The Planar Topology of Functional Programs

Martine Schlag

Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

Abstract

The use of the applicative language (FP) in VLSI design has been advocated, because it provides not only the structure of a circuit, but the planar organization of its components and their interconnections. In this paper, the level of geometric detail implied by the functional programming style is formalized. The notion of 'planar topology' of an integrated circuit layout is defined and shown to be the appropriate level of geometric information to infer from an FP specification of a circuit. This definition provides a normal form for the representation of the planar topology of a layout which is not only unique (modulo local operations), but is optimal over all representations of the same planar topology with respect to topological cost measures. This normal form is exploited to improve the wiring of the layouts; it is realized by the application of transformations to the FP specification. The specification of a carry-save array multiplier is used as an example to illustrate how this optimization reduces the effort required to specify an integrated circuit.

1. Introduction

The current complexity of VLSI design can only be managed by the application of CAD tools at all levels of the design process. In order to be effective, these tools must be flexible enough to be tailored to any specific design. In bottom-up design, the designer is faced with the dilemma of using tools which require the complete specification of the placement of modules and the interconnections between them [Oust81, Oust84], or relinquishing control over the layout to a tool's algorithm [Rive82]. Top-down synthesis tools are capable of generating layouts from high-level specifications. Examples would include various register-transfer silicon compilers that have been proposed and built [Joha79, Sisk82]. Generally, neither type of tool provides any estimate of the area or delays of the circuit during the synthesis process. That is, designers do not know the effects of their decisions on the performance until the design is complete.

A compromise between the complete specification of an integrated circuit and the synthesis of the layout by a tool, is to use a specification which provides an intermediate level of geometric detail of the layout. Several researchers [Shee84, Pate85] have advocated the use of the applicative language (FP) [Back78] in VLSI design, precisely because it provides not only the structure of a circuit, but the planar organization of the components and their interconnections. By exploiting the geometric information embedded in FP specifications, an environment in which designers can rapidly explore various

alternative designs for their algorithms, can be provided [Pate85]. An algorithm can be specified at any arbitrary level of abstraction and the system rapidly evaluates performance parameters (e.g. speed, area, etc.) so that designers can make informed decisions during the synthesis process. Using an applicative language to specify a hardware algorithm, ties together the specification of the algorithm, the synthesis of the circuit and the evaluation of the implementation.

In this paper, the level of geometric detail implied by the functional programming style is formalized. The notion of 'planar topology' of an integrated circuit layout is defined and shown to be the appropriate level of geometric information to infer from an FP specification of a circuit. This definition provides a normal form for the representation of the planar topology of a layout which is unique (modulo local operations), and optimal over all representations of the same planar topology with respect to topological cost measures. This normal form is realized by the application of transformations to the FP specifications.

2. VLSI Design in FP

FP as proposed in [Back78] is not suitable for specifying sequential circuits due to it's lack of state. Meshkinpour [Mesh85] and Sheeran's μFP [Shee84] extend Backus' FP language with operators to handle sequential circuits. A different approach is taken in νFP [Pate85]. νFP extends the FP language proposed by Backus with additional functional forms and primitives. In contrast to μFP [Shee84], which extends FP's semantics to operate on streams, the semantics of νFP are the same as those of FP when it is used to specify algorithms. Sequential circuits are described in νFP by folding designated functional forms to implement them in time (as sequential circuits) rather than in space (as combinational circuits).

A program in νFP (as in FP) is a function that maps objects into objects. Objects are either atomic (numbers or strings) or sequences of objects. The distinguished atom $\perp$ denotes an undefined value. By definition, any sequence which contains $\perp$ as an element is itself undefined and thus equal to $\perp$. The primitive functions of νFP consist of

arithmetic functions,	+ : (1,5) → 6	* : (3,2) → 6
logical functions,	nandg : (1,0) → 1	org : (0,0) → 0
predicates,	atom : (1,2) → F	= : (3,3) → T
selector functions,	3 : (2,(4,5),6,(8,(9,10))) → 6	last : (1,4,6) → 6

and structure modifying functions,

trans : ((1,2,3),(4,5,6)) → ((1,4),(2,5),(3,6)) apndl : (1,(2,3,4)) → (1,2,3,4)
distl : (x, (a,b,c)) → ((x,a),(x,b),(x,c)) distr : ((a,b,c),x) → ((a,x),(b,x),(c,x)).

Functional forms are used to combine primitive functions into more complex functions.

compose	(f @ g) : x → f : (g : x)	**apply to all**	&f : $(x_1,...,x_n)$ → (f:x_1,...,f:x_n)
construct	[f,g,h] : x → (f:x, g:x, h:x)	**right insert**	!f : $(x_1,..,x_n)$ → f:(x_1, !f:$(x_2,...,x_n)$)
constant	%k : x → k if x is not $\perp$	**seq**	seq(f) : $(x_1,..,x_n) \to (y_1,z_1,\dots,z_{n-1})$ where $(y_2,z_2,\dots,z_{n-1})$ = seq(f) : $(x_2,..,x_n)$ and (y_1,z_1) = f:(x_1,y_2)

Owing to the natural specification of parallelism in FP-like languages, they are suited to describing parallel hardware algorithms. These specifications (or programs) are executable. In addition, since such programs are *referentially transparent*, it is possible to have an algebra of programs which may be used to reason about their behavior. Either method may be used in conjunction with one another to convince the designer that the program implements the envisioned algorithm. Specifications can also be executed with a symbolic input to extract the topological structure of the algorithm. Therefore, there is a direct relationship between the structure of an algorithm written in FP and the planar topology of its layout.

3. The Level of Geometry Afforded by FP

Each functional form of an FP function implies the planar organization of the circuitry of its sub-functions. More formally, a function is a sub-graph with an input arc labeled with its input object and an output arc labeled correspondingly with its output object. Each functional form combines the sub-graphs of its sub-functions to form a new sub-graph. This may entail adding nodes which distribute and collect inputs and outputs as dictated by the particular functional form. The planar embedding of the graph of a functional form is obtained from the planar embeddings of the graphs corresponding to its the sub-functions. The graph and its embedding is obtained by the symbolic execution of the FP program. The graph on the left in Figure 1 is extracted in this fashion from its FP program.

To obtain a circuit from this planar graph, circuit elements (or sub-circuits) are substituted for the nodes and the arcs connecting the nodes are expanded. Each atom (null is not an atom) in an object will be given its own wire in the expansion of an arc. Since only one object is passed between functions, the organization of connections around function boundaries is straightforward. Each arc along which an object passed is expanded into a group of wires, one wire for each atom in the object. The ordering of these wires is inherited from the list structure of the object. The graph on the right in Figure 1 is obtained from the graph on the left by expanding its arcs. In this fashion, a planar embedding of the structure of the application of an FP program is obtained. FP programs yield planar graphs because crossings and branchings (fanout) are hidden inside nodes.

To obtain a layout from this planar graph the appropriate circuit structures (components and wiring) are be substituted for its nodes. In top-down design the exact dimensions of these structures may not be known until the final stages in the synthesis of the design. Since the placement of a structure is dependent on its dimensions as well as those of neighboring structures, only the planar topology of the FP program remains fixed as the design is refined during its synthesis. Unfortunately, interpreting the wiring implied by this graph by directly substituting the wiring required by its nodes results in either inefficient layouts or complex specifications since detailed attention must be given to the exact wiring patterns being generated. This is further complicated by uncertainty about the dimensions of components which can affect the wiring. A more flexible interpretation of the geometry implied by an FP program is the class of layouts which can be obtained from the literal interpretation by topological transformations of the plane, reshaping and stretching of the wire nets without picking them up out of the plane. Only the global routing and planar organization of the components is fixed. In the following sections, this notion of the planar topology of a circuit layout is explored: it is formally defined and its representation is shown to have a normal form. The realization of this normal form for the planar graphs generated by FP specifications is obtained by applying transformations to the FP programs.

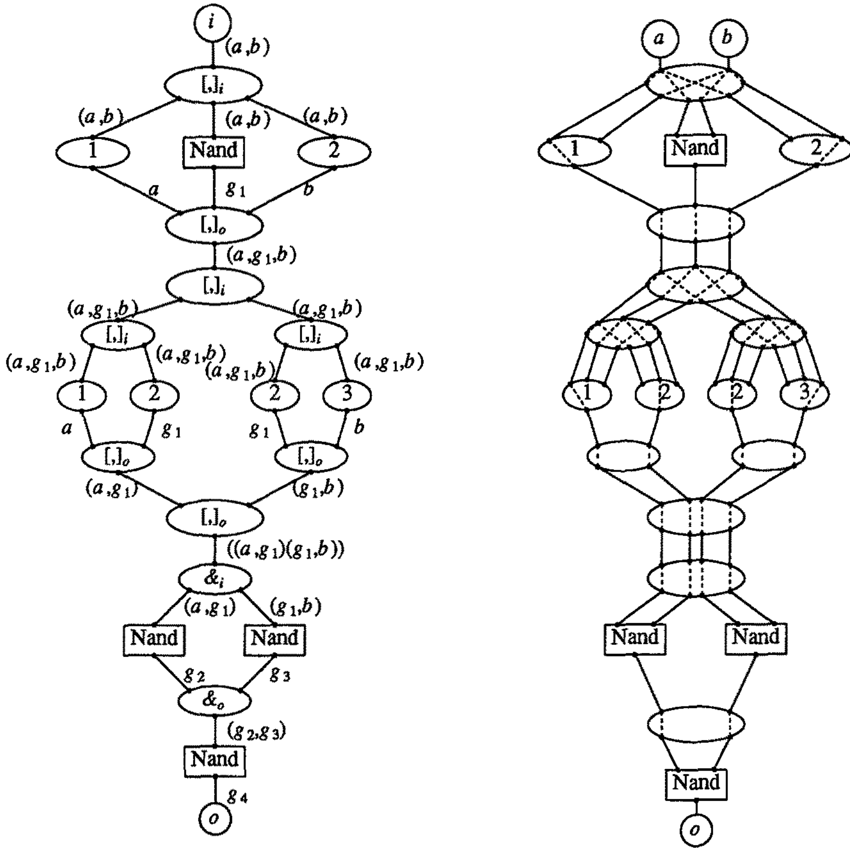

Figure 1. The planar graphs obtained from the symbolic execution of **Nand @ &Nand @ [[1,2],[2,3]] @ [1,Nand,2]** : (a,b).

4. Planar Topology of a Layout

To specify the embedding of a circuit in the plane, it is first necessary to adopt a method for specifying the embeddings of planar graphs. The cyclic ordering of edges around the vertices of a connected graph uniquely defines its embedding in a closed surface of minimal genus [Edmo60, Youn63]. Hence the embeddings of a connected planar graph in the sphere are uniquely characterized by the cyclic order of edges around their vertices in these embeddings. Using the topological equivalence of the sphere (minus it's north pole) and the plane under stereographic projection, an embedding of a graph in the plane is uniquely specified by providing (in addition to the cyclic ordering of edges around its vertices) the exterior window of the embedding in the plane. That is, the order in which the vertices which lie on the exterior face of the graph would be encountered during a clockwise traversal of the exterior window. Note that this does not imply that a planar embedding exists for any cyclic ordering of edges around the vertices, but rather that the embedding is uniquely characterized by its edge orderings and exterior window if it does exist.

In general, circuits do not correspond to planar graphs (or hypergraphs) and even if they do, it may be desirable to lay them out in a non-planar fashion; asymptotic upper bounds on the area of layouts of planar graphs have been obtained from non-planar embeddings [Leis80, Vali81]. To extend the notion of planar topology to non-planar embeddings, a special type of node is introduced into the graph to implement crossings and branchings. A *planar circuit* is an embedded planar graph consisting of three types of nodes, B, R and IO-nodes. The B-nodes are the circuit elements (black-boxes) while the R-nodes are interconnection primitives. IO-nodes are nodes of degree one which lie on the external window of the embedding and serve as the inputs and outputs of the circuit. The embedding of this planar graph is uniquely specified by listing the ordering of edges around each node and the ordering of the input and output nodes of the circuit around the exterior; the input and output nodes are required to reside on the exterior window. Each of the R-nodes is accompanied by a partition of its edges. The partition indicates edges which must be interconnected within the R-node. Figure 2a shows a typical R-node and its partition. Note that nodes may have self-loops such as the one depicted in Figure 2b. Self-loops are said to be *trivial* if they do not enclose any other nodes. The loops formed by the edges **e**, **f**, and **j** are trivial self-loops while those formed by the edges **a** and **g** are not.

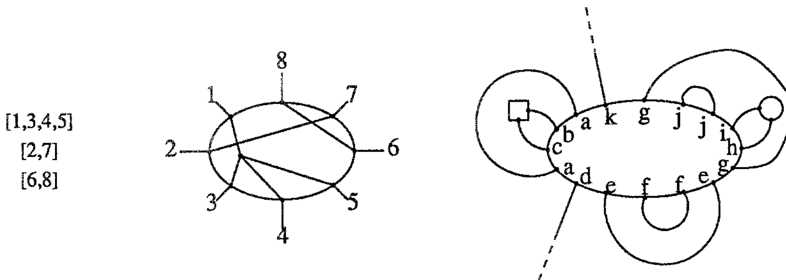

Figure 2. a) An R-node and its partition. b) An R-node with self-loops.

Given a layout, draw a set of simple disjoint closed curves in the plane such that all of the branchings and crossings of the layout lie within these curves and the circuit components lie outside of these curves. If each area enclosed by one of these curves is represented by an R-node, then a planar circuit is obtained. The layout is then said to be *covered* or *represented* by this planar circuit. The covering of a layout by a planar circuit is certainly not unique since there is more than one way to draw these curves. However, any two planar circuits which cover a given layout are related by a simple set of operations which can transform one planar circuit into the other. These operations not only define the equivalence between planar circuits which cover the same layout, but also provide the equivalence between planar circuits which cover layouts which can be topologically transformed into each other: obtained by the planar movement of components and the reshaping and stretching their interconnections. Three operations and their inverses are sufficient to attain this equivalence.

Merging R-nodes.

Any two R-nodes which are connected by one or more edges can be merged along those edges. Figure 3 shows an example of a merge. The edges on which the merge takes place are subsumed by the new set of partitions of the new R-node to preserve connectivity of the underlying circuit. A merge may create a *trivial* self-loop: a self-loop which does not enclose any other nodes of the planar circuit.

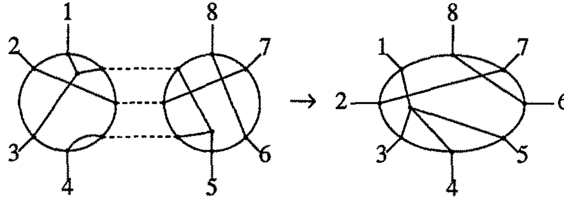

Figure 3. A merge of two R-nodes.

Cleanly Dividing R-nodes.

Cleanly dividing an R-node results in two new R-nodes with no new edges. Since the connectivity of the circuit must be preserved, this operation is only possible when the cyclic ordering of the edges of an R-node can be divided so that no partition is represented on both sides of the division. Figure 4 shows an example of a clean divide of an R-node along the dashed line. This is the only clean divide possible for this R-node.

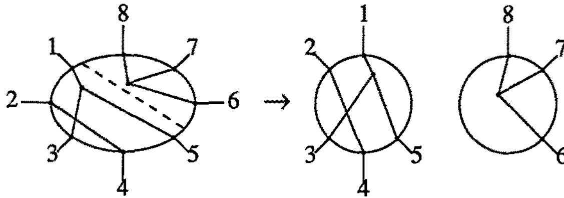

Figure 4. A clean divide of an R-node.

Removing trivial R-nodes.

A *trivial* R-node is an R-node of degree two whose edges are connected within the R-node. A trivial R-node can be removed leaving behind a wire. Figure 5 illustrates a removal.

These three operations and their inverses are guaranteed to produce planar circuits, providing the planar circuit on which they are applied possesses some connectivity properties. Otherwise, it is possible for a clean divide operation to disconnect a planar circuit's graph. However, if the underlying circuit represented by a planar circuit is connected when its inputs and outputs are joined (shorted), then the operations cannot disconnect the planar circuit's graph. This connectivity property is a reasonable restriction since it merely requires that each circuit component and wire be connected (via wires and components) to at least one input or output. Under this restriction, the operations and their inverses define equivalence classes of planar circuits which cover layouts with the same planar topology.

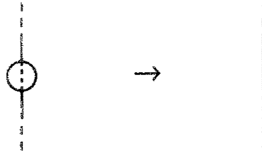

Figure 5. A removal of a trivial R-node.

Definition

Two planar circuits, A and A', are *homeomorphic* if there exists a finite sequence of operations transforming A into A'.

By showing that any representation of a layout by a planar circuit is homeomorphic to the 'smallest' planar circuit representing the same layout (the one in which each R-node covers as few branch points and crossings as possible), the equivalence between planar circuits which cover the same layout is obtained.

Proposition

Two planar circuits which cover the same layout are homeomorphic.

The operations on planar circuits also provide the means by which to simulate the changes in the layout which occur as components are moved around and their interconnections are stretched and reshaped while staying in the plane. These changes in the layout can be simulated by operations on the planar circuits which cover the layouts. Conversely, the operations on planar circuits can be realized by these changes in the layout.

Proposition

Layouts can be transformed into each other by orientation-preserving topological transformations of the plane and individual path deformations in the space which is obtained by removing the interior of the modules from the plane, if and only if any two planar circuits covering their respective layouts are homeomorphic.

The planar topology of an integrated circuit is then defined as follows.

Definition

The *planar topology* of a layout is the class of homeomorphic planar circuits which contains a planar circuit representing the layout.

5. Optimizing Planar Topology

If an FP program specifies the planar topology of a layout rather than a particular planar circuit, then an entire class of homeomorphic planar circuits must be considered in obtaining the layout of the FP program. The layout procedure may be sensitive to the particular planar circuit chosen from a class of homeomorphic planar circuits. Selecting the best layout with a given planar topology, would require

the consideration of all planar circuits within a class of homeomorphic planar circuits. However, the planar circuits within a class which need to be considered can be limited by requiring their R-nodes to have certain properties. Only those planar circuits within each class of homeomorphic planar circuits possessing these properties need be considered, since these planar circuits are optimal (under topological cost measures) within their class of planar circuits, if the layout procedure satisfies some assumptions. Since a planar circuit with these properties is unique (modulo one operation) within it's class of homeomorphic planar circuits, these properties provide a normal form for planar circuits.

Definitions

An R-node is *indivisible* if it cannot be cleanly divided even after permuting the order of adjacent self-loops.

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

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

One last operation involving a single R-node (which is in fact a composite of two previously defined operations) is required. This operation is treated as a single operation rather than as two separate operations since the uniqueness of the normal form can be only up to this operation.

Refoldings

A refolding of a node is in fact two operations, a divide (clean or inverse merge) 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.

The uniqueness, which gives a one-to-one correspondence between the R-nodes of homeomorphic maximal-indivisible planar circuits, is stated as follows.

Theorem

If A and A' are maximal-indivisible homeomorphic planar circuits, then A can be transformed into A' by a sequence of operations consisting only of refoldings.

The uniqueness of a homeomorphic maximal-indivisible planar circuit can only be up to refoldings since an R-node can never completely surround a B-node of the planar circuit. Merging an R-node with itself along a set of its self-loops may result in a doughnut R-node which encloses B-nodes. In this case, the information as to the relative position of the wires internal to the R-node with respect to the B-node, would be lost. Figure 6 contains a layout which can be covered by both of the maximal-indivisible planar circuits in Figure 7. The planar circuit on the right in Figure 7 could be obtained from the one on the left by dividing it's R-node along the diagonal in the lower left and then merging the two resulting R-nodes along the wire connecting them at the upper right.

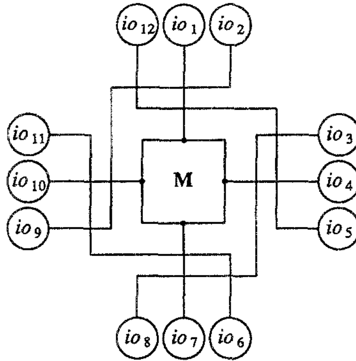

Figure 6. A layout which can be covered two distinct maximal-indivisible planar circuits.

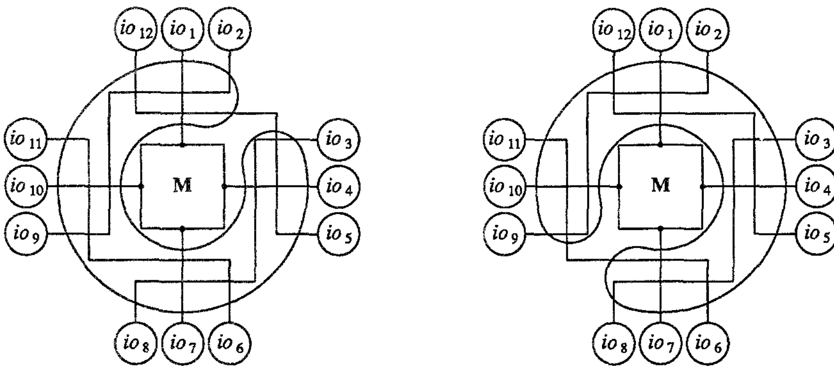

Figure 7. Two maximal-indivisible planar circuits which cover the layout in Figure 6.

The uniqueness of a normal form is often shown by deducing the Church-Rosser property from a local confluence property [Newm42]. Unfortunately, homeomorphic planar circuits do not lend themselves to this technique since the uniqueness can only be up to refoldings and refoldings can accomplish 'forward moves' when the planar circuit is not maximal-indivisible. Instead, the uniqueness is established by showing that an arbitrary sequence of operations leading to a maximal-indivisible planar circuit can be replaced by a sequence of operations consisting of specific types of operations in a particular order and resulting in the same planar circuit. The proof which is given in detail in [Schl86] consists of several steps in which the sequence of operations is rearranged by moving inverse merges and inverse clean divides to the end of the sequence and placing the other operations in a specific order. In the process, these inverse operations must cancel and can be dropped from the sequence since the final planar circuit is maximal-indivisible. If the original planar circuit is also maximal-indivisible then the sequence can be further rearranged by removing operations whose effects cancel, until only refoldings remain. Since homeomorphic maximal-indivisible planar circuits are unique modulo refolding operations and admit no merges or clean divides, this is defined to be the normal form for planar circuits. By examining the R-nodes and applying the appropriate operations, a planar circuit can be transformed into an equivalent maximal-indivisible planar circuit.

The rationale for obtaining a maximal-indivisible planar circuit is an issue which affects the procedure which will produce the layout from a planar circuit. The particular choice of planar circuit from among homeomorphic ones to which the layout procedure is applied may affect the quality of the layout. The underlying assumption is that the procedure produces a layout which is represented by the given planar circuit. With a few assumptions on the cost functions which measure the layout and the layout procedure, maximal-indivisible planar circuits can be shown to be optimal with respect to other homeomorphic planar circuits. The assumptions needed are the natural ones implied by the normal form.

1. Merges do not increase the cost.

2. Clean divides do not increase the cost.

3. Permuting self-loops to allow clean divides does not increase the cost.

4. Trivial R-nodes have zero cost.

At the topological level, these are natural assumptions to be made since the cost functions measure topological characteristics. Examples of topological cost functions which meet these assumptions are, the number of pairwise crossings required, the minimum number of layers needed if each net is restricted to one layer, and the minimum number of connections (module pins) which need to be dropped so that the remaining modules pins can all be connected on a single layer. At the geometrical level, these assumptions imply that the procedures which generate the layout implicitly examine the inverses of the operations of merging and cleanly 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 structures of the layout. It is easier for the layout procedure to determine how to 'glue' unconnected adjacent sections of the layout and to decompose R-nodes rather than to have to deal with non-maximal or divisible R-nodes. In the latter case, an optimal layout procedure would realize operations which are equivalent to merges and clean divides.

To assert that the costs of all homeomorphic maximal-indivisible planar circuits are the same requires an additional assumption. The problem is that even though an R-node is maximal-indivisible, permuting the relative order of a pair of adjacent self-loops can affect its internal wiring complexity. If the layout procedure is allowed to order these adjacent self-loops of an R-node as it sees fit, then the costs of homeomorphic maximal-indivisible planar circuits are the same. With this additional assumption, the following result is obtained from the theorem.

Lemma

Under the given assumptions, a maximal-indivisible planar circuit is optimal within its class of homeomorphic planar circuits.

6. The Planar Topology of FP Programs

Using the results of the previous section, it suffices to apply the layout procedure to a maximal-indivisible planar circuit in the class defined by the FP program. In this section, the transformation of an FP program's planar circuit into a maximal-indivisible one is explored. The planar circuit of an FP program is specified recursively in terms of its functional forms. Rather than flattening an FP program into a planar circuit and losing the structural information provided by its functional forms, this information is retained by representing the planar circuit as a tree in terms of its functional forms (it's computation tree). This information is exploited so as to efficiently map the planar circuit to a layout.

Each sub-tree in the computation tree corresponds to a function application. The functional form or primitive function as well as the input and output objects are stored in the root of the sub-tree. A primitive function is a leaf, and a functional form has as its children, the sub-trees corresponding to its sub-function applications. This computation tree is extracted by symbolically executing the FP program. Figure 8 shows the computation tree corresponding to the FP program of the planar circuits in Figure 1. The normal form of the planar circuit is realized by rearranging its computation tree. In general, it is not possible to rearrange the tree so that only maximal R-nodes are generated. However, the rearrangement of the computation tree is often sufficient to allow the layout procedure to effectively consider maximal-indivisible R-nodes.

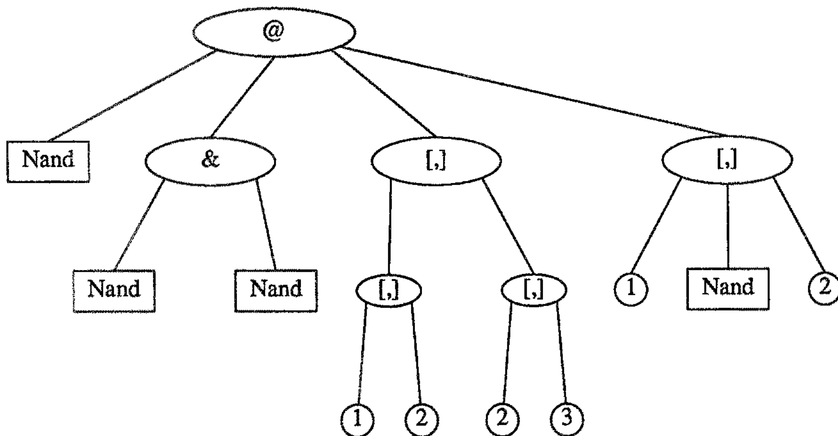

Figure 8. The computation tree of **Nand @ &Nand @ [[1,2],[2,3]] @ [1,Nand,2]**

In order to perform the operations to transform a planar circuit into its normal form, the planar circuit must possess certain connectivity properties. The planar circuits obtained from FP programs do not in general satisfy these connectivity properties. Much of the structure implied by an eager evaluation of a functional program may not be useful. This not only poses a problem in the application operations to the planar circuit, but may result in inefficient layouts since unnecessary components and wiring are generated. Before operating on the planar circuit, it is 'pruned' to remove useless wiring and components. The pruning is accomplished by a second pass in the reverse direction (from output to input) over the computation tree in which the objects stored in the nodes have their atoms tagged as useful or

not. These tagged objects are used by the layout procedure to avoid the generation of useless circuit structures.

The merging of R-nodes is accomplished by rearranging the computation tree to collect adjacent R-nodes (R-nodes which share edges) within a sub-tree. The planar circuit of a sub-tree which does not contain any B-nodes (a routing sub-tree) can be enclosed 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, this region forms a single R-node. Treating the routing sub-tree as a single R-node produces the same effect as combining all of the sub-tree's R-nodes using merges and inverse clean divides. If the computation tree can be rearranged so that adjacent R-nodes are located within a the same routing sub-tree, then maximal R-nodes will be obtained. The rearrangements of the tree which attempt to produce this effect are based on FP identities.

The R-nodes of a planar circuit are the result of the application of the primitive (structure modifying) functions and the distribution of inputs or collection of outputs required by a functional form. Only the distribution of the inputs to the sub-functions of a **construct** functional form can result in an R-node which cannot be decomposed into trivial R-nodes and removed. To extract this possibly non-trivial R-node generated by the **construct** functional form, a new functional form called a **projection** is introduced. 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** functional form in that instead of applying each h_i to the input object, h_i is applied only to x_i where the input object is of the form $(x_1, \ldots, x_n)$. If the input object, x, is not of this form then $\{h_1, \ldots, h_n\}:x$ is undefined ($\perp$). This new functional form provides the following identity for FP functions,

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

If $[g_1, \ldots, g_n]$ does not contain any B-nodes then it forms a routing sub-tree and can be treated with as a single R-node. By replacing a function, f by $f@Id$, $[f_1, \ldots, f_n]$ can be rewritten 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. This R-node is the same as the R-node that would have been generated to distribute the input object to each of the sub-functions of $[f_1, \ldots, f_n]$.

Once the routing node generated for the **construct** is extracted and placed within its own routing sub-tree, adjacent R-nodes in the planar circuit are a result of only the **compose, right insert** and **seq** functional forms. Ignoring for the moment the **right insert** and **seq** functional forms, the following identities can be used to group adjacent R-nodes within the same sub-tree of the computation tree.

1. $f_1@(f_2@f_3) = (f_1@f_2)@f_3$
   The associativity of the **compose** functional form is used to gather R-nodes within routing sub-trees.

2. $[h_1@g_1, \ldots, h_n@g_n] \equiv \{h_1, \ldots, h_n\}@[g_1, \ldots, g_n]$
   This identity is used to extract the R-node which distributes the input object from its **construct** functional form. In practice, the g_i's will correspond to routing sub-trees.

3. $\{h_1@g_1, \ldots, h_n@g_n\} \equiv \{h_1, \ldots, h_n\}@\{g_1, \ldots, g_n\}$
   This identity is used to extract routing sub-trees from **projection** functional forms.

4. $\&f:(x_1, \ldots, x_n) = \{f_1, \ldots, f_n\}:(x_1, \ldots, x_n)$ where $f_i = f$ for $1 \le i \le n$.
   The **apply-to-all** functional form is handled as **projection** using this identity.

5. $\{h_1, \ldots, h_n\}@\{g_1, \ldots, g_n\} \equiv \{h_1@g_1, \ldots, h_n@g_n\}$
   This identity is applied in order to maximize the parallelism. If there is a **compose** functional form which has two adjacent children which are **projection** functional forms, then they are combined using this identity.

The identities above are applied by a bottom-up recursive procedure which rearranges a computation tree without **seq's** and **right-inserts**. These rearrangements have no effect on the planar circuit generated since the connectivity and the embedding of the planar circuit is preserved. However, if the layout procedure treats a routing sub-tree as a single R-node, then these rearrangements realize effects equivalent to the application of operations to the planar circuit to merge its R-nodes. Hence these rearrangements do not alter the planar topology, but bring the planar circuit closer to a maximal-indivisible representative of its planar topology.

If the layout procedure treats a sub-tree with no B-nodes in it as a single R-node, instead of generating its planar circuit, then two R-nodes in the rearranged computation tree 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 **compose** which are separated by nested **projections**, 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 **projections**.

Unfortunately this property does not extend readily to trees containing **right inserts** and **seqs**. Each **right insert** and **seq** could be rewritten, using the appropriate identity, as a combination of **composes** and **projections**. However, these transformations would sacrifice the structural information provided by the presence of these forms. Instead, the rearrangement procedure is applied individually to the children of **right inserts** and **seqs** functional forms and any routing sub-trees which result, are pulled out.

The techniques described in this section have been extended to the sequential versions of the **apply-to-all, right insert** and **seq** functional forms of vFP. Only the 'pruning' of the planar circuit is substantially different. Pruning is more complex in this case, since the usefulness of structure must be determined for the several applications which are mapped to it, rather than the single application in the combinational case.

7. Example

The specification of a simple carry-save array multiplier serves to illustrate the flexibility offered by fixing only the planar topology of the design as it is refined. A multiplication algorithm for unsigned digits written in FP will be developed starting with the following high-level description:

```
mult       ≡  2@repeat@[1,initialize,2]
repeat     ≡  (nul@1->id; repeat@stage)
stage      ≡  [tlr@1, +@[*@[last@1,3],2], *@[radix,3] ]
radix      ≡  %2
initialize ≡  %0
```

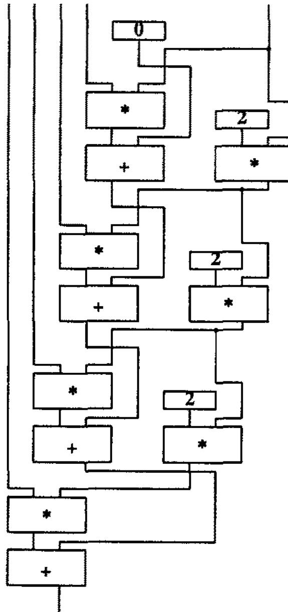

Figure 9. The abstract sketch of the **mult**:$((y_4,y_3,y_2,y_1),x)$.

Three arguments are passed from stage to stage. The first is the multiplier, the second is the sum of the partial products, and the last is the multiplicand. One bit of the multiplier is consumed at each stage. The partial product is added to the product of one bit of the multiplier and the multiplicand. The multiplicand is shifted (by multiplying it by the radix) left to prepare for the next stage. Eventually, the second argument is the product. The layout of the planar circuit implied by this multiplication algorithm is given in Figure 9.

This high-level algorithm is realized in hardware by implementing the radix 2 multiplication with a shift to realign the multiplicand. The multiplication of the multiplicand by a single digit of the multiplier is achieved by the function **select**. A carry propagate adder performs the addition with half and full adders. Conditionals are used in **CPA** and **HA** to determine the number of bits to be summed in each column. Figure 10 contains the layout of the planar circuit of this version of **mult** which is given below.

```
mult       ≡  2@repeat@[1,&initialize@2,2]
repeat     ≡  (nul@1->id; repeat@stage)
stage      ≡  [tlr@1, addthem@[select@[last@1,3],2], shift@3]
select     ≡  &(nul@2->2;andg)@distl
shift      ≡  apndr@[id,%()]
addthem    ≡  CPA@apndr@[trans,%()]
CPA        ≡  seq((nul@2->HA@1;(nul@1@1->HA@[2@1,2];FA)))
HA         ≡  (nul@1->id;(nul@2->[2,1];[andg,xorg]))
FA         ≡  [org@[1,2],3]@apndl@[1,HA@[2,3]]@apndr@[HA@1,2]
initialize ≡  %0
```

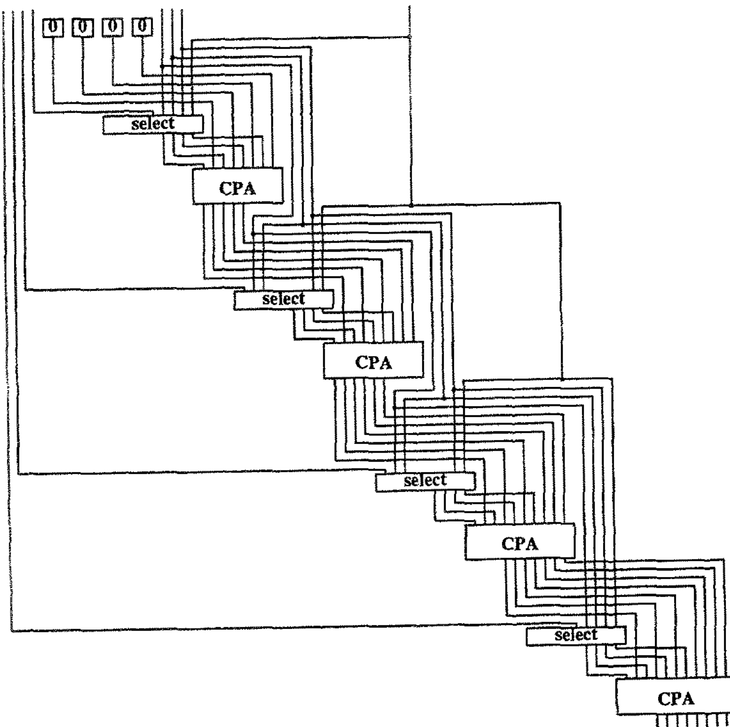

Figure 10. The abstract sketch of the **mult**:$((y_4,y_3,y_2,y_1),(x_4,x_3,x_2,x_1))$.

To increase the performance of the multiplier, the technique of carry-save addition is employed to avoid lengthy carry propagations at the intermediate stages. Instead of generating the sum at each stage, the partial product is generated in the form of two vectors and passed to the next stage. Hence, separate vectors of carry and sum bits are retained instead of the actual sum. The object being passed from stage to stage is now more complex. It consists of the multiplier, sum and carry bit vectors and the multiplicand. Note that the null object, '()', is appended to the left of the sum vector at each stage so that it matches the length of the carry vector and the multiplicand. Since the null object does not contain any atoms, wiring is not a concern. The specification of **mult** is now as follows.

```
mult       ≡  finish@repeat@setup
repeat     ≡  (nul@1->id; repeat@stage)
setup      ≡  apndl@[1,trans@2]@[1,&[initialize,initialize,id]@2]
stage      ≡  unweave@[1,addthem@2]@weave
weave      ≡  [tlr@1,&([2,3,1,4]@apndl)@distl@[last@1,trans@[2,3,4]]]
addthem    ≡  &(nul@2->[2,4,1];(nul@1->HA*;FA*))
unweave    ≡  [1,apndl@[%(),3@2],apndr@[1@2,%()],apndr@[2@2,%()]]@[1,trans@2]
FA*        ≡  [1@1,2,2@1]@[FA@[tlr,3]@tlr,4]@[1,2,andg@[3,4],4]
HA*        ≡  [1@1,2,2@1]@[HA@tl@tlr,4]@[1,2,andg@[3,4],4]
finish     ≡  CPA@apndr@[trans@[2,3],%()]
CPA        ≡  seq((nul@2->HA@1;(nul@1@1->HA@[2@1,2];FA)))
FA         ≡  [org@[1,2],3]@apndl@[1,HA@[2,3]]@apndr@[HA@1,2]
HA         ≡  (nul@1->id;(nul@2->[2,1];[andg,xorg]))
initialize ≡  %0
```

At each stage, the sum and carry vectors and the multiplicand are *woven* together into the proper columns. The rightmost bit of the multiplier is distributed to each column and a full adder (except in the leftmost column where a half adder is used since there is no sum bit) is used at each column to add its sum bit, carry bit and the product of the multiplier bit with the multiplicand bit. The resulting sum and carry bits and the multiplicand are then *unwoven* for the next stage. As in the previous version, conditionals are used to detect the existence of a sum bit and invoke either the half or full adder function accordingly. Finally, the product is generated by the function **finish** which employs a carry-propagate adder to sum up the sum and carry bit vectors. Note that the sum and carry bit vectors have been initialized to zeros to simplify matters. In practice, the first two partial products should be used in their place.

The layout is extracted with the functions **HA*** and **FA*** defined as primitives. Figures 11 and 12 were both obtained from the application of the previous FP program to a symbolic object consisting of two 4-bit vectors. The same layout procedure (in which routing sub-trees are treated as single *R*-nodes) produced both layouts. However, the layout in Figure 11 was obtained by first rearranging the computation tree as described in the previous sections, before extracting its planar circuit. This specification of a carry-save array multiplier is simpler than the FP specifications appearing in [Schl84] where details of the wiring between stages had to be considered explicitly so as to produce a reasonable layout.

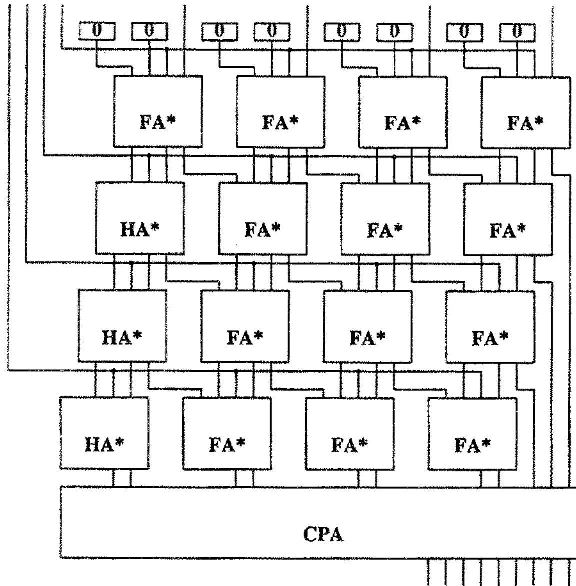

Figure 11. The layout of the optimized planar circuit of **mult**:$((y_4,y_3,y_2,y_1), (x_4,x_3,x_2,x_1))$.

8. Concluding Remarks

The objective of this research is to develop a formal high-level language approach to the specification, simulation, performance evaluation, and chip layout planning for VLSI systems. The use of a high-level applicative language (vFP) is motivated by the geometric detail provided by this programming style. The level of geometry afforded by the functional programming style is the subject of this paper. The planar topology of an integrated circuit is formally defined and demonstrated to be the desired level of geometric information to infer from a functional program. A normal form which is optimal with respect to topological cost measures is derived and used to improve the wiring of the layouts. The specification of a carry-save array multiplier is used to illustrate how this optimization reduces the effort required to specify integrated circuits.

9. Acknowledgements

The author is indebted to Milos Ercegovac, Sheila Greibach and Dorab Patel for their help, suggestions and support. A special thank you to Chan Pak Kuen for his help, and in particular for staying up to read late night versions of this paper.

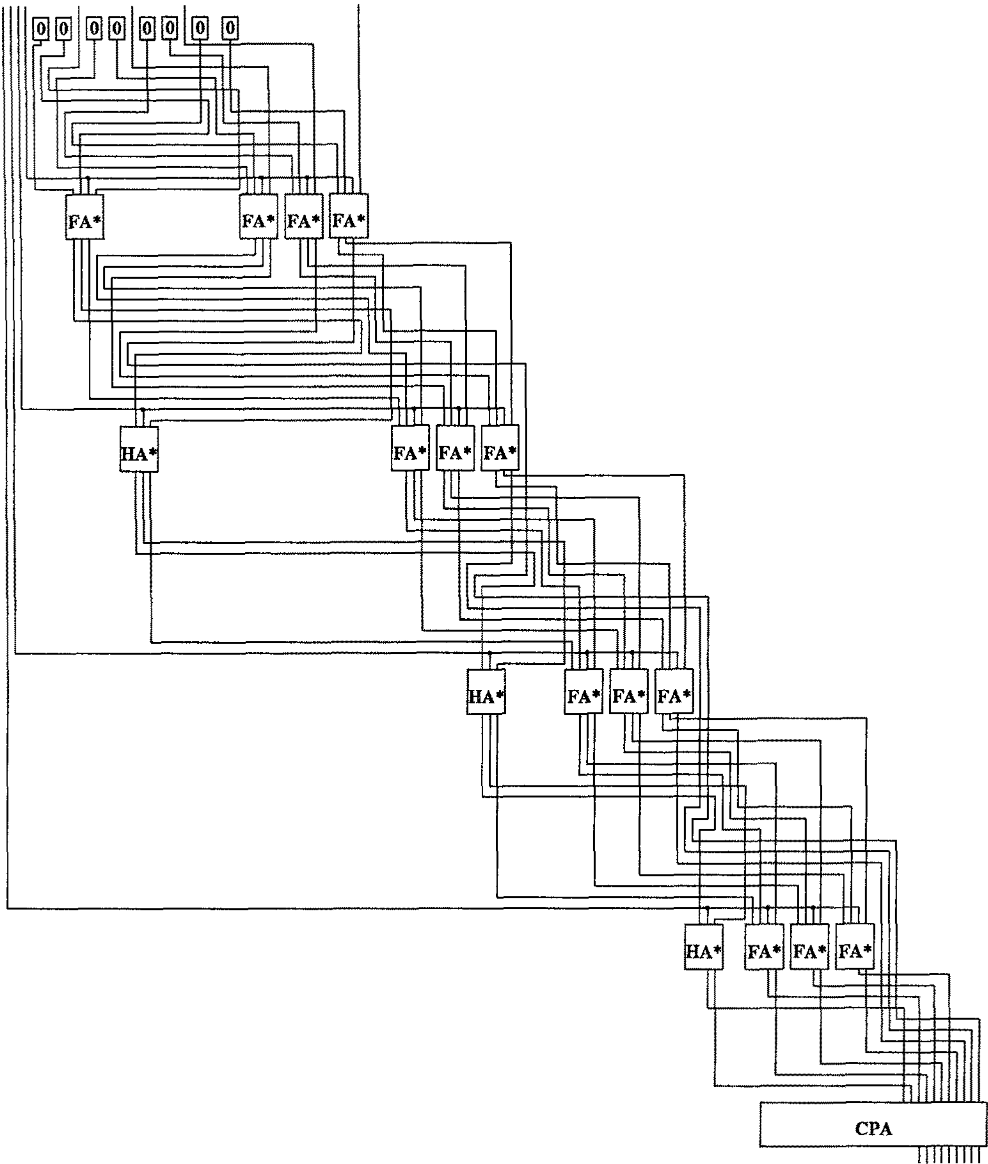

Figure 12. The layout of the untreated planar circuit of **mult**:$((y_4,y_3,y_2,y_1), (x_4,x_3,x_2,x_1))$.

References

[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).

[Edmo60] Edmonds, J. R., "A combinatorial representation for polyhedral surfaces," *American Mathematical Society Notices*(7), p.646 (1960).

[Joha79] Johannsen, D., "Bristle Blocks: A Silicon Compiler," pp. 310-313 in *Proceedings 16th Design Automation Conference*, San Diego, California (June 1979).

[Leis80] Leiserson, C. E., "Area-Efficient Graph Layouts (for VLSI)," pp. 270-281 in *Proceedings 21st IEEE Symposium on Foundations of Computer Science* (1980).

[Mesh85] Meshkinpour, F. and M. D. Ercegovac, "A Functional Language for Description and Design of Digital Systems: Sequential Constructs," pp. 238-244 in *Proceedings of the 22nd Design Automation Conference* (June 1985).

[Newm42] Newman, M. H. A., "On Theories with a Combinatorial Definition of `Equivalence´," *Annals of Mathematics* **43**(2), pp.223-243 (April 1942).

[Oust81] Ousterhout, J. K., "Caesar: An Interactive Editor for VLSI Layouts," *VLSI Design* **II**(4), pp.34-38 (fourth quarter 1981).

[Oust84] Ousterhout, J. K, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "Magic: A VLSI Layout System," pp. 152-159 in *Procceedings of the 21st Design Automation Conference* (June 1984).

[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).

[Rive82] Rivest, R. L., "The 'PI' (Place and Interconnect) System," pp. 475-481 in *Proceedings 19th Design Automation Conference*, Las Vegas, Nevada (June 1982).

[Schl84] Schlag, M., "Extracting Geometry from FP for VLSI Layout," CSD-840043, University of California, Los Angeles, Los Angeles, California (October 1984).

[Schl86] Schlag, Martine, "Layout from a Topological Description," PhD Dissertation, Available as Technical Report CSD-860039, University of California, Los Angeles (July 1986).

[Shee84] Sheeran, M., "muFP, a language for VLSI design," pp. 104-112 in *Proceedings ACM Symposium on LISP and Functional Programming* (1984).

[Sisk82] Siskind, J. M., J. R. Southard, and K. W. Crouch, ''Generating Custom High Performance VLSI Designs from Succinct Algorithms,'' *MIT Conference on Advanced Research in VLSI*, pp.28-40 (January 1982).

[Vali81] Valiant, L. G., ''Universality Considerations in VLSI Circuits,'' *IEEE Transactions on Computers* **C-30**(2), pp.135-140 (February 1981).

[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).