Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

FLAG: AN FP BASED VLSI LAYOUT GENERATOR

Winthrop John Wu

May 1989
CSD-890023

FLAG: An FP Based VLSI Layout Generator

Winthrop John Wu

May 1, 1989

Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Miloš Ercegovac for all his guidance and patience. Next, I would like to thank my committee members, Tomas Lang and Yuval Tamir for their comments during the writing of this work. I would also like to thank my parents for their continual encouragement and support during this work. In addition, I would like to thank Dorab Patel and Martine Schlag, whose work formed the basis for much of this work. Finally, I would like to thank all my friends for their help and support during the writing of this work.

Abstract

Rapid prototyping and architectural exploration are not possible in typical design systems due to either the amount of designer effort required or the lack of designer control over the final design. An FP based VLSI Layout Generator (FLAG) is presented whose main objective is to provide rapid prototyping and architectural exploration as part of the design approach. FLAG takes as input a behavioral description of the circuit, written in an applicative language (FP) and generates a VLSI layout from it. A number of circuits are created, ranging in complexity from an exclusive-OR to a carry-save multiplier. For each circuit, a VLSI layout is created by hand and by FLAG. The FLAG and hand layouts are compared against each other on the basis of overall layout area, circuit delay and circuit design time.

Chapter 1

Introduction

1.1 Problem

Typical VLSI design systems take some form of schematic input and convert it into a physical layout. The schematic input can take the form of a functional description, block diagram, circuit diagram, behavioral description, or digital algorithm. The conversion of the schematic input to physical implementation can either be accomplished by hand or the process can be automated.

If the schematic conversion is performed by hand, the designer is responsible for all aspects of the translation of the schematic input into a physical layout. This approach possesses the following drawbacks:

- It is slow and tedious: All aspects of the conversion must be performed by hand.
- It is error-prone: Human error is an unavoidable part of the process.
- It requires VLSI expertise: Since the designer is performing the physical layout, he must be aware of the various VLSI design issues that may affect the design.
- The scaling and combining of different circuits cannot be done easily.

Because of the drawbacks of this approach, rapid prototyping is not possible and architectural exploration is too costly in terms of time and effort to include as part of the design process.

Another method of schematic conversion is to automate the entire process and minimize the designer's participation in the design process through the use of automatic layout tools and silicon compilers. Automatic layout tools take a description of the circuit in terms of modules and net-lists and generates masks while silicon compilers generate circuit elements and interconnections from a behavioral description. Examples of automatic layout tools and silicon compilers are GAELIC, DELILA, Genesil, MacPitts, CAPRI, SCHOLAR and ARSENIC. This approach has the advantage that rapid prototyping is now feasible. All that is required from the designer is a schematic description of the intended circuit. A drawback of this approach is that the designer often has little or no control over the structure of the physical layout and there is no way to guarantee that the final design will correspond to the designer's intentions. This becomes an obstacle to architectural exploration since the designer has lost the ability to specify the structure of the final circuit.

The goal of this thesis is to present a VLSI design approach whose objective is to provide rapid prototyping and architectural exploration as part of the overall design cycle. The approach is based upon FLAG, (FP based VLSI Layout Generator), which takes an algorithm written in a functional programming language, FP, as input and automates the generation of a physical layout while still allowing the designer to retain control over the design of the resulting physical layout. This work is based upon the theoretical work of [Schl84,Schl86] and upon her implementation of the symbolic interpreter. Related work has been done by [Laht81,Mesh84,Pate85,Worl86,Shee84].

The circuits generated by FLAG are meant to be a first approximation of the hand-generated ones and are not meant to replace them. We restrict our attention to combinational circuits and systems; sequential circuits and systems will not be considered here.

1.2 Approach

The goal of FLAG is to allow architectural exploration to become part of the typical design cycle. The "philosophy" of the FLAG approach can be stated as follows.

- Automating the tedious aspects of the design cycle with the intention of minimizing the designer intervention.
- Providing the designer with control over the overall structure of the final design.
- Promoting a hierarchical design approach.
- Providing quick feedback about a particular design at various levels of abstraction.
- Providing an interface to lower-level VLSI CAD tools.

An abstract view of the FLAG design cycle is shown in Figure 1.1. The design process can be broken down into three steps:

1. System Behavior and Structural Description
2. Symbolic Interpretation & Topological Extraction
3. Physical Layout and Circuit Simulation

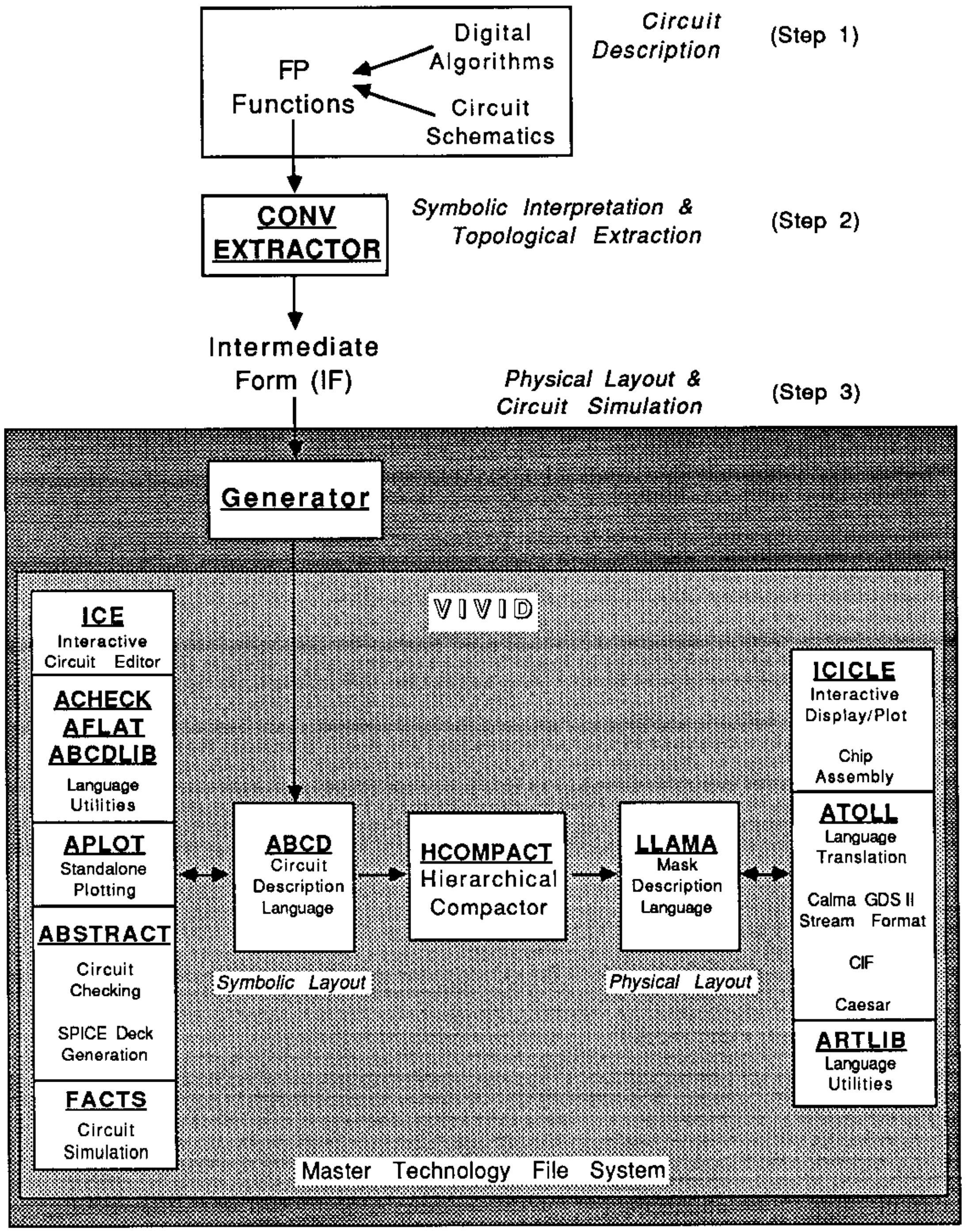
1.3 Circuit Description

In Step 1, a description of the target circuit's behavior and structure is generated in an applicative language, FP[1], that has been modified to facilitate its us as a hardware description language. The description can be created from an algorithm or circuit diagram and is written as a series of FP functions. The manner in which a circuit's structure is described will be discussed in the next section. The advantages gained from describing the circuit with an applicative hardware description language are:[2]

- Describes the structure and behavior of a circuit at a high-level.

[1]J. Backus, "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), 613-641.

[2]Dorab Patel, Martine Schlag, and Miloš Ercegovac, "νFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms," **Functional Programming Languages and Computer Architecture,** J.P. Jouannaud, Ed. Nancy, France: Springer-Verlag Lecture Notes in Computer Science, September 1985, pp. 238-255.



Digital Algorithms
FP Functions
Circuit Schematics
Circuit Description
(Step 1)
CONV EXTRACTOR
Symbolic Interpretation & Topological Extraction
(Step 2)
Intermediate Form (IF)
Physical Layout & Circuit Simulation
(Step 3)
Generator
VIVID
ICE Interactive Circuit Editor
ACHECK AFLAT ABCDLIB Language Utilities
APLOT Standalone Plotting
ABSTRACT Circuit Checking SPICE Deck Generation
FACTS Circuit Simulation
ABCD Circuit Description Language
HCOMPACT Hierarchical Compactor
LLAMA Mask Description Language
Symbolic Layout
Physical Layout
ICICLE Interactive Display/Plot Chip Assembly
ATOLL Language Translation Calma GDS II Stream Format CIF Caesar
ARTLIB Language Utilities
Master Technology File System

Figure 1.1: Abstract View of the FLAG Design Process

- The description can be used to simulate the circuit. This permits rapid testing and preliminary debugging of designs.
- Generic functions can be defined, independent of the size of their arguments; for example; a function which adds two bit vectors of any size can be defined.
- In FP, the combining forms specify precedences and parallelism between functions. The use of these forms allows various algorithmic structures to be recognized and exploited.
- The FP combining forms interconnect and instantiate functions yielding graphs with functions as nodes (see [Schl86]) allowing the structure of the function to be extracted.
- Computations can be viewed, in FP, as consisting of two types of activities: directing data movement and changes in value. In FP, the delineation between these two activities is often explicit and can be used to facilitate the extraction of structural information from an FP function (see [Schl86]).
- The algebraic properties of FP present the possibility of transforming an algorithm by applying algebraic identities to its FP specification.

1.4 Symbolic Interpretation

In Step 2, the circuit description is symbolically interpreted[3], through the use of symbolic objects, to create a computation graph of the circuit. The symbolic output object generated by the FP combining forms can be determined from the symbolic outputs of their sub-functions until only nodes corresponding to primitives remain. The resulting graph is the computation graph for the circuit. The replacement of nodes by the structure of their associated functions can be monitored to obtain a hierarchical representation of the computation graph.

The unit of information represented by a single element (atom) of a symbolic object 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

[3]Martine Schlag, "Extracting Geometry from FP for VLSI Layout," UCLA, Los Angeles, California, Tech. Rep., CSD-840043, October 1984., pp.8-11.

most likely be a bit, while in a Fast Fourier Transform (FFT), each atom could represent a complex number. Once the level of representation of the atoms is fixed, the FP primitives of an FP expression can be classified into one of the following two categories.

Computational Primitives These functions have the potential to generate atoms which are not copies of atoms in the input object and/or whose effect is determined by the value of the 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 cannot 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.

1.4.1 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 list is generated for each atom occurring in the computation graph; it is a list of each occurrence of the atom as an input or output object of a primitive. The connectivity of the computation graph can be described by a hypergraph[4]. The primitives of the computation graph are the

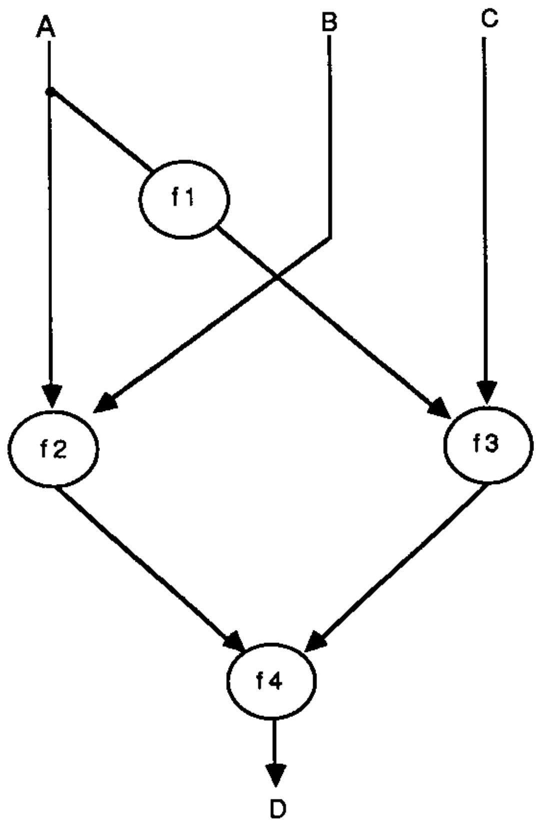
[4]A hypergraph is the generalization of a graph to higher dimensions. It consists of a set of vertices and a set of "hyperedges." Each hyperedge is a non-empty subset of vertices. See Berge,C. **Graphs and Hypergraphs**, North-Holland 1973.

vertices of the hypergraph and the net lists describing the interconnections of the atoms in the hypergraph are the hyperedges.

The 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 the hypergraph. However, FP functional forms and routing functions contain information which can be used to "layout" this hypergraph. Each functional 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 combining forms and routing primitives. Figure 1.2 shows a computation graph and its associated hypergraph.

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. 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 is comprised of three types of nodes. The first type is a primitive of the computation graph or vertex of the hypergraph. The second type is a branch node which is used in representing a hyperedge (net-list). The third type is a crossing node which is needed to obtain a planar graph. The crossing node always has four incident edges, two pairs, each pair belonging to a wire. Examples of these nodes can be found in Figure 1.3. The edges interconnecting these nodes correspond to a single atom and thus can be mapped 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



Nodes: {A,B,C,f1,f2,f3,f4,D}

Hyperedges:

{A,f1,f2}
{B,f2}
{C,f3}
{f1,f3}
{f2,f4}
{f3,f4}
{f4,D}

Figure 1.2: Computation Graph and its Associated Hypergraph

graph belonging to this hyperedge. Figure 1.3 shows the planar embedding for the computation graph of Figure 1.2.

It is possible to define for each FP expression 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 expressions can predictably affect the positions in the "real layout" of nodes of the computation graph. See [Schl86] for more detailed information.

1.4.2 Intermediate Form Generation

The planar graph created from the FP description of the circuit is translated into a format, referred to as the Intermediate Form (IF), for representing the "topological" structure of FP expressions. The intermediate form represents the planar graph and its embedding created from an FP expression. 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. Figure 1.4 breaks the planar graph of Figure 1.3 into cross-sections. 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 are not explicit; elements sharing the same vertical coordinates are only ordered horizontally.

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.

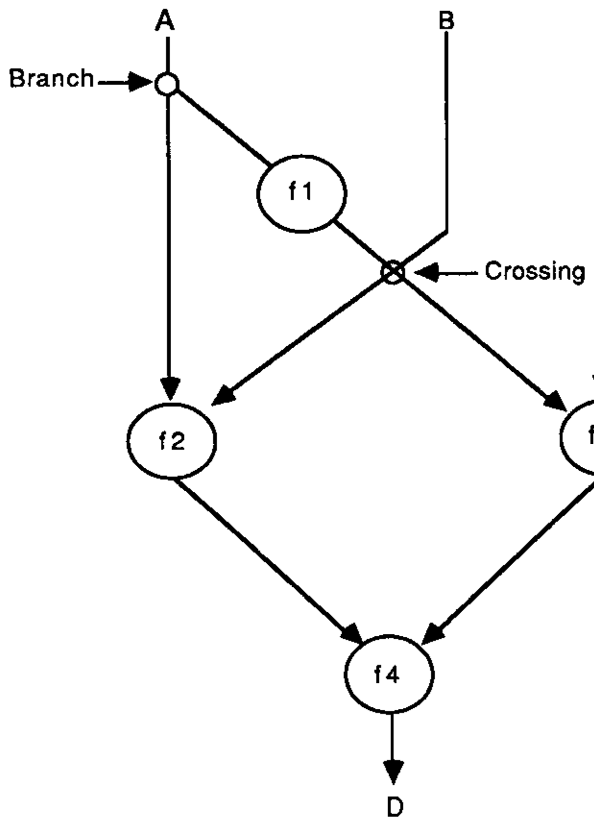


Figure 1.3: Planar Embedding of the Computation Graph

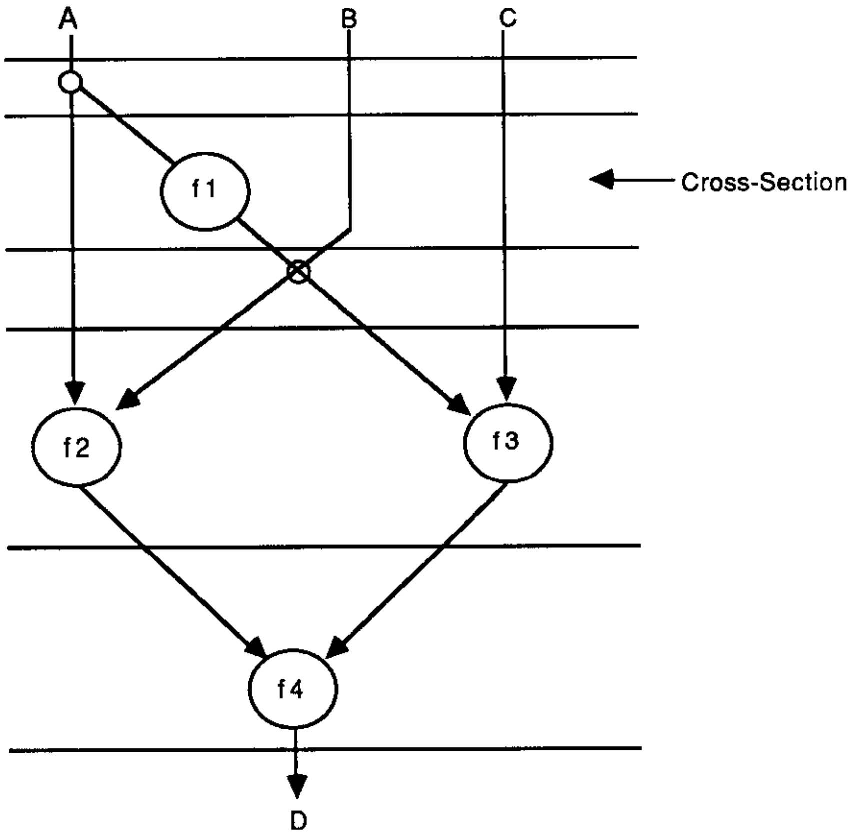


Figure 1.4: Planar Graph Broken into Cross Sections

Formally, the IF consists of FP objects of the following form[5],

$$< PS \;\; CS_1\, CS_2 \ldots \; CS_n > \text{for } n \geq 0,$$

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

$$CS_i =< x_1\; x_2 \ldots \; x_m > \text{ where } x_j \text{ is one of the following,}$$

Free wire An atom (not \$, *, $+$, $\wedge$, and t). Elements of this type are wires which traverse the cross-section without being crossed by any other wires.

Crossing $< * \; w \; * \; u_1 \; u_2 \; \ldots \; u_h >$ such that exactly one u_h is $+$ and at least one is $\wedge$. 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 '$\wedge$'. The other atoms are wires which traverse this cross-section.

Box $< \$ \; \mathit{level} \; \#\mathit{levels} \; \mathit{id} \; \mathit{label} \; \$ \; i_1 \; i_2 \ldots \; i_k \; \$ o_1 \; o_2 \ldots \; o_l \; \$ >$
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 \$ acts as delimiters between these atoms, the input atoms, i_1, i_2, $\ldots i_n$ and the output atoms, o_1, o_2, $\ldots o_m$.

[5]Schlag, "Extracting Geometry from FP," pp.16-19.

Refer to [Schl84] for graphical interpretations of these elements and [Liao83] for a detailed discussion of the process for the assignment of horizontal coordinates to elements.

1.5 Physical Layout and Circuit Simulation

In Step 3, the IF is translated into ABCD, a circuit description language for VIVID[6], and used as input for the VIVID system to perform the physical layout and circuit-level analysis. VIVID stands for the Vertically Integrated VLSI Design System developed at the Microelectronics Center of North Carolina [Roge86, Roge85]. The system is based on a symbolic, virtual-grid design methodology that greatly reduces the design time of custom VLSI circuits. This methodology makes it possible to provide, in a single integrated system, several features: technology independent tools for a wide range of MOS processes (CMOS, nMOS, SOI); scale independent circuit designs; open architecture that simplifies both integration with existing tools and creation of new tools; fast layout debugging using symbolic level circuit simulation; and fully automated mask generation and automated chip assembly. Figure 1.5 presents a conceptual view of the VIVID system[7]. It is divided into two main sections: symbolic layout and physical layout. The symbolic layout section contains the tools used to generate and manipulate symbolic, virtual-grid layouts, specified in ABCD (A Better Circuit Description) Language. The ABCD language provides a direct and well-defined interface to the symbolic, virtual-grid portion of the VIVID System. In the physical layout section, the mask descriptions generated by the compactor are specified in LLAMA (Layout Language for Mask Artwork) Language. Tools are provided to translate between LLAMA and other mask description languages.

[6]C. D. Rogers, and S. W. Daniel, and J. B. Rosenberg, "An Overview of VIVID, MCNC's Vertically Integrated Symbolic Design System," **IEEE Design Automation Conference,** 1985, pp.62-68

[7]Rogers, "An Overview of VIVID," pp.62-68.

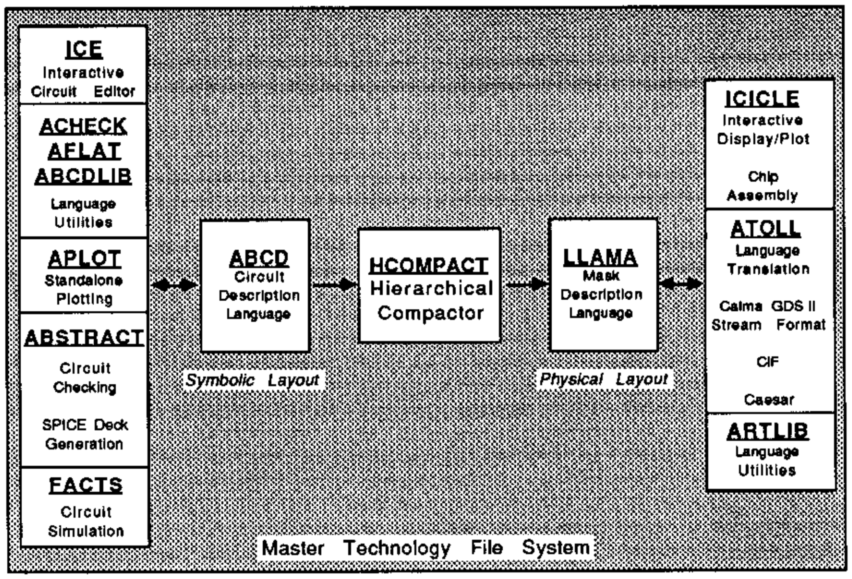


Figure 1.5: Conceptual View of the VIVID System

1.5.1 Symbolic, Virtual-Grid Layout

Symbolic, virtual-grid layout[8] can be viewed as an evolutionary refinement of physical mask layout. In physical mask layout, the designer specifies the circuit by drawing a set of polygons that indicate how to create a mask for each layer in the fabrication process. At the physical mask level, the basic elements of circuit design (such as transistors or contact cuts) are composite structures. Each transistor or contact cut is composed of polygons on several layers that are sized and positioned according to the design rules of the target fabrication process. In physical mask layout, each time one of these composite structures is needed, it is re-created from the component polygons. Symbolic layout provides a solution that eliminates this tedious and error-prone task.

With symbolic layout, symbols are provided to represent the most common structures. The designer organizes the symbols into a layout and the computer translates them into the proper mask representation. In its simplest form, the

[8]Rogers, "An Overview of VIVID," pp.62-68.

translation is done by replacing each symbol with a fixed collection of polygons that implement the desired structure. A more flexible approach to symbol translation is to associate parameters with the symbols and to have a program use the parameters for generating a broad range of structures. For example, the symbol for a transistor might be accompanied by two parameters that specify the width and length of the gate region. The transistor generation program would then use parameters to size the transistor when constructing the mask layout.

Like symbolic design, virtual-grid layout is an extension of physical mask design. In physical mask design, the layout is usually created on a grid. The spacing of the grid represents some "real" spacing (for example, 3 u) and the designer uses the grid as an aid to establish the correct spacing between objects. The function of the virtual grid is the same as for a "real" grid except that the spacing between grid lines does not represent a fixed physical spacing. A symbol's placement captures only the relative geometry of the circuit. (For example, transistor A is above and to the right of transistor B) The actual spacing between two adjacent grid lines is determined by HCOMPACT, the compactor program. The compactor examines the objects on adjacent grid lines and, based on the design rules, determines the correct spacing between the grid lines. It does not perform any optimizations upon the circuit design.

1.5.2 Layout Verification

The VIVID System provides two tools for verifying symbolic virtual-grid layouts; a symbolic level circuit extractor and an interactive circuit simulator[9] early in the layout process. The symbolic level circuit extraction is performed by the ABSTRACT (ABCD Circuit Extractor) program. ABSTRACT references the MTF System to calculate the electrical parameters associated with each circuit element. The calculated values are, by necessity, estimates since the mask generation has not been performed. However, these estimates are relatively accurate for all of the primitives except wires, which are directly dependent upon the final size of the layout. Reasonable estimates of wire length can be obtained by assuming that the spacing between the virtual grid lines will average out over the design. This average grid spacing parameter is coded in the MTF System

[9]Rogers, "An Overview of VIVID," pp.62-68.

and can be tuned by the designer according to the technology being used and the performance of the compactor.

In conjunction with the circuit extraction, ABSTRACT performs error checking and provides the designer either textual or, via ICE, graphical feedback. The types of errors it can detect are overlapping or improper abutment of cells, unconnected or short-circuited components, and improperly named signals.

Circuit simulation is performed by the FACTS (Fast Circuit Simulator) program. The simulator has been designed for MOS simulations and can be used with circuits as large as several thousand devices. The speed of FACTS results from its selection of models and internal structure. Only MOSFET models are used and FACTS precalculates tables of simulation values before beginning a simulation. During the simulation, the designer can choose between two types of current modeling: a simple transistor current model or a more accurate second order model with saturation, linear, and cutoff regions; channel length modulation; drain and source threshold dependence; and capacitance modeling. FACTS also monitors all node voltages and, when the changes are small, increases the time step to avoid redundant or insignificant calculations.

Because of its simpler modeling and the use of symbolic, virtual-grid extraction , FACTS does not provide the accuracy of a full network analysis program. However, FACTS fills a gap between such programs and logic level simulators. It is faster than a detailed circuit simulator but still accurate enough to provide the waveform information necessary for debugging the analog behavior of a circuit. FACTS offers interactive features (such as probe capability to interrupt a simulation and observe any set of circuit nodes) to support its function as a debugging tool.

1.6 Previous Work

[Laht81] showed that a functional style language could be used to specify combinational circuits and investigate their behavior. [Shee84] extended this to sequential systems. [Schl84,Schl86] investigated the extraction of topological information from functional programs. [Mesh84] analyzed timing in functionally specified combinational and sequential systems. [Pate85] explored a design environment based upon functional programs. [Worl86] examined a functional style description for digital systems. [Lieb82,Lim65,Mori82,Vieg84] developed various hardware description languages. [Joha79,Ance83] developed a silicon compiler and a methodology for silicon compilation.

1.7 Objective

Chapter 2 scrutinizes the design cycle with and without FLAG. Chapter 3 discusses some circuits designed with FLAG and examines the results. Chapter 4 is the conclusion. A summary of the syntax of FP is provided in Appendix A. A user's manual for FLAG is provided in Appendix B.

Chapter 2

Design Environment

2.1 Conventional Design Cycle

As is shown in Figure 2.1, a conventional design cycle for hand created VLSI circuits consists of the following 5 steps:

In the **System Definition** stage, the designer defines the functionality of the desired circuit from a circuit design, digital algorithm, circuit schematic, etc. The designer creates a behavioral description of the circuit which describes the system functionality. In the **Architectural Definition** stage, the architecture of the circuit is defined. From this architecture, a block diagram of the circuit is created and the circuit is partitioned into modules. Additionally, the function of the various modules is defined here. In the **Logic Implementation** stage, the modules are created and tested by the designer. Since the designer is responsible for the creation of the modules, he must be knowledgable about all aspects of VLSI circuit design. In the **System Integration** stage, the modules are integrated into the overall design and the module interconnections are laid out. Since this is done by hand, the process of system integration is slow and tedious. Further, errors can be introduced because of human error. These errors are unfortunately difficult to detect and tedious to correct. Finally, in the **System Test** stage, the functionality of the circuit produced is verified to ensure that it operates as intended. This is a slow process and may not be possible for large circuits.

In the typical design process, there are two feedback loops. The **System Integration - System Verification** loop attempts to uncover and correct any

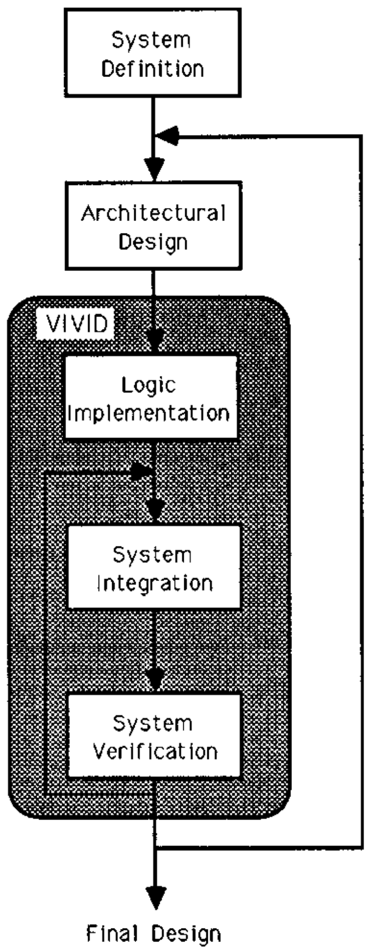


Figure 2.1: Conventional Design Cycle

interconnection errors that resulted from the **System Integration** stage. Unfortunately, the process of error detection is slow and difficult. The correction of these errors must be done by hand and can be a rather tedious and possibly difficult task. The introduction of additional errors by the designer is not beyond expectation either. In the **Architectural Design - System Verification** loop, major design errors are detected and an attempt made to correct them. Possible errors include

- System functionality errors
- Timing errors
- Input-Output errors

These errors generally require a redesign of the circuit and a reiteration of the entire design cycle.

In the conventional design cycle, architectural exploration can not be introduced as part of the design cycle due to the large amount of time required in the generation of a single physical layout. Alteration of the system architecture is justifiable only if a major design error is discovered. It isn't reasonable for a designer to invest the time and effort necessary to change the system architecture unless a major payoff is evident. Therefore, architectural exploration becomes too expensive an activity to be included as a part of the design process because the probable gains do not outweigh the costs of the activity.

2.2 Limitations

A typical hand-based design cycle suffers from the following limitations:

- The logic implementation and system integration steps are implemented from the bottom up.
- The logic implementation and system integration steps require VLSI design expertise by the designer.
- The logic implementation and system integration steps are tedious and error-prone.

- The scaling or combining of circuits requires re-execution of the system integration-system verification loop thereby requiring a large investment of time and effort by the designer.

These limitations of a typical design cycle are sufficient to prevent architectural exploration from being included as part of the design cycle.

2.3 Advantages

A typical hand-based design cycle provides the following advantages:

- The designer has complete control over the layout produced.
- The designer can perform some hand optimizations upon the circuit at design time resulting in smaller and more efficient circuits.

2.4 FLAG Design Cycle

As is shown in Figure 2.2, the FLAG design cycle consists of the following 3 steps:

In the **System Specification and Design** stage, the designer creates a behavioral description of the digital algorithm or circuit design of interest in an applicative language, FP. To verify the correctness of the behavioral description, the description can be simulated functionally by the FP interpreter. Embedded in the behavioral description is the desired architecture for the physical implementation. In the **Layout Generation** stage, a physical layout is created from the behavioral description by FLAG and the VIVID. In the **Circuit Analysis** stage, the VIVID tools are used to examine the physical and electrical characteristics of the circuit for acceptability.

In the FLAG design cycle, there is only a single feedback loop namely the **System Specification - Circuit Analysis** loop. In this loop, there is little designer intervention required. As a result, the translation is quick and no additional errors can result from the translation of the behavioral description to a physical implementation. In addition, a different physical design can be created by modifying the behavioral description. No additional work is required and the designer retains control over the physical layout.

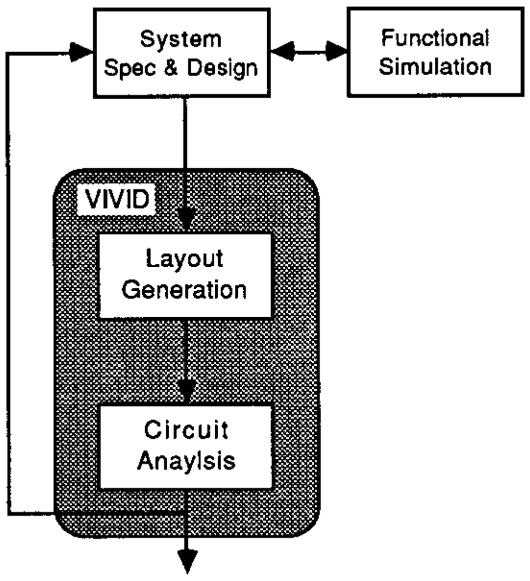


Figure 2.2: FLAG Design Cycle

In the FLAG design cycle, the time required to generate a physical implementation from a behavioral description is kept to a minimum. Therefore, it becomes feasible to experiment with various system architectures in order to find the most efficient design.

2.5 Limitations

The FLAG design cycle suffers from the following limitations:

- Flow of data between modules is in a vertical direction. This is due to the current mapping scheme between the behavioral description and its physical realization.
- The designer does not have complete control over the placement of circuit elements within a layout.
- Designs are not as small as possible.

- Module inputs enter only at the top of modules and their outputs appear only at the bottom.

These limitations are due mostly to the applicative language used to describe the system and the mapping used to translate the behavioral description into a physical layout.

2.6 Advantages

The new design cycle provides the following advantages:

- Shortens the design cycle. It eliminates the tedious and error-prone portion of the design cycle and automates it.
- Designers are freed from low-level VLSI design issues.
- A correct FP description of the system produces a layout that is functionally correct.
- The scaling of circuits can be performed without requiring modification of the circuit description or additional effort by the designer.
- Different circuits can be combined together by combining their respective circuit descriptions without requiring a large investment of time and effort by the designer.
- A hierarchical design style is supported.
- Any given behavioral description corresponds to a single layout.

The advantages of the new design cycle make it possible to include architectural exploration as part of the design process. They shorten the design cycle enough that the designer can experiment with different architectures without investing a large amount of time.

2.7 An Example

The design of a binary decoder will be followed through both design cycles to illustrate the differences between them. A binary decoder takes N binary inputs and generates 2^N binary outputs. For any particular input pattern only one output is true.

2.7.1 Typical Design Cycle

System Definition:

Input: $< X_{N-1}, \ldots, X_0 >$

Output: $< Y_{2^N-1}, \ldots, Y_0 >$ where:

$$Y_i = \begin{cases} 1 & \text{if } i = \sum_{j=0}^{N-1} X_j \times 2^j \\ 0 & \text{Otherwise} \end{cases}$$

Architectural Design A block diagram for the decoder design is given in Figure 2.3. The only modules required are inverters and 2-input AND-gates.

Logic Implementation Trivial since the design for an inverter and an AND-gate are fairly standard and are probably part of any cell library.

System Integration The modules must be placed by hand according to the block diagram and the interconnections between them must be laid out. Unfortunately, this is often done by hand and is quite slow and tedious.

System Verification The design must be simulated in order to ensure correct operation of the circuit. If any errors are found, they must be corrected by hand.

2.7.2 FLAG Design Cycle

System Specification and Design:

Input: $< X_{N-1}, \ldots, X_0 >$

Output: $< Y_{2^N-1}, \ldots, Y_0 >$ where:

$$Y_i = \begin{cases} 1 & \text{if } i = \sum_{j=0}^{N-1} X_j \times 2^j \\ 0 & \text{Otherwise} \end{cases}$$

The function **decode** takes a vector of arbitrary length and returns a fully decoded version of the input. The format of the input is $< x_{n-1} \ldots x_0 >$.

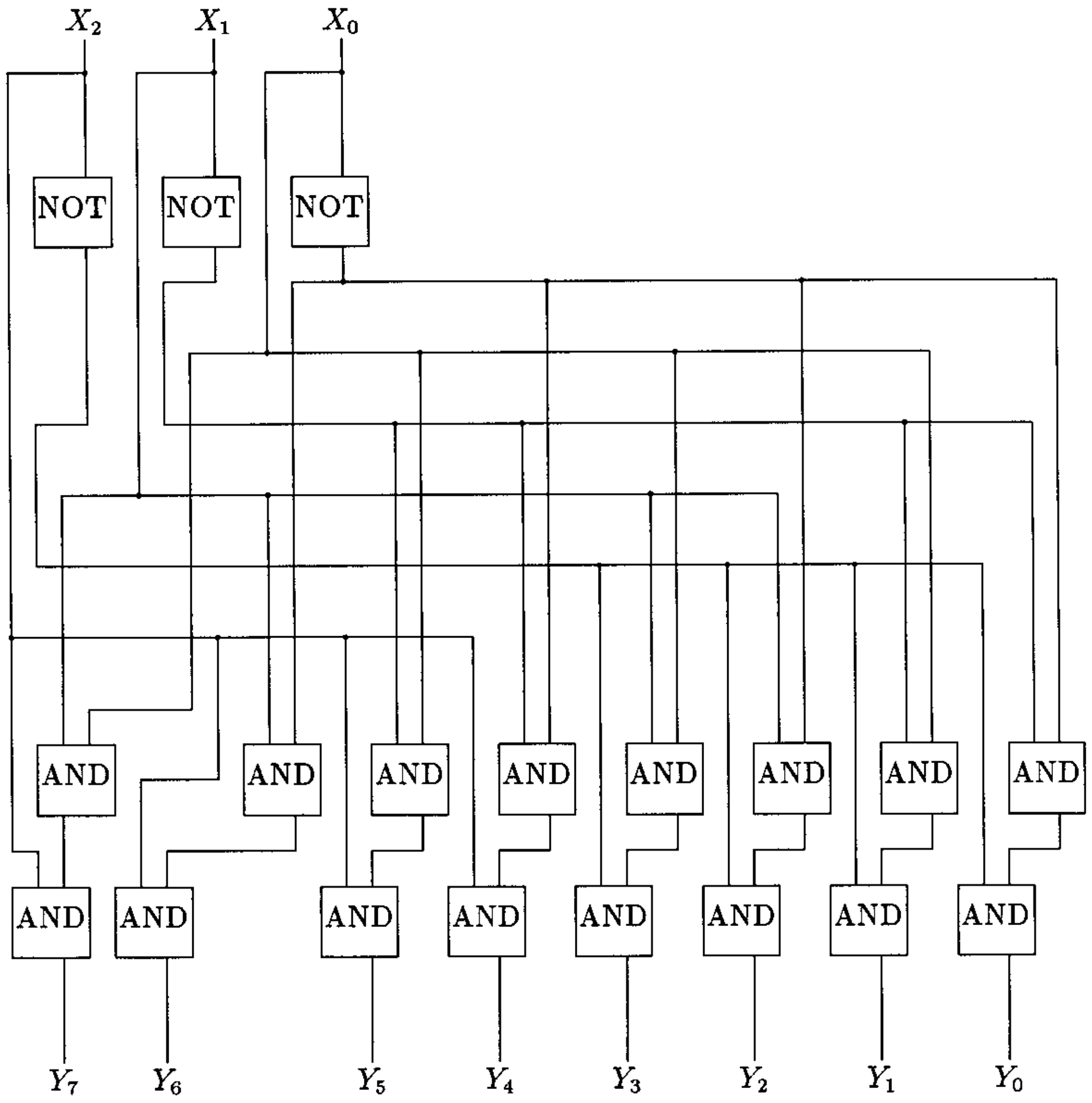


Figure 2.3: 3-Bit Decoder Block Diagram: Conventional Design

The format of the output is $< y_{2^n-1} \ldots y_0 >$, where

$$y_k = \begin{cases} 1 & \text{if } \underline{X} = k \\ 0 & \text{otherwise} \end{cases}$$

The **decode**[1] function can be broken into the following steps[2].

```
If (length of input = 1)
    1 Perform a decode function upon a single bit
Else
    1 Split the input into two vectors
    2 Recursively apply decode to each vector to fully decode it
    3 Form all possible pairs from the two decoded vectors
    4 Reduce pairs to single bit vectors
```

```
DEFINE decode
  IF (length = 1)
  THEN decode_1@1
  ELSE
    &andg @
    cross_match @
    &decode @
    split
  ENDIF
END
```

The function **decode_1** takes an input of a single-bit and returns a fully decoded version of the input. The format of the input is $< x >$. The format of the output is $< x\,\overline{x} >$.

[1]John Shelby Worley, "A Functional Style Description of Digital Systems", MS Thesis UCLA 1986, pp. 78-80.

[2]See Appendix A for a description of the FP language.

```
DEFINE decode_1
  [id, notg]
END
```

The function **cross_match** takes an input of two vectors of arbitrary length and returns a vector containing all the possible pairs that can be formed from the input. The format of the input is $<< x_{2^{\frac{n}{2}}-1} \cdots x_0 >< y_{2^{\frac{n}{2}}-1} \cdots y_0 >>$. The format of the output is $<< x_{2^{\frac{n}{2}}-1}\, y_{2^{\frac{n}{2}}-1} > \cdots < x_{2^{\frac{n}{2}}-1}\, y_0 >< x_{2^{\frac{n}{2}}-2}\, y_{2^{\frac{n}{2}}-2} > \cdots < x_0/; y_0 >>$.

```
DEFINE cross_match
  concat @
  &distl @
  distr
END
```

Correct functioning of the design can be verified through FP interpreter.

Layout Generation: The FP description is used by the layout generator to produce a physical layout of the proposed decoder design. A block diagram of the design is given in Figure 2.4.

Circuit Analysis: The physical layout's characteristics, area, module placement, delay are examined. If the design is not satisfactory, then another architecture can be created by modifying the FP description and re-running the layout generator.

2.7.3 Discussion

In the typical design cycle, the system integration-system verification loop is the most time-consuming and tedious from the perspective of the designer. A lot of time and effort is spent in producing the physical realization of a proposed design since every aspect of the physical layout process must be performed by hand. Any errors introduced during the layout process must be detected and corrected by hand. With the FLAG design cycle, the designer is relieved of the burden of translating the block diagram into a physical layout. The designer only has

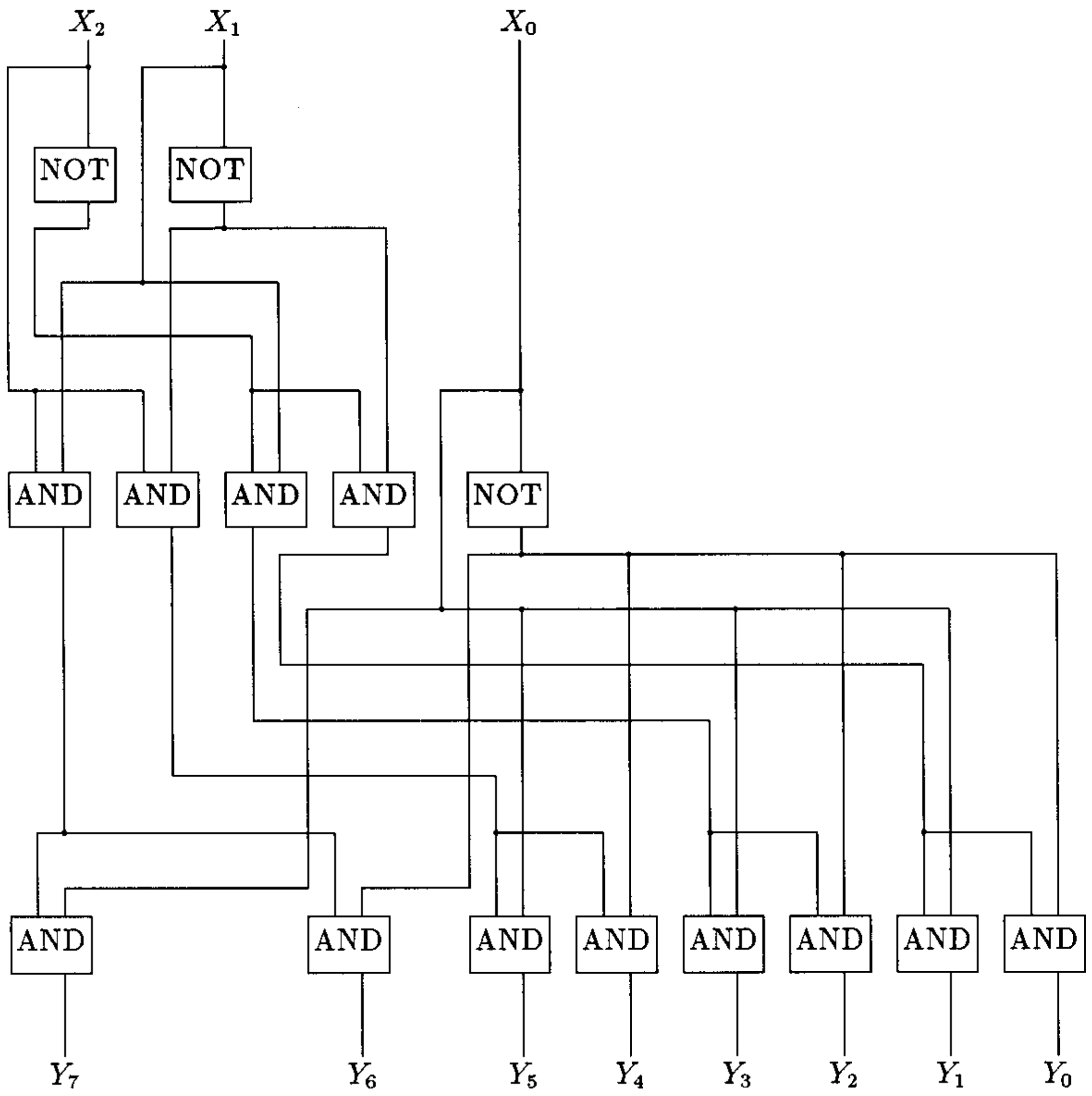


Figure 2.4: 3-Bit Decoder Block Diagram: FLAG Design

to generate the FP description of the circuit. FLAG creates a physical layout directly from the FP specification of the architecture without any additional input from the designer. An advantage of this is that no design errors will be in the physical layout if the FP specification of the design is correct. During the creation of the FP specification, the designer can verify the correctness of the specification with the FP interpreter. Checking the FP specification is a much simpler and quicker process than laying out an entire circuit and having to perform a circuit simulation to see that it functions correctly. With FLAG, once the FP description of the circuit has been created, a simple version of the circuit can be created and tested. For instance, with the decoder, a 2-bit decoder can be created and examined. If the resulting circuit is acceptable, a decoder of arbitrary length can be created without any additional work from the designer. In the typical design cycle, it is not possible to scale circuits as easily as with FLAG. Another advantage, provided by FLAG, is that composite circuits can be created by just combining the the FP descriptions of different circuits.. In the normal design cycle, the creation of a composite circuit would require at least the re-execution of the system integration-system verification loop which would require a lot of work from the designer. The FLAG design cycle frees the designer from the unimportant activities of module placement and interconnection and allows him to concentrate upon the more interesting activity of architecture design.

Chapter 3

Design and Evaluation Experiments

A number of examples is presented here in order to examine some of the features and drawbacks of FLAG. The examples range from SSI logic networks to complex LSI logic networks. For each example, a CMOS design is produced by hand and by FLAG. The CMOS layouts are then compared on the basis of area, the bounding box of the layout being used as an approximation; percentage of white space present in the design; delay through the circuit; and design time required. Design time is defined as the time needed to create a physical layout from an algorithmic specification of a circuit. Since VIVID translates a symbolic layout (ABCD) into a physical layout, the design time for the hand-generated circuits is the time needed to create the symbolic layout (ABCD). For the FLAG circuits, since FLAG translates the behavioral description into a symbolic layout (ABCD) for VIVID, the design time for the FLAG circuits is the time needed to generate the correct behavioral description.

3.1 Simple Combinational Circuits

In this section, several SSI-level modules, including an exclusive-OR, AND-OR network, and a NAND network are designed and a discussion of the results is given in Sec. 3.1.4.

3.1.1 Exclusive-Or

The exclusive-OR gate examined here will perform the exclusive-OR operation of two inputs. A canonical sum of products expression for the exclusive-OR of two inputs is: $(\overline{a}b) + (a\overline{b})$. Two approaches can be taken in the design of the exclusive-OR gate. Either a design based upon logic gates or a design based upon transmission gates can be used. Designs based upon transmission gates tend to be smaller in area and faster in terms of delay in comparison to their logic gate based counterparts. A logic diagram for the transmission gate exclusive-OR gate is shown in Figure 3.1[1]. The operation of the transmission gate exclusive-OR gate can be explained as follows:

1. When A is high, $\overline{A}$ is low. Transistor pair 1 and 2 act as an inverter with $\overline{B}$ appearing as the output. The pass gate formed by the transistors 3 and 4 is open.
2. When A is low, $\overline{A}$ is high. The pass gate (transistor 3 and 4) is closed, passing B to the output. The inverter pair (transistor 1 & 2) is disabled.

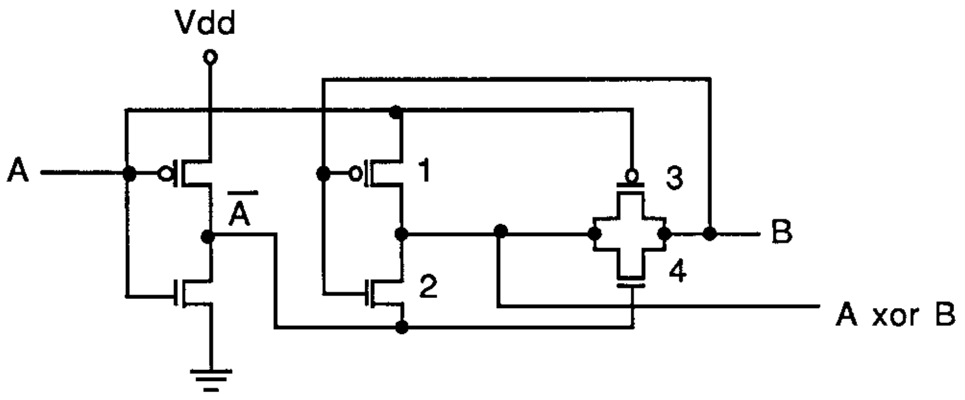


Figure 3.1: XOR Schematic

In the description of the transmission-gate exclusive-OR, it is necessary to generate functional descriptions for the circuit elements used in the implementation: n-transistor, p-transistor, and a pass-gate. Functionally, the n-transistor

[1]N. Weste and K. Eshraghian, **Principles of CMOS VLSI Design: A Systems Perspective,** (Reading, Massachusetts: Addison-Wesley, 1985), pp. 317.

acts like a gateway, passing its data if the gate input is high and turning itself off if the gate signal is low. This can be described functionally as an 'AND'ing together of the gate and data input. The format of the input is $< gate, data >$ and the format of the output is $< out >$.

```
DEFINE n
  andg
END
```

The p-transistor behaves in a similar fashion as the n-transistor except that it passes its data through when the gate input is low. Therefore, in description the gate input is inverted before the 'AND'ing is done. The format of the input is $< gate, data >$ and the format of the output is $< out >$.

```
DEFINE p
  andg @ [notg@1,2]
END
```

The pass-transistor has two control signals, cntlA and cntlB, and a data input. It also acts like a gateway, this time passing its output if cntlB is high and cntlA is low. Otherwise, the output is turned off. The format of the input is $< cntlA, cntlB, data >$ and the format of the output is $< out >$. The pass-gate can then be described as follows:

```
DEFINE pass
  !andg @ [notg@1,2,3]
END
```

In the implementation of the exclusive-OR, the outputs of the various circuit elements are 'wire-OR'ed together. Functionally, this situation can be described an 'OR'ing together of the outputs as long as one and only one output is active at any time. The format of the input is $< inA, inB >$ and the format of the output is $< out >$.

```
DEFINE wor
  org
END
```

Now that all the circuit elements have been described, the description for the exclusive-OR can be shown. In the first line, $\overline{A}$ is generated. A, $\overline{A}$, B are then routed to the various circuit elements. The outputs of these elements are then 'wired-OR'ed together in the last line. The format of the input is $< A, B >$ and the format of the output is $< A \oplus B >$.

```
DEFINE xor
  !wor @
  [pass,p@[3,1],n@[3,2]] @
  [1,notg@1,2]
END
```

A block diagram of the FLAG design is given in Figure 3.2. The layout that is produced from this expression is shown in Figure 3.3. The layout for the hand design is shown in Figure 3.4. A comparison of the designs for a two input exclusive-OR gate is shown in Table 3.1.

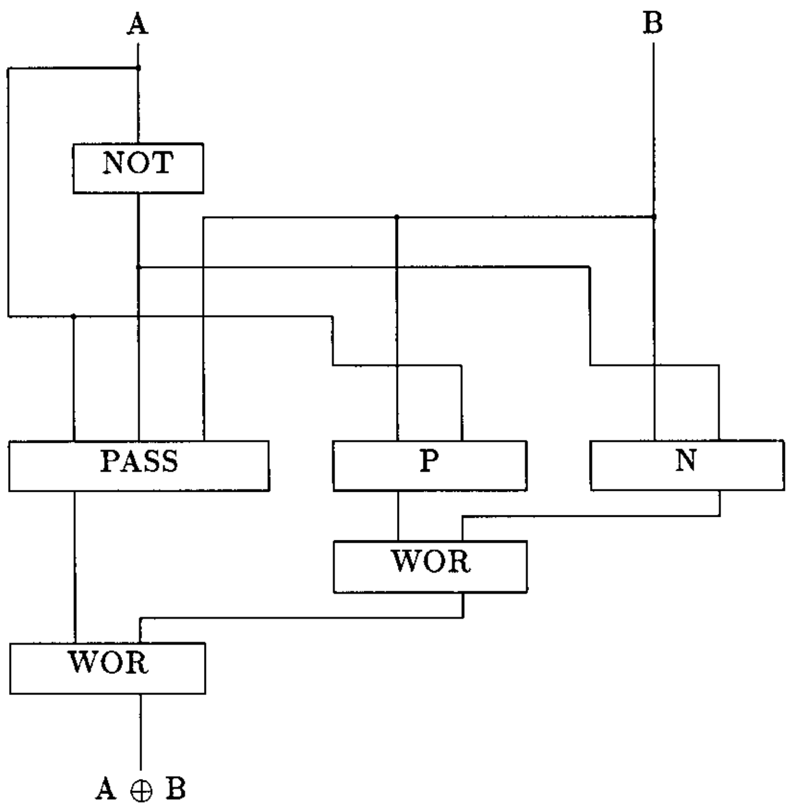


Figure 3.2: Exclusive Or Logic Diagram: FLAG Design

XOR	**FLAG**	**Hand**
Area	89×205	49×72
White Space	66%	44%
Average Delay	4.3 ns	2.3 ns
Maximum Delay	4.5 ns	2.3 ns
Design Time	1 hour	3 hours

Table 3.1: Exclusive Or Comparison

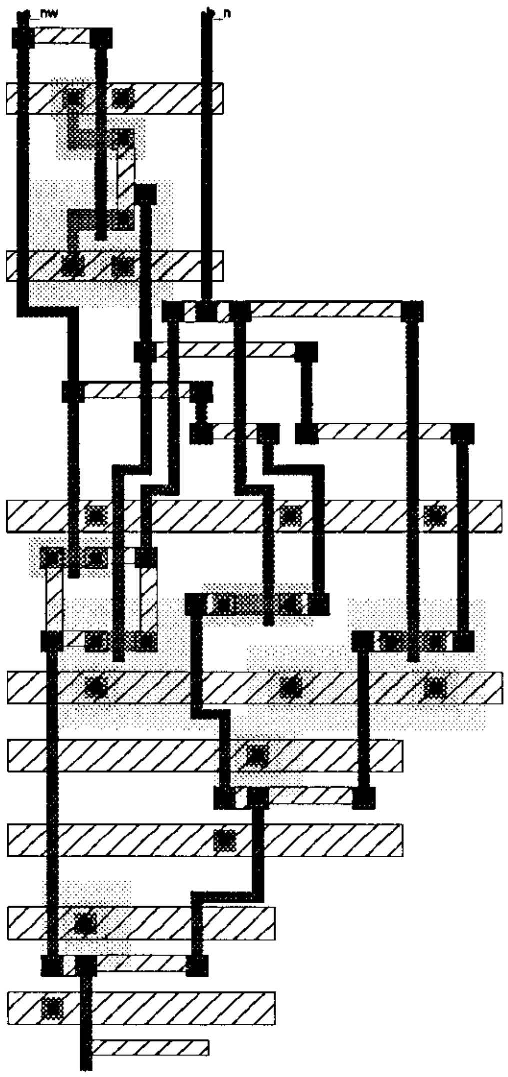


Figure 3.3: Exclusive-OR: FLAG layout

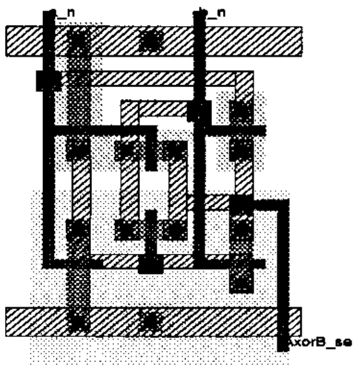


Figure 3.4: Exclusive-OR: Hand layout

3.1.2 AND-OR Networks

In this subsection, an arbitrary AND-OR network will be designed. The boolean expression is: $f(a,b,c) = (\overline{a}b) + (a\overline{b}c) + (b\overline{c})$ The hand implementation of this expression requires 9 logic gates: three inverters to form the complements of a, b, c, four 2-input AND-gates to form the product terms, and two 2-input OR-gates to form the final sums. A logic diagram for the circuit is shown in Figure 3.5.

The functional description of the network follows the boolean expression. First, the complement of the inputs is generated. Next, the various product terms are created and then 'OR'ed together in the last line.

```
DEFINE and-or(a,b,c)
  !org @
  [andg@[2@1,1@2],andg@[1@2,2@3],!andg@[1@1,2@2,1@3]] @
  &[id,notg]
END
```

The layout that is produced from this FP description is shown in Figure 3.6.

For the hand design of the AND-OR network, we have a choice between using pass transistors or logic gates in the design. Logic gates were selected since an AND-OR network is being synthesized. The layout for the hand design is shown in Figure 3.7. A comparison of the designs for the AND-OR network is given in Table 3.2.

After closer examination of Table 3.2, we see that the FLAG version of the AND-OR network operated faster than the hand version. At first, this seems surprising; but upon closer examination of the designs, it becomes clear why this is so. The reason for the discrepancy in speed between the FLAG version of the AND-OR network and the hand version is due to the critical path of some of the product terms of the AND-OR network. Table 3.3 lists the critical paths for each product term of the AND-OR networks for both layouts. The term $(\overline{a}b)$ is the one of interest. In the FLAG layout, the term $(\overline{a}b)$ had a delay of three gate delays while undergoing four gate delays in the hand layout. This resulted in the FLAG circuit operating faster than the hand circuit.

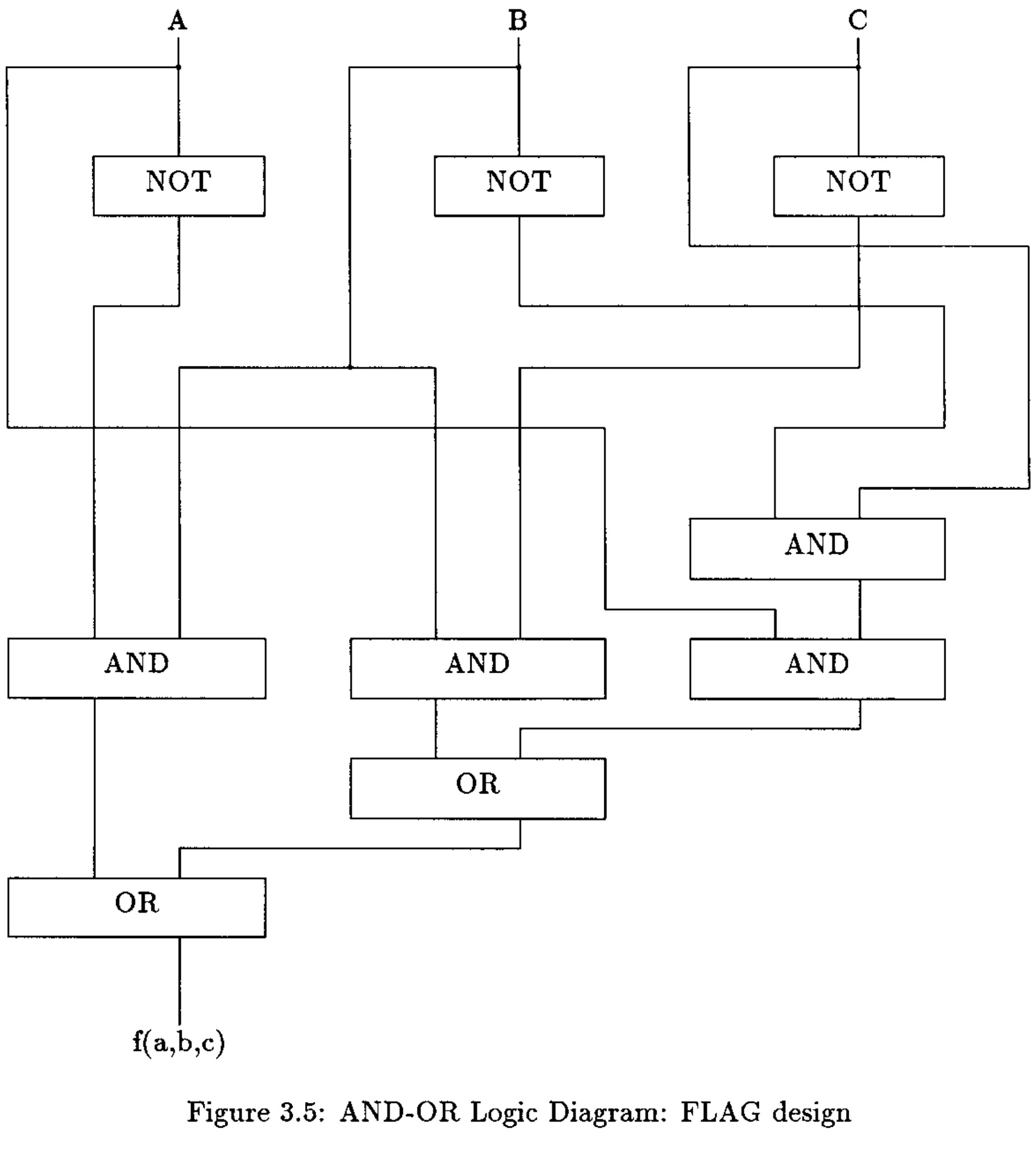


Figure 3.5: AND-OR Logic Diagram: FLAG design

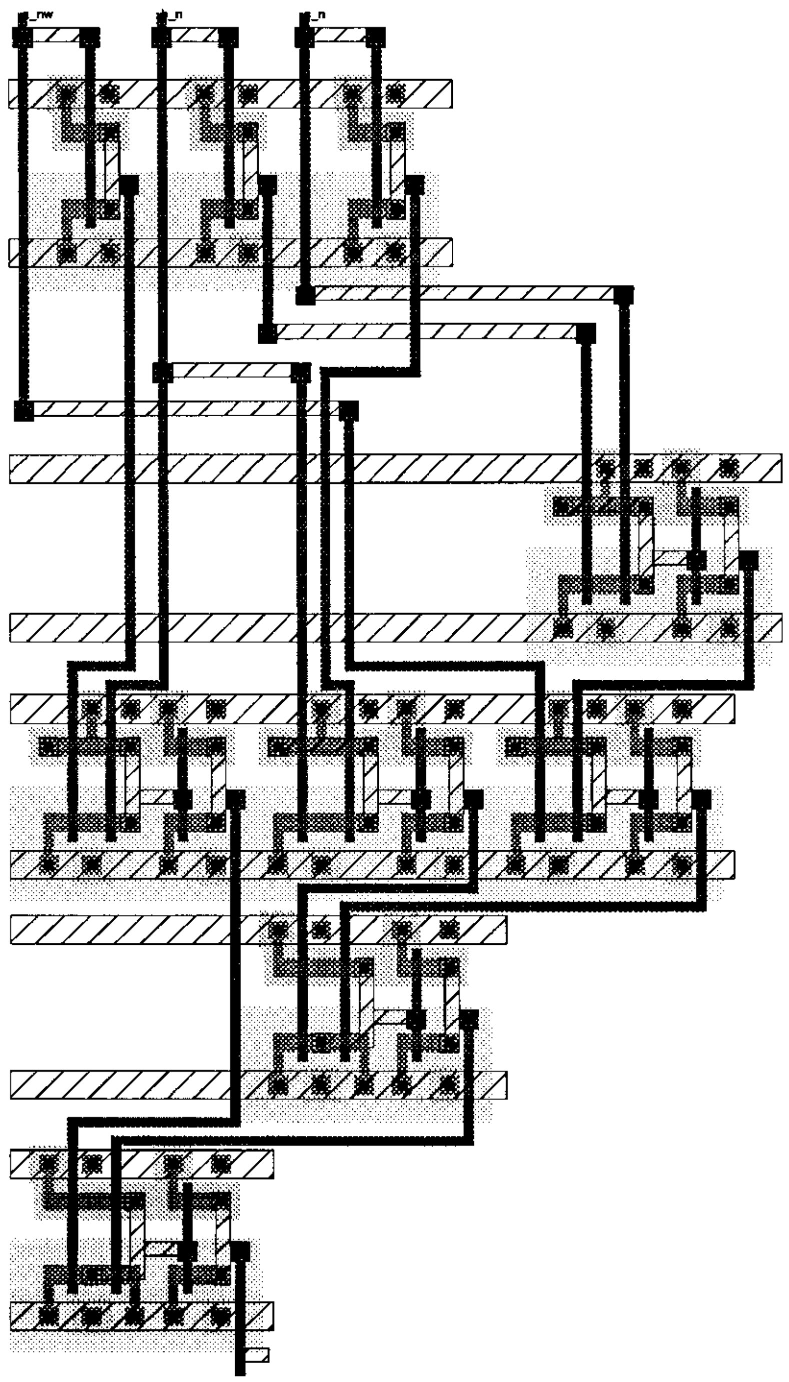


Figure 3.6: AND-OR: FLAG layout

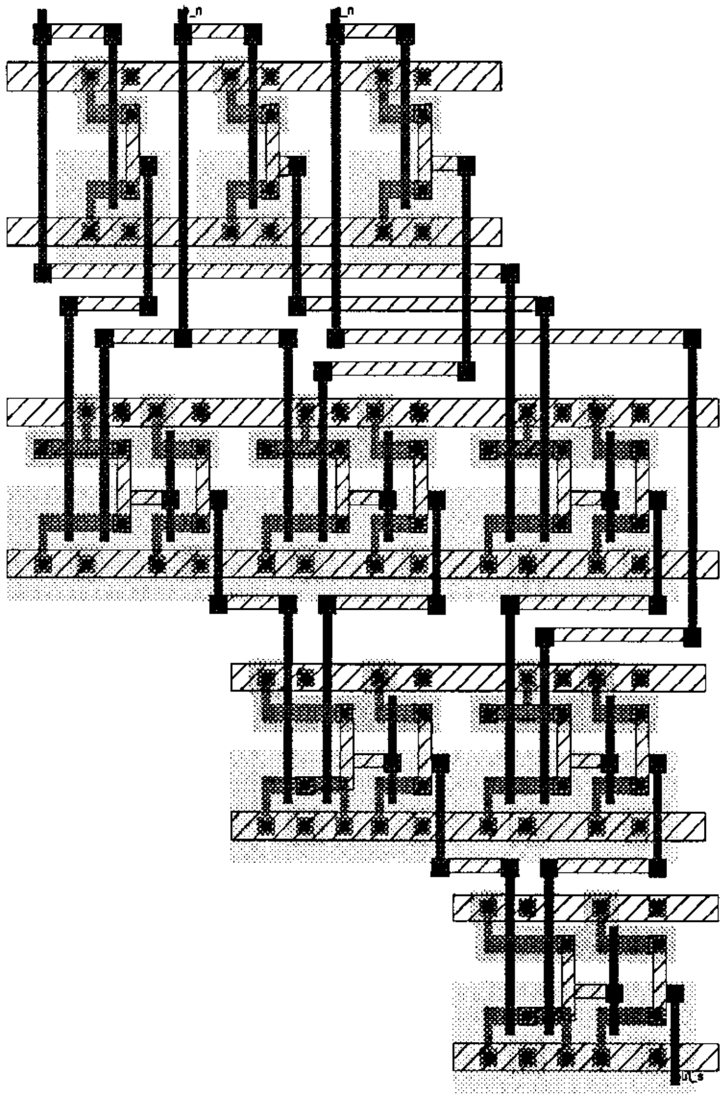


Figure 3.7: AND-OR: Hand layout

AND-OR	**FLAG**	**Hand**
Area	162×288	163×234
White Space	56%	55%
Average Delay	14.1 ns	16.3 ns
Maximum Delay	16.5 ns	20 ns
Design Time	10 minutes	$2\frac{1}{3}$ hours

Table 3.2: AND-OR Network Comparison

Term	FLAG	Hand
$\overline{a}b$	3 gate delays	4 gate delays
$a\overline{b}c$	5 gate delays	4 gate delays
$b\overline{c}$	4 gate delays	4 gate delays

Table 3.3: Critical Paths in AND-OR Network

3.1.3 NAND Networks

In this subsection, an arbitrary NAND network will be designed. The boolean expressions were arbitrarily chosen and are:

$$f_1(a,b,c,d) = \overline{a}c + \overline{b}c + \overline{d}$$
$$f_2(a,b,c,d) = \overline{a}\overline{b} + \overline{b}cd + abcd$$

The implementation of these expressions requires 8 2-input NAND gates. By performing the following algebraic manipulations upon the expressions f_1 and f_2, equations are created which are simpler to implement than the original expressions.

For $f_1(a,b,c,d)$:

$$\overline{a}c + \overline{b}c + \overline{d}$$
$$(\overline{a} + \overline{b})c + \overline{d}$$

For $f_2(a,b,c,d)$:

$$\begin{aligned}
&\overline{a}\overline{b} + \overline{b}cd + abcd \\
&\overline{a}\overline{b} + cd(\overline{b} + ab) \\
&\overline{a}\overline{b} + a\overline{a} + cd(\overline{b} + ab) \\
&\overline{a}\overline{b} + a\overline{a}(b + \overline{b}) + cd(\overline{b} + ab) \\
&\overline{a}\overline{b} + a\overline{a}b + a\overline{a}\overline{b} + cd(\overline{b} + ab) \\
&\overline{a}\overline{b} + a\overline{a}b + cd(\overline{b} + ab) \\
&\overline{a}(\overline{b} + ab) + cd(\overline{b} + ab) \\
&(\overline{b} + ab)(\overline{a} + cd)
\end{aligned}$$

The resulting expressions for $f_1(a,b,c,d)$ and $f_2(a,b,c,d)$ require 9 NAND gates to implement: three for $f_1(a,b,c,d)$ and six for $f_2(a,b,c,d)$. A logic diagram for the NAND network is shown in Figure 3.9.

The functional description of the network follows the boolean expression.

```
DEFINE nand-network(a,b,c,d)
  [ lins(nandg)@[1,2,3,4],
    nandg @ [id,id] @ nandg @
      [ lins(nandg)@[1,2,2], rins(nandg)@[1,3,4] ]
  ]
END
```

The layout that is produced from this FP description is shown in Figure 3.8.

For the hand design of the NAND network, we again have a choice between using transmission gates or logic gates in the design. Logic gates were selected since a NAND network is being synthesized. In the hand design, a single NAND gate can be removed since two of the NAND gates make use of the output of the same gate. This reduces the number of NAND gates required in the design to eight. The layout for the hand design is shown in Figure 3.10. A comparison of the designs for the test NAND network is given in Table 3.4.

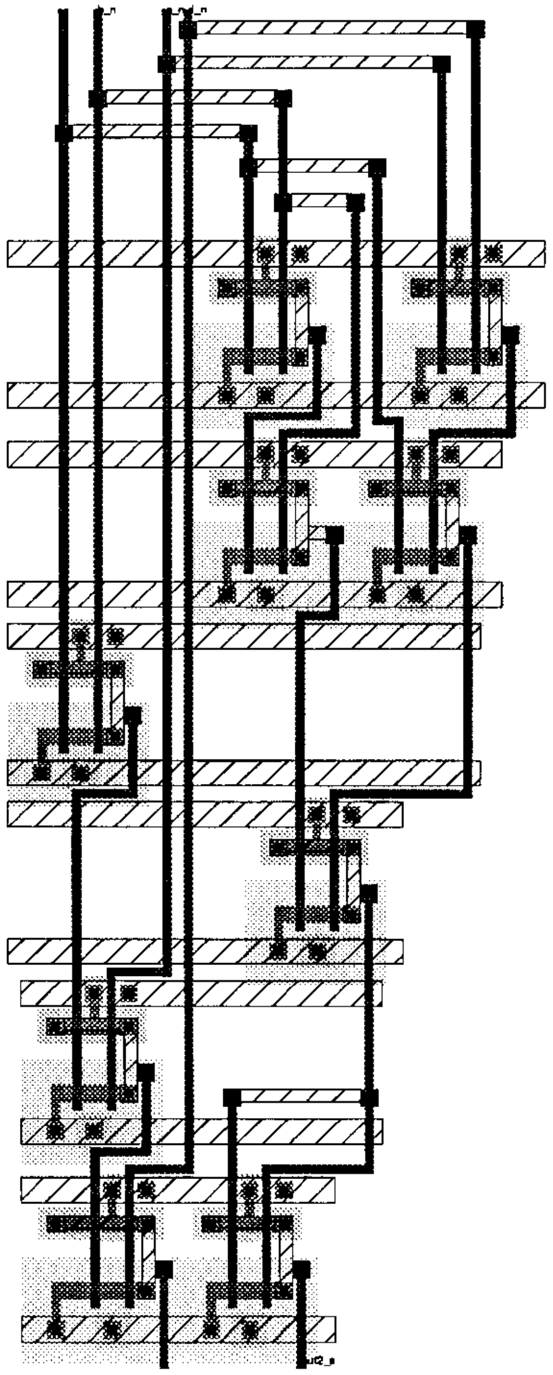


Figure 3.8: NAND: FLAG layout

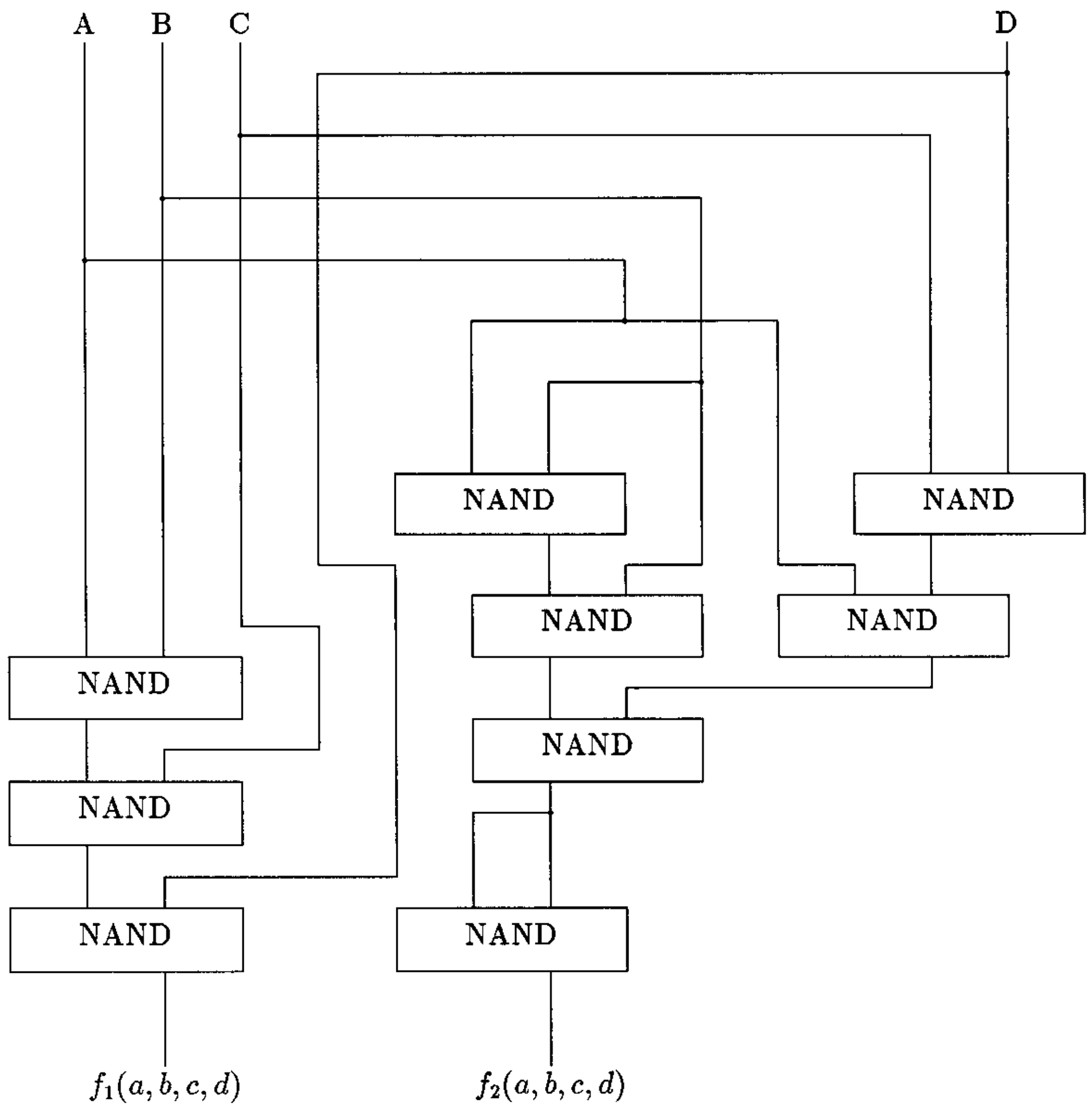


Figure 3.9: NAND Network Logic Diagram: Hand design

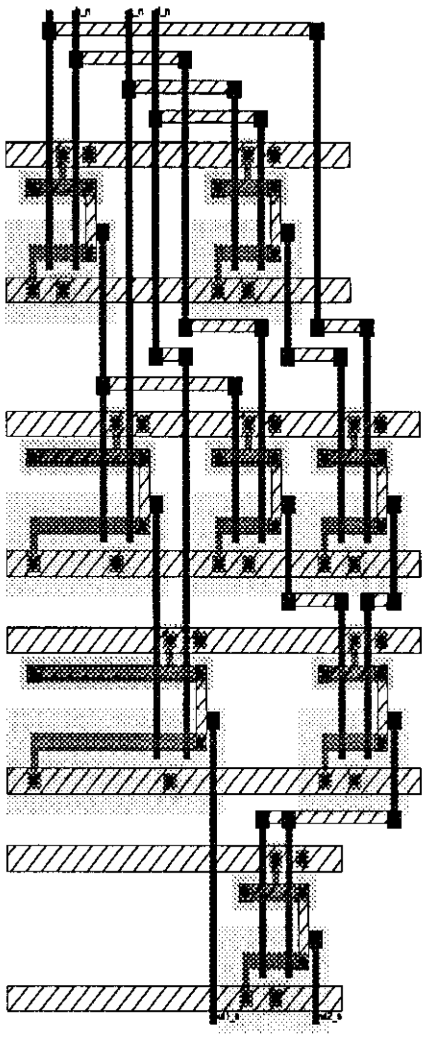


Figure 3.10: NAND: Hand layout

NAND	**FLAG**	**Hand**
Area	125×322	125×241
White Space	54%	52%
Average Delay	11.4 ns	10.3 ns
Maximum Delay	14 ns	14 ns
Design Time	10 minutes	$2\frac{1}{6}$ hours

Table 3.4: NAND Network Comparison

3.1.4 Concluding Remarks

As the three previous examples have shown, the hand layouts are smaller and a little faster than their FLAG produced counterparts, with the exception of the AND-OR network. The differences between the hand and FLAG layouts can be attributed to the fact that the designer was able to perform some optimization upon the hand versions of the circuits during the design cycle. Additionally, during the layout process, the designer could place and pack the circuit elements together in such a fashion as to minimize the total area used by the circuit and the delay of the circuit. The optimizations were made at the expense of the designer since additional time and effort had to be expended by the designer in order to perform the optimizations upon the hand design. In the FLAG produced circuits, the circuit elements were placed in a top to bottom fashion. This resulted in circuits that were larger and slower than their hand counterparts.

3.2 MSI Modules

In this section, several MSI-level modules: a full adder, a multiplexor, an encoder, and a ripple carry adder are designed. The results are discussed in Sec. 3.2.5.

3.2.1 Full Adder

A full adder is a combinational circuit that takes three inputs and adds them together to produce a 2 bit vector: <carry, sum>. The switching expressions are

$$\begin{aligned} Sum &= ABC + A\overline{B}\overline{C} + \overline{A}\overline{B}C + \overline{A}B\overline{C} \\ Carry &= AB + A(A + B) \end{aligned}$$

A logic diagram for a full adder is given in Figure 3.11[2]. A transistor schematic for a full adder is given in Figure 3.12[3]. Since the carry out signal ($\overline{Carry}$) is used in the generation of Sum, Sum will be delayed with respect to $Carry$. This facilitates the use of such a circuit in a n-bit ripple carry adder where the carry signal must "ripple" through the stages.

[2]Weste, pp. 312.

[3]Weste, pp. 312.

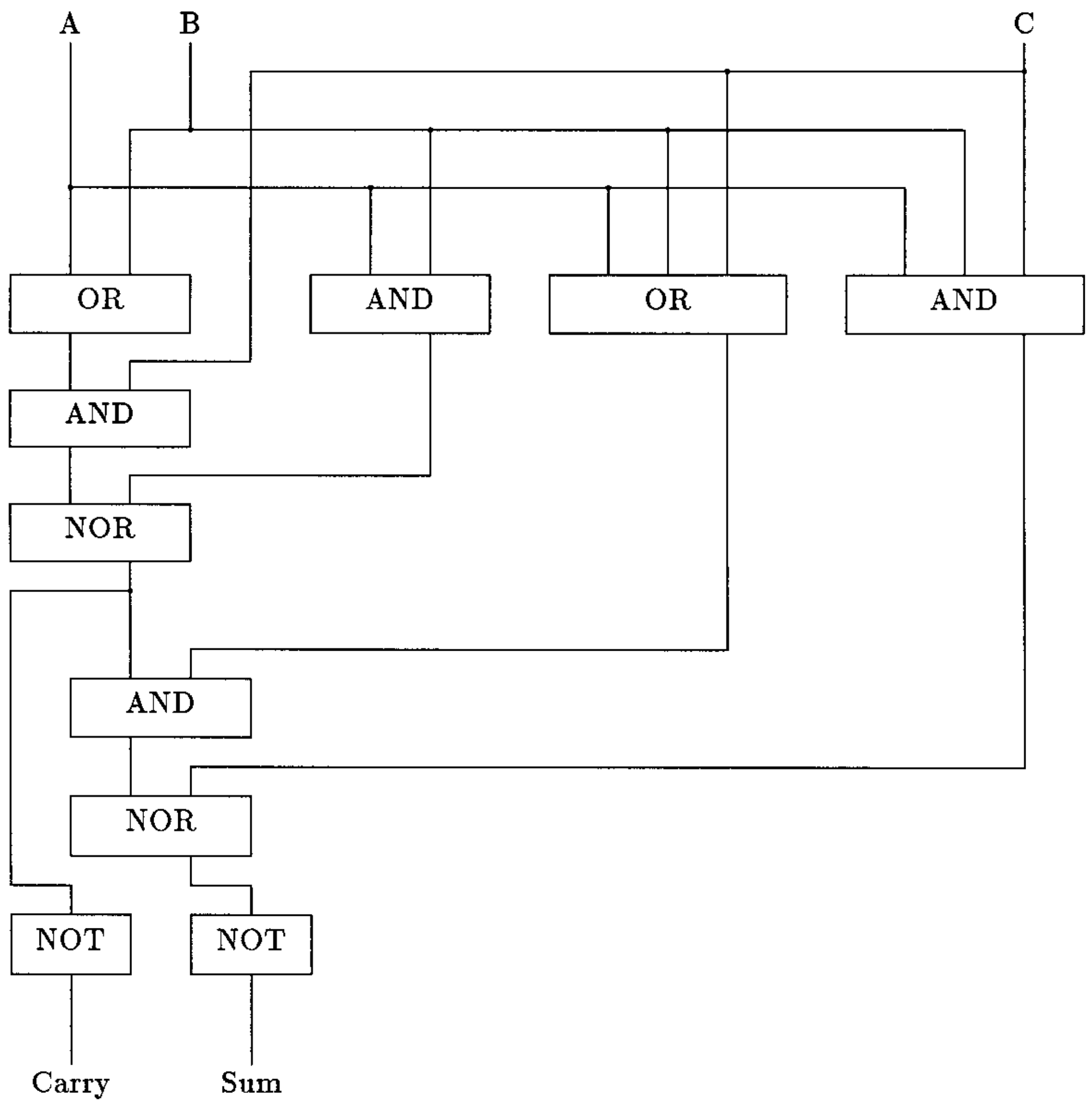


Figure 3.11: Full Adder Logic Diagram

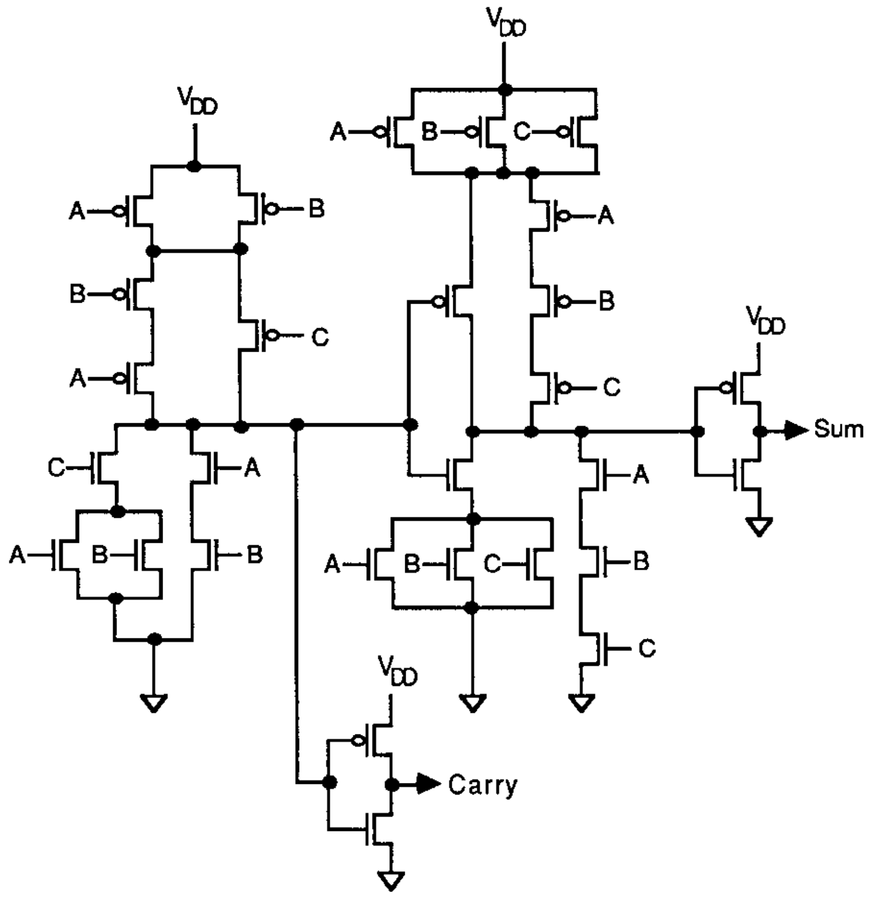


Figure 3.12: Full Adder schematic

The FLAG description of the full adder is given below. The function **fulladder** takes a vector of three inputs and returns the sum of the inputs. The format of the input is $< a\ b\ c >$. The format of the output is $< Carry\ Sum >$. The function can be broken down into the following steps.

1. Connect up all pullup and pulldown transistors to the inputs according to the transistor schematic for the full adder.

2. Wire-OR the pullup and pulldown transistors together according to the transistor schematic for the full adder.

3. Connect up the n and p transistors to generate $\overline{Carry}$ according to the transistor schematic for the full adder.

4. Connect up the n and p transistors to generate $\overline{Sum}$ according to the transistor schematic for the full adder.

5. Complement $\overline{Carry}$ and $\overline{Sum}$.

```
DEFINE full_adder
  &notg @
  [ 1, !wor@[ p@[1,2], 3, n@[1,4], 5 ] ]@
  [ !wor@[ p@[2,p@[1,4]],
           p@[3,4], n@[3,5],
           n@[1,6]],
           7,
           p@[3,p@[2,p@[1,7]]],
           8,
           n@[3,n@[1,9]]
         ]@
  [ 1,
    2,
    3,
    wor@[4,5],
    wor@[6,7],
    8,
    !wor@[9,10,11],
    !wor@[12,13,14],
    15
  ] @ concat @
  [ id,
    &Pup@[1,2],
    &Ndn@[1,2,2],
    &Pup@[1,2,3],
    &Ndn@[1,3,2,2]
  ]
END
```

Since the design of the full adder makes use of pullup and pulldown transistors, functional descriptions of these circuit elements must be created.

Functionally, the pullup transistor returns a high signal (1) if its gate input is low and turns itself off otherwise. This can be described functionally as a 'Not'ing of the gate input. The format of the input is $< gate >$ and the format of the output is $< out >$.

```
DEFINE Pup
  notg
END
```

Functionally, the pulldown transistor returns a low signal if its gate input is high and turns itself off otherwise. This can be described functionally as an 'AND'ing together of the gate input and a low signal (0). The format of the input is $< gate >$ and the format of the output is $< out >$.

```
DEFINE Ndn
  andg @ [%0,id]
END
```

The function definitions for **P, N, WOR** can be found on pages 32 and 32.

A block diagram of the FLAG design is given in Figure 3.13 The layout produced by the FP description for the full adder is shown in Figure 3.14 The layout produced from the hand design is shown in Figure 3.15. A comparison of the designs for the full adder is given in Table 3.5. Clearly, the FLAG design is inferior with respect to the hand design. The reasons for this are discussed in Sec. 3.2.5.

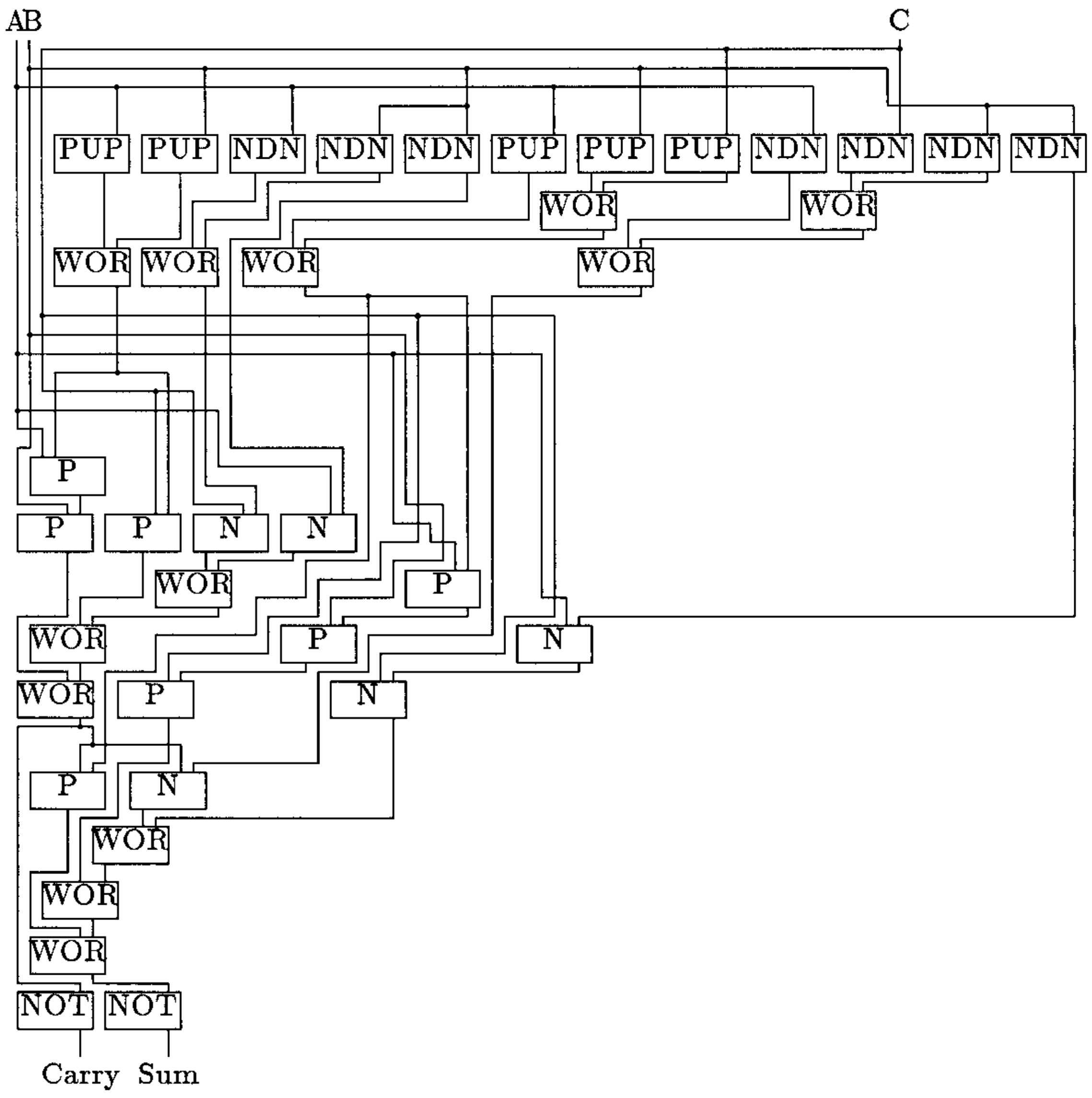


Figure 3.13: Full Adder Logic Diagram: FLAG design

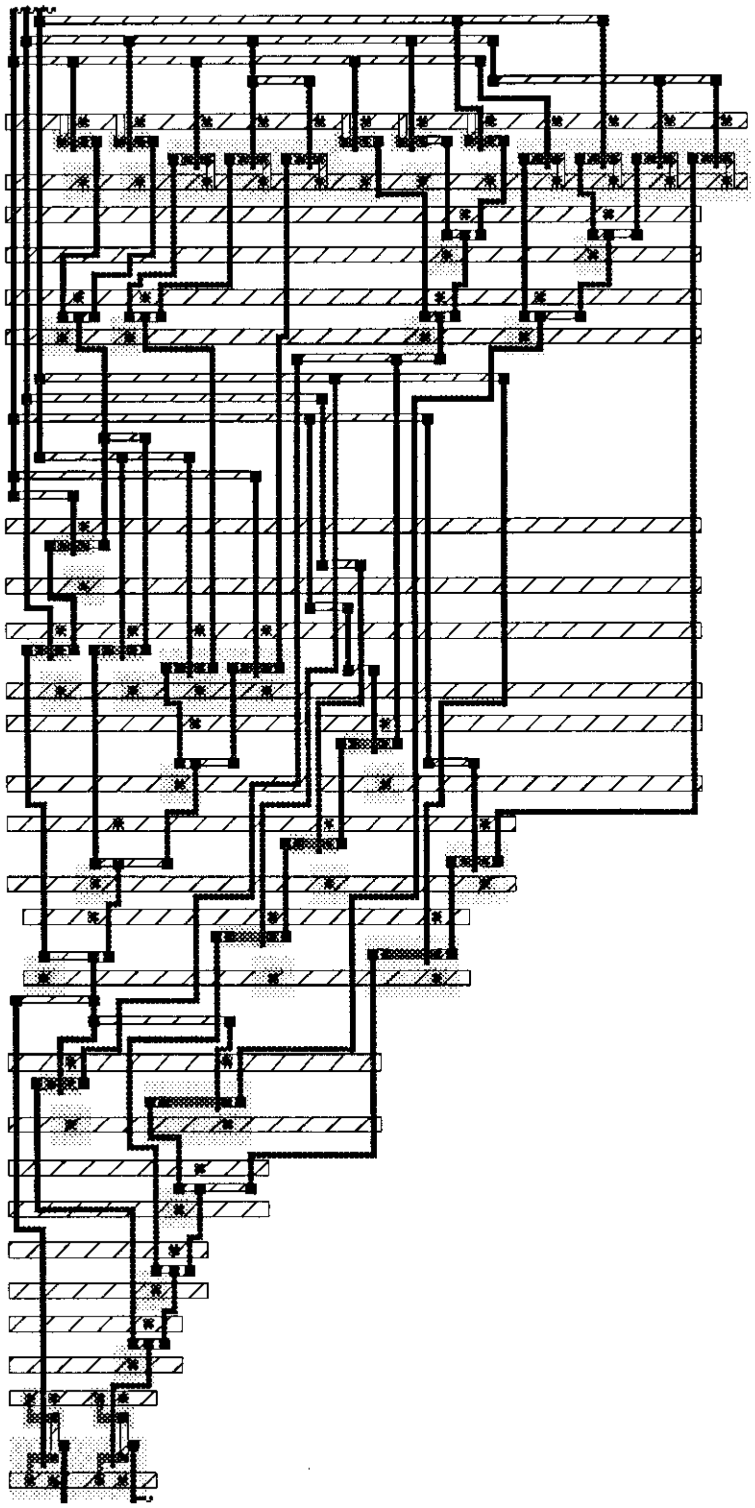


Figure 3.14: Full Adder: FLAG layout

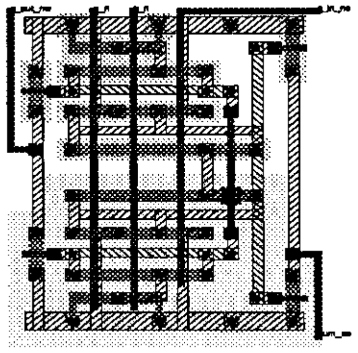


Figure 3.15: Full Adder: Hand layout

Full Adder	**FLAG**	**Hand**
Area	289×594	87×17
White Space	65%	58%
Average Delay	23 ns	14 ns
Maximum Delay	30 ns	22 ns
Design Time	2 hours	6 hours

Table 3.5: Full Adder Comparison

3.2.2 Multiplexor

A multiplexor is a combinational system which performs a selection function. It takes N control inputs, $\underline{c}=< c_{N-1}, ..., c_0 >$ and 2^N data inputs, $\underline{d}=< d_{2^{N-1}}, ..., d_0 >$, each of which can consist of either a single bit vector or multiple bit vector. The output is selected as shown below:

$$Y = d_{\underline{c}}, \qquad \underline{c} = \textstyle\sum_{i=0}^{N-1} c_i \times 2^i$$

In the multiplexor, the N control signals are first decoded. Each of these decoded control signals is then combined with each bit of the corresponding data vector and then OR'ed to produce the output.

In the design of the multiplexor, a choice between a logic gate implementation and a transmission gate implementation can be made. A transmission gate acts like a gateway; when it is turned on, data can pass through. When the gate is turned off, the output is in a high impedance state. This allows the output of multiple transmission gates to be connected together in a wired-OR fashion, avoiding the large AND-OR networks required for the logic gate implementation of the multiplexor. One drawback of transmission gates is that they require both the complemented and uncomplemented form of their respective control signals. The transmission gate design for the multiplexor was chosen. A logic diagram of the design of a two-input multiplexor is given in Figure 3.16.

The function **mux**[4] takes two inputs, a vector of control inputs and a vector of data vectors and implements a selection function using the control inputs to choose one data vector to send as the output. The format for the input is $<< Cntl_{N-1} \ldots Cntl_0 >< < d_{n-1}^{m-1} \ldots d_0^{m-1} > \ldots < d_{n-1}^0 \ldots d_0^0 >>>$. The format for the output is $< d_{n-1}^i \ldots d_0^i >$, $0 \leq i < m,\ i = \sum_{j=0}^{N-1} Cntl_j \times 2^j$ The multiplexor function can be broken into the following steps.

1a Decode the control signals and generate their complements

1b Insure that the data is in the proper form

2 Route the decoded control signals to their respective data vectors

[4]Worley, "Functional Style Description," pp. 86.

3a For each data vector, distribute its control signal across all its component bits

3b Route control and data signals to pass-gates

4 Connect together pass-gate outputs to form output vector

Since pass-gates require both the complement and uncomplemented version of its control signal, it is necessary to generate the complement of the decoded control signals. The outputs of the pass-gates are wire-OR'ed together so that the K^{th} bit of all the data vectors are connected together to form the K^{th} bit of the output and this is done for all the bits of the output.

```
DEFINE mux
  &!wor @
  trans @
  &(&(pass@apndr)@distl) @
  trans @
  [ &[notg,id]@decode@1, &(IF (input = atom))@2
                           THEN [id]
                           ELSE id
                           ENDIF
  ]
END
```

The definition of the function **wor** can be found on page 32 and the function **pass** can be found on page 32.

The function **decode** takes a vector of arbitrary length and returns a fully decoded version of the input. The format of the input is $< x_{n-1} \ldots x_0 >$. The format of the output is $< y_{2^n-1} \ldots y_0 >$, where

$$y_k = \begin{cases} 1 & \text{if } \underline{X} = k \\ 0 & \text{otherwise} \end{cases}$$

The **decode** function can be broken into the following steps.

If (length of input = 1)

1 Perform a decode function upon a single bit

Else

1 Split the input into two vectors

2 Recursively apply **decode** to each vector to fully decode it

3 Form all possible pairs from the two decoded vectors

4 Reduce pairs to single bit vectors

```
DEFINE decode
  IF (length = 1)
  THEN decode_1@1
  ELSE
    &andg @
    cross_match @
    &decode @
    split
  ENDIF
END
```

The function **decode_1** takes an input of a single-bit and returns a fully decoded version of the input. The format of the input is $< x >$. The format of the output is $< x\, \overline{x} >$.

```
DEFINE decode_1
  [id, notg]
END
```

The function **cross_match** takes an input of two vectors of arbitrary length and returns a vector containing all the possible pairs that can be formed from the input. The format of the input is $<< x_{2^{\frac{n}{2}}-1} \ldots x_0 >< y_{2^{\frac{n}{2}}-1} \ldots y_0 >>$. The format of the output is $<< x_{2^{\frac{n}{2}}-1}\; y_{2^{\frac{n}{2}}-1} > \ldots < x_{2^{\frac{n}{2}}-1}\; y_0 >< x_{2^{\frac{n}{2}}-2}\; y_{2^{\frac{n}{2}}-2} > \ldots < x_0\; y_0 >>$.

```
DEFINE cross_match
  concat @
  &distl @
  distr
END
```

The layout produced from the FP expression is shown in Figure 3.17. The logic diagram of the hand design of a multiplexor is given in Figure 3.18. The layout for the hand design is shown in Figure 3.19. A comparison of the designs for the multiplexor is given in Table 3.6.

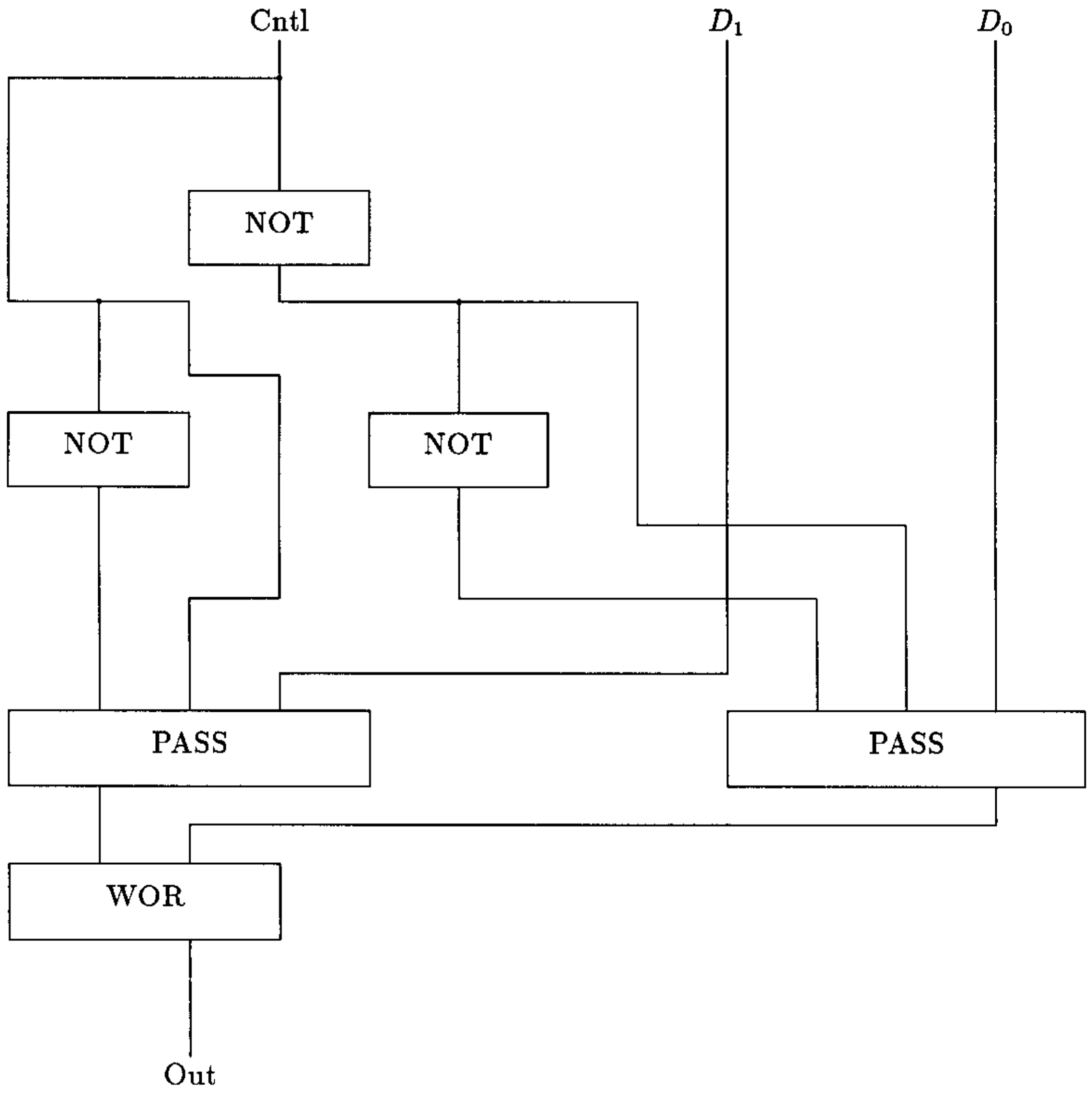


Figure 3.16: 2-Input Multiplexor Logic Diagram: FLAG design

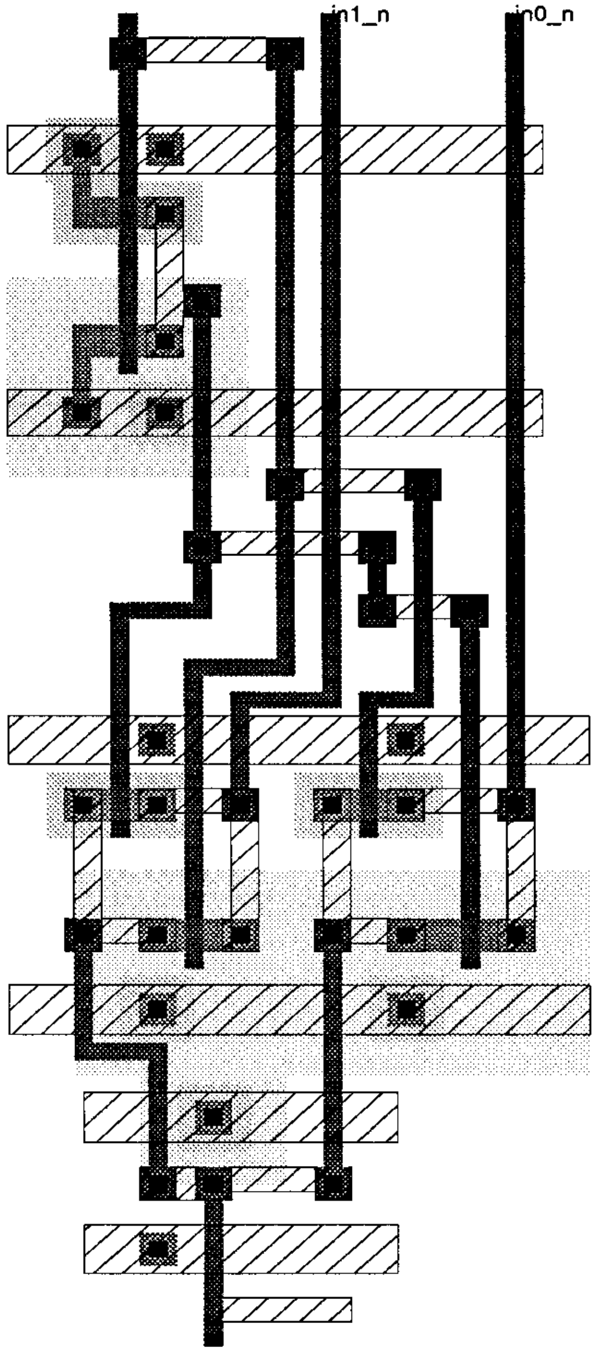


Figure 3.17: 2-Input Multiplexor: FLAG layout

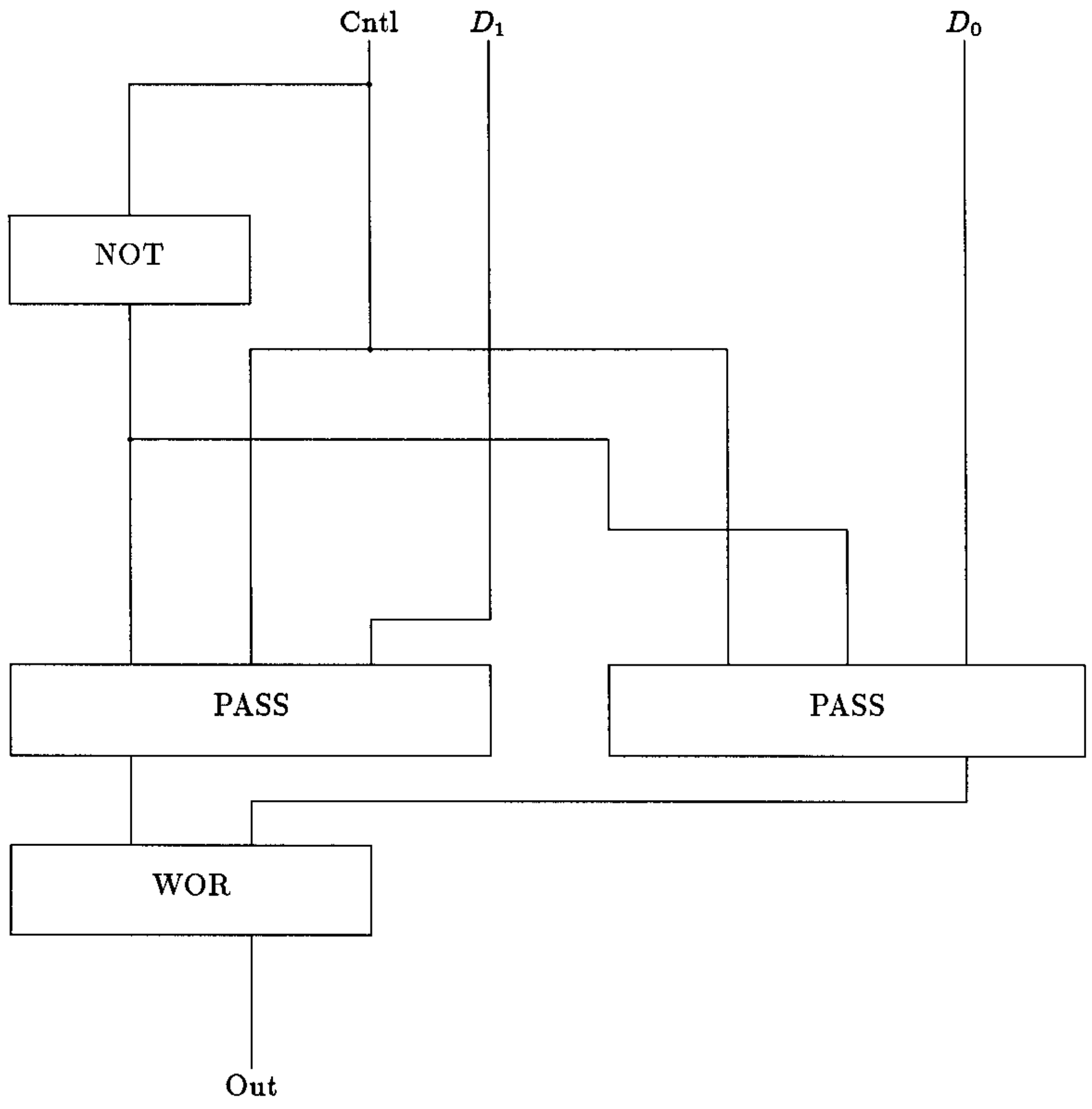


Figure 3.18: 2-Input Multiplexor Logic Diagram: Hand design

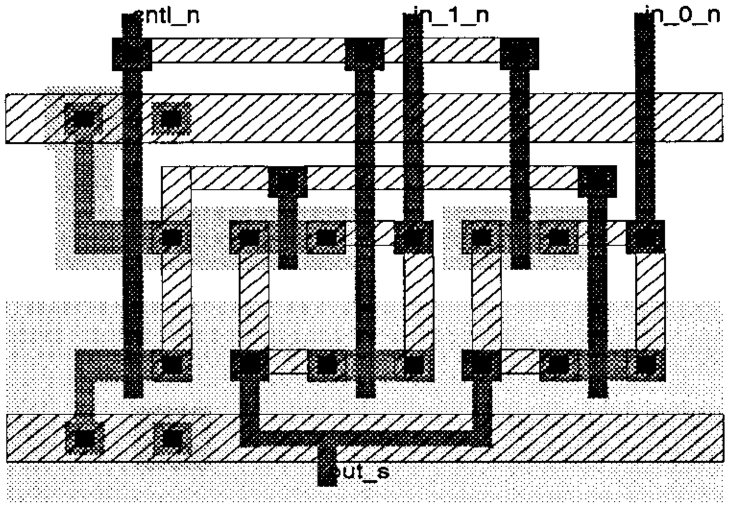


Figure 3.19: 2-Input Multiplexor: Hand layout

Multiplexor	FLAG	Hand
Area	63 × 165	74 × 61
White Space	57%	53%
Average Delay	2.5 ns	1.3 ns
Maximum Delay	4 ns	2 ns
Design Time	30 minutes	2 hours

Table 3.6: Multiplexor Comparison

3.2.3 Encoder

An encoder is a combinational system which performs an encoding function upon 2^N inputs $\underline{X} = (x_{2^N-1}, ..., x_0)$ to produce a N-bit output vector, $\underline{Y} = (y_{N-1}, ..., y_0)$. Only one of the 2^N inputs can be true at any time and $\underline{Y}$ represents the index of the input line that is true as a binary number. A high level description of an encoder is:

$$\underline{Y} = i \qquad \text{if } x_i = 1$$
$$\underline{Y} = \sum_{j=0}^{N-1} y_j 2^j$$

An encoder is typically implemented as a collection of OR-gates. Each bit of the output examines a set of 2^{N-1} of the input lines. If any of the input lines within a set is true, then the output bit associated with that set is true. The set of inputs examined by each output bit varies with bit position. An algorithm for determining the coverage sets is given below.

For the K^{th} bit of the output (0 is the least significant position).

- Divide the inputs into groups of size 2^K
- 'OR' together the 1^{st}, 3^{rd}, 5^{th}, 7^{th}, ... groups starting with the most significant group.
- Result is the K^{th} bit of the output.

For the $(K+1)^{th}$ bit of the output (0 is the least significant position).

- Pair off the groups created for the K^{th} bit position starting with the most significant group.
- Concatenate the pairs to form new groups of $2^{(K+1)}$ size.
- 'OR' together the $1^{st}, 3^{rd}, 5^{th}, 7^{th}$, ... groups starting with the most significant group.
- Result is the $(K+1)^{th}$ bit of the output.

A logic diagram for the design of an eight-bit encoder is given in Figure 3.20.

The function **encoder**[5] takes an input of 2^N inputs and performs an encoding function upon them to generate a N-bit output vector. The format of the input is $< x_{2^{2n}-1} \ldots x_0 >$. The format of the output is $< y_{n-1} \ldots y_0 >$. In the functional description, the input is converted into a collection of vectors if it isn't already and the function **encode_bits** is called.

```
DEFINE encoder
  encode_bits @
  &(IF (input is an atom))
    THEN [id]
    ELSE id
    ENDIF
END
```

The function **encode_bits** takes a collection of vectors and recursively applies the encoding algorithm presented earlier to generate the output. The format of the input is $<< x_{2^{2n}-1} > \ldots < x_0 >>$. The format of the output is $< y_{n-1} \cdots y_0 >$. The operation of the function can be broken into the following steps.

```
If (length of input = 2)
    1 Select the first group and reduce it to a single output
Else
    (For the $(K+1)^{th}$ bit)
    1a Pair off the input and concatenate them together
    (For the $K^{th}$ bit)
    1b Select the $1^{st}$, $3^{rd}$, $5^{th}$, ... groups and combine into a single vector
    (For the $(K+1)^{th}$ bit)
    2a Apply encode_bits upon the result
    (For the $K^{th}$ bit)
    2b Reduce the vector to a single output
```

[5]Worley, "Functional Style Description," pp. 83-85.

```
DEFINE encode_bits
  IF (length = 2)
  THEN [assoc_or@1]
  ELSE
    apndr @
    [ encode_bits @
        &concat@pair ,
      assoc_or @
        concat @ &1@pair
    ]
  ENDIF
END
```

The function **assoc_or** takes a vector of elements as inputs and implements a tree of OR gates to reduce the input into a single result. The format of the input is $< x_{i-1} \ldots x_0 >$. The format of the output is $< y >$.

```
DEFINE assoc_or
  IF (length = 1)
  THEN id
  ELSE
    org @
    &assoc_or @
    split
  ENDIF
END
```

The layout produced from the FP expression is shown in Figure 3.21 The hand design is shown in Figure 3.22 A comparison of the designs for the encoder is given in Table 3.2.3.

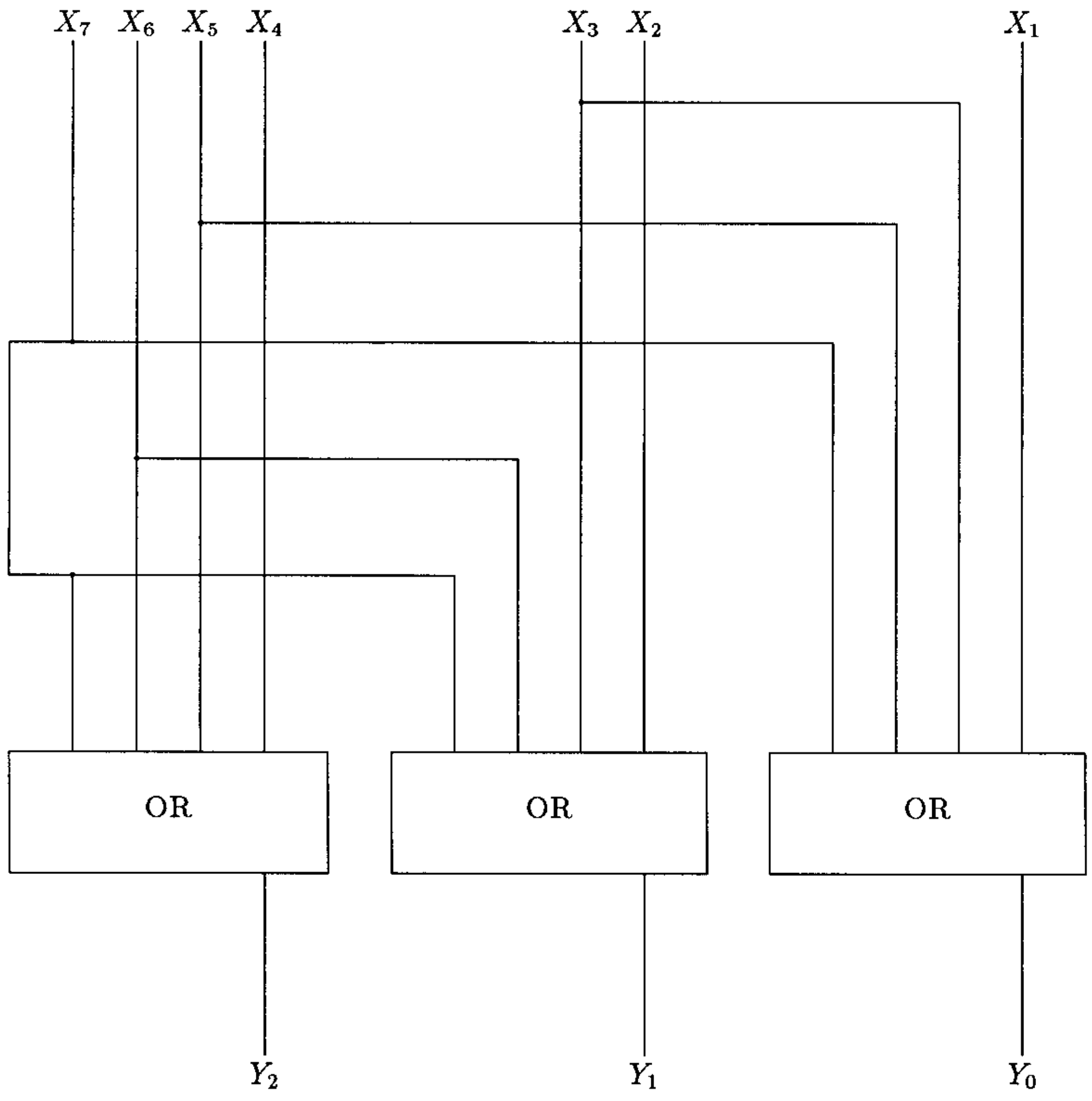


Figure 3.20: 8-Bit Encoder Logic Diagram

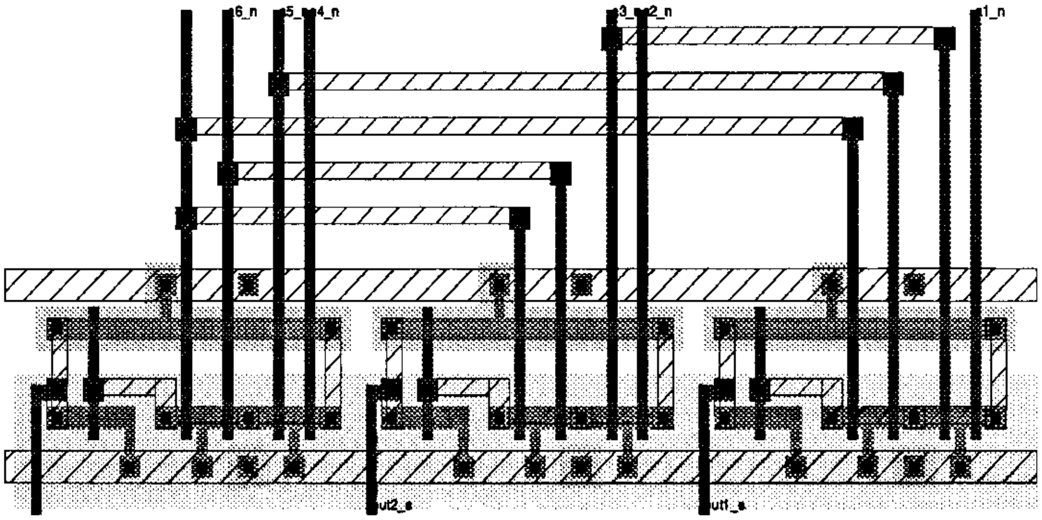


Figure 3.21: 8-Bit Encoder: FLAG layout

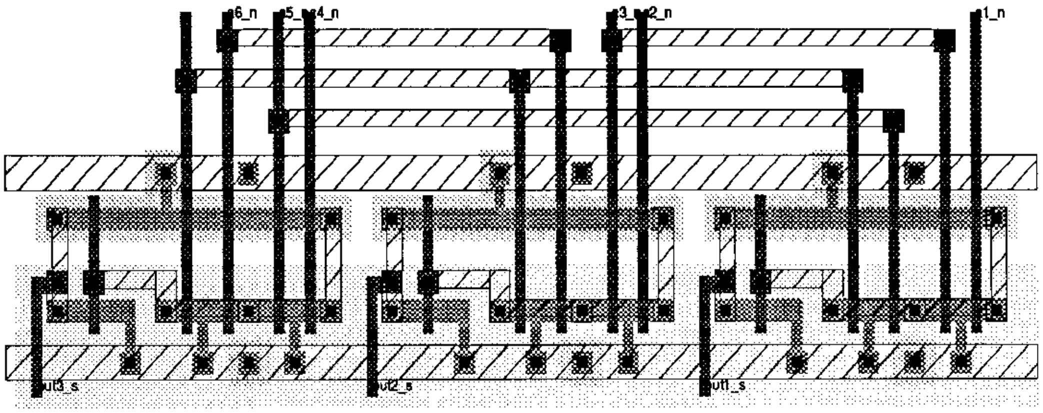


Figure 3.22: 8-Bit Encoder: Hand layout

Encoder	**FLAG**	**Hand**
Area	201×91	201×67
White Space	57%	55%
Average Delay	11.4 ns	11.3 ns
Maximum Delay	20 ns	20 ns
Design Time	30 minutes	2 hours

Table 3.7: Encoder Comparison

3.2.4 Ripple Carry Adder

A ripple carry adder is an iterative network of full adders that performs the addition of two n-bit numbers, $\underline{a} = (a_{n-1}, \ldots, a_0)$ & $\underline{b} = (b_{n-1}, \ldots, b_0)$. The i^{th} full adder has inputs: a_i, b_i, c_i(carry-in) and produces as output: s_i, c_{i+1}(carry-out). It implements the following arithmetic expression:[6]

$$s_i \ + \ 2c_{i+1} \ = \ a_i \ + \ b_i \ + \ c_i$$

Which leads to the following solution:

$$s_i \ = \ (a_i + b_i + c_i) \ mod \ 2$$

$$c_{i+1} = \lfloor (a_i + b_i + c_i)/2 \rfloor$$

A logic diagram for the design of a typical ripple carry adder is shown in Figure 3.23.

The function **rca** takes three inputs, two vectors and a carry-in and returns the sum of the inputs. The format of the input is $<< X_{n-1} \ldots X_0 >< Y_{n-1} \ldots Y_0 > \ \ C_{in} >$. The format of the output is $< C_n \ S_{n-1} \ldots S_0 >$. The function **rca** can be broken into the following steps.

1 Form bit pairs from the two input vectors

2 Add carry-in to least significant bit pair

3 Perform a sequential add upon the bit pairs

```
DEFINE rca
  seq(newfa) @
  apndr @
  [trans@[1,2],3]
END
```

The function **newfa** takes two inputs, a vector of two inputs and a carry-in and returns the sum of the input. The format of the input is $<< a \ b > \ \ c_{in} >$. The format of the output is $< Carry \ Sum >$.

[6]Miloš D. Ercegovac, Tomas Lang, **Digital Systems and Hardware/Firmware Algorithms** (New York: John Wile & Sons, 1985), pp.239

```
DEFINE newfa
  [ org@[1,2],3 ] @
  concat @
  [ [1], ha@[2,3] ] @
  concat @
  [ ha@1, [2] ]
END
```

The function **ha** takes a vector of two elements as input and returns their sum. The format of the input is $< a\ b >$. The format of the output is $< Carry\ Sum >$.

```
DEFINE ha
  [ andg, xorg ]
END
```

The layout produced from the FP expression is shown in Figure 3.24. The hand design is shown in Figure 3.25. A comparison of the designs for the ripple carry adder is given in Table 3.2.4.

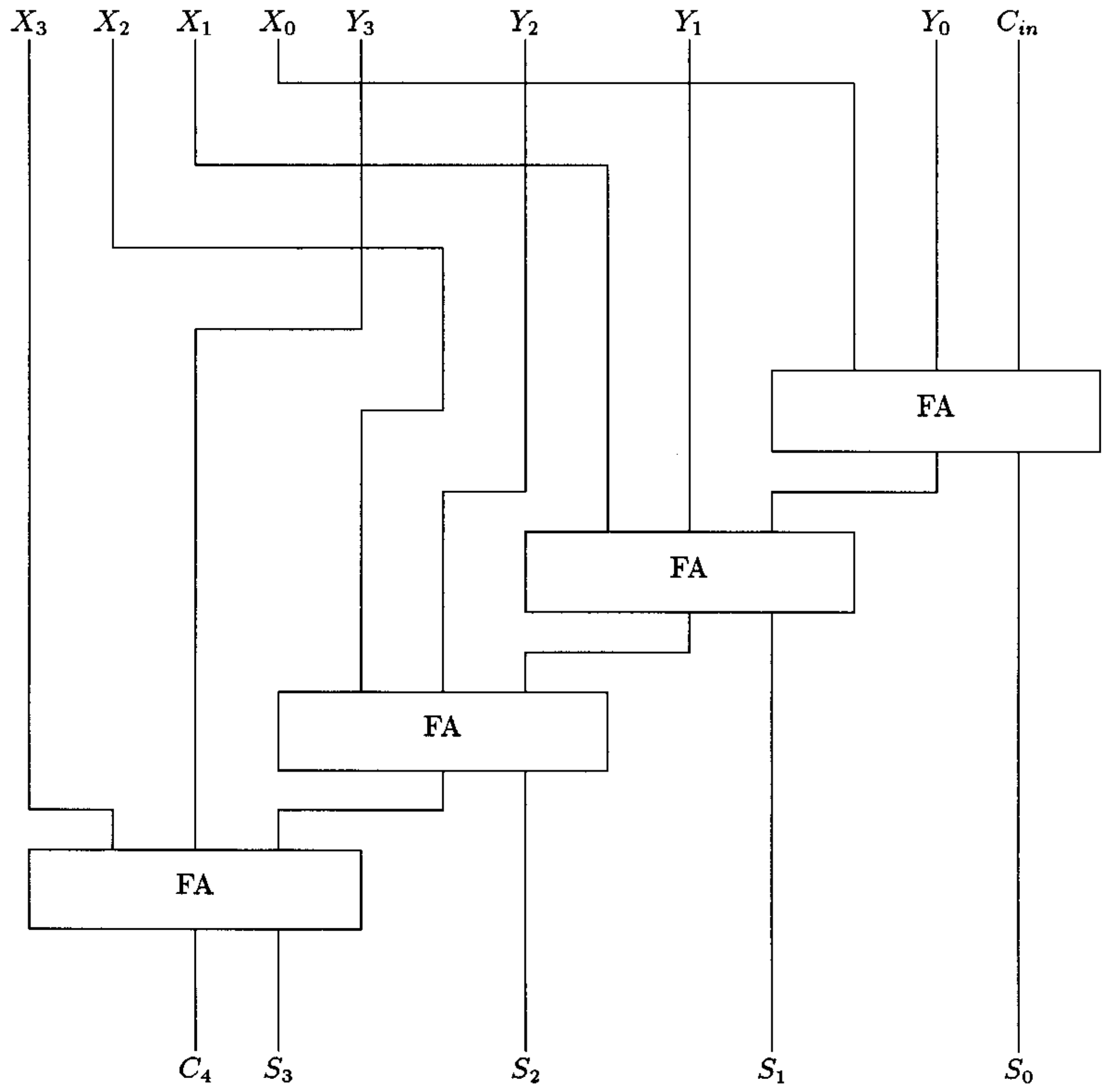


Figure 3.23: 4-Bit Ripple Carry Adder Logic Diagram

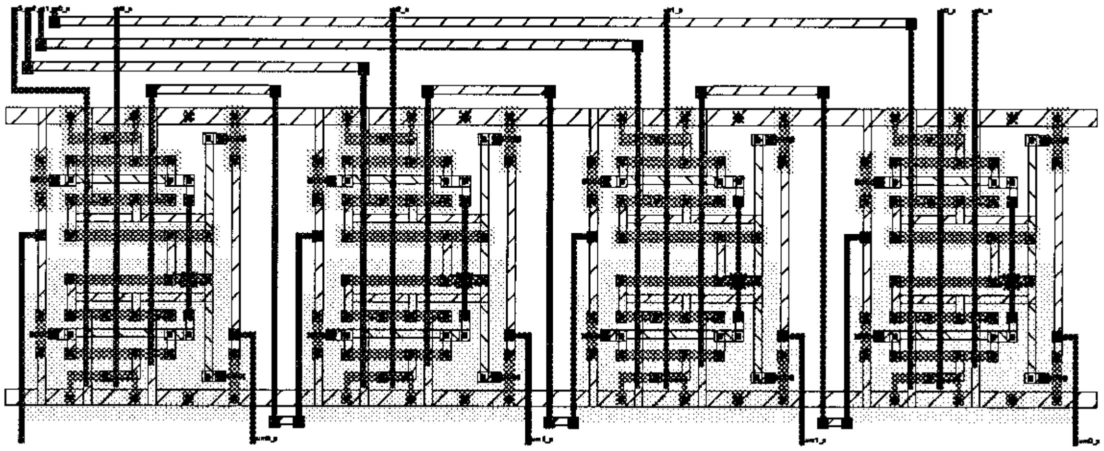


Figure 3.24: 4-Bit Ripple Carry Adder: FLAG layout

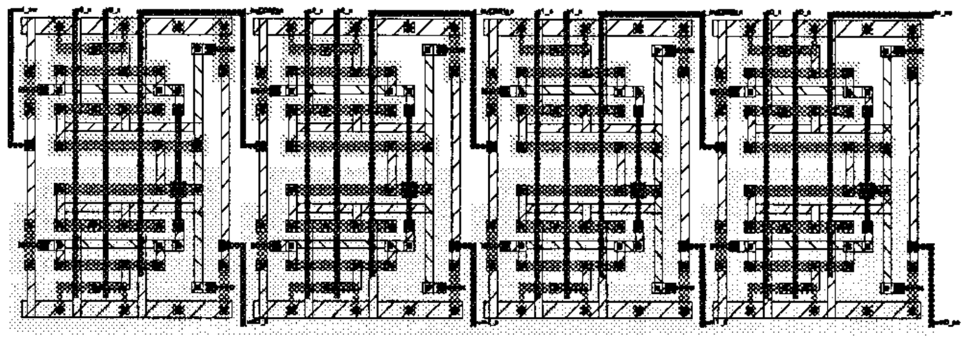


Figure 3.25: 4-Bit Ripple Carry Adder: Hand layout

Ripple Carry Adder	**FLAG**	**Hand**
Area	409×157	346×114
White Space	70%	58%
Average Delay	39.38 ns	38.12 ns
Maximum Delay	70 ns	70 ns
Design Time	15 minutes	$1\frac{1}{2}$ hours

Table 3.8: Ripple Carry Adder Comparison

3.2.5 Concluding Remarks

With the exception of the full adder, the FLAG generated layouts compared pretty much as expected with the hand produced layouts. The FLAG circuits were a little larger and a bit slower than their hand counterparts but nothing unexpected occurred. The reason for the poor performance of the FLAG version of the full adder against its hand counterpart is due to the designer. At the expense of design time, the designer was able to pack the circuit elements for the full adder much closer together than FLAG could. This circuit packing by the designer resulted in a layout that was much smaller and a little faster than the FLAG version but at the expense of design time.

The carry-out lines of the ripple carry adder illustrate a drawback of FLAG produced circuits. Ideally, the carry-out lines between the adders should be short and direct as in the hand layout of the ripple carry adder. Due to the following constraints, imposed by FLAG, the carry-out lines end up being long and windy.

- Data must flow in a vertical direction
- Inputs appear at the top of modules
- Outputs appear at the bottom of modules

These constraints are some of the reasons why FLAG generated circuits end up being larger than their hand counterparts.

FLAG offers an advantage in the design of "scalable" circuits like the multiplexor, encoder and ripple carry adder. Small or simple versions of the circuit can be created and tested; if acceptable, the circuit can be "scaled" to any arbitrary size without requiring any additional work from the designer. By hand, the "scaling" of these kind of circuits cannot be performed with the same ease as is possible with FLAG.

3.3 LSI Modules

In this section, two LSI-level modules, a conditional sum adder and a carry-save multiplier are designed. The resulting designs are discussed in Sec. 3.3.3.

3.3.1 Conditional Sum Adder

A conditional adder is a fast adder which reduces the carry propagation problem by generating distant carries and using these carries to select the true sum outputs from two simultaneously generated provisional sums under different carry input conditions[7].

An example of the conditional sum algorithm is given in Table 3.9[8]. In the example, *addend*, *augend*, and *true* bits are designated by A, B, and S respectively. *Carries* are indicated by C. Subscripts indicate the bit positions and superscripts "1" and "0" refer to the assumption that there was a carry or no carry into the lowest-order bit position of a section. The absence of a superscript indicates a true sum or a true carry. Table 3.9 describes the concept of conditional-sum addition. $S^0(k)$ and $S^1(k)$ denote two provisional sums, each consisting of multiple sections with k addend/augend columns per section. There are $\lceil n/k \rceil$ sections in $S^0(k)$ or $S^0(k)$ for n-bit addition. Simultaneous additions are performed on all sections independently. Let $C^0(k)$ and $C^0(k)$ be the provisional carry sequences formed by the carries out of all the sections in $S^0(k)$ and $S^0(k)$, respectively.

The addition process is completed in t steps, where the integer

$$t = \lceil \, log_2 \, n \, \rceil$$

At the ith step, $S^0(k)$ and $S^1(k)$ are formed with section size

$$k = 2^{i-1}$$

The grouping of the summand columns into disjoint sections starts from the lowest-order right end to the left end. When n is not an integer power of 2, the leftmost section may have less than k columns. The successive section-carry outputs are used to select the true sum outputs.

In Table 3.9 we have an example for $n = 5$. Therefore, ($t = \lceil \, log_2 \, 7 \, \rceil = 3$) steps are required to complete the addition. At step one, each section has only a single column and the five sections in $S^0(1)$ and $S^0(1)$ are formed by bitwise modulo-2 addition with corresponding carry sequences $C^0(1)$ and $C^1(1)$. At step

[7]J. Sklansky, "Conditional Sum Adder Logic," **IRE Transactions on Electronic Computers,** June 1960, pp. 226-231.

[8]K. Hwang, **Computer Arithmetic: Principles, Architecture, and Design** (New York, New York: John Wiley & Sons, 1979), pp. 78-80.

A =	1	0	0	1	1
B =	0	1	1	0	1
$S_i^0(1)$	1	1	1	1	0
$C_{i+1}^0(1)$	0	0	0	0	1
$S_i^1(1)$	0	0	0	0	
$C_{i+1}^1(1)$	1	1	1	1	
$S_i^0(2)$	1	1	1	0	0
$C_{i+1}^0(2)$	0	0		1	
$S_i^1(2)$	0	0	0		
$C_{i+1}^1(2)$	1	1			
$S_i^0(4)$	1	0	0	0	0
$C_{i+1}^0(4)$	0	1			
$S_i^1(4)$	0				
$C_{i+1}^1(4)$	1				
Sum	0	0	0	0	0
Carry	1				

Table 3.9: Example of addition by a conditional sum adder

two, each section contains two columns of addend.augend bits and three sections are formed in $S^0(2)$ and $S^1(2)$ each by 2-bit addition with the carry out forming the 4-bit provisional carry sequences $C^0(2)$ and $C^1(2)$. This process continues in a similar fashion except the section size doubles for each additional step. The final step reveals the true sum S and true carry out C as the desired output.

A logic diagram for a four-bit conditional sum adder is shown in Figure 3.26. Note that the multiplexors used in the implementation of the conditional adder are not the pass-gate variety discussed in the previous section. The pass-gate multiplexors cannot be used in multiplexor trees since the pass-gates degrade the signal passing through them and require restoring logic to return the signal to its full strength. The logic-gate implementation of a multiplexor is used instead of the pass-gate implementation due to the problem of signal degradation. The implementation can be partitioned into 3 stages:

1. Transform input vectors into bit pairs.

2. Generate two provisional sums for each bit pair.

3. Recursively pair together (l)-bit provisional sums to generate ($l+1$)-bit provisional sum until the final result is obtained.
   Example:
   $$\text{If}\left\{\begin{array}{l} PSum_i =<< c^1_{i+1}s^1_i >< c^0_{i+1}s^0_i >> \\ PSum_{i+1} =<< c^1_{i+2}s^1_{i+1} >< c^0_{i+2}s^0_{i+1} >> \end{array}\right.$$
   Then $\quad PSumNew_i =<< PSumNew^1 >< PSumNew^0 >>$ where

$$\begin{array}{ll} < PSumNew^1 >=< c^1_{i+2}\ s^1_{i+1}\ s^1_i > & \text{if } c^1_{i+1} = 1 \\ < PSumNew^1 >=< c^0_{i+2}\ s^0_{i+1}\ s^1_i > & \text{if } c^1_{i+1} = 0 \\ \\ < PSumNew^0 >=< c^1_{i+2}\ s^1_{i+1}\ s^0_i > & \text{if } c^0_{i+1} = 1 \\ < PSumNew^0 >=< c^0_{i+2}\ s^0_{i+1}\ s^0_i > & \text{if } c^0_{i+1} = 0 \end{array}$$

The function **cond_sum_adder** takes as input, two vectors of arbitrary length and a carry-in and returns the sum of the input. The addition of the inputs is performed according to the conditional sum algorithm presented previously. The format of the input is $<< X_{N-1} \ldots X_0 > < Y_{N-1} \ldots Y_0 > \ C_{in} >$. The format of the output is $< C_N\ S_{N-1} \ldots S_0 >$. The function **cond_sum_adder** performs the addition in the following steps.

1 Arrange the input into bit pairs

2 Generate provisional sums for each bit pair

3 Recursively combine provisional sums until the final result is obtained

```
DEFINE cond_sum_adder
  rec_comb   @
  form_sums  @
  msetup
END
```

The function **msetup** takes as input two vectors of arbitrary length and a carry-in and rearranges the input into bit pairs. The format of the input is $<< X_{N-1} \ldots X_0 > < Y_{N-1} \ldots Y_0 > \ C_{in} >$. The format of the output is $<<< X_{N-1}\ Y_{N-1} >< X_{N-2}\ Y_{N-2} > \ldots >< X_0\ Y_0\ C_{in} >>$. The function **msetup** first combines the two input vectors into bit pairs and then appends the carry-in to the least most significant bit pair.

```
DEFINE msetup
  [tlr@1,apndr@[last@1,2]] @
  [trans@[1,2],3]
END
```

The function **form_sums** takes a set of bit pairs as input and generates provisional sums for all bit pairs. For the least significant bit pair a true sum is generated instead of a provisional sum. The format of the input is $<<< X_{N-1}\ Y_{N-1} >< X_{N-2}\ Y_{N-2} > \ldots >< X_0\ Y_0\ C_{in} >>$. The format of the output is $<<< C^1_N\ S^1_{N-1} > < C^0_N\ S^0_{N-1} >> << C^1\ S_0 >>>$.

```
DEFINE form_sums
  apndr @
  [&cha@1, [newfa@2]]
END
```

The definition of the function **newfa** is given on page 69.

The function **cha** takes a bit pair as input and generates both provisional sums for the input ($C_{in}\ \epsilon\{0,1\}$). The format of the input is $< X_i\ Y_i >$. The format of the output is $<< C^1_{i+1}\ S^1_i >< C^0_{i+1}\ S^0_i >>$.

```
DEFINE cha
  [ [org@[1,2], nxor@[1,2]], ha@[1,2] ]
END
```

The definition of the function **ha** is given on page 70.

The function **nxor** takes a vector of two inputs and generates $\overline{(a \oplus b)}$. The format of the input is $< a\ b >$. The format of the output is $< c >$

```
DEFINE nxor
  norg @
  [ norg@[1,2], norg@[2,3] ] @
  [1, norg, 2]
END
```

The function **rec_comb** takes as input a collection of provisional sums and recursively combines them until a true sum is obtained and returned. The format of the input is $<<< C^1_N\ S^1_{N-1} > < C^0_N\ S^0_{N-1} >> \ldots\ << C^1\ S_0 >>>$. The format of the output is $< C_N \; S_{N-1} \ldots S_0 >$. The operation of **rec_comb** can be broken down into the following steps.

```
If (length of input = 1)
    1 Return the true sum
Else
    1 Pair off and combine provisional sums
    2 Recursively combine provisional sums together until true sum is obtained.
```

```
DEFINE rec_comb
  IF (length of input = 1)
  THEN
    id
  ELSE
    rec_comb @
    group
  ENDIF
END
```

The function **group** takes a collection of provisional sums as input and pairs them off from the right (least most significant). Each of these pairs is combined together as described earlier and returned as the output. The format of the input is $<<< C_N^1 \; S_{N-1}^1 > < C_N^0 \; S_{N-1}^0 >> \ldots \; << C_1 \; S_0 >>>$. The format of the output is $<<< C_N^1 \; S_{N-1}^1 \; S_{N-2}^1 > < C_N^0 \; S_{N-1}^0 \; S_{N-2}^1 >> \ldots \; << C_1 \; S_0 >>>$.

```
DEFINE group
  IF (length of input is odd )
  THEN
    apndl @ [first, &reduce@pair@tl]
  ELSE
    &reduce@pair
  ENDIF
END
```

The function **reduce** takes two groups of provisional sums and combines them together to form an expanded conditional sum. The format of the input is $<<< C_N^1\ S_{N-1}^1 > < C_N^0\ S_{N-1}^0 >> << C_{N-1}^1\ S_{N-2}^1 > < C_{N-1}^0\ S_{N-2}^0 >>>$. The format of the output is $<< C_N^1\ S_{N-1}^1\ S_{N-2}^1 >><< C_N^0\ {}_{N-1}^0\ S_{N-2}^0 >>$. The operation of **reduce** can be broken down into the following steps.

1 Distribute the provisional sums for the $(N-1)^{th}$ position to each provisional sum for the $(N-2)^{th}$ position.

2 Create expanded provisional sum for the $(N-1)$ to the $(N-2)$ bit positions.

```
DEFINE reduce
  &select @
  &apndr @ distl
END
```

The function **select** takes a vector containing two provisional sums and an intermediate sum. It returns an expanded provisional sum as the output as previously described. The format of the input is $<<< C_N^1\ S_{N-1}^1 >< C_N^0\ S_{N-1}^0 >>< C_{N-1}^i\ S_{N-2}^i >>$, $i \in \{0,1\}$. The format of the output is $< C_N^i\ S_{N-1}^i\ S_{N-2}^i >$.

```
DEFINE select
  concat @ [ mux@[ [first@last],tlr ], tl@last ]
END
```

The definition of the function **mux** is given on page 56.

The layout produced from the FP expression is shown in Figure 3.27. The hand layout is shown in Figure 3.28. A comparison of the designs for the conditional sum adder is given in Table 3.10.

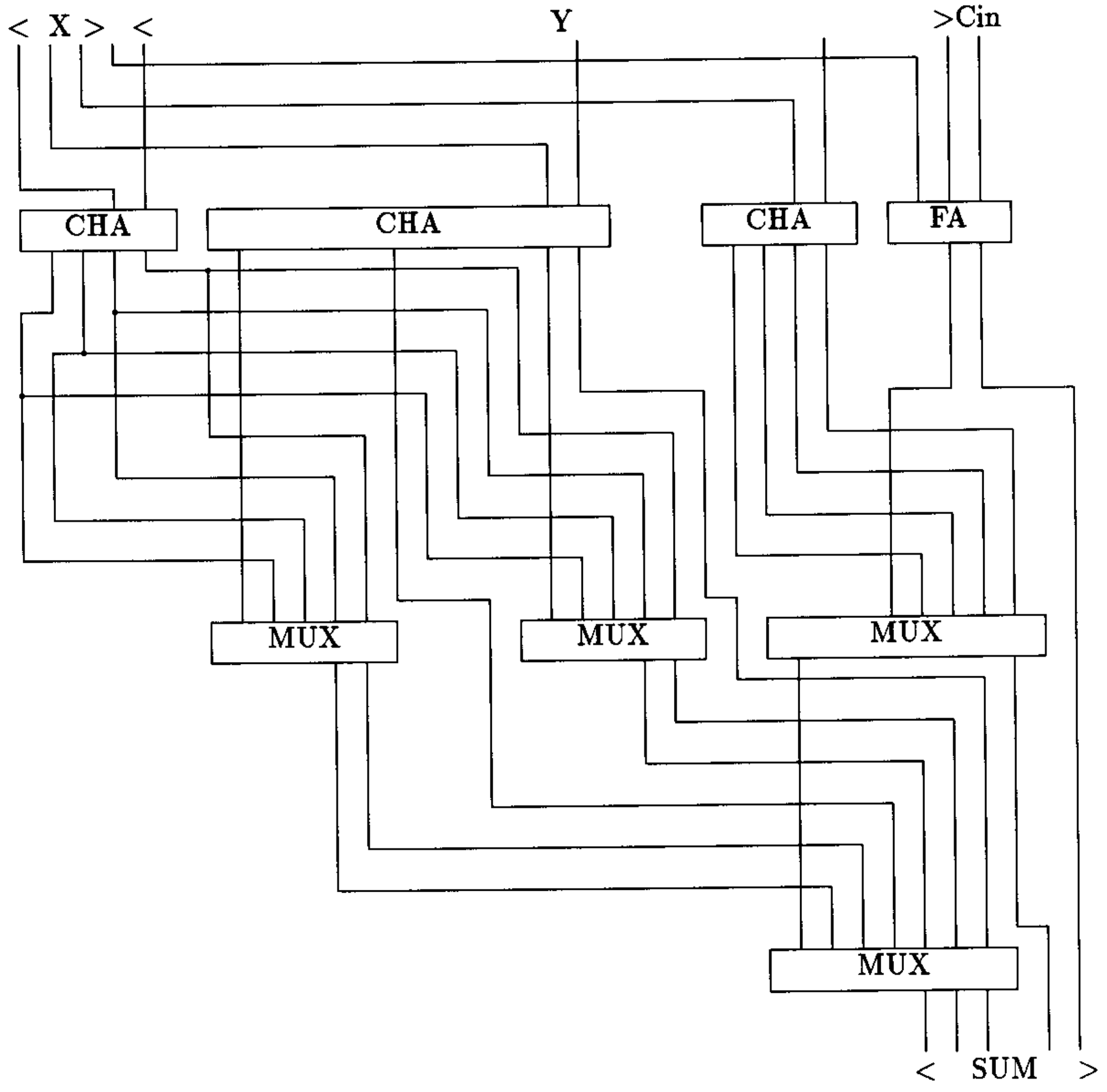


Figure 3.26: 4-bit Conditional Adder Logic Diagram

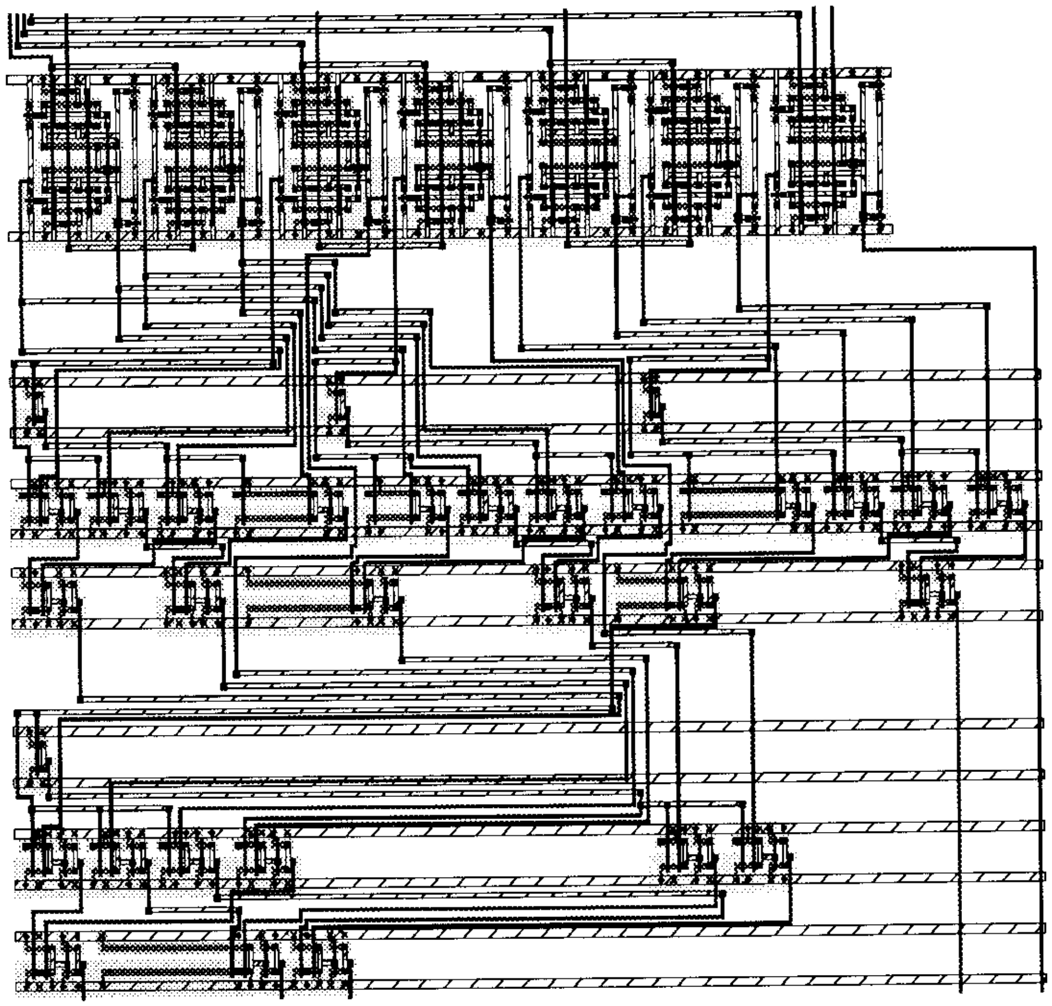


Figure 3.27: 4-bit Conditional Sum Adder: FLAG layout

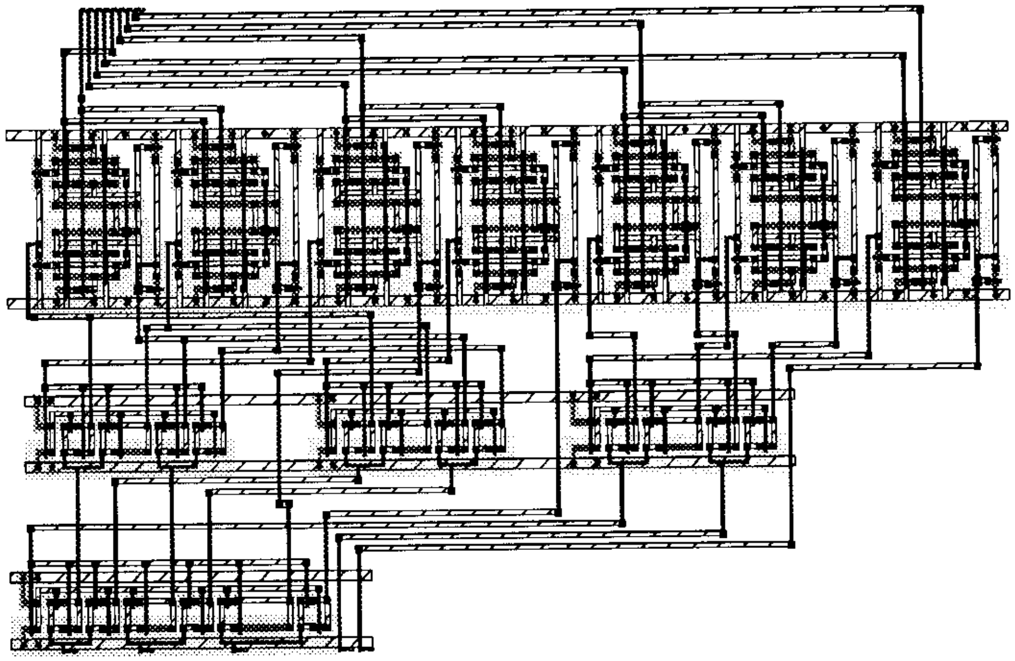


Figure 3.28: 4-bit Conditional Sum Adder: Hand layout

Conditional Sum Adder	FLAG	Hand
Area	738 × 639	649 × 390
White Space	54%	53%
Average Delay	36.6 ns	25.9 ns
Maximum Delay	63 ns	45 ns
Design Time	5 hours	15 hours

Table 3.10: Conditional Sum Adder Comparison

3.3.2 Carry Save Multiplier

A carry-save multiplier performs the multiplication of two bit vectors through the use of a linear array of carry-save adders. A carry-save adder saves the carry propagation for multiple additions until all additions have been completed and then takes a number of stages to complete the carry propagation. A high-level description for the multiplier is:

$$\underline{X} = < x_{N-1} \dots x_0 >$$
$$\underline{Y} = < y_{M-1} \dots y_0 >$$
$$\sum_{i=0}^{N-1} x_i 2^i \times \underline{Y}$$

The operation of the carry-save multiplier can be broken into 5 stages.

1. Generate $x_0 \times \underline{Y}$
2. Generate $x_1 \times \underline{Y}$
3. Generate $x_2 \times \underline{Y}$ and reduce the three partial products generated into 2 vectors, carry and sum
4. For each remaining bit in $\underline{X}$, generate $x_i \times \underline{Y}$ and incorporate it into the carry and sum vectors already generated.
5. Add together the two vectors, carry and sum, to generate a single result.

A logic diagram for the FLAG design of a carry save multiplier is given in Figure 3.29.

The function **mult**[9] takes two vectors of arbitrary length as input and returns the product of the input as the result. The format of the input is $<< x_{m-1}\ x_{m-2} \dots x_0 >< y_{n-1}\ y_{m-2} \dots y_0 >>$, $m > 3\ \&\ n > 2$. The format of the output is $< p_{m+n-1}\ p_{m+n-2} \dots p_0 >$. The **mult** calls the function **csmult** to generate the partial products and combine them into two vectors. The function **finaladd** to generate the final product.

```
DEFINE mult
  finaladd @
```

[9]Schlag, "Extracting Geometry from FP," pp. 52-56.

```
  csmult
END
```

The function **csmult** takes two vectors as input and generates the partial products from the input. The partial products are reduced to two vectors through the use of a carry-save adder network. The function **csmult** can be broken down into the following steps.

1 Generate $x_0 \times \underline{Y}$

2 Generate $x_1 \times \underline{Y}$

3 Generate $x_2 \times \underline{Y}$ and perform a carry-save addition upon the partial products in order to reduce them into a partial sum.

4 For each remaining x_i, generate $x_i \times \underline{Y}$ and perform a carry-save addition to incorporate the partial products into the partial sum.

```
DEFINE csmult
  ckstage @
  stage3 @
  stage2 @
  stage1
END
```

The function **stage1** generates $x_0 \times \underline{Y}$.

```
DEFINE stage1
  [ tlr@1,
    concat@&[1,andg]@pair@concat@[ [1@2],
                                   concat@distl@[last@1,tl@2],
                                   [last@1]
                                 ]
  ]
END
```

The function **stage2** generates $x_1 \times \underline{Y}$.

```
DEFINE stage2
  [ 1,
    concat@[ [1,[andg]]@1@2, concat@&[1,[andg@[1,2],3]]@2@2 ],
    3
  ]@
  [ 1, [1@2,&[2@2,1,1@2]@2@2], 3 ] @
  [ tlr@1,
    [ [1@2,last@1], distl@[last@1,pair@tlr@tl@2] ],
    [last@2]
  ]
END
```

The function **stage3** generates $x_2 \times \underline{Y}$ and adds the existing partial products together using a carry-save adder approach.

```
DEFINE stage3
  regroup @
  csave1  @
  msetup
END
```

The function **msetup** merely rearranges the signals in order to generate the next wave of partial products and to perform the carry save addition.

```
DEFINE msetup
  [ tlr@1, [ [1@2,last@1],
    &[1@2,[1,2@2]]@distl@[last@1,pair@tlr@tl@2] ],
    apndl@[last@2,3]
  ]
END
```

The function **csave1** generates the partial products and combines them together into two vectors.

```
DEFINE csave1
  [ 1,
    concat@[ [1,andg]@1, [op1@1@2], &op2@tl@2 ]@2,
    apndl@[ha@1,tl]@3
  ]
END
```

The definition of the function **ha** is given on page 70.

The function **op2** generates a single partial product and performs the addition of three inputs.

```
DEFINE op2
  [ [ org@[1,1@2], 3 ], 2@2 ] @
  [ 1@1, ha@[2@1,2], 3 ] @
  [ ha@1, andg@2, 2@2 ]
END
```

The definition of the function **ha** is given on page 70.

The function **op1** generates a single partial product and performs the addition of two inputs.

```
DEFINE op1
  [ [1@1,2], 2@1 ]@
  [ ha@[1@1,andg@2], 2@2 ]
END
```

The definition of the function **ha** is given on page 70.

The functions **regroup** and **regp** merely rearranges the signals for the next stage.

```
DEFINE regroup
  [ 1, apndr@[ tlr@2, [last@2,1@1@3] ], apndl@[2@1@3, tl@3] ] @
  [ 1, regp@2, 3 ]
END


DEFINE regp
  IF (length of input = 2)
  THEN id;
  ELSE
    concat@[ [1,[2,1@1@3]], regp@concat@[[2@1@3, 2@3], tl@tl@tl] ]
  ENDIF
END
```

The function **ckstage** recursively generates all remaining partial products and adding them into the partial sum.

```
DEFINE ckstage
  IF (length@1 = 1)
  THEN laststage;
  ELSE
    ckstage@normalstage
  ENDIF
END
```

The function **normalstage** generates $x_i \times \underline{Y}$ and adds the existing partial products together using a carry-save adder approach.

```
DEFINE normalstage
  regroup @ csave @ msetup
END
```

The function **csave** generates the partial products and combines them together into two vectors.

```
DEFINE csave
  [ 1, concat@[[1,andg]@1,&op2@2]@2, apndl@[ha@1,tl]@3 ]
END
```

The definition of the function **ha** is given on page 70.

The function **laststage** generates $x_{n-1} \times \underline{Y}$ and adds the last set of partial products together using a carry-save adder approach.

```
DEFINE laststage
  lastregroup @ lastcsave @ msetup
END
```

The function **lastcsave** generates the partial products and combines them together into two vectors.

```
DEFINE lastcsave
  [ concat@[[andg@1],&lop2@2]@2, apndl@[ha@1,tl]@3 ]
END
```

The definition of the function **ha** is given on page 70.

The function **lop2** generates a single partial product and performs the addition of three inputs.

```
DEFINE lop2
  [org@[1,1@2],2@2] @ [1@1,ha@[2@1,2]] @ [ha@1,andg@2]
END
```

The definition of the function **ha** is given on page 70.

The functions **lastregroup** merely rearranges the signals for the final ripple carry addition stage.

```
DEFINE lastregroup
  concat @
  [ pair@tlr@apndl@[1,concat@tl]@1,
    [ apndl@[ [2@last@1,1@1@2],apndl@[2@1@2,tl@2] ] ]
  ]
END
```

The function **finaladd, cfa, fa, ha** perform the final reduction of the partial sums into the final product.

```
DEFINE finaladd
  concat@[ seq(cfa)@concat@[tlr,[ [1], 2 ]@1@last], tl@last ]
END


DEFINE cfa
  IF (length@1 = 1)
  THEN ha@[1@1,2];
  ELSE
    newfa
  ENDIF
END
```

The definition of the function **newfa** is given on page 69 and the definition of the function **ha** is given on page 70.

The layout produced from the FP expression is shown in Figure 3.30. A logic diagram of the hand design for the carry save multiplier is given in Figure 3.31.

The layout produced from the hand design is shown in Figure 3.32. A comparison of the designs for the carry-save multiplier is given in Table 3.11.

As is evident from Figure 3.32, the hand layout can be compacted further. However, the hand layout is good enough to illustrate the differences between the hand and FLAG layouts. Further compaction of the hand layout would serve to only increase the design time and reduce the area of the layout while having a negligible effect upon the speed of the layout. It will not affect the results of the comparison strongly enough to alter the conclusion.

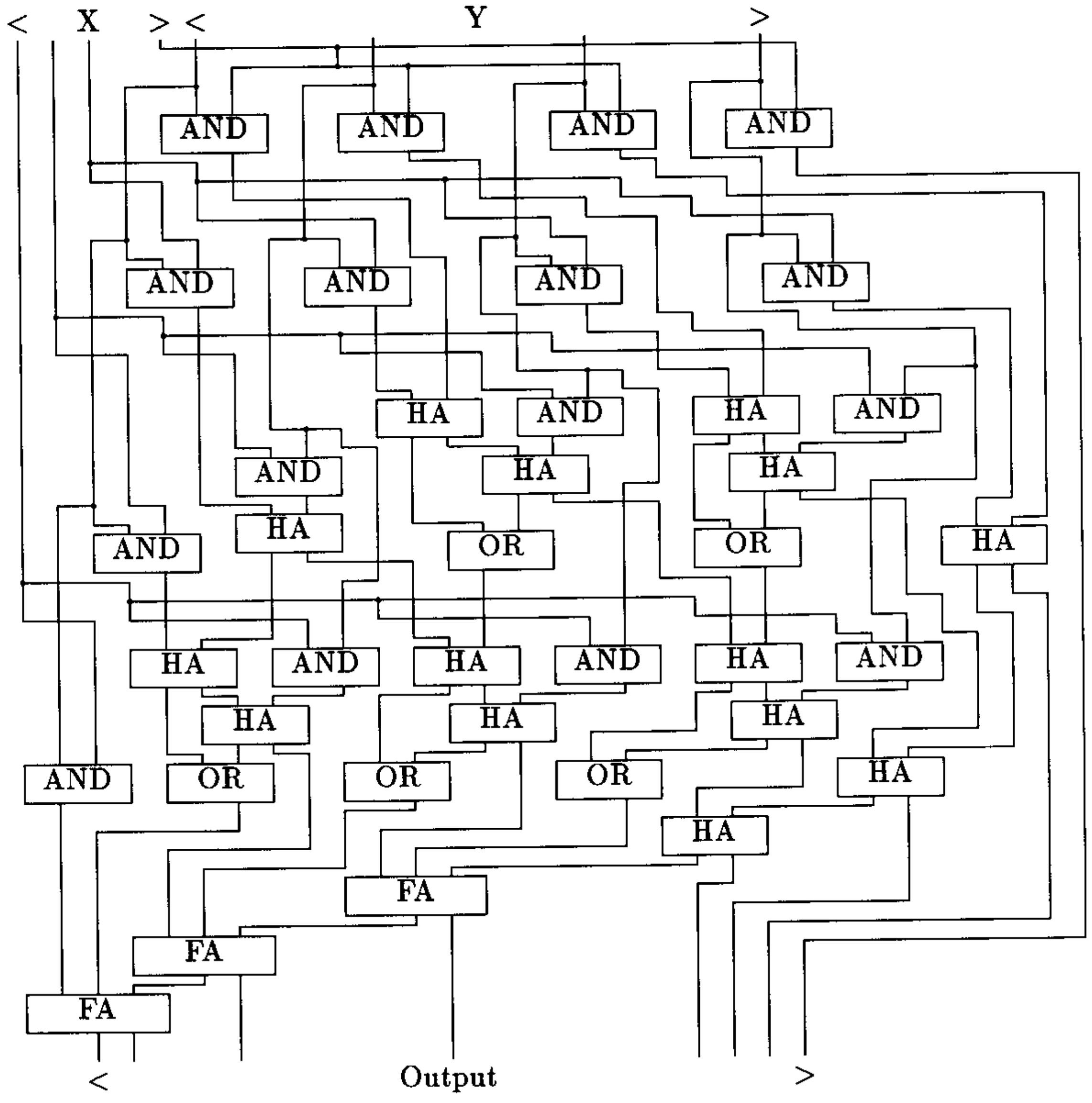


Figure 3.29: A 4 × 4 Carry-Save Multiplier Logic Diagram: FLAG design

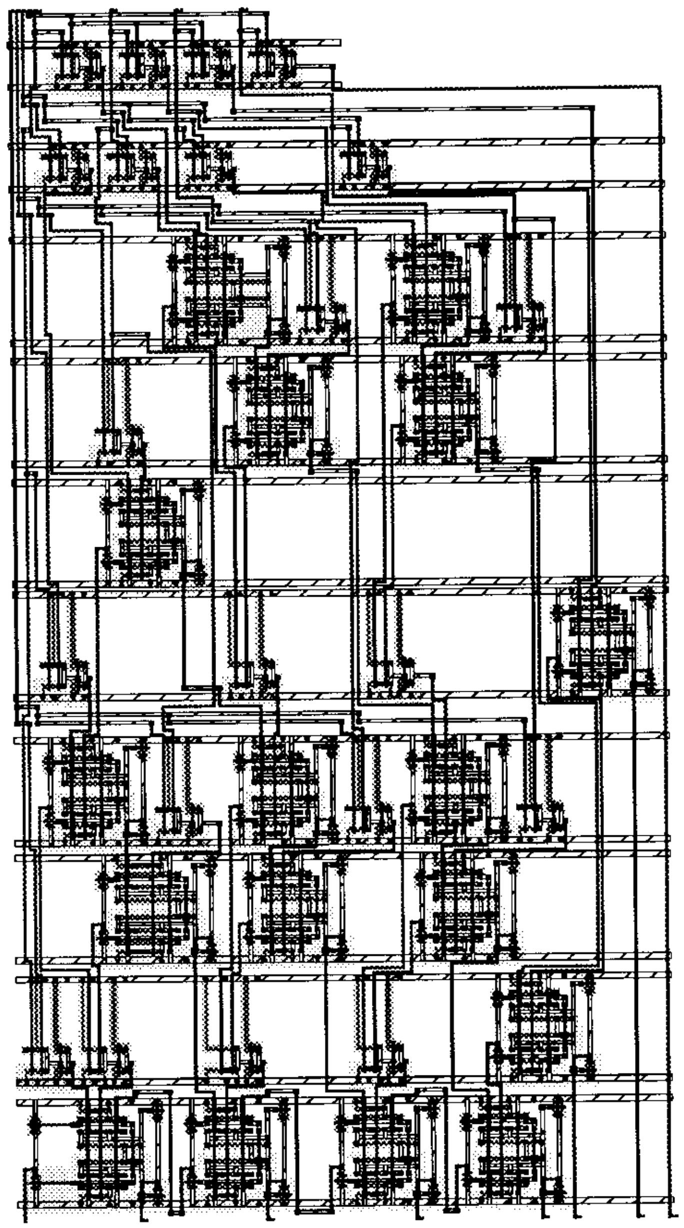


Figure 3.30: A 4 × 4 Carry Save Multiplier: FLAG layout

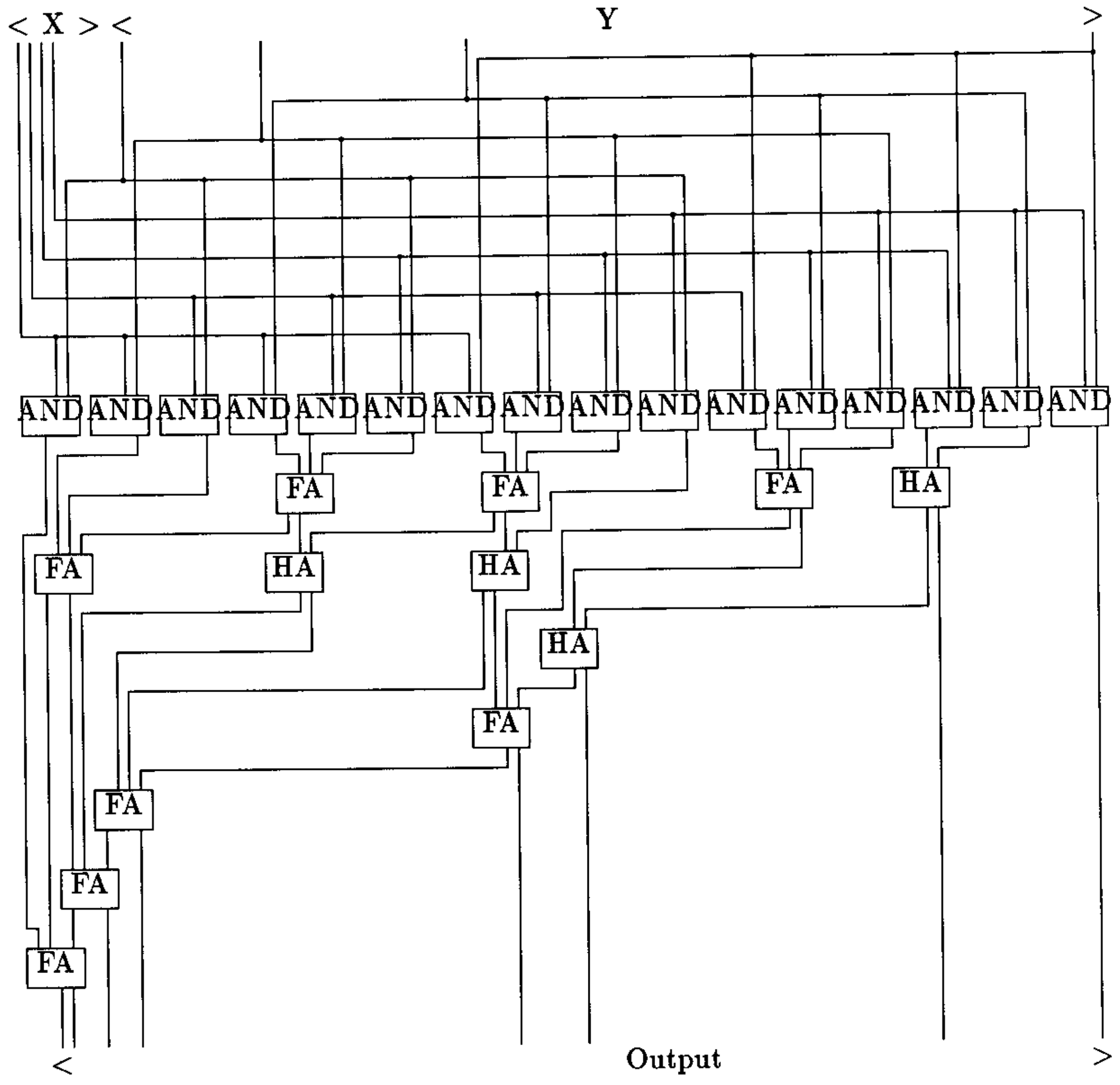


Figure 3.31: A 4 × 4 Carry Save Multiplier Logic Diagram: Hand design

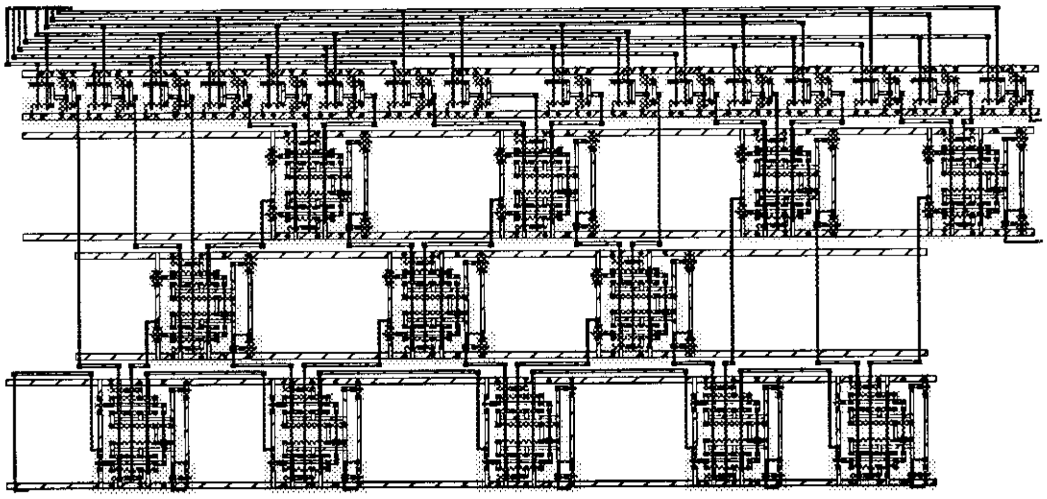


Figure 3.32: A 4 × 4 Carry Save Multiplier: Hand layout

Carry-Save Multiplier	**FLAG**	**Hand**
Area	556 × 1159	880 × 469
White Space	66%	38%
Average Delay	72 ns	50 ns
Maximum Delay	70 ns	70 ns
Design Time	5 hours	15 hours

Table 3.11: Carry-Save Multiplier Comparison

3.3.3 Concluding Remarks

The conditional sum adder and carry save multiplier circuits display FLAG's ability to create circuits at various levels of abstraction. Note that the FLAG design for the conditional sum adder and carry save multiplier make use of the same hand-designed modules that the hand layout does. With FLAG, circuits can be described at a higher level to make use of previously designed modules. In the conditional sum adder, for instance, the conditional half adder (**cha**) and full adder (**newfa**) are described only functionally. There is no need to describe them at a lower level since existing designs for them are to be used in the FLAG design of the conditional sum adder. Overall, these circuits do not compare that unfavorably against their hand-generated counterparts. Sure, the FLAG designs are somewhat larger and slightly slower than the hand versions of these circuits, but in terms of design time, the FLAG circuits required a much smaller amount of time to design as compared to their hand counterparts. Additionally, the FLAG layouts did not need to be tested once the layout had been finished. This is due to the fact that the FLAG layouts are a direct extension of the FP description and if the FP description is correct then the layout will also be correct. This is not the case for the hand layouts, though. The hand layouts must be simulated throughout the design process to insure that the design functions as intended. These circuit simulations were quite slow for these complex logic examples and they did lengthen the design time necessary to complete the hand layouts by a substantial margin. From these examples we can see that the FLAG layouts do not generally end up being as small or as fast as their hand counterparts, but they require much less effort and time to generate than their hand counterparts.

3.4 Composite Circuit

In this section a composite circuit is created. The circuit is an adder based upon the conditional addition algorithm presented in the preceding section. Instead of creating provisional sums for each bit pair of the input, the composite adder splits the input bit-pairs into 4 groups and generates provisional sums for each group with a ripple carry adder. A logic diagram for the composite adder is given in Figure 3.33.

The function **composite_adder** takes as input, two vectors of arbitrary length and returns the sum of the input. The addition of the inputs is performed according to the conditional sum algorithm presented previously. The format of the input is $<< X_{N-1} \ldots X_0 > < Y_{N-1} \ldots Y_0 >>$. The format of the output is $< C_N \; S_{N-1} \ldots S_0 >$. The function **composite_adder** performs the addition in the following steps.

1 Arrange the input into bit pairs and divide into four groups

2 Generate provisional sums for each group

3 Recursively combine provisional sums until the final result is obtained

```
DEFINE composite_adder
  rec_comb   @
  hform_sums  @
  hsetup
END
```

The definition of the function **rec_comb** can be found on page 79.

The function **hsetup** takes as input two vectors of arbitrary length and rearranges the input into bit pairs. The bit pairs are then divided into four groups. The format of the input is $<< X_{N-1} \ldots X_0 > < Y_{N-1} \ldots Y_0 >>$. The format of the output is $<<< X_{N-1}\, Y_{N-1} >< X_{N-2}\, Y_{N-2} > \ldots < X_{N-\lceil\frac{N}{4}\rceil}\, Y_{N-\lceil\frac{N}{4}\rceil} >> << X_{N-(\lceil\frac{N}{4}\rceil+1)}\, Y_{N-(\lceil\frac{N}{4}\rceil+1)} > \ldots < X_{N-2\lceil\frac{N}{4}\rceil}\, Y_{N-2\lceil\frac{N}{4}\rceil} >> \ldots << X_{N-(3\lceil\frac{N}{4}\rceil+1)}\, Y_{N-(3\lceil\frac{N}{4}\rceil+1)} > \ldots < X_0\, Y_0 >>>$. The function **hsetup** first combines the two input vectors into bit pairs and then divides the bit pairs into four groups.

```
DEFINE hsetup
  concat @
  &split@split @
  trans
END
```

The function **hform_sums** takes the four groups of bit pairs as input and generates a provisional sum for each group except for the last group, in which a true sum is generated instead. The format of the input is $<<< X_{N-1}\, Y_{N-1} >< X_{N-2}\, Y_{N-2} > \ldots < X_{N-\lceil\frac{N}{4}\rceil}\, Y_{N-\lceil\frac{N}{4}\rceil} >> << X_{N-(\lceil\frac{N}{4}\rceil+1)}\, Y_{N-(\lceil\frac{N}{4}\rceil+1)} > \ldots < X_{N-2\lceil\frac{N}{4}\rceil}\, Y_{N-2\lceil\frac{N}{4}\rceil} >> \ldots << X_{N-(3\lceil\frac{N}{4}\rceil+1)}\, Y_{N-(3\lceil\frac{N}{4}\rceil+1)} > \ldots < X_0\, Y_0 >>>$. The format of the output is $<<< C_N^1\, S_{N-1}^1 \ldots S_{N-\lceil\frac{N}{4}\rceil}^1 > < C_N^0\, S_{N-1}^0 \ldots S_{N-\lceil\frac{N}{4}\rceil}^0 >> \ldots << C_{N-2\lceil\frac{N}{4}\rceil}^1\, S_{N-2\lceil\frac{N}{4}\rceil-1}^1 \ldots S_{N-3\lceil\frac{N}{4}\rceil}^1 >< C_{N-2\lceil\frac{N}{4}\rceil}^0\, S_{N-2\lceil\frac{N}{4}\rceil-1}^0 \ldots S_{N-3\lceil\frac{N}{4}\rceil}^0 >>< C_{N-3\lceil\frac{N}{4}\rceil}\, S_{N-3\lceil\frac{N}{4}\rceil-1} \ldots S_0 >>$. (If $N = 16$, the format of the output would be: $<<< C_{16}^1\, S_{15}^1\, S_{14}^1\, S_{13}^1\, S_{12}^1 >< C_{16}^0\, S_{15}^0\, S_{14}^0\, S_{13}^0\, S_{12}^0 >> \ldots << C_8^1\, S_7^1\, S_6^1\, S_5^1\, S_4^1 >< C_8^0\, S_7^0\, S_6^0\, S_5^0\, S_4^0 >>< C_4\, S_3\, S_2\, S_1\, S_0 >>$).

```
DEFINE hform_sums
  apndr @
  [ &condrca@tlr, [hrca@last] ]
END
```

The function **hrca** takes as input a group of bit pairs and returns the true sum as the output. The format of the input is $<< X_3\ Y_3 > \ldots < X_0\ Y_0 >>$. The format of the output is $< C_4\ S_3\ S_2\ S_1\ S_0 >$.

```
DEFINE hrca
  apndr @
  [seq(newfa)@apndr@[1,1@2]  , 2@2] @ [tlr, ha@last]
END
```

The definition of the function **newfa** can be found on page 69 and the definition of the function **ha** can be found on page 70.

The function **condrca** takes a group of bit pairs as input and generates the provisional sums as the output. The format of the input is $<< X_i\ Y_i > \ldots < X_{i-3}\ Y_{i-3} >>$. The format of the output is $<< C^1_{i+1}\ S^1_i\ S^1_{i-1}\ S^1_{i-2}\ S^1_{i-3} >< C^0_{i+1}\ S^0_i\ S^0_{i-1}\ S^0_{i-2}\ S^0_{i-3} >>$. The function **condrca** can be broken down into the following steps.

1 Generate provisional sums for least significant bit pair.

2 Distribute rest of input to each provisional sum.

3 Generate provisional sums for the rest of the input.

```
DEFINE condrca
  &( apndr@[ seq(newfa)@apndr@[1,1@2]  , 2@2 ]) @
  distl @
  [ tlr, cha@last ]
END
```

The definition of the function **newfa** can be found on page 69 and the definition of the function **cha** can be found on page 78. The layout for the composite adder is shown in Figure 3.34.

Table 3.12 compares the delays for a ripple carry adder, a conditional adder and the composite adder for two 16-bit operands. From Table 3.12, it is clear that the composite adder outperforms either the RCA or the CA. This example illustrates the ease with which a designer, using FLAG, can design composite circuits that outperform already existing circuits. Very little work is needed to

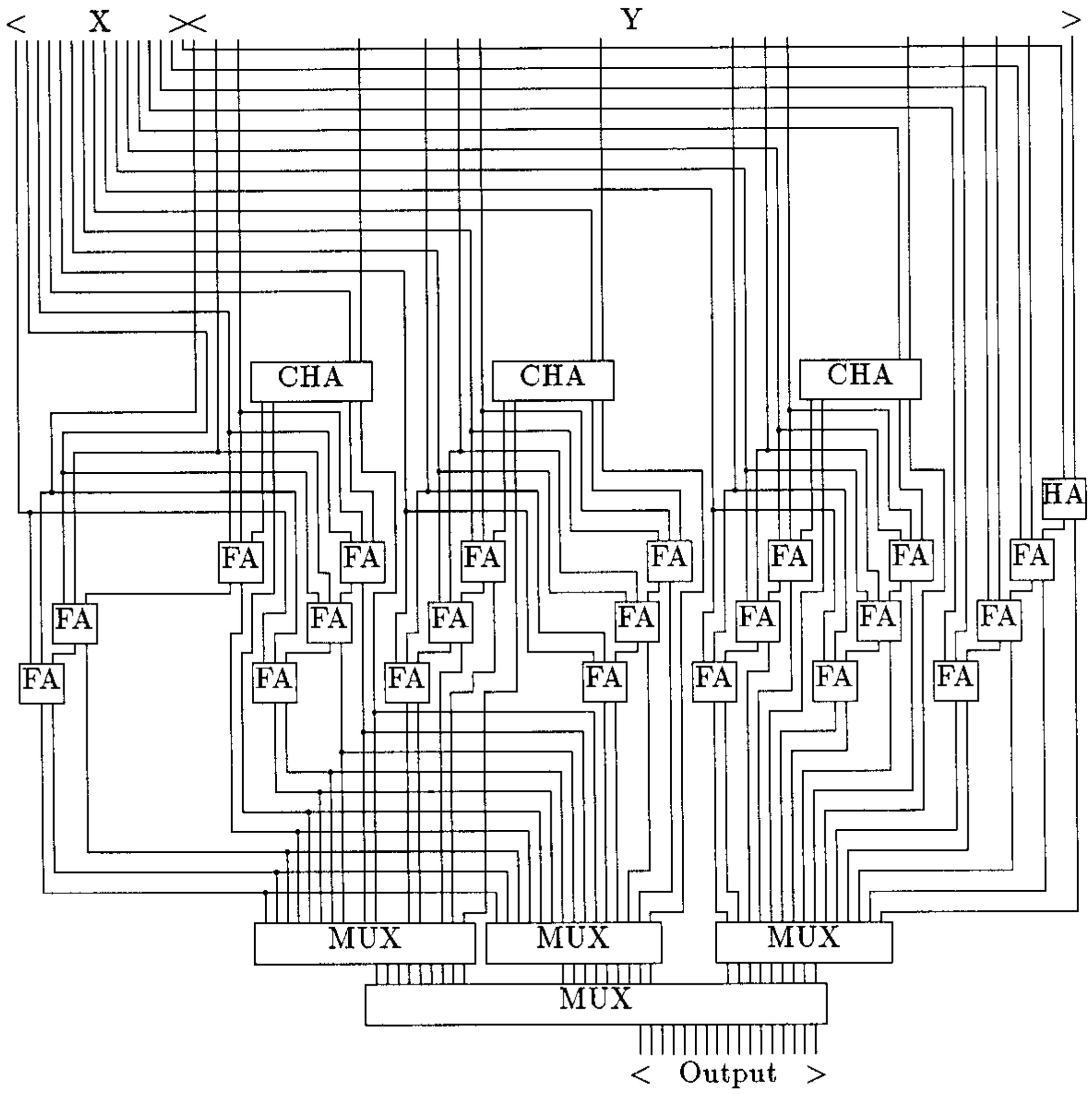


Figure 3.33: 16-bit Composite Adder Logic Diagram

	Avg. Delay	Max. Delay
RCA	75 ns	160 ns
CA	73 ns	145 ns
Composite	65 ns	115 ns

Table 3.12: Adder Delay Comparison

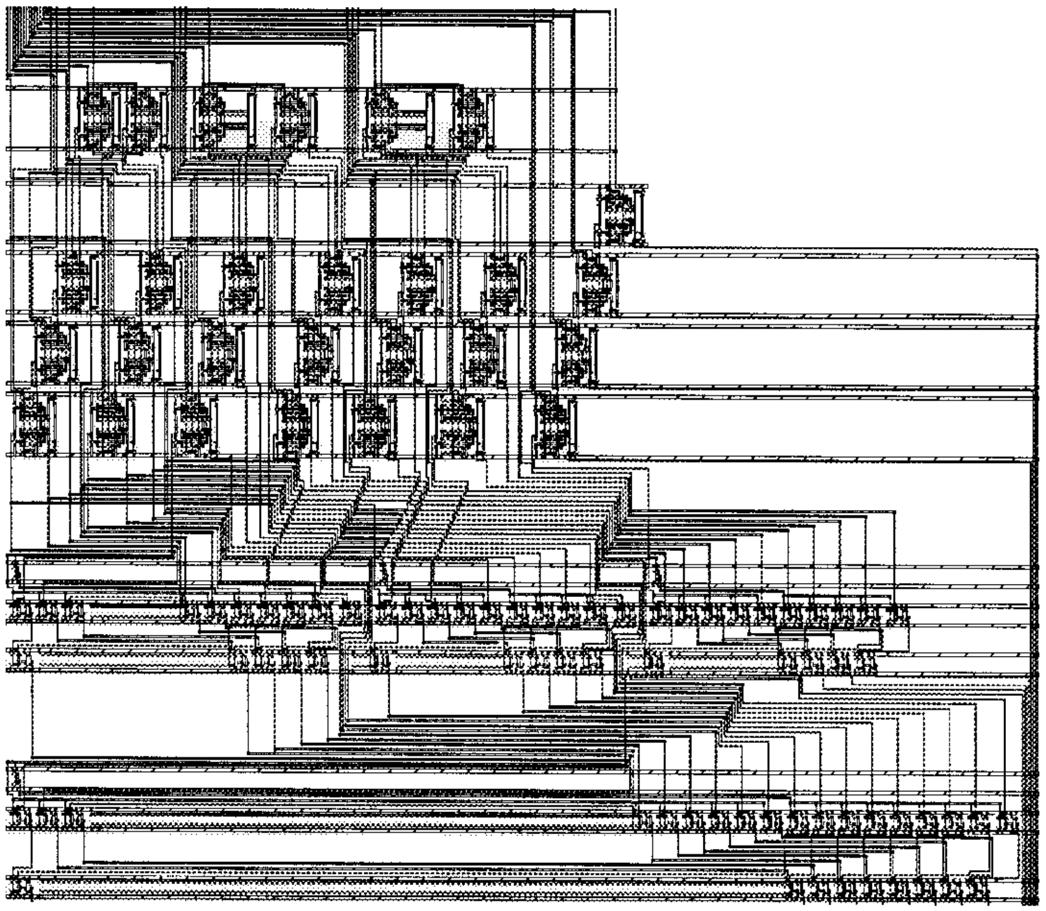


Figure 3.34: 16-bit Composite Adder layout

create the composite circuit. The designer merely needs to modify and manipulate existing circuit descriptions. No additional work from the designer's point of view is required after the circuit descriptions have been created. Without FLAG, a designer would have to start from scratch or nearly so and create the circuit by hand. This would involve a considerable investment of time and effort on the part of the designer and once done can not be altered easily. On the other hand, with FLAG, only a modification of the circuit description is needed to make a small change or generate an entirely different circuit. With FLAG, a designer can play around with different circuits rather easily while this can not be done quite so readily by hand.

Chapter 4

Summary and Conclusions

Typical VLSI design systems take some form of schematic input and convert it into a physical layout. This conversion can be done by hand, which is a rather lengthy and tedious process or it can be automated, in which case the designer loses the ability to control the structure of the physical implementation. In either case, architectural exploration during the design cycle is not feasible. The idea, then, is to modify the design cycle so that architectural exploration can be incorporated as part of the design process. A VLSI layout generator, FLAG, is presented as one possible approach. From a behavioral description of the circuit, created by the designer, FLAG produces a VLSI realization of the circuit with no additional input from the designer. Embedded in the behavioral description of the circuit is the overall topology of the physical layout as dictated by the designer. This allows the designer to retain control over the overall structure of the physical implementation of the circuit. Different circuit designs can be created merely by modifying the behavioral description of the circuit in question. The designer can then examine different solutions and choose the best from among them to be implemented by hand.

In order to examine FLAG, a number of circuits of varying complexity were created in CMOS and compared to hand-produced versions of the same circuits. These test circuits ranged in complexity from simple combinational circuits, like an exclusive-or gate to complex logic networks, like a conditional adder and a carry-save multiplier. For each of the test circuits, a layout was produced by hand and by FLAG. These circuits were then critiqued on the basis of the total area used, % of white space present in the design, overall speed of the circuit,

$\frac{FLAG}{hand}$	**Area**	**White Space**	**Average Delay**	**Maximum Delay**	**Design Time**
XOR	5.17	1.5	1.89	2	$\frac{1}{3}$
AND-OR	1.22	1.01	0.87	0.83	$\frac{1}{14}$
NAND	1.34	1.04	1.11	1	$\frac{1}{14}$
FA	116.1	1.12	1.64	1.36	$\frac{1}{3}$
MUX	2.3	1.08	1.87	2	$\frac{1}{4}$
ENCODER	1.35	1.03	1.01	1	$\frac{1}{4}$
RCA	1.62	1.2	1.03	1	$\frac{1}{6}$
CA	1.86	1.02	1.41	1.4	$\frac{1}{3}$
CSM	1.56	1.73	1.44	1	$\frac{1}{3}$

Table 4.1: Ratios of FLAG vs Hand Design Parameters

and design time required.

In general, FLAG produced circuits did not compare that poorly against their hand counterparts. The FLAG circuits were not predicted to be as good as their hand counterparts; but the results were interesting. Table 4.1 lists, for each test circuit, the FLAG/hand ratios for the various test parameters. From Table 4.1, it can be seen that the FLAG circuits were generally larger and slower than their hand counterparts but required considerably less time to develop in contrast to the hand versions. In general, the lower the level of design for a circuit, the worse the FLAG version fared in comparison to the hand version. This is borne out in Table 4.1, the test circuits that FLAG generated the poorest layouts for were the xor, full adder and multiplexor. Not surprising, these circuits were designed at the lowest level of design, namely the transistor level. For circuits that were designed at a higher level, like the encoder and ripple carry adder, FLAG generated layouts that were closer to their hand rivals in terms of layout area and circuit speed. This can be attributed to FLAG's ability to incorporate hand-designed modules into its designs.

Overall, it seems that the less regular the circuit, the greater the disparity in design times between FLAG and the hand versions. The hand generated circuits were, in general, smaller and faster than the corresponding FLAG circuits due to the optimizations performed upon them by the designer during the layout process. Unfortunately, these optimizations are made at the expense of design time by the designer. Included among these optimizations is the packing of circuit elements in the horizontal direction and vertical direction as well as the optimizations that can be done at the logic level for some circuits. The packing together of circuit elements, by the designer, yields a much more significant decrease in circuit area at the transistor level than at higher levels of design. The xor, full adder and multiplexor were designed at the transistor level and had a much higher ($\frac{FLAG}{hand}$) area ratio than the other circuits which were designed at the gate level or higher.

What these examples have shown is that a designer using FLAG can generate circuits that are a reasonable approximation of their hand counterparts at a fraction of the time and effort required to generate a hand design. For instance, a designer using FLAG can "scale" circuits to an arbitrary size without any additional effort. Or the designer can create "composite" circuits merely by combining various circuit descriptions together, with greater ease and more quickly than would be possible by hand. By incorporating FLAG into the design cycle, a designer can perform architectural exploration and experiment with different circuit designs. This allows the designer to choose the most appropriate circuit design and implement that design by hand.

Future Work

The future work can be broken up into roughly three areas:

- Improvements to the FP Interpreter
- Improvements to the FP language
- Improvements to FLAG

In the FP interpreter, the currently available debugging features need to be expanded. To help debug the FP programs, the ability to stop, single step or trace functions should be added. The ability to perform some timing analysis while inside the FP interpreter should also be added. Another area of improvement lies within the readability of FP programs. As the FP language stands now, the FP specifications are rather cryptic, even to the experienced FP user. Having the ability to symbolically reference inputs and to automate the routing of data between functions would go a long way to making FP programs more readable and easier to write. In order to generate more efficient designs, FLAG must be expanded to provide the designer with greater control over the placement of circuit elements within the physical circuit. This includes providing the designer with control over the vertical placement of the circuit elements as well as the horizontal control he now has. To facilitate circuit simulation, the automatic placement of pins in the test circuit by FLAG would be a step toward speeding up of the process.

Bibliography

[Ance83] Anceau, F., "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits Specified by Algorithms," *Third CALTECH Conference on Very Large Scale Integration,* R. Bryant, Ed. Rockville, Maryland: Computer Science Press, 1983, pp.15-31.

[Back78] 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.

[Bade83] Baden, S., "Berkeley FP User's Manual, Rev. 4.1," MS, University of California, Berkeley, California, March 1983.

[Coll84] Collet, R., "Silicon Compilation: A Revolution in VLSI Design," *Digital Design,* August 1984, pp.88-95.

[Erce85] Ercegovac, Miloš D., Tomas Lang, *Digital Systems and Hardware/Firmware Algorithms,* New York: John Wile & Sons, 1985, pp. 239.

[Hwan79] Hwang, K., *Computer Arithmetic: Principles, Architecture, and Design,* New York, New York: John Wiley & Sons, 1979, pp. 78-81.

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

[John84] Johnson, S. C., "VLSI Circuit Design Reaches the Level of Architectural Description," *Electronics,* 3 May 1984, pp.121-128.

[Laht81] Lahti, D. O., "Applications of a Functional Programming Language," UCLA, Los Angeles, California, Tech. Rep. CSD-810403, April 1981.

[Liao83] Liao, Y. Z. and C. K. Wong, "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. CAD-2, No. 2, April 1983, pp. 62-69.

[Lieb82] Lieberherr, Karl J. and Svend E. Knudsen, "ZEUS: A Hardware Description Language for VLSI," *Proceedings of the 20th IEEE Conference on Design Automation,* June 27-29, 1982, pp.17-20.

[Lim65] Lim, Willie Y-P., "HISDL - A Structural Design Language," *Communications of the ACM,* November 1965, pp. 823-830.

[Mead80] 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.

[Mori82] Morison, J. D., N. E. Peeling and T. L. Thorp, "ELLA: A Hardware Description Language." *Proceedings of ICCC-82,* September 28 - October 1, 1982, pp.604-607.

[Pate85] Patel, D., M. Schlag, M. Ercegovac, "νFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms," in *Functional Programming Languages and Computer Architecture,* J.P. Jouannaud, Ed. Nancy, France: Springer-Verlag Lecture Notes in Computer Science, September 1985, pp. 238-255.

[Robi82] Robinson, P. and J. Dion, "Programming Languages for Hardware Description," *Proceedings of the 20th IEEE Conference on Design Automation,* June 27-29, 1982, pp.12-16.

[Roge85] Rogers, C. D., and S. W. Daniel, and J. B. Rosenberg, "An Overview of VIVID, MCNC's Vertically Integrated Symbolic Design System," *IEEE Design Automation Conference,* 1985, pp.62-68.

[Roge86] Rogers, C. D., "The VIVID Symbolic Design System: Current Overview and Future Directions," *IEEE Design & Test,* February 1986, pp.75-81.

[Rose85] Rosenberg, J. B., "Auto-Interactive Schematics to Layout translation'" *IEEE Design Automation Conference,* 1985, pp.82-87.

[Schl] Schlag, M., "FP Layout Manual," Internal Report, UCLA, Los Angeles, California, 12 September 1986.

[Schl84] Schlag, M. D. F., "Extracting Geometry from FP for VLSI Layout," UCLA, Los Angeles, California, Tech. Rep., CSD-840043, October 1984.

[Schl86] Schlag, M. D. F., "Layout from a Topological Description," UCLA, Los Angeles, California, Tech. Rep. PhD Dissertation, July 1986.

[Shah85] Shahdad, Moe, Roger Lipsett, Erich Marschner, Kellye Sheehan, Howard Cohen, Ron Waxman, and Dave Ackley, "VHSIC Hardware Description Language," *IEEE Computer,* February 1985, pp. 94-103.

[Shee84] Sheeran, M., "mFP, A Language for VLSI Design," *Proceeding of the 1984 ACM Conference on LISP and Functional Programming,* August 6-8, 1984, pp. 104-112

[Shiv79] Shiva, S. G., "Computer Harware Description Languages - A Tutorial," *Proceedings of the IEEE,* December 1979, pp. 1605-1615.

[Skla60] Sklansky, J., "Conditional Sum Adder Logic," *IRE Transactions on Electronic Computers,* June 1960, pp. 226-231.

[Vieg84] Viega, P. M. B. and M. J. A. Lanca, "HARPA: A Hierarchical Multi-Level Hardware Description Language," *Proceedings of the 21st IEEE Conference on Design Automation,"* June 25-27, 1984, pp.59-65.

[VIVID] *VIVID Designer Documentation,* Version 1.2, MCNC, Microelectronics Center of North Carolina, May 1986

[Worl84] Worley, J. S., "The UCLA Functional Programming Language," Internal Report, UCLA, Los Angeles, California, 15 June 1984.

[Worl86] Worley, J. S., "A Functional Style Description of Digital Systems," UCLA, Los Angeles, California, Tech. Rep. MS Thesis, CSD-860054, February 1986.

[West85] Weste, N. and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective,* Reading, Massachusetts: Addison-Wesley, 1985.

Appendix A

Syntax of FP

A.1 Objects

The set of objects Ω consists of atoms and sequences $< x_1,\ x_2, \ldots x_k >$ (where the $x_i \in \omega$). The atoms uniquely determine the set of valid objects and consists of the numbers, 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:

$$\begin{array}{ccc} \perp & 3.1415 & 1234567 \\ ab & "CD" & <1,<2,3>> \\ <> & \mathbf{T} & <a,<>> \end{array}$$

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'[Bac78].

A.2 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:

$$\begin{array}{ll} + :< 7,8 > \equiv 15 & \mathbf{tl} :< 1,2,3 > \equiv < 2,3 > \\ \mathbf{1} :< a,b,c,d > \equiv a & \mathbf{2} :< a,b,c,d > \equiv b \end{array}$$

A.3 Functions

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

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

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expression: [McC60]

$$p_1 \ \Rightarrow \ e_1; \ldots; \ p_n \ \Rightarrow \ e_n; \ e_{n+1}$$

This statement is interpreted as follows: return function e_1 if the predicate 'p_1' is true, otherwise e_2 if 'p_2' is true, $\ldots, e_n$ if 'p_n' is true. If none of the predicates are satisfied then default to e_{n+1}. It is assumed that $x, x_i, y, y_i, z_i \in \Omega$.

A.3.1 Selector Functions

For a nonzero integer, μ

$$\mu : x \equiv$$

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

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

$$\perp$$

The user should not that the function symbols 1,2 3, ... are to be distinguished from the atoms 1,2,3 ...

$$\textbf{last} : x \equiv$$

$$x = <> \Rightarrow <>;$$

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

$$\perp$$

$$\textbf{first} : x \equiv$$

$$x = <> \Rightarrow <>;$$

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

$$\perp$$

A.3.2 Tail Functions

$$\textbf{tl} : x \equiv$$

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

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

$$\perp$$

$$\textbf{tlr} : x \equiv$$

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

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

$$\perp$$

A.3.3 Distribute From Left and Right

$$
\begin{aligned}
\textbf{distl} : x \;\equiv\; & \\
& x = < y, <> > \Rightarrow <>; \\
& x = < y, < z_1, z_2, \ldots, z_k >> \Rightarrow << y, z_1 >, \ldots, < y, z_k >>; \\
& \perp
\end{aligned}
$$

$$\mathbf{distr} : x \equiv$$

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

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

$$\perp$$

A.3.4 Identity

$$\mathbf{id} : x \equiv x$$

$$\mathbf{out} : x \equiv x$$

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.

A.3.5 Append Left and Right

$$\mathbf{apndl} : x \equiv$$

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

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

$$\perp$$

$$\mathbf{apndr} : x \equiv$$

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

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

$$\perp$$

A.3.6 Transpose

$$\mathbf{trans} : x \equiv$$

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

$$x = < x_1, x_2, \ldots, x_k > \Rightarrow < y_1, y_2 \cdots, y_m >;$$

$$\perp$$

where $x_i =< x_{i,1}, \ldots, x_{i,m} > \wedge\ y_j =< y_{1,j}, \ldots, y_{k,j} >$, $1 \leq i \leq k$, $1 \leq j \leq m$.

$$\begin{aligned}
\textbf{reverse} : x \ \equiv & \\
& x =<>\Rightarrow<>; \\
& x =< x_1, x_2, \ldots, x_k >\Rightarrow< x_k, \ldots, x_1 >; \\
& \perp
\end{aligned}$$

A.3.7 Rotate Left and Right

$$
\begin{aligned}
\textbf{rotl} : x \ \equiv \ & \\
& x = <> \Rightarrow <>; \\
& x = < x_1 > \Rightarrow < x_1 >; \\
& x = < x_1, x_2, \ldots, x_k > \ \wedge k \geq 2 \Rightarrow < x_2, \ldots, x_k, x_1 >; \\
& \perp \\
\textbf{rotr} : x \ \equiv \ & \\
& x = <> \Rightarrow <>; \\
& x = < x_1 > \Rightarrow < x_1 >; \\
& x = < x_1, x_2, \ldots, x_k > \ \wedge k \geq 2 \Rightarrow < x_k, x_1, \ldots, x_{k-2}, x_{k-1} >; \\
& \perp \\
\textbf{concat} : x \ \equiv \ & \\
& x = << x_{11}, \ldots, x_{1k} > \ldots < x_{m1}, \ldots, x_{mp} >> \ \wedge k, m, n, p > 0 \\
& \quad \Rightarrow < x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots x_{m1}, \ldots, x_{mp} >; \\
& \perp
\end{aligned}
$$

Concatenate removes all occurrences of the null sequence:

$$\textbf{concat} : << 1, 3 >, <>, < 2, 4 >, <>, < 5 >> \equiv < 1, 3, 2, 4, 5 >$$

pair : $x \equiv$

$x = < x_1, x_2, \ldots, x_k > \wedge k > 0 \wedge k$ is even

$\Rightarrow << x_1, x_2 >, \ldots, < x_{k-1}, x_k >>;$

$x = < x_1, x_2, \ldots, x_k > \wedge k \geq 0 \wedge k$ is odd

$\Rightarrow << x_1, x_2 >, \ldots, < x_k >>;$

$\perp$

split : $x \equiv$

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

$x = < x_1, x_2, \ldots, x_k > \wedge k > 1$

$\Rightarrow << x_1, \ldots, x_{\lceil \frac{k}{2} \rceil} >, < x_{\lceil \frac{k}{2} \rceil + 1}, \ldots, x_k >>;$

$\perp$

A.3.8 Predicate (Test) Functions

atom : $x \equiv$

$x \in atoms \Rightarrow \mathbf{T};$

$x \neq \perp \Rightarrow \mathbf{F};$

$\perp$

$$\mathbf{eql} : x \equiv$$
$$x = < y, z > \ \wedge y = z \Rightarrow \mathbf{T};$$
$$x = < y, z > \ \wedge y \neq z \Rightarrow \mathbf{F};$$
$$\perp$$

Also less than ($<$), greater than ($>$), greater than or equal ($\geq$), less than or equal ($\leq$), not equal (**neql**); '=' is a synonym for **eql**.

$$\mathbf{null} : x \equiv$$
$$x = <> \Rightarrow \mathbf{T};$$
$$x \neq \perp \Rightarrow \mathbf{F};$$
$$\perp$$

$$\mathbf{length} : x \equiv$$
$$x = < x_1, x_2, \ldots, x_k > \Rightarrow k;$$
$$x = <> \Rightarrow 0;$$
$$\perp$$

A.3.9 Predicate operators: and, or, not, xor

$$\mathbf{and} :< x, y > \;\equiv$$

$$x = \mathbf{T} \;\wedge\; y \in \{\mathbf{T}, \mathbf{F}\} \Rightarrow y;$$

$$x = \mathbf{F} \;\wedge\; y \in \{\mathbf{T}, \mathbf{F}\} \Rightarrow \mathbf{F};$$

$$\perp$$

$$\mathbf{or} :< x, y > \;\equiv$$

$$x = \mathbf{F} \;\wedge\; y \in \{\mathbf{T}, \mathbf{F}\} \Rightarrow y;$$

$$x = \mathbf{T} \;\wedge\; y \in \{\mathbf{T}, \mathbf{F}\} \Rightarrow \mathbf{T};$$

$$\perp$$

$$\mathbf{not} : x \;\equiv$$

$$x = \mathbf{T} \Rightarrow \mathbf{F};$$

$$x = \mathbf{F} \Rightarrow \mathbf{T};$$

$$\perp$$

$$\mathbf{xor} :< x, y > \;\equiv$$

$$x \in \{\mathbf{T}, \mathbf{F}\} \;\wedge\; y \in \{\mathbf{T}, \mathbf{F}\} \;\wedge\; x = y \Rightarrow \mathbf{F};$$

$$x \in \{\mathbf{T}, \mathbf{F}\} \;\wedge\; y \in \{\mathbf{T}, \mathbf{F}\} \;\wedge\; x \neq y \Rightarrow \mathbf{T};$$

$$\perp$$

A.3.10 Arithmetic/Logical

$$+ : x \;\equiv$$

$$x =< y, z > \wedge\, y, z \text{ are numbers} \Rightarrow y + z;$$

$$\perp$$

$$- : x \;\equiv$$

$$x =< y, z > \wedge\, y, z \text{ are numbers} \Rightarrow y - z;$$

$$\perp$$

$$* : x \;\equiv$$

$$x =< y, z > \wedge\, y, z \text{ are numbers} \Rightarrow y \times z;$$

$$\perp$$

$/ : x \equiv$

$x = < y, z > \wedge y, z$ **are numbers** $\Rightarrow y \div z;$

$\perp$

A.3.11 Circuit Primitives

$$\begin{aligned}
\textbf{andg} :< x, y > \;\equiv\;& \\
& x = 1 \;\wedge\; y \in \{0,1\} \Rightarrow y; \\
& x = 0 \;\wedge\; y \in \{0,1\} \Rightarrow 0; \\
& \perp
\end{aligned}$$

$$\begin{aligned}
\textbf{org} :< x, y > \;\equiv\;& \\
& x = 0 \;\wedge\; y \in \{0,1\} \Rightarrow y; \\
& x = 1 \;\wedge\; y \in \{0,1\} \Rightarrow 1; \\
& \perp \\
\textbf{xorg} :< x, y > \;\equiv\;& \\
& x \in \{0,1\} \;\wedge\; y \in \{0,1\} \;\wedge\; x = y \Rightarrow 0; \\
& x \in \{0,1\} \;\wedge\; y \in \{0,1\} \;\wedge\; x \neq y \Rightarrow 1; \\
& \perp \\
\textbf{nandg} :< x, y > \;\equiv\;& \\
& x = 1 \;\wedge\; y \in \{0,1\} \Rightarrow \overline{y}; \\
& x = 0 \;\wedge\; y \in \{0,1\} \Rightarrow 1; \\
& \perp
\end{aligned}$$

$$\begin{aligned}
\textbf{norg} :< x, y > \;\equiv\;& \\
& x = 0 \;\wedge\; y \in \{0,1\} \Rightarrow \overline{y}; \\
& x = 1 \;\wedge\; y \in \{0,1\} \Rightarrow 0; \\
& \perp \\
\textbf{notg} : x \;\equiv\;& \\
& x = 1 \Rightarrow 0; \\
& x = 0 \Rightarrow 1; \\
& \perp
\end{aligned}$$

A.3.12 Library Routines (Used in Simulations)

$$
\begin{aligned}
\textbf{sin} : x \quad &\equiv \\
&x \text{ is a number} \Rightarrow \sin(x); \\
&\perp \\
\textbf{asin} : x \quad &\equiv \\
&x \text{ is a number} \wedge \mid x \mid \leq 1 \Rightarrow \sin^{-1}(x); \\
&\perp
\end{aligned}
$$

$$\textbf{cos} : x \equiv$$

x is a number $\Rightarrow \cos(x)$;

$\perp$

$$\textbf{acos} : x \equiv$$

x is a number $\wedge \mid x \mid \leq 1 \Rightarrow \cos^{-1}(x)$;

$\perp$

$$\textbf{exp} : x \equiv$$

x is a number $\Rightarrow e^x$;

$\perp$

$$\textbf{log} : x \equiv$$

x is a positive number $\Rightarrow \ln x$;

$\perp$

$$\textbf{mod} :< x, y > \equiv$$

x, y are numbers $\Rightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor$;

$\perp$

$$\textbf{sqrt} :< x, y > \equiv$$

x is a positive number $\Rightarrow \sqrt{x}$;

$\perp$

A.4 Combining Forms

Combining forms define new functions by operating on function and object parameters of the form. The resultant expressions can be compared and contrasted to the value-oriented expressions of traditional programming languages. The distinction lies in the domain of the operators; combining forms manipulate functions, while traditional operators manipulate values.

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

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

The **Constant** function takes an object parameter:

$$\%x : y \equiv (y = \perp \Rightarrow \perp ; x),\ \forall x, y \in \Omega$$

The function $\% \perp$ always returns $\perp$.

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

A.4.1 Compose

$$(\sigma @ \tau) : x \quad \equiv \quad \sigma : (\tau : x)$$

A.4.2 Construct

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

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

$$[+, /] : <3, 0> = <3, \perp> = \perp$$

A.4.3 Apply-to-All

$$\begin{aligned}
&\&\sigma : x \quad \equiv \\
&\qquad x = <> \Rightarrow <> \\
&\qquad x = <x_1, x_2, \ldots, x_k> \Rightarrow <\sigma : x_1, \sigma : x_2, \ldots, \sigma : x_k> \\
&\qquad \perp
\end{aligned}$$

A.4.4 Conditional

$$\begin{aligned}
&(\xi \rightarrow \sigma ; \tau) : x \quad \equiv \\
&\qquad (\xi : x) = \mathbf{T} \Rightarrow \sigma : x; \\
&\qquad (\xi : x) = \mathbf{F} \Rightarrow \tau : x; \\
&\qquad \perp
\end{aligned}$$

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy's conditional expression, and the *combining form* introduced here. In the former case the result is a value, while in the latter case the result is a function. Unlike Backus' FP, the conditional form must be enclosed in parenthesis, e.g.,

absolute_value (isNegative → -@[%0,id];id)

A.4.5 Constant

$$\%x : y \equiv$$
$$y = \perp \Rightarrow \perp \; ; \; x, \; \forall x \in \Omega$$

This function returns its object parameter as its result.

A.4.6 Right Insert

$$!\sigma : x \equiv$$
$$x = <> \Rightarrow e_f : x;$$
$$x = < x_1 > \Rightarrow x_1;$$
$$x = < x_1, x_2, \ldots, x_k > \wedge k \geq 2 \Rightarrow \sigma :< x_1, !\sigma :< x_2, \ldots, x_k >>;$$
$$\perp$$

e.g., $!+ :< 4, 5, 6 >= 15$. If σ has a right identity element e_f, then $!\sigma :<>= e_f$, e.g.,

$$!+ :<>= 0 \text{ and } !* :<>= 1$$

Currently, identity functions are defined for + (0), − (0), ∗ (1), /(1), and also for **andg** (1), **org** (0), **xorg** (0). All other functions default to bottom(⊥).

A.4.7 Left Insert

$$\textbf{lins}(\sigma) : x \equiv$$
$$x = <> \Rightarrow e_f : x;$$
$$x = < x_1 > \Rightarrow x_1;$$

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

$\quad \Rightarrow \sigma :< \mathbf{lins}(\sigma) :< x_1, \ldots, x_{k-1} >, x_k >;$

$\perp$

A.4.8 Right Seq

$$
\begin{aligned}
\mathbf{seq}(\sigma) : x \equiv\ & \\
& x = <> \Rightarrow e_f : x; \\
& x = < x_1 > \Rightarrow < x_1 >; \\
& x = < x_1, x_2 > \Rightarrow \sigma :< x_1, x_2 >; \\
& x = < x_1, x_2, \ldots, x_k > \ \wedge k > 2 \ \Rightarrow < y_1, y_2, \ldots, y_k >; \\
& \bot
\end{aligned}
$$

where $< z_2, y_3, \ldots, y_k >= \mathbf{seq}(\sigma) :< x_2, \ldots, x_k >$ and $< y_1, y_2 >= \sigma :< x_1, z_2 >$.

A.4.9 Left Seq

$$
\begin{aligned}
\mathbf{seqL}(\sigma) : x \equiv\ & \\
& x = <> \Rightarrow e_f : x; \\
& x = < x_1 > \Rightarrow < x_1 >; \\
& x = < x_1, x_2 > \Rightarrow \sigma :< x_1, x_2 >; \\
& x = < x_1, x_2, \ldots, x_k > \ \wedge k > 2 \ \Rightarrow < y_1, y_2, \ldots, y_k >; \\
& \bot
\end{aligned}
$$

where $< y_1, \ldots, y_{k-2}, z_{k-1} >= \mathbf{seqL}(\sigma) :< x_1, \ldots, x_{k-1} >$ and
$< y_{k-1}, y_k >= \sigma :< z_{k-1}, x_2 >$.

A.5 Time Domain Functions and Combining Forms

The functions **sopi**, **posi**, **invd**[1] are all functions which are equivalent to the FP function **id** but correspond to specific constructs in the layout necessary to sequentialize or parallelize the communication of objects between functions. There are time-domain equivalents to the **Apply-to-All**, the **Right** and **Left Insert** and the **Right** and **Left Seq** combining forms. These are denoted respectively by, **tapall, tlins, tseq, tseqL**. They are equivalent in terms of meaning to their

[1]Martine Schlag, "FP Layout Manual," MS, UCLA, Los Angeles, California, 12 September 1986, pp. 21

counterparts, however they assume a different implementation of their inputs and outputs with respect to time and space.

A.6 User Defined Functions

An FP definition is entered as follows:

{fn-name fn-form},

where *fn-name* is an ascii string consisting of letters, numbers and the underline symbol, and *fn-form* is any valid combining form, including a single primitive or defined function. For example, the functions

```
{factorial (zero?->%1; *@[id,factorial@2@[id,%1]])}

{zero? eql@[id,%0]}
```

form a recursive definition of the factorial function. Since FP systems are applicative it is permissible to substitute the actual definition of a function for any reference to it in a combining form: if $f \equiv 1@2$ then

$$f : x \; \equiv \; 1@2 : x, \; \forall x \in \Omega$$

References to undefined functions bottom out:

$$f : x \; \equiv \bot \; , \; \forall x \in \Omega, f \notin F$$

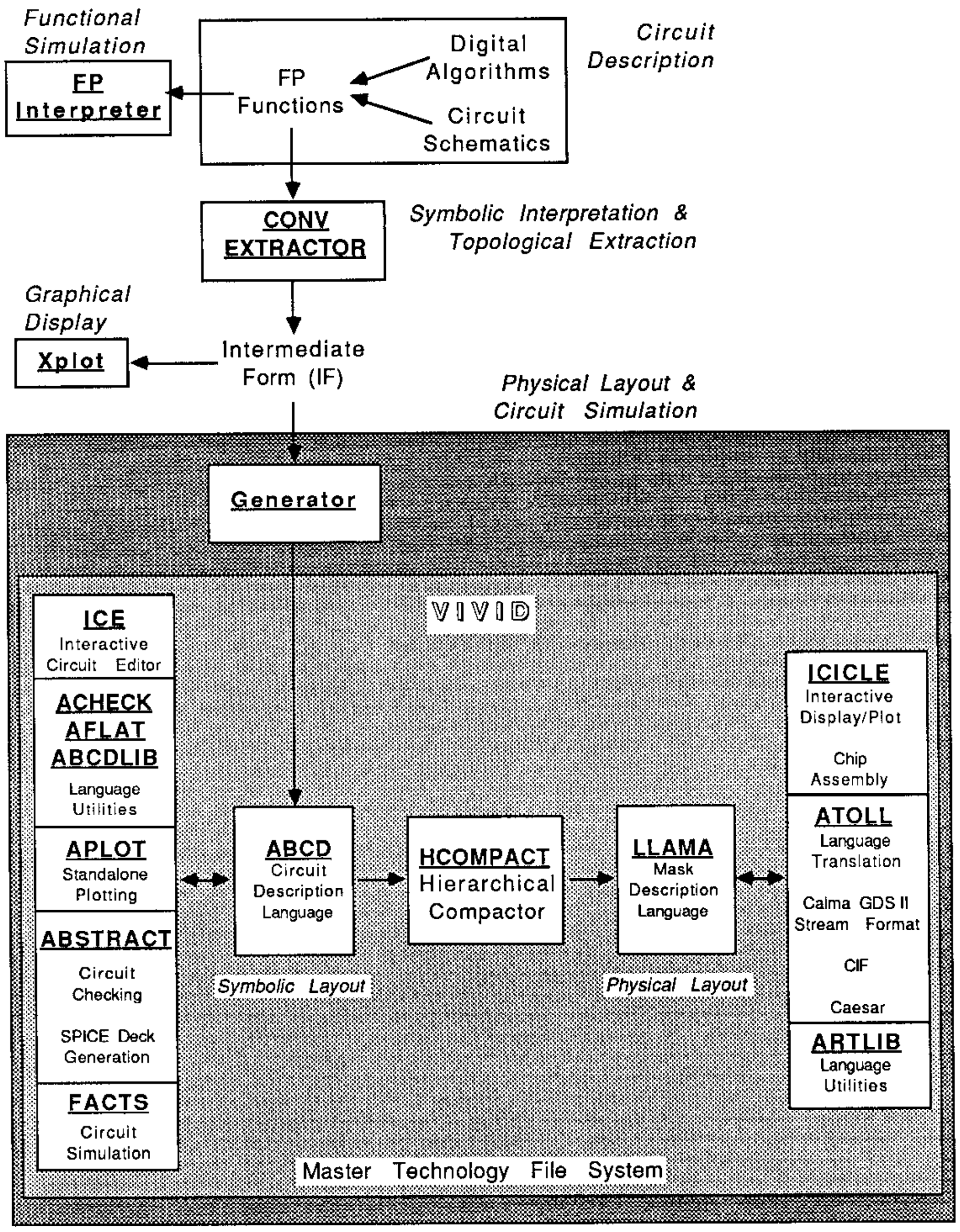
Appendix B

FLAG User Manual

This manual describes how to use FLAG in the generation of circuit layouts from FP descriptions. Figure B.1 presents an abstract view of FLAG. FLAG can be broken down into four sections.

- Functional Simulation
- Symbolic Interpretation and Topological Extraction
- Physical Layout and Circuit Simulation
- Graphical Display

The input into FLAG is a collection of FP functions created by the circuit designer from either a circuit diagram or digital algorithm. The input entered in by the user is shown in **boldface**.



Functional Simulation
FP Interpreter
FP Functions
Digital Algorithms
Circuit Schematics
Circuit Description
CONV EXTRACTOR
Symbolic Interpretation & Topological Extraction
Graphical Display
Xplot
Intermediate Form (IF)
Physical Layout & Circuit Simulation
Generator
VIVID
ICE Interactive Circuit Editor
ACHECK AFLAT ABCDLIB Language Utilities
APLOT Standalone Plotting
ABSTRACT Circuit Checking SPICE Deck Generation
FACTS Circuit Simulation
ABCD Circuit Description Language
Symbolic Layout
HCOMPACT Hierarchical Compactor
LLAMA Mask Description Language
Physical Layout
ICICLE Interactive Display/Plot Chip Assembly
ATOLL Language Translation Calma GDS II Stream Format CIF Caesar
ARTLIB Language Utilities
Master Technology File System

Figure B.1: Abstract View of FLAG

B.1 Functional Simulation

Functional simulation of the FP circuit description allows the designer to determine if the description correctly models the circuit of interest. The Berkeley FP interpreter [Bade83] is used to simulate the circuit description functionally.

B.1.1 Input

The input file consists of

FP functions syntax: { fname fdef }

Comments syntax: # Comments

Example:

```
# nandg(a,b) (a,b in {0,1})

{nandg (eq@[%1,1]->-@[%1,2];%1)}

# xorg(a,b) (a,b in {0,1})

{xorg nandg@[nandg@[1,2],nandg@[2,3]]@[1,nandg,2]}
```

B.1.2 Steps

1. Start the Berkeley FP interpreter.
   /usr/local/bin/fp
   The following should appear on your screen:

   ```
   FP, v. 4.2, (4/28/83)
   ```

2. Load FP functions. Let us say that the file **test.fp** contains the FP function definitions.
)load test
   The name of each function is printed as it is loaded into the FP interpreter.

```
{nandg}
{xorg}
```

3. Test the functions. See [Bade83] for a list of system commands for the Berkeley FP interpreter.

```
       )fns
nandg         xorg
       )pfn nandg
{nandg (eq@[% 1,1]->-@[% 1,2];% 1)}
       )pfn xorg
{xorg nandg@[nandg@[1,2],nandg@[2,3]]@[1,nandg,2]}

       xorg:<0 0>
0
       xorg:<0 1>
1
       xorg:<1 0>
1
       xorg:<1 1>
0
```

4. Exit the interpreter when done.
   cntl-D

B.1.3 Comments

The circuit primitives presented in Appendix A are not available in the Berkeley FP interpreter and must be created by the designer.

B.2 Symbolic Interpretation and Topology Extraction

The first step in the generation of a layout from a circuit description is to generate an IF (intermediate form) description of the circuit. The IF is obtained by performing a topology extraction upon the results of a symbolic interpretation of the circuit description using symbolic objects. The interpreter/extractor (henceforth referred to as the "extractor") sits within a T interpreter, affording the full power of the T environment. It is necessary, then, to translate the FP functions into a series of T commands which define the FP functions for the extractor before the extractor can be run.

B.2.1 Input

The input file consists of

FP functions syntax: { fname fdef }

T commands syntax: #.T command

Comments syntax: # Comments

Example:

```
# nandg(a,b) (a,b in {0,1})

{nandg (eq@[%1,1]->-@[%1,2];%1)}

#.(dfbx 'nandg 1 nand 3)

# xorg(a,b) (a,b in {0,1})

{xorg nandg@[nandg@[1,2],nandg@[2,3]]@[1,nandg,2]}
```

B.2.2 Output

The output file consists of an IF (Intermediate Form) description of the circuit in the format presented in (section IF syntax).
Example:

```
((G00257 G00258)
 (G00257 (* G00258 * + ^ ^))
 ((* G00257 * + ^ ^) G00258 G00258)
 (G00257 G00257 G00258 G00258)
 (G00257 ($ B 1 G00266 NAND 3 $ G00257 G00258 $ G00259 $) G00258)
 (G00257 (* G00259 * + ^ ^) G00258)
 (G00257 G00259 G00259 G00258)
 (($ B 1 G00264 NAND 3 $ G00257 G00259 $ G00260 $)
  ($ B 1 G00265 NAND 3 $ G00259 G00258 $ G00261 $)
 )
 (G00260 G00261)
 (($ B 1 G00263 NAND 3 $ G00260 G00261 $ G00262 $))
 (G00262)
)
```

B.2.3 Steps

Suppose the file **test.fp** contains the FP specification for an XOR gate as presented above. The steps necessary to perform symbolic interpretation and topology extraction upon the XOR are given below.

1. Convert FP functions into T commands by passing it through a filter: **conv**. **/u/gs8/winthrop/extractor/bin/conv test.fp**
   The following should appear on the screen:

   ```
   Reading file : test.fp
   Creating file : test.t ...  finished.
   ```

   If there is a syntax error in one of the FP functions, the following will occur.

   ```
   Creating file : test.t ... syntax error
   ```

2. Set the environment variable FPLAYDIR to the directory containing the layout system. This is achieved in the C shell by:
   setenv FPLAYDIR /u/gs8/winthrop/extractor/bin
   or in the Bourne shell by:
   FPLAYDIR=/u/gs8/winthrop/extractor/bin ;
   export FPLAYDIR

3. Start the T interpreter.

   t -h 3000000

   The following should appear on the screen:

   ```
   2999992 bytes per heap, 131071 bytes reserved
   T 3.1 (5) MC68000/UNIX  Copyright (C) 1988 Yale University
   T Top level
   >
   ```

4. Load the layout interpreter.

   (load '(FPLAYDIR lcmd))

   The following should appear:

   ```
   ;Loading  ~winthrop/extractor/bin/lcmd.t into USER-ENV
   Loading FP Layout Interpreter ...
   ;Loading ~winthrop/extractor/bin/prop.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/clint0.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/typedefs.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/cgraph.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/ghost.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/aux3.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/layout.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/pred0.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/oper0.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/oper2.t into USER-ENV
   [Redefining syntax BINARY-OP] [Redefining syntax UNARY-OP]
   ;Loading ~winthrop/extractor/bin/oper5.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/struc0.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/struc2.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/struc5.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/walk.t into USER-ENV
   ;Loading ~winthrop/extractor/bin/arrange.t into USER-ENV
   ```

```
Loaded.
;no value
>
```

5. Load converted FP functions.
   (load 'test)
   The following should appear on the screen:

```
;Loading  test.t into USER-ENV
NANDG #{Procedure 1} XORG XORG
>
```

   The file **test.t** must be in the current directory. If you wish to load a file from another directory, you must enter its full pathname as well as its **.t** extension and enclose it in double quotes. ~ and other shell-meta characters are not expanded by T.
   (load "/u/gs8/winthrop/thesis/appendix/test.t")

6. Perform symbolic interpretation and topology extraction. The T function **quick-layout** requires an FP function and an input. The input for the FP function is given as a list object (an FP object with <, >'s replaced with (,)'s and the ',' replaced by spaces). It will execute all the commands necessary to produce the IF file. The name of the IF file will be the name of the FP function with a **.d** extension. The layout of the function **xorg** for two inputs would be obtained as follows:
   (quick-layout 'xorg '(1 1))
   The following should appear on the screen:

```
Computation Graph Completed.
Marking of computation graph started ....  completed.

Layout started ....  completed.
```

```
Writing layout into file xorg.d .... written.
;no value
```

The name of the output file can be specified as a third argument to **quick-layout**; the **.d** extension will be appended to the name provided.

```
> (quick-layout 'xorg '(1 1) '/tmp/junk)}

Computation Graph Completed.
Marking of computation graph started ....  completed.

Layout started ....  completed.

Writing layout into file /tmp/junk.d .... written.
;no value
```

7. Terminate T session.
   (exit)

B.2.4 Comments

There is one interpreter command which should be mentioned here. The interpreter can be instructed to ignore the graphical interpretation of any function and instead represent the function as a box. You can instruct the interpreter to do this through the **dfbx** command. You must provide the name of the function and the height of the box (a positive integer) and a label to be displayed in the box.

```
> (dfbx 'XORG 1 xor)
#{procedure 1}
>
```

In addition, you can optionally specify the width of the box as was done for the function **nandg** in **test.fp**. However, the actual width given to the box will be at least 1+ max { #inputs, #outputs } of the function. The command remains in effect during the T session until you redefine the function (this occurs if you reload the file containing the definition of the function) or issue another **dfbx** command with the same function.

B.3 Physical Layout and Circuit Simulation

After the IF description of the circuit has been generated, the next step is to generate a physical layout of the circuit from its IF description. The symbolic layout generator takes the IF description, performs some compaction and wire straightening, and produces a symbolic layout of the circuit, written in ABCD. The symbolic layout is passed on to the collection of VIVID tools to perform the physical layout and circuit simulation.

B.3.1 Input

The input consists of the following set of files.

- IF description of the circuit (typically with a **.d** extension).
- For each predesigned module to used in the circuit:

  modulename.ab ABCD description of module. The design for the modules are subject to the following constraints.

  - Vdd & Vss lines must run horizontally and span the entire width of the module.
  - Vdd & Vss lines must extend 1 grid unit past any circuit element.
  - Module inputs can only appear at the top of the module and must be in polysilicon.
  - Module outputs can only appear at the bottom of the module and must be in polysilicon.
  - Module inputs and outputs must extend 1 grid unit past any Vdd or Vss line.

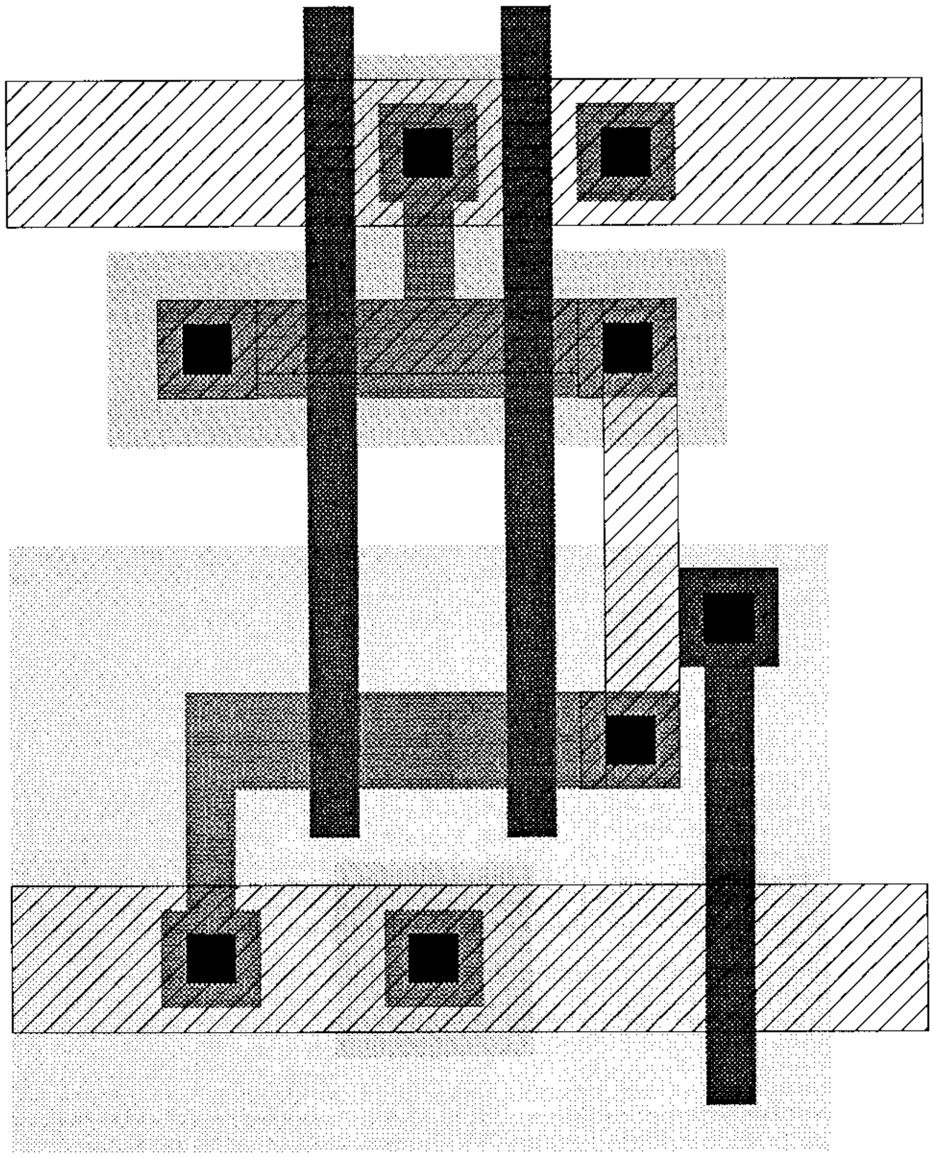


Figure B.2: Example Layout of a 2-Input NAND

An layout for a two-input NAND gate is shown in Figure B.2.

modulename File listing positions of all the inputs and outputs for module. The syntax for the file is given below.

```
module_width     module_height
distance of 1st input from left side of module
distance of next input to right of previous input
.
.
distance of next input to right of previous input
distance of 1st output from left side of module
distance of next output to right of previous output
.
.
distance of next output to right of previous output
distance of 1st Vdd line from top of module
distance of next Vdd line below previous Vdd line
.
.
distance of next Vdd line below previous Vdd line
-1 (Acts as delimiter)
distance of 1st Vss line from top of module
distance of next Vss line below previous Vss line
.
.
distance of next Vss line below previous Vss line
-1 (Acts as delimiter)
```

Example: two-input NAND gate

```
7 6
2
2
6
```

```
1
-1
5
-1
```

A graphical interpretation of the file is given in Figure B.3.

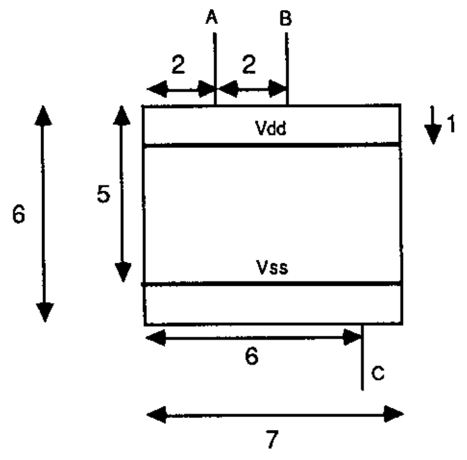


Figure B.3: Graphical Interpretation of NAND

Example: xor

`NAND.ab   NAND   xorg.d`

Also mention module description files

B.3.2 Output

The output consist of the following set of files.

***.ab** Symbolic layout of circuit.

***(000-999).ab** Leaf cells of the symbolic layout.

***.fa** Circuit simulation file.

***.ll** Physical layout of circuit in LLAMA format.

***.cif** Physical layout of circuit in CIF format.

Example: xor

```
xor.ab    xor.fa
xor.ll    xor.cif
xor000.ab xor005.ab
xor001.ab xor006.ab
xor002.ab xor007.ab
xor003.ab xor008.ab
xor004.ab xor009.ab
```

B.3.3 Steps

Suppose the file **xorg.d** contains the IF specification for an XOR gate as presented above.

1. Generate a symbolic layout of the circuit from its IF description.
   /u/gs8/winthrop/bin/generate xorg.d xor
   The following should appear on your screen.

   ```
   generating graph
   generating constraints
   calling compaction routine
   straightening wires
   creating output files
   ```

   xorg.d is the name of the file containing the IF description of the xor gate. **xor** is the name of the symbolic layout generated.

2. Create a circuit simulation file from the symbolic layout.
   /s/s1/VIVID/bin/abstract -t scmos xor
   The circuit simulation will be in the file **xor.fa**. The **-t scmos** option indicates that a scalable CMOS technology was used in the design of the circuit. The value of the **-t** option will depend upon the technology actually used in the design of the circuit.

3. Run the circuit simulator.
   /s/s1/VIVID/bin/facts xor

4. Generate a physical layout in LLAMA.
   /s/s1/VIVID/bin/hcompact -t scmos xor
   The physical layout created will be in the file **xor.ll** The **-t scmos** option indicates that a scalable CMOS technology was used in the design of the circuit. The value of the **-t** option will depend upon the technology actually used in the design of the circuit.

5. Translate the physical layout from LLAMA into CIF.
   /s/s1/VIVID/bin/atoll -t scmos xor.ll xor.cif
   The file **xor.cif** will contain a description of the physical layout in the CIF (CalTech Intermediate Format) format. The **-t scmos** option indicates that a scalable CMOS technology was used in the design of the circuit. The value of the **-t** option will depend upon the technology actually used in the design of the circuit. A hardcopy of the physical layout cam be produced from the CIF file by executing the following command.
   cifp xor.cif | lpr
   This command will produce a Postscript version of the file which can be printed.

B.3.4 Comments

Refer to [VIVID] for more detailed information about the various VIVID tools and ABCD and LLAMA syntax.

B.4 Graphical Display

The ability to graphically display the IF description has also been provided. The program **Xplot** takes an IF circuit description as input and generates a graphical representation of the FP function which can be displayed on the terminal or a hard copy can be generated.

B.4.1 Input

The input file consists of an IF description of the circuit as generated by the **extractor**.
Example(xorg.d):

```
((G00257 G00258)
 (G00257 (* G00258 * + ^ ^))
 ((* G00257 * + ^ ^) G00258 G00258)
 (G00257 G00257 G00258 G00258)
 (G00257 ($ B 1 G00266 NAND 3 $ G00257 G00258 $ G00259 $) G00258)
 (G00257 (* G00259 * + ^ ^) G00258)
 (G00257 G00259 G00259 G00258)
 (($ B 1 G00264 NAND 3 $ G00257 G00259 $ G00260 $)
  ($ B 1 G00265 NAND 3 $ G00259 G00258 $ G00261 $)
 )
 (G00260 G00261)
 (($ B 1 G00263 NAND 3 $ G00260 G00261 $ G00262 $))
 (G00262)
)
```

B.4.2 Output

The output consists of either a graphical display in a X window or a file of TEX commands which create a graphical representation of the circuit.

B.4.3 Steps

Suppose the file **xorg.d** contains the IF specification for an XOR gate as presented above. The steps necessary to generate a graphical representation of the XOR are given below.

1. Start **/u/gs8/winthrop/extractor/bin/Xplot**
   Xplot xorg.d
   The following should appear:

```
********** HELLO *********

Processing file : xorg.d


 10  LINES 34 NODES  67 EDGES

END OF FILE
number of ops 4


Compacting

Tracing wires
Tracing wires completed

***** Compaction completed

12 wires to be fixed
Fixing wires completed
Command(q = quit, d = draw, p = psfilt, t = tex)?
```

2. The IF has now been compacted, the wires straightened and a graphical representation of the circuit can be (**d**)isplayed in a X window, or a hard copy generated can be obtained by generating a TEX file. In the following, a TEX file is produced by typing t and then providing responses following the :'s. The dimensions are in inches. The default name of the output file is obtained by removing the **.d** extension from the input file name (if it ended with **.d**) and appending **.tex**. The default file name is displayed and will be used if **return** is hit in response to the file name prompt.

```
Command(q = quit, d = draw, p = psfilt, t = tex)?t
```

```
 xsize (max = 7.5) : 5.75

 ysize (max = 10.0) : 5.75

 labels (y or n) : y

 point size (6-12) : 6

 labels on, point size : 6
iy=5.750000 height=21 yscale=0.264286,
ix=5.750000 width=7 xscale=0.821429

 Everything ok (y or n) : y

 Enter file name (xorg.tex):
 Writing 'xorg.tex' .  File 'xorg.tex' written.
```

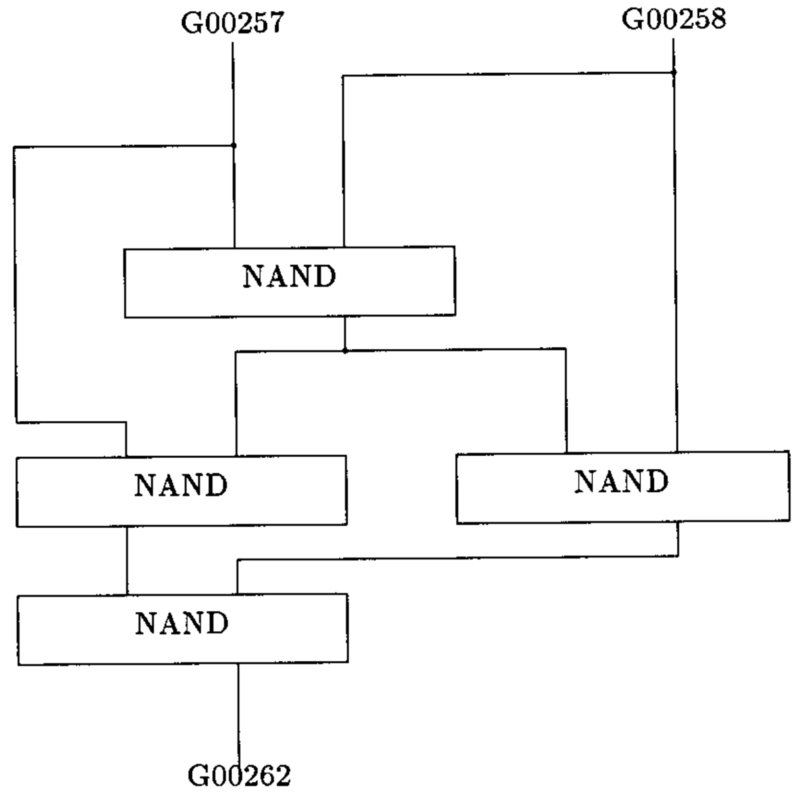


Figure B.4: Graphical Representation of **xorg.d**

The TEX file can be included within a LATEX document using the following commands:

```
\begin{figure}
\include{xorg}
\centerline{\box\graph}
\end{figure}
```

Figure B.4 was generated by including the TEX file produced by **Xplot** from **xorg.d**. The labels on the inputs and outputs are unique identifiers generated by the extractor. These can be edited but have been left in Figure B.4.

3. Exit **Xplot**

```
Command(q = quit, d = draw, p = psfilt, t = tex)?q
```