B$ if $B \geq 1$ and $m \geq 1$. Thus each block must have its pins on the top and bottom. We now show how the arrangement of the blocks corresponds to a partition. The following procedure is used to obtain the partitions.

Assume all blocks are initially unmarked and repeat the following procedure until no more blocks remain.

1. Create a new partition. Select the unmarked block which has a minimal y-coordinate (all blocks are of the same height), mark it and add it as the first member of the new partition. Associate the lower y-coordinate of this block with this partition.

2. Examine the remaining unmarked blocks. Add any blocks to the current partition whose lower y-coordinate is no greater than $4m^2B$ plus the coordinate of the current partition. Mark any blocks added.

3. Repeat steps 1 and 2 until all blocks are marked.

In this manner we obtain a partition $A_1, \ldots, A_k$. We will argue that k must in fact be m and the sum of the sizes of each partition must be B. This will be based on the following property of the partitions.

1. If $c(A_i)$ denotes the coordinate of A_i, then $c(A_i) + 4M^2B < c(A_{i+1})$.

2. Any two blocks in the same partition have no horizontal overlap.

The first property is clear by construction. The second property is observed by noting that if two blocks have a horizontal overlap then their lower coordinates must differ by more than $4m^2B$. We first argue that $k \le m$. Suppose so $k > m$. The upper coordinate of the minimal block in A_k is at least

$$k(4m^2B) \ge (m+1)(4m^2B) = 4m^3B+4m^2B > 4m^3B+3m^2+4m+1 = 2y$$

for $m \ge 3$ and $B \ge 2$. Thus $k \le m$. Now consider each partition. Since no two blocks within a partition can have a horizontal overlap we must have

$$\sum_{a \in A_i} s(a) \le x = 3m+1+B.$$

In addition to this, if there are h blocks in the A_i then there are $3m-h$ other blocks whose lower coordinate is either above and below the coordinate of A_i. Note that all of the blocks of A_i have a vertical overlap which is at a minimum of one point. Since each block must connect both to b_1 and b_2 there must be at least $3m-h$ vertical grid lines in between the blocks of A_i which are used for these connections. In addition, there is one unit of space in between the blocks. This gives us the following more stringent bound,

$$3m-h+h+1+\sum_{a \in A_i} s(a) \le x = 3m+1+B \text{ which implies, } \sum_{a \in A_i} s(a) \le B.$$

Since the sum of all of the $s(a)$'s is mB and there are at most m partitions, we must have $k=m$ and $\sum_{a \in A_i} s(a) = B$ for each $1 \le i \le m$.

To conclude that the problem of optimal layout of planar circuits is in fact NP-hard

in the strong sense, it suffices to observe that the **3-Partition** problem which was reduced to it is NP-complete in the strong sense and that the construction given is polynomial in $m + \log B$. Note that the solution to the construction is also polynomial in $m + \log B$.

□

The planar circuit in the construction used to reduce **3-Partition** could have resulted from an FP expression. If we have primitives, f_1, f_2, f_3 and f_4 which generate the boxes b_1, b_2, b_3 and b_4 as well as a primitive f_{a_i} generating the box b_{a_i} for each $1 \le i \le 3m$, then the FP function below could map to the planar circuit in the construction.

$$f_4 \mathbin{@} [f_3, f_{a_1}, \ldots, f_{a_{3m}}, f_4] \mathbin{@} f_1$$

4.3 Packing Planar Circuits into Cross-sections

In this section and the next, we present a mapping of planar circuits generated from FP expressions to abstract layouts. In a later section we will discuss how to transform these abstract layouts into actual layouts. The procedure detailed in these sections is designed to produce a layout efficiently. It does not attempt to optimize the layout.

The mapping consists of two steps. First the wires, branchings and crossings necessary to implement the R-nodes are generated and packed vertically along with the wires and components of the planar circuit. The result of the vertical packing is a sequence of horizontal cross-sections. Each cross-section is a list in left to right order of parts of boxes, wires, crossings and branchings which occupy the cross-

section. To obtain the abstract layout, these cross-sections are stacked vertically constraints are generated to make them "fit together" and horizontal compaction is then performed. This will be discussed in the next section. In this section we describe the mapping to sequences of cross-sections.

The computation tree defined by an FP expression as described in the last chapter is mapped to sequences of cross-sections. A cross-section is a horizontal slab of the layout. Imagine drawing horizontal lines across the layout. If these lines are sufficiently close so that each box hits at least two lines and every pair of horizontal wire segments with different y-coordinates is separated by at least one line, then we can order the "boxes and wires" which intersect this cross-section, in left to right order. We will refer to the representation of the planar circuits by sequences of cross-sections as the "intermediate form," IF. Instead of flattening the hierarchical representation of the planar circuit provided by the combining forms and then attempting to pack planar circuits into the intermediate form, we decompose the packing in terms of the computation tree. Each leaf of the computation tree generates a part of the planar circuit already packed into cross-sections and each combining form packs the IF's generated by its sub-trees. Some combining forms have alternative methods of packing. At the moment the layout procedure does not itself choose which method to use, but relies on the designer to label a combining form when a method other than the default is desired. This type of hierarchical representation could be imposed on any planar circuit and then used to obtain an abstract layout. However, the quality of the layout will depend on how well this hierarchy captures the layout. We first describe the intermediate form and then discuss the mapping.

Formally, the IF consists of FP objects of the following form,

$<CS_1\ CS_2\ \cdots\ CS_n>$ for $n \geq 0$,

where each CS_i is a cross-section. A cross-section is a list of FP objects each corresponding to wires, branchings, crossings and parts of boxes.

$CS_i = <x_1\ x_2\ \cdots\ x_m>$ where x_j is one of the following.

1. *Free wire*

   An atom other than $, *, +, ^, *t*.

   Elements of this type are wires which traverse the cross section without being crossed by any other wires.

2. *Crossing*

   $<*\ w\ *\ u_1\ u_2\ \cdots\ u_h>$ such that each u_i is an atom other than $, *, *t*, and at least two of the u_i's belong to {+, ^}.

   This type of element represents the wire crossings and branchings necessary for realizing the *R*-nodes of the planar circuit. The atom 'w' corresponds to the wire which must cross and/or branch in this cross-section. A horizontal wire is created and is connected to wires in the previous cross-section corresponding to the atoms '+'. For each '^' a new wire labeled 'w' is created and connected to the horizontal track. The other atoms are wires which traverse this cross-section without connecting to the horizontal wire.

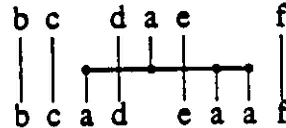
3. *Box*

   $<\$\ \textit{level}\ \#\textit{levels}\ \textit{id}\ \textit{label}\ \textit{width}\ \$\ i_1\ i_2\ \cdots\ i_k\ \$\ o_1\ o_2\ \cdots\ o_l\ \$>$

   Elements of this type correspond to the parts of components which are to be

Free Wires : a

Crossings : < * a * b c ^ d + e ^ ^ f >



Boxes :

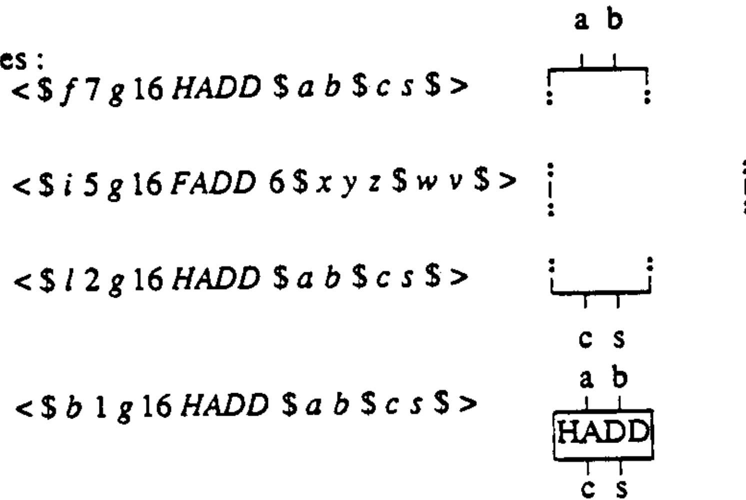


Figure 4.7 Graphical interpretation of the three element types.

drawn as boxes. Boxes may occupy any number of cross-sections. This format allows the specification of how many cross-sections a box will occupy. The *level* is **f**, **l**, **i** or **b**, indicating whether this is the first, last, intermediate or both (when a box is wholly contained within one) cross-section which the box occupies; an element of this type is instantiated for each cross-section in which it appears. The next three atoms are, respectively, the number of cross-sections occupied by this box, a unique identifier (which can be used to distinguish a box from others with the same

label), and a label to be displayed with the box. The next element is optional; if it is present it is the width of the box. The $'s act as delimiters between these atoms, the input atoms, $i_1, i_2 \cdots i_n$ and the output atoms $o_1, o_2 \cdots o_m$.

The graphical interpretations of these three types of elements are illustrated in Figure 4.7.

The following IF is generated for the exclusive-or function implemented with four Nand gates given in Section 3.1, **mxor**.

Nand @ &Nand @ [[1,2],[2,3]] @ [1,Nand,2] ; < a , b >

```
((a b)
 ((* a * + ^ ^) (* b * + ^ ^))
 (a a b b)
 (a ($ f 2 G6 Nand $ a b $ G7 $) b)
 (a ($ 1 2 G6 Nand $ a b $ G7 $) b)
 (a (* G7 * + ^ ^) b)
 (a G7 G7 b)
 (($ f 2 G8 Nand $ a G7 $ G9 $) ($ f 2 G10 Nand $ G7 b $ G11 $))
 (($ 1 2 G8 Nand $ a G7 $ G9 $) ($ 1 2 G10 Nand $ G7 b $ G11 $))
 (G9 G11)
 (($ f 2 G12 Nand $ G9 G11 $ G13 $))
 (($ 1 2 G12 Nand $ G9 G11 $ G13 $))
 (G13)
)
```

Figure 4.8 illustrates how this FP object could be interpretated graphically. The dashed lines separate the cross-sections.

We illustrate with examples the packing provided for each combining form and the leaves of the computation tree. Each sub-tree returns a packing of its planar circuit with the inputs on top and the outputs on the bottom. The packing performed for each of the combining forms is essentially some combination of two basic

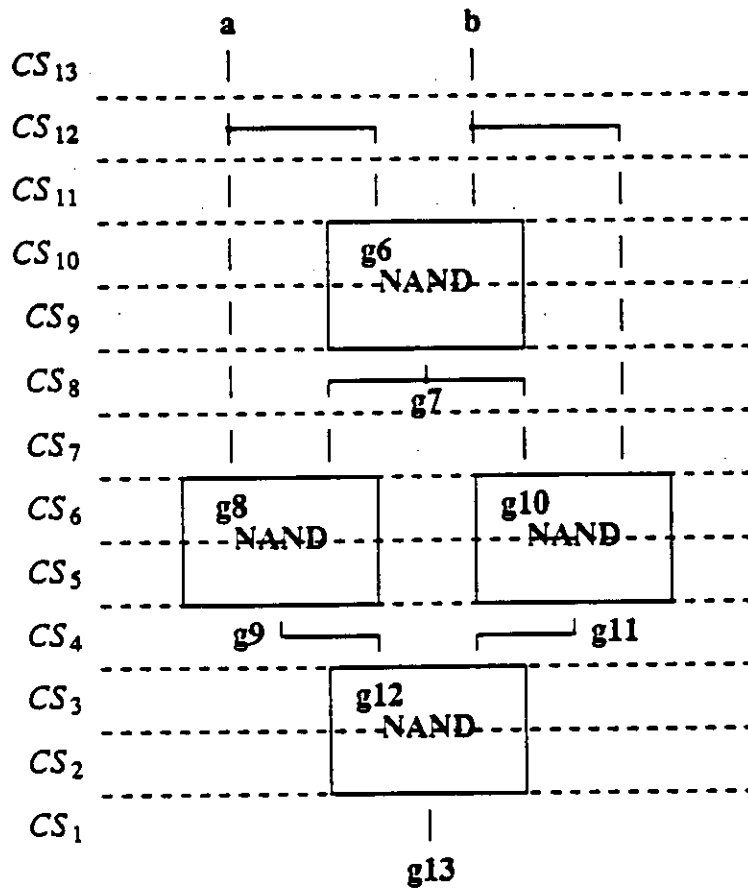


Figure 4.8 The graphical interpretation of the IF for **mxor**.

packings: packing vertically and packing horizontally. For example, for the **Compose** we pack the sub-trees vertically and for the **Construct** we pack the sub-trees horizontally and then pack the R-node u_D on top.

Packing vertically is the simplest of the two operations; since each IF is a sequence of cross-sections which will be stacked vertically, we simply concatenate the two sequences of cross-sections in order to stack one set on top of the other. Packing horizontally is more complicated. We have two stacks of cross-sections which we would like to pack horizontally, side by side. If the stacks are not of the same height then we extend the shorter one by repeating a cross-section consisting

only of the input wires of the sequence. We are in effect stretching the wires so that both IF's have the same height. Once the number of cross-sections in each sequence (stack) is the same, we concatenate the cross-sections which are adjacent horizontally. To accomplish this we step through the sequences taking one cross-section from each sequence and concatenating this pair of cross-sections. We can generalize this to accommodate more than two sequences at a time.

As discussed earlier a combining form may have more than one method of packing the IF's of its sub-trees; we will show two packings for the **Seq** combining form.

The packing of the leaves of the computation tree which correspond to computational primitives consist of cross-sections containing the box generated for the computational primitive whose width and number of cross-sections can be specified. Defaults are specified for FP primitives. In addition, any sub-tree corresponding to a defined FP function can be represented as a box. Functions generating packings for each of the FP routing primitives are provided. Each function generates cross-sections consisting of crossings necessary to implement the the routing primitive in question. There is a general routing function which is used for the u_D R-nodes required for the **Construct** combining form. This function is also used to route arbitrary sub-trees which correspond to R-nodes (have only R-nodes as leaves). As discussed in Section 3.8, the merging of R-nodes is acheived by considering a routing sub-tree to be one R-node. This function begins first by attempting to cleanly divide the R-node in question and then routes the resulting R-nodes. Thus the clean divides necessary to obtain a maximal-indivisible planar circuit are performed by this function.

In the following, the IFs of several FP expressions are given. The angle brackets ('<','>') are replaced by parentheses. An additional cross-section at the top and bottom are added; these cross-sections would not be part of the IF of these expressions were they to appear as a sub-tree. They are added only to the IF of roots of computation trees to provide a more visible set of inputs and outputs. The input FP expression appears in bold type. New symbols which are generated are labeled gN where N is the nth symbol generated. We begin with FP primitives.

These first two primitives are computational and generate boxes. The first cross-section generated for a box is a cross-section in which the inputs appear as free wires. This provides a gap between boxes.

andg ; <1 1>

```
(G1 G2)
(G1 G2)
(($ F 2 G4 AND 3 $ G1 G2 $ G3 $))
(($ L 2 G4 AND 3 $ G1 G2 $ G3 $))
(G3)
```

notg ; 1

```
(G5)
(G5)
(($ B 1 G7 NOT 2 $ G5 $ G6 $))
(G6)
```

apndr is a routing primitive in which the relative positions of the atoms do not change. In this case no routing is required so no cross-sections are generated. This is equivalent to an *R*-node which can be completely divided into trivial nodes. This is in agreement with our definition of a reasonable layout procedure since nothing is generated for this type of *R*-node.

```
apndr ; ((a b c d e) f)
(G8 G9 G10 G11 G12 G13)
(G8 G9 G10 G11 G12 G13)
```

Several cross-sections are generated for **trans** in order to realize the routing of this *R*-node. Because the *R*-node corresponds to an FP routing primitive we use the routing function associated with it.

```
trans ; <<x1 x2 x3> <y1 y2 y3> <z1 z2 z3>>

(G14 G15 G16 G17 G18 G19 G20 G21 G22)
(G14 G15 G16 G17 G18 (* G19 * + G20 G21 ^) G22)
(G14 G15 G16 G17 (* G18 * + G20 ^) G21 G19 G22)
(G14 G15 (* G16 * + G17 G20 G18 G21 ^) G19 G22)
(G14 (* G15 * + G17 G20 ^) G18 G21 G16 G19 G22)
(G14 G17 G20 G15 G18 G21 G16 G19 G22)
```

The following packings are the standard packings of the combining forms. Alternate packings can be used. An alternate packing for the **Seq** combining form will be given as well. A line which is indented is the continuation of a cross-section from the previous line.

The packing of **Compose** is the vertical packing.

```
notg @ andg ; <1 0>

(G23 G24)
(G23 G24)
(($ F 2 G28 AND 3 $ G23 G24 $ G25 $))
(($ L 2 G28 AND 3 $ G23 G24 $ G25 $))
(G25)
(($ B 1 G27 NOT 2 $ G25 $ G26 $))
(G26))
```

Construct is one of the more complicated forms to pack. It must generate the necessary cross-sections to implement the routing required by its u_D R-node and then pack this on top of the horizontal packing of its sub-trees.

[andg, org, norg, nandg] ; <1 0>

```
(G29 G30)
(G29 (* G30 * + ^ ^ ^ ^))
((* G29 * + ^ G30 ^ G30 ^ G30 ^) G30)
(G29 G30 G29 G30 G29 G30 G29 G30)
(($ F 2 G35 AND 3 $ G29 G30 $ G31 $)
       ($ F 2 G36 OR 3 $ G29 G30 $ G32 $)
       ($ F 2 G37 NOR 3 $ G29 G30 $ G33 $)
       ($ F 2 G38 NAND 3 $ G29 G30 $ G34 $))
(($ L 2 G35 AND 3 $ G29 G30 $ G31 $)
       ($ L 2 G36 OR 3 $ G29 G30 $ G32 $)
       ($ L 2 G37 NOR 3 $ G29 G30 $ G33 $)
       ($ L 2 G38 NAND 3 $ G29 G30 $ G34 $))
(G31 G32 G33 G34)
```

The constant form generates a box with no inputs and whose outputs are a symbolic representation of the object it is applied to.

%<new object> <anything>

```
()
(($ B 1 G41 "NEW" 2 $ $ G39 $)
       ($ B 1 G42 "OBJECT" 2 $ $ G40 $))
(G39 G40))
```

Apply-to-All is simply the horizontal packing of its sub-trees. The combining form introduced in Section 3.8, **Projection**, is handled in the same manner.

& andg ; <<1 1> <1 0> <0 1> <0 0>>

```
(G42 G43 G44 G45 G46 G47 G48 G49)
(G42 G43 G44 G45 G46 G47 G48 G49)
(($ F 2 G54 AND 3 $ G42 G43 $ G50 $)
      ($ F 2 G55 AND 3 $ G44 G45 $ G51 $)
      ($ F 2 G56 AND 3 $ G46 G47 $ G52 $)
      ($ F 2 G57 AND 3 $ G48 G49 $ G53 $))
(($ L 2 G54 AND 3 $ G42 G43 $ G50 $)
      ($ L 2 G55 AND 3 $ G44 G45 $ G51 $)
      ($ L 2 G56 AND 3 $ G46 G47 $ G52 $)
      ($ L 2 G57 AND 3 $ G48 G49 $ G53 $))
(G50 G51 G52 G53))
```

The **Right Insert** combining form packs each subtree horizontally with the list of inputs of the sub-trees to its left. It then packs these vertically with the leftmost on top.

! andg ; <1 1 1 1 1 1>

```
(G58 G59 G60 G61 G62 G63)
(G58 G59 G60 G61 G62 G63)
(G58 G59 G60 G61 ($ F 2 G73 AND 3 $ G62 G63 $ G64 $))
(G58 G59 G60 G61 ($ L 2 G73 AND 3 $ G62 G63 $ G64 $))
(G58 G59 G60 G61 G64)
(G58 G59 G60 ($ F 2 G72 AND 3 $ G61 G64 $ G65 $))
(G58 G59 G60 ($ L 2 G72 AND 3 $ G61 G64 $ G65 $))
(G58 G59 G60 G65)
(G58 G59 ($ F 2 G71 AND 3 $ G60 G65 $ G66 $))
(G58 G59 ($ L 2 G71 AND 3 $ G60 G65 $ G66 $))
(G58 G59 G66)
(G58 ($ F 2 G70 AND 3 $ G59 G66 $ G67 $))
(G58 ($ L 2 G70 AND 3 $ G59 G66 $ G67 $))
(G58 G67)
(($ F 2 G69 AND 3 $ G58 G67 $ G68 $))
(($ L 2 G69 AND 3 $ G58 G67 $ G68 $))
(G68)
```

Seq can be packed in a similar manner as the **Right Insert** except that each subtree must also be packed horizontally with the list of outputs of sub-trees to its

right.

seq(hadd) ; <0 1 0 1 0 1 0>

```
(G70 G71 G72 G73)
(G70 G71 G72 G73)
(G70 G71 ($ F 3 G82 HADD 4 $ G72 G73 $ G74 G75 $))
(G70 G71 ($ I 3 G82 HADD 4 $ G72 G73 $ G74 G75 $))
(G70 G71 ($ L 3 G82 HADD 4 $ G72 G73 $ G74 G75 $))
(G70 G71 G74 G75)
(G70 ($ F 3 G81 HADD 4 $ G71 G74 $ G76 G77 $) G75)
(G70 ($ I 3 G81 HADD 4 $ G71 G74 $ G76 G77 $) G75)
(G70 ($ L 3 G81 HADD 4 $ G71 G74 $ G76 G77 $) G75)
(G70 G76 G77 G75)
(($ F 3 G80 HADD 4 $ G70 G76 $ G78 G79 $) G77 G75)
(($ I 3 G80 HADD 4 $ G70 G76 $ G78 G79 $) G77 G75)
(($ L 3 G80 HADD 4 $ G70 G76 $ G78 G79 $) G77 G75)
(G78 G79 G77 G75)
```

This packing is vertical; the sub-trees are packed one on top of the other and the wires are extended to the top and bottom most cross-sections. Another packing of the Seq for the same FP expression is given below. In this packing, the sub-trees are packed horizontally and the wires which connect the sub-trees are routed back up in between the sub-trees.

```
(G83 G84 G85 G86)
(G83 (* G89 * ^ ^) G84 (* G87 * ^ ^) G85 G86)
(($ F 3 G93 HADD $ G83 G89 $ G92 G91 $) G89
        ($ F 3 G94 HADD $ G84 G87 $ G89 G90 $) G87
        ($ F 3 G95 HADD $ G85 G86 $ G87 G88 $))
(($ I 3 G93 HADD $ G83 G89 $ G92 G91 $) G89
        ($ I 3 G94 HADD $ G84 G87 $ G89 G90 $) G87
        ($ I 3 G95 HADD $ G85 G86 $ G87 G88 $))
(($ L 3 G93 HADD $ G83 G89 $ G92 G91 $) G89
        ($ L 3 G94 HADD $ G84 G87 $ G89 G90 $) G87
        ($ L 3 G95 HADD $ G85 D $ G87 G88 $))
(G92 G91 (* G89 * + +) G90 (* G87 * + +) G88)
(G92 G91 G90 G88)
```

As discussed earlier, conditionals are evaluated and only the branch selected is retained.

(atom → notg; nandg) ; <1 0>

```
(G95 G96)
(G95 G96)
(($ F 2 G98 NAND 3 $ G95 G96 $ G97 $))
(($ L 2 G98 NAND 3 $ G95 G96 $ G97 $))
(G97)
```

(atom → notg; nandg) ; 1

```
(G99)
(G99)
(($ B 1 G101 NOT 2 $ G100 $ G99 $))
(G100)
```

The last example illustrates the difference between the packing obtained by interpreting a routing sub-tree, using the packings of its leaves and combining forms, rather than considering the sub-tree as one *R*-node and using the general packing function. The first IF is the one obtained by interpreting the computation tree while the second is obtained by applying the general routing function to the root of the sub-tree, which is equivalent to packing the maximal-indivisible circuit in this case. The advantage of laying out a maximal-indivisible planar circuit is clearly seen in this example since the former IF uses six cross-sections while the latter uses only three.

[distl@[1,[2,3]],distl@[3,[5,4,6]],trans@[[9,7],[8,6]]] ; <a b c d e f g h i>

```
(G8 G9 G10 G11 G12 G13 G14 G15 G16)
(G8 G9 G10 G11 G12 (* G13 * + ^ ^) G14 G15 G16)
(G8 G9 (* G10 * + ^ ^) G11 G12 G13 G13 G14 G15 G16)
(G8 G9 G10 G10 G11 G12 G13 G13 G14 (* G15 * + G16 ^))
(G8 G9 G10 G10 G11 G12 G13 (* G13 * + G14 G16 ^) G15)
(G8 G9 G10 G10 (* G11 * + G12 ^) G13 (* G14 * + G16 ^)
        (* G13 * + G15 ^))
((* G8 * + ^ G9 ^) G10 (* G10 * + ^ G12 ^ G11 ^) G13 G16
        (* G14 * + G15 ^) G13)
(G8 G9 G8 G10 G10 G12 G10 G11 G10 G13 G16 G15 G14 G13)
```

```
(G7 G8 G9 G10 G11 G12 G13 G14 G15)
(G7 G8 G9 G10 G11 G12 G13 (* G14 * + G15 ^))
(G7 G8 G9 (* G10 * + G11 ^) G12 (* G13 * + G15 G14 ^))
((* G7 * + ^ G8 ^)
       (* G9 * + ^ ^ G11 ^ G10 ^)
       (* G12 * + ^ G15 G14 G13 ^))
(G7 G8 G7 G9 G9 G11 G9 G10 G9 G12 G15 G14 G13 G12)
```

The mapping from the computation tree to IF is implemented in T. The IF can be written in a file to be retained.

4.4 Mapping Sequences of Cross-sections to Abstract Layouts

The final step necessary to obtain the sketch is to assign coordinates to elements of the IF: the wires, crossings, branchings and boxes. Elements can be assigned the vertical coordinates of their cross-sections, but obtaining horizontal coordinates is more involved since conflicts with elements in adjoining cross-sections must be resolved. Each element has a geometrical interpretation within its cross-section (as illustrated in Figure 4.7). To position the elements, spacing constraints between adjacent elements in a cross-section and possibly elements in the cross-sections immediately above and below, must be respected. To find horizontal positions, the IF is examined cross-section by cross-section to build a horizontal constraint graph encompassing these spacing constraints. Horizontal positions can then be obtained from this graph.

In a horizontal constraint graph two nodes are connected by a directed edge reflecting a constraint between these two nodes. An edge of length d from node n_1 to node n_2 corresponds to the constraint expressed by the inequality,

$$p_1 + d \leq p_2,$$

where p_1 and p_2 are the positions (coordinates) of n_1 and n_2 respectively. If d is non-negative then n_2 must be to the right of n_1 by at least d. If d is negative then n_1 cannot be to the right of n_2 by more than $-d$. The horizontal constraint graph reflects the interactions among these inequalities and provides a convenient data structure for resolving them.

The constraint graph is constructed by traversing each cross-section. Spacing constraints are generated for each element with the elements to its left and in the cross-section above. The type and number of constraints depends on the elements involved and is detailed below. Nodes in this graph will correspond to vertical line segments or boxes. For each node, the list of its outgoing edges, its indegree, its vertical coordinates and its connections to other nodes is recorded. For boxes, the width, rightmost output and input are also recorded. As each cross-section is traversed, the list of nodes connecting to elements in the next cross-section is maintained. This list will be simultaneously traversed with the next cross-section to establish connections between elements in these two adjacent cross-sections.

Free Wires

> A free wire can have a horizontal jog in the cross-section since it is not crossed. The node for this wire in the cross-section above is extended halfway down into the current cross-section and then a new node is created for this wire. This new node is not directly constrained to the previous one, enabling this wire to jog either to the right or left. Constraints with nodes to the right and left in this cross-section will determine the direction of this jog.

Crossings : $<* \; w \; * \; u_1 \; u_2 \, , \ldots , \; u_h>$

For a crossing the list of u_i's is traversed. Each u_i which is not a '+' or a '^' corresponds to a wire segment which traverses this cross-section and cannot have a vertical jog since it is being crossed. (It is assumed that both u_1 and u_n are either '+' or '^'.) The node in the previous cross-section corresponding to this wire is extended down through this cross-section. For each '^' encountered, a new node is created which corresponds to a segment extending from the middle of the cross-section downward. For the '+', its node in the previous cross-section is extended halfway into this cross-section. Connections between the new nodes and the node corresponding to '+' are recorded with these nodes.

Boxes

A single node is used to represent a box. If this is the first level of this box then a new node is generated. The nodes in the cross-section above corresponding to its inputs are not extended. The position of a box is its leftmost edge. The width of the box is set to,

$$1 + \max\{\# \, inputs, \; \# \, outputs\}.$$

For an intermediate or the last level, the node in the previous cross-section corresponding to this box is extended down through the cross-section. For the last level new nodes are created for each of the outputs. These nodes correspond to vertical segments of length zero which sit at the bottom of the cross-section. (These nodes will be extended down when the next cross-section is processed). The rightmost input and output nodes of the box are recorded.

Figure 4.9 illustrates the general scheme for generating constraints. There are three possible constraints to be generated when node n_4 is processed (n_4 may be the same as n_2).

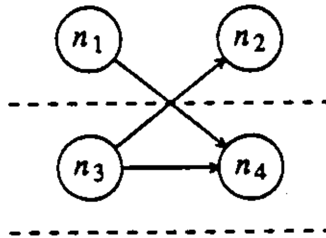


Figure 4.9 Constraints generated when node n_4 is processed.

Constraints are unit distance unless a box is involved. Constraints of length one are generated between a box and its leftmost input and output, however spacing constraints between a box and other nodes use one plus the width of the box as the distance. The constraints that have been mentioned so far, all have positive distances and result in an acyclic graph. Negative constraints are needed to preserve the width of the boxes. The inputs and outputs of a box can be no further right from the box node than its width minus one. The length of these edges is one minus the width of the box. Figure 4.10 shows the constraints between a box node and its input and output nodes. Figure 4.11 contains the elements of the graph for the example in Figure 4.8 and Figure 4.12, its constraint graph.

To facilitate the following discussion, an additional node, designated as the root, with an edge of length zero to every other node, is added to the graph. In an acyclic constraint graph, the unique optimal solution to the constraints is obtained by assigning to each node n, the length of a longest root to n path, as its position relative

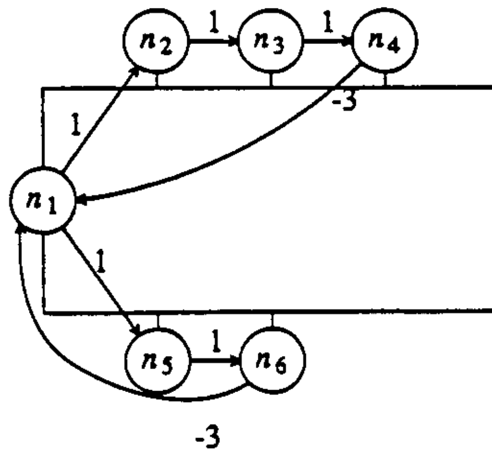


Figure 4.10 Constraints generated for node n_1, a box.

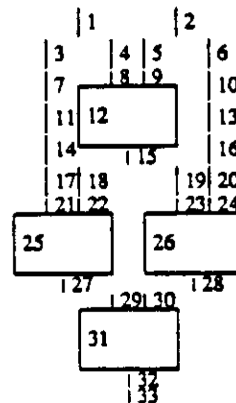


Figure 4.11 Elements for example of Figure 4.8.

to the root's. This solution is optimal in that no other set of positions satisfying the inequalities can assign a smaller position to any node. In [Liao83], it is shown that this result can be extended to an arbitrary digraph as long as the digraph does not contain a positive cycle. This is a natural restriction since a positive cycle corresponds to an inconsistency in the inequalities. In this case the constraints involved need to be adjusted to make the graph consistent.

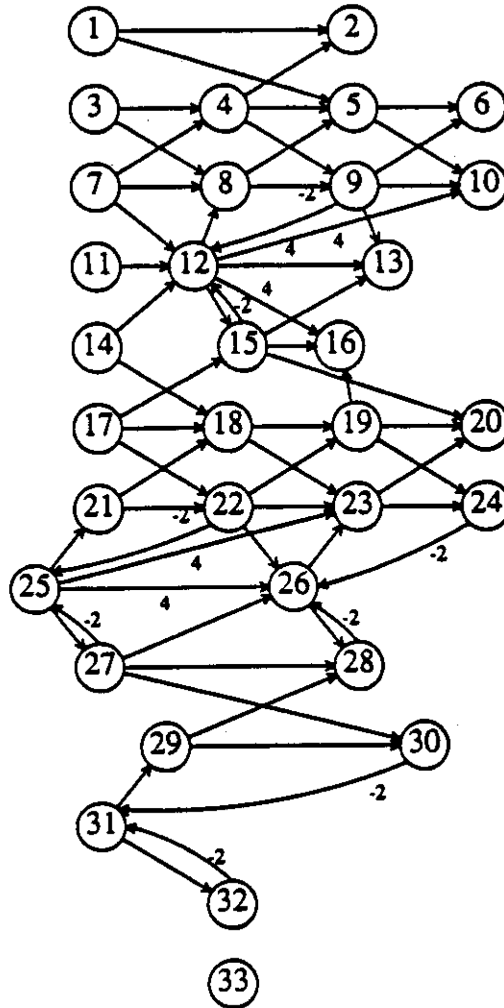


Figure 4.12 Constraint graph for example of Figure 4.8.

In an acyclic constraint graph, longest paths can be obtained by traversing the graph respecting the inherent partial ordering of the nodes. (The nodes can be ordered so that no edge connects a node with one that precedes it in the order.) When a node is visited, its outgoing edges are examined to determine if their inequalities are satisfied and if necessary, the positions of the nodes at the other end of these edges are adjusted to satisfy the constraints.

$$p_2 \leftarrow \max\{p_2, p_1 + d\} \tag{4.2}$$

In the algorithm proposed by Liao and Wong the negative edges (back edges) are treated separately. Since the only edges of length zero are from the root, and the graph has no positive cycles, the subgraph induced by the non-negative edges is acyclic. The positions of the nodes obtained in this subgraph are obtained and then the back edges are examined. If a back edge is not satisfied, then the position of the node is adjusted as in (4.2). The acyclic subgraph is traversed again updating positions as in (4.2). This procedure is repeated until all the back edges are satisfied. After repeating this process 1 + # *of back edges* times, if the back edges are not all satisfied, then the graph is inconsistent.

This algorithm is appropriate when the graph is consistent. Unfortunately not all graphs generated from IF's of FP expressions are consistent. Figure 4.13 contains an example in which the IF produces a constraint graph with a positive cycle. The problem occurs because the vertical packing leaves one horizontal track between boxes A,B,C,D and E, and this is not enough to bring the four wires into alignment for box E.

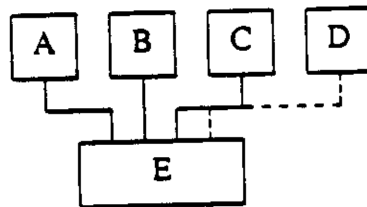


Figure 4.13 A situation corresponding to an inconsistent Constraint Graph.

This situation can be resolved either by widening the boxes (in this case box E) or adding more horizontal tracks. The former solution is used since it does not require changing the IF and the inconsistency may disappear when actual sizes of objects are

used to obtain a 'real layout,' as will be discussed in the next section. The positions obtained from traversing an inconsistent constraint graph are in general distorted since they are the result of traversing positive cycles several times. A heuristic algorithm is used which attempts to identify inconsistencies early and adjust the sizes of boxes. Once the boxes have been widened sufficiently to make the constraint graph consistent, the algorithm of Liao and Wong is applied to it. As in the algorithm of Liao and Wong, the acyclic subgraph induced by the positive edges is traversed repeatly and the back edges are adjusted in between each traversal. However not all the back edges are respected. The algorithm examines the back edges in the order imposed on them by their destination nodes. The first back edge which is not satisfied and was not previously adjusted is recorded and its destination node is adjusted as in (4.2). If this back edge is again not satisfied after subsequent traversals, the algorithm widens the box so that the back edge would be satisfied by the current positions of its inputs and outputs. The graph must eventually become consistent since eventually all back edges will be made more negative, and the positive edges remain fixed in value.

When the graph becomes consistent, the algorithm of Liao and Wong is then applied to it. The algorithm which resolves the inconsistencies is heuristic in that it guesses which box should be widened based on the partial ordering. It does not always identify the correct box to be widened; it may widen boxes unnecessarily. To mitigate this effect, after applying Liao and Wong's algorithm to the constraint graph that it has made consistent, the program inspects the boxes to determine if any of the widened boxes can be shrunk.

This heuristic algorithm does not guarantee the optimal solution when the graph is consistent, but it deals with inconsistent constraint graphs, yielding a reasonable though not perfect solution. To obtain the optimal solution when the graph is consistent, Liao and Wong's algorithm is attempted first and the heuristic algorithm is resorted to only when the graph is found to be inconsistent.

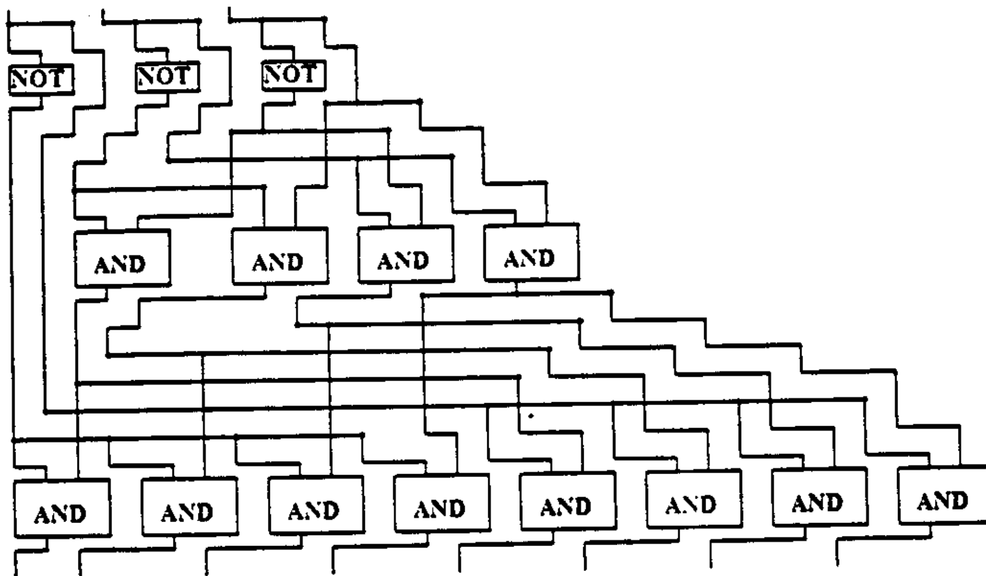


Figure 4.14 Before straightening the wires.

By using the constraint graph to obtain positions, the wires and boxes are pushed to the left as much as possible. Although this minimizes the area, it has the undesirable effect of routing the wires with unnecessary detours and bends. To remove this effect, the wires are sorted from right to left, and each is pulled back to the right to straighten it out as much as possible. An examples of the routing before and after 'straightening' the wires are given in Figures 4.14 and 4.15. Any boxes which were widened are inspected again after straightening the wires to determine if they can now be shrunk back to their original sizes.

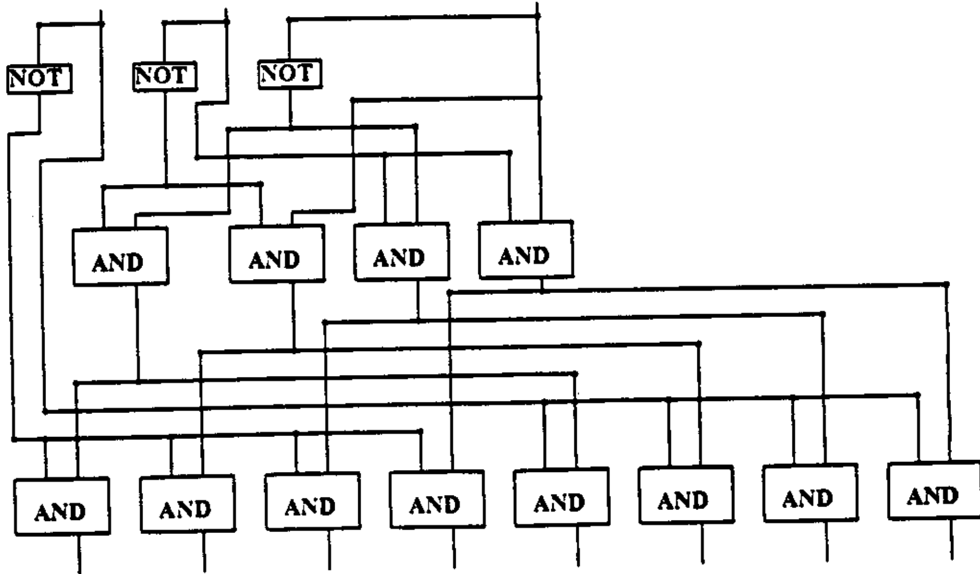


Figure 4.15 After straightening the wires.

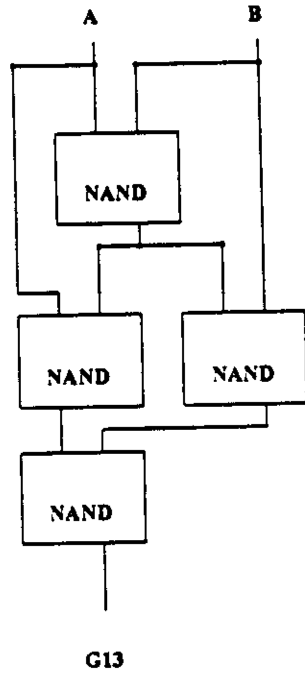
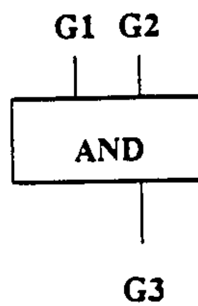


Figure 4.16 Abstract layout obtained for example of Figure 4.8

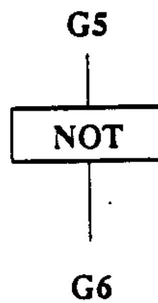
To continue the example, Figure 4.16 is the abstract layout generated for the example of Section 3.1, **mxor**.

To complete this section, we show the abstract sketches corresponding to the IF's generated in the Section 4.3.

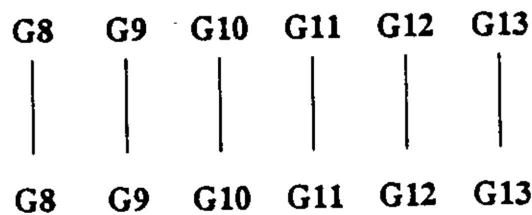
andg ; <1 1>



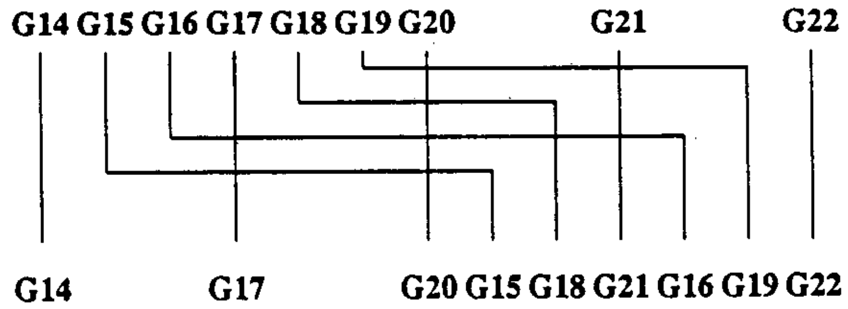
notg ; 1



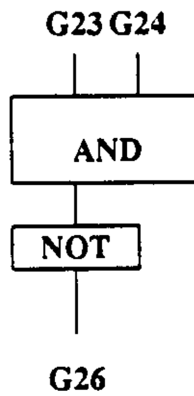
apndr ; ((a b c d e) f)



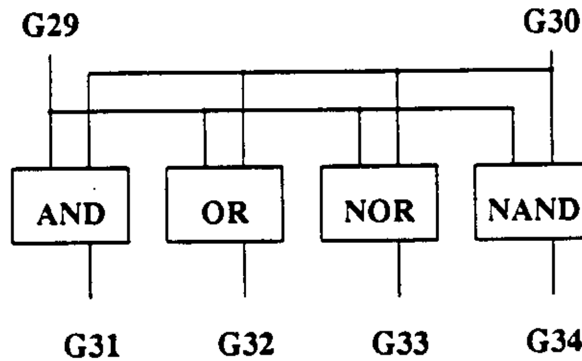
trans ; <<x1 x2 x3> <y1 y2 y3> <z1 z2 z3>>



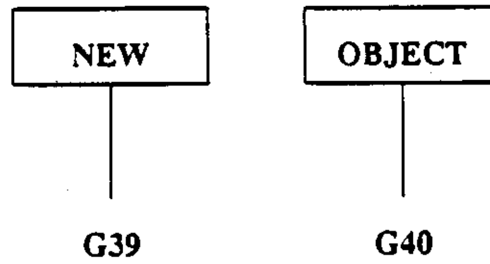
notg @ andg ; <1 0>



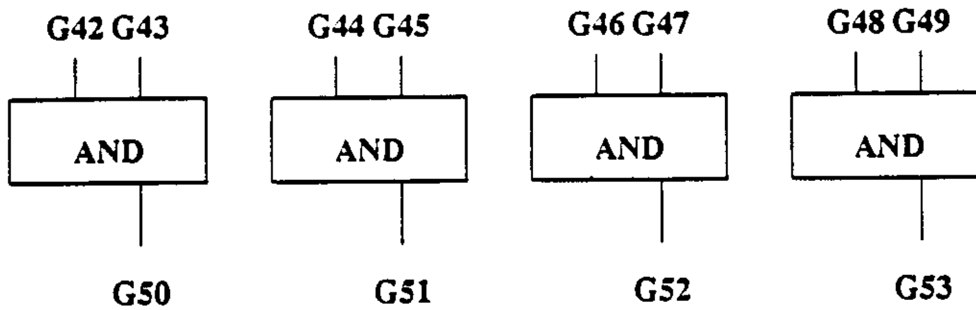
[andg, org, norg, nandg] ; <1 0>



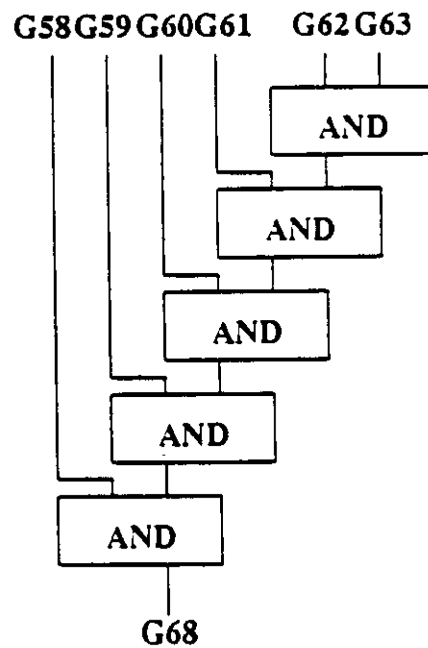
%<new object> <anything>



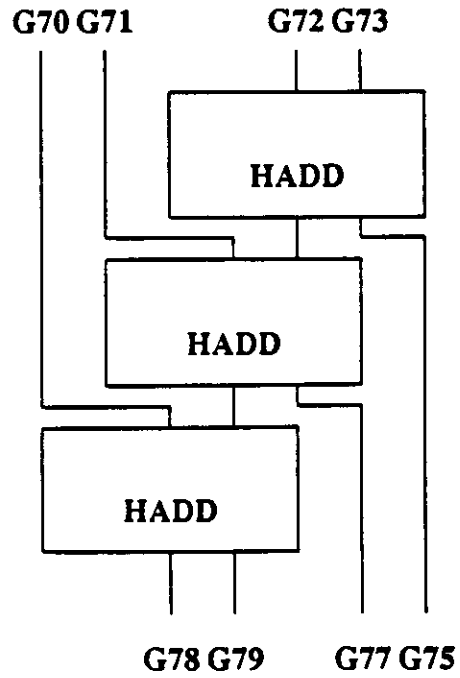
& andg ; <<1 1> <1 0> <0 1> <0 0>>



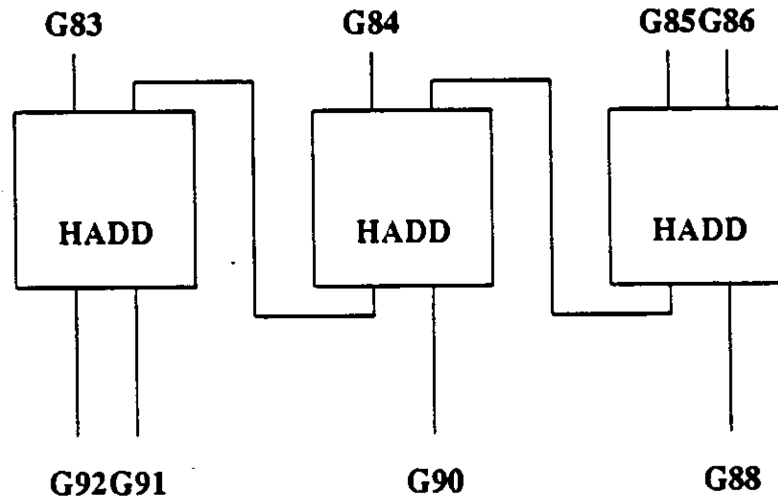
! andg ; <1 1 1 1 1 1>



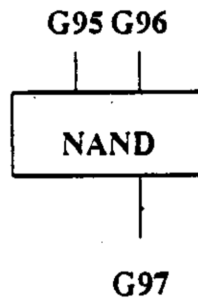
seq(hadd) ; <0 1 0 1 0 1 0>



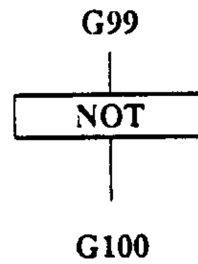
The second packing of seq.



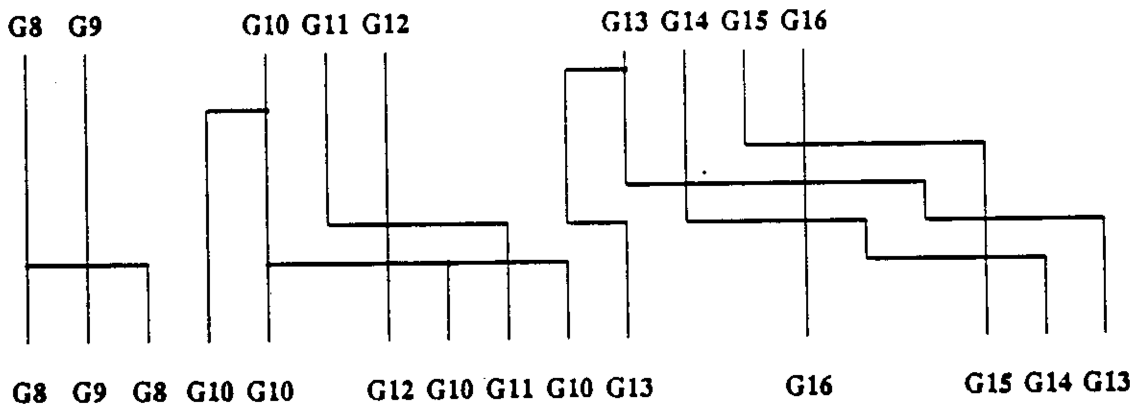
(atom → notg; nandg) ; <1 0>

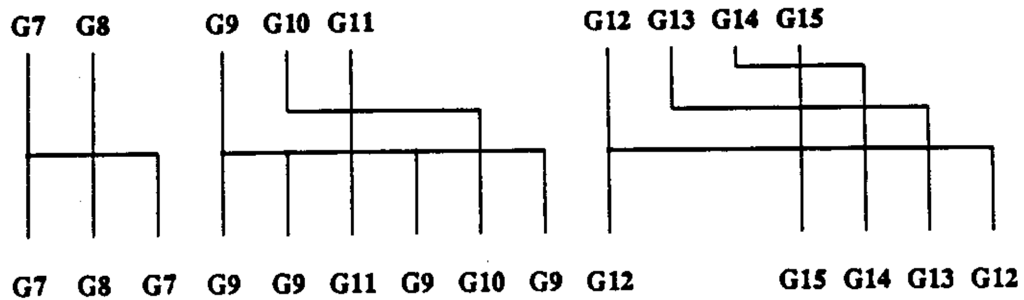


(atom → notg; nandg) ; 1



[distl@[1,[2,3]],distl@[3,[5,4,6]],trans@[[9,7],[8,6]]] ; <a b c d e f g h i>





Abstract layouts of more interesting functions will be given in the next chapter.

4.5 Transforming Abstract Layouts into Layouts

Ideally, a symbolic layout tool could take over the layout once the wires have been assigned layers. However since no such tool was available we generated fixed geometry directly from the IF. To transform an abstract layout to a layout, the layers on which the wires will be implemented must be selected and the layouts of the boxes must be specified and incorporated. The selection of layers to use for routing depends on how the power and ground wires will be organized and the number and characteristics of the layers available. Only two layers are required to implement the wires in the manner in which the R-nodes are implemented in the IF; all wires can be on one layer except for the horizontal wires generated for crossings. The method of packing the R-nodes should be varied according to the number of layers which are available. Rather than selecting the layers, we generate the wires on two abstract layers and the designer can decide which two layers to use.

In addition to the assignment of wires to layers, we must know the dimensions of the boxes and the constraints on where the inputs and outputs of boxes must be placed. The inputs and outputs must occur on the same layer used for wiring. Once we have all of this information, we can determine the exact dimensions of the elements of the IF and the minimum spacing required between them. The height of each vertical cross-section can be determined and the horizontal compaction can be performed using real values rather than the abstract ones.

Determining Heights of Cross-sections

In processing the IF to build the horizontal constraint graph, we determine the minimum height of each cross-section according to the elements it contains.

1. A box whose height is m lambdas and which occupies n cross-sections in all, requires each of the cross-sections it occupies to be at least $\left\lceil \frac{m}{n} \right\rceil$ lambdas.

2. Cross-sections containing crossings are required to be large enough to accommodate a contact and the distance between it and a wire.

3. Cross-sections in which jogs occur are required to be at least the width of a wire plus the minimum distance between wires.

Once the minimum height of each cross-section is determined, the vertical coordinates are assigned to the cross-sections by summing the heights of all of the previous cross-sections. This cannot be done until after all of the horizontal compaction and straightening of wires is completed since it is necessary to know whether there is a jog in a cross-section. The coordinate of each cross-section is then used as the y-coordinate of the objects contained in it. Since boxes may not occupy

the entirety of the cross-sections allocated to them, their output wires are stretched upwards to meet the actual boundary of the box.

The horizontal coordinates of elements are determined by compaction using real distances for the edges rather than abstract units. The position of outputs and inputs along the bottom and tops of boxes are specified by the minimum distance required between them as well as the ends of the boxes for the first and last in each case.

We will illustrate the process of transforming an abstract layout into a layout with the FP function implementing the exclusive-or which was introduced in Section 3.1 as an example whose abstract layout was given in Figure 4.16. We choose to implement it in *nmos* and hence we must design a **Nand** gate in *nmos* and provide its layout. Figure 4.17 contains the circuit diagram of an *nmos* **Nand** gate and Figures 4.18 and 4.19 contain two layouts of this circuit.

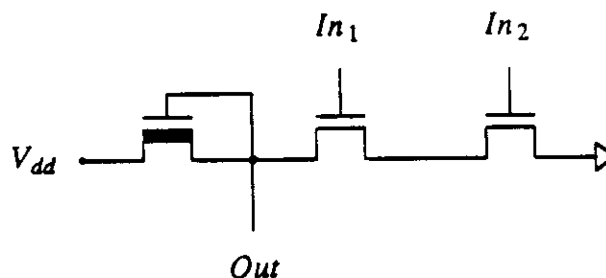


Figure 4.17 The schematic of the *nmos* **Nand** gate

Figure 4.18 A **Nand** gate in *nmos*

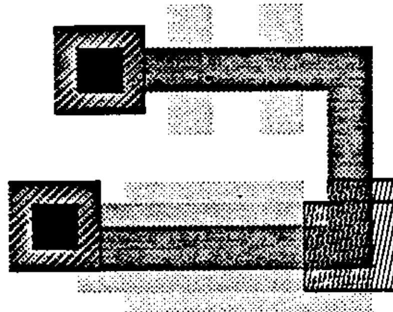


Figure 4.19 Another Nand gate in *nmos*

Notice that these two implementations both have the inputs on top and outputs at the bottom as required. The output in each case is the polysilicon rectangle used to form the enhancement transistor. We choose to implement the wires in polysilicon. Since there are no cross-overs in this example the second layer used for crossings is not used; we do not need to select a layer for routing the crossings. This is essentially all of the decisions that are necessary for this example. It remains only to provide this information to the IF compactor. This information is provided in a file. The following is the file describing the layout with the first Nand gate.

```
Vertical interwire dist: 2
Horizontal interwire dist: 2
Vertical wire width: 2
Horizontal wire width: 2
Contact size: 0
Contact horizontal wire distance: 0
Contact vertical wire distance: 0
Number of box types: 1

Box name : Nand
Width : 31
Height : 6
Inoutlist : 20 4 7 -1 5 20 -1
```

The 'Inoutlist' is the spacing required between the left edge of the box and the leftmost input followed by the spacing required between subsequent pairs of inputs, and then the space required between the rightmost input and the right edge of the box. A −1 acts as the delimiter and then same information is provided for the outputs. Notice that in this case the spacing required for the outputs does not sum up to the width of the box, providing the allowing the output to connect to the box anywhere within a range. The description of the second **Nand** gate is:

Vertical interwire dist: 2
Horizontal interwire dist: 2
Vertical wire width: 2
Horizontal wire width: 2
Contact size: 2
Contact horizontal wire distance: 0
Contact vertical wire distance: 0
Number of box types: 1

Box name : Nand
Width : 16
Height : 11
Inoutlist : 7 4 5 -1 5 2 -1

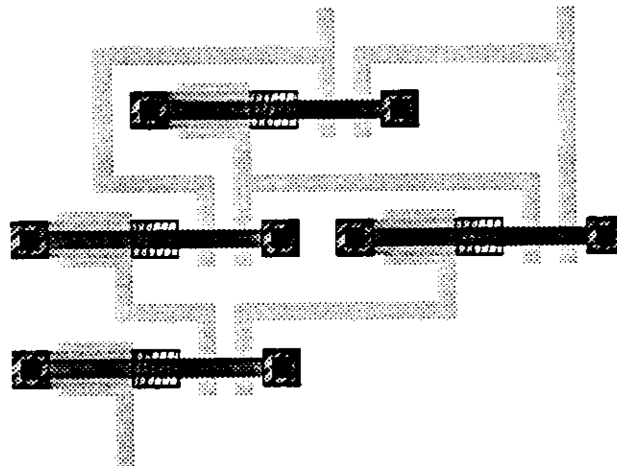


Figure 4.20 The layout of **mxor** using the first **Nand** gate

When the IF compactor is invoked with the IF corresponding to this function and these two files, the layouts in Figures 4.20 and 4.21 are generated.

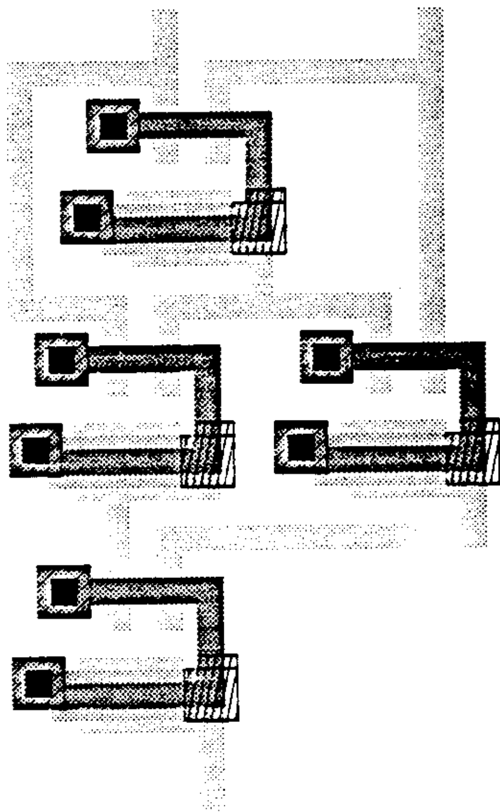


Figure 4.21 The layout of **mxor** using the second **Nand** gate

To complete the layout we must connect the power and ground contacts of the boxes. Figures 4.22 and 4.23 show the completed layouts.

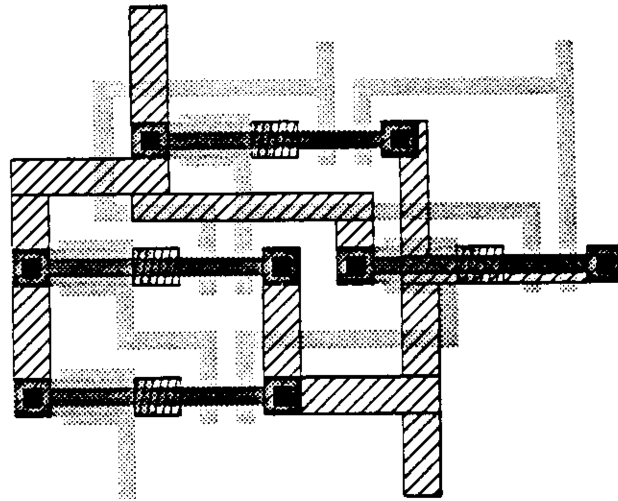


Figure 4.22 The completed layout of **mxor** using the first **Nand** gate

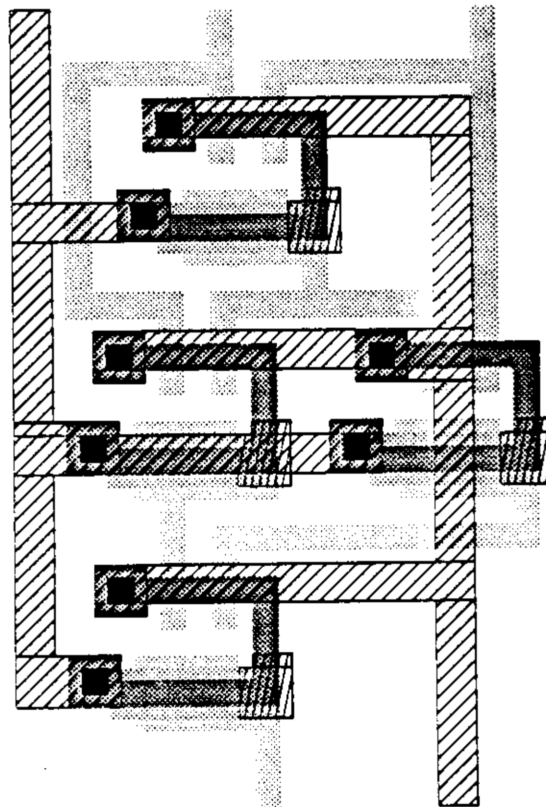


Figure 4.23 The completed layout of **mxor** using the second **Nand** gate

The layout of a more interesting function will be given in Chapter 5.

Summary

In this chapter, we have obtained 'abstract layouts' and actual layouts of the planar circuits resulting from FP expressions. Abstract layouts were defined to be layouts in which wires have zero width and the circuit elements are mapped to the grid such that all dimensions and coordinates are in grid units, and the sides of components and path segments of wires fall on grid lines. We showed that the problem of minimizing the area of a layout of a planar circuit is NP-hard. We provided a method for obtaining abstract layouts of planar circuits resulting from FP expressions. This method relied on the computation tree of an FP expression, using each combining form to combine the layouts of its sub-functions. The method consisted of packing circuit elements into horizontal cross-sections and then resolving constraints between these cross-sections by performing horizontal compaction. In Section 4.5 the transformation of 'abstract layouts' into actual layouts was described.

CHAPTER 5

Examples

In this chapter, examples of FP specifications and their sketches are presented to illustrate the correspondence between programming style in FP and resulting sketches. All of the sketches and layouts presented in this chapter were produced directly from the FP specifications, by tools written to implement of the methods described in Chapters 3 and 4. The specifications are the input to an FP interpreter which constructs the computation tree of the FP expression and generates the IF which is then passed to the compactor. The compactor produces an abstract sketch on a graphics terminal or on a laser writer; the layout is produced in the format accepted by the graphics editor, Magic [Oust83]. The FP interpreter is written in T [Rees83] and the compactor is written in C [Kern78]. These tools were developed and operate on a DEC VAX 11/750 under the UNIX operating system.

The performance of the heuristic compaction algorithm is most clearly visible in the FFT Section 4.5. One of the examples has been transformed to a layout. The actual FP specifications will be given. To distinguish functions defined in the specification from FP primitives, the former will start with an upper case letter. FP primitives and their definitions can be found in the Appendix. Lines beginning with #'s are comments.

5.1 Decoders

The decoder was one of the first circuits to be described in FP and has suffered numerous examinations since. A decoder accepts a bit vector of length n and generates a bit vector of length 2^n in which the ith bit is set if and only if i is the binary number represented by input vector. The original FP specification written by Lahti[La81] is:

```
{Decode !DecStage@&OneDecode}
{OneDecode [notg,id]}
{DecStage &andg@concat@&distl@distr}
```

The function Decode first obtains the complements of the inputs and then inserts the function DecStage from the right, consuming one variable and its complement at each stage.

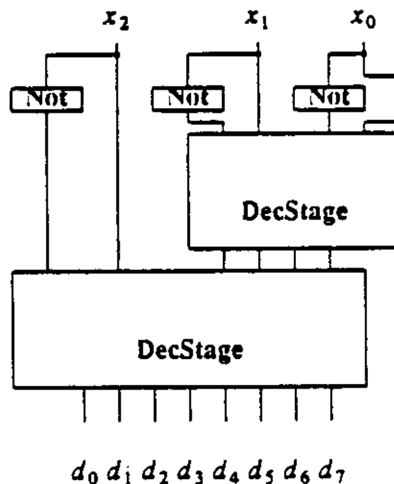


Figure 5.1 Decoder with DecStage as a primitive.

The input to DecStage is

$$\langle\langle \overline{x_i}\, x_i \rangle \langle d_0\, d_1\, \cdots\, d_{m-1} \rangle\rangle \quad \text{where} \quad m=2^{i-1},$$

and the output is

$$\langle f_0\, f_1\, \cdots\, f_{2m-1} \rangle \quad \text{where for} \quad 0<j<m-1, \quad f_j=\overline{x_j}\cdot d_j \quad \text{and} \quad f_{j+m}=x_j\cdot d_j.$$

This is accomplished by transforming the input object into,

$$\langle\langle \overline{x_i}\, d_0 \rangle \langle \overline{x_i}\, d_1 \rangle \cdots \langle \overline{x_i}\, d_{m-1} \rangle \langle x_i\, d_0 \rangle \langle x_i\, d_1 \rangle \cdots \langle x_i\, d_{m-1} \rangle\rangle$$

and applying **andg** to each pair in this list. Figure 5.1 is the sketch resulting from this algorithm with **DecStage** represented as a component, while in Figure 5.2 the definition of **DecStage** is evaluated.

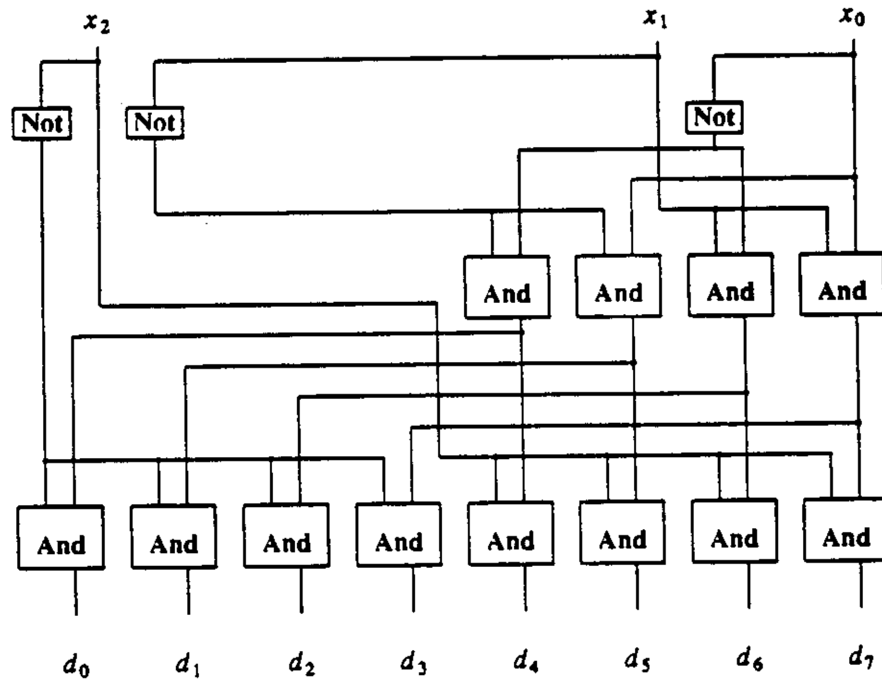


Figure 5.2 Decoder with **andg** and **notg** as primitives.

The compaction algorithm pushes the boxes to the left. In this case the **notg's** are pulled unnecessarily to the left. Space would be saved by pushing them back to the

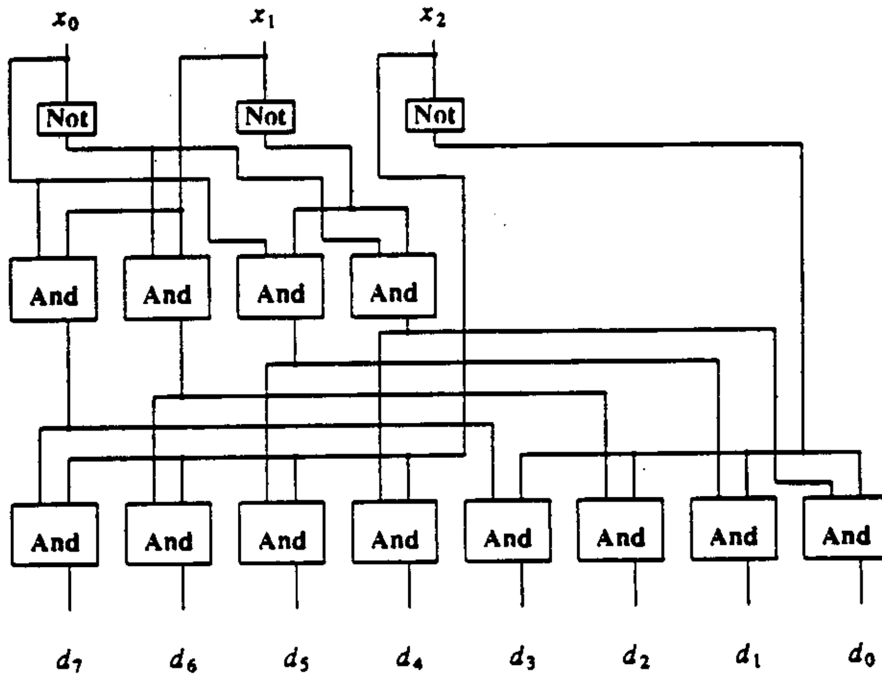


Figure 5.3 The *flipped* IF of Figure 5.2

right. The format of the intermediate form can easily be reversed (i.e., flipped to obtain the mirror image) allowing the compaction to be performed in the other direction (to the right). Figure 5.3 is the sketch obtain by *flipping* the intermediate form of the sketch in Figure 5.2.

Lahti's description of a decoder is a recursive switching function definition based on the primitives **andg** and **notg**. Figures 5.4 and 5.5 contain the sketches of decoders whose specifications are in terms of lower level primitives. These designs correspond to those of [Mead80]. Their FP definitions are more complicated than Lahti's.

```
# Nor-decoder ####################
{NorDecode &PU@Repeat@[1,split@2]@Setup}
{Repeat (null@1→concat@2;Repeat@Stage)}
```

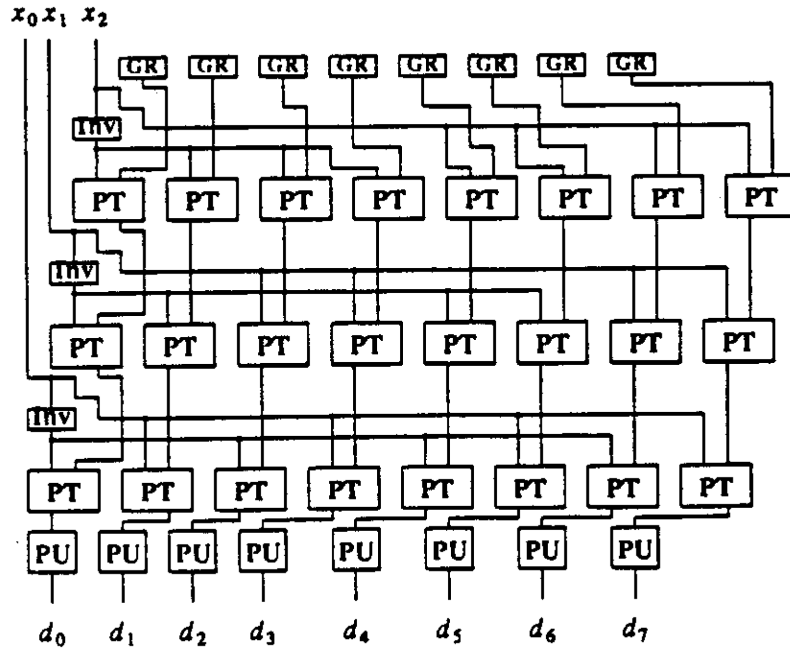


Figure 5.4 Nor Decoder

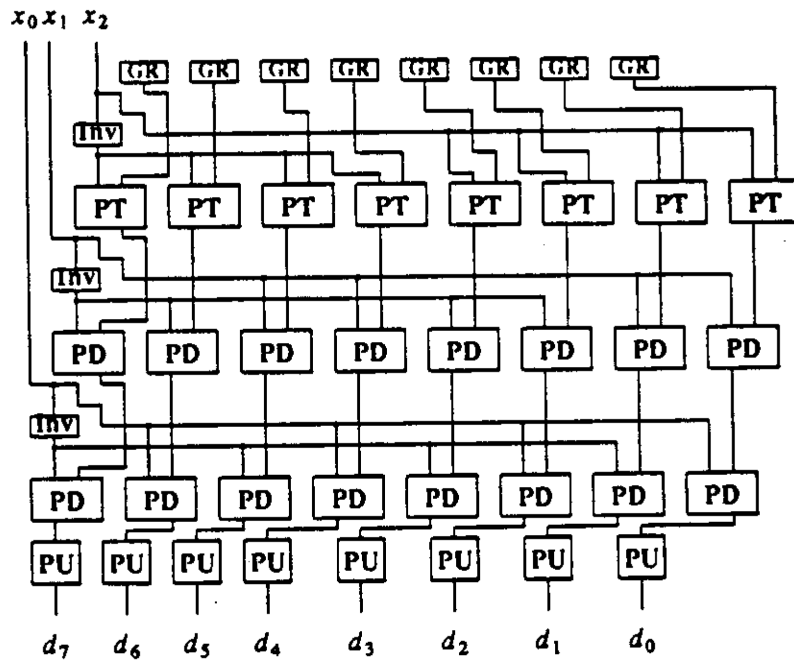


Figure 5.5 Nand Decoder

```
{Stage [tlr@1,concat@&(split@&PT)@Odistl
                    @[Inv@last@1,Edistl@[last@1,2]]]}

# Nand-decoder #####################

{NandDecode &PU@RepeatPD@Stage@[1,split@2]@Setup}

{RepeatPD (null@1→concat@2;RepeatPD@StagePD)}

{StagePD [tlr@1,concat@&(split@&PD)@Odistl
                    @[Inv@last@1,Edistl@[last@1,2]]]}

# Functions used by both decoders ################

{Edistl 1@[concat@&[1@2,distl@[1,2@2]]@distl@[1,pair@2]]}

{Odistl 1@[concat@&[distl@[1,1@2],2@2]@distl@[1,pair@2]]}

{Setup [id,&GR@Expand@[id,[tl@[id]]]]}

{Expand (eql@[length@1,%0]→2;Expand@[tl@1,concat@[2,2]])}

{PT org@[notg@1,2]}

{PD andg@[notg@1,2]}

{PU id}

{Inv notg}

{GR %0}
```

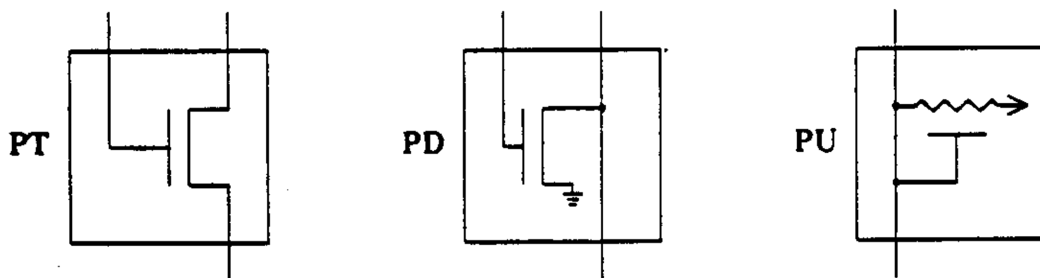


Figure 5.6 The primitives, Pass Transistor, Pull Down, and Pull Up

This specification is in terms of non-functional primitives, (i.e. pass transistors, pullups and pulldowns) which are pictured in Figure 5.6. With the knowledge of how

these elements are intended to function in this circuit (their direction of flow), they can be represented by FP functions. Of course, the designer is responsible for insuring that these elements do in fact correspond to their FP definitions in practice.

5.2 Carry-Save Array Multiplier

The following is an FP specification of a carry-save array multiplier. The specification is generic in that it will multiply any two bit vectors of length greater than 3. The algorithm consists of stages, each of which consumes one bit of the multiplier and performs a row-reduction using full adders on the column sums of the preceding stages and the multiplicand 'anded' by a bit of the multiplier. The output after these stages, consists of two bits per column for the n leftmost columns and a single bit for the $m-1$ rightmost columns; a carry-propagate adder is applied to these columns to obtain the final sum.

```
# Carry Save Array Multiplier

#          multiplier                  multiplicand
# input <<y_m y_m-1 ... y_2 y_1> <x_n x_n-1 ... x_2 x_1>> for m>3 and n>2.

# output <s_m+n s_m+n-1 ... s_2 s_1>

# ******** the function *******
{Mult FinalAdd@Csmult}

{Csmult CkStage@Stage3@Stage2@Stage1}

{Stage1 [tlr@1,concat@&[1,andg]@pair@concat
                    @[[1@2],concat@distl@[last@1,tl@2],[last@1]]]}

{Stage2 [1,concat@[[1,[andg]]@1@2,concat@&[1,[andg@[1,2],3]]@2@2],3]
        @[1,[1@2,&[2@2,1,1@2]@2@2],3]
        @[tlr@1,[[1@2,last@1],distl@[last@1,pair@tlr@tl@2]],[last@2]]}

{Stage3 Regroup@Csave1@Setup}

#check for last stage and repeat NormalStage
{CkStage (eql@[length@1,%1]→LastStage;CkStage@NormalStage)}
```

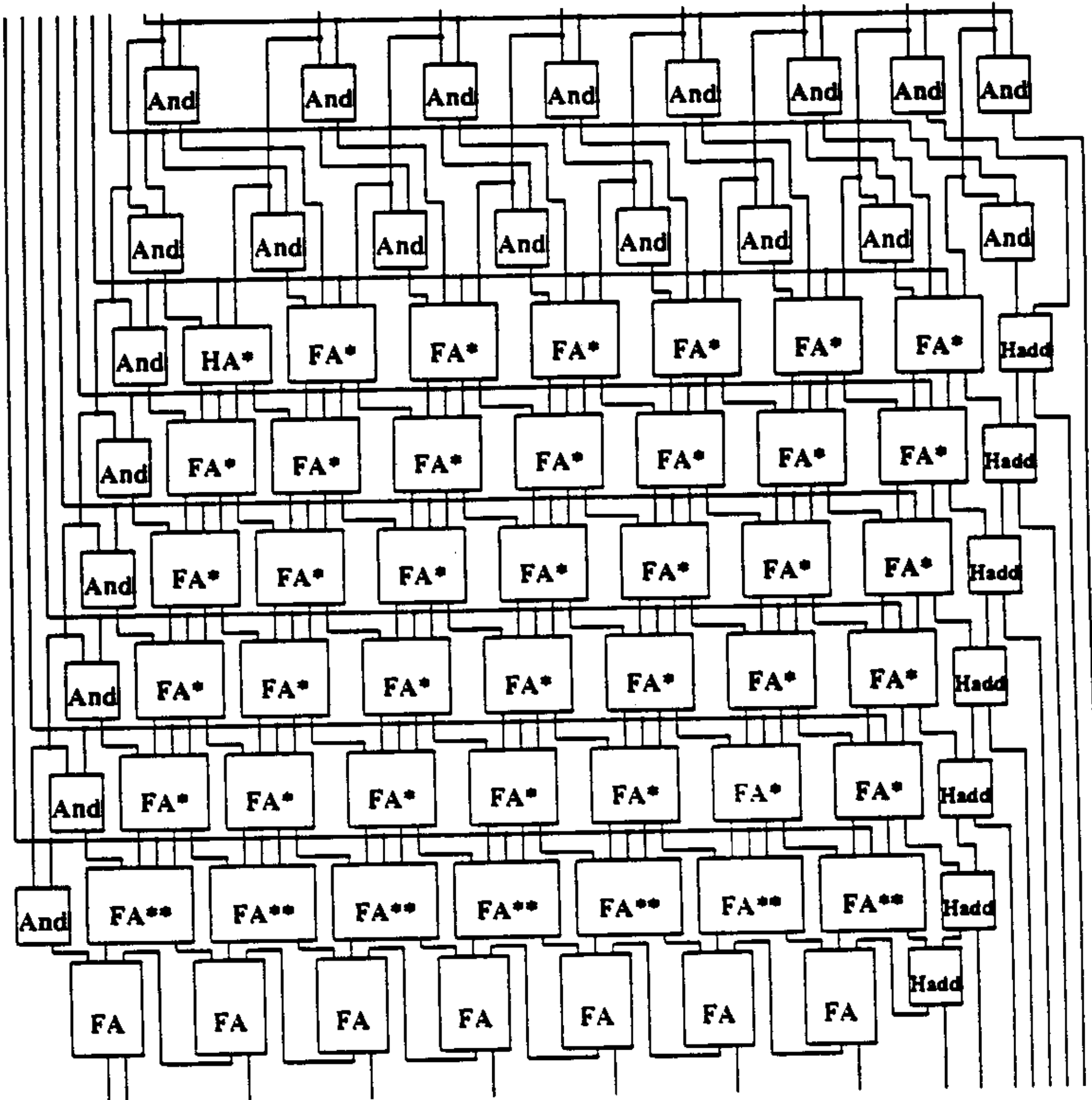


Figure 5.7 The sketch of the function **Mult** with the functions HA*, FA*, and FA** represented as components.

{NormalStage Regroup@Csave@Setup}

{LastStage LastRegroup@LastCsave@Setup}

{Setup [tlr@1,
[[1@2,last@1],&[1@2,[1,2@2]]@distl@[last@1,pair@tlr@tl@2]],
apndl@[last@2,3]]}

```
{Csave [1,concat@[[1,andg]@1,&OP2@2]@2,apndl@[Hadd@1,tl]@3]}

{Csave1 [1,concat@[[1,andg]@1,[OP1@1@2],&OP2@tl@2]@2,
        apndl@[Hadd@1,tl]@3]}

{LastCsave [concat@[[andg@1],&LOP2@2]@2,apndl@[Hadd@1,tl]@3]}

{LastRegroup concat@[pair@tlr@apndl@[1,concat@tl]@1,
                [apndl@[[2@last@1,1@1@2],apndl@[2@1@2,tl@2]]]]}

{Regroup [1,apndr@[tlr@2,[last@2,1@1@3]],
               apndl@[2@1@3,tl@3]]@[1,Regp@2,3]}

{Regp (eql@[%2,length]→id;
        concat@[[1,[2,1@1@3]],Regp@concat@[[2@1@3,2@3],tl@tl@tl]])}

{FinalAdd concat@[seq(CFA)@concat@[tlr,[[1],2]@1@last],tl@last]}

{CFA (eql@[%1,length@1]→Hadd@[1@1,2];FA)}

# FA* = OP2 : <<a b> <y x>> → <<c x> s> where 2c + s = (a + b + yx)
{OP2 [[org@[1,1@2],3],2@2]
      @[1@1,Hadd@[2@1,2],3]
      @[Hadd@1,andg@2,2@2]}

# HA* = OP1 : <<a> <y x>> → <<c x> s> where 2c + s = (a + yx)
{OP1 [[1@1,2],2@1]@[Hadd@[1@1,andg@2],2@2]}

# FA** = LOP2 : <<a b> <y x>> → <c s> where 2c + s = (a + b + yx)
{LOP2 [org@[1,1@2],2@2]@[1@1,Hadd@[2@1,2]]@[Hadd@1,andg@2]}

# FA : <<a b> c> → <c s> where 2c + s = (a + b + c)
{FA [org@[1,1@2],2@2]@[1@1,Hadd@[2@1,2]]@[Hadd@1,2]}

# Hadd : <<a b> → <c s> where 2c + s = (a + b)
{Hadd [andg,xorg]}
```

Figure 5.7 is the sketch obtained of the function **Mult** with the functions **HA***, **FA***, **FA****, and **Hadd** represented as primitives. The sketches of the functions **HA***, **FA***, and **FA**** are given in Figure 5.8. Figure 5.9 contains a sketch of the same FP expression with the functions **HA***, **FA*** and **FA**** evaluated. The effect is to 'splice' their definitions into the sketch. Notice that the boundaries of the primitives **HA***, **FA*** and **FA**** are not respected and their geometry is not always the same; it

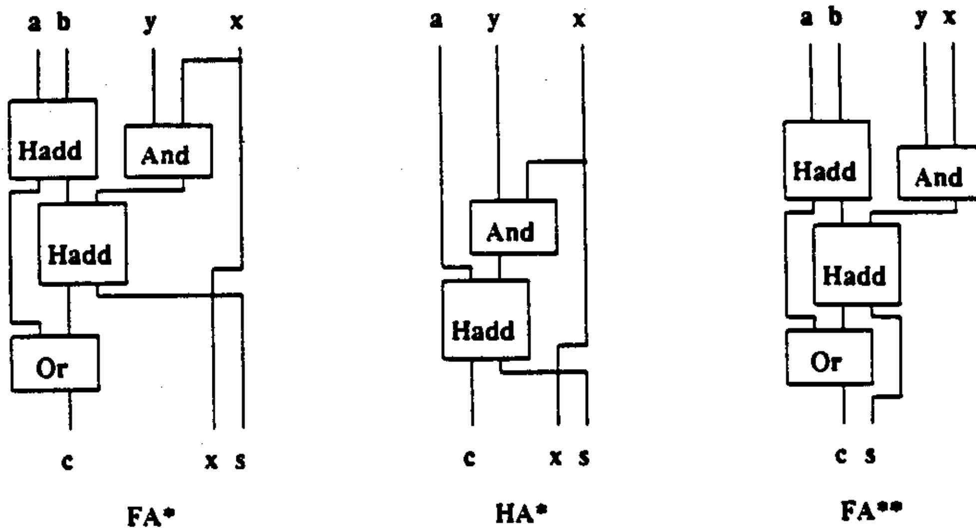


Figure 5.8 The functions **HA***, **FA*** and **FA****.

may adapt to its environment.

The FP specification of this algorithm is one of the most complicated. Each of the first three stages, the last stage and the other stages have different specifications since they are all slightly different. The difficulty in writing FP functions is often in determining the exact structure of the object being passed from one function to the next, particularly when functions are nested several times within **Constructs**. The exact structure of the object being passed must be known in order to write the FP function. It is useful to annotate FP specifications with the structures expected and produced by its functions.

This design was realized as a layout. The first sketch in which the functions

HA*, **FA***, and **FA**** are not expanded was selected, since the routing of power and ground to the components in this case is straight forward. Power and ground are connected to the components by two interleaved combs whose teeth run horizontally

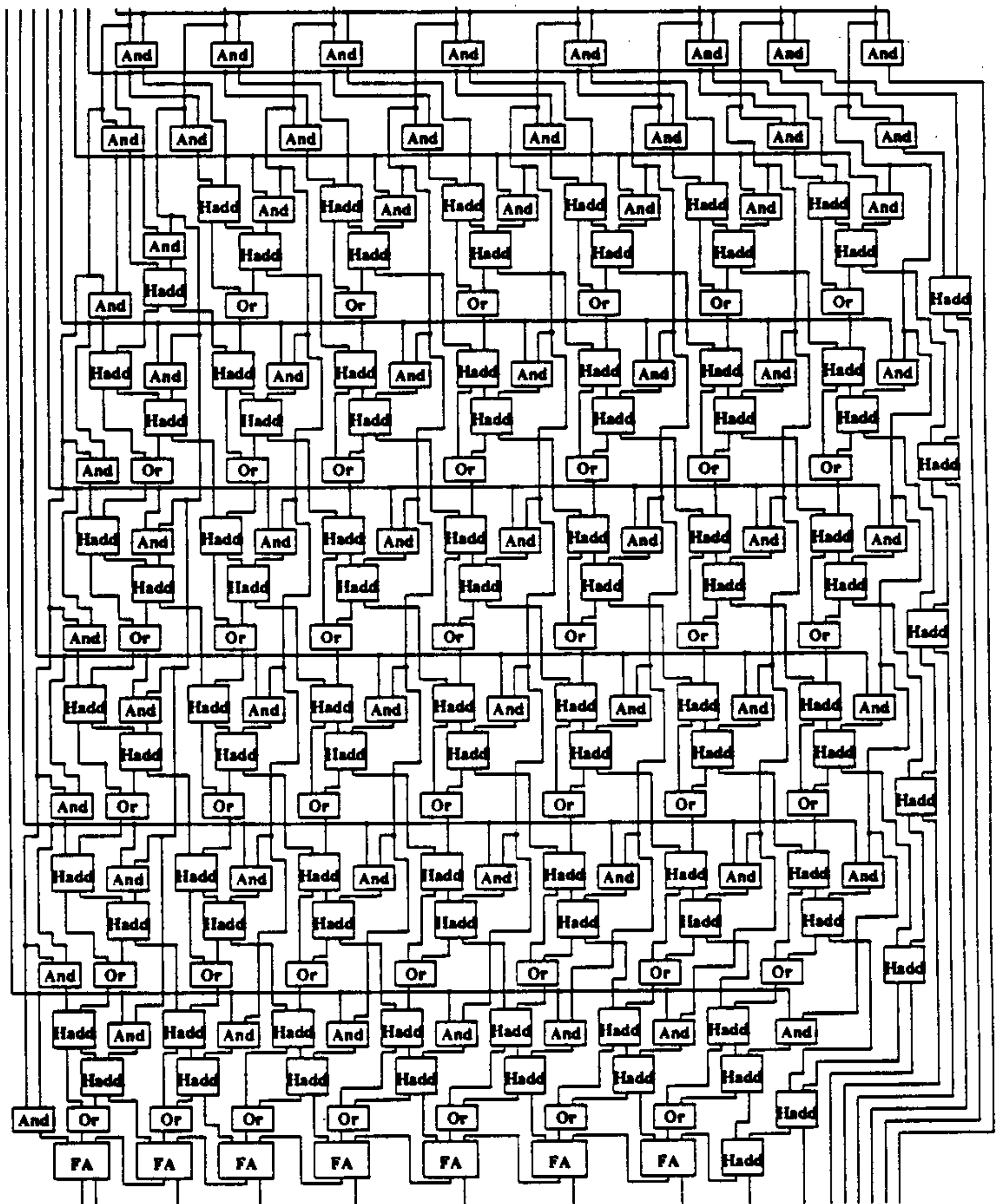


Figure 5.9 Mult with the functions HA*, FA*, FA** expanded.

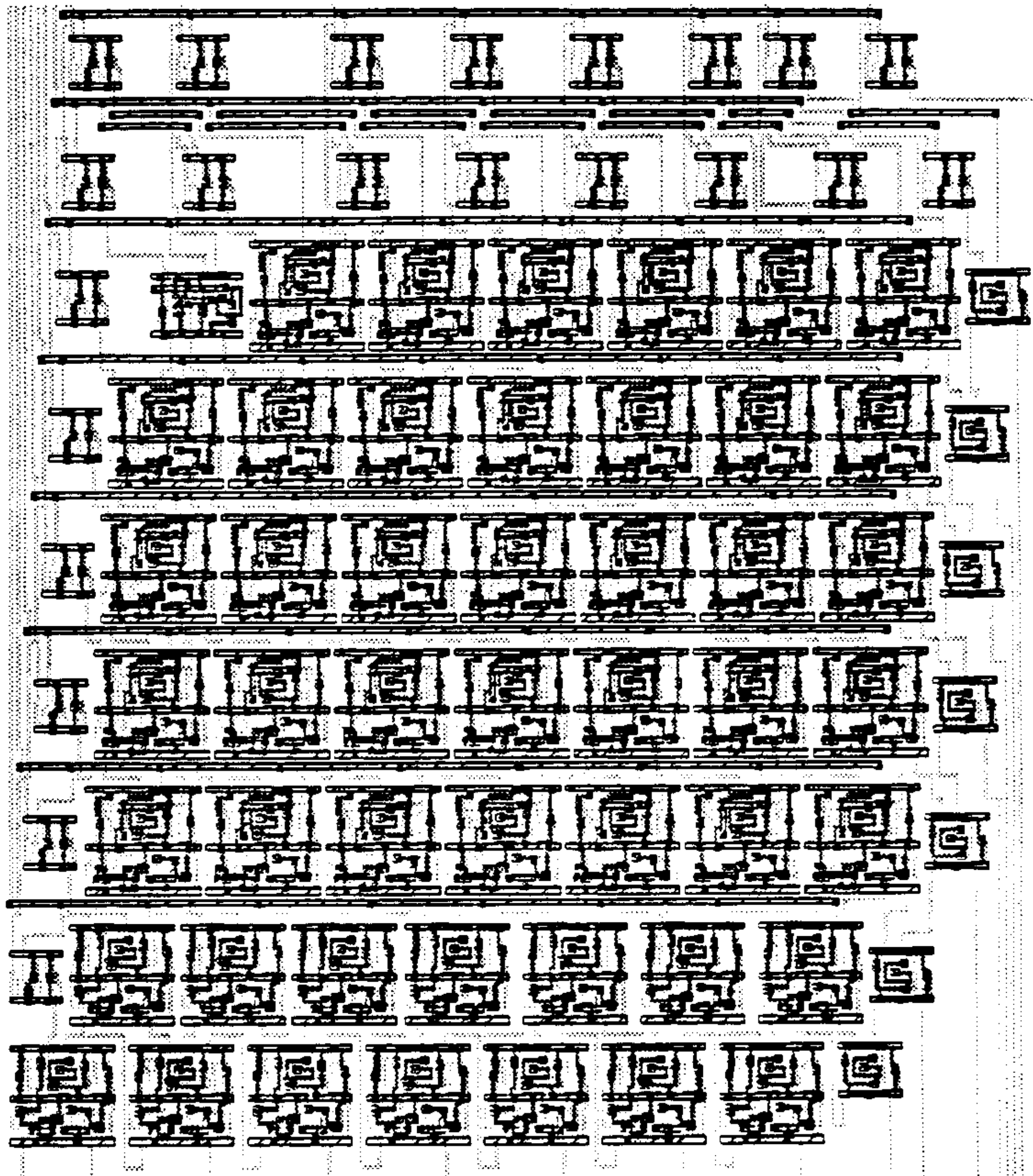


Figure 5.10 The layout of Mult

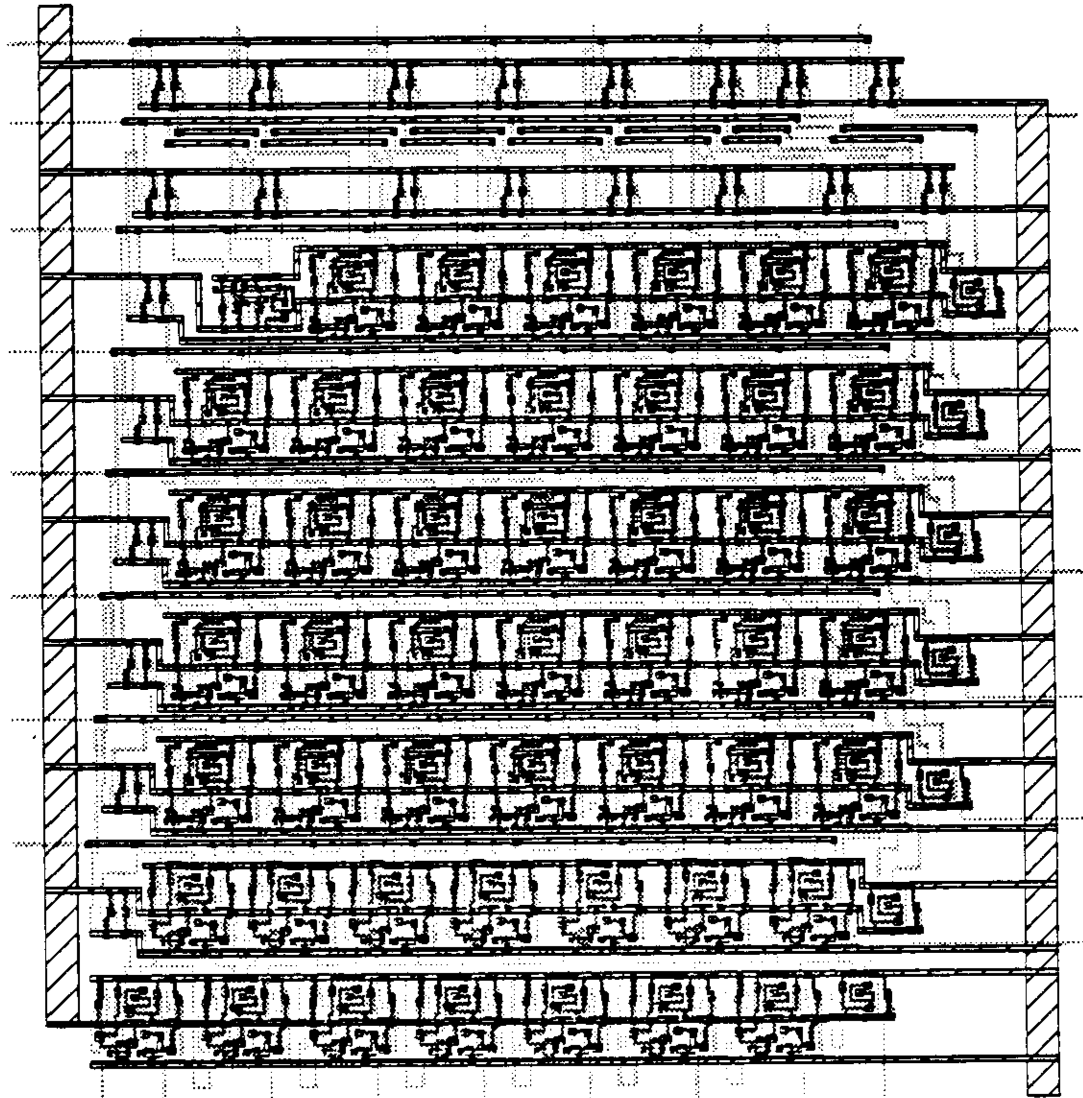


Figure 5.11 The completed layout of **Mult**

along the rows. Since power and ground run horizontally in metal, we use metal only for the horizontal wires of crossings; the remaining wires are routed in polysilicon. To obtain the layout in Figure 5.10, each of the primitives, **And**, **FA***, **FA****, **FA**, **HA*** and **Hadd** were designed with their inputs and outputs in polysilicon. The dimensions of these components and the exact locations of their inputs and outputs were provided to the compactor in the format described in Section 4.5. The compactor generated the positions of the components and the metal and polysilicon routing and contacts for their interconnections. This layout, which is given in Figure 5.10, is obtained by including the layouts of the components at the coordinates given by the compactor. To finish the layout the power and ground was added using a graphics editor. In addition, the inputs and outputs were redirected from the top and bottom respectively, to the sides. This final layout is in Figure 5.11.

5.3 Carry Chain Adder

The FP specification of a carry chain adder [Bren80] is considered. The specification is generic in that it adds two bit vectors of length 2^n for $n>0$. The input consists of the 2^n pairs of bits to be added with the leftmost pair, containing the least significant bits, $((a_1,b_1),(a_2,b_2),(a_3,b_3), \ldots, (a_{2^n},b_{2^n}))$.
For $1 \le i \le j \le 2^n$,

$$P_{i,j} = \begin{cases} 1 & \text{if a carry into column i would propagate as a carry out of column j} \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad G_{i,j} = \begin{cases} 1 & \text{if adding columns i through j causes a carry out of column j} \\ 0 & \text{otherwise} \end{cases}$$

The computation is performed by computing the carries for each column, $G_{1,i}$ and

then obtaining the sum bit using,

$$s_i = G_{1,i-1} \oplus P_{i,i} \text{ for } 1 < i \leq 2^n, \; s_1 = P_{1,1}, \text{ and } s_{2^n+1} = G_{1,2^n} \tag{5.1}$$

$P_{1,i}$ and $G_{1,i}$ are computed for each i by using the following identities, implemented by the function **PG**.

$$\text{For } i \leq j < h, \; P_{i,h} = P_{i,j} P_{j+1,h} \text{ and } P_{i,h} = (G_{i,j} P_{j+1,h}) + G_{j+1,h}$$

The initial $P_{i,i}$ and $G_{i,i}$ are computed by the function **PG1**.

$$P_{i,i} = a_i \oplus b_i, \quad G_{i,i} = a_i b_i \quad .$$

The computation of $P_{1,i}$ and $G_{1,i}$ is achieved in two steps by the function **GetCarries**. The following is the specification of **GetCarries**.

```
# input = ((a0,b0),(a1,b1),(a2,b2),...,(a2**n - 1,b2**n - 1))
{GetCarries SecondHalf@split@1@FirstHalf@&PG1}

{FirstHalf (eql@[length,%1]→id;FirstHalf@&Stage1@pair)}

{Stage1 concat@[&D@1,&D@tlr@2,[PG@[last@1,last@2]]]}

{SecondHalf (eql@[length,%1]@1→Done;SecondHalf@concat@
        [split@&D@1,Stage2@tl]@apndr@[concat@&[id,last]@tlr,last])}

{Stage2 concat@&([apndr@[&D@tlr@1@2,PG@[1,last@1@2]],&D@2@2]
                @[1,split@2])@pair}

{D id}

{Done id}
```

First **GetCarries** computes $(P_{i,i},G_{i,i})$ by applying **PG1** to the two bits in each column and then it applies **FirstHalf**. **FirstHalf** computes $(P_{i-2^k+1,i},G_{i-2^k+1,i})$ for each column $i=(2m+1)2^k$ where m is an integer. This is accomplished by arranging each column (i.e. its pair (P,G)) in a group of its own and then recursively applying the function **Stage1** to pairs of groups until only a single group remains. **Stage1** combines a pair of groups computing a new (P,G) for the last column of the second group by applying **PG** to the last columns of the two groups; the pair of groups is then concatenated to form one group. All other columns are unchanged; the function **D** which is given the definition **id** is applied to them. When all columns are in a single group **GetCarries** applies the function **SecondHalf** to compute the final (P,G)'s. **SecondHalf** is also recursive, terminating when each column is in a group by itself. At each step the final (P,G)'s of decreasing multiples of powers of 2 are computed. Assume that in the previous step $P_{1,i}$ and $G_{1,i}$ have been computed for each column $i=m2^k$. In the next step to compute the (P,G)'s of columns which are multiples of 2^{k-1}, it is necessary only to compute new (P,G)'s for columns $i=(2m+1)2^{k-1}=m2^k+2^{k-1}$, the odd multiples of 2^{k-1}. The current (P,G) in column i is $(P_{i-2^{k-1}+1,i},G_{i-2^{k-1}+1,i})$. $(P_{1,i},G_{1,i})$ can be obtained by applying **PG** to the current (P,G) and $(P_{1,m2^k},G_{1,m2^k})$. Initially the columns are divided into two groups and since **FirstHalf** computed the final (P,G)'s for powers of 2, the last column (a multiple of 2^{n-1}) has its final value. At each step **SecondHalf** duplicates the last column from each group and then applies **Stage2** after removing the first group. **Stage2** takes each group, splits it into two and computes new (P,G)'s for the last column in the left group of each new pair using the duplicated column immediately to the left of the group. The first group is then appended to the result of **Stage2**.

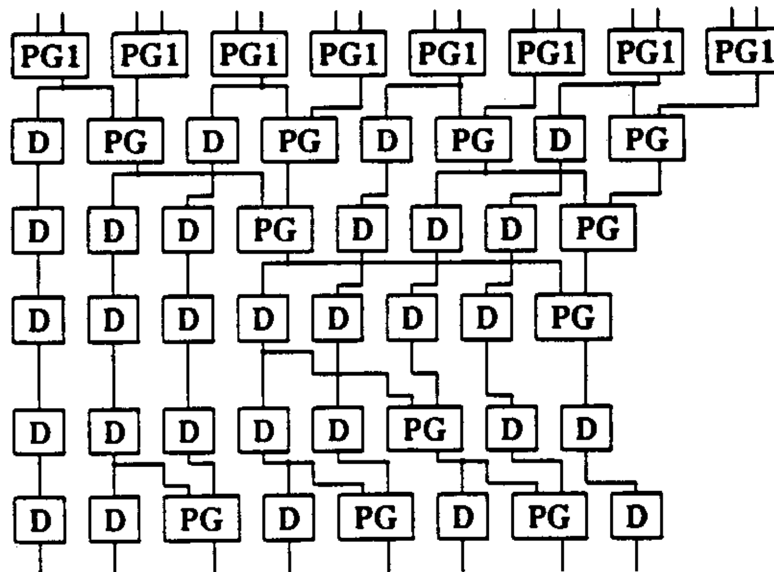


Figure 5.12 The sketch of **GetCarries** with each (P,G) as a wire

Figure 5.12 is the sketch of **GetCarries** in which the pair (P,G) for each column is represented by a single wire. This is accomplished by directing the interpreter to draw **PG1**, **PG** and **D** as boxes and by giving **PG1** and **PG** symbolic definitions.

(define-symbolic **PG1** input=(a b) output = (c))

(define-symbolic **PG** input=(a b) output = c)

(drawbox **PG1** label=PG1 ht=2)

(drawbox **PG** label=PG ht=2)

(drawbox **D** label=D ht=2)

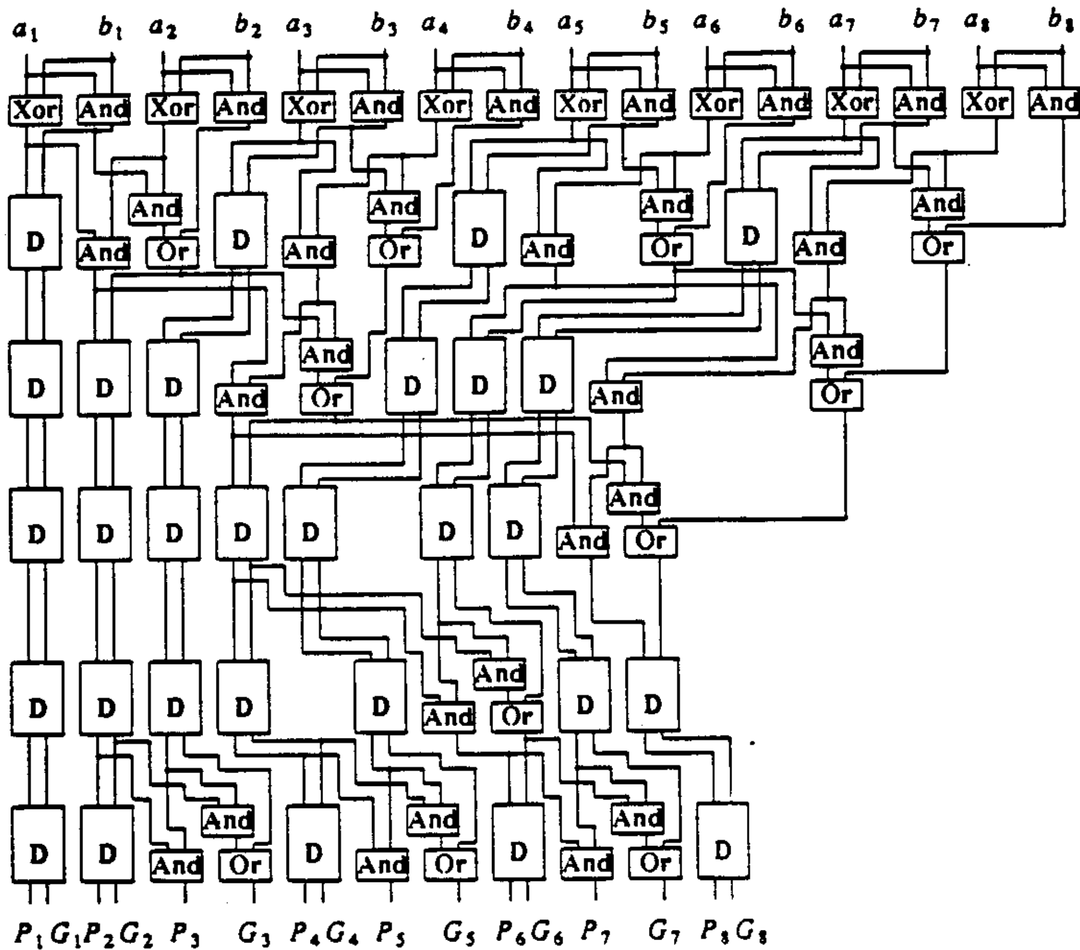


Figure 5.13 The sketch of GetCarries with PG and PG1 expanded

Figure 5.13 is the sketch of GetCarries with each wire corresponding to a bit this time and with the specifications of the functions PG and PG1 'filled in.' The specifications of PG1 and PG are,

{PG1 [[xorg,andg]]}

{PG [andg@[1@1,1@2],org@[andg@[2@1,1@2],2@2]]}

By performing the operations on the computation tree described in Section 3.8 to merge connected routing regions, the routing can be improved. Figure 5.14 is the result of these operations. Note that one less horizontal track is used in between each stage.

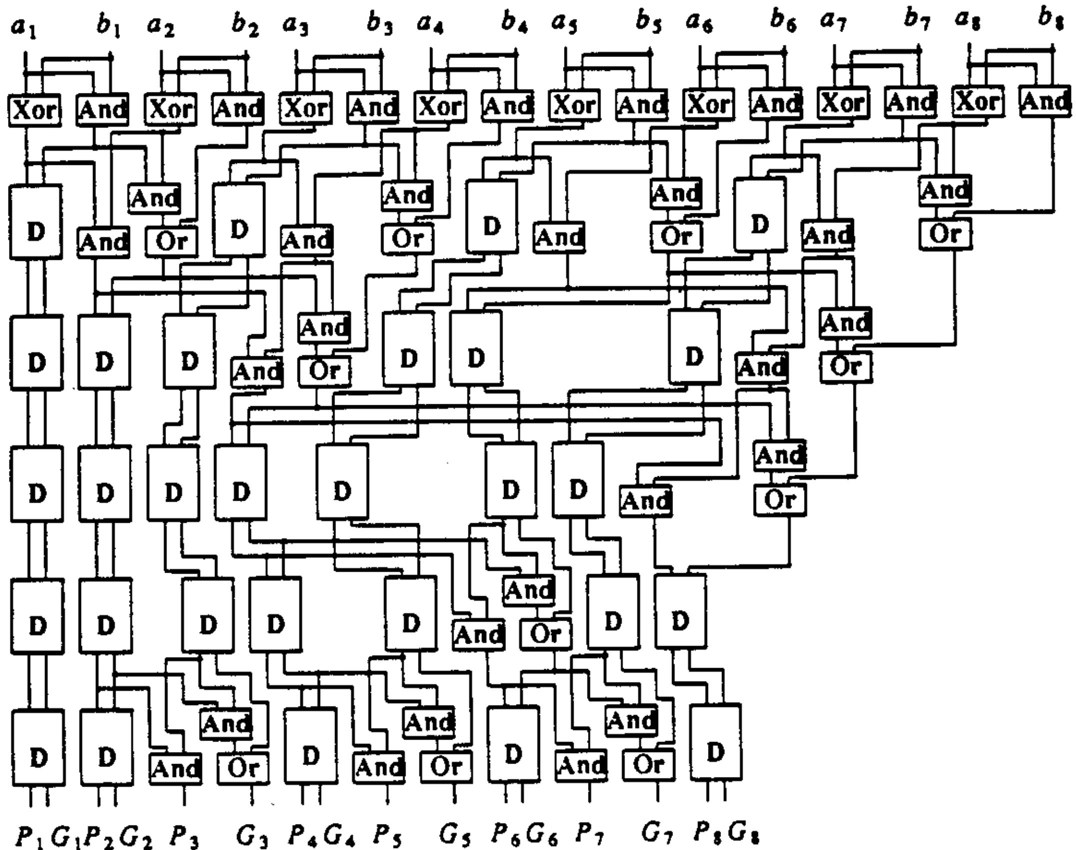


Figure 5.14 The sketch of maximal planar circuit of **GetCarries**

To compute the final sum we will need only the G_i's. The definition of **Done** is changed to retain only the G of each column. Notice that the layout interpreter generates only those wires and boxes which have paths to an output. The function **Done** in the previous specification is be modified as follows.

{Done &(2@1)}

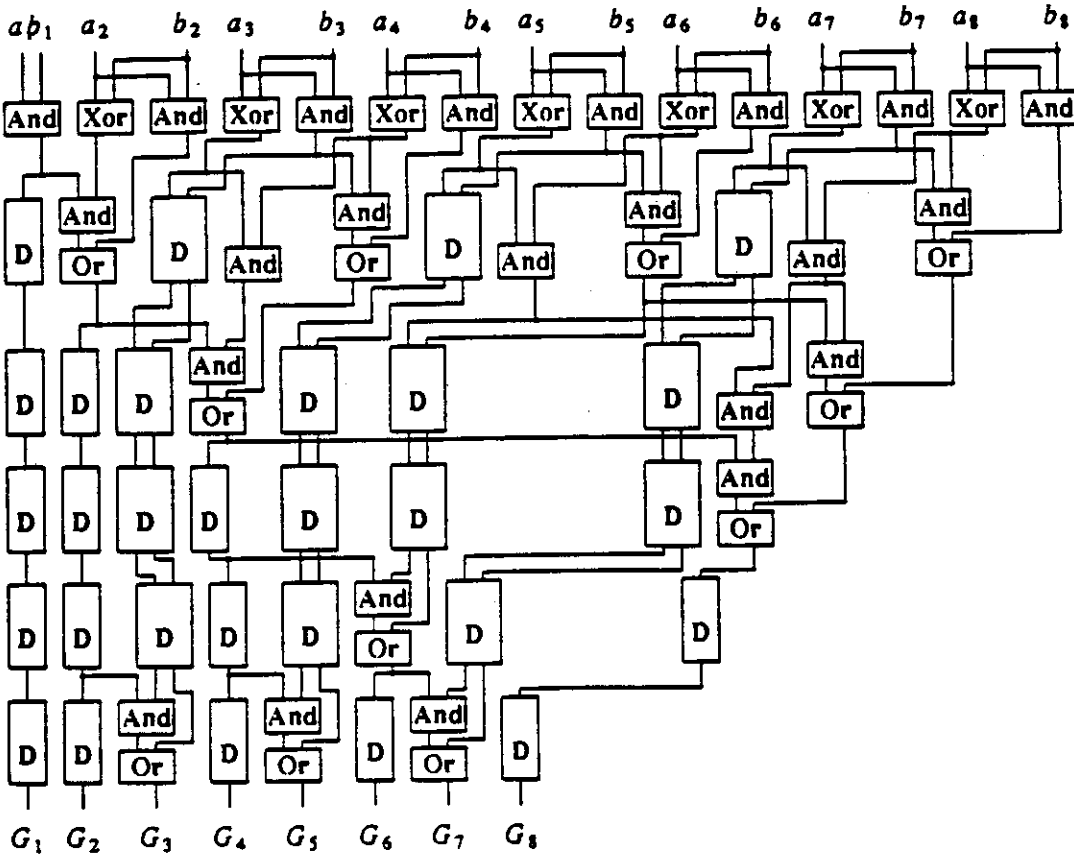


Figure 5.15 The pruning of the planar circuit of Figure 5.14

This gives the sketch in Figure 5.15. The specification of this function is greatly simplified by having unnecessary structure pruned away rather than having to modify the specification to avoid generating it.

To obtain the final sum by (5.1) it is necessary to combine the first P in each column, $P_{i,i}$ with the G of its left neighbor $G_{1,i-1}$. One way of doing this would be to duplicate each $P_{i,i}$ generated by **PG1**, route them along the side and then merge them back into the columns to compute the final sum as in Figure 5.16. The additional area required for routing makes this an unattractive alternative. A better design would be to route the $P_{i,i}$ along with the (P,G) down its own column. This extension is easily handled in FP by modifying the function **PG** so that it duplicates

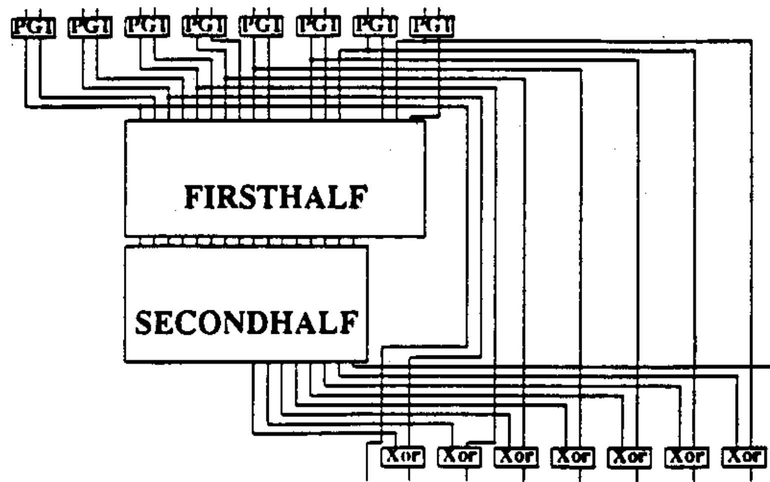


Figure 5.16 An inefficient design

its first argument if it receives only two arguments in a column and simply passes on the extra argument otherwise. The specification is modified as follows.

```
{Add concat@[[1],&xorg@pair@tl@tlr,[last]]
        @concat@apndl@[1@1,&([1,3]@1)@tl]@GetCarries}

{PG apndl@[D@1@2,(null@tl@tl@2→OldPG;OldPG@[1,tl@2])]
        @(null@tl@tl@1→id;[tl@1,2])}

{OldPG [andg@[1@1,1@2],org@[andg@[2@1,1@2],2@2]]}

{Done id}
```

The function **Add** applies **GetCarries** and then handles the columns according to (5.1) to obtain the final sum bits. Figure 5.17 is the sketch of **Add**.

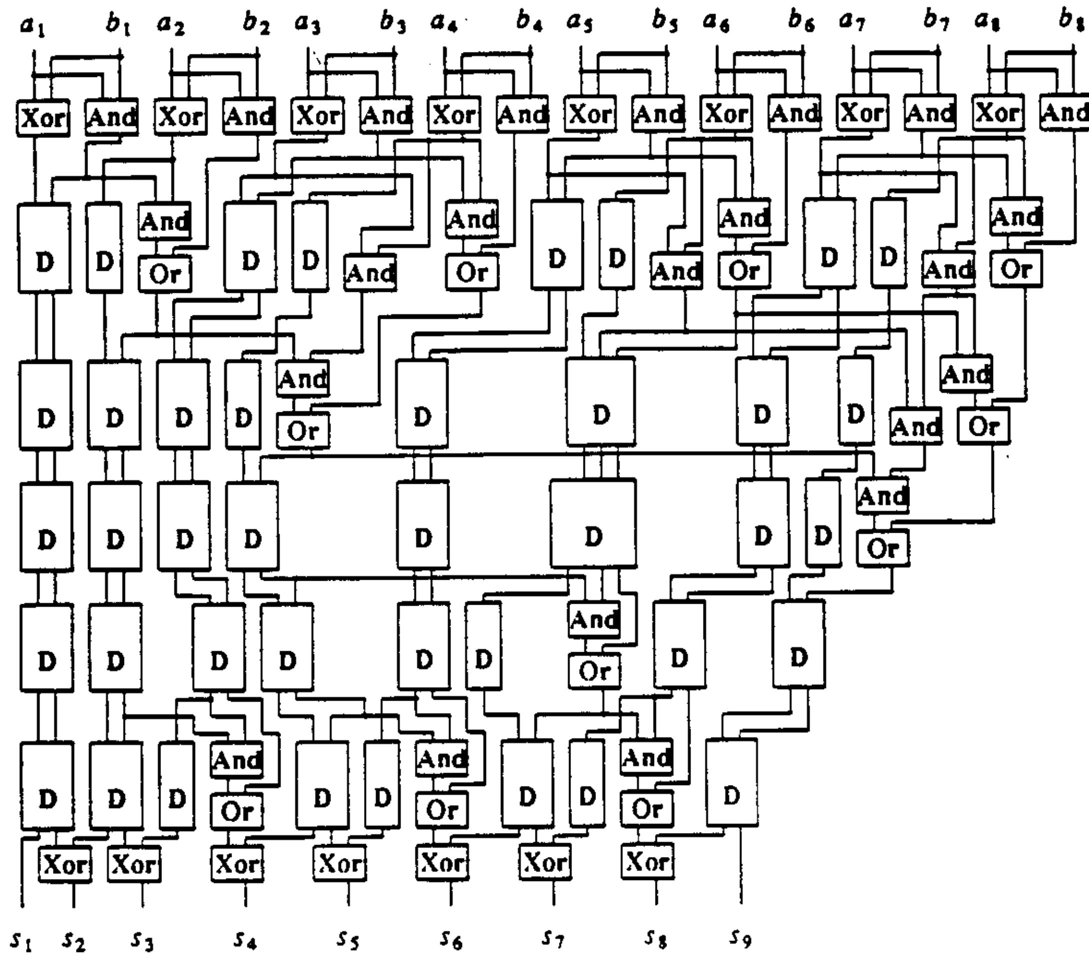


Figure 5.17 The sketch of Add

4.4 Tally

The tally circuit counts the number of 1's in its input. The ith output is 1 if there are exactly i inputs which are 1. The definition is recursive; it computes the tally of n-1 inputs and then considers the nth input, adding a 0 to either side of the

previous result, according to the value of this n^{th} input.

```
# main function
{Tally (eql@[length,%1]→One;More)}
{One [id,Inv]@1}
{More &SEL@distl@[[Inv,id]@1,pair@concat
              @[[%0],concat@&[id,id]@Tally@tl,[%0]]]}
{SEL Wor@&SW@trans}
{Wor org}
{SW andg}
{Inv notg}
```

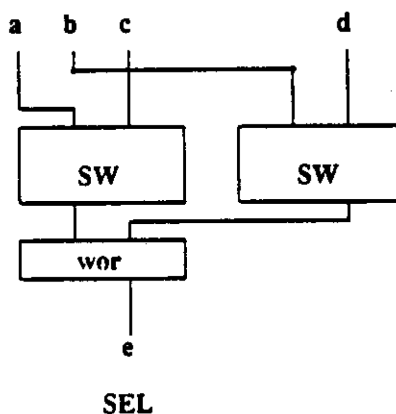


Figure 5.18 The function SEL.

As in the Decoder, a more compact placement can be obtained by compacting to the right instead of the left. However, this problem disappears when the sketch of the algorithm is obtained with lower level primitives in Figure 5.21. Again the primitives SW and Wor can be represented by gates in the FP definition.

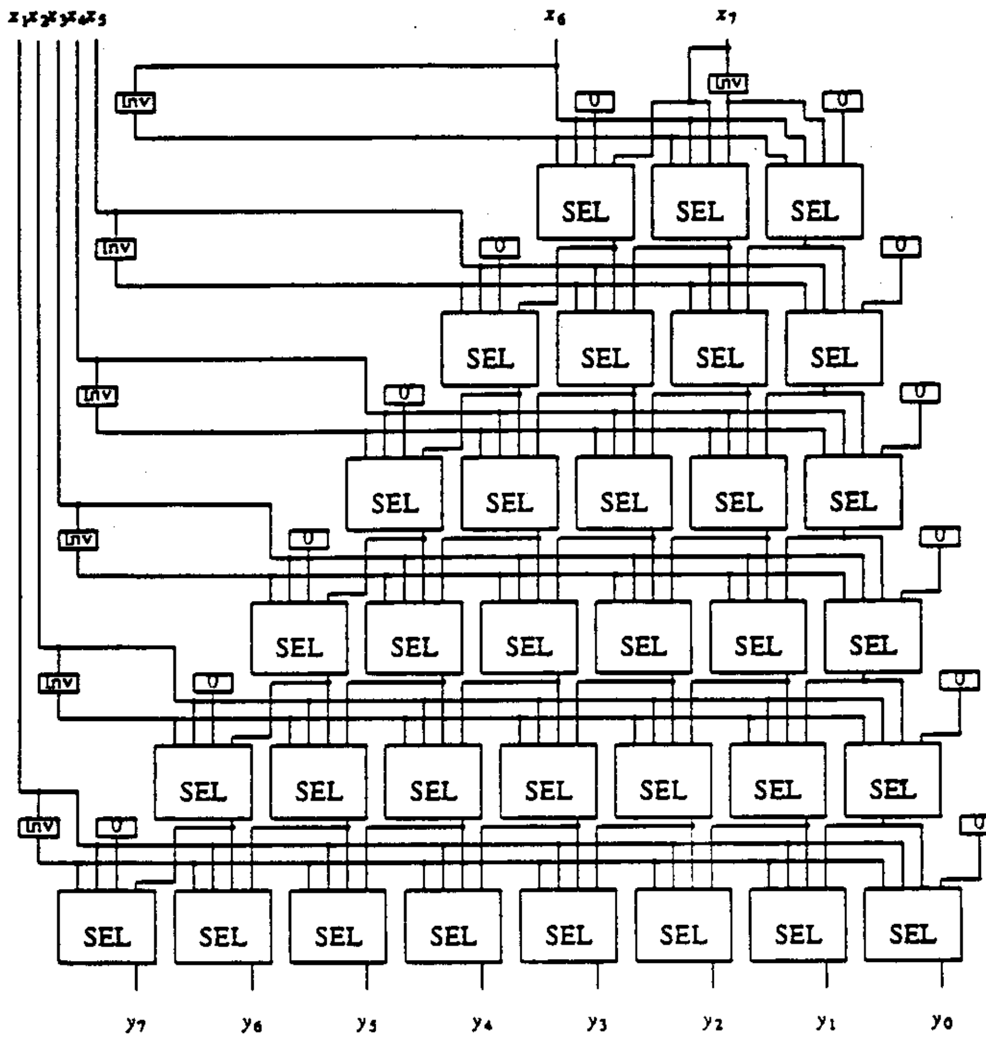


Figure 5.19 Tally circuit with **SEL** as a primitive.

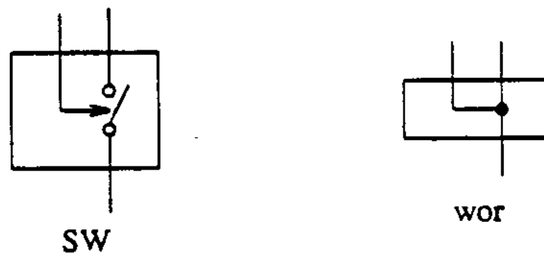


Figure 5.20 The functions SW and Wor.

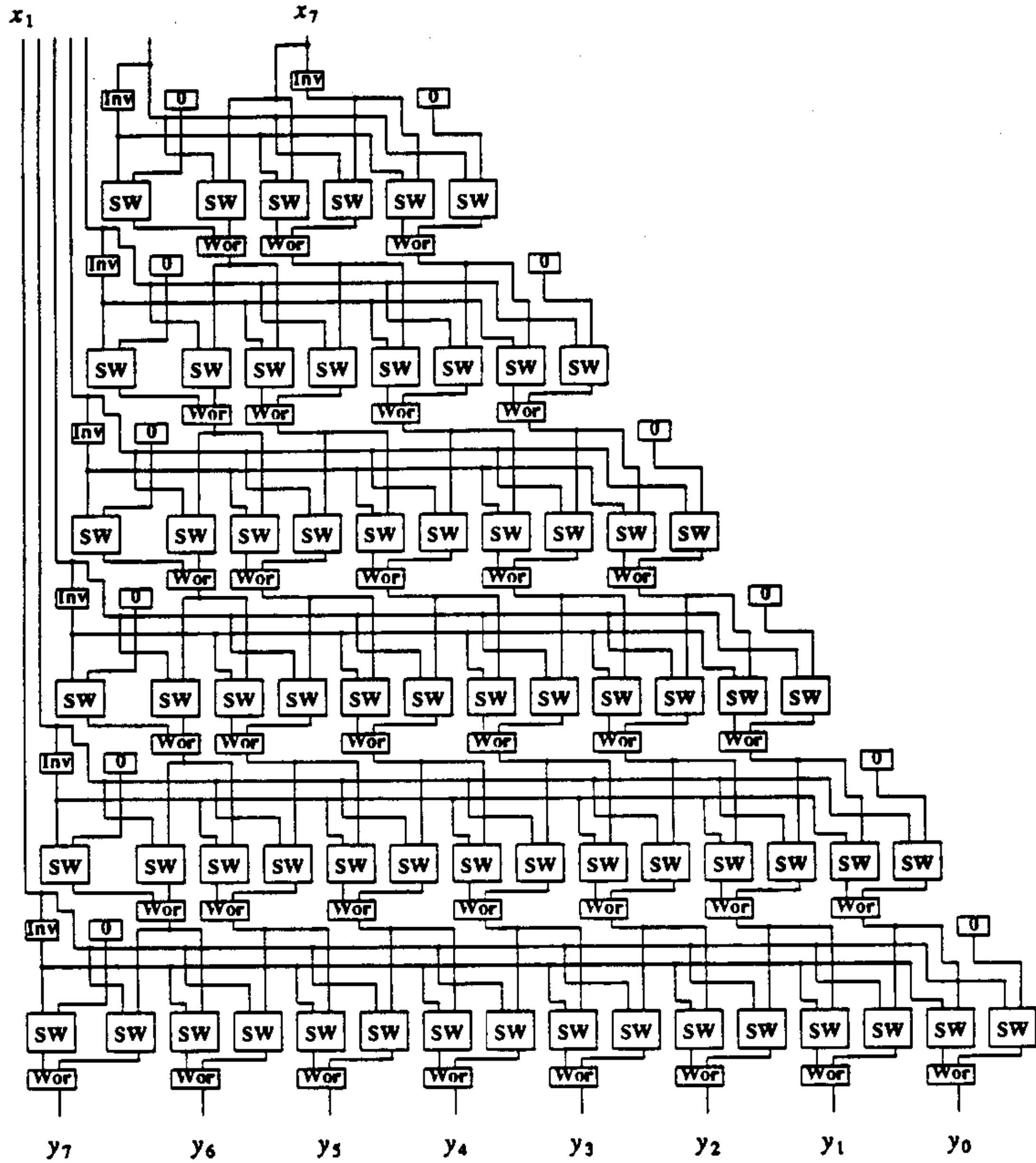


Figure 5.21 Tally circuit with SW and Wor as primitives.

4.5 FFT

This is an example of a higher level algorithm: an algorithm to compute a 2^n point FFT. It illustrates the ability to experiment with the different algorithms at a high level. Two algorithms are presented, one based on the Butterfly and Bit Reversal permutations and the other based on the shuffle permutation [Park80]. Figures 5.22 and 5.23 are the sketches of the two algorithms with the permutations represented by boxes. The two functions are written for any 2^n point input.

```
# 2^n point FFT
#
# input : (2^n complex numbers)
#(z0 z1 z2 z3 z4 z5 z6 z7 z8 z9 z10 z11 z12 z13 z14 z15)

# or 2^n pairs of real numbers
#((x0y0)(x1y1)(x2y2)(x3y3)(x4y4)(x5y5)(x6y6)(x7y7)
#(x8y8)(x9y9)(x10y10)(x11y11)(x12y12)(x13y13)(x14y14)(x15y15))
#
######################################################
# Traditional FFT Algorithm - Butterflies and Bit reversal
######################################################

{Fft Bitrv@FftStages}

{FftStages (eql@[length,%2] → W ; &FftStages@split@concat
                                    @Bfly@concat@&W@Bfly)}

{Bfly concat@[Shuffle@[1,3],Shuffle@[2,4]]@concat@&trans@split@pair}

{Bitrv (eql@[length,%2]→id;&Bitrv@trans@pair)}

######################################################
# Shuffle - Unshuffle Algorithm
######################################################

{Fft (End→Flatten;Fft@VunShuffle@Stage)}

{VunShuffle (Bottom → UnShuffle ; Recons@VunShuffle@concat)}

{UnShuffle concat@trans@pair}

{Recons (Bottom→ split; &Recons)}

{Stage (Bottom→ split@concat@&W@Shuffle@split;&Stage)}
```

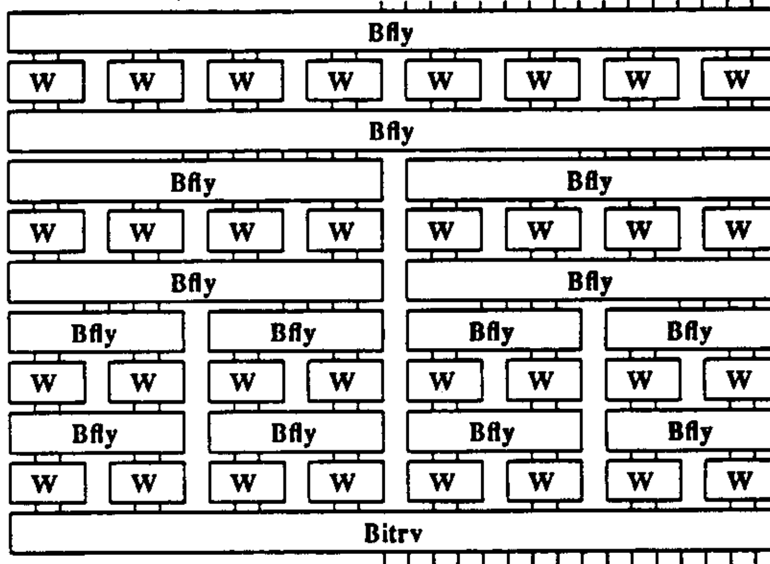


Figure 5.22 FFT with Butterfly and Bit-Reversal Permutations

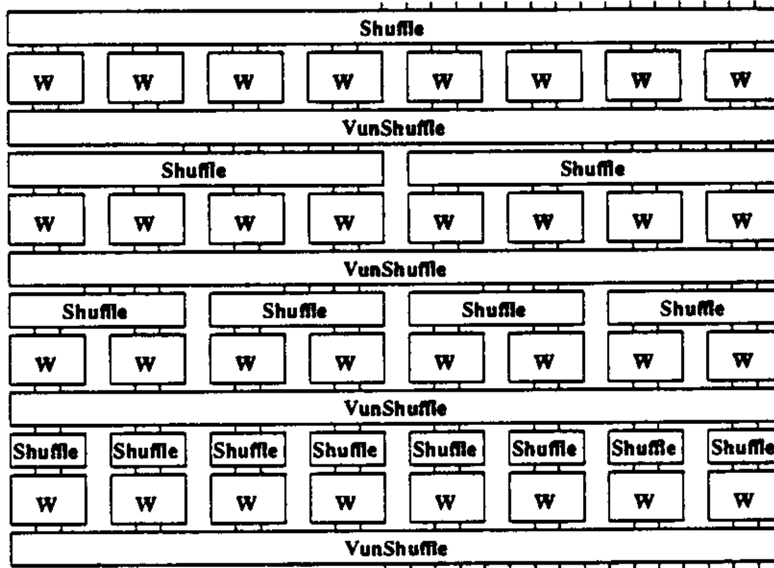


Figure 5.23 FFT with Shuffle Permutation

```
{End (Bottom→eql@[length,%1];End@1)}

# defined for complex numbers
{Bottom atom@1}

# defined for real numbers
{Bottom atom@1@1}

{Flatten (Bottom→ id; Flatten@concat)}

###### Definition of W

{W [Cadd,Csub]@[1,Cmul@[2,u0]]}

{u0 [u,u]}

{u %1}

{Cadd &+@Shuffle}

{Csub &-@Shuffle}

{Cmul [-@[*@[1@1,1@2],*@[2@1,2@2]],
      +@[*@[1@1,2@2],*@[2@1,1@2]]]}

{Shuffle trans}
```

The shuffle-unshuffle algorithm is more complicated and relies on the function **Bottom** to identify an actual point. This function would have to change depending on the representation level of the points, that is, whether an atom corresponds to a complex number or a pair of real numbers for example. The adjustment of **Bottom** is left to the programmer. Both algorithms for a 16-point FFT are displayed in Figures 5.22 and 5.23. Each wire represents a complex number and the W is marked as a primitive as well as the permutations.

In Figure 5.24 each wire is also a complex number, but the permutations, **Bfly** and **Bitrv**, are no longer primitives. Figure 5.25 contains the primitives for the sketches in Figures 5.22 and 5.23. The primitive **W** is represented with complex numbers as atoms. In Figure 5.26 each wire now represents a real number but the

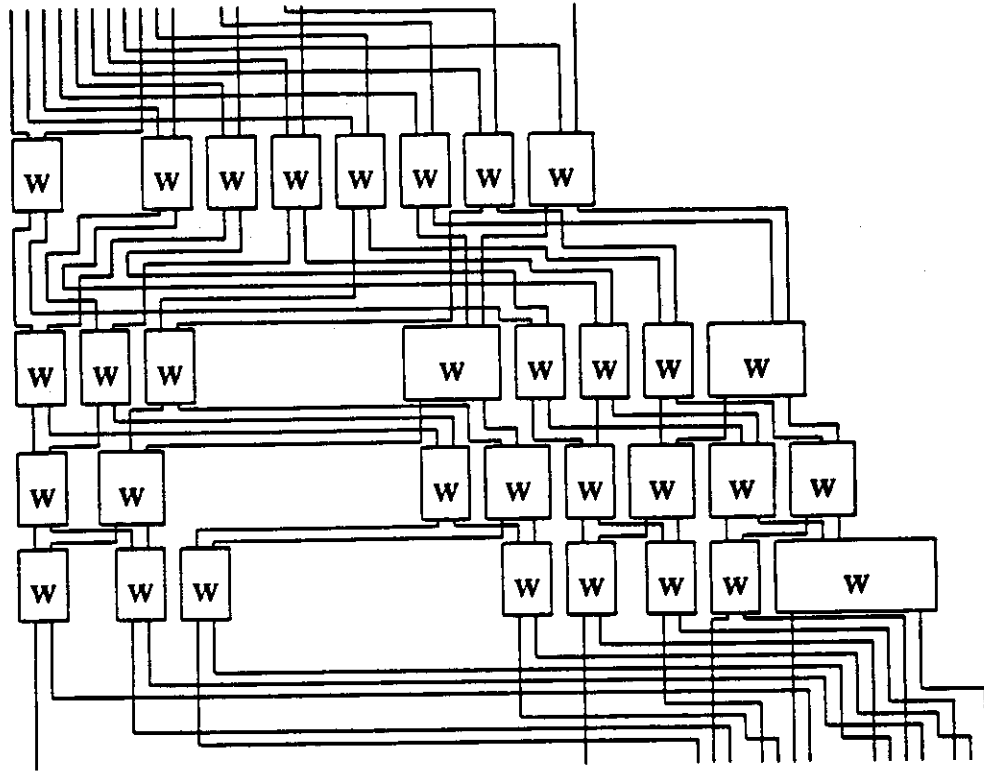


Figure 5.24 FFT algorithm of Figure 5.22 with only W as a primitive.

number of points has been reduced to 8 in order to fit the sketch on one page. The level of representation could be lowered even further until each wire corresponds to a bit. The same FP code would be used but each time functions at a lower level would be marked as primitives. By examining the sketches, it is apparent that the routing area required to implement the FFT in this manner is prohibitive.

The performance of the compaction algorithm is most visible in this example. The constraint graphs of Figures 5.22, 5.23 and 5.4 were inconsistent and required adjustment. For Figures 5.22 and 5.23, the compaction went through two phases. In the first phase, the boxes were enlarged to obtain a consistent constraint graph, and then the wires were fixed. However, after pulling the wires back to the right, it became possible to retract some of the boxes, so a second compaction was

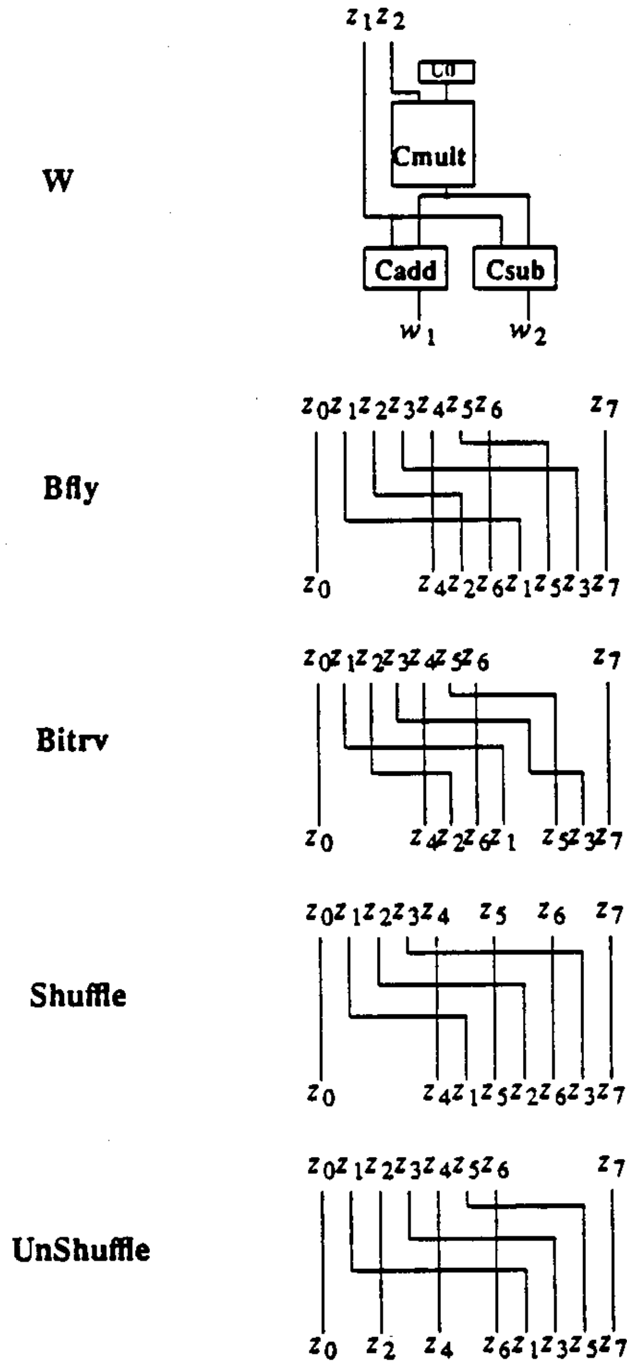


Figure 5.25 The primitives W, Bfly, Bitrv, Shuffle and UnShuffle.

performed, (on a constraint graph which was now consistent). The sketch for Figure 5.24 is not as "nice"; the routing of the permutations forces this inefficient placement. This problem is somewhat alleviated in Figure 5.26 since the lower level primitives place less constraints on the routing.

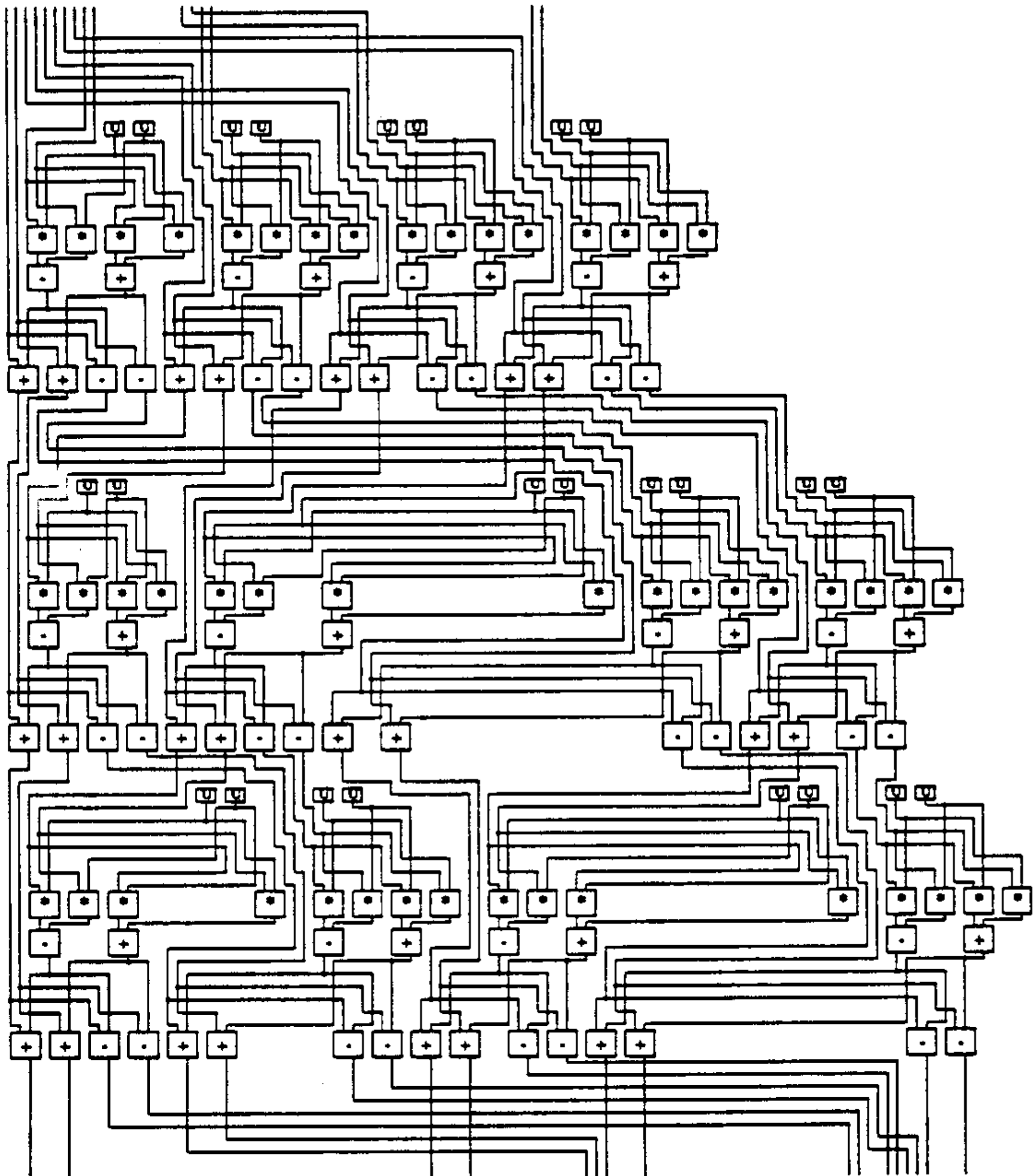


Figure 5.26 8-point Butterfly-FFT algorithm with the primitive W expanded.

5.6 Inner Product

This example from [Pate85] illustrates how to transform a space implementation of an algorithm into a time implementation by applying transformations to the FP specification. The following FP function computes the inner product of two vectors of numbers.

{IP !*@&+@trans}

The algorithm consists of three steps. Numbers from the two vectors are first paired by the function **trans**, each pair is then multiplied, and the results are added up. Figure 5.27 shows the space implementation of this algorithm with two vectors of length four. A wire here represents a number.

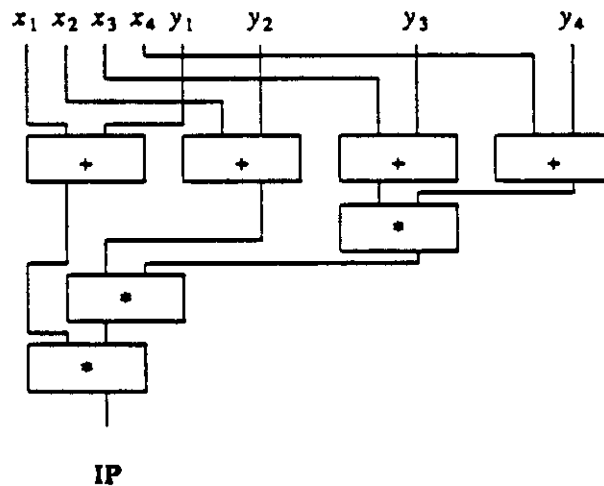


Figure 5.27 Inner Product.

Using the following space/time identities from Section 3.6 we can apply transformations to IP to transform it to a sequential implementation.

$$\&f \equiv \mathrm{D}^{-1} \ @\ \mathrm{POSI} \ @\ \&^T f \ @\ \mathrm{SOPI}$$

$$!f \equiv \mathrm{D}^{-1} \ @\ !^T f \ @\ \mathrm{apndr} \ @\ [\mathrm{SOPI} \ @\ \mathrm{tlr}\ ,\ \mathrm{last}]$$

Using these identities on the **Right Insert** and **Apply-to-All** combining forms of **IP** gives the function,

$\mathrm{D}^{-1} @ !^T + @\mathrm{apndr}@[\mathrm{SOPI} @ \mathrm{tlr},\ \mathrm{last}]@\mathrm{D}^{-1}@\mathrm{POSI}@\&^T * @\mathrm{SOPI}@\mathrm{trans}.$

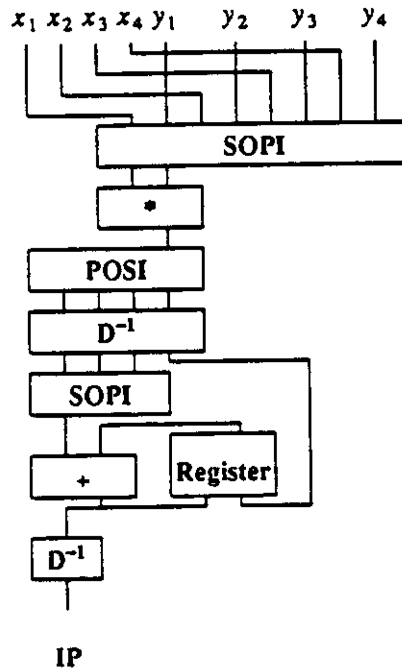


Figure 5.28 Time implementation of the Inner Product.

Figure 5.28 contains the sketch of this function. In this sequential implementation, all of the pairs are multiplied and accumulate in the **POSI** before any addition is performed. To improve it we would like to to cancel the **SOPI** and **POSI** in the center so that the additions can be performed as the results of the multiplier become available. The problem is that not all of the outputs of the **SOPI** go to the **POSI**; one is used to provide the initial value of the register of the **Right Insert**. As it is, the

POSI and SOPI cannot be canceled in this description. However there are two ways of altering the specification so that the SOPI and POSI cancel: by loading the register with the identity element for the operation (in this case the multiplicative identity) or by multiplying the first two pairs of numbers in parallel. These choices correspond respectively, to the following two identities,

$$!f \equiv !f \text{ @ apndr @ [id , \%}I]$$

where I is the identity for f and

$$\mathbf{D^{-1}} \text{ @ } \mathbf{POSI} \text{ @ } \&^T f \text{ @ } \mathbf{SOPI}$$

$$\equiv \text{apndr @ } [D^{-1} \text{ @ } \mathbf{POSI} \text{ @ } \&^T f \text{ @ } \mathbf{SOPI} \text{ @ tlr , } f \text{ @ last}].$$

Applying these two identities gives,

$$\mathbf{D^{-1}} \text{ @ } !^T\!+ \text{ @ apndr @ [} \mathbf{SOPI} \text{ @ tlr , last] @ apndr @ [id , \%1]}$$

$$\text{@ } \mathbf{D^{-1}} \text{ @ } \mathbf{POSI} \text{ @ } \&^T * \text{ @ } \mathbf{SOPI} \text{ @ trans}$$

and

$$\mathbf{D^{-1}} \text{ @ } !^T\!+ \text{ @ apndr @ [} \mathbf{SOPI} \text{ @ tlr , last] @ apndr}$$

$$\text{@ [} \mathbf{D^{-1}} \text{ @ } \mathbf{POSI} \text{ @ } \&^T * \text{ @ } \mathbf{SOPI} \text{ @ tlr , * @ last] @ trans}$$

Using the following identities,

$$\mathbf{POSI} \text{ @ } \mathbf{D^{-1}} \text{ @ } \mathbf{SOPI} \equiv \mathbf{SOPI} \text{ @ } \mathbf{D^{-1}} \text{ @ } \mathbf{POSI} \equiv \text{id}$$

$$f \text{ @ } \mathbf{D^{-1}} \equiv \mathbf{D^{-1}} \text{ @ } f \qquad \text{for } f \neq \mathbf{POSI, SOPI}$$

$$\text{[tlr , last] @ apndr} \equiv \text{apndr @ [tlr , last]} \equiv \text{id}$$

these can be further reduced to,

$$\mathbf{D}^{-1} \ @ \ !^T\!+ \ @ \ \text{apndr} \ @ \ [\text{id}\ ,\ \%1] \ @ \ \&^T\!* \ @ \ \mathbf{SOPI} \ @ \ \text{trans}$$

and

$$\mathbf{D}^{-1} \ @ \ !^T\!+ \ @ \ \text{apndr} \ @ \ [\&^T\!* \ @ \ \mathbf{SOPI} \ @ \ \text{tlr}\ ,\ * \ @ \ \text{last}] \ @ \ \text{trans}.$$

We can perform one more optimization by noticing that we can pair the numbers more easily in time rather than space. This is reflected by the following identity,

$$\text{SOPI} \ @ \ \text{trans} \equiv \text{trans} \ @ \ \&\text{SOPI}$$

and results in the two specifications,

$$\mathbf{D}^{-1} \ @ \ !^T\!+ \ @ \ \text{apndr} \ @ \ [\text{id}\ ,\ \%1] \ @ \ \&^T\!* \ @ \ \text{trans} \ @ \ \&\mathbf{SOPI}$$

and

$$\mathbf{D}^{-1} \ @ \ !^T\!+ \ @ \ \text{apndr} \ @ \ [\&^T\!* \ @ \ \text{trans} \ @ \ \&(\mathbf{SOPI} \ @ \ \text{tlr})\ ,\ * \ @ \ \&\text{last}].$$

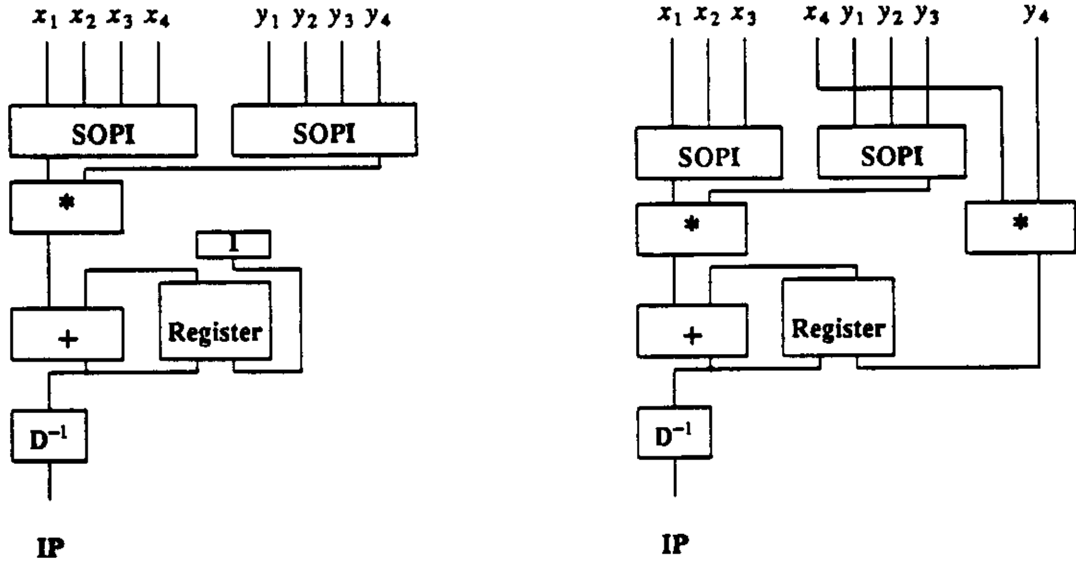


Figure 5.29 Optimized time implementations of the Inner Product.

The tradeoff between these two descriptions can be seen from Figure 5.29 which contains the sketches of both of these functions. The first requires an additional cycle since its SOPI's must process four inputs, while the second requires three cycles but an additional multiplier as well.

5.7 Memory

In this example we give a high level description of a digital memory. We start by describing what we would expect a memory to do. We want it to accept an instruction consisting of a read/write flag, an address and a word to write in the memory in the case the flag indicates a write, and the output should be the word selected by the address in the case of a read. Since there is no state in FP, in order to describe a memory we must construct a function which whose input also contains the current state and which generates the next state in its output. We can then fold the **Seq** of this function to provide the required feedback. We also cannot have conditional input or output. Hence the input to this function memory consists of a read/write flag, an address, a word and the current memory contents. The output consists of an output word and the new memory contents. The specification is generic allowing any address and word size.

```
# MemFct : <<R/W <addr> <word>> <contents>>
#                                          → <<new contents> <word>>
#
# input : <<flag < a_0 ... a_{n-1} > < b_1 ... b_m >> memory object >
#
# where memory object is < w_0 ... w_{2^n-1} > and w_i is <b_{1,i} ... b_{m,i}>
#

{MemFct [trans@1,2]@trans@&([tl,1]@seq(cell)@apndr@[id,%0])
                @&trans@distl@[1@1,&distl@trans@[2@1,trans@2]]
                @[AddrDecode@1,2]}

{AddrDecode [distl@[1,Decode@2],3]}
```

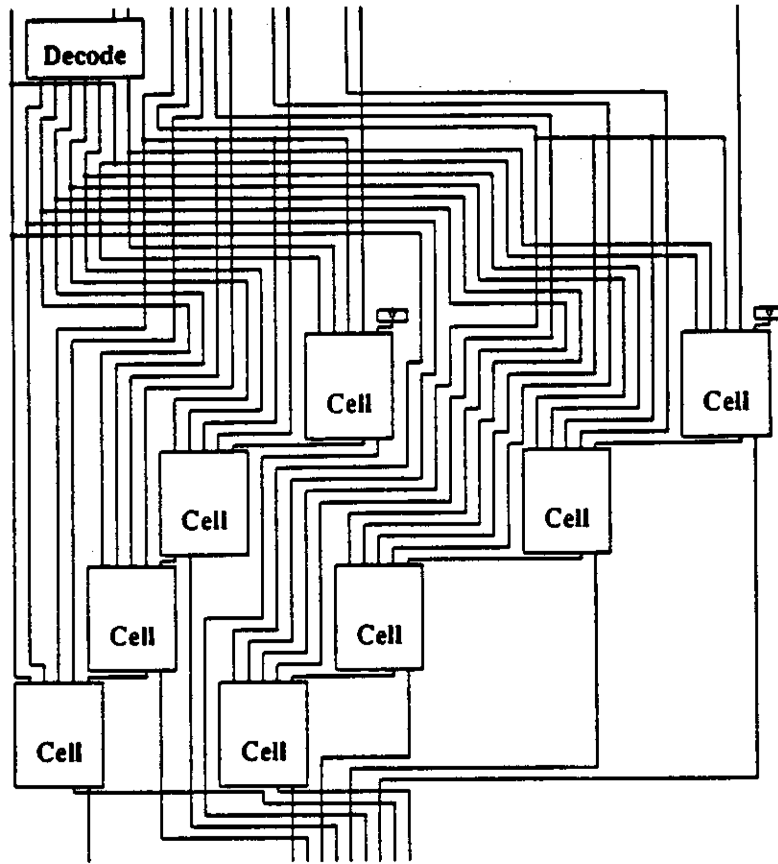


Figure 5.30 The sketch of **MemFct**.

```
{Cell [org@[andg@[2@1@1,2@2@1],2],org@[norg@1,andg@2]@distl
                               @[andg@1@1,[notg@2@2@1,1@2@1]]]}
```

The function **Decode** can be found in Section 5.1. Figure 5.30 shows the sketch obtained from **MemFct** with $n = 2$ and $m = 2$. The description of the memory is then obtained from the Seq of Memfct. We also provide the initial memory contents using the **Constant** combining form and remove the final memory contents by applying **tl** to the output.

```
{Memory tl@seq(MemFct)@apndr@[id,%<<0,0>,<0,0>,<0,0>,<0,0>>]}
```

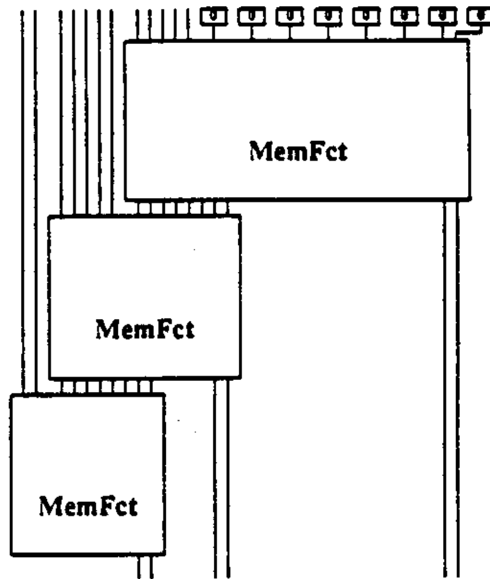


Figure 5.31 Space implementation of **Memory**.

Applying Memory to a sequence consisting of three instructions for $n = 2$ and $m = 2$ gives the sketch in Figure 5.31. Here **MemFct** is represented as a primitive. This is the space implementation of Memory. To obtain the time implementation we apply the transformation,

$$seq(f) \equiv \mathbf{D}^{-1} \text{ @ apndl @ [1 , } \mathbf{POSI} \text{ @ tl] @ } seq^{T}(f) \text{ @ apndr}$$

$$\text{@ [} \mathbf{SOPI} \text{ @ tlr , last]}$$

which gives,

{Memory $\mathbf{D}^{-1}$@apndl@[1,**POSI**@tl]@seq^{T}(MemFct)@apndr@[**SOPI**@tlr,last]}.

This gives the sketch in Figure 5.32 which is not quite what we had in mind. The problem is that the feedback for the memory is at the level of **MemFct** instead of at the **Cell** level. Fortunately, by applying transformations to **Memory** we can move the outer Seq inside the inner Seq and to perform the feedback at the **Cell** level. The

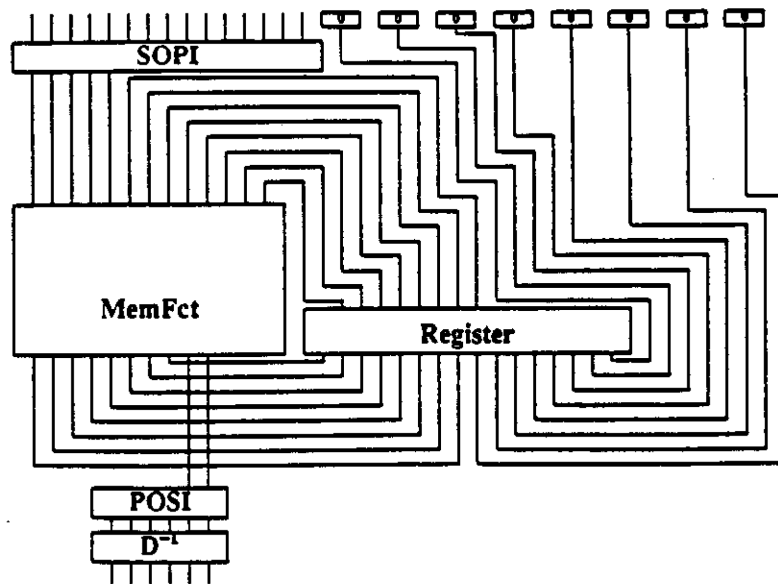


Figure 5.32 Time implementation of **Memory**.

outer **seq** becomes a **Right Insert** since the output of each cell corresponding to its current value is no longer an output of **MemFct**; it is routed back by the internal **Seq**.

```
{Memory D^-1@POSI@trans@&(2)@2
        @!([1@2,&([1@2,&^T(org)@trans@[tl@Cell,2@2]])
        @distl@[[1,1@2],2@2]])@apndr@[2,[1,&([id,&^T(%0)])
        @trans@3]]@[&^T(1), trans@&^T(Decode@2),&^T(3)]@SOPI}

{Cell seq^T([org@[norg@1,andg@2]
                @distl@[andg@1@1,[notg@2,2@1]],andg@[1@1@1,2]])
              @apndr@[trans@[trans@1,1@2],%0]}
```

Figure 5.33 shows the sketch of this version of **Memory** with $n = 3$ and $m = 4$. Figure 5.34 contains the new version of **Cell** which has the time implementation of the **Seq**. Note that we have also transformed the specification to pull the initialization constant to the **Cell** level. This is still a high level description of a Memory; lower level primitives should be substituted into the cells.

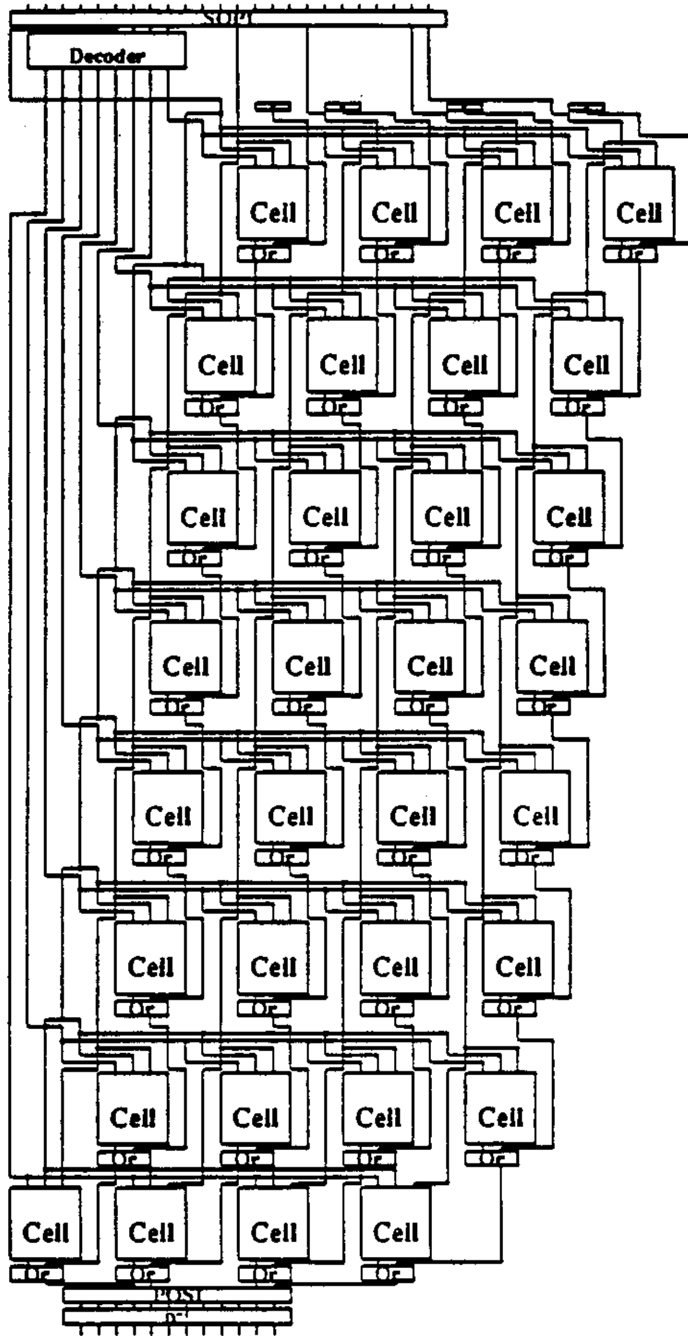


Figure 5.33 Optimized time implementation of **Memory**.

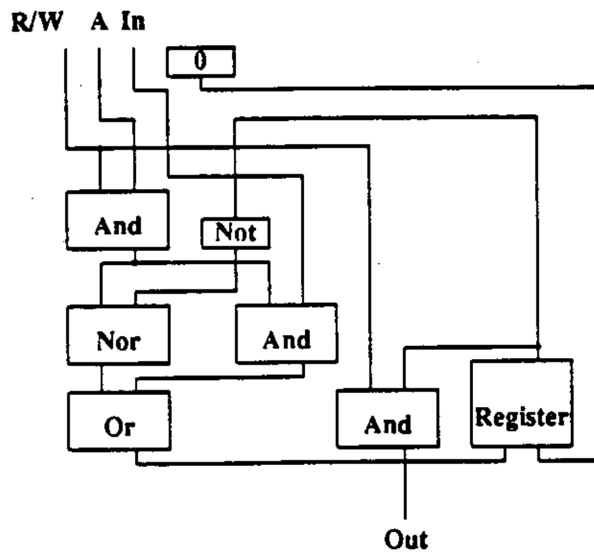


Figure 5.34 The sketch of **Cell**.

Summary

Several examples have been presented in this chapter to illustrate the style of specifying circuits in FP. Sketches and layouts were obtained directly from the FP specifications given, using tools which were constructed to implement the methods described in Chapters 3 and 4. The following advantages are illustrated with these examples.

1. The ability to refine high level specifications to lower levels using varying levels of abstraction.

2. The availability of graphical feedback during the refinement of the design, allowing the geometric consequences of the design to be assessed.

3. The ability to transform the specifications, in particular to explore the time and space tradeoffs.

These features allow the designer to experiment rapidly with different alternatives, pursuing to lower levels only those designs which prove themselves at higher levels. However, this method of specifying circuits currently suffers from the following limitations:

1. The current implementation of the mapping from planar circuits to layouts requires inputs and outputs to be on the top and bottom of components, respectively. This limitation excludes the horizontal flow of signals between components. Extensions to encompass this type of flow should be investigated.

2. FP specifications are not readily decipherable since the exact structure of intermediate objects must often be deduced in order to interpret them. However, it is this feature of using the structure of the objects to differentiate between signals which allows the planar topology of the circuit to be captured. Hence syntactic enhancements to the language should preserve this property.

CHAPTER 6

Improving Planar Topology

In this chapter, the problem of altering the planar topology to improve its layout is examined. We first discuss measures for planar circuits and the transformations we will consider to improve them. After selecting a particular cost measure, we consider the complexity of optimizing it and then present a method for improving planar circuits represented by computation trees with respect to this measure.

6.1 Measuring Planar Circuits

Unfortunately it is difficult to define an absolute cost measure for planar circuits such as the size of the final layout, since the quality of the final layout can depend largely on the method used in transforming the planar circuit to a layout rather than on the planar circuit itself. Two different planar circuits capturing the same circuit may both be optimal for different layout techniques. However, there are some measures which are topological in nature and can be used to compare planar circuits although there is no guarantee that their relative merits will manifest themselves in the final layout.

These measures involve maximizing the 'planarity' of the circuit. The plane graph of a planar circuit already provides a planar embedding. Thus measuring the 'planarity' of a planar circuit amounts to measuring the 'planarity' or 'wiring

complexity' of its R-nodes. There are three measures of R-nodes which come to mind.

1. Number of Crossings

   The minimum number of crossings with which the R-node can be implemented.

2. Number of Non-crossing Wires

   The size of a maximum subset of pins of an R-node whose connections can all be wired within the R-node without cross-overs. Note that we can always include at least one pin from each partition.

The first measure minimizes the number of crossings without regard to the number of contacts used. Whenever we have a crossing, we must route one of the wires on a different layer. If we were forced to return the wire to the initial layer immediately after the crossing then the number of crossings might be a good measure since it would correspond to the number of contacts needed. However this is not the case; once a wire is on another layer it should be able to cross wires on the initial layer for no additional cost; it should only return to the initial layer after it has crossed as many wires as required.

The second measure attempts to maximize the number of 'easy connections' and banish the remaining ones to other layers. By 'easy connections' we mean a set of pins which can be connected without crossings. In this sense it more closely approximates the final routing cost since once a connection is excluded from the set, there is no cost associated with it with respect to the other wires. However this means that there is no bound on the complexity of wiring the banished wires. The

hope is that in maximizing the number of connections made on the initial layer, the routing of the remaining wires will be relatively simple.

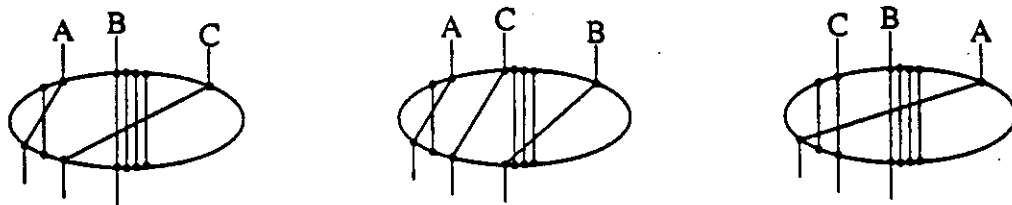


Figure 6.1 The initial configuration and optimal configurations with respect to the two cost measures.

Figure 6.1 illustrates the difference between the two cost measures. Suppose we have the possibility of exchanging the connections of the pins labeled A, B and C of the leftmost R-node in Figure 6.1. The middle R-node in Figure 6.1 is the optimal arrangement if we want to minimize the number of crossings, while the rightmost R-node is the arrangement with the maximum number of uncrossed wires.

Another criterion for the choice of a cost measure is its computational requirements. We did not consider the number of contacts as a measure since computing the minimum number of contacts required for routing an R-node whose partitions all consist of two pins, has been shown to be NP-complete [Mare84]. This measure also depends on implementation decisions such as the number of layers available. Computing the minimum number of crossings in for an R-node with partitions of size two is immediate. Finding a largest set of non-crossing wires in this case can also be handled easily if the R-node is bipartite, while computing the minimum number of crossings does not generalize as readily.

Definition 6.1

An R-node, $u = u_1 \cdots u_n$, is said to be *bipartite*, if there exist i and j such that for each partition of u, one of the sequences $u_i \cdots u_{j-1}$ and $u_j \cdots u_{i-1}$ contains exactly one pin of the partition. This pin is called the *source* of the partition. For partitions of size two, both sequences contain exactly one pin of the partition and one of these pins is arbitrarily selected as the source.

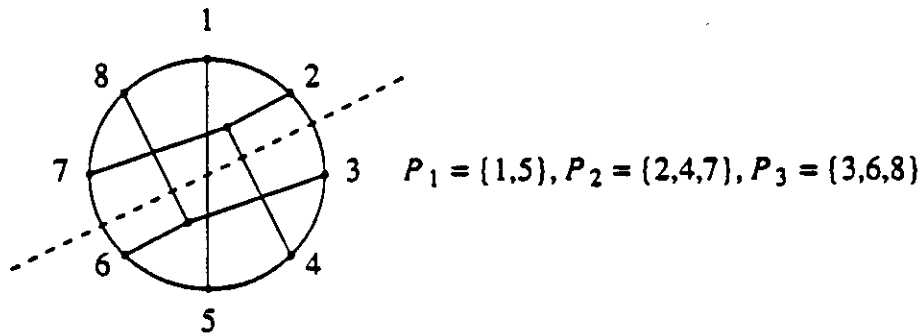


Figure 6.2 A bipartite R-node

Figure 6.2 contains a bipartite R-node. In the R-node in Figure 6.2, the sequence of pins is divided into 7,8,1,2 and 3,4,5,6. P_1 has one pin in both subsequences, P_2 has one pin in the second and P_3 has one pin in the first. We first present a simple algorithm to find the largest set of non-crossing pins of a bipartite R-node with partitions of size two. We then discuss how to generalize this procedure to the case of arbitrary sized partitions.

To obtain a maximal set of non-crossing pins of an R-node, we generate its conflict graph which will have the partitions as nodes and edges between partitions which do not conflict (partitions whose connections can be made without crossings). Finding the largest non-crossing set of pins is then equivalent to finding a maximum weight clique (a set of nodes in which each pair is joined by an edge) in this graph.

Finding a maximum clique is in general NP-complete [Gare79]. However Even and Pnueli [Even72] have shown that in the case of transitively orientable graphs it is quite simple.

Definition 6.2

A *transitively orientable* graph is an undirected graph in which the nodes can be numbered so that when the edges are oriented from low to high, the resulting directed graph is acyclic and transitive (the existence of edges $(\overrightarrow{v,u})$ and $(\overrightarrow{u,w})$ implies the existence of the edge $(\overrightarrow{v,w})$).

We observe that in the case of bipartite R-nodes with partitions of size two, the resulting graph is transitively orientable. This was first observed in [Gopa83]. The numbering of the nodes is obtained by ordering the partitions according to the order of their first pins within one of the two subsequences of the bipartite R-node. Then for $i<j<k$ it is clear that if P_i, P_j and P_j, P_k are not in conflict, then P_i, P_k are not in conflict. In the bipartite R-node in Figure 6.3, this order would be P_1,P_2,P_3,P_4 is the R-node were divided as shown and the subsequence 7,8,1,2 was used. The conflict graph and its transitive orientation are shown in Figure 6.4.

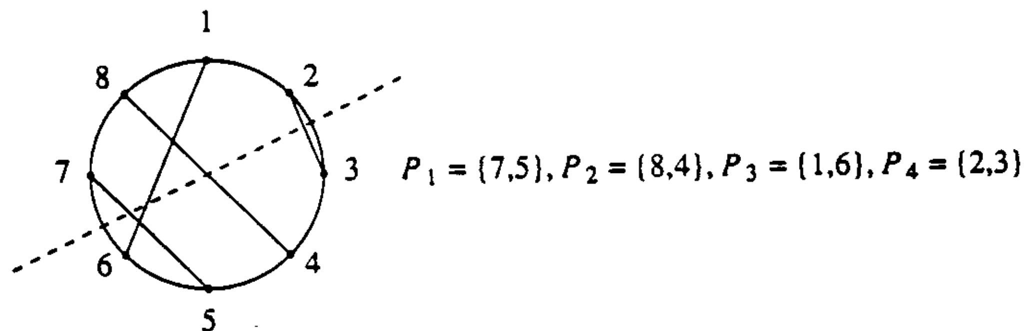


Figure 6.3 A bipartite R-node with partitions of size two.

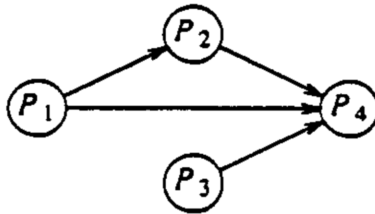


Figure 6.4 The transitively oriented conflict graph of the *R*-node in Figure 6.3.

Finding the maximum clique for transitively oriented weighted graphs is simple. We compute for each node j the maximum clique to which it belongs in the subgraph induced by the nodes $1,2,\ldots,j$. Let $c(j)$ denote this value. Then $c(j)$ is computed as follows where $w(j)$ denotes the weight of node j.

1. $c(1) = 1$

2. $c(j) = \max\limits_{\overrightarrow{(i,j)}}(w(j)+c(i))$

To see how this works, suppose we have computed $c(i)$ for each $i < j$. To compute $c(j)$ it suffices to examine all the edges which enter j and the $c(i)$'s of these nodes. If there is an edge from k to j for $k < j$ then adding j to any clique in which k is the highest numbered node will result in another clique by the transitivity property. To obtain the maximum clique we simply keep track for each j, of the incoming edge by which it attains its value. After obtaining the $c(j)$'s for each node, the size of maximum clique is the maximum of the $c(j)$'s over all of the nodes and the corresponding clique can be enumerated by tracing backward over the saved edges starting from a node with a maximal $c(j)$. This gives an algorithm whose time and space requirements are bounded by $O(n^2)$ for a graph of n nodes.

Unfortunately, this algorithm does not extend to arbitrary R-nodes even when all partitions consist of two pins. The crucial transitively orientable property no longer holds. Figure 6.5 contains such an R-node and its conflict graph. Picking either orientation for the edge from A to E in the conflict graph forces the other orientations of the other edges and produces a directed graph which is not transitive.

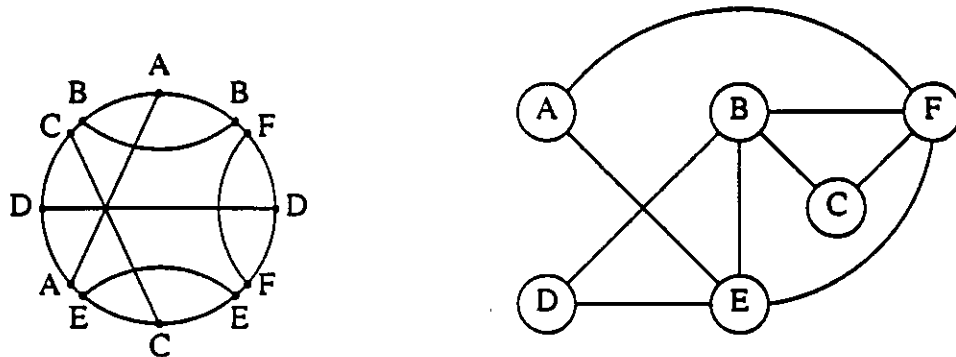


Figure 6.5 An R-node whose conflict graph is not transitively orientable.

Fortunately, the R-nodes generated from FP expressions are bipartite. In Chapter 3 we saw that the planar circuits resulting from FP expressions were directable and the inputs and outputs of each R-node or sub-tree of the computation tree were not interleaved. Hence dividing the sequences of of an R-node into inputs and outputs gives the two required subsequences for bipartite R-nodes since each partition has exactly one input pin. The more unrealistic of the two assumptions needed to use Even and Pnueli's algorithm is the constraint on the size of the partitions. Although many partitions do in practice consist of two pins, there are enough which do not to make it impractical to ignore them. We will generalize to arbitrary sized partitions as follows. Suppose we have a bipartite R-node u whose sequence is divided into two the subsequences required for a bipartite R-nodes and we have a partition of u of size $k+1$ for $k>1$, whose source pin is u_0. The other pins

in this partition, $u_1 \cdots u_k$ are in the other subsequence. We replace u_0 by $u_{0,k} \cdots u_{0,1}$ in u's sequence in this order. We also replace the partition by the k partitions,

$$\{u_{0,i}, u_i\} \text{ for } 1 \le i \le k.$$

We have simulated the original partition of $k+1$ pins by k partitions of two pins each. We do this for each partition of size greater than two and then find the maximum set of non-crossing pins. To obtain the maximum set of non-crossing pins we proceed as follows.

1. We obtain the maximum set of non-crossing pins of this bipartite node with partitions of size two.

2. For each pair $\{u_{0,i}, u_i\}$ if only one of the two pins is in the maximum set of non-crossing pins we include $u_{0,i}$ rather than u_i.

3. We remove the $u_{0,i}$'s and restore u_0.

Since the new pins, $u_{0,i}$'s, were all adjacent, if they were connected to their respective u_i's in the maximal set of non-crossing pins obtained in the first step, then u_0 can be connected to these u_i's, which are the ones remaining in the final maximal set of non-crossing pins. Note that this is not quite the same definition of cost since connections between non-source pins are not counted in the final maximal set of non-crossing pins. Only connections between pins and their source pins are counted.

We can find the maximum non-crossing set of pins under the above restrictions. In Section 2.8, we argued that a maximal indivisible planar circuit was optimal with respect to cost measures which satisfied certain assumptions. We discuss these assumptions with respect to the maximum set of non-crossing pins as

defined above. We define the cost of a node to be the number of pins which are not part of the maximum non-crossing set.

Definition 6.3

The cost of an R-node, u, is $c(u) = |u| - m(u)$ where $m(u)$ is the number of pins in a maximal non-crossing set of pins for u. The cost of a planar circuit is $c(A) = \sum_{u \in R\text{-nodes}} c(u)$.

We show how the following assumptions made about the cost functions in Section 2.8, are satisfied by this cost function.

Assumption 1:

The cost function over the layout is an additive function of the components, B-nodes, and the implementations of R-nodes. It is defined by

$$c(A) = \sum_{x \in B\text{-nodes}} C(x) + \sum_{u \in R\text{-nodes}} c(u)$$

where $C(x)$ is the cost associated with B-node x and $c(u)$ is the cost of implementing R-node u.

Assumption 2:

If u and v are two adjacent R-nodes then $c(u) + c(v) \geq c(M(u,v))$.

Assumption 3:

If z is an R-node which can be cleanly divided into u and v then $c(z) \geq c(u) + c(v)$.

Assumption 4:

If u is an R-node and T is an untangling of u then $c(u) \geq c(T(u))$.

Assumption 5:

If u is a trivial R-node then $c(u)=0$.

Assumption 1 is satisfied by setting the cost of a B-node to be 0. Assumptions 3, 4 and 5 are clearly satisfied for our cost function. Assumption 2 requires closer examination. Suppose we have two R-nodes u and v connected by a set of n wires whose pins are adjacent on both u and v. Suppose p and q are maximal set of non-crossing pins for u and v, respectively. We partition p and q into p_1, p_2 and q_1, q_2 according to whether the pins belong to the set of n adjacent wires or not. Let w be the node which results from the merging of u and v along the n wires. Then $|w| = |u| + |v| - 2n$ since exactly $2n$ pins will disappear as a result of the merge. The set of pins $p_1 \cup q_1$ will be part of w and will not be in conflict within w. Hence we have $m(w) \geq |p_1| + |q_1|$. Since both p_2 and q_2 consist of no more than n pins we have,

$$
\begin{aligned}
c(u)+c(v) &= |u| - m(u) + |v| - m(v) \\
&= |u| + |v| - (|p_1| + |p_2| + |q_1| + |q_2|) \\
&= |u| + |v| - (|p_1| + |q_1|) - (|p_2| + |q_2|) \\
&\leq |u| + |v| - (|p_1| + |q_1|) - 2n \\
&\leq |w| - (|p_1| + |q_1|) \\
&\leq |w| - m(w) \\
&\leq c(w)
\end{aligned}
$$

These five assumptions are satisfied by this cost function. However, the last

requirement mentioned in Section 2.8, is not in general true.

$$c(RE_{+}(u)) = c(u).$$

The problem is that a refolding can increase or decrease the number of wires and hence pins as a result of its unclean divide and/or merge. Our planar circuits however will not have R-nodes with self-loops. We specifically avoided merging the R-nodes which would create nodes with self-loops, preferring to consider the arrangement of the wires connecting the R-nodes separately rather than as self-loops on one R-node. In this case, the optimization we are attempting to perform tries to select the best refolding by examining the wires which would become the self-loops of the maximal R-node.

In the next section we describe the transformations we will consider in order to improve a planar circuit.

6.2 Transforming Planar Circuits

Before we can optimize a planar circuit, we must decide on the class of planar circuits which we will consider as alternatives. At the most this class could include any planar circuit which captures the same circuit as the given one. Unfortunately this would require us to solve the general routing and placement problem which we have avoided by restricting ourselves to a fixed planar topology. As discussed earlier, routing and placement comprise many intractable problems. Since we do not want to have to consider the general layout problem we will limit ourselves to certain types of operations which preserve the characteristics of the original planar circuit, while exploiting its symmetries.

In Section 2.8, we argued that the layout procedure should consider permuting the order of the self-loops of an R-node in order to reduce the internal complexity of wiring the R-node. So at the least we should consider permuting the order of adjacent self-loops. In addition, in Section 3.8 we did not perform all merges possible and ended up with some non-maximal R-nodes. We must also consider reordering these wires between R-nodes in order to guarantee that we do not obtain a sub-optimal result by implementing non-maximal R-nodes.

Components often have pins which are equivalent, that is, perform the same function when a subset of the inputs are interchanged. This is the case with all commutative operators such as the boolean functions AND, OR, NAND, NOR and XOR. Since many components are constructed from these elements they have equivalent pins which could be interchanged to reduce the complexity of the adjacent R-nodes. In addition to the symmetry between pins, we can also exploit the symmetry provided by the representation of a planar circuit by the computation tree. The combining forms **Construct** and **Apply-to-All** offer us the possibility of reordering their subfunctions to reduce the wiring complexity of adjacent routing trees.

Unfortunately, the problem of optimizing our cost function is still hard. In the next section we show that it is NP-hard in the case in which we consider only the operation of switching equivalent pins of components. This problem arises in layout and is known as the pin alignment problem, [Schl84] and [Schl85]. However we will be able to perform more powerful transformations to improve the 'pin alignment' of planar circuits represented by computation trees by exploiting the symmetries implied by the combining forms. We will consider the following

transformations.

We are given a planar circuit $A=(P,IO,B,R,W)$ and for each B and R-node, $u=u_1\cdots u_{n_u}$, a set of permutations of its pins, $\Pi(u)=\{\pi_i\}$ where each π_i is a permutation of the integers $1,2,\ldots,n_u$. We require that $\Pi(u)$ contain the identity permutation. We say that π_i is a legal permutation for $u=u_1\cdots u_{n_u}$, if the pair of pins $u_j,\pi_i(u_j)$ are connected by wires to the same B or R-node for each $1\le j\le n$. Note that the identity permutation is always legal since this is trivially true if $u_j=\pi_i(u_j)$.

Definition 6.4

A *legally derived planar circuit* from (A,Π) for $A=(P,IO,B,R,W)$, is a planar circuit $A'=(P,IO,B',R',W)$ for which there exists a mapping $f_P:P\rightarrow P$ satisfying the following conditions.

a. The B and R-nodes of A' consist of the B and R-nodes of A respectively, under the mapping f_P which extends to sequences of pins as follows: $f_P(u)=(f_P(u_1)\cdots f_P(u_{n_u}))$ for $u=u_1\cdots u_{n_u}$.

b. For each B or R-node u, $f_P(u)=\pi_i(u)$ for some $\pi_i\in\Pi(u)$. For R-nodes, the partitions of $f_P(u)$ are those of u under the mapping f_P.

c. The sequence of IO pins which forms the boundary is the same in A' and A.

Note that we are simply permuting the pins of B and R-nodes along their boundaries. The wires remain unchanged. The resulting embedding of the plane graph implied by this circuit is the same. To see this it suffices to observe that if we ignore the internals of the R-node and B-nodes, what is occurring is simply a relabeling of the wires. The cyclical ordering of the edges of the nodes in the plane graph is preserved

under this relabeling. If we have a hierarchical representation of the planar circuit, such as the one afforded by a computation tree, then we can extend this type of transformation to ones which alter the plane graph by considering sub-sections of the planar circuit as B-nodes.

In the next section we show that optimizing our cost function for planar circuits under only the transformations given above is NP-complete. Subsequently we will present a method for improving planar circuits with respect to this measure on planar circuits represented by computation trees, considering the additional transformations afforded by this representation.

6.3 The Complexity of Pin Alignment in Planar Circuits

In this section, the problem of optimizing a planar circuit with respect to our cost measure and using only a simple transformation is shown to be NP-complete. Our construction will result in a maximal indivisible planar circuit whose R-nodes are all bipartite and have only partitions of size two. In addition we will allow any permutation of R-nodes. Each B-node will have at most two possible permutations including the identity. We first present the problem we will reduce to it, **Planar 3SAT.**

SAT

Given (X,C) where X is a set of variables, $X=\{x_1,x_2,\ldots,x_n\}$, and C is a collection of clauses, $C=\bigcup_{i=1}^{i=m} c_i$ each $c_i=\{z_i^1,\ldots,z_i^{j_i}\}$ is a clause of literals drawn from X (i.e. a subset of size j_i of $X\bigcup\bar{X}$), is there an assignment of values, true or false, to the

variables such that each clause contains at least one literal with the value true?

3SAT

Given an instance of SAT such that each clause has at most three literals ($c_i = \{z_i^1, z_i^2, z_i^3\}$), is it satisfiable?

3SAT has been shown to be NP-complete as well as SAT. The problem which will be used to show the NP-completeness of pin alignment problem is **Planar 3SAT**.

Planar 3SAT

Given an instance of 3SAT, (X,C), such that the following graph, denoted by $G(X,C)$ is planar, is (X,C) satisfiable?

$$G(X,C) = (V,E) \quad \text{where,} \quad V = X \cup C \quad \text{and} \quad E = \{ (x_i, c_j) \mid \text{if } x_i \in c_j \text{ or } \overline{x}_i \in c_j \}$$

Lichtenstein [Lich82] has shown **Planar 3SAT** to be NP-complete by reducing **3SAT** to it. In his reduction, the graph of an arbitrary instance of **3SAT** is mapped into the plane and then modified by adding new clauses and variables to remove crossings, thus providing an instance of **Planar 3SAT** and a planar realization of its graph. In Lichtenstein's construction, $G(X,C)$ also contained edges to interconnect the nodes corresponding to the variables in a simple closed path. These edges have been omitted here from the definition of $G(X,C)$ without loss of generality. To facilitate the reduction, it will be assumed that the instance of **Planar 3SAT** also comes with an embedding in the plane which is specified by the clockwise cyclical ordering of the edges around each node and the outer boundary. This extension to the definition of **Planar 3SAT** does not affect its NP-completeness since Lichtenstein's construction provides this information. The decision problem of pin alignment in the case of planar circuits is as follows.

Planar Circuit Pin Alignment

Given a planar circuit A whose R-nodes are bipartite and consist only of partitions of two pins, a set of permutations for its B-nodes Π with at most two permutations per B-node, and an integer K, is there a legally derived planar circuit A' from A and Π whose cost does not exceed K ?

The problem of planar circuit wire pin alignment will be shown to be **NP-hard** by reducing **Planar 3SAT** to it.

Construction

Suppose $X=\{x_1,x_2,\ldots,x_n\}$ and $C=\{c_1,c_2,\ldots,c_m\}$ is an instance of **Planar 3SAT**. If a variable appears in both its complemented and uncomplemented form in a clause then the clause can be removed since any assignment will satisfy it. If a variable appears only in one form but more than once, then remove all but one copy from the clause. This can be repeated until all clauses have at most one literal corresponding to any variable. A new version of **Planar 3SAT** is obtained which has a solution if and only if the original had one and its graph is a subgraph of the original's so it can inherit its embedding from it. Hence it will be assumed that no variable occurs twice within the same clause. In addition, we can assume that $G(X,C)$ does not have any isolated nodes since these would correspond to variables which do not occur in any clause or empty clauses either of which can be removed without affecting the existence of a solution.

We have an embedding in the plane of $G(X,C)$. To construct the instance of PCPA we will replace each variable node and clause node by an R-node and some B-nodes. In addition a B-node will be placed on each edge. These B-nodes will each

have two permutations. The choice of which permutation to use for these B-nodes will correspond to an assignment of a truth value to the literal represented by this edge. The R-nodes and B-nodes which replace the variable and clause nodes of $G(X,C)$ will serve to enforce a consistent assignment of values to literals corresponding to the same variable and will insure that at least one literal in each clause is true.

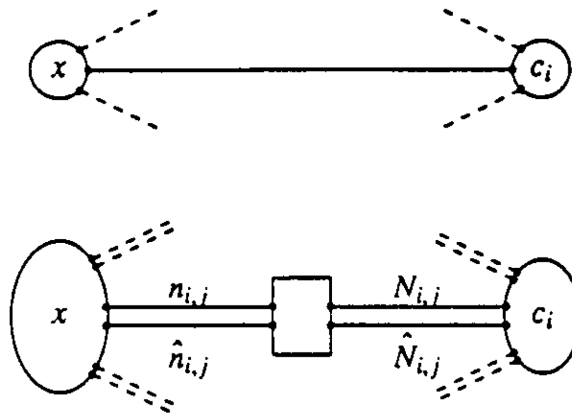


Figure 6.6 The B-node and wires which replace an edge of $G(X,C)$

For each edge of $G(X,C)$, e_i^j, corresponding to a literal z_i^j, we create four wires, $n_{i,j}$, $\hat{n}_{i,j}$, $N_{i,j}$ and $\hat{N}_{i,j}$. We will also create a B-node as in Figure 6.6 which has four pins, two on each side, each belonging to one of the four wires. The $n,\hat{n}$ wires connect two of the pins to the R-node corresponding to the variable R-node, while the $N,\hat{N}$ wires connect the other two pins to the clause R-node. Two permutations will be allowed for this B-node, the identity and the one in which pins of $n_{i,j}$ and $\hat{n}_{i,j}$ exchange as well as those of $N_{i,j}$ and $\hat{N}_{i,j}$. Thus this B-node would either be

$$n_{i,j}\, N_{i,j}\, \hat{N}_{i,j}\, \hat{n}_{i,j} \ \text{ or } \ \hat{n}_{i,j}\, \hat{N}_{i,j}\, N_{i,j}\, n_{i,j}.$$

Figure 6.7 contains the other possible configuration for the B-node.

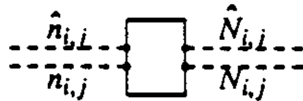


Figure 6.7 The alternate configuration for the B-node replacing an edge of $G(X,C)$

The configuration selected will correspond to an assignment to the literal. The R-nodes for the variables and clause node will insure that all literals corresponding to a given variable have the same assignment and that at least one literal of each clause is satisfied.

Consider a node corresponding to a variable x. Let M_x be $Max(\ \{z_i^j \mid z_i^j = x\},\ \{z_i^j \mid z_i^j = \bar{x}\}\ )$, and m_x be $Min(\ \{z_i^j \mid z_i^j = x\},\ \{z_i^j \mid z_i^j = \bar{x}\}\ )$, the maximum and minimum of the number of instances of uncomplemented and of complemented literals corresponding to x among the clauses. By assumption, $M_x > 0$. We construct a B-node and an R-node as shown in Figure 6.8. The wires arrive at the R-node in the same cyclical order as the edges of the node x. In Figure 6.8 they are shown entering from the right, in this clockwise order from top to bottom. The selection of the edge which is first in the sequence is arbitrary, however if this node is on the boundary we require that the cyclic order begin and end so that the exterior occurs before the first and after the last edge in the sequence. This will put the B-node on the exterior in this case and which will enable us to add an IO-node to it later, to ensure the reachability condition.

Let us denote the pins which belong to the wires $n_{i,j}$ and $\hat{n}_{i,j}$, as $p_{i,j}$ and $\hat{p}_{i,j}$. They will each be connected to another pin $q_{i,j}$ and $\hat{q}_{i,j}$ which exits the R-node on the left and is connected by a wire to the B-node in Figure 6.8. There will be only one configuration for this B-node. If $M_x > m_x$, then below the last edge in this

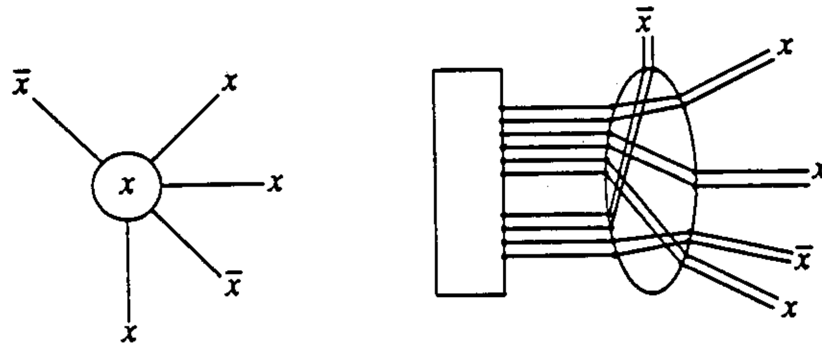


Figure 6.8 The R-node and B-node which replace a variable node of $G(X,C)$.

sequence a new B-node with $2*(M_x - m_x)$ is added and connected to the R-node as in Figure 6.9. In this case two extra pins where needed since there was one more x than $\bar{x}$. These pins insure that the number of literals corresponding to x and $\bar{x}$ are the same. Finally we add two more B-nodes each with $2B$ pins at the top and the bottom of the sequence where $B = Max\{M_x \mid x \in X\}$. We order the pins on the big B-node to force a consistent truth assignment to the literals corresponding to x. We divide the q pins into two groups corresponding to x and $\bar{x}$. The $2*(M_x - m_x)$ are added to the smaller of the two groups so that the two groups contain the same number of pins.

If the topmost literal is x then we place the pins in the group corresponding to $\bar{x}$'s on top followed by the other group preserving the order in which they enter the R-node within the group. If the topmost literal corresponds to $\bar{x}$, we do the reverse. The pins corresponding to the two sets of $2B$ pins are placed in between these two groups so that they cross each other. See Figures 6.8 and 6.9. Within each group the $q_{i,j}$ pin is placed in the same order with respect to the $\hat{q}_{i,j}$ as the $p_{i,j}$ is with respect to the $\hat{p}_{i,j}$ pin. As a result the two connections $(q_{i,j}, p_{i,j})$ and $(\hat{q}_{i,j}, \hat{p}_{i,j})$ will not conflict inside the R-node unless the pins $p_{i,j}$, $\hat{p}_{i,j}$ are interchanged as the result of selecting the alternative configuration for the B-node to which they are connected by

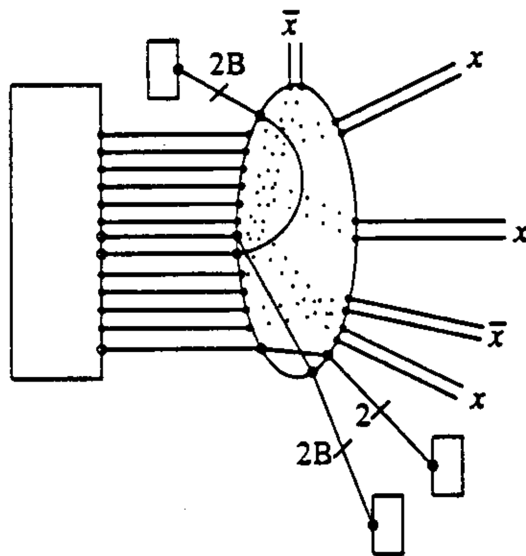


Figure 6.9 The additional pins and B-nodes added to the variable R-node.

$n_{i,j}$, $\hat{n}_{i,j}$.

Now consider a node corresponding to a clause c_i. This node has one, two or three edges in $G(X,C)$. Figure 6.10 shows the three possible cases. Figure 6.10 also contains the R-node and B-node that is generated in each case. Essentially an R-node is generated with pins for each of the N,$\hat{N}$ wires which belong to the clause. These wires enter from the right as pictured and exit from the R-node in the opposite order. Thus it is possible to connect at most one of the pair of wires within this R-node and only if the alternate configuration for the B-node on which these wires originate, is selected. This is the configuration which is required to connect the corresponding n,$\hat{n}$ wires within the variable R-node.

In order to obtain a planar circuit, we must add IO-nodes and ensure that the reachability property holds. It is clear that any connected component of $G(X,C)$ has pins which are connected through B-nodes to every other pin since the B-nodes of

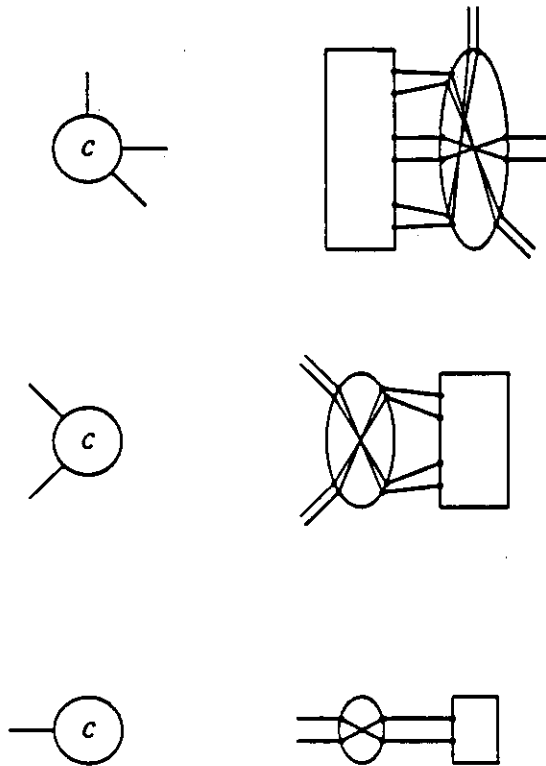


Figure 6.10 The R-node and B-node generated for a clause node of $G(X,C)$.

the variable and clause nodes achieve this connectivity. To insure the reachability property it suffices to add an IO-node on the boundary of each component. There is at least one variable node on the boundary of each component. We add an IO-pin connected to its B-node. Remember that the B-node would be on the exterior in this case. Finally we list these IO-nodes in any order as the boundary.

To complete the construction, K must be specified.

$$K = \sum_{i=1}^{m} 2(|c_i|-1) + \sum_{x \in X} 6(B+M_x)$$

This number reflects the fact that at most $6(B+M_x)$ pins of each variable R-node can be wired without crossings and only one pair of wires can be connected without

crossings within each clause R-node. Since all partitions are of size two we will refer to them as wires. We say that a partition or wire is connected if both pins are included in the maximum set of non-crossing pins. Note that the cost of an R-node in this case is the number of unconnected wires.

Claim: The constructed instance of PCPA has a solution if and only if the instance of **Planar** 3SAT has a solution.

Proof: Suppose first that there is a solution to (X,C). This solution corresponds to a solution for the PCPA instance as follows. Consider the wires, $n_{i,j}$, $\hat{n}_{i,j}$, $N_{i,j}$ and $\hat{N}_{i,j}$ which correspond to the literal z_i^j. If this literal is true under the assignment to the variables, then select the alternate configuration for the corresponding B-node so that the wires N and $\hat{N}$ can be connected within their clause R-node. Otherwise, if z_i^j is false, leave the B-node in the original configuration in which the n and $\hat{n}$ wires are in the proper order to be connected within their variable R-node.

Since the assignment satisfies the clauses, each clause must have at least one pair of wires which can be connected. Now consider a variable node. Note that the pins corresponding to the literals which are false are not in conflict with each other within the R-node. We can connect them along with one of the two sets of $2B$ wires, giving $6(M_x+B)$ pins which can be connected without crossings. Hence we obtain a legally derived planar circuit from A which has cost K.

Now suppose that there exists a solution to the **PCPA** instance. Clearly at least $2(|c_i|-1)$ pins cannot be connected at each clause R-node, leaving at most one two wires or $2(|c_i|+1)$ pins which might be connected. Now consider the cost of a variable R-node. It is not possible to connect both sets of $2B$ wires since they must cross within the R-node, so at least $2B$ wires are not connected. If any of wires of

the upper set of $2B$ wires are connected, then none of the wires which are in the upper group of wires can be connected, so we already have $2(B+M_x)$ wires which are not connected. Similarly if any of the wires of the bottom $2B$ wires are connected, then the $2(B+M_x)$ wires corresponding to the the upper $2B$ pins and the lower group of wires cannot be connected. The remaining case to consider is if both sets of the $2B$ sets of wires are not connected. Since $4B \geq 2(B+M_x)$, in all three cases we end up with at least $2(B+M_x)$ wires which cannot be connected. Summing over all of the R-nodes, we have already reached the maximum number of wires which cannot be connected in order to have a planar circuit with cost at least K. Even if $4B=4(B+M_x)$, we cannot not afford not to connect any of the $4B$ wires since the remaining wires would still have conflicts and could not all be connected. Thus each variable R-node must have pins from one of the two sets of $2B$ wires and the hence the $2M_x$ pins of the wires which cross it cannot be connected. Note that we have already excluded $2(B+M_x)$ wires from being connected at each variable R-node. All other pins besides these and the $2(|c_i|-1)$ unconnected wires at each clause node must be connected in order to meet the upper bound, K

This solution to PCPA corresponds to an assignment satisfying (X,C) as follows. Consider a variable, x. At least one literal corresponding to x exists ($M_x>0$). Consider the two sets of $2M_x$ wires connected and unconnected in the variable R-node corresponding to x. The sets of uncomplemented and complemented literals are contained in one or the other of these two sets; either all of the wires corresponding to uncomplemented literals are connected or all of the wires corresponding to complemented literals are connected, but never some from both sets. Assign a value to x such that the set of literals corresponding to connected segments are false and the set of literals corresponding to remaining pins are true. Now consider a clause

c_i. It must have at least one pair of wires $N_{i,j}$ and $\hat{N}_{i,j}$ corresponding to the literal z_i^j which are connected. Note that both $N_{i,j}$ and $\hat{N}_{i,j}$ can be connected only if the alternate configuration for the B-node has been selected. Consider the variable R-node for x which corresponds to the literal z_i^j. The two pins corresponding to $n_{i,j}$ and $\hat{n}_{i,j}$ can be connected only if the original configuration of the B-nodes was selected. Since this is not the case, at least one of these two wires is not connected. Both of these wires belong to the same group of $2M_x$ wires which are unconnected, which means that they correspond to a literal which is true and the assignment given to the variables satisfies the clause.

□

The NP-completeness of this problem follows from the observation that we need only guess a set of permutations, generate the corresponding f_P, and check whether it results in a legally derived planar circuit of cost less than K.

6.4 Using Computation Trees to Improve Planar Circuits

In this section, we present a method for improving planar circuits with respect to the transformations described in Section 6.2. In addition, we will consider transformations which arise by considering a sub-tree whose root is either a {,} or a & combining form, to be a B-node in which the pins can be permuted corresponding to the reordering of the sub-trees. The technique involves representing the possible permutations for the sub-trees of a node in the computation tree and resolving adjacent sub-trees to reduce the cost of intervening R-nodes.

We represent the set of possible permutations in a hierarchical fashion. The allowed permutations are represented by a forest called a *permutation forest*, imposed on the sequences of input and output connections. The leaves correspond to the input and output connections. Each internal node is labeled either A,R or F, indicating that the order of its children can be permuted in any order, can only be reversed or is fixed, respectively. The numbers of trees in the sequences of inputs and outputs are the same, and they are paired by an imaginary connection. Figure 6.10 contains a permutation forest. The permutations allowed are those which can be obtained by reordering the pairs of trees, permuting the order of the children of a node labeled A and reversing the order of the children of a node labeled R. A configuration which exercises every possible change in the example of Figure 6.10 is,

$$\begin{array}{ccccccccccccccc} q & o & n & p & a & b & c & d & e & m & l & k & j & i & g & f & h \\ 13 & 14 & 12 & 3 & 4 & 5 & 2 & 1 & 11 & 10 & 8 & 9 & 6 & 7 \end{array}$$

We assume that the permutations of the actual B-nodes of the planar circuit can be represented in this manner as well. The permutation forest for a **Projection** or **Apply-to-All** combining form is obtained by simply concatenating the permutation forests of its children in left to right order. Connections between sub-trees result from the **Compose**, **Right Insert** and **Seq** combining forms. We will concentrate on the **Compose**. The **Right Insert** and **Seq** combining forms use the same methods.

We must resolve the permutation forests of the sub-trees of a **Compose** and generate its permutation forest. There are essentially three steps in the procedure. We assume that a sub-tree whose root is a **Compose** is in normal form as described

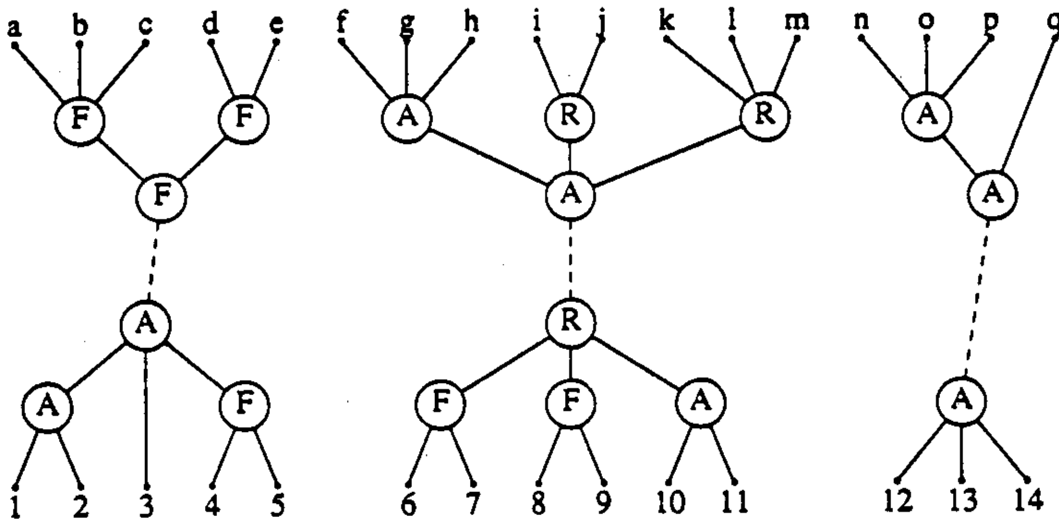


Figure 6.10 A permutation forest.

in Section 3.8.

1. Since the **Compose** tree is in normal form, no two routing sub-trees are adjacent in its list of children. We first combine the permutation forests of adjacent non-routing sub-trees. The procedure is called Simplematch and is described below. We obtain a new permutation forest for the combined sub-trees. We repeat this for every pair of adjacent non-routing trees until the list of children consists of an alternating sequence of routing and non-routing sub-trees.

2. The first and last children of a **Compose** tree in normal form are non-routing trees. We refer to these the *terminal* sub-trees. For each permutation forest of a non-routing non-terminal sub-tree, we select an order for its pairs of trees. This procedure attempts to preserve the separation between trees of permutation forests that might be lost if they were combined first with their neighbors. It is called Global-Align and also is described below.

3. The last procedure is used to select a permutation from the permutation forests which minimizes the cost of the routing sub-trees. Since the **Compose** is in normal form, every routing sub-tree is bordered on either side by two non-routing sub-trees. We need to select a configuration for these two sub-trees which minimizes the cost of the *R*-node sandwiched in between them. We must then combine and obtain a new permutation forest representing the remaining possible configurations of the combined sub-trees. This procedure is called Complexmatch.

We now describe the three procedures.

Simplematch

Suppose we have two adjacent non-routing sub-trees whose configurations are represented by permutation forests. We will refer to the connections between them as the internal connections and the remaining as external. The internal connections between the two sub-trees can all be made without crossings in this case. We traverse the list of internal connections until we reach a point in the list which is in between trees in both permutation forests. Each time we reach such a break point, we combine the trees on the sequences of external connections that have accumulated since the last break. If there is more than one tree then we combine them under a new node. We do this for both sets of external trees. The label for the roots of the external trees are determined as follows. If it is already labeled A, it remains that way. Otherwise, it is labeled F unless one of the sets of trees contained only one tree and was labeled A or R internally. Figure 6.12 contains two permutation forests and the permutation forest which is generated by Simplematch. There is one break point in this case just after the third and second trees respectively

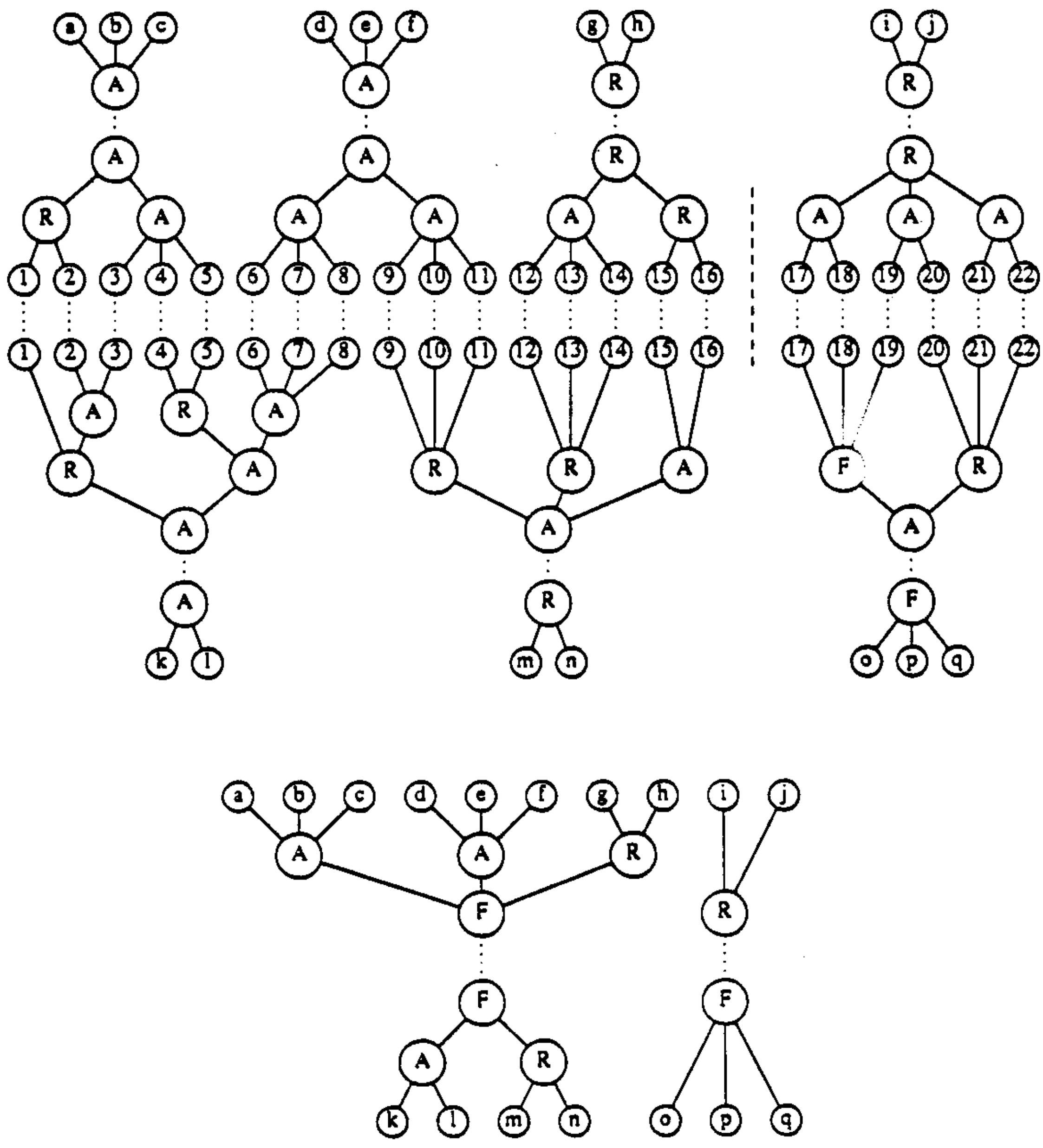
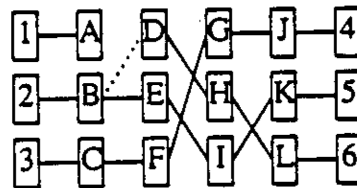


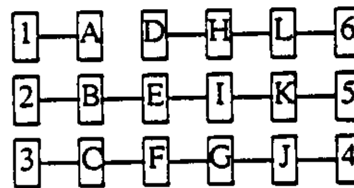
Figure 6.12 Resolving two adjacent Permutation Forests.

in the top and bottom forests. The first three trees on the exterior half of the top forest become sub-trees of a node labeled F as well as the first two trees of the exterior half of the bottom forest. This is because the order of these trees can no longer be permuted when the connections between the leaves of the corresponding

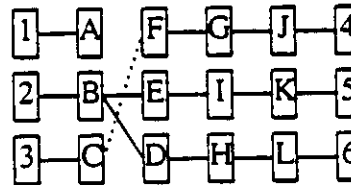
interior trees are made. The remaining exterior trees in both forests retain the same labeling for their roots.



a) Initial Ordering of Trees



b) Optimal Ordering of Trees



c) Final Ordering of Trees

Figure 6.13 Ordering the trees of Permutation Forests of a **Compose.**

Global-Align

In Global-Align we have an alternating sequence of permutation forests connected through R-nodes. We wish to select the best ordering of the trees for each permutation forest. We set up the following graph as in Figure 6.13a. The nodes correspond to the trees of the permutation forests and the nodes of each forest are arranged in a column. The forests are in left to right order corresponding to their order in the list of children of the **Compose**. The edges connect trees in adjacent columns and are weighted by the number of connections between the two trees. Nets

which involve more than three trees are not counted in this weighting. For each column we select the maximum set of edges such that each node has only one edge and remove the other edges. This amounts to finding a maximum pairing. Each node now has at most two edges, one on either side as in Figure 6.13a where the dotted edge between B and D has been removed. If all but one of the columns could be permuted then we could order the columns such that none of the edges cross as in Figure 6.13b. However the two sets of trees at either end can not be permuted. Thus we must find the smallest set of edges which can be removed to allow all but the terminal columns to be ordered such that none of the edges cross. To accomplish this, we use the same method as the one described in Section 6.1 for computing the cost of a bipartite R-node with partitions of size two. We build a conflict graph with each path which extends from the leftmost to the rightmost column as a node. The edges in this conflict graph, are between paths which do not cross and the weights of the nodes correspond to the cheapest edge on the path. We then find a maximum clique and remove the cheapest edges of the paths (nodes) which are not part of this clique. We can then order the columns so that the remaining edges do not cross. If this order is not a total order, there is some freedom in arranging the paths. In this case, any edges of the nodes of this path which were discarded before the pairing are considered in decreasing order of weight. This gives us the ordering for each of the internal permutation forests. In Figure 6.13c we removed the edge from C to F and then restored the edge from B to D.

Complexmatch

After applying Global-Align we assume that the order of the permutation forests is fixed. We will refer to them as permutation trees now since we could join

them with a node labeled F. Complexmatch is more involved than Simplematch since it must select permutations to minimize the intervening R-nodes. We try to find the maximum set of non-crossing pins as described in Section 6.1. The procedure is a heuristic one which proceeds hierarchically. It starts from the top of the permutation tree selecting an ordering for the children of both of the roots. The method used depends on the pair of labels of the two roots. There are six different cases, (F,F), (F,A), (F,R), (R,R), (R,A) and (A,A). There is nothing to decide in (F,F), (R,A) can be handled as (F,A) and (R,R) can be decided as (F,R). There are only three real cases that need to be handled, (F,A), (A,A) and (F,R). (F,R) is the easiest to solve. We simply try both ways, and pick the best one. The remaining cases are (F,A) and (A,A).

The problem reduces to the following. We have a bipartite graph whose two sets of nodes are to be arranged along the top and bottom sides of a box. There are edges between the nodes of differing weights, and our task is to select a subset of the edges of maximum weight such that the nodes can be ordered along the top and bottom sides of the box so that these edges do not cross. In one case both sets of nodes can reordered arbitrarily, while in the other case one set has a fixed ordering. We consider the former case first. We first observe that the subsets of edges for which the nodes can be reordered to avoid crossings induce a particular type of graph.

Definition 6.5

A *spine* is an undirected tree, in which each node has at most two neighbors which have degree greater than one. The nodes which have two such neighbors of degree greater than one, are called *vertebrae*.

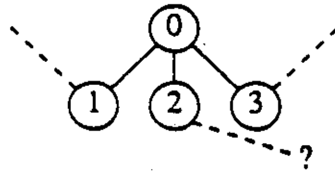


Figure 6.14 A tree which cannot be part of a spine.

It can be shown that a subgraph which can be arranged so that no two edges cross, must be a spine or a collection of spines. This is both a necessary and sufficient condition. To see this, suppose we have a node labeled 0, as in Figure 6.14, which has three neighbors of degree greater than one labeled 1, 2 and 3. Assume without loss of generality that node 0 sits on the top and nodes 1, 2 and 3 sit on the bottom such that 2 is between 1 and 3. Since 2 has degree two it must be connected to another node which must be placed in the top row. There is no way to do this without crossing the edges from node 0 to nodes 1 and 3 or drawing the edge outside the box. This means that the resulting subgraph can only consist of spines. Suppose on the other hand that we have a collection of spines. We can place each spine separately since they are not connected. The vertebrae of a spine form a path. We call the path consisting of vertebrae and the two nodes at either end of degree greater than one, the *backbone* of the spine. All other nodes of the spine are connected to a node of the backbone. To arrange the spine so that none of its edges cross, it suffices to place the backbone in an alternating 'zig-zag' fashion. Figure 6.15 contains a spine and its arrangement. To obtain the arrangement we started with the end of the backbone which is connected to node 1 and 2 and placed this node in the top row. We then traversed the backbone, placing its nodes by alternating between the top and bottom rows. The remaining all have degree one and are placed between their vertebra's two neighbors on the backbone, or at the end if they are their edge is to

one of the ends of a backbone.

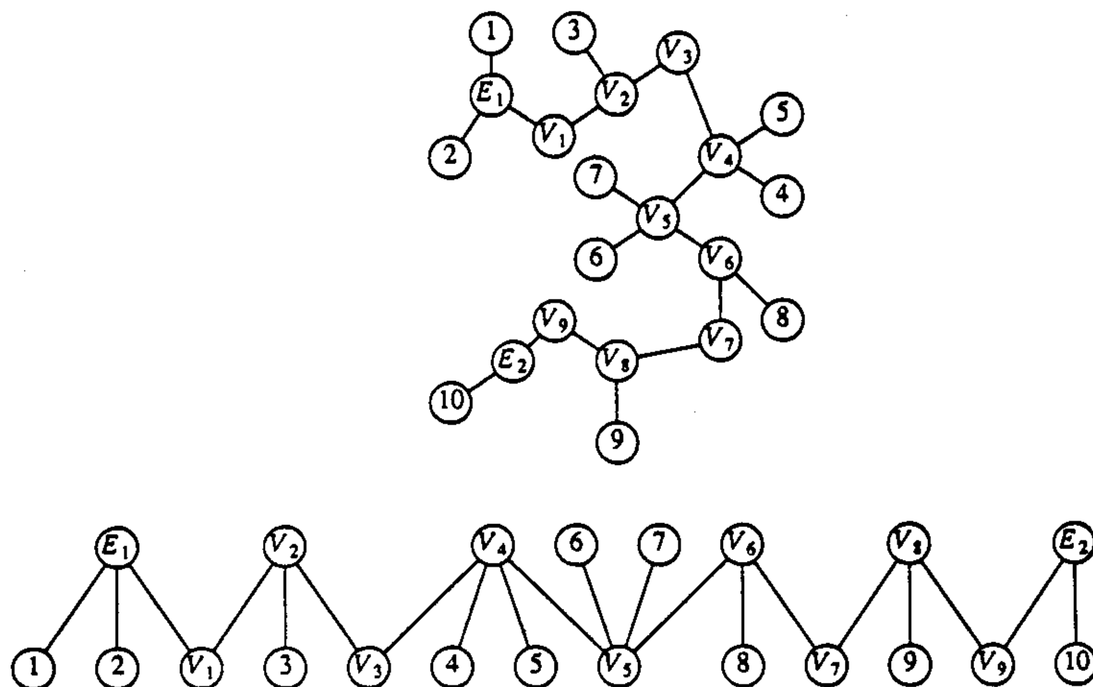


Figure 6.15 The arrangement of a Spine.

Unfortunately, computing a maximum collection of spines for a graph can be shown to be NP-complete. The problem is similar in nature to the longest path problem, to which the Hamiltonian Path and Circuit problems can be reduced. The construction given to reduce **Vertex Cover** to Hamiltonian Circuit in [Gare79] can be easily modified to show that finding the maximum spine in a bipartite graph is NP-hard. It is however easily computed when the graph is a tree. The approach we take is to compute a maximum spanning tree for each connected component of the graph and then obtain the maximum collection of spines from this spanning forest.

Computing a maximum spine in a tree proceeds as follows. There are four kinds of nodes in a collection of spines.

Vertebrae (V)

They have two neighbors with degree greater than 1.

Ends of Backbones (E)

These nodes have degree greater than one and have exactly one neighbor of degree greater than one.

Leaves (L)

These are nodes of a spine with degree one.

Free (F)

These are nodes of degree 0, trivial spines.

On a tree, the maximum collection of spines can be computed recursively. We pick a node arbitrarily to be the root of the tree and orient the edges to obtain a directed tree rooted at this node. We can then refer to the parent and children of a node, as well as the sub-tree rooted at a node. For each node, we compute the best collection of spines within the sub-tree rooted at the node in each of the four roles. This is easily computed from the same computation performed on its children. The value of the best spine tree for a leaf of the tree is 0 since it is the trivial spine containing no edges. Suppose we have a node u with children $u_1 \cdots u_n$ and we have computed the best spine collections for each of the sub-trees rooted at its children with each child in each of the four roles. Let e_i denote the weight of the edge connecting u to its ith child, u_i. We use the notation, $c(u,T)$ to denote the value of a best collection of spines in the sub-tree rooted at u in the role T. $c(u)$ is the maximum of the $c(u,T)$'s

over the four roles. We compute the $c(u,T)$'s for each role T as follows.

Free

If u is a free node, then we select the best solution for each child and this is the value of u as a free node.

$$c(u,F) = \sum_{k=1}^{n} c(u_k)$$

Vertebra

If u is to be a Vertebra, then two of its children must be either Vertebrae or Ends of Backbones. In addition, any of its children which are Free can become leaves. Suppose u_i and u_j are the two children which would become part of the backbone, then the cost of u as a Vertebra would be

$$c(u,V) = e_i + \max\{c(u_i,V), c(u_i,E)\} + e_k + \max\{c(u_j,V), c(u_j,E)\}$$

$$+ \sum_{\substack{k=1 \\ k \neq i,j}}^{n} \max\{c(u_k,F)+e_k, c(u_k)\}.$$

We must select i and j to maximize this value. To do this we compute two values for each u_m, its contribution if it is selected as one of the two spine nodes,

$$a_m = e_m + \max\{c(u_m,V), c(u_m,E)\}$$

and its contribution otherwise,

$$b_m = \max\{c(u_m,F)+e_m, c(u_m)\}.$$

We must then select i and j to maximize,

$$a_i + a_j + \sum_{\substack{k=1 \\ k \neq i,j}}^{n} b_k.$$

This is realized by selecting the i and j which maximize $a_m - b_m$. We record i and j in order to reconstruct the spine if u is assigned this role.

End of Backbone

In this case, one of u's children must be an End of Backbone, and will become a Vertebra. The rest can either become leaves or remain in their best state. We must select i to maximize,

$$c(u,E) = e_i + \max\{c(u_i,V), c(u_i,E)\} + \sum_{\substack{k=1 \\ k \neq i}}^{n} \max\{c(u_k,F)+e_k, c(u_k)\}.$$

The child u_i which maximizes this expression can be found in the same manner as in the previous case. This information is also retained to reconstruct the solution at the end. Technically we should require that at least one child be picked up as a leaf in this case, however there is no harm in considering u to be an End of Backbone instead of Leaf.

Leaf

For u to become a leaf one of its children must be a Vertebra. In this case we must select i to maximize,

$$c(u,L) = c(u_i,V) + e_i + \sum_{\substack{k=1 \\ k \neq i}}^{n} \max\{c(u_k,F)+e_k, c(u_k)\}.$$

In this manner, the maximum collection of spines for each role is computed from the leaves upward. Once we have computed these four values for the root, the

best collection of spines is extracted by assigning the root its best role, and traversing the tree from root to leaves assigning the roles and collecting the edges which will be part of the spines as follows. Suppose we have a node u to which we have assigned a role. If it is a Free node, then to each of its children we assign a role with maximum value. No edges are included. If u is a Vertebra, then two of its children were recorded and we assign them either Vertebra or End of Backbone whichever is highest. We include the edges to both of these children. The remaining children are assigned the role Free if $c(u_i,F)+e_i>c(u_i)$ and their best role otherwise. In the former case, the edges to these children are included. If u is an End of Backbone, then only one of its children is recorded and we assign it either the role Vertebra or End of Backbone whichever is highest and include the edge to this child. The remaining children are handled as in the case of the Vertebra. The last case is when u is a Leaf, the recorded child is assigned the role of a Vertebra, its edge is included and the remaining children are assigned their best roles.

The algorithm above gives the best collection of spines for a tree using time and space linear in the size of the tree, however our graphs will not in general be trees. Our approach was to find a maximum spanning tree and then apply the algorithm to this tree. Unless the maximum spanning tree happens to be a spine itself, we will end up with more than one spine. It may then be possible to add other edges which were not part of the maximum spanning tree, to the spines. We do this in a greedy manner, examining the remaining edges in order of decreasing value.

This leaves the case in which one side of the graph is fixed. In this case the solution can not just be any set of spines since the order of the nodes on one side is fixed. We solve this case by a greedy approach. We simply add edges in order of

decreasing weight.

The overall strategy in Complexmatch is a backtracking approach. We examine the first level of the trees on both side. We pretend that the connections are directly between the nodes at this level and obtain the best ordering using one of the methods described, depending on the labels of the roots. We then examine the subtrees of these nodes to see if in fact the connections which were assumed to be possible can in fact be made. This is achieved by applying Complexmatch again to nodes at this lower level in the permutation trees. If the number of connections is significantly less, we recompute the best solution at the level above using this value as the number of connections possible between the nodes.

Once we are satisfied with the set of connections and we record the selected configurations, and the procedure Simplematch is applied assuming only the connections in the maximum set are made. This is necessary to obtain the permutation forests representing the possible configurations of the external connections of the Compose.

The approach described in this section is comprised of heuristics. However, it provides the possibility of performing much more powerful transformations, particularly those afforded by Global-Align, than would be possible without the hierarchical representation.

Summary

The problem of improving the wiring complexity of a planar circuit has been examined in this chapter. Cost measures for planar circuits and the operations to be applied were discussed in Sections 6.1 and 6.2. The maximum number of non-

crossing connections which can be made within an R-node was selected as the cost measure, and operations which exchanged equivalent pins of circuit modules and exploited symmetries in the computation tree were considered. The problem of optimizing planar circuits by exchanging equivalent pins was shown to be NP-complete in Section 6.3. A method based on the computation tree was presented in Section 6.4.

CHAPTER 7

Conclusion

In the preceding chapters, we have formally defined the notion of the planar topology of layouts and implemented a mapping from behavioral specifications in a functional language, of circuits and their planar topology to abstract and actual layouts. Several examples of specifications and their layouts were presented. In addition, the optimization of the planar topology of these specifications was considered. In these sections we summarize the preceding chapters listing the contributions, and discuss some topics for future research.

7.1 Summary

The use of FP as a specification for circuits and their layouts has been investigated. The specification was assumed to provide the 'planar topology' of the layout, i.e., the planar organization of the circuit components and their interconnections. In using such a specification we have provided a constructive method for layout from a behavioral description. Since only the 'planar topology' of the layout is specified and the construction of the layout parallels the construction of the circuit in the behavioral description, this method provides a larger degree of geometric flexibility than other constructive methods.

The concept of the 'planar topology' of a layout was defined in Chapter 2. By a result of Edmond's it is sufficient to specify the cyclic ordering of edges around

nodes and the exterior window to uniquely specify the topology of an embedding in the plane of a graph. Circuits are not in general planar and are more often hypergraphs than graphs. Hence to define the 'planar topology' of a layout we introduced 'planar circuits', planar graphs which capture the branchings and crossings of the layout inside nodes. The variety of planar circuits which can be used to represent a layout and our desire to consider planar topology to be invariant under the movement of objects in the plane, motivated the introduction of a group of operations on planar circuits to simulate this movement and local reorganization of the wiring. We defined the 'planar topology' of a layout to be the equivalence class of planar circuits under these operations, containing the representatives of the layout. By placing conditions on the nodes of the planar circuit which represent its routing, we were able to define a normal form for planar circuits and show its uniqueness within its equivalence class modulo one operation. This result is useful since the layout of a planar circuit in normal form is optimal with respect to the layouts of planar circuits within its equivalence class, if the layout procedure satisfies certain assumptions.

In Chapter 3 we developed a mapping from FP expressions to planar circuits. Because unnecessary structure (circuit components and interconnections), can result from the literal interpretation of FP as a circuit, the mapping was divided into two steps. In the first phase, the literal interpretation of an FP expression resulted in a weak planar circuit. A weak planar circuit is a planar circuit with some of the connectivity requirements relaxed. We discussed the 'pruning' of weak planar circuits to remove this unnecessary structure. We showed that the maximal pruning of a planar circuit was unique and that under certain conditions (which are met by the weak planar circuits resulting from FP expressions) this pruning can be

performed given only the outputs which are required. This allowed us to decompose the pruning in terms of the combining forms of the FP function, providing an efficient method for directly computing the planar circuit of an FP expression, generating only the structures which would survive the pruning. We then extended this mapping to incorporate sequential circuits. This was accomplished by folding the space implementations of the combining forms **Apply-to-All**, **Right Insert** and **Seq** into time implementations using time-space transformations [Pate85]. The sequential versions of these combining forms use the same physical structure to perform the applications of their sub-function rather than creating a new structure for each application. The pruning of the sequential versions of these forms was given. The implementation of the mapping preserved the hierarchical structure afforded by the FP combining forms, by representing the planar circuit with a computation tree. In the last section we performed operations, corresponding to FP identities, on the tree to transform the planar circuit into normal form.

In Chapter 4, we addressed the problem of obtaining layouts from planar circuits. We defined an 'abstract layout' and showed that finding the 'abstract layout' with the smallest size was NP-hard. By exploiting the computation tree we synthesized a layout of planar circuits generated from FP expressions. In this procedure, the boxes and wires of the planar circuit were packed into horizontal cross-sections using the computation tree and then horizontal compaction was performed. In the last section we showed how to obtain actual layouts as well.

In Chapter 5, we presented several examples of FP specifications of circuits and 'abstract layouts' and an actual layout obtained from an FP expression to illustrate the features of using FP to describe circuits as well as the resulting layouts.

The ability of the system to provide graphical feedback through 'abstract layouts' at varying levels of abstraction during the synthesis of the design was demonstrated in these examples. The geometric flexibility in specifying layouts by their 'planar topology' was also apparent.

The problem of optimizing the planar topology of a layout by altering its planar circuit was addressed in Chapter 6. Cost measures for planar circuits were discussed and as well as the scope of the operations which should be considered. We selected a simple cost measure and showed the problem of optimizing it by switching equivalent pins alone (the simplest of the operations within the scope of operations allowed), to be NP-complete. However by exploiting the representation afforded by the FP expression (its computation tree), we provided a method to improve the planar circuit permitting more global operations than would be possible without such a representation.

We list the contributions of this thesis below.

1. A definition of the 'planar topology' of a layout in Section 2.4.

2. The proof of uniqueness (modulo refoldings) of a normal form for planar circuits which is optimal within its class, in Sections 2.5, 2.6 and 2.7.

3. A mapping from FP to combinational and sequential circuits in Sections 3.4 3.5 and 3.6. An implementation of this mapping and the transformations to put the planar circuit into normal form in Sections 3.7 and 3.8.

4. The proof of the NP-hardness of obtaining a minimum size layout from a planar circuit in Section 4.2.

5. A method for synthesizing the layout of planar circuits generated from FP expressions exploiting the computation tree in Sections 4.3, 4.4 and 4.5.

6. The implementation of these methods on DEC VAX 11/750; a system producing both 'abstract layouts' and actual layouts from FP expressions.

7. The proof of the NP-completeness of maximizing the alignment of a planar circuit by exchanging equivalent pins in Section 6.3.

8. A method for improving both the alignment of planar circuits by not only exchanging equivalent pins, but by exploiting the symmetries represented by the combining forms of the computation tree in Section 6.4.

With this work we hope to have provided a new method for obtaining layouts by recognizing that behavioral specifications can provide valuable information about the structure of the circuit and its embedding. By specifying the planar topology of the circuit we did not avoid the intractable problems of layout. We have dealt with these problems by exploiting the organization of the planar topology provided by the behavioral specification, information which a combinatorial method would have to discover.

7.2 Future Research

The topics for future research which come to mind fall into two categories, enhancements to the current system design environment and new areas of research that should be explored.

The following enhancements to the system producing layouts from FP expressions should be considered.

Routing of Power and Ground

The routing of power and ground could be incorporated within an FP specification as in the Nor and Nand Decoders in the Section 4.1. However this might be awkward in higher level algorithms and it is not always desirable for the routing of power and ground to flow in the same direction as the data. Automatic routing of power and ground possibly based on the computation tree would remove the last step necessary to obtain the layout.

Two Dimensional Data Flow

The packing of the planar circuits described in Chapter 4 requires that the inputs of each box enter from the top and that the outputs exit from the bottom. A more two-dimensional approach would allow the inputs and outputs to enter and exit form the sides as well. The layout of the horizontal versions of the combining forms **Right Insert** and **Seq** would benefit and new combining forms might also exploit this. An alternate form of the **Compose** which folds itself like a snake to meet an aspect ratio as described in [Leis80] is another possibility.

Context Sensitive Implementations of the *R*-nodes

The method which implements the *R*-nodes attempts to minimize the number of horizontal tracks used without regard to the actual positions of the boxes to which these wires are connected. As shown in Figure 7.1, this sometimes leads to a 'poor routing.' In the example in the Figure 7.1, the *R*-node has one wire (the solid wire) which needs to cross four other wires. This *R*-node was implemented with the minimum number of tracks by using one track to route this wire across the other four. However in this case, because of other

routing, six tracks were available and the four wires ended up with horizontal jogs to the left. The solid wire could have been routed straight down across all four of them providing a 'better routing.' It would be nice to predict such occurrences. However, it seems difficult to do so without performing the horizontal compaction. This problem could be alleviated by providing some form of feedback to the procedure which implemented the R-node or by doing some local optimizations on the wiring after the compaction.

Figure 7.1

The following two areas of research are motivated by the results in this thesis.

Complexity Classes of Planar Circuits

The 'chip complexity' of an algorithm refers to the time and area required to perform the computation in an integrated circuit. Typically a lower bound on area is shown by counting the number of wires which must cross a particular line drawn across the chip [Vuil83]. Hence the bound on the area is a result of the wiring complexity. It may be possible to obtain tighter bounds using planar circuits since the routing is completely characterized by the R-nodes. In particular it may be interesting to explore the lower bounds in terms of the size of R-nodes required to perform computations.

Algebraic Transformations

One of the reasons for selecting FP as the specification language was its algebraic nature which provides transformations preserving the behavior while altering the structure of the FP function and accordingly, its layout. Now that we have a mapping from FP to 'planar topology' it is possible to assess the consequences of these transformations. We have exploited some of the simpler transformations in the preceding chapters, however there may be many more to discover and exploit. A transformation system should be provided and the degree to which it can be automated to improve the layout and satisfy constraints should explored. Incorporating the laws of boolean algebra should provide even more powerful transformations at the switching expression level.

References

[Ance83] Anceau, F., "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits Specified by Algorithms," pp. 15-31 in *Third CALTECH Conference on Very Large Scale Integration*, ed. R. Bryant, Computer Science Press, Rockville, Maryland (1983).

[Back78] Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM, Turing Award Lecture* **21**(8), pp.613-641 (August 1978).

[Bren80] Brent, R. P. and H. T. Kung, "The Chip Complexity of Binary Arithmetic," pp. 190-200 in *Proceedings 12th ACM Symposium on the Theory of Computing* (May 1980).

[Dole81] Dolev, D. and H. Trickey, "On linear embedding of planar graphs," (1981). unpublished manuscript see [Jo82].

[Dona80] Donath, W. E., "Complexity Theory and Design Automation," pp. 412-419 in *Proceedings 17th Design Automation Conference*, Minneapolis, Minnesota (June 1980).

[Edmo60] Edmonds, J. R., "A combinatorial representation for polyhedral surfaces," *American Mathematical Society Notices*(7), p.646 (1960).

[Even72] Even, S. and A. Pnueli, "Permutation Graphs and Transitive Graphs," *Journal of the Association of Computing Machinery* **19**(3), pp.400-410 (July 1972).

[Fode80] Foderaro, J. K., "The Franz Lisp Manual," , University of California, Berkeley, Berkeley, California (1980).

[Gajs85] Gajski, Daniel D., "Silicon Compilation," *VLSI Systems Design*, pp.48-63 (November 1985).

[Gare79] Garey, Michael R. and David S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, CA (1979).

[Gopa83] Gopal, I. S., D. Coppersmith, and C. K. Wong, "Optimal Wiring of Movable Terminals," *IEEE Transactions on Computers* C-32(9), pp.845-858 (Septmeber 1983).

[Joha79] Johannsen, D., "Bristle Blocks: A Silicon Compiler," pp. 310-313 in *Proceedings 16th Design Automation Conference*, San Diego, California (June 1979).

[John84] Johnson, Steven, in *Synthesis of Digital Designs from Recursion Equations*, MIT Press, Boston, Mass (1984).

[Karp83] Karplus, K., "CHISEL An extension to the Programming Language C for VLSI Layout," STAN-CS-82-959, Stanford University, Stanford, California (February 1983).

[Kern78] Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey (1978).

[Kirk83] Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science* 220(4598), pp.671-680 (May 1983).

[Kirk84] Kirkpatrick, S., "Optimization by Simulated Annealing: Quantitative Studies," *Journal of Statistical Physics* 34(5&6), pp.975-986 (1984).

[Laht81] Lahti, D. O., "Applications of a Functional Programming Language," CSD-810403, UCLA, Los Angeles, California (April 1981).

[Leis80] Leiserson, C. E., "Area-Efficient Graph Layouts (for VLSI)," pp. 270-281 in *Proceedings 21st IEEE Symposium on Foundations of Computer Science* (1980).

[Liao83] Liao, Y. Z. and C. K. Wong, "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* CAD-2(2), pp.62-69 (April 1983).

[Lich82] Lichtenstein, David, "Planar Formulae and Their Uses," *SIAM Journal of Computing* 11(2), pp.329-343 (May 1982).

[Lieb83] Lieberherr, K. J. and S. E. Knudsen, "ZEUS: A Hardware Description Language for VLSI," pp. 17-23 in *Proceedings 20th Design Automation Conference*, Miami Beach, Florida (June 1983).

[Lipt82] Lipton, R. J., S. C. North, R. Sedgewick, J. Valdes, and G. Vijayan, "ALI: a Procedural Language to Describe VLSI Layouts," pp. 467-474 in *Proceedings 19th Design Automation Conference*, Las Vegas, Nevada (June 1982).

[Mare84] Marek-Sadowska, M., "An Unconstrained Topological Via Minimization Problem for Two-Layer Routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* CAD-3(3), pp.184-190 (July 1984).

[McCa60] McCarthy, J., "Recursive Functions of Symbolic expressions and their Computation by Machine," *Communications of the ACM* 3(4), pp.184-195 (April 1960).

[Mead80] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts (1980).

[Mesh84] Meshkinpour, F., "On Specification and Design of Digital Systems Using an Applicative Hardware Description Language," MS Thesis, UCLA, Los Angeles, California (1984).

[Newm42] Newman, M. H. A., "On Theories with a Combinatorial Definition of 'Equivalence'," *Annals of Mathematics* 43(2), pp.223-243 (April 1942).

[Oust83] Ousterhout, J. K, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "A Collection of Papers on Magic," UCB/CSD 83/154, University of California, Berkeley, Berkeley, California (December 1983).

[Park80] Parker, D. S., "Note on Shuffle/Exchange-Type Switching Networks," *IEEE Transactions on Computers* C-39(3), pp.213-222 (March 1980).

[Pate85] Patel, D., M. Schlag, and M. Ercegovac, "vFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms," pp. 238-255 in *Functional Programming Languages and Computer Architecture*, ed. J.P. Jouannaud, Springer-Verlag Lecture Notes in Computer Science, Nancy, France (September 1985).

[Rees83] Rees, Jonathan A., Norman I. Adams, and James R. Meehan, "The T Manual," , Yale University, New Haven, Connecticut (March 1983).

[Roge86] Rogers, C. D., "The VIVID Symbolic Design System: Current Overview and Future Directions," *IEEE Design & Test*, pp.75-81 (February 1986).

[Sahn80] Sahni, S. and A. Bhatt, "The Complexity of Design Automation Problems," pp. 402-411 in *Proceedings 17th Design Automation Conference*, Minneapolis, Minnesota (June 1980).

[Schl83] Schlag, M., Y. Z. Liao, and C. K. Wong, "An Algorithm for Optimal Two Dimensional Compaction of VLSI Layouts," *Integration, the VLSI Journal* 1(2 & 3), pp.179-209 (October 1983).

[Schl84] Schlag, M. D. F., L. S. Woo, and C. K. Wong, "Maximizing Pin Alignment by Pin Permutations," *Integration, the VLSI Journal* 2, pp.279-307 (1984).

[Schl85] Schlag, M. D. F., E. J. Yoffa, P. S. Hauge, and C. K. Wong, "A Method for Improving Cascode-Switch Macro Wirability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* CAD-4(2), pp.150-155 (April 1985).

[Shee84] Sheeran, M., "muFP, a language for VLSI design," pp. 104-112 in *Proceedings ACM Symposium on LISP and Functional Programming* (1984).

[Vali81] Valiant, L. G., "Universality Considerations in VLSI Circuits," *IEEE Transactions on Computers* C-30(2), pp.135-140 (February 1981).

[Vuil83] Vuillemin, J., "A Combinatorial Limit to the Computing Power of VLSI Circuits," *IEEE Transactions on Computers* C-32(3), pp.294-300 (March 1983).

[Will78] Williams, J., "STICKS- A Graphical Compiler for High Level LSI Design," pp. 289-295 in *Proceedings 1978 National Computer Conference* (May 1978).

[Youn63] Youngs, J. W. T., "Minimal Imbeddings and the Genus of a Graph," *Journal of Mathematics and Mechanics* 12(2), pp.303-315 (March 1963).

Appendix: Description of FP

Objects

The set of objects Ω consists of the atoms and sequences $<x_1, x_2, \ldots, x_k>$ (where the $x_i \in \Omega$). (Lisp users should note the similarity to the list structure syntax, just replace the parenthesis by angle brackets and commas by blanks. There are no 'quoted' objects, i.e., 'abc). The atoms uniquely determine the set of valid objects and consist of the numbers (of the type found in FRANZ LISP [Fode80]), quoted ascii strings ("abcd"), and unquoted alphanumeric strings (abc3). There are three predefined atoms, **T** and **F**, that correspond to the logical values 'true' and 'false', and the undefined atom $\perp$, *bottom*. *Bottom* denotes the value returned as the result of an undefined operation, e.g., division by zero. The empty sequence, <> is also an atom. The following are examples of valid FP objects:

$\perp$	1.47	3888888888888
ab	"*CD*"	$<1,<2,3>>$
<>	**T**	$<a,<>>$

There is one restriction on object construction: no object may contain the undefined atom, such an object is itself undefined, e.g., $<1,\perp> \equiv \perp$. This property is the so-called bottom preserving property [Back78].

Application

This is the single FP operation and is designated by the colon (":"). For a function ϕ and an object x, $\phi{:}x$ is an application and its meaning is the object that results from applying ϕ to x (i.e., evaluating $\phi(x)$). We say that ϕ is the *operator* and that x is the *operand*. The following are examples of applications:

$$+{:}<7,8> \equiv 15 \qquad \mathbf{tl}{:}<1,2,3> \equiv <2,3>$$

$$\mathbf{1}{:}<a,b,c,d> \equiv a \qquad \mathbf{2}{:}<a,b,c,d> \equiv b$$

Functions

All functions (F) map objects into objects, moreover, they are *strict*:

$$\phi : \bot \equiv \bot, \ \forall \phi \in \mathbf{F}$$

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expression: [McCa60]

$$p_1 \rightarrow e_1 ; \cdots ; p_n \rightarrow e_n ; e_{n+1}$$

This statement is interpreted as follows: return function e_1 if the predicate 'p_1' is true , . . . , e_n if 'p_n' is true. If none of the predicates are satisfied then default to e_{n+1}. It is assumed that x, x_i, y, y_i, $z_i \in \Omega$.

Selector Functions

For a nonzero integer μ,

$$\mu : x \equiv$$
$$x{=}<x_1, x_2, \ldots, x_k> \wedge 0 < \mu \leq k \rightarrow x_\mu;$$
$$x{=}<x_1, x_2, \ldots, x_k> \wedge -k \leq \mu < 0 \rightarrow x_{k+\mu+1}; \bot$$

The user should note that the function symbols **1,2,3,...** are to be distinguished from the atoms 1,2,3,....

last : $x \equiv$

x=<> $\rightarrow$ <> ;

x=<$x_1, x_2, \ldots, x_k$> $\wedge\ k \geq 1 \rightarrow x_k ; \perp$

first : $x \equiv$

x=<> $\rightarrow$ <> ;

x=<$x_1, x_2, \ldots, x_k$> $\wedge\ k \geq 1 \rightarrow x_1 ; \perp$

Tail Functions

tl : $x \equiv$

x=<x_1> $\rightarrow$ <> ;

x=<$x_1, x_2, \ldots, x_k$> $\wedge\ k \geq 2 \rightarrow$ <$x_2, \ldots, x_k$> ; $\perp$

tlr : $x \equiv$

x=<x_1> $\rightarrow$ <> ;

x=<$x_1, x_2, \ldots, x_k$> $\wedge\ k \geq 2 \rightarrow$ <$x_1, \ldots, x_{k-1}$> ; $\perp$

Note: There is also a function **front** that is equivalent to **tlr**.

Distribute from left and right

distl : $x \equiv$

x=<y, <>> $\rightarrow$ <>;

x=<y, <$z_1, z_2, \ldots, z_k$>> $\rightarrow$ <<y, z_1>,...,<y, z_k>>; $\perp$

distr : $x \equiv$

x=<<>,y> $\rightarrow$ <>;

x=<<$y_1, y_2, \ldots, y_k$>,z> $\rightarrow$ <<y_1, z>,...,<y_k, z>>; $\perp$

Identity

id : $x \equiv x$

out : $x \equiv x$

Out is similar to **id**. Like **id** it returns its argument as the result, unlike **id** it prints its result on *stdout* – It is the only function with a side effect. *Out* is intended to be used for debugging only.

Append left and right

apndl : $x \equiv$

x=<y, <>> → <y>;

x=<y, <z_1, z_2, . . . , z_k>> → <y,z_1, z_2, . . . , z_k>; ⊥

apndr : $x \equiv$

x=<<>,z> → <z>;

x=<<y_1, y_2, . . . , y_k>,z> → <y_1, y_2, . . . , y_k, z>; ⊥

Transpose

trans : $x \equiv$

x=<<>,...,<>> → <>;

x=<x_1, x_2, . . . , x_k> → <y_1, . . . ,y_m>; ⊥

where x_i = <$x_{i,1}$, . . . ,$x_{i,m}$> $\wedge$ y_j = <$x_{1,j}$, . . . ,$x_{k,j}$>, $1 \le i \le k$, $1 \le j \le m$.

reverse : $x \equiv$

x=<> →;<>

x=<x_1, x_2, . . . , x_k> → <x_k, . . . ,x_1>; ⊥

Rotate Left and Right

rotl : $x \equiv$

x=<> → <>; x=<x_1> → <x_1>;

x=<x_1, x_2, . . . , x_k> $\wedge$ $k \ge 2$ → <x_2, . . . ,x_k,x_1>; ⊥

rotr : $x \equiv$

x=<> → <>; x=<x_1> → <x_1>;
x=<$x_1, x_2, \ldots, x_k$> ∧ $k \geq 2$ → <$x_k, x_1, \ldots, x_{k-2}, x_{k-1}$>; ⊥

concat : $x \equiv$

x=<<$x_{11}, \ldots, x_{1k}$>, . . . , <$x_{m1}, \ldots, x_{mp}$>> ∧ k, m, n, $p > 0$
→ <$x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mp}$>; ⊥

Concatenate removes all occurrences of the null sequence:

concat : <<1,3>,<>,<2,4>,<>,<5>> ≡ <1,3,2,4,5>

pair : $x \equiv$

x=<$x_1, x_2, \ldots, x_k$> ∧ k>0 ∧ *k is even*
→ <<x_1, x_2>, . . . , <x_{k-1}, x_k>>;
x=<$x_1, x_2, \ldots, x_k$> ∧ k>0 ∧ *k is odd*
→ <<x_1, x_2>, . . . , <x_k>>; ⊥

split : $x \equiv$

x=<x_1> → <<x_1>,<>>;
x=<$x_1, x_2, \ldots, x_k$> ∧ k>1
→ <<$x_1, \ldots, x_{\lceil k/2 \rceil}$>,<$x_{\lceil k/2 \rceil+1}, \ldots, x_k$>>; ⊥

Predicate (Test) Functions

atom : $x \equiv x \in$ *atoms* → T; $x \neq \bot$ → F; ⊥

eql : $x \equiv x$ =<y,z> ∧ $y=z$ → T; x=<y,z> ∧ $y \neq z$ → F; ⊥

Also less than (<), greater than (>), greater than or equal (>=), less than or equal (<=), not equal (~=);'=' is a synonym for eql.

null : $x \equiv x$=<> → T; $x \neq \bot$ → F; ⊥
length : $x \equiv x$ = <$x_1, x_2, \ldots, x_k$> → k; x=<> → 0; ⊥

Predicate operators, And, or, not, xor

and : $<x,y> \equiv x=\mathbf{T} \rightarrow y\,;\, x=\mathbf{F} \rightarrow \mathbf{F};\bot$

or : $<x,y> \equiv x=\mathbf{F} \rightarrow y\,;\, x=\mathbf{T} \rightarrow \mathbf{T};\bot$

not : $x \equiv x=\mathbf{T} \rightarrow F\,;\, x=\mathbf{F} \rightarrow \mathbf{T};\bot$

xor : $<x,y> \equiv$

$x=\mathbf{T} \wedge y=\mathbf{T} \rightarrow \mathbf{F};\, x=\mathbf{F} \wedge y=\mathbf{F} \rightarrow \mathbf{F};$
$x=\mathbf{T} \wedge y=\mathbf{F} \rightarrow \mathbf{T};\, x=\mathbf{F} \wedge y=\mathbf{T} \rightarrow \mathbf{T};\bot$

Arithmetic/Logical

$+ : x \equiv x=<y,z> \wedge y,z$ are numbers $\rightarrow y+z\,;\bot$

$- : x \equiv x=<y,z> \wedge y,z$ are numbers $\rightarrow y-z\,;\bot$

$* : x \equiv x=<y,z> \wedge y,z$ are numbers $\rightarrow y \times z\,;\bot$

$/ : x \equiv x=<y,z> \wedge y,z$ are numbers $\wedge\, z \neq 0 \rightarrow y+z\,;\bot$

Circuit Primitives

andg : $<x,y> \equiv$

$x=1 \wedge y=1 \rightarrow 1;\, x=0 \wedge y=0 \rightarrow 0;$
$x=1 \wedge y=0 \rightarrow 0;\, x=0 \wedge y=1 \rightarrow 0;\bot$

org : $<x,y> \equiv$

$x=1 \wedge y=1 \rightarrow 1;\, x=0 \wedge y=0 \rightarrow 0;$
$x=1 \wedge y=0 \rightarrow 1;\, x=0 \wedge y=1 \rightarrow 1;\bot$

xorg : $<x,y> \equiv$

$x=1 \wedge y=1 \rightarrow 0;\, x=0 \wedge y=0 \rightarrow 0;$
$x=1 \wedge y=0 \rightarrow 1;\, x=0 \wedge y=1 \rightarrow 1;\bot$

nandg : $<x,y> \equiv$

$x=1 \wedge y=1 \rightarrow 0;\, x=0 \wedge y=0 \rightarrow 1;$
$x=1 \wedge y=0 \rightarrow 1;\, x=0 \wedge y=1 \rightarrow 1;\bot$

norg : $<x,y> \equiv$

$x=1 \wedge y=1 \rightarrow 0;\, x=0 \wedge y=0 \rightarrow 1;$
$x=1 \wedge y=0 \rightarrow 0;\, x=0 \wedge y=1 \rightarrow 0;\bot$

notg : $x \equiv x=1 \rightarrow 0;\, x=0 \rightarrow 1;\bot$

Library Routines

sin : $x \equiv x$ is a number $\rightarrow sin(x); \bot$

asin : $x \equiv x$ is a number $\wedge\ |x| \leq 1 \rightarrow \sin^{-1}(x); \bot$

cos : $x \equiv x$ is a number $\rightarrow cos(x); \bot$

acos : $x \equiv x$ is a number $\wedge\ |x| \leq 1 \rightarrow \cos^{-1}(x); \bot$

exp : $x \equiv x$ is a number $\rightarrow e^{x}; \bot$

log : $x \equiv x$ is a positive number $\rightarrow ln(x); \bot$

mod : $<x,y> \equiv xndy$ are numbers $\rightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor ; \bot$

sqrt : $x \equiv x$ is a number $\rightarrow \sqrt{x}\,; \bot$

Combining Forms

Combining forms define new *functions* by operating on function and object *parameters* of the form. The resultant expressions can be compared and contrasted to the *value*-oriented expressions of traditional programming languages. The distinction lies in the domain of the operators; combining forms manipulate functions, while traditional operators manipulate values.

One combining form is **Compose.** For two functions ϕ and ψ the form ϕ @ ψ denotes their composition $\phi \circ \psi$:

$$(\phi\ @\ \psi) : x \equiv \phi{:}(\psi{:}x), \quad \forall\ x \in \Omega$$

The *constant* function takes an object parameter:

$$\%x : y \equiv y = \bot \rightarrow \bot;\ x, \quad \forall\ x,y \in \Omega$$

The function $\%\bot$ always returns $\bot$.

In the following description of the combining forms, we assume that ξ, ξ_i, ϕ, ϕ_i, τ, and τ_i are functions and that x, x_i, y are objects.

Compose

$(\phi\ @\ \tau):x \equiv \phi:(\tau:x)$

Construct

$[\phi_1, \ldots, \phi_n]:x \equiv <\phi_1:x, \ldots, \phi_n:x>$

Note that construction is also bottom-preserving, e.g.,

$[+,/]:<3,0> = <3,\bot> = \bot$

Apply-to-All

$\&\,\phi: x \equiv$
$x=<> \rightarrow <>;$
$x=<x_1, x_2, \ldots, x_k> \rightarrow <\phi:x_1, \ldots, \phi:x_k>; \bot$

Conditional

$(\xi \text{ -> } \phi; \tau):x \equiv$
$(\xi:x)=\mathbf{T} \rightarrow \phi:x;$
$(\xi:x)=\mathbf{F} \rightarrow \tau:x; \bot$

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy's conditional expression, and the *combining form* introduced here. In the former case the result is a *value*, while in the latter case the result is a *function*. Unlike Backus' FP, the conditional form *must* be enclosed in

parenthesis, e.g.,

(isNegative -> - @ [%0,id] ; id)

Constant

$\%x : y \equiv y = \perp \rightarrow \perp ; x, \quad \forall x \in \Omega$

This function returns its object parameter as its result.

Right Insert

$!\phi : x \equiv$

$x = <> \rightarrow e_f : x;$

$x = <x_1> \rightarrow x_1;$

$x = <x_1, x_2, \ldots, x_k> \wedge k \geq 2 \rightarrow \phi : <x_1, !\phi : <x_2, \ldots, x_k>>; \perp$

e.g., !+:<4,5,6>=15.

If ϕ has a right identity element e_f, then $!\phi : <> = e_f$, e.g.,

!+ : <>=0 and !* : <>=1

Currently, identity functions are defined for + (0), - (0), * (1), / (1), also for **and** (1), **or** (0), **xor** (0). All other unit functions default to bottom ($\perp$).

Seq

$seq(\phi) : x \equiv$

$x = <> \rightarrow e_f : x;$

$x = <x_1> \rightarrow x_1;$

$x = <x_1, x_2> \rightarrow \phi : <x_1, x_2>;$

$x = <x_1, x_2, \ldots, x_k> \wedge k > 2 \rightarrow <y_1, \ldots, y_k>; \perp$

where $<z_2, y_3, \ldots, y_k> = seq(\phi) : <x_2, \ldots, x_k>$

and $<y_1, y_2> = \phi : <x_1, z_2>$

User Defined Functions

An FP definition is entered as follows:

{fn-name fn-form},

where *fn-name* is an ascii string consisting of letters, numbers and the underline symbol, and *fn-form* is any valid combining form, including a single primitive or defined function. For example, the functions

{factorial (zero?→%1; *@[id,factorial@-@[id,%1]])}

{zero? eql@[id,%0]}

form a recursive definition of the factorial function. Since FP systems are applicative it is permissible to substitute the actual definition of a function for any reference to it in a combining form: if $f \equiv 1@2$ then $f : x \equiv 1@2 : x, \quad \forall x \in \Omega$.

References to undefined functions bottom out:

$f : x \equiv \bot \forall x \in \Omega, f$ **F**