University of California

Los Angeles

Extracting Geometry from FP for VLSI Layout

Martine Schlag

October 1984

This research has been supported in part by the ONR Contract N00014-83-K-0493, by the MICRO-Rockwell Research Grant and an Atlantic Richfield Foundation Doctoral Fellowship.

Table of Contents

1. Introduction

The use of CAD 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 on a number of planes (artwork). To use these tools the designer must provide a description of the circuit in one of the following forms suitable for the particular the tool.

1. Functional description in terms of primitives at various levels

2. Circuit description

3. Stick Diagram/Symbolic Layout

4. Fixed Geometry

The last level is the desired final format. Descriptions at higher levels are used to simulate and analyze the circuit. Unfortunately, many of the characteristics needed to realistically estimate the feasibility and performance of a design are geometric and only manifest themselves at the lower two levels. For this reason, circuit extractors and other tools are necessary to map back from the lower levels to the ones above in order to simulate the circuit, and the design process usually requires several iterations.

The systems of CAD tools can be divided into two classes, those in which the geometry is provided by the designer and those in which the system must generate it.

Specifying fixed geometry entails the use of a graphics editor with which elements can be placed and positioned [Ou81,OHM83]. The most recent sophisticated tools in this area now allow the user to place elements and provide "automatic routing," [OHM83]. In such a system, there is usually a library available from which predesigned elements can be pulled and placed in the current design. The major draw back of this type of system is that the objects it is handling are not very flexible and cannot adapt well to the different interconnection, size and shape requirements they may encounter in various environments. Much of the work involved in making the pieces fit is left to the designer. In addition this type of system allows the introduction of errors in to the design process, requiring the use of error detection tools (i.e. Design Rule checkers) and circuit extractors.

Describing a circuit with a stick diagram[Will78] or a symbolic layout[West81], also entails the use of a graphics editor. However in this case, the designer places symbols which are then expanded by the system into fixed geometry. Compaction is then used to position the expanded objects. The success of compaction can depend largely on the initial placement of the elements.

In the second type of system, circuit elements and their interconnections are specified or compiled from a high level description and tools are used to place these elements and route their interconnections[Riv82]. Most of the associated geometric optimization problems have been shown to be NP-hard, leaving only heuristic algorithms as candidates for these jobs, making the tools slow and the results inefficient and awkward. The problem arises because placement and routing are tightly coupled. Many placement tools estimate channel widths before placing elements. Some placement tools attempt to measure the connectivity of elements before placing them. On the other hand, routing can not begin until the elements have been placed. It is awkward to deal separately with routing and placement.

In both types of systems, there is a gap between the non-geometric and the geometric descriptions. Describing a circuit by geometry, entails the use of an extractor or a separate definition provided by the designer to simulate and analyze its behavior at the functional and circuit levels. On the other hand describing it without geometry requires the use of placement and routing tools, hiding or perhaps even destroying any correspondence between the functional description of a circuit and its geometry.

In this work, the extent to which geometric information can be inferred directly from a functional description of the circuit, is studied. A Functional Programming Language (FP) is used to specify computations for implementation in VLSI. The input to this tool is an algorithm written in FP, and the output is a visual representation of the behavioral and structural organization implied by this algorithm. The idea is not to generate VLSI circuits automatically but rather to provide an interactive tool by which algorithms can be examined for feasibility in terms of time and space constraints. As will be discussed, it can not be expected that a complete low-level description of a layout will be the direct result of an FP specification of an algorithm. However there is enough information in an FP program to obtain a *sketch* of a circuit, reflecting its spatial organization and interconnection pattern, which could serve as a floor plan.

This scheme is attractive because it allows the semantics of the circuit (its functional description) to follow the circuit through design stages and guide design decisions. There may be ways of dealing reasonably with the geometric optimization problems by using behavioral information. Knowledge of the behavior of a circuit removes the need to optimize globally, since the critical portions of the circuit can be identified and optimized at the expense of others. The corresponding behavioral and structural hierarchy can also be exploited to improve the efficiency of these tools. Inferring geometry directly from a functional specification does not mean that geometric tools such as channel routers and compactors are no longer necessary. They can be used more efficiently within this framework since more control over where and how they are applied is available.

In the following sections a brief description of FP will be given, as well as a discussion of the level of representation of circuits, the structural implications of FP and the translation process from FP to circuits. In the third section, the implementation of a system generating *sketches* from FP expressions will be described. Several applications will be demonstrated in the fourth section.

2. Preliminaries

2.0 FP and its Salient Features

The FP language, as described in [Ba78], 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 '?' 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. Functional 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 functional forms. The appendix contains a formal description of FP[Ba83], including the lists of primitives and functional forms of Berkeley FP and T FP (the versions available at UCLA).

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; a function which adds two bit vectors of any size can be defined.

Combining forms 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.

A computation can be viewed as consisting of two types of activities: directing data movement and 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.

In FP, the only bindings are those of functions to function names. 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.

2.1 Describing Circuits in FP

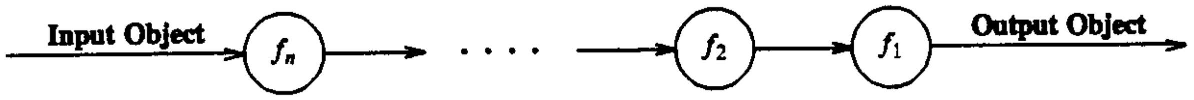
The concept of state does not exist within an FP program. Whatever information is needed for a computation must appear in the input of the function performing the computation. The result of a computation is otherwise independent of its environment. When a function is invoked (applied) it is evaluated 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 behaviour 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 must 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] and [Mesh84].

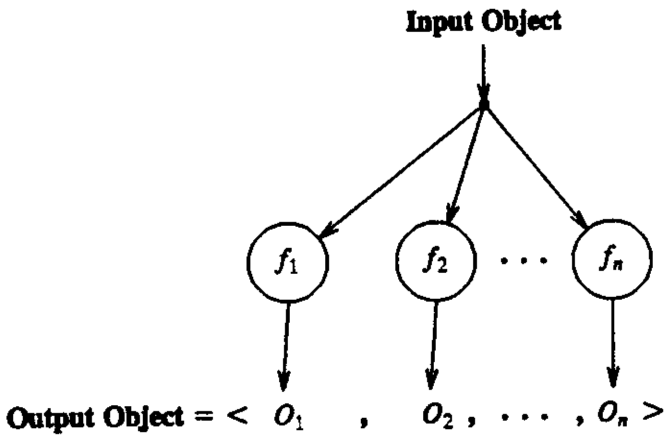
2.2 The Structural Implications of FP

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 graphs with functions as nodes (the conditional form will be discussed later).

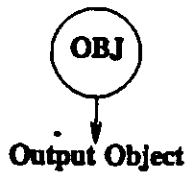
a) Compose $f_1@f_2@...@f_n$



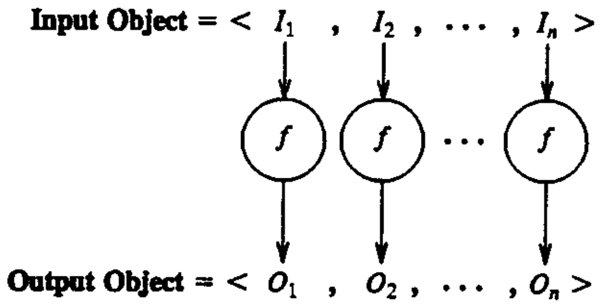
b) Construct $[f_1,f_2,...,f_n]$



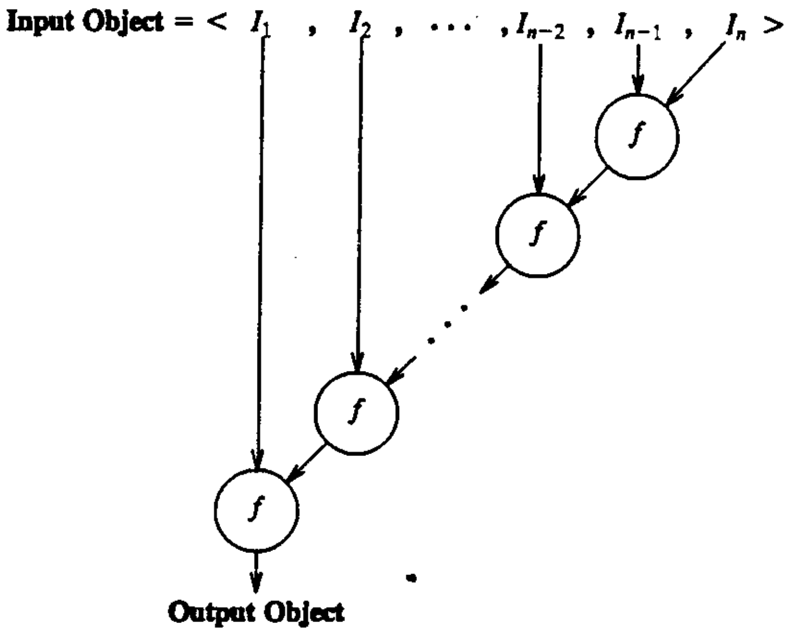
c) Constant %OBJ



d) Apply to All $\&f$


Input Object = < I_1 , I_2 , ... , I_n >
f
f
f
Output Object = < O_1 , O_2 , ... , O_n >

e) Right Insert $!f$


Input Object = < I_1 , I_2 , ... , I_{n-2} , I_{n-1} , I_n >
f
f
f
f
Output Object

f) Tree Insert $|f$

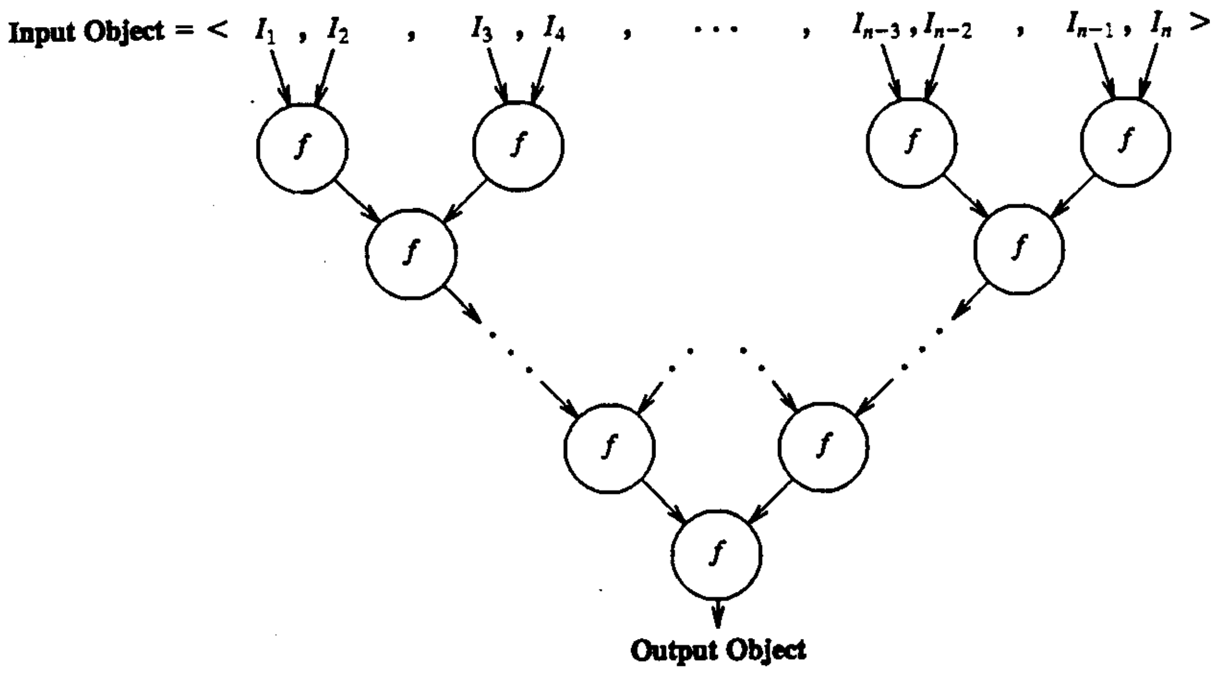

Input Object = < I_1 , I_2 , I_3 , I_4 , ... , I_{n-3} , I_{n-2} , I_{n-1} , I_n >
f
Output Object

Figure 2.1. The structures of the FP functional forms

For a few of the forms (**Apply to All, Right Insert, Tree Insert**), this 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.

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. In the next section, the amount of structural information which should be inferred from this graph is discussed.

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. For the purposes of this work, every atom is mapped to a separate wire.* This decision places limitations on the class of circuits which can be described.

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.

*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."

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 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 (e.g. iota) can not be used. Computational primitives are the primitive components of the layout, while routing primitives yield connectivity between intermediate input and output objects.

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. 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 2.2.

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.

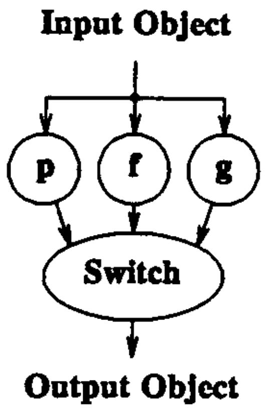


Figure 2.2 Realization of a Non-structural Conditional

Since each invocation of a function results in a new implementation, all recursions are unfolded completely. Thus each recursion must be terminated by a structural predicate, a conditional of the second type. With these restrictions, only acyclic graphs computation graphs can be described; only combinational circuits can be generated from combinational primitives. The addition of new forms to describe sequential circuits is not considered here.

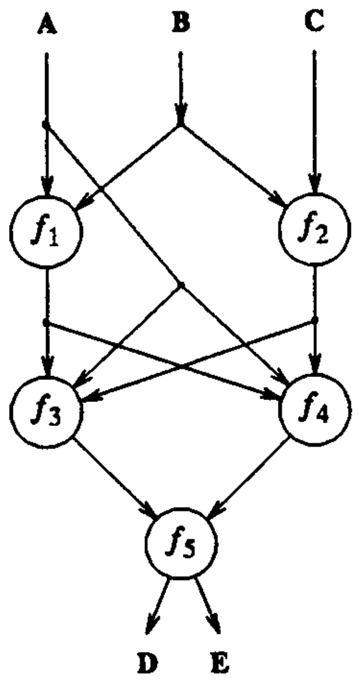
2.3 Defining the Corresponding Structure of an FP expression

The amount of information extracted and retained from the routing functions and FP functional forms of an FP expression is what defines the term "corresponding structure." At a minimum this information includes the connectivity of the computation graph: the enumeration of the primitives as boxes and their interconnections by net lists. A net is generated for each atom occurring in the computation graph; it is a list of each occurrence of the atom in an input or output object of a primitive. A box is also associated with each occurrence of an atom in the external input and output objects. Thus the connectivity of the computation graph is effectively a hypergraph*.

Example:

*A hypergraph is the generalization of a graph to higher dimensions. It consists of a set of vertices and a set of "hyperedges." Each hyperedge is a non-empty subset of vertices. See Berge,C. **Graphs and Hypergraphs**, North-Holland 1973.

The following computation graph would correspond to the hypergraph below.



Nodes : A, B, C, f_1, f_2, f_3, f_4, f_5, D, E

Hyper edges :

$\{A, f_1, f_3, f_4\}$
$\{B, f_1, f_2\}$
$\{C, f_2\}$
$\{f_1, f_3, f_5\}$
$\{f_2, f_3, f_4\}$
$\{f_3, f_5\}$
$\{f_4, f_5\}$
$\{f_5, D\}$
$\{f_5, E\}$

Figure 2.3 A computation graph and its Hypergraph

This hypergraph is obtained by traversing the computation graph with symbolic objects keeping track of each atom input to a primitive and each new atom generated by a primitive. The routing primitives can be executed during this traversal to remove them as primitives. Only the connectivity generated by the FP functional forms and routing functions is retained in this hypergraph. However FP functional forms and routing functions contain information which can be used to "layout" this hypergraph. From Figure 2.1 it is clear that more than connectivity can be acquired. Each form implies a spatial (planar) organization of its components and each routing primitive, a routing pattern. Thus the "structure" of an FP expression must encompass the connectivity of the computation graph and may contain additional information extracted from the forms and routing primitives.

In deciding how much information should be included as part of the "structure" of an FP expression, the following criteria should be considered.

Functionality

This concerns the method by which structure is extracted from an FP expression. In order to preserve functionality, the structure of a particular FP function applied to particular object should be the same regardless of which FP expression this application occurs in. Functionality makes it possible to obtain structure hierarchically. However it precludes the possibility of obtaining a "real layout" directly from an FP expression since fixed geometry does not exhibit this type of independence; the dimensions and positions of boxes and wires within an FP function maybe affected by interconnection requirements with other functions. Functionality also permits the reuse of the previously extracted structure when an FP expression reoccurs.

Distance from the real layout

Since a "real layout" can not be obtained as the "structure" of an FP expression, the "distance" or level of abstraction of this "strucutre" from a real layout should be considered. Additional tools must be used to transform this "structure" into a "real layout." The larger the distance the more complicated these tools will be. Although only a *sketch* is desired, the distance from the a "real layout" limits the level at which the circuit can be measured and analyzed. In addition a larger distance may limit the power and complicate the analysis of algebraic transformations applied to the FP expression since the effects of these algebraic transformations may be masked by the tools used to obtain the "real layout." On the other hand a smaller distance puts a larger burden on the programmer to write an efficient FP expression; the programmer must have some knowledge of th structure manifested by the FP functional forms and routing functions. It is hoped that by automating the application of algebraic transformations, less will be required of the programmer. This may be an optimistic view.

The structure of an FP function could be defined merely as the connectivity of the computation graph , but its distance from "real layouts" will require the use of conventional routing and placement tools. Hence algebraic transformations can only affect circuit level characteristics of the layout such as number and type of components, fan-outs and fan-ins, and critical paths. Wire length, area, parasitics and other geometric characteristics of the layout may be difficult to predict since these will depend primarily on the behavior of the routing and placement tools used to map the hypergraph into the plane.

A definition of structure "closer" to fixed geometry and yet retaining functionality would be more advantageous. "Relative geometry" or "topology" can be extracted from an FP expression by using the ordering of the atoms within an FP object and retaining the spatial organization implied by the FP functional forms. In this type of structure, the relative placement of elements is specified without specifying their dimensions or exact coordinates.

Formally, the "topology" of an FP expression can be defined as an embedding in the plane of a graph corresponding to the hypergraph. This planar graph comprises three types of nodes. The first type is a node of the computation graph. The second type is a branch node which is used in representing a hyperedge (a net). The third type is a crossing node which is needed to obtain a planar graph. This node always has four incident edges, two pairs, each pair belonging to a wire. The edges interconnecting these nodes correspond to a single atom and thus can be mapped directly to wires. In essence, each hyperedge of the computation graph is mapped to a tree whose interior nodes are either branch or crossing nodes and whose leaves are the original nodes of the computation graph belonging to this hyperedge. Figure 2.4 contains such a graph for the example of Figure 2.3. The branch nodes are blank and the crossing nodes have crosses.

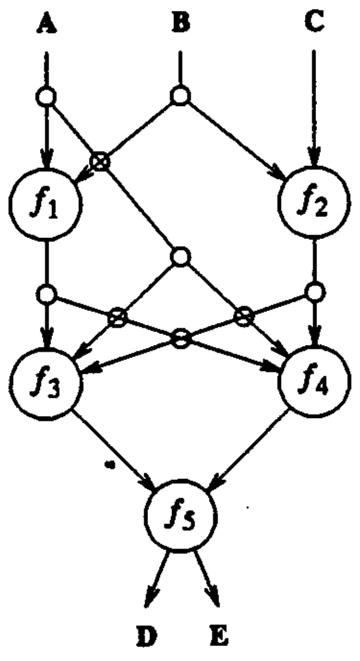


Figure 2.4 A planar embedding of the Hypergraph of Figure 2.3

It is possible to define for each FP expression (with the restrictions outlined in the previous section) such a planar graph along with its embedding. In particular it is possible to use FP objects in representing this "structure." To obtain fixed geometry, some type of compaction tool must be employed. Although the exact positions of elements are unknown, the fixed geometry obtained will still reflect the "topology" of the FP expression. Thus algebraic transformations on the FP expression can predictably affect the positions in the "real layout" of nodes of the computation graph. An implementation in which this type of structure is extracted and then mapped to fixed geometry is described in the next section. This is still a high level representation of the circuit but it provides a floor plan from which size, shape and wire lengths can be estimated.

3. Implementation

In these sections an implementation of a system generating *sketches* from FP expressions is described. A format for representing "topological" structure which will be referred to as the Intermediate Form (IF) is given, followed by a description of an interpreter generating this format and then a procedure for obtaining geometry from this format.

3.0 Representing Structure with FP Objects

The intermediate form represents the "topological" structure described in the last section; the structure of an FP expression is defined to be a planar graph and its embedding. A planar graph and its embedding is represented by dividing the plane into horizontal slices (cross-sections), and for each cross-section, listing the elements of the graph within or spanning the cross-section from left to right. Absolute vertical coordinates can be assigned to elements by allowing elements of the graph to inherit their vertical position from the cross-sections containing them. The horizontal coordinates, however, are not explicit; elements sharing the same vertical coordinates are only ordered horizontally. Since the vertical coordinates are explicit in this data structure, this IF is not purely "topological." However, whether this IF is in fact purely "topological" depends on how it is interpreted to obtain the actual coordinates of elements. In Figure 3.1 horizontal lines are imposed on the graph of Figure 2.4 to divide it into cross-sections. Notice that the elements within each cross-section can be ordered from left to right.

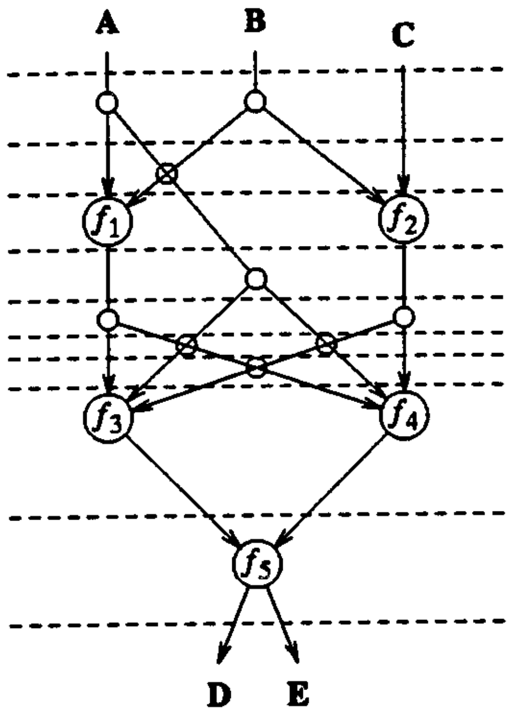


Figure 3.1 The graph of Figure 2.4 divided into Cross-sections.

This IF can be represented by FP objects. The use of FP objects to represent structure, allows the derivation of structure to be implemented within the FP framework. The IF is a list of cross-sections with the symbolic output object (of the FP expression) tagged on to the front. The symbolic output object is provided for the traversal of the computation graph. As the graph is traversed, the symbolic output object is removed, new cross-sections are added to the front, and the new symbolic output object is put on the front.

Formally, the IF consists of FP objects of the following form,

$<\mathbf{PS}\ \mathbf{CS}_1\ \mathbf{CS}_2\ \cdots\ \mathbf{CS}_n>$ for $\mathbf{n}\geq\mathbf{0}$,

where **PS** is an FP object not containing the atoms, **$,*,+,^**, and **t** (these atoms are reserved for use as delimiters) and each $\mathbf{CS}_i$ is a cross-section. A cross-section is a list of FP objects each corresponding to elements of the graph.

$CS_i = <x_1\ x_2\ \cdots\ x_m>$ where x_j is a,

1. *Free wire*

An atom (not \$,*,+,^,t).

Elements of this type are wires which traverse the cross section without being crossed by any other wires.

or,

2. *Crossing*

$<*\ w\ *\ u_1\ u_2\ \cdots\ u_h>$ such that exactly one u_h is + and at least one is ^ .

This type of element represents the wire crossings and branchings necessary for realizing the connections of the computation graph. The atom 'w' is in the position corresponding to '+' and must be distributed to each position corresponding to a '^'. The other atoms are wires which traverse this cross-section.

or,

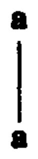
3. *Box*

<\$ level #levels id label \$ $i_1\ i_2\ \cdots\ i_k$ **\$** $o_1\ o_2\ \cdots\ o_l$ **\$>**

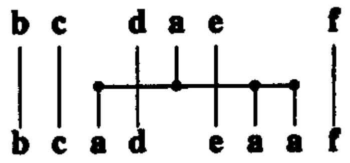
Elements of this type correspond to the primitives which are to be drawn as boxes. The format allows the specification of how many cross-sections a box will occupy. In a strictly "topological" IF this is not necessary since the dimensions of the elements are not relevant. However if the cross-sections are used to assign vertical coordinates to these elements, this format is necessary to allow boxes to have varying sizes. 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 **$** signs act as delimiters between these atoms, the input atoms, $i_1, i_2 \cdots i_n$ and the output atoms $o_1, o_2 \cdots o_m$.

Graphical interpretations of these three types of elements are illustrated in Figure 3.2.

Free Wires : a

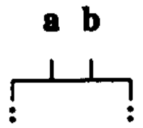


Crossings : < * a * b c ^ d + e ^ ^ f >

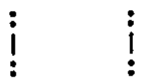


Boxes :

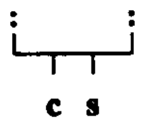
< $ f 7 g16 HADD $ c s $ a b $ >



< $ i 5 g16 HADD $ c s $ a b $ >



< $ l 2 g16 HADD $ c s $ a b $ >



< $ b 1 g16 HADD $ c s $ a b $ >

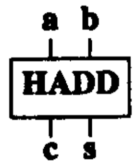


Figure 3.2 Graphical interpretation of the three element types.

Example:

The following FP object is the structure of an exclusive-or function (*mxor*) implemented with four nand gates.

FP definition of *mxor*:

```
nandg@[nandg@[1,2],nandg@[2,3]]@[1,nandg@[2,3],4]@concat@&[id,id]
```

IF generated for *mxor* applied to the symbolic input, *(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 3.3 illustrates how this FP object could be interpretated graphically. The dashed lines separate the cross-sections.

The representation of FP functions defined by Backus is as follows,

1. If f is an FP primitive then the atom F represents it.

   (Capitals are used to distinguish the actual function from the atom representing it.)

   Examples: **CONCAT** for concat, **TRANS** for trans (selectors are unchanged)

2. If h is a functional form then **<H, F_1 F_2 $\cdots$ F_n>** represents the FP function obtained by applying form h to f_1,f_2 $\cdots$ f_n.

The functional forms provided by this interpreter are,

compose

<COMP F_1 F_2 $\cdots$ F_n>=f_1@f_2@$\cdots$@f_n

construct

<BUILD F_1 F_2 $\cdots$ F_n>=[f_1,f_2 $\cdots$ f_n]

apply to all

<APALL F_1 $\cdots$>=&f_1

constant function **F_1**

<KONS F_1 $\cdots$>=%f_1

right insert

<RINS F_1 $\cdots$>=!f_1

tree insert

<TINS F_1 $\cdots$>=|f_1

conditional

<CONDL F_1 F_2 F_3 $\cdots$>=(f_1 $\rightarrow$ f_2 ; f_3)

The map ρ is used to map the representations of FP functions into FP functions; ρ is a mapping from FP atoms to FP functions. The interpretation is obtained from the function μ. Backus uses the following notation for expressions.

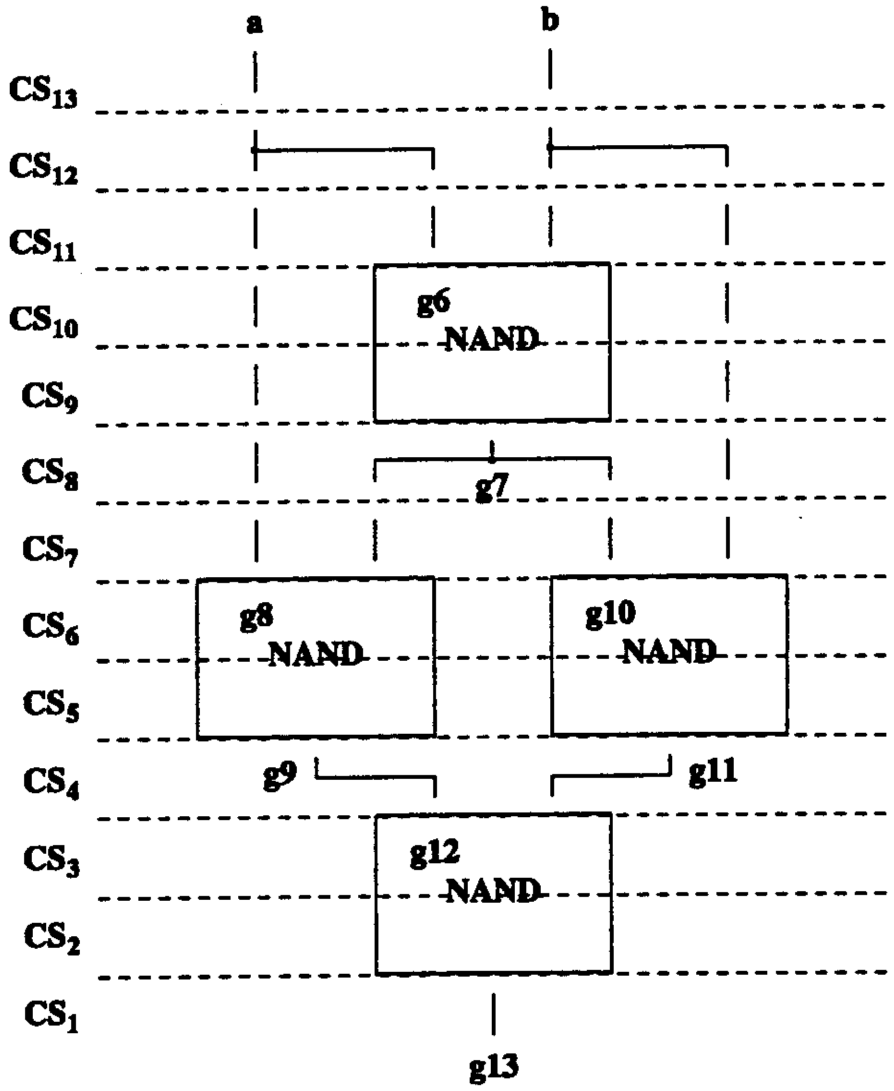


Figure 3.3 The graphical interpretation of the IF for **mxor**.

□

3.1 Using Formal FP Systems to Construct FP Interpreters

In a Formal FP system the interpretation of FP functions is embedded within the FP framework. Since the value of an FP expression is independent of its computation environment, an FP interpreter can be viewed as a function mapping FP expressions to FP objects. To formulate an FP interpreter as a function, FP functions are first represented by FP objects and then two maps, μ and ρ, defined in [Ba78], are introduced.

1. Any FP object is an expression.

2. If x and y are expressions then $<x:y>$ is an expression.

3. If $x_1, x_2 \cdots x_n$ are expressions then so is $< x_1\ x_2\ \cdots\ x_n>$.

The second type of expression is used to denote the application of x to y. Up till now, only FP expressions of the second type in in which both x and y are FP objects have been mentioned. However, to define an interpreter a broader class of expressions is used. The map μ evaluates FP expressions; it is defined as follows in [Will82]:

1) $\mu(?) = ?$

2) $\mu(a) = a$ if a is an atom

3) $\mu\ (<x_1\ x_2 \cdots x_n>) = <\mu(x_1)\ \mu(x_2) \cdots \mu(x_n)>$

4) $\mu(<x:y>)=?$ if $x=?$

5) $\mu(<x:y>)=\mu(\rho(x)(\mu(y)))$ if x is an atom

6) $\mu(<x:y>)=\mu(<x_1:<x,y>>)$ if $x=<x_1\ x_2 \cdots x_n>\quad n\geq 1$

7) $\mu(<x:y>)=\mu(<\mu(x):y>)$ otherwise

In addition a new function, apply, is used to create expressions,

apply($<x,y>$) = $<x:y>$.

Examples:

In the following F_1, F_2, F_3, and x are atoms.

The form compose (@) which forms the composition of functions can be defined as follows,

ρ(COMP) = (eq@[length@1,%2] $\rightarrow$ apply@[2@1,2];

apply@[tlr@1,apply@[last@1,2]]).

μ(<<COMP F_1 F_2 F_3>:x>)

= μ(ρ(COMP)(<<COMP F_1 F_2 F_3> x>))

= .

= .

= .

= .

= μ(f_1(μ(<<COMP F_2 F_3>:x>)))

The form build ([...]) is defined,

ρ(BUILD) = &apply@distr@[tl@1,2]

μ(<<BUILD F_1 F_2 F_3>:x>)

= μ(ρ(BUILD)(<<BUILD F_1 F_2 F_3> x>>))

= μ(&apply@distr@[tl@1,2](<<BUILD , F_1 F_2 F_3> x>))

= μ(&apply@distr(<<F_1 F_2 F_3> x>))

= μ(&apply(<<F_1, x> <F_2,x> <F_3, x>>))

= μ(<apply(<F_1, x>) apply(<F_2, x>) apply(<F_3, x>)>)

= μ(<<F_1:x> <F_2:x> <F_3:x>>)

= <μ(<F_1:x>) μ(<F_2:x>) μ(<F_3:x>)>

= <μ(ρ($F_{1)}$(μ(x))) μ(ρ(F_2)(μ(x))) μ(ρ(F_3)(μ(x)))>)

= < f1(x), f2(x), f3(x) >

□

An implementation of the maps μ and ρ is an FP interpreter; it traverses the expression, operating as the identity on FP objects and applying $\rho(x)$ to y when it encounters $<x:y>$. The actual computation is performed by the functions which the map ρ retrieves. An interpreter could be defined in FP by simply defining $\rho(F)$ to be itself for each FP function and defining each of the forms as in the examples. However, for efficiency, the computations performed by $\rho(F)$ are written in Lisp. The selection of Lisp is a natural one since FP objects exist in Lisp and many of the FP primitives have counterparts in Lisp. To implement an FP interpreter three things must be done:

1. FP expressions must be represented.

2. $\rho(F)$ must be defined in Lisp for each FP function and form.

3. The evaluation of FP expressions, the map μ, must be implemented.

To facilitate steps 1 and 3 short cuts are taken. The FP expressions to be evaluated are assumed to be of the type $<x:y>$ where x and y are both FP objects. The distinction between $<x,y>$ and $<x:y>$ is made by context; $<x:y>$ occurs as $\mu(<x,y>)$. Under these conditions the rules for evaluating FP expressions can be simplified as follows;

1) $\rho(x)(y) = ?$ if $\rho(x) = ?$

2) $\mu(<x,y>) = \rho(x)(y)$ if x is an atom

3) $\mu(<x,y>) = \rho(x_1)(<x,y>)$ if $x = <x_1\ x_2\ \cdots\ x_n>$ $n \geq 1$

Apply, is now defined by , $apply(<x,y>) = \mu(<x,y>)$.

This simplification is realized by observing that μ is the identity on FP objects, that a representation of an FP function is either an atom or its first element is an atom, and that rule 7 is unnecessary if $apply(<x,y>)$ is replaced by $\mu(<x,y>)$ whenever it occurs. To implement μ it now suffices to check if **x** is an atom and accordingly retrieve $\rho(\mathbf{x})$ or $\rho(\mathbf{x}_1)$. The major part of the interpreter is the definition of ρ. Two different methods are available for defining new functions and forms. The map ρ can be extended to new atoms by either providing a Lisp definition, or in FP, by using string replacement:

$$\rho(\text{NAME}) = \text{apply@[\%F,id]},$$

where F is the representation of the FP definition of NAME.

The map μ obtains the value of the FP expression <x:y>, but the same technique can be used to interpret other properties of <x:y>. In particular, a map μ' from FP expressions to FP objects which yields the structure of the FP expression can be defined in the same manner as μ by substituting ρ' for ρ.

1) $\rho'(\mathbf{x})(y) = ?$ if $\rho'(\mathbf{x}) = ?$

2) $\mu'(<\mathbf{x},y>) = \rho'(\mathbf{x})(y)$ if **x** is an atom

3) $\mu'(<\mathbf{x},y>) = \rho'(\mathbf{x}_1)(<\mathbf{x},y>)$ if $\mathbf{x} = <\mathbf{x}_1\ \mathbf{x}_2 \cdots \mathbf{x}_n>$ $\quad n \geq 1$

For μ', y is assumed to be of the form $<PS\ CS_1\ CS_2\ \cdots\ CS_n>$ (n=0 is allowed). If F represents an FP function and PS is the symbolic input to be operated on, then the initial input to μ' is <F <PS>>. The Lisp definitions retrieved by $\rho'(F)$ are more complicated than those of ρ. For computational primitives, cross-sections containing a box are generated. For routing primitives cross-sections with crossings are generated to route any atoms which change position or branch. In either case the output is, $<PS'\ CS_1\ CS_2\ \cdots\ CS_m\ CS_1\ CS_2\ \cdots\ CS_n>$ where, PS' is the symbolic output object of F applied to PS and $CS_1, CS_2 \cdots CS_m$ are the cross-sections corresponding to the structure of F applied to PS.

Examples:

In the following session with an FP interpreter, the structure of various FP primitives is extracted. The Lisp prompt is → and the **pplint2** is the Lisp function which corresponds to μ. The angle brackets ('<','>') are replaced by parentheses. The input FP expression appears in bold type. New symbols generated by the interpreter are labeled gN where N is some integer. **pplint2** formats the IF, printing each cross-section on a separate line, omitting the top level of parentheses, and flattening the final symbolic output (i.e. removing its structure).

```
→ (pplint2 '(andg ((a b))))
(g5)
(($ 1 2 g4 and $ a b $ g5 $))
(($ f 2 g4 and $ a b $ g5 $))
(a b)
```

```
→ (pplint2 '(notg (c)))
(g6)
(($ b 1 g7 not $ c $ g6 $))
(c)
```

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.

```
→ (pplint2 '(apndr (((a b c d e) f))))
(a b c d e f)
```

The **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, but PS is replaced by PS'.

```
→ (pplint2 '(trans (((x1 x2 x3) (y1 y2 y3) (z1 z2 z3)))))
(x1 y1 z1 x2 y2 z2 x3 y3 z3)
(x1 (* x2 * + y1 z1 ^) y2 z2 x3 y3 z3)
(x1 x2 (* x3 * + y1 z1 y2 z2 ^) y3 z3)
(x1 x2 x3 y1 (* y2 * + z1 ^) z2 y3 z3)
(x1 x2 x3 y1 y2 (* y3 * + z1 z2 ^) z3)
```

□

Several cross-sections are generated for **trans** in order to realize the new positions of the atoms in the output object. The actual symbolic output of this application would be ((x1 y1 z1)(x2 y2 z2)(x3 y3 z3)) instead of the flattenned version, (x1 y1 z1 x2 y2 z2 x3 y3 z3). The predefined Lisp functions retrieved by ρ′ generate these routing patterns. These functions can take into account the size and structure of the particular object they must route.

ρ′ must also be defined for each of the functional forms. A line which is indented is the continuation of a cross-section from the previous line.

```
→ (pplint2 '((comp notg andg) ((a b))))
(g6)
(($ b 1 g7 not $ g5 $ g6 $))
(g5)
(($ 1 2 g4 and $ a b $ g5 $))
(($ f 2 g4 and $ a b $ g5 $))
(a b)
```

The definition of composition is the same as for ρ.

```
→ (pplint2 '((build andg org norg nandg) ((c d))))
(g9 g11 g13 g15)
(($ 1 2 g8 and $ c d $ g9 $) ($ 1 2 g10 or $ c d $ g11 $)
                    ($ 1 2 g12 nor $ c d $ g13 $) ($ 1 2 g14 nand $ c d $ g15 $))
(($ f 2 g8 and $ c d $ g9 $) ($ f 2 g10 or $ c d $ g11 $)
                    ($ f 2 g12 nor $ c d $ g13 $) ($ f 2 g14 nand $ c d $ g15 $))
(c d c d c d c d)
((* c * + ^ d ^ d ^ d ^) d)
(c (* d * + ^ ^ ^ ^))
```

Build is one of the more complicated forms to specify. It takes the list of objects (IF) returned by its sub-functions applies a special transpose (which extends the objects to make them all the same length) and then it must route each atom in the input object to the sub-functions which require it. In this example, the four sub-functions generate the same number of cross-sections and the atoms c and d are routed to each of the four.

```
→ (pplint2 '((kons ((new object))) (anything)))
(g16 g17)
(($ b 1 g18 new object $ $ g16 g17 $))
nil
```

The constant form generates a box with no inputs and whose outputs are a symbolic representation of the object it is applied to. The empty list is generated as the first cross-section. The input object including any cross-sections is discarded.

```
→ (pplint2 '((apall andg) (((a1 b1) (a2 b2) (a3 b3) (a4 b4)))))
(g20 g22 g24 g26)
(($ 1 2 g19 and $ a1 b1 $ g20 $) ($ 1 2 g21 and $ a2 b2 $ g22 $)
                    ($ 1 2 g23 and $ a3 b3 $ g24 $) ($ 1 2 g25 and $ a4 b4 $ g26 $))
(($ f 2 g19 and $ a1 b1 $ g20 $) ($ f 2 g21 and $ a2 b2 $ g22 $)
                    ($ f 2 g23 and $ a3 b3 $ g24 $) ($ f 2 g25 and $ a4 b4 $ g26 $))
(a1 b1 a2 b2 a3 b3 a4 b4)
```

Apall is similar to **build** in that the list of IFs of each of its sub-computations is obtained and then an extending transpose is applied. It differs in that no routing of the input object is required.

```
→ (pplint2 '((rins andg) ((a1 a2 a3 a4 a5 a6))))
(g36)
(($ 1 2 g35 and $ a1 g34 $ g36 $))
(($ f 2 g35 and $ a1 g34 $ g36 $))
(a1 g34)
(a1 ($ 1 2 g33 and $ a2 g32 $ g34 $))
(a1 ($ f 2 g33 and $ a2 g32 $ g34 $))
(a1 a2 g32)
(a1 a2 ($ 1 2 g31 and $ a3 g30 $ g32 $))
(a1 a2 ($ f 2 g31 and $ a3 g30 $ g32 $))
(a1 a2 a3 g30)
(a1 a2 a3 ($ 1 2 g29 and $ a4 g28 $ g30 $))
(a1 a2 a3 ($ f 2 g29 and $ a4 g28 $ g30 $))
(a1 a2 a3 a4 g28)
(a1 a2 a3 a4 ($ 1 2 g27 and $ a5 a6 $ g28 $))
(a1 a2 a3 a4 ($ f 2 g27 and $ a5 a6 $ g28 $))
(a1 a2 a3 a4 a5 a6)
```

The Lisp function retrieved for Right Insert obtains the IFs of the sub-function inserted on the symbolic input object. Routing is provided by adding each input atom to the cross-sections up until the cross-section in which it is consumed by the sub-function.

```
→ (pplint2 '((tins andg) ((b1 b2 b3 b4 b5 b6 b7 b8 b9 b10))))
(g54)
(($ 1 2 g53 and $ g44 g52 $ g54 $))
(($ f 2 g53 and $ g44 g52 $ g54 $))
(g44 g52)
(($ 1 2 g43 and $ g40 g42 $ g44 $) ($ 1 2 g51 and $ g48 g50 $ g52 $))
(($ f 2 g43 and $ g40 g42 $ g44 $) ($ f 2 g51 and $ g48 g50 $ g52 $))
(g40 g42 g48 g50)
(($ 1 2 g39 and $ g38 b3 $ g40 $) ($ 1 2 g41 and $ b4 b5 $ g42 $)
                ($ 1 2 g47 and $ g46 b8 $ g48 $) ($ 1 2 g49 and $ b9 b10 $ g50 $))
(($ f 2 g39 and $ g38 b3 $ g40 $) ($ f 2 g41 and $ b4 b5 $ g42 $)
                ($ f 2 g47 and $ g46 b8 $ g48 $) ($ f 2 g49 and $ b9 b10 $ g50 $))
(g38 b3 b4 b5 g46 b8 b9 b10)
(($ 1 2 g37 and $ b1 b2 $ g38 $) b3 b4 b5 ($ 1 2 g45 and $ b6 b7 $ g46 $) b8 b9 b10)
(($ f 2 g37 and $ b1 b2 $ g38 $) b3 b4 b5 ($ f 2 g45 and $ b6 b7 $ g46 $) b8 b9 b10)
(b1 b2 b3 b4 b5 b6 b7 b8 b9 b10)
```

Tree insert is similar in its ρ and ρ' definitions. The latter differs only in that atoms which are not consumed at a particular level must be routed to the next level by adding them to intervening cross-sections. □

As discussed earlier, certain restrictions must be placed on the use of the conditional. It is assumed that only structural predicates occur when ρ' encounters a conditional. The Lisp function retrieved by ρ' evaluates the predicate by applying μ to the FP expression formed with the predicate and PS. This Lisp function then applies μ' to the FP expression formed with the selected sub-function and the input object.

```
→ (pplint2 '((condl atm notg nandg) ((a b))))
(g56)
(($ 1 2 g55 nand $ a b $ g56 $))
(($ f 2 g55 nand $ a b $ g56 $))
(a b)
```

```
→ (pplint2 '((condl atm notg nandg) (a)))
(g57)
(($ b 1 g58 not $ a $ g57 $))
(a)
```

ρ′ can be extended to new functions by the same two methods as ρ: by providing either a Lisp or FP definition. The former method provides a mechanism to direct functions to be represented as a boxes. A function may have a different definition with respect to ρ and ρ′. This mechanism is used to deal with conditionals which are not directives but rather part of the computation. A simple switch can be defined in FP by,

```
eq@[1,%1]→2;3,
```

and represented as a box with three inputs and one output or it can be implemented with gates,

```
org@&andg@[[1,2],[notg@1,3]].
```

Figure 3.4 illustrates these choices. Of course the correspondence between a function's behaviour and structure is no longer assured if the definitions provided to ρ and ρ′ are not the same.

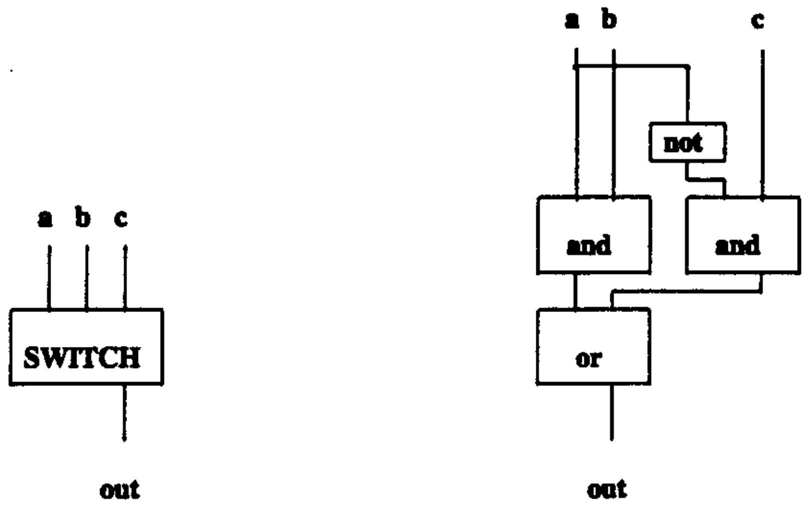


Figure 3.4 Two representations for a conditional.

3.2 Graphical Interpretation of the Intermediate Form

To obtain a visual representation of the structure of an FP expression, the IF must be mapped to fixed geometry. The procedure detailed in this this section is designed to produce a picture efficiently. It does not attempt to optimize the placement of the 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 3.2). 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 by manipulating 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 $\mathbf{n}_1$ to node $\mathbf{n}_2$ corresponds to the constraint expressed by the inequality,

$$\mathbf{p}_1 + \mathbf{d} \leq \mathbf{p}_2,$$

where $\mathbf{p}_1$ and $\mathbf{p}_2$ are the positions (coordinates) of $\mathbf{n}_1$ and $\mathbf{n}_2$ respectively. If **d** is non-negative then $\mathbf{n}_2$ must be to the right of $\mathbf{n}_1$ by at least **d**. If **d** is negative then $\mathbf{n}_1$ cannot be to the right of $\mathbf{n}_2$ by more than $-\mathbf{d}$. 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 : $<* \, \mathbf{w} \, * \, \mathbf{u}_1 \, \mathbf{u}_2 \, , \, . \, . \, . \, , \, \mathbf{u}_n>$

For a crossing the list of $\mathbf{u}_i$'s is traversed. Each $\mathbf{u}_i$ 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 $\mathbf{u}_1$ and $\mathbf{u}_n$ 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,

$$1 + \max\{\#\ \textbf{inputs},\ \#\ \textbf{outputs}\}.$$

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 3.5 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).

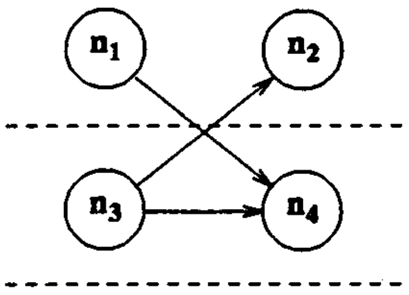


Figure 3.5 Constraints generated when node n_4 is processed.

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 3.6 shows the constraints between a box node and its input and output nodes.

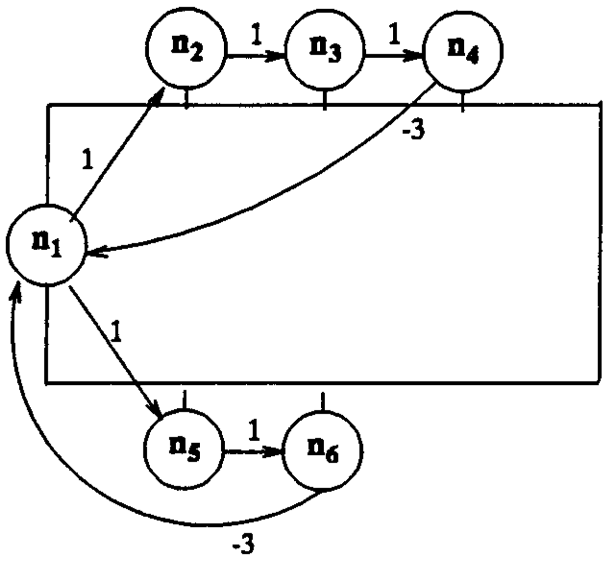


Figure 3.6 Constraints generated for node n_1, a box.

Example:

Figure 3.7 contains the elements of the graph for the example in Figure 3.3 and Figure 3.8, its constraint graph.

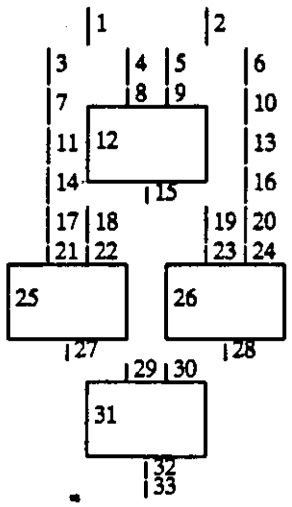


Figure 3.7 Elements for example of Figure 3.3.

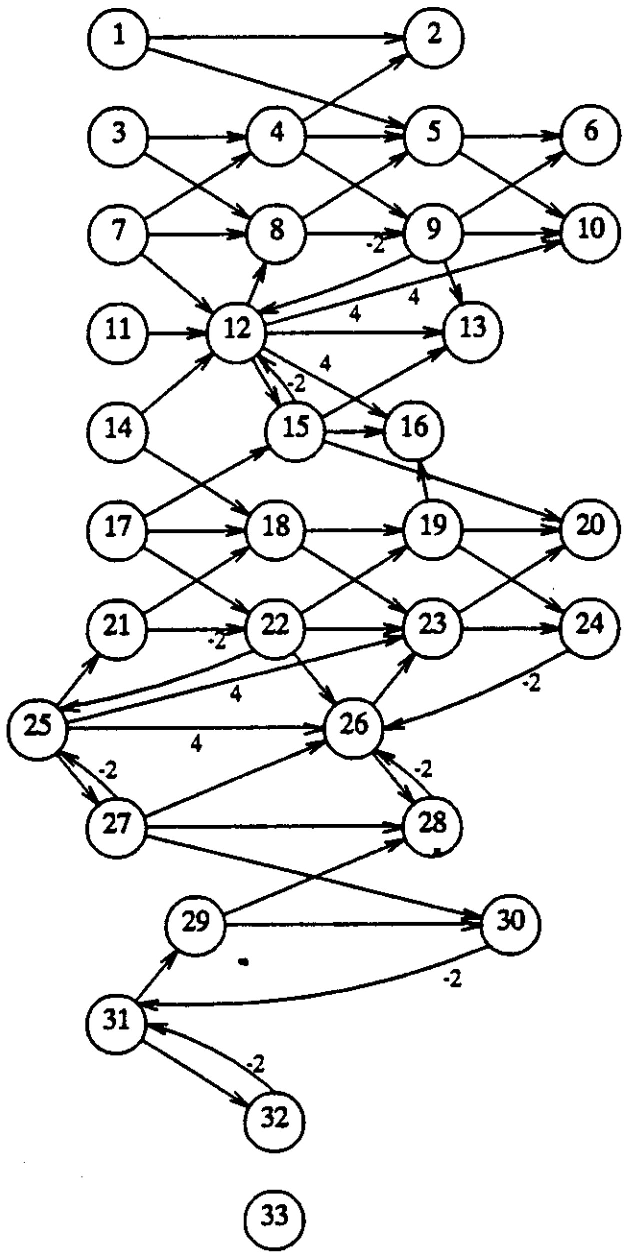


Figure 3.8 Constraint graph for example of Figure 3.3.

□

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. This solution is optimal in that no other set of positions satisfying the inequalities can assign a smaller position to any node. In [LiWo83], 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.

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.

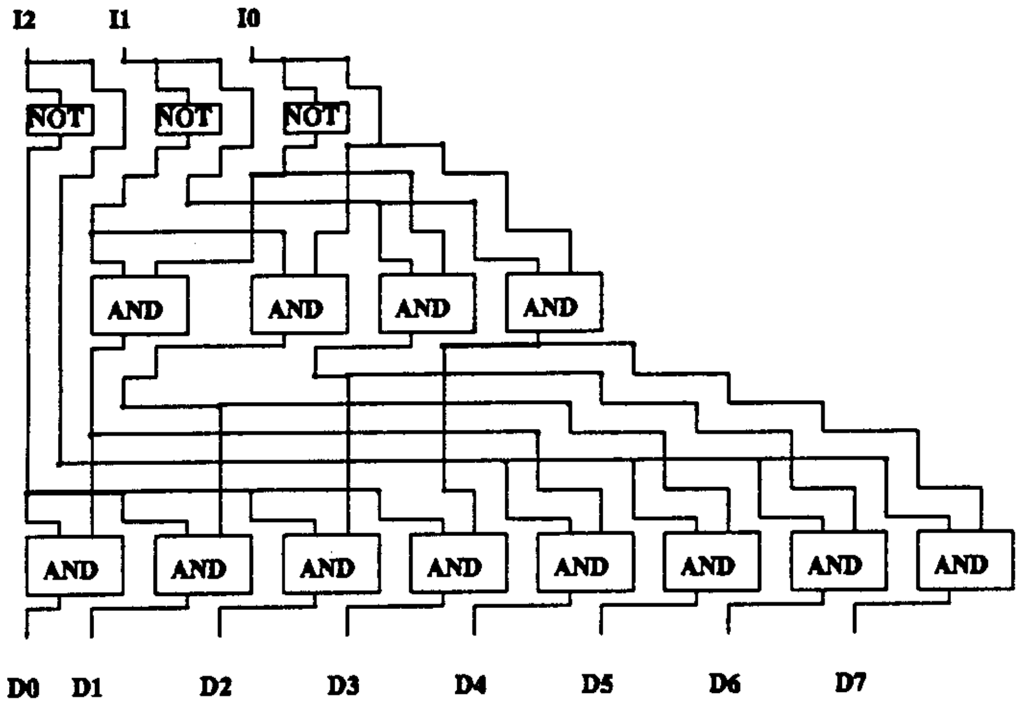
(*) $\quad p_2 \leftarrow \max\{p_2, p_1 + d\}$

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 (*). The acyclic subgraph is traversed again updating positions as in (*). 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. However when the graph is not consistent, the positions obtained are distorted, since they maybe the result of traversing positive cycles several times. In this application the constraint graph is often not consistent and this case must be dealt with. A heuristic algorithm is used which attempts to identify inconsistencies early and adjust them before the positions of other nodes become distorted. 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 (*). If this back edge is again not satisfied after subsequent traversals, the algorithm ignores it. Ignoring a back edge corresponds to widening a box. Unfortunately, adjusting the position of the destination node to satisfy a back edge, may violate previously satisfied back edges causing them to be unnecessarily widened after the next traversal.

This heuritic 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 heuristic algorithm is resorted to only when the graph is found to be inconsistent.

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 example of the routing before and after "straightening" the wires is shown in Figure 3.9.



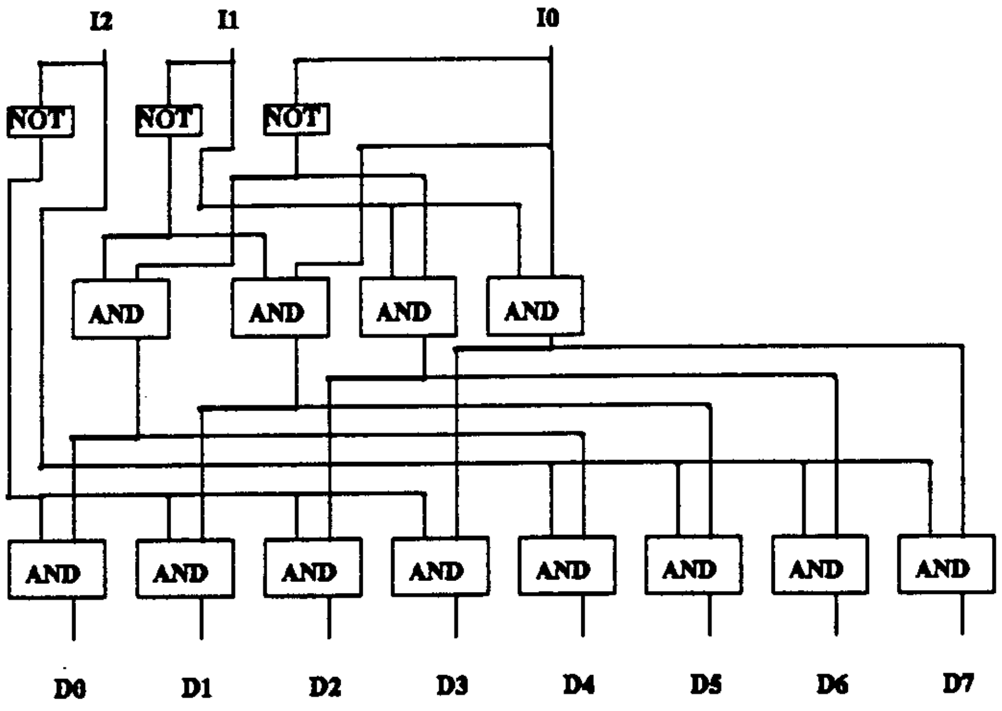


Figure 3.9 Before and after straightening the wires.

To complete the example, Figure 3.10 is the picture generated for the example of Figure 3.3

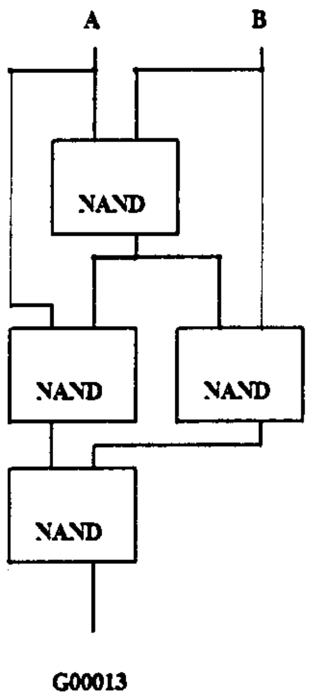


Figure 3.10 Sketch obtained for example of Figure 3.3, **mxor**.

4. Applications

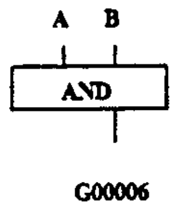
In these sections examples of algorithms written in FP and their sketches are presented to illustrate the correspondence between programming style in FP and resulting sketches. The performance of the heuristic compaction algorithm is most clearly visible in the last section (the FFT).

4.0 Introductory Examples

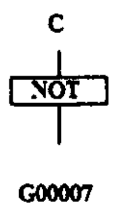
Before tackling "real" algorithms, the sketches of the FP functions presented earlier as examples are displayed. These correspond to the IFs from Section 3.1.

Examples:

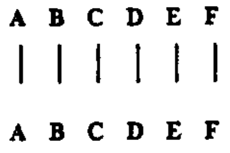
1. → (clint2 '(andg ((a b))))



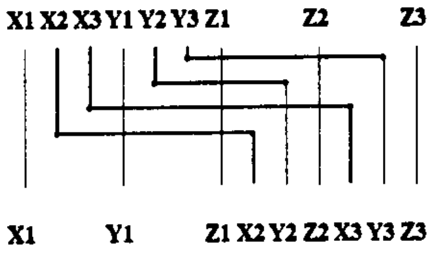
2. → (clint2 '(notg (c)))



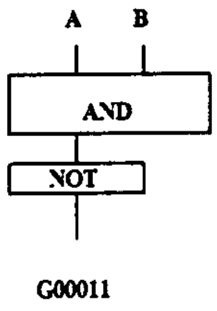
3. → (clint2 '(apndr (((a b c d e) f)))



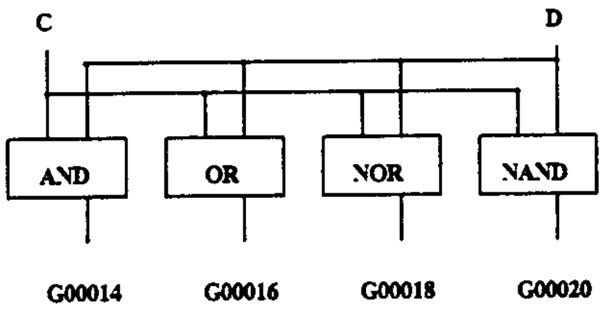
4. → (clint2 '(trans (((x1 x2 x3) (y1 y2 y3) (z1 z2 z3)))))



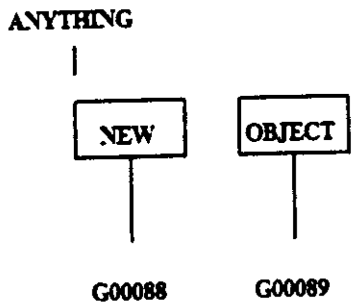
5. → (clint2 '((comp notg andg) ((a b))))



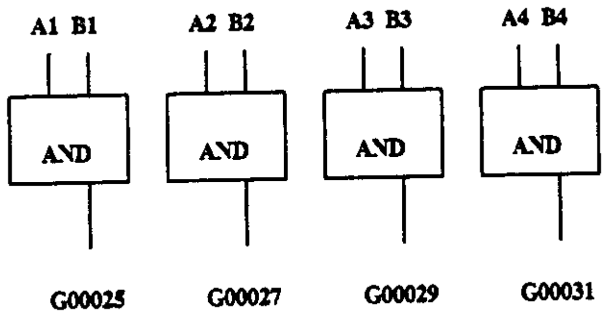
6. → (clint2 '((build andg org norg nandg) ((c d))))



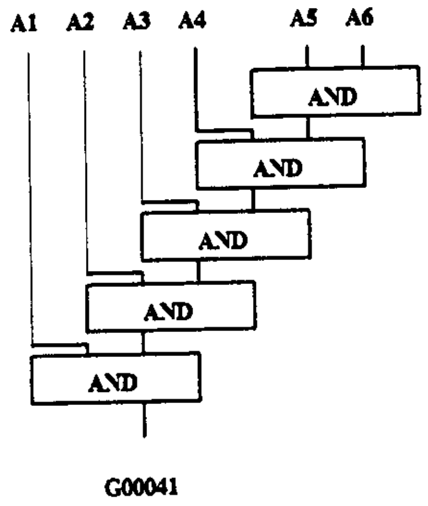
7. → (clint2 '((kons ((new object))) (anything)))



8. → (clint2 '((apall andg) (((a1 b1) (a2 b2) (a3 b3) (a4 b4)))))



9. → (clint2 '((rins andg) ((a1 a2 a3 a4 a5 a6))))



10. → (clint2 '((tins andg) ((b1 b2 b3 b4 b5 b6 b7 b8 b9 b10))))

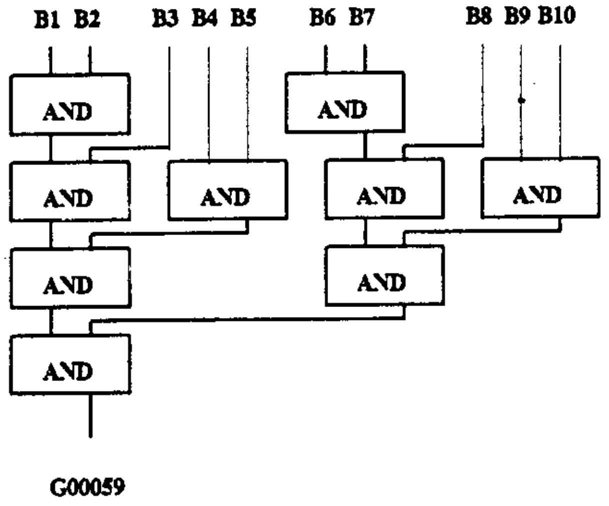


Figure 4.0.1

In writing FP programs to obtain sketches, the following two aspects of programming style should be considered.

Hanging Wires

Hanging wires occur when arguments are "brought" to a function which does not "need" them. Examples of FP primitives which may generate hanging wires, are **tl**, **tlr** and any selector (i.e. **1**, **2**, **3**, etc.). In addition the **Kons** form generates a function which does not use its input at all. This may result in a hanging wire as in Example 7. In addition to occupying unnecessary space, hanging wires present a problem for the compaction algorithm, since wires must be matched from one cross-section to the next in setting up the constraint graph. One solution is to write a post processor which removes hanging wires, but for the moment FP expressions must be written so that hanging wires do not occur. This does not mean that **tl**, **tlr**, and **Kons**, cannot be used. They can be used within the form **Build**, as long as each atom in the input object to the **Build** is needed by at least one sub-function of the **Build**. The following example of an FP function with hanging wires actually occurred in the specification of an algorithm. The 12 rightmost wires are left hanging.

[&tl,1@trans] (build (apall tl) (comp 1 trans))

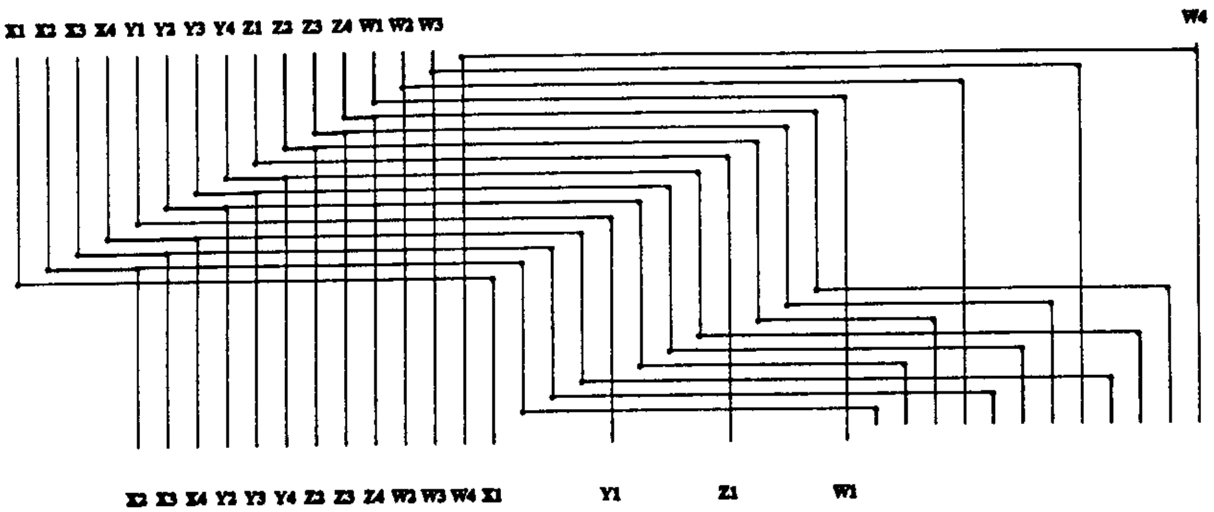

X1 X2 X3 X4 Y1 Y2 Y3 Y4 Z1 Z2 Z3 Z4 W1 W2 W3
W4
X2 X3 X4 Y2 Y3 Y4 Z2 Z3 Z4 W2 W3 W4 X1
Y1
Z1
W1

Figure 4.0.2 Hanging Wires

The hanging wires are generated by the selector 1, which does not need the remainder of the atoms in the object produced by **trans**. In execution, this function is wasteful of time and space by unnecessarily computing the function **trans**. A better version is obtained by replacing **(comp 1 trans)** by **(apall 1)**.

Routing Functions

The second aspect concerns the routing implied by the FP specification. **Build** is the only form for which routing maybe required. The routing pattern currently provided for the **Build** busses one atom at a time starting with the rightmost. See Example 6. Any other routing is the direct result of the use of routing primitives. This can lead to inefficient routing when the sub-functions of a **Build** are themselves routing functions. For example,

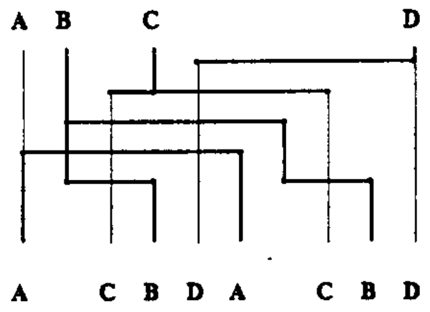


Figure 4.0.3 **[shuffle,shuffle]** applied to ((**a** **b**) (**c** **d**))

The atoms are first routed to each of the **shuffles** and then the routing of each **shuffle** is performed. To resolve this problem and facilitate the writing of efficient routing functions, a new form, **sBuild** represented by ⟦ , ⟧, is introduced. It differs from **Build** in that its sub-functions must all be routing functions and its ρ' definition evaluates the sub-functions using μ instead of μ'. The effect is to ignore the routing implied by the sub-functions, and route the input object to the output object using the routing pattern provided with **Build**. **sBuild** is equivalent to **Build** in terms of input-output behaviour.

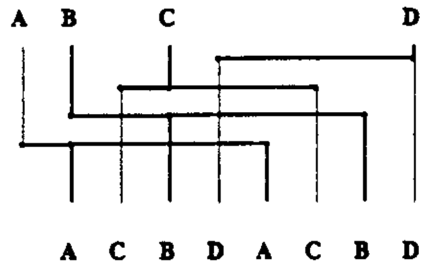
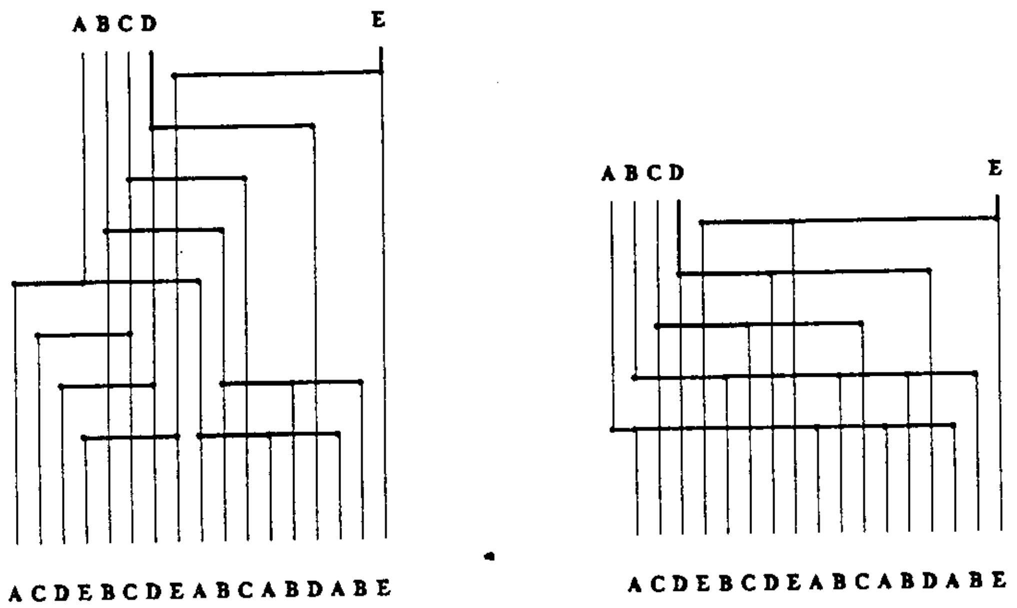


Figure 4.0.4 ⟦**shuffle,shuffle**⟧ applied to ((**a** **b**) (**c** **d**))

In cases where some of the routing can be performed in parallel (using the same horizontal tracks), **Build** maybe preferable to **sBuild**. Two routing functions are presented in Figure 4.0.5. In the first, **sBuild** is preferable while **Build** is better in the second.


A B C D
E
A C D E B C D E A B C A B D A B E
A B C D
E
A C D E B C D E A B C A B D A B E
a) [distr,distl] vs. ⟦distr,distl⟧

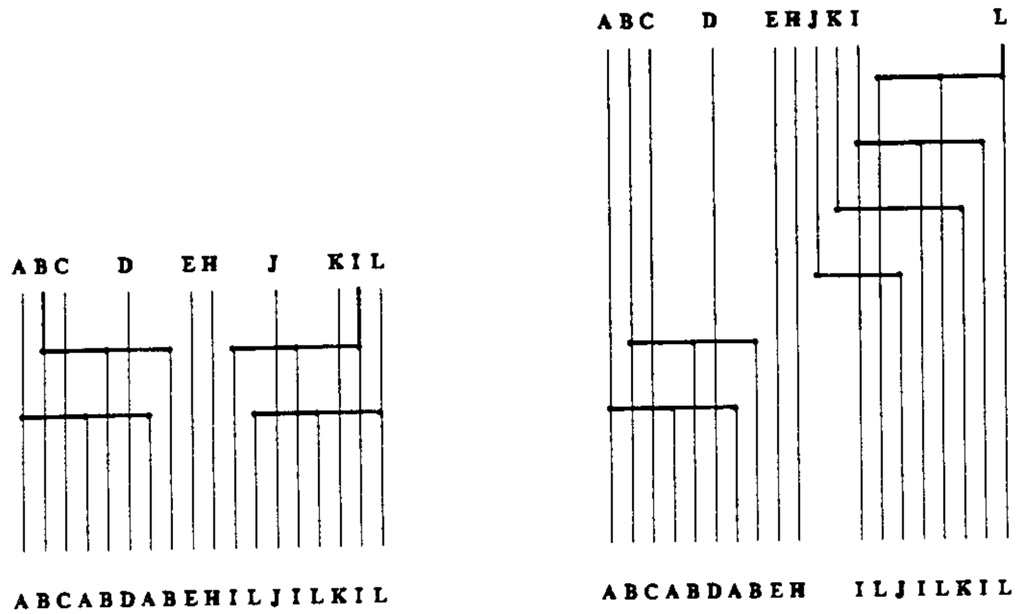

A B C D E H J K I L
A B C A B D A B E H I L J I L K I L
A B C D E H J K I L
A B C A B D A B E H I L J I L K I L
b) [distl@1,distr@2] vs. ⟦distl@1,distr@2⟧

Figure 4.0.5 [...] vs. ⟦ ... ⟧

Since only the input and output objects of **sBuild** determine its routing, the sub-functions of an **sBuild** can produce hanging wires, as long as every input to the sBuild is needed by one of its sub-functions. This alleviates the burden of writing FP functions which do not produce hanging wires.

In the following sections, the output labels generated by the interpreter (e.g. G00046) have been replaced with more appropriate ones. In addition, the FP definitions are given in the more compact Berkeley FP syntax.

4.1 Decoder

The FP specification for a decoder was one of the first algorithms examined and has suffered numerous examinations since. The original FP specification written by Lahti[La81] is,

```
{decode !decstage@&onedecode}

{onedecode [notg id]}

{decstage &andg@cncat@&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.

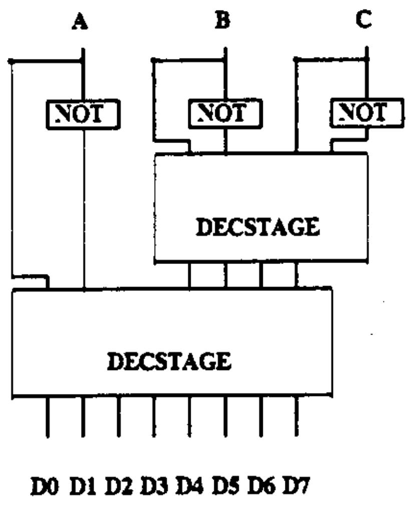


Figure 4.1.1 Decoder with Decstage as a primitive.

The input to **decstage** is,

$$<<\bar{x}_i\ x_i><d_0\ d_1\ \cdots\ d_{m-1}>> \quad \text{where} \quad m=2^{i-1}$$

and the output is,

$$<f_0\ f_1\ \cdots\ f_{2m-1}> \quad \text{where for} \quad 0<j<m-1, \quad f_j=\bar{x}_j \cdot d_j \quad \text{and,} \quad f_{j+m}=x_j \cdot d_j$$

This is accomplished by transforming the input object into,

$$<<\bar{x}_i\ d_0><\bar{x}_i\ d_1>\ \cdots\ <\bar{x}_i\ d_{m-1}><x_i\ d_0><x_i\ d_1>\ \cdots\ <x_i\ d_{m-1}>>$$

and applying **andg** to each pair in this list. The picture resulting from this algorithm is in Figure 4.1.1.

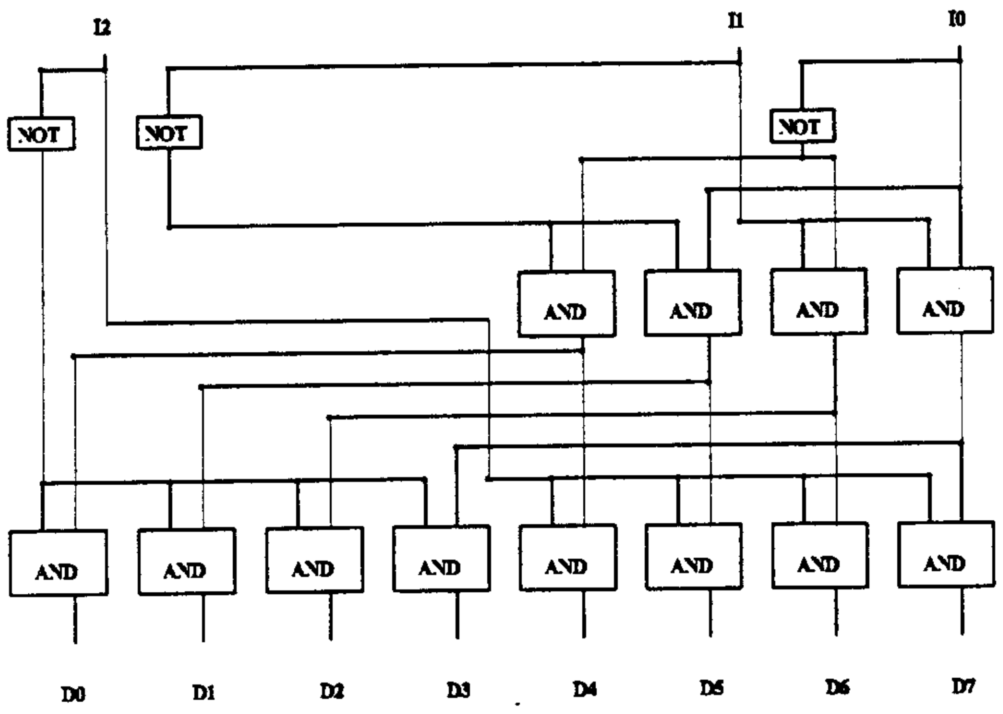


Figure 4.1.2 Decoder with andg and notg as primitives.

The compaction algorithm pushes the boxes to the left. In this case the not gates are pulled unnecessarily to the left. Space would be saved by pushing them back to the 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 4.1.3 is the sketch obtain by *flipping* the intermediate form of the sketch in Figure 4.1.2.

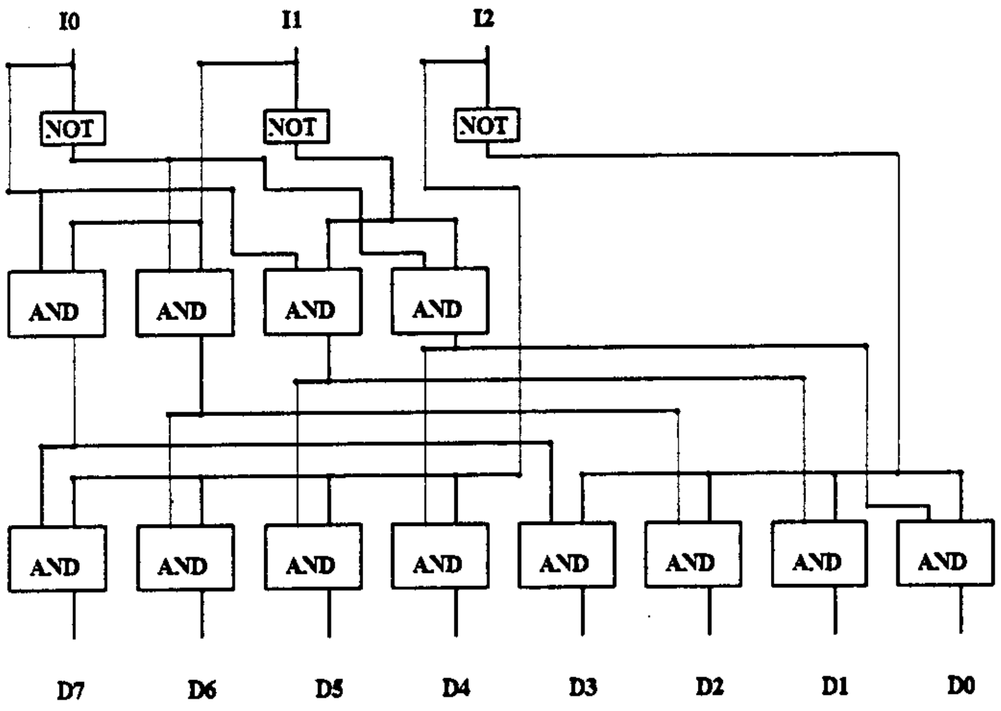
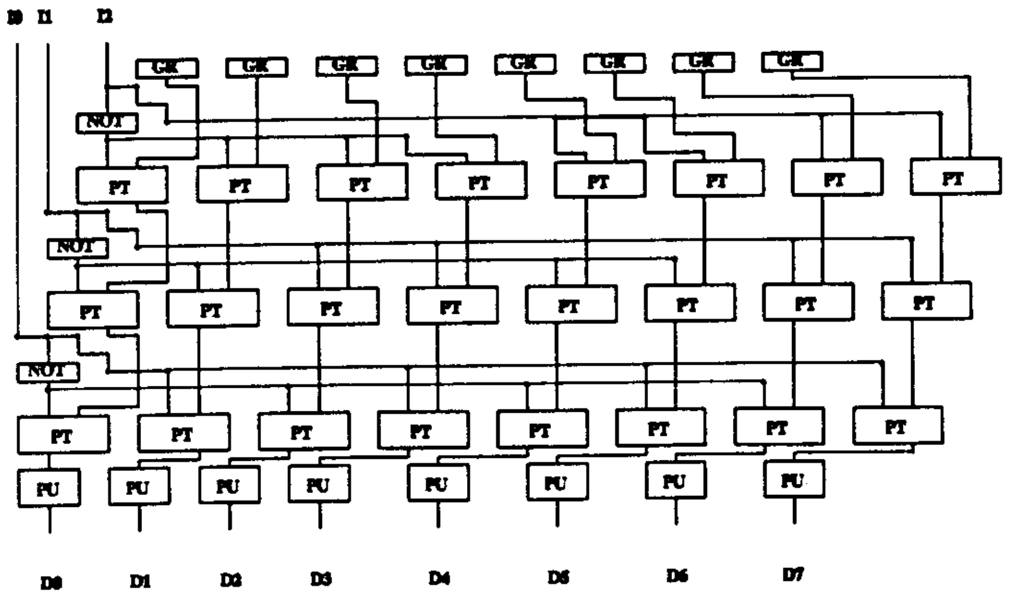


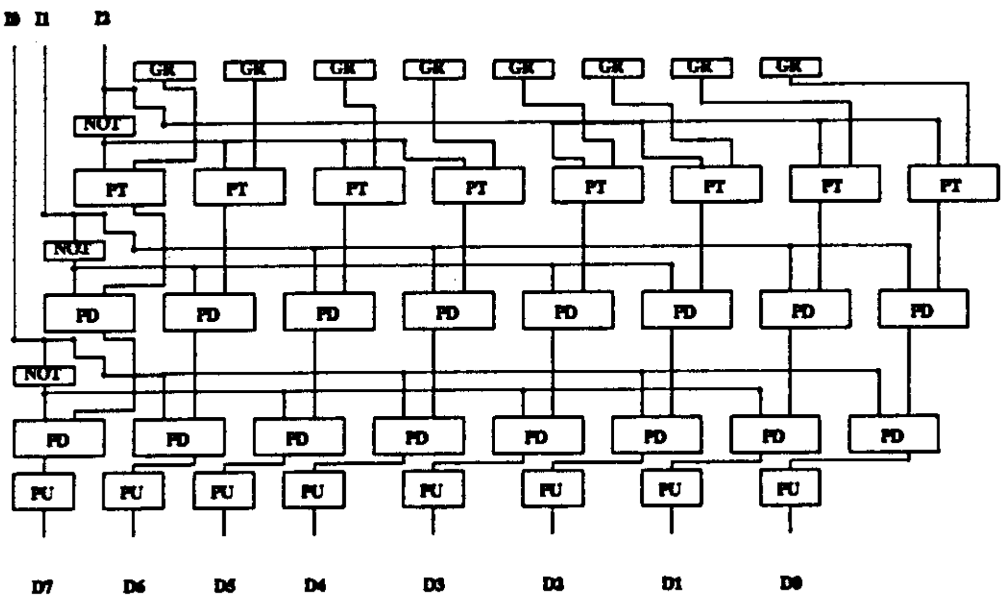
Figure 4.1.3 The *flipped* IF of Figure 4.1.2.

Unfortunately neither of these pictures corresponds to a realistic design since they are based on a recursive switching function definition. The more realistic designs in Figure 4.1.4 correspond to those of [MeCo80] but their FP definitions are more complicated.

```
# Nor-decoder ###################
{nordecode &PU@repeat@[1,split@2]@setup}
{repeat (nul@1->cncat@2;repeat@stage)}
{stage [tlr@1,cncat@&(split@&PT)@odistl@[notg@last@1,edistl@[last@1,2]]]}
# Nand-decoder ###################
```

```
{nanddecode &PU@repeatpd@stage@[1,split@2]@setup}

{repeatpd (nul@1->cncat@2;repeatpd@stagepd)}

{stagepd [tlr@1,cncat@&(split@&PD)@odistl@[notg@last@1,edistl@[last@1,2]]]}

# Functions used by both decoders ###############

{edistl 1@⟦cncat@&[1@2,distl@[1,2@2]]@distl@[1,pair@2]⟧}

{odistl 1@⟦cncat@&[distl@[1,1@2],2@2]@distl@[1,pair@2]⟧}

{setup [id,&GR@expand@⟦id,[tl@[id]]⟧]}

{expand (eql@[lngth@1,%0]->2;expand@⟦tl@1,cncat@[2,2]⟧)}

{PT org@[notg@1,2]}

{PD andg@[notg@1,2]}

{PU id}

{GR %0}
```





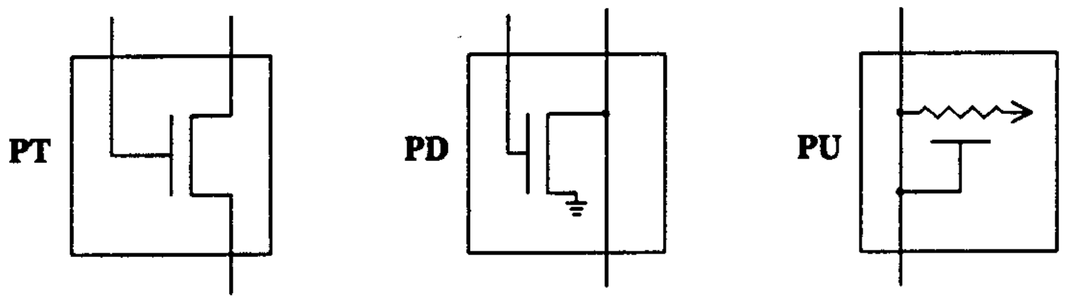


Figure 4.1.4

This specification is in terms of non-functional primitives, (i.e. pass transistors, pullups and pulldowns). However 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 programmer is responsible for insuring that these elements do in fact correspond to their FP definitions in practice.

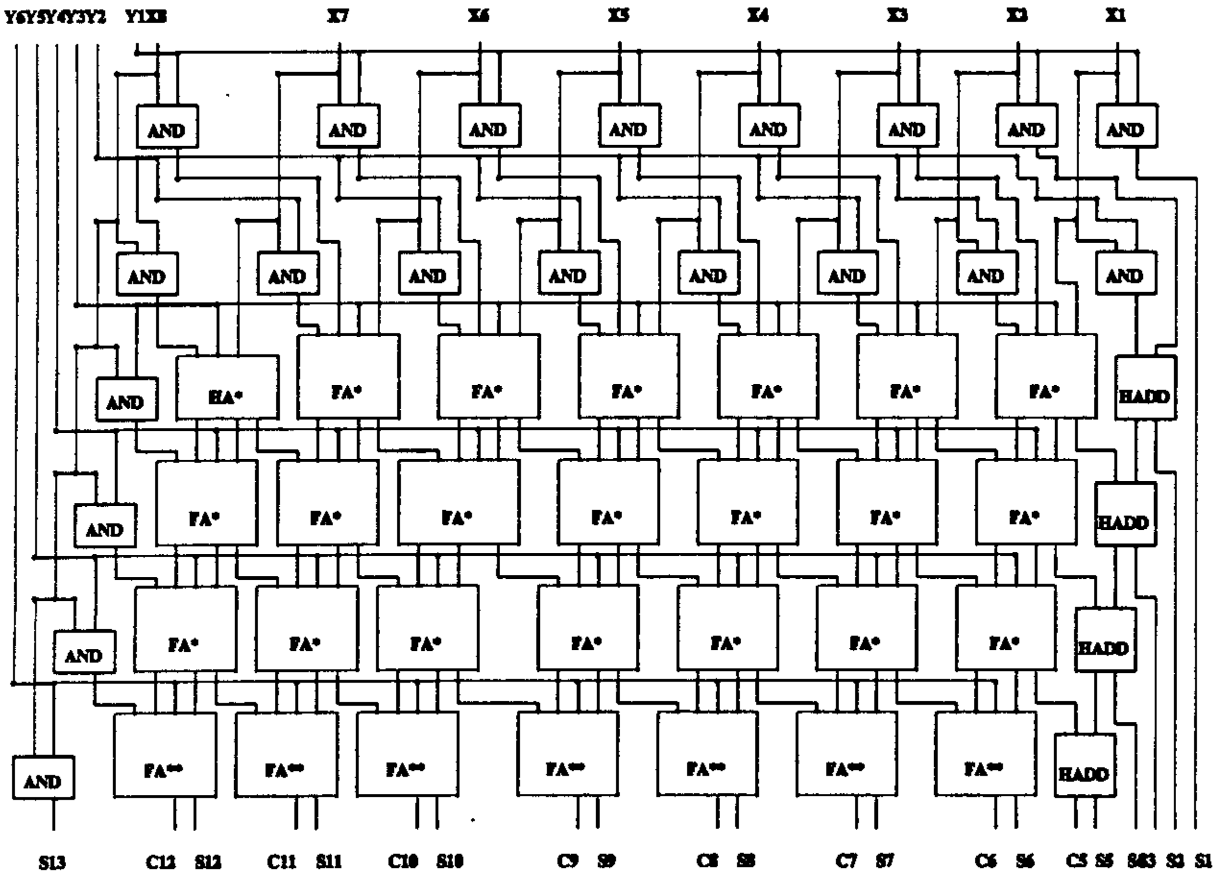
4.2 Carry-Save Array Multiplier

The following is an FP specification of a carry-save array multiplier. The specification is generic; it will multiply any two bit vectors of length greater than 3. Basically 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 the bit of the multiplier. The output consists of two bits per column for the n leftmost columns and a single bit for the m-1 rightmost columns; an adder should be applied to the leftmost m columns to obtain the final sum.

```
# Carry Save Array Multiplier

#            multiplier                  multiplicand
# input <<ym y(m-1).....y2 y1> <xn x(n-1) ....x2 x1>>  m>3 and n>2.

# output <<s(m+n-1) c(m+n-2)> <s(m+n-2) c(m+n-3)> ...<s(m) c(m-1)> <s(m-1)... s1>>

# ******** the function *******
{csmult ckstage@stage3@stage2@stage1}

# set up initial structure
{stage1 [tlr@1,cncat@&[1,andg]@pair@cncat@[[1@2],cncat@distl@[last@1,tl@2],[last@1]]]}

{stage2 [1,cncat@[[1,[andg]]@1@2,cncat@&[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
{ckstage (eql@[lngth@1,%1]->laststage;ckstage@normalstage)}

{normalstage regroup@csave@setup}

{laststage lastregroup@lastcsave@setup}

{setup [tlr@1,
        [[1@2,last@1],&align1@distl@[last@1,pair@tlr@tl@2]],
        apndl@[last@2,3]]}

{align1 [1@2,[1,2@2]]}

{csave [1,cncat@[op0@1,&op2@2]@2,apndl@[hadd@1,tl]@3]}

{csave1 [1,cncat@[op0@1,[op1@1@2],&op2@tl@2]@2,apndl@[hadd@1,tl]@3]}

{lastcsave [cncat@[[andg@1],&lop2@2]@2,apndl@[hadd@1,tl]@3]}

{regroup [1,apndr@[tlr@2,[last@2,1@1@3]],apndl@[2@1@3,tl@3]]@[1,regp@2,3]}

{regp (eql@[%2,lngth]->id;cncat@[[1,[2,1@1@3]],
                                regp@cncat@[[2@1@3,2@3],tl@tl@tl]])}

{lastregroup [pair@tlr@apndl@[1,cncat@tl]@2,apndl@[2@last@2,3]]}

# 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]}

{op0 [1,andg]}

{hadd [andg,org]}
```

Figure 4.2.1 is the sketch obtained of the function **csmult** with the functions **HA***, **FA***, **FA****, and **HADD** represented as primitives. The internal specifications of the functions **HA***, **FA***, and **FA**** are also given.



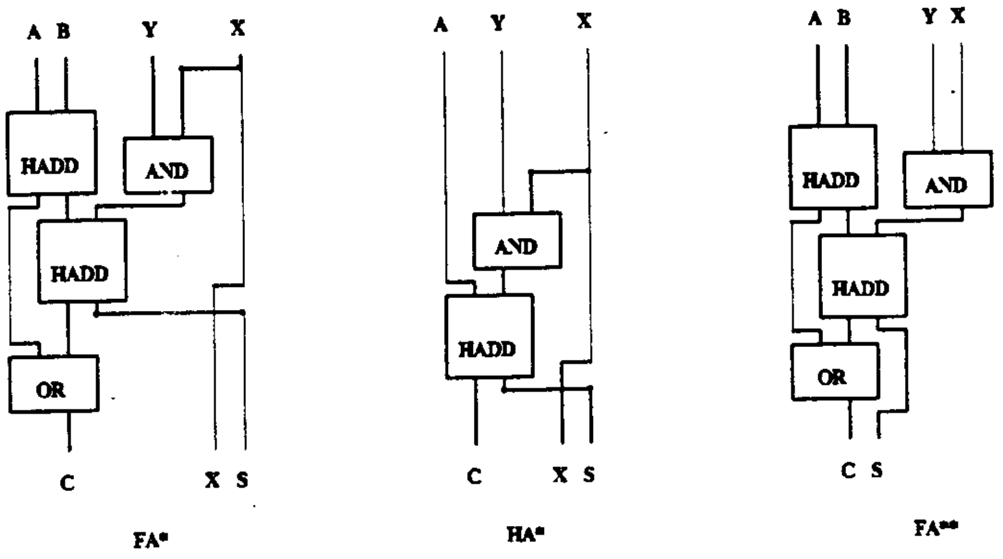


Figure 4.2.1 Csmult with **HA***, **FA*** and **FA**** as primitives.

By defining each FA* directly in terms of **hadd** and gates, vertical space is saved. If instead, FA* were defined as,

{FA* FA@apndr@[1,andg@2]}

then the **andg** would have been placed above the **hadd**. Whenever the form **Compose** is invoked, the arguments are brought to the same vertical level resulting in a waste of space. Figure 4.2.2 contains a sketch of the same FP expression without **HA***, **FA*** and **FA**** marked as primitives; 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 may adapt to its environment.

The FP specification of this algorithm is one of the most complicated. This is due to the difficulty in efficiently writing the structural transformations required between stages, and the need to specify each of the first three stages, the last stage and the other stages separately since they are all slightly different. The form **sBuild** was useful in achieving the routing between stages. The difficulty in writing FP functions is often in determining the exact structure of the object being passed from one function to the next. Especially when functions are nested several times within **Builds**.

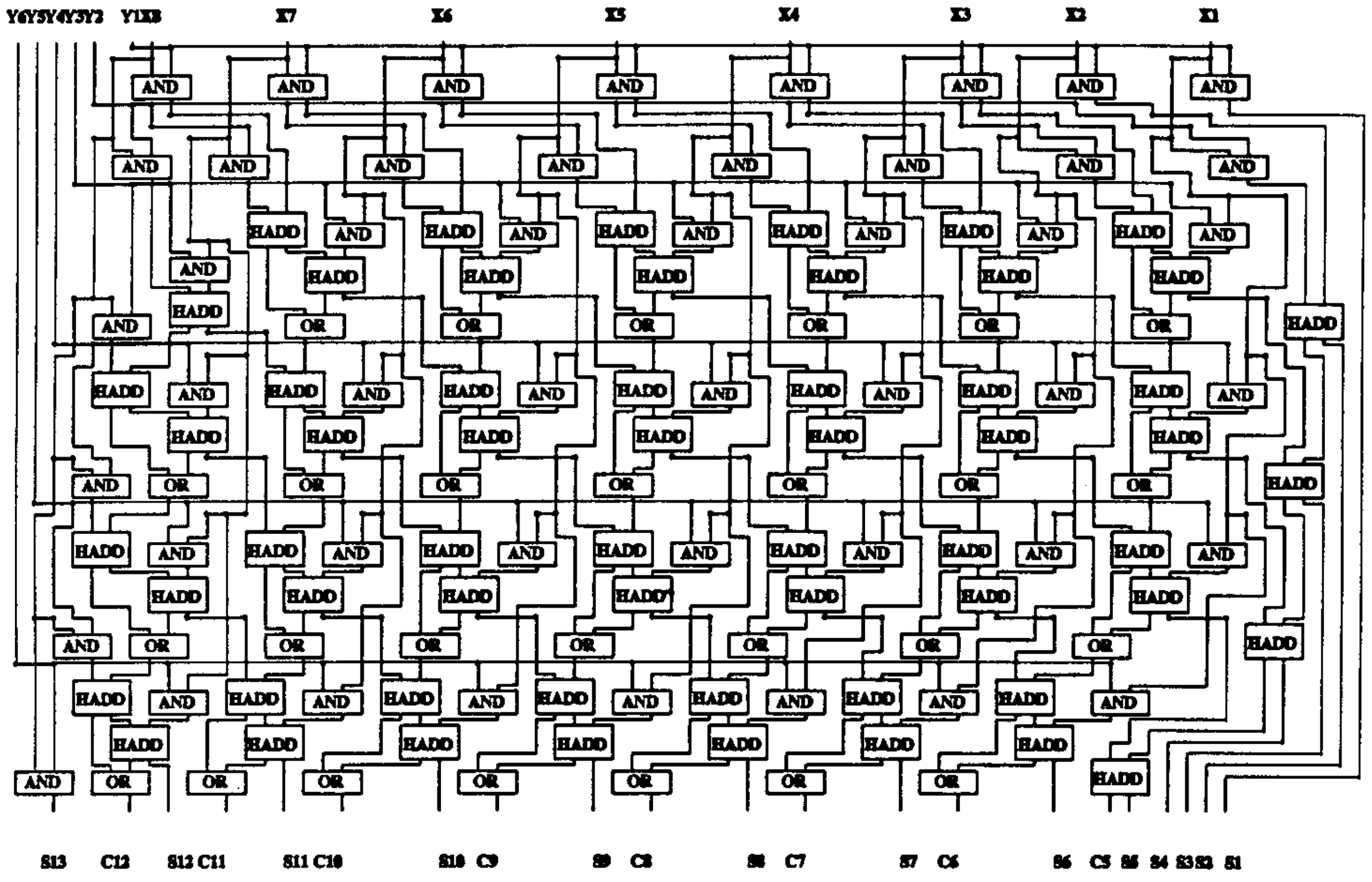

Y6Y5Y4Y3Y2 Y1X8
X7
X6
X5
X4
X3
X2
X1
AND
OR
HADD
S13 C12 S12 C11 S11 C10 S10 C9 S9 C8 S8 C7 S7 C6 S6 C5 S5 S4 S3 S2 S1

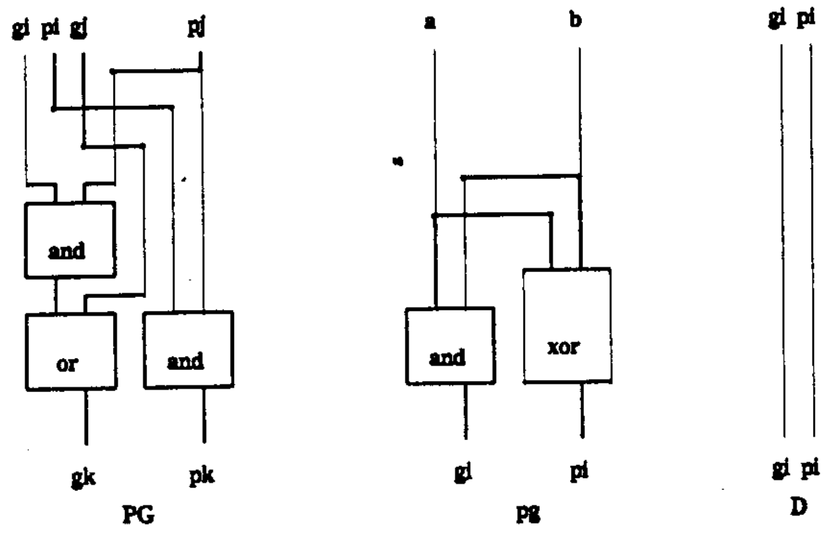
Figure 4.2.2 Csmult with **andg**, **org** and **hadd** as primitives.

4.3 Brent-Kung Adder

This algorithm [BrKu80] is designed to add any two bit vectors of size 2^n. It determines the carry for each column. To obtain the sum this carry must be combined with the two input bits of the column.

```
# input = <<a_1 b_1><a_2 b_2><a_3 b_3> · · · <a_n b_n>>    wheren = 2^n
#
# a_i and b_i are bits

{bkadd firsthalf@&pg}

{stage1 cncat@[&D@1,&D@tlr@2,[PG@[last@1,last@2]]]}

{firsthalf (eq@[length,%1]->secondhalf@split@1;firsthalf@&stage1@pair)}


{secondhalf (eq@[length,%1]@1->done;
                                    secondhalf@cncat@[split@&D@1,stage2]
                                    @apndr@[cncat@&[id,last]@tlr,last])}

{stage2 cncat@&([apndr@[&D@tlr@1@2,
                         PG@[1,last@1@2]],
               &D@2@2]
              @[1,split@2])
          @pair@tl}

{done &1}

{pg [[andg,xorg]]}

{PG [org@[andg@[1@1,2@2],1@2], andg@[2@1,2@2]]}

{D id}
```

The specification is based on three primitives, **PG**, **pg** and **D**. Figure 4.3.1 contains the FP definitions of these primitives and the computation of the carries for the addition of two 16-bit vectors.



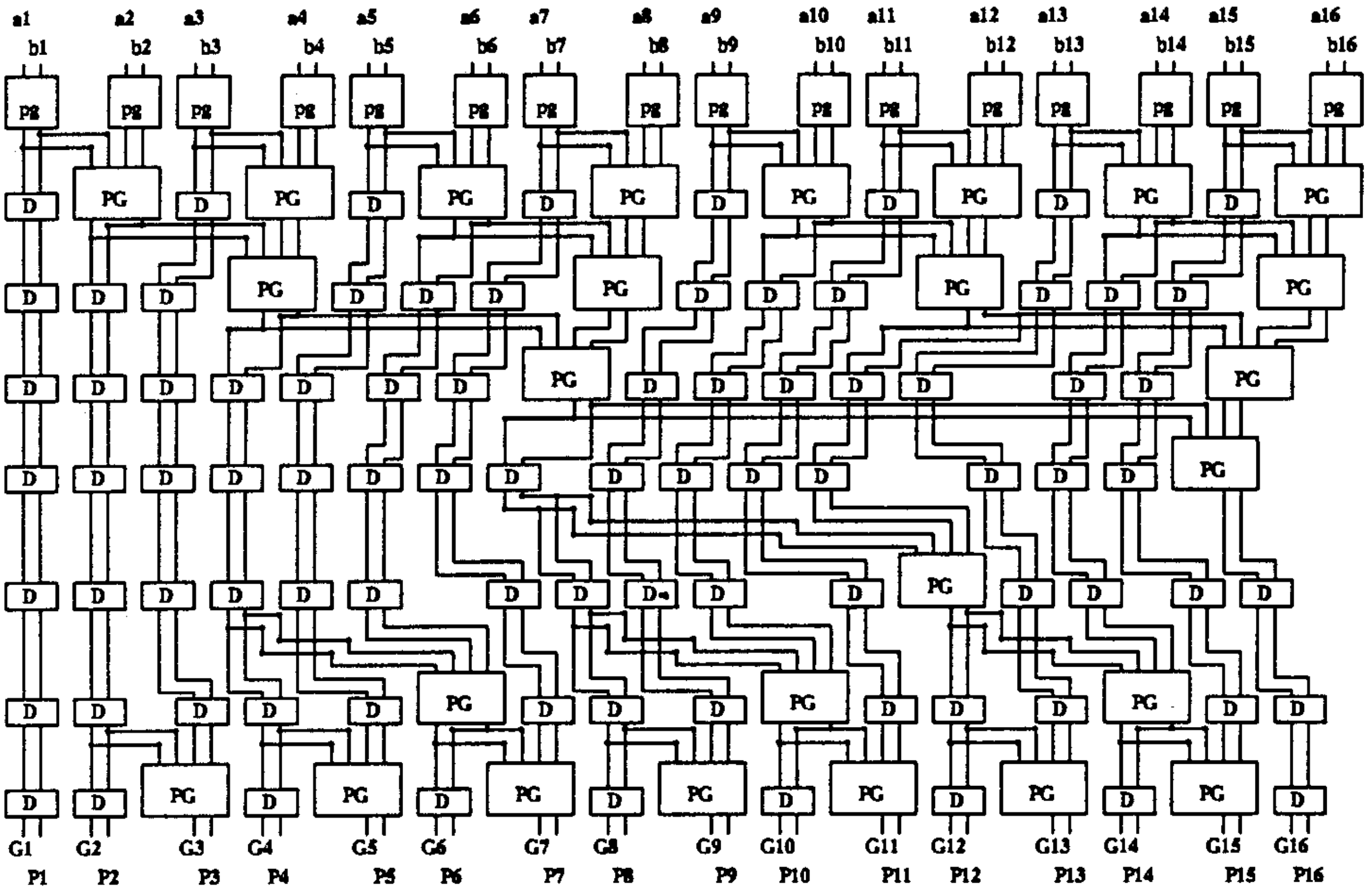

a1 b1 a2 b2 a3 b3 a4 b4 a5 b5 a6 b6 a7 b7 a8 b8 a9 b9 a10 b10 a11 b11 a12 b12 a13 b13 a14 b14 a15 b15 a16 b16
pg
PG
D
G1 P1 G2 P2 G3 P3 G4 P4 G5 P5 G6 P6 G7 P7 G8 P8 G9 P9 G10 P10 G11 P11 G12 P12 G13 P13 G14 P14 G15 P15 G16 P16

Figure 4.3.1 Computation of Carries for n = 16

This example raises the interesting problem of recognizing hanging wires. To obtain the final result, only the first output of each column G_i is required, thus in fact the second output should be considered a hanging wire. In Figure 4.3.1 this algorithm is sketched with each bit represented as a wire. However, the basic unit being passed among the primitves are pairs of bits. If each pair is represented by a wire as in Figure 4.3.2, then in effect there are no hanging wires. Clearly the notion of hanging wire is one which depends on the level of representation. To write the algorithm so that it has no hanging wires at a particular level, requires going into the details of this level, usually sacrificing regular patterns at higher levels and compact FP definitions.

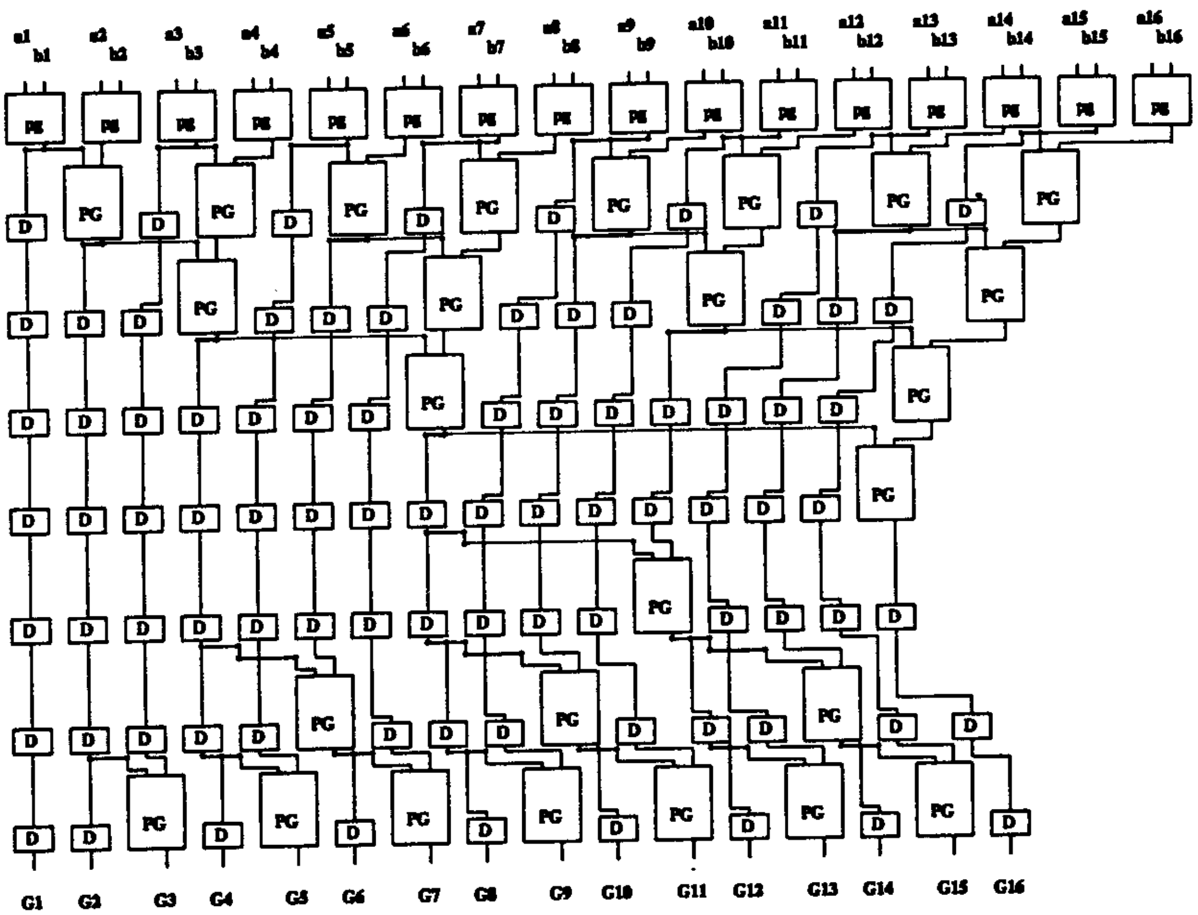

a1 b1 a2 b2 a3 b3 a4 b4 a5 b5 a6 b6 a7 b7 a8 b8 a9 b9 a10 b10 a11 b11 a12 b12 a13 b13 a14 b14 a15 b15 a16 b16
PE
PG
D
G1 G2 G3 G4 G5 G6 G7 G8 G9 G10 G11 G12 G13 G14 G15 G16

Figure 4.3.2 Higher level of representation for Figure 4.3.2.

An appealing way to deal with hanging wires, would be to have them recognized and removed automatically. This is a topic currently being examined. This is not as straight forward a task as it seems, since removing a hanging wire may create new ones, and make it possible to remove boxes or at least some of their inputs. It will be necessary to define exactly what is meant by a function "needing" an input before hanging wires can be removed automatically. For example, in the Brent-Kung adder, a hanging wire which is the output of a delay, **D**, can be removed as well as its corresponding input to the delay and the **and** gate of the last **PG** in each column can also be removed. Having the hanging wires removed automatically, might have simplified the definition of the Carry-Save Array multiplier presented in the last section.

4.4 Tally

The tally circuit counts the number of 1's in its input. The i^{th} 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 n^{th} input, adding a 0 to either side of the previous result, according to the value of this n^{th} input.

```
# main function
{tally (eql@[lngth,%1]->one;more)}
{one [id,notg]@1}
{more &SEL@distl@[[notg,id]@1,pair@cncat@[[%0],cncat@&[id,id]@tally@tl,[%0]]]}
{SEL wor@&SW@trans}
{wor org}
{SW andg}
```

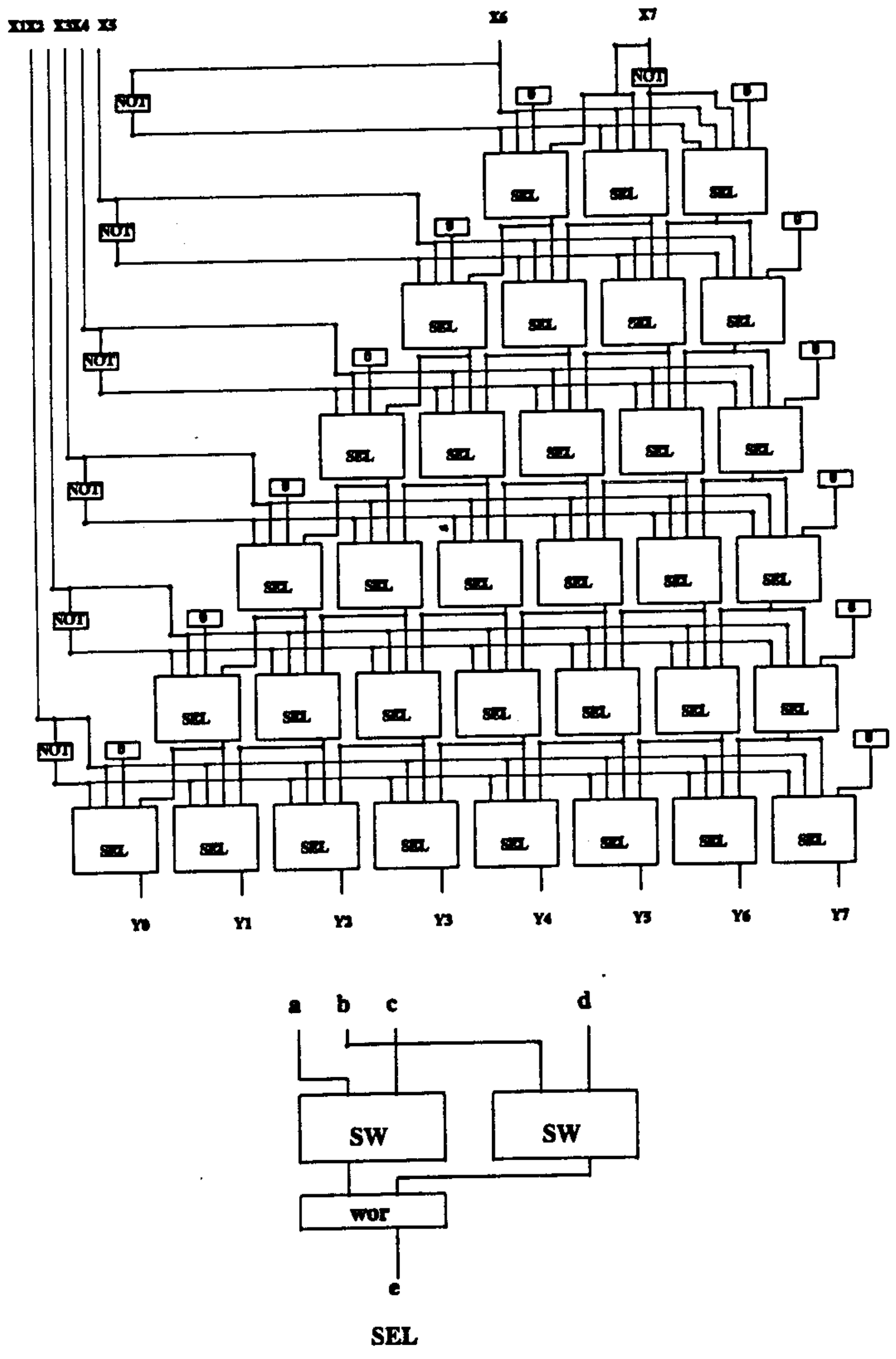
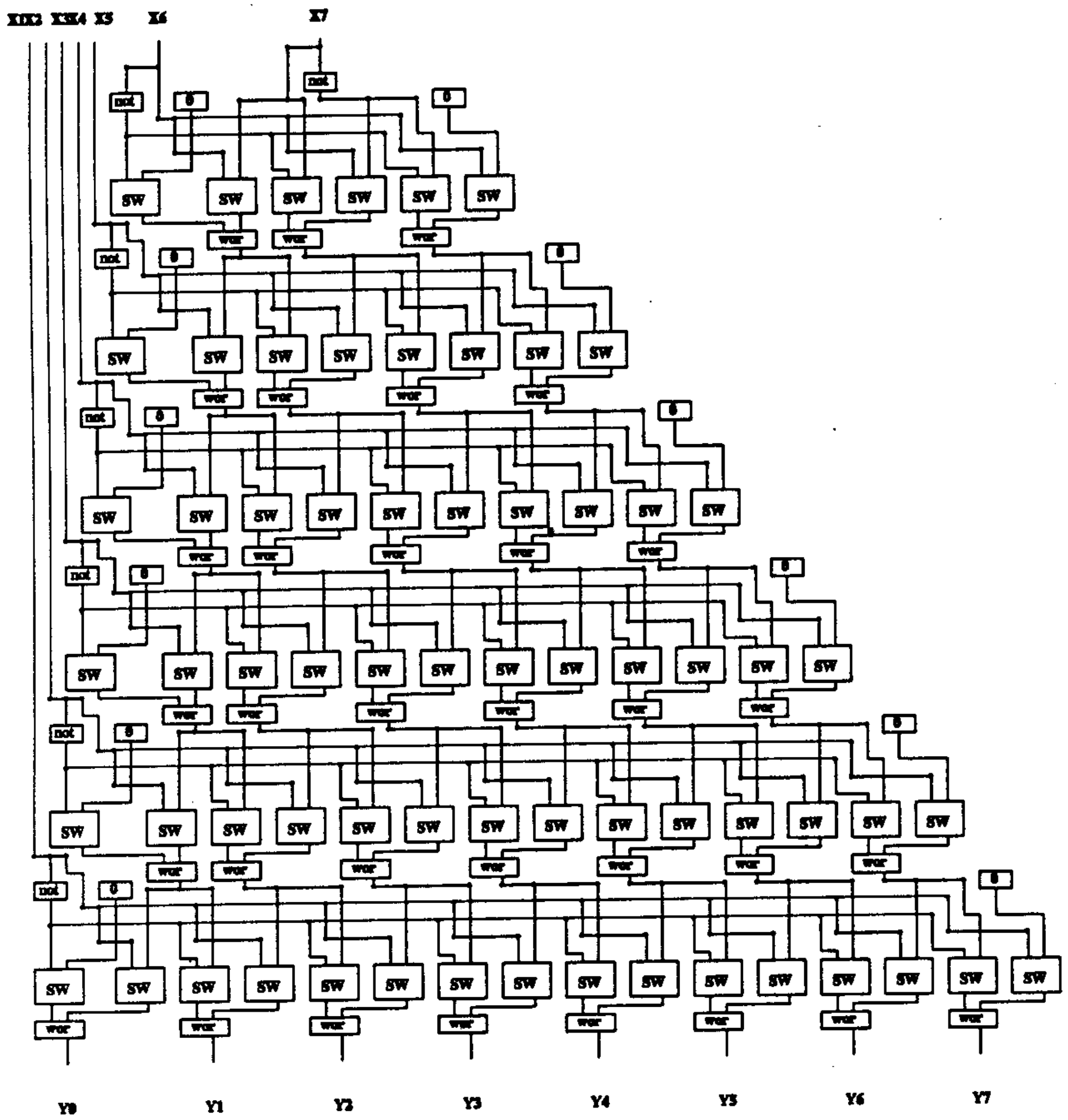


Figure 4.4.1 Tally circuit with SEL as a primitive.

As in the Decoder, a more compact placement can be obtained by compacting to the right instead of the left. However, this problem dissappears when the sketch of the algorithm is obtained with lower level primitives in Figure 4.4.2.



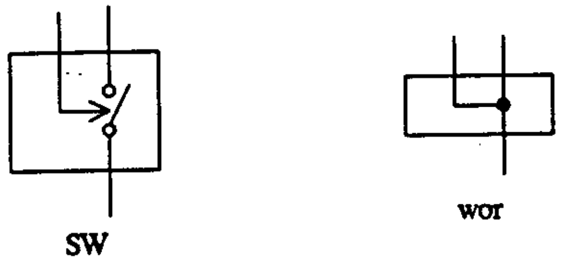


Figure 4.4.2 Tally circuit with **SW** and **wor** as primitives.

Again the primitives **SW** and **wor** can be represented by gates in the FP definition.

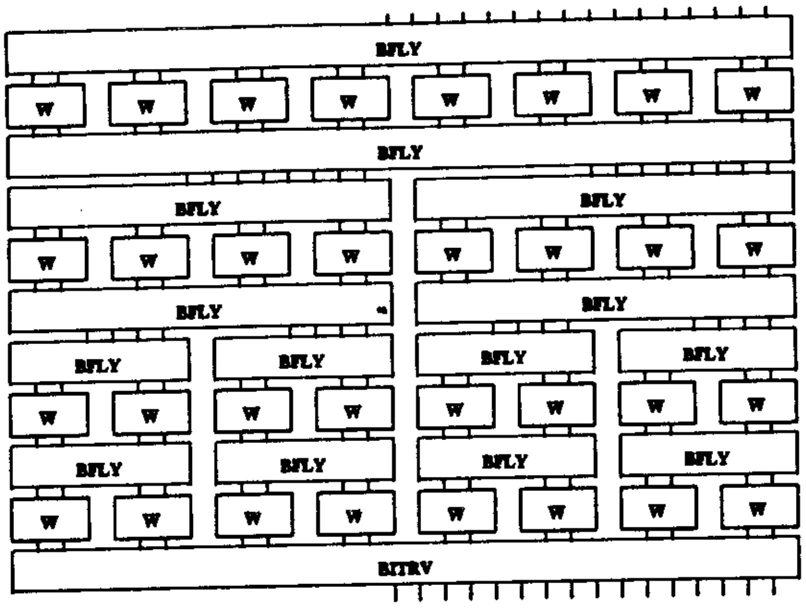
4.5 FFT

This last example is of a higher level algorithm; an algorithm to compute a 2^n point FFT. Two algorithms are presented, the one based on the Butterfly and Bit Reversal permutations and the other based on the shuffle permutation[Pa80]. Figure 4.5.1 shows the two algorithms with the permutations represented by boxes. The two functions are written for any 2^n point input.

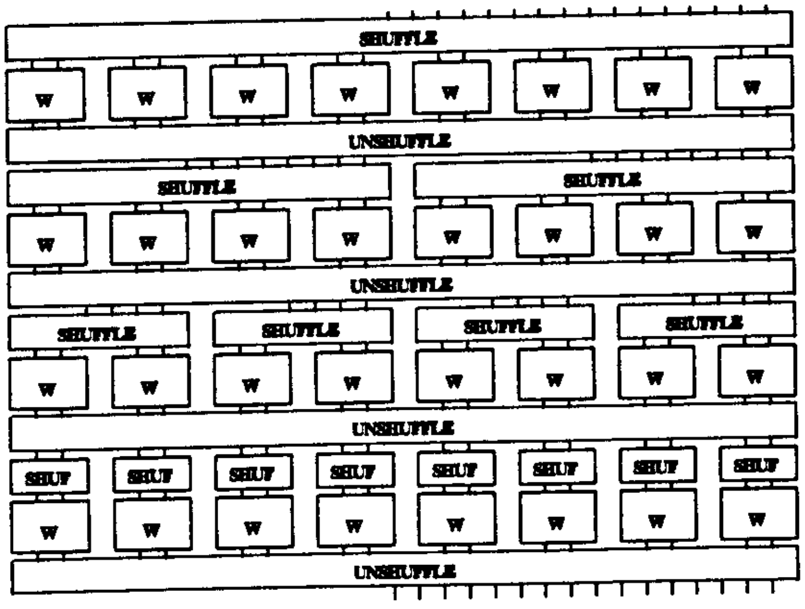
```
#; 2^n point fft
#
# input : (2^n complex numbers)
#(z0 z1 z2 z3 z4 z5 z6 z7 z8 z9 z10 z11 z12 z13 z14 z15)
# (or 2^n pairs of real numbers)
#((x0 y0) (x1 y1) (x2 y2) (x3 y3) (x4 y4) (x5 y5) (x6 y6) (x7 y7)
# (x8 y8) (x9 y9) (x10 y10) (x11 y11) (x12 y12) (x13 y13) (x14 y14) (x15 y15))
#
##################################################
# Traditional FFT Algorithm - Butterflies and Bit reversal
##################################################

{fft Bitrv@fftstages}

{fftstages (eql@[lngth,%2] -> W ; &fftstages@split@cncat@Bfly@cncat@&W@Bfly)}

{Bfly 1@[[cncat@[shuffle@[1,3],shuffle@[2,4]]@cncat@&trans@split@pair]]}

{Bitrv (eql@[lngth,%2]->id;&Bitrv@trans@pair)}

##################################################
# Shuffle - Unshuffle Algorithm
##################################################

{fft (end->flatten;fft@vunshuffle@stage)}
```

```
{vunshuffle (bottom -> unshuffle ; recons@vunshuffle@cncat)}

{unshuffle cncat@trans@pair}

{recons (bottom-> split; &recons)}

{stage (bottom-> split@cncat@&W@shuffle@split;&stage)}

{end (bottom->eql@[lngth,%1];end@1)}

# defined for complex numbers
{bottom atm@1}

# defined for real numbers
{bottom atm@1@1}

{flatten (bottom-> id; flatten@cncat)}

##################################################
# Definition of W
##################################################

{W [cadd,csub]@[1,cmul@[2,u0]]}

{u0 [u,u]}

{u %1}

{cadd &add@shuffle}

{csub &sub@shuffle}

{cmul [sub@[mult@[1@1,1@2],mult@[2@1,2@2]],
      add@[mult@[1@1,2@2],mult@[2@1,1@2]]]}
```

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 Figure 4.5.1. Each wire represents a complex number and the W is marked as a primitive as well as the permutations.



a) FFT with Butterfly and Bit-Reversal Permutations



b) FFT with Shuffle Permutation

Figure 4.5.1 Two algorithms for a 16-point FFT.

In Figure 4.5.2 each wire is also a complex number, but the permutations, **Bfly** and **Bitrv**, are no longer primitives.

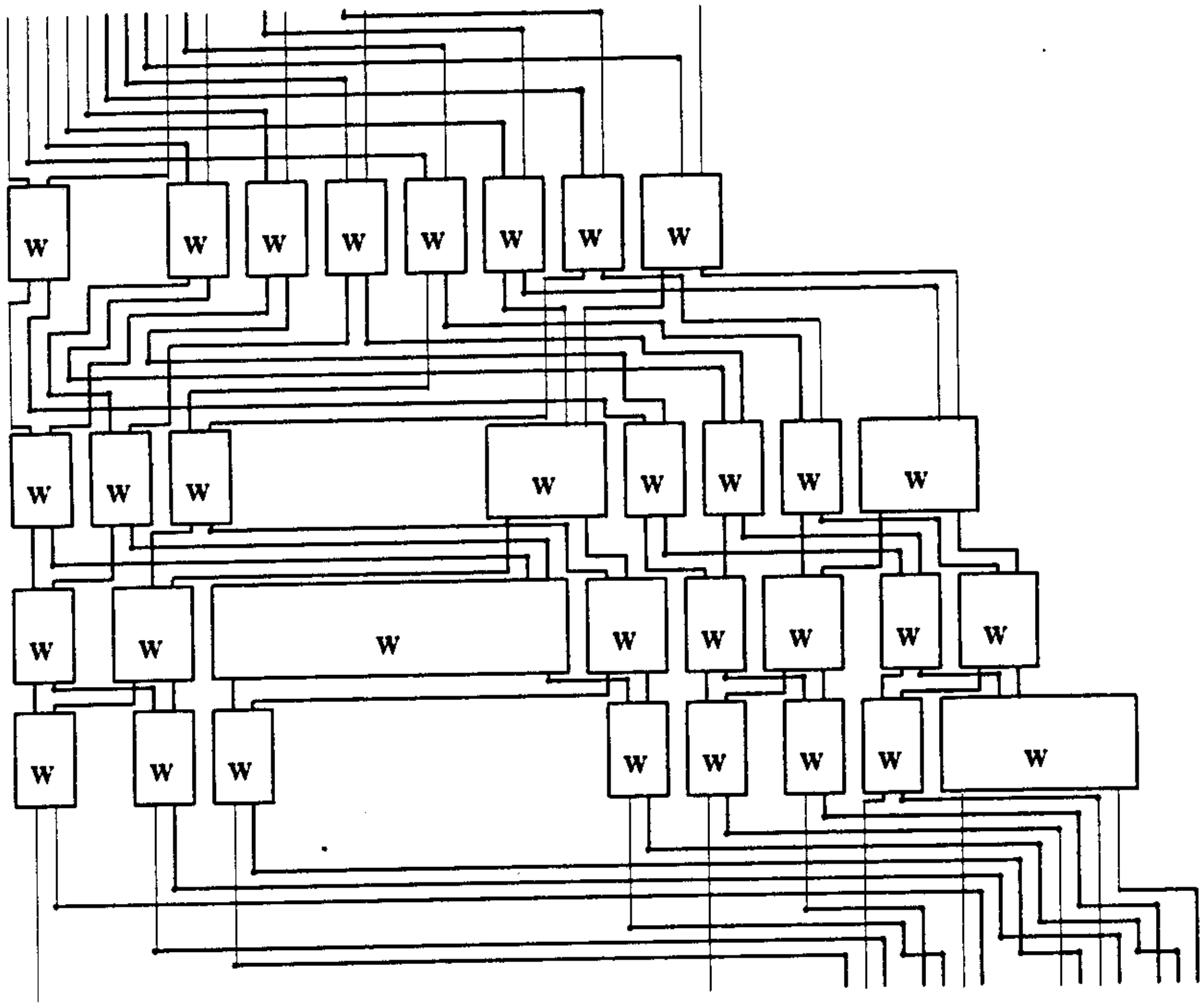
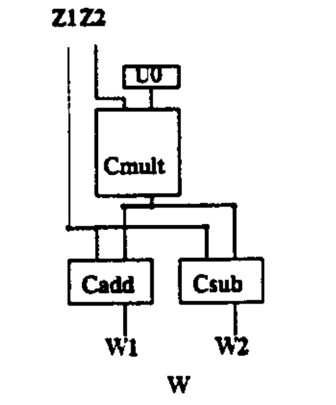
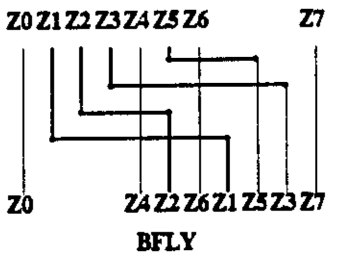
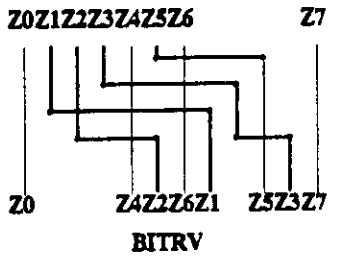


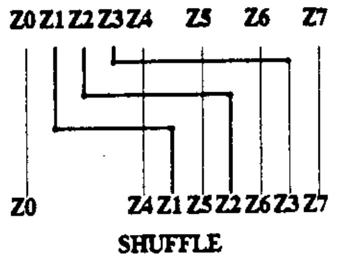
Figure 4.5.2 FFT algorithm of Figure 4.5.1a with only **W** as a primitive.

Figure 4.5.3 contains the primitives for the sketches in Figure 4.5.1. The primitive **W** is represented with complex numbers as atoms.









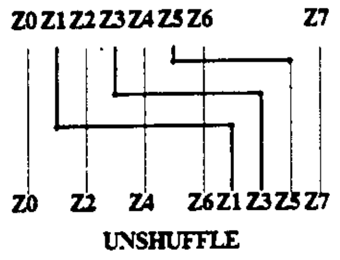


Figure 4.5.3 The primitives **W**, **BFLY**, **BITRV**, **SHUFFLE** and **UNSHUFFLE**.

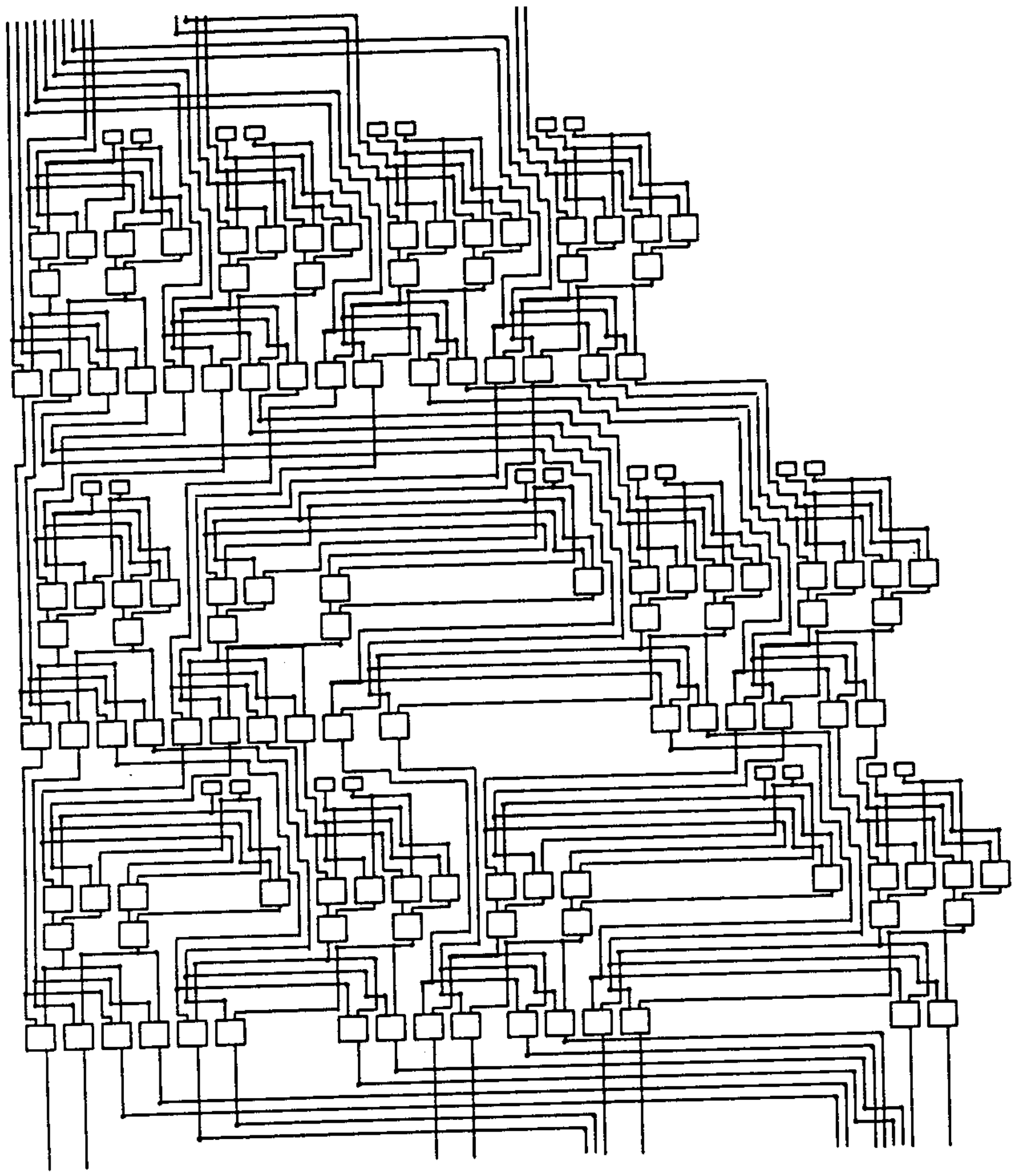


Figure 4.5.4 8-point Butterfly-FFT algorithm with the primitive W expanded.

In Figure 4.5.4 each wire now represents a real number but the 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 primtives.

The performance of the compaction algorithm is most visible in this example. The constraint graphs of Figures 4.5.1a, 4.5.1b and 4.5.2 were inconsistent and required adjustment. For Figures 4.5.1a and 4.5.1b, 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 performed, (on a constraint graph which was now consistent). The sketch for Figure 4.5.2 is not as "nice"; the routing of the permutations forces this inefficient placement. This problem is somewhat alleviated in Figure 4.5.4 where lower level primitives are represented.

5. Conclusion

The following sections contain impressions of using FP to specify circuits. Disadvantages as well as advantages are discussed and a few directions for improvements and enhancements are given.

5.0 Advantages

Specifying circuits with FP offers the following advantages.

1. Functionality

   It is appealing to specify circuits by function rather than structure and obtain the structure as a result of the algorithm. Pre-designed components can be stored in a library by their FP code, providing flexibility to adapt to different environments and technologies. Generic definitions can be provided making it unnecessary to check size parameters; the designer can retrieve the function "shift" without specifying the input size.

2. Control over placement and routing

   Although a circuit is specified by function, the designer has a great deal of control over the spatial organization of the circuit. Specifying with FP also provides the designer with the possibility of applying "place and route" tools on only selected portions and hierarchically. Placement and routing are not considered as separate operations; wires are instantiated and positioned as elements in the same manner as boxes.

3. Turn-around time

   The time from specification to sketch is suitable for an interactive design environment. This is important since the goal is to estimate space and time characteristics before committing to a particular algorithm. Visual feedback is an important feature for a system of design tools.

5.1 Current Limitations

The implementation described in section 3 has the following limitations.

1. Combinational circuits

   For FP to be truly useful, the problem of describing sequential circuits must be addressed. From the geometric point of view, the IF is still suitable for describing these circuits, since the direction a signal flows on a wire is irrelevant to the compactor.

2. Uni-directional flow of information

   This is the most serious limitation. The flow of data is always seen as moving in one direction; inputs are always on one side and the outputs on the other. This is to some extent the result of human perception of computation. More sophisticated functional forms will be required to combine functions in more complex ways. Systolic arrays, for example, will require special handling.

3. One-dimensional compaction

   The current mapping from the IF to a sketch utilizes the vertical coordinates of the IF. A more two-dimensional approach might yield better results. This is an enhancement which is being considered.

5.2 Future Directions

The following improvements and enhancements are being considered.

1. Detection and Removal of Hanging Wires

   As discussed in Section 4.0, the automatic removal of hanging wires would facilitate the writing of FP specifications. This addition is not expected to pose any difficulty.

2. More sophisticated mapping of IF

   A two-dimensional approach in mapping the IF to fixed geometry would improve the quality of the sketches and also remove some of the burden associated with specifying circuits using FP. The cost of this improvement may be in the time required to map from the IF to fixed geometry and in less control over placement.

3. Prediction of Placement based on routing requirements

   From the IF it may be possible to directly detect the conflicts in routing which result in the enlargement of boxes and gaps in the placement. These enlargements and gaps would be removed by adding more horizontal tracks in strategic locations. The FFT in section 4.5 is an example of a specification whose sketch might benefit from this type of transformation.

4. Algorithmic transformations

   One of the tasks in which algebraic transformations could be used is in the "untangling" of the placement. Untangling a sketch by transforming its FP specification is possible since the amount of "tangling" is measurable directly from the IF, without obtaining fixed geometry. The degree of "tangling" of a placement would be measured from the amount of crossing wires. By flipping entire functions, at various levels in the hierarchy, the number of crossings would be reduced. This problem is NP-hard in the general case. By operating on an FP specification instead of a collection of boxes and wires, those transformations which would improve the placement might be easier to identify. Of course, the optimal placement may be difficult to obtain or recognize, and might not even be reachable, using only algebraic transformations.

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 [Fod80]), 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" [Ba78].

Application

This is the single FP operation and is designated by the colon (":"). For a function σ and an object x, $\sigma{:}x$ is an application and its meaning is the object that results from applying σ to x (*i.e.*, evaluating $\sigma(x)$). We say that σ is the *operator* and that x is the *operand*. The following are examples of applications:

+:<7,8>	$\equiv$	15	**tl**:<1,2,3>	$\equiv$	<2,3>
1 :$<a,b,c,d>$	$\equiv$	a	**2** :$<a,b,c,d>$	$\equiv$	b

Functions

All functions (F) map objects into objects, moreover, they are *strict*:

$$\sigma{:}\perp \equiv \perp, \ \forall \sigma \in \mathbf{F}$$

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expressions [Mc60]:

$p_1 \to e_1 ; \cdots ; p_n \to e_n ; e_{n+1}$

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,

μ : $x \equiv$

$x=<x_1, x_2, \ldots, x_k> \wedge 0 < \mu \leq k \to x_\mu$;

$x=<x_1, x_2, \ldots, x_k> \wedge -k \leq \mu < 0 \to x_{k+\mu+1}$; $\perp$

pick : $<n,x> \equiv$

$x=<x_1, x_2, \ldots, x_k> \wedge 0 < n \leq k \to x_n$;

$x=<x_1, x_2, \ldots, x_k> \wedge -k \leq n < 0 \to x_{k+n+1}$; $\perp$

The user should note that the function symbols **1,2,3,...** are to be distinguished from the atoms 1,2,3,....

last : $x \equiv$

$x=<> \to <>$;

$x=<x_1, x_2, \ldots, x_k> \wedge k \geq 1 \to x_k$; $\perp$

first : $x \equiv$

$x=<> \to <>$;

$x=<x_1, x_2, \ldots, x_k> \wedge k \geq 1 \to x_1$; $\perp$

Tail Functions

tl : $x \equiv$

$x=<x_1> \to <>$;

$x=<x_1, x_2, \ldots, x_k> \wedge k \geq 2 \to <x_2 , \ldots , x_k>$; $\perp$

tlr : $x \equiv$

$x=<x_1> \to <>$;

$x=<x_1, x_2, \ldots, x_k> \wedge k \geq 2 \to <x_1 , \ldots , x_{k-1}>$; $\perp$

Note: There is also a function **front** that is equivalent to **tlr**.

Distribute from left and right

distl : $x \equiv$

$x = <y, <>> \rightarrow <>;$

$x = <y, <z_1, z_2, \ldots, z_k>> \rightarrow <<y, z_1>, \ldots, <y, z_k>>; \perp$

distr : $x \equiv$

$x = <<>, y> \rightarrow <>;$

$x = <<y_1, y_2, \ldots, y_k>, z> \rightarrow <<y_1, z>, \ldots, <y_k, z>>; \perp$

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 : $x \equiv$

$x = <y, <>> \rightarrow <y>;$

$x = <y, <z_1, z_2, \ldots, z_k>> \rightarrow <y, z_1, z_2, \ldots, z_k>; \perp$

apndr : $x \equiv$

$x = <<>, z> \rightarrow <z>;$

$x = <<y_1, y_2, \ldots, y_k>, z> \rightarrow <y_1, y_2, \ldots, y_k, z>; \perp$

Transpose

trans : $x \equiv$

$x = <<>, \ldots, <>> \rightarrow <>;$

$x = <x_1, x_2, \ldots, x_k> \rightarrow <y_1, \ldots, y_m>; \perp$

where $x_i = <x_{i1}, \ldots, x_{im}> \wedge y_j = <x_{1j}, \ldots, x_{kj}>$,

$1 \leq i \leq k$, $1 \leq j \leq m$.

reverse : $x \equiv$

$x = <> \rightarrow ; <>$

$x = <x_1, x_2, \ldots, x_k> \rightarrow <x_k, \ldots, x_1>; \perp$

Rotate Left and Right

rotl : x ≡

$x=<> \rightarrow <>; x=<x_1> \rightarrow <x_1>;$

$x=<x_1, x_2, \ldots, x_k> \wedge k\geq 2 \rightarrow <x_2, \ldots, x_k, x_1>; \perp$

rotr : x ≡

$x=<> \rightarrow <>; x=<x_1> \rightarrow <x_1>;$

$x=<x_1, x_2, \ldots, x_k> \wedge k\geq 2 \rightarrow <x_k, x_1, \ldots, x_{k-2}, x_{k-1}>; \perp$

concat : x ≡

$x=<<x_{11}, \ldots, x_{1k}>, <x_{21}, \ldots, x_{2n}>, \ldots, <x_{m1}, \ldots, x_{mp}>> \wedge k, m, n, p > 0 \rightarrow$
$<x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mp}>; \perp$

Concatenate removes all occurrences of the null sequence:

concat : <<1,3>,<>,<2,4>,<>,<5>> ≡ <1,3,2,4,5>

pair : x ≡

$x=<x_1, x_2, \ldots, x_k> \wedge k>0 \wedge k \textit{ is even} \rightarrow <<x_1,x_2>, \ldots, <x_{k-1},x_k>>;$

$x=<x_1, x_2, \ldots, x_k> \wedge k>0 \wedge k \textit{ is odd} \rightarrow <<x_1,x_2>, \ldots, <x_k>>; \perp$

split : x ≡

$x=<x_1> \rightarrow <<x_1>,<>>;$

$x=<x_1, x_2, \ldots, x_k> \wedge k>1 \rightarrow <<x_1, \ldots, x_{\lceil k/2 \rceil}>, <x_{\lceil k/2 \rceil+1}, \ldots, x_k>>; \perp$

iota : x ≡

$x=0 \rightarrow <>;$

$x \in \mathbf{N}^+ \rightarrow <1,2,\ldots,x>; \perp$

Predicate (Test) Functions

atom : x ≡ $x \in atoms \rightarrow$ **T**; $x \neq \perp \rightarrow$ **F**; $\perp$

eq : x ≡ $x=<y,z> \wedge y=z \rightarrow$ **T**; $x=<y,z> \wedge y \neq z \rightarrow$ **F**; $\perp$

Also less than (<), greater than (>), greater than or equal (>=), less than or equal (<=), not equal (˜=); '=' is a synonym for **eq.**

null : x ≡ $x=<> \rightarrow$ **T**; $x \neq \perp \rightarrow$ **F**; $\perp$

length : $x \equiv x = <x_1, x_2, \ldots, x_k> \rightarrow k; x=<> \rightarrow 0; \perp$

Arithmetic/Logical

$+ : x \equiv x=<y,z> \wedge$ *y,z are numbers* $\rightarrow y+z; \perp$

$- : x \equiv x=<y,z> \wedge$ *y,z are numbers* $\rightarrow y-z; \perp$

$* : x \equiv x=<y,z> \wedge$ *y,z are numbers* $\rightarrow y \times z; \perp$

$/ : x \equiv x=<y,z> \wedge$ *y,z are numbers* $\wedge z \neq 0 \rightarrow y \div z; \perp$

And, or, not, xor

and : $<x,y> \equiv x=\mathbf{T} \rightarrow y; x=\mathbf{F} \rightarrow \mathbf{F}; \perp$

or : $<x,y> \equiv x=\mathbf{F} \rightarrow y; x=\mathbf{T} \rightarrow \mathbf{T}; \perp$

xor : $<x,y> \equiv$

$x=\mathbf{T} \wedge y=\mathbf{T} \rightarrow \mathbf{F}; x=\mathbf{F} \wedge y=\mathbf{F} \rightarrow \mathbf{F};$

$x=\mathbf{T} \wedge y=\mathbf{F} \rightarrow \mathbf{T}; x=\mathbf{F} \wedge y=\mathbf{T} \rightarrow \mathbf{T}; \perp$

not : $x \equiv x=\mathbf{T} \rightarrow F; x=\mathbf{F} \rightarrow \mathbf{T}; \perp$

Library Routines

sin : $x \equiv x$ is a number $\rightarrow sin(x); \perp$

asin : $x \equiv x$ is a number $\wedge |x| \leq 1 \rightarrow \sin^{-1}(x); \perp$

cos : $x \equiv x$ is a number $\rightarrow cos(x); \perp$

acos : $x \equiv x$ is a number $\wedge |x| \leq 1 \rightarrow \cos^{-1}(x); \perp$

exp : $x \equiv x$ is a number $\rightarrow e^{x}; \perp$

log : $x \equiv x$ is a positive number $\rightarrow ln(x); \perp$

mod : $<x,y> \equiv x$ **and** y are numbers $\rightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor ; \perp$

Functional Forms

Functional 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; functional forms manipulate functions, while traditional operators manipulate values.

One functional form is *composition*. For two functions ϕ and ψ the form ϕ @ ψ denotes their composition $\phi \circ \psi$:

$$(\phi \ @ \ \psi) : x \equiv \phi:(\psi:x), \ \ \forall \ \ x \in \Omega$$

The *constant* function takes an object parameter:

$\%x{:}y \equiv y{=}\bot \rightarrow \bot; x, \quad \forall \; x,y \in \Omega$

The function %⊥ always returns "⊥".

In the following description of the functional forms, we assume that ξ, ξ_i, σ, σ_i, τ, and τ_i are functions and that x, x_i, y are objects.

Composition

$(\sigma \; @ \; \tau){:}x \equiv \sigma{:}(\tau{:}x)$

Construction

$[\sigma_1, \ldots, \sigma_n]{:}x \equiv <\sigma_1{:}x, \ldots, \sigma_n{:}x>$

Note that construction is also bottom-preserving, *e.g.*,

$[+,/]{:}<3,0> \; = \; <3,\bot> \; = \; \bot$

Condition

$(\xi \; \text{->} \; \sigma; \tau){:}x \equiv$
$\quad (\xi{:}x){=}\mathbf{T} \rightarrow \sigma{:}x;$
$\quad (\xi{:}x){=}\mathbf{F} \rightarrow \tau{:}x; \bot$

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy's conditional expression, and the *functional 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

$\%x{:}y \equiv y{=}\bot \rightarrow \bot; x, \qquad \forall \; x \in \Omega$

This function returns its object parameter as its result.

Right Insert

$!\sigma \; {:}x \equiv$
$\quad x{=}<> \rightarrow e_f{:}x;$
$\quad x{=}<x_1> \rightarrow x_1;$
$\quad x{=}<x_1, x_2, \ldots, x_k> \wedge k{>}2 \rightarrow \sigma{:}<x_1, !\sigma{:}<x_2, \ldots, x_k>>; \bot$

e.g., $!+{:}<4,5,6>=15.$

If σ has a right identity element e_f, then $!\sigma{:}\langle\rangle = e_f$, *e.g.*,

$!+{:}\langle\rangle=0$ and $!*{:}\langle\rangle=1$

Currently, identity functions are defined for + (0), − (0), * (1), / (1), also for **and** (T), **or** (F), **xor** (F). All other unit functions default to bottom ("⊥").

Tree Insert

$|\sigma : x \equiv$

$x=\langle\rangle \rightarrow e_f{:}x;$

$x=\langle x_1\rangle \rightarrow x_1;$

$x=\langle x_1, x_2, \ldots, x_k\rangle \wedge k>1 \rightarrow$

$\sigma : \langle |\sigma : \langle x_1, \ldots, x_{\lceil k/2\rceil}\rangle , |\sigma : \langle x_{\lceil k/2\rceil+1}, \ldots, x_k\rangle\rangle;\perp$

e.g.,

$|+{:}\langle 4,5,6,7\rangle \equiv +{:}\langle +{:}\langle 4,5\rangle,+{:}\langle 6,7\rangle\rangle \equiv 15$

Tree insert uses the same identity functions as right insert.

Apply to All

$\&\sigma{:}\, x \equiv$

$x=\langle\rangle \rightarrow \langle\rangle;$

$x=\langle x_1, x_2, \ldots, x_k\rangle \rightarrow \langle \sigma{:}x_1, \ldots, \sigma{:}x_k\rangle; \perp$

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 functional form, including a single primitive or defined function. For example the function

{factorial ! @ iota}*

is the non-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 functional form: if $f \equiv 1@2$ then $f : x \equiv 1@2 : x, \quad \forall\, x\in\Omega$.

References to undefined functions bottom out:

$f{:}x \equiv \perp \forall x\in\Omega,\, f\notin \mathbf{F}$

The following table gives the correspondence between the primitives and forms of Berkeley FP and the ones implemented in the system described in Section 3.

Primitives			
Berkeley FP	Lisp or T FP	Berkeley FP	Lisp or T FP
$\mu \in \mathbf{Z}$	μ	pick	pick
last	last	first	first
tl	tl	tlr	tlr
distl	distl	distr	distr
id	id	out	out
apndl	apndl	apndr	apndr
trans	trans	reverse	rvers
rotl	rotl	rotr	rotr
concat	cncat	pair	pair
split	split	iota	iota
atom	atm	eq or =	eql
null	nul	length	lngth
<	lsth	<=	lseq
>	grth	>=	greq
~=	neql	+	fadd
-	fsub	*	fmult
/	fdiv	and	andp
or	orp	xor	xorp
not	notp	sin	sin
cos	cos	asin	asin
acos	acos	log	log
mod	mod	exp	fexp

Forms	
Berkeley FP	Lisp or T FP
$f_1@f_2@\ldots@f_n$	(comp $f_1\ f_2\ \ldots\ f_n$)
$[f_1,f_2,\ldots,f_n]$	(build $f_1\ f_2\ \ldots\ f_n$)
(p->f;g)	(condl p f g)
!f	(rins f)
&f	(apall f)
|f	(tins f)
%Obj	(kons Obj)

In addition the Lisp and T FP interpreters contain the following 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 : $<x,y> \equiv$

$x=1 \wedge y=1 \rightarrow 0; x=0 \wedge y=0 \rightarrow 0;$

$x=1 \wedge y=0 \rightarrow 1; x=0 \wedge y=1 \rightarrow 1; \perp$

nandg : $<x,y> \equiv$

$x=1 \wedge y=1 \rightarrow 0; x=0 \wedge y=0 \rightarrow 1;$

$x=1 \wedge y=0 \rightarrow 1; x=0 \wedge y=1 \rightarrow 1; \perp$

norg : $<x,y> \equiv$

$x=1 \wedge y=1 \rightarrow 0; x=0 \wedge y=0 \rightarrow 1;$

$x=1 \wedge y=0 \rightarrow 0; x=0 \wedge y=1 \rightarrow 0; \perp$

notg : $x \equiv x=\mathbf{T} \rightarrow F; x=\mathbf{F} \rightarrow \mathbf{T}; \perp$

sqrt : $x \equiv x$ is a number $\rightarrow \sqrt{x}; \perp$

shuffle : $x \equiv$

$x=<<>,<>> \rightarrow <>;$

$x=<<z_1, \ldots, z_m>,<y_1, \ldots, y_m>> \rightarrow <<z_1,y_1>, \ldots, <z_m,y_m>>; \perp$

References

[Ba78] Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM, Turing Award Lecture,* Vol. 21, No. 8, August 1978, pp. 613-641.

[Ba83] Baden, S., "Berkeley FP User's Manual, Rev. 4.1," University of California, Berkeley, Berkeley, California, March 1983.

[BrKu80] Brent, R. P. and H. T. Kung, "The Chip Complexity of Binary Arithmetic," in *Proceedings 12th ACM Symposium on the Theory of Computing,* May 1980, pp. 190-200.

[Fo80] Foderaro, J. K., "The Franz Lisp Manual," University of California, Berkeley, Berkeley, California, 1980.

[La81] Lahti, D. O., "Applications of a Functional Programming Language," UCLA, Los Angeles, California, Tech. Rep. CSD-810403, April 1981.

[LiWo83] 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,* Vol. CAD-2, No. 2, April 1983, pp. 62-69.

[Mc60] McCarthy, J., "Recursive Functions of Symbolic expressions and their Computation by Machine," *Communications of the ACM,* Vol. 3, No. 4, April 1960, pp. 184-195.

[MeCo80] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Reading, Massachusetts: Addison-Wesley, 1980.

[Mesh84] Meshkinpour, F., "On Specification and Design of Digital Systems Using an Applicative Hardware Description Language," UCLA, Los Angeles, California, Tech. Rep. MS Thesis, 1984.

[Ou81] Ousterhout, J. K., "Caesar: An Interactive Editor for VLSI Layouts," *VLSI Design,* Vol. II, No. 4, fourth quarter 1981, pp. 34-38.

[OHM83] Ousterhout, J. K, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "A Collection of Papers on Magic," University of California, Berkeley, Berkeley, California, Tech. Rep. UCB/CSD 83/154, December 1983.

[Pa80] Parker, D. S., "Note on Shuffle/Exchange-Type Switching Networks," *IEEE Transactions on Computers,* Vol. C-29, No. 3, March 1980, pp. 213-222.

[Riv82] Rivest, R. L., "The 'PI' (Place and Interconnect) System," in *Proceedings 19th Design Automation Conference*, Las Vegas, Nevada: June 1982, pp. 475-481.

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

[West81] Weste, N., "Virtual Grid Symbolic Layout," in *Proceedings 18th Design Automation Conference*, 1981, pp. 225-233.

[Will78] Williams, J., "STICKS- A Graphical Compiler for High Level LSI Design," in *Proceedings 1978 National Computer Conference*, May 1978, pp. 289-295.

[Will82] Williams, J., "Notes on the FP Style of Functional Programming," in *Functional Programming and Its Applications*, Darlington Henderson Turner, Ed. Cambridge, England: Cambridge University Press, 1982.