SYNTHESIS OF CUSTOM INTEGRATED CIRCUITS FROM A HIGH-LEVEL BEHAVIORAL DESCRIPTION

Steven Hennick Kelem

July 1987
CSD-870028

UNIVERSITY OF CALIFORNIA

Los Angeles

Synthesis of Custom Integrated Circuits From a High-Level Behavioral Description

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

by

Steven Hennick Kelem

1987

© Copyright by

Steven Hennick Kelem

1987

The dissertation of Steven Hennick Kelem is approved.

[illegible]

Bill Mitchell

[illegible]

Bruce Rothschild

[illegible]

Vance Tyree

[illegible]

Bertram Bussell, Committee Co-Chair

[illegible]

Miloš Ercegovac, Committee Co-Chair

University of California, Los Angeles

1987

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

VITA

November 12, 1952	Born, North Hollywood, California
1976	B.S. Mathematics-Computer Science, University of California, Los Angeles
1978	M.S. Computer Science, University of California, Los Angeles

PUBLICATIONS

Steven H. Kelem, *A Compiler for Silicon: An Automatic Method for the Translation of High-Level Algorithms into Integrated Circuit Masks*, Technical Report ATR-85(8497)-1, The Aerospace Corporation (September 1982).

Robert Cuykendall, Anton Domic, William H. Joyner, Steve C. Johnson, Steve Kelem, Dennis McBride, Jack Mostow, John E. Savage, and Gabriele Saucier, Design Synthesis and Measurement, *VLSI ∩ Software Engineering Workshop Report* (October 1982).

Robert Cuykendall, Anton Domic, William H. Joyner, Steve C. Johnson. Steve Kelem, Dennis McBride, Jack Mostow, John E. Savage, and Gabriele Saucier, Design Synthesis in VLSI and Software Engineering, *The Journal of Systems and Software*, 4(1):7–12 (April 1984), This is a reprint of the workshop report.

John J. Helly, Jr., William V. Bates, Mel Cutler, and Steve Kelem, *A Representational Basis for the Development of a Distributed Expert System for Space Shuttle Control*, Technical Report, NASA, Houston, Texas (May 1984).

Steven H. Kelem, *A Method for the Automatic Translation of Algorithms From a High-Level Language into Self-Timed Integrated Circuit Masks*, Technical Report TR-0084A (5920–03)-1, The Aerospace Corporation (March 1985).

Steven H. Kelem, A Method for the Automatic Translation of Algorithms from a High-Level Language into Self-Timed Integrated Circuits, *IEEE Circuits and Devices Magazine*, 1(2):17–19,44 (March 1985).

Steven H. Kelem, *A Method for Compact Two-Layer Routing of Permutations in Less Than n^2 Time*, Technical Report TR-0086A (2920–03)-1, The Aerospace Corporation (February 1987).

ABSTRACT OF THE DISSERTATION

Synthesis of Custom Integrated Circuits

From a High-Level Behavioral Description

by

Steven Hennick Kelem

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1987

Professor Bertram Bussell, Co-Chair

Professor Miloš Ercegovac, Co-Chair

This dissertation describes an innovative method for synthesizing error-free prototype integrated circuit masks from a behavioral-level algorithmic specification. The designer expresses algorithms and data types without having to specify a layout for their implementation or having to rely on time-consuming methods to obtain an implementation. The layout of the designed circuit is not explicitly specified by the designer, and it is determined from the algorithmic description. The circuits are produced quickly, but not necessarily as compactly as can

be done with exhaustive, time-consuming, automatic or manual techniques. The circuits produced are properly formed (i.e. obey the fabrication design rules) so non-behavioral attributes such as area, speed, and power can be estimated directly from these designs. When the data types in a design are changed, different operators corresponding to the new types are utilized and the non-behavioral attributes of the resulting circuit change. Comparisons of these attributes are the basis for choosing one design over others.

This method of designing digital circuits is intended to meet the needs of a specialized set of problems, namely the design of special-purpose computing circuits. Such circuits typically are non-standard designs which require many iterations in the design process to obtain a cost-effective design. Therefore a rapid-prototyping design method, mapping algorithms given at a high level into circuit structures is highly desirable. This work is motivated by these needs and attempts to provide such a design tool.

CHAPTER 1

Introduction

This dissertation proposes a novel solution to a fundamental problem in the design of special-purpose integrated circuits. Because such circuits are often nonstandard, as are the subcircuits from which they are constructed, many iterations through the design process are needed to produce cost-effective designs. The hypothesis of this dissertation is that the higher the level of description, the easier it is for a designer to specify an algorithm that is to be implemented as an integrated circuit that meets an even higher-level requirement specification. A behavioral, functional level of description allows the designer to choose between variations on details of a given algorithm. Using existing design methods to determine whether a particular implementation of a functional description meets the design requirements is a difficult and lengthy task.

Many IC design systems translate logic or transistor schematics (or a textual equivalent) into IC masks. Specifying these schematics is a time-consuming and tedious process (even with the aid of a computer) and the schematic is not related closely to the algorithm that the schematic implements. If a data type or some other aspect of the algorithm is changed, it is often difficult to apply those changes to the schematic without changing much (or all) of the schematic. When a designer implements an algorithm that will be on an integrated circuit, it is important for the designer to experiment with variations on the desired algorithm. Designers experiment to find a form of an algorithm that will fit in the desired area of a chip, meet the power budget for the chip, and operate at or above the desired speed. Other possible variations on an algorithm might result in a circuit that is more testable, fault tolerant, reliable, or temperature insensitive.

This thesis discusses a method for specifying an algorithm that will be implemented on a custom integrated circuit. The synthesis process described here has time complexity of less than n^2, and so scales well for large problems. The circuits produced by this method are not as area-efficient as manually-designed circuits. Once several variations on an algorithm are produced, a designer can select the few that best meet the specifications and then apply more exhaustive methods to make those variations more area-efficient.

To set the stage for the thesis, it is useful to examine the integrated circuit design process. An integrated circuit can be described at several different *levels of abstraction*. In top-down design, one starts with a very abstract description of the desired system and then carefully refines the description until the desired complexity is reached. The desired complexity for a complete description contains the target primitives or predefined descriptions that contain the target primitives. Some of the levels of abstraction for the systematic definition (refinement) of integrated circuits were given in [Cuyk82,Cuyk84] and are repeated here.

Requirements: The specification of the overall performance, area, and I/O for the circuit.

Abstract Algorithms: The behavior of the circuit without a binding for the actual operations and data types.

Concrete Algorithms: The behavior of the circuit expressed in a machine-independent programming language. The operators and data types are specified.

Programming: The machine language for the circuit if it is programmable.

Register Transfer: The behavior described in terms of states during which data is transferred between registers in the circuit.

Logic: The circuit description in terms of logic components and their interconnections.

Discrete Circuit: Logic functions in terms of transistors, resistors, capacitors, etc.

Topography: A circuit in which physical dimensions are absent, but in which relative positioning is expressed.

Masks: Transistors are defined by the intersections of polygonal areas on masks that are used in the fabrication process for integrated circuits.

This list stops short of the steps required for integrated circuit fabrication.

Typically a design is specified at one of these levels, and one or more designs at a lower level is synthesized from that specification. If some of these designs meet the requirements specification, then they can be further refined into lower-level designs. It is difficult to know whether a component of a design at some level of abstraction will meet the requirements specification unless it is designed down to some level where the design attributes can be measured, estimated, or simulated. Then they can be compared to the requirements specification. This means that integrated circuits have to be designed down to the mask level before a designer can determine whether the design meets the requirements. Unfortunately each transition between layers of abstraction results in many different design alternatives. If there were only two design possibilities at each of the seven levels, then 128 designs would have to be created and analyzed for adherence to the requirements specification. In reality, there are many more than two design possibilities at each level, and so the number of designs that need to be analyzed is enormous.

Mask Topography

Several techniques have been developed to reduce the number of design options at different levels of abstraction. At the lowest level of abstraction—the integrated circuit mask level, several efforts of note have resulted in techniques with fewer design alternatives for the production of IC masks. The first is the lambda-based simplified design rules [Mead80,Sequ82]. These rules have the following characteristics.

- The number of fabrication layers available to the designer to design with is reduced.

- The interactions between fabrication layers is simplified.
- The designer's idealized masks are isolated from physical masks by the introduction of a scale factor named *lambda* (λ).
- All dimensions are integral multiples of lambda.
- Designs are isolated from peculiarities of individual IC manufacturers' design rules and fabrication lines.

Another method that has reduced the number of mask design alternatives is the *gate array.* These are large arrays of gates which have been fabricated except for the metal layers which serve to interconnect the inputs, outputs, and power. The gates are those which the manufacturer believes will be easily incorporated into a design. Typically the gates are as simple as transistors or multiple-input nand or nor gates. In any case, these gates are prearranged in two-dimensional arrays with space between the rows and columns for the designer-specified routing. The set of masks for the pre-fabricated layers of the gates are the same for all of the wafers for a gate array, and only the masks for the metal layers are customized for a particular design. This reduces the overall cost of producing the circuits since the cost of the pre-fabricated layers can be amortized over many designs.

Topography

A method that reduces the number of design alternatives at the topographical level is a graphical method called *sticks* [Will77]. This method represents transistors and wires as thin lines instead of rectangles. Topological information is represented in the relative positions of the elements in a design.

Logic and Topography

At the logic and topographical levels, some particular layouts are used to simplify design, which has the effect of reducing the number of design alternatives. These are *Programmable Logic Arrays* (PLA), *Standard Cells*, and *Datapaths.* PLA's are compact structures for implementing sums of product terms. With the addition of clocked registers, finite-state machines can be built from PLA's.

Standard Cells that are circuits laid out with a common height but may have differing widths. The cells are placed horizontally adjacent to one another. Cell rows are separated by wire routing which interconnects the functions' inputs and outputs.

A *datapath* is an organization at the system layout level. This results from the separation of data and control operators in a processor. These two sections are created separately and then the appropriate wires run between them. Usually all the data operators operate on the same sized data and so it is convenient to arrange all these operators in a row, or path, with buses running above and below the row.

Silicon Compilers

Finally, at the higher levels of abstraction, there are several translation methods called *silicon compilers.* These compilers translate between different levels of abstraction. The only thing common to all of the silicon compilers is that they take a textual representation of a design at some level of abstraction and translate it to a design at a lower level of abstraction.

Johannsen's *Bristle Blocks* compiler [Joha78] enables a designer to specify an algorithm for defining the layout of cells and placing and routing these cells. His compiler synthesizes only cells that can be abutted, thus avoiding the routing problem. Cells are stretched vertically or horizontally so that the input and output ports on adjacent cells will line up when placed next to each other.

Similarly, Johnson's language *Xi* [John83] is an extension of the programming language C [Kern78] that allows a designer to specify algorithms for interconnecting cells. Assignment statements correspond to bindings to signals or registers. The output of *xi* is a list of cells and nets that are interconnected. This information is fed to a place and route program which searches exhaustively for a placement of the cells and a routing of the wires that interconnect them.

Ayres' language *RELAY* (*REcursive Logic ArraYs*) [Ayre79,Ayre83] is a language for describing behavior in terms of synchronous logic. RELAY, written in the general-purpose language *ICL* (*Integrated Circuit Language*) [Ayre79,Ayre83], gives the designer full access to the ICL language. The logical formulae in the synchronous logic are translated into an optimized disjunctive form. From this form, ICL is used to simulate the logic and to translate the logic into clocked PLA structures. RELAY programs are an unusual marriage between logic and layout. The designer needs to simultaneously specify a circuit's behavior and its geometric implementation.

The input to Hamachi's language *PEG* (*PLA Equation Generator*) [Hama85] is a specification of a finite state machine that is isomorphic to a Moore machine [Moor55]. States are explicitly described. Sequential state transitions are implicit in the ordering of the states in the specification. Non-sequential state transitions are explicitly stated. The finite state machines descriptions produced contain only the control and state portions of the machine.

Siskind, et al [Sisk81] produced a language and translator called *MacPitts*[1]. This language is at the register transfer level of abstraction, allowing specification of parallel communicating processes in a Lisp-like syntax. The target architecture is a combination of finite state machines, one for each of the parallel processes in the source code, and a data path unit. The datatypes consist only of the integer and Boolean types. A functional simulator detects simultaneous write attempts to the same register.

Patel, Schlag, and Ercegovac [Pate85,Schl84,Schl86] designed and implemented ν*FP*, an extension of Backus' functional programming language, *FP* [Back78]. This allows the multi-level specification, evaluation, and synthesis of hardware from an applicative language. Layouts and routing at the schematic (block diagram)

[1]MacPitts is not an acronym. The name is an "in-joke" of some sort.

level or at the mask level can be obtained. The applicative language is especially amenable to formal transformation methods. Transformations are applied to programs to reduce the overall area of a circuit and to improve routing.

Pangrle and Gajski have programs called *SLICER* (a state synthesizer) and *SPLICER* (resource allocator) [Pang86]. SLICER transforms an algorithm specified at the register transfer or concrete algorithm level into a dependency graph. It then performs a critical path analysis of the graph to find the longest path through the graph. When the critical path has been found, all computations in the algorithm are assigned to (possibly multiple) states. SPLICER then performs a (backtracking) algorithm to allocate resources (computational elements) to the different states. The backtracking aspect of this program might try all possible mappings between the computational elements and the dependency graph nodes. This program synthesizes a circuit with the least amount of resources while executing as fast as the critical path will allow.

The system described here takes a description written in a functional language and produces integrated mask specifications for prototyping. A fixed layout procedure is executed for each construct in the language, but these procedures include some heuristics that quickly find a viable layout. Fast routing techniques and the hierarchical nature of the language and specifications allow circuits to be synthesized quickly so that their speed, power, and area can be quickly determined.

Design Correctness

If the synthesis process is manual (or automatic but not trustworthy), then it is necessary to ensure the process has been done correctly. This involves one of the following techniques.

verification A formal proof that certain properties hold in the both the source and target levels of abstraction.

validation Exhaustive simulation of both levels of abstraction and comparison of the results.

simulation Modeling both levels of abstraction at some subset of the possible input cases and comparison of the results.

The next chapter describes a language (ALICS—Algorithmic Language for Integrated Circuit Specification) for the specification of the behavior of algorithms to be implemented in integrated circuitry. The language is based on Algol 68 [Wijn75] and FP [Back78], with some features borrowed from νFP [Mesh85,Schl86]. The data-typing features and procedural organization were taken from Algol 68. Algol's general assignment operators (and main memory) were removed as in FP—assignment through the "von Neumann bottleneck" is not an easily specified operation. However, well-structured assignments are reintroduced inside sequential domains. Sequential constructs from νFP were unified into a single construct. This construct packages sequential behavior within a domain so that it appears to be functional from outside the domain.

The third chapter discusses the synthesis method. A method is given for efficiently analyzing an ALICS source program. This consists of lexical analysis, parsing, and the transformation of these expressions into a structural representation of the program. This structure is mapped onto a tree of interconnected cells that implement the functions in the program. A method is demonstrated for placing cells hierarchically and for routing of functions and conditional expressions. Efficient methods are defined for routing of permutations and one-to-many mappings. These mappings are needed for the routing of signals from a function's formal parameters to the points where the parameters are referenced. Finally, methods for measuring non-behavioral attributes of the designs and the validity of this prototyping method are discussed.

Chapter four shows the time complexity of the synthesis algorithm to be n^2. To be useful, a prototyping system needs to work quickly. The time complexity gives a good indication of how long it will take for a chip design to be synthesized from an algorithmic description. The complexity gives a more important indication of how much longer a larger design will take to be synthesized. A synthesizer might operate quickly for small designs, but if its complexity is exponential and the circuits are large, it will probably be infeasible to compare design alternatives. Less complex synthesis algorithms facilitate faster synthesis of large designs, and so more design alternatives can considered by a designer.

The fifth chapter illustrates the synthesis method through some examples. Features of ALICS are demonstrated a few at a time. First some simple switching expressions are expressed in the context of an ALICS program. This shows how to declare operators and how to bind pre-defined circuitry with ALICS operators. Next the switching expressions are included inside conditional expressions. The third example illustrates the definition of an operator that will work on any length operand through the use of compile-time recursion. The next example shows the definition of an operand that makes the best use of available 2, 3, and 4-input nand gates in order to efficiently implement a reduction tree. Finally, the definition of new data types is illustrated with the definition of two types of adders. Simulation of the resulting circuitry allows trade-offs to be made between the two designs.

The concluding chapter summarizes the main ideas of the method, its implementation, and its key features. Several open questions and research topics are also identified.

CHAPTER 2

The ALICS Language

2.1 Relation of the Language to Hardware

This chapter discusses the relationship of the algorithmic language ALICS (Algorithmic Language for Integrated Circuit Specification) to the specification of hardware that is synthesized. ALICS is a programming language in which an abstraction of the desired digital behavior is specified.

The placement and routing of components on the target integrated circuit are not specified in the algorithmic language. Instead, this information is automatically derived from the specified behavior of each algorithm.

Each construct in ALICS has semantics that is either interpreted at compile-time or translated into circuitry which, when activated, implements the semantics of the constructs. The choice of which type of semantics depends not only on the individual constructs but on the actual values in the constructs.

2.2 Level of Description

ALICS is a means for expressing the digital behavior of the circuitry that will be built. ALICS supports hierarchical behavioral description. The lowest level of description is that of switching expressions and a form of state-machines. The highest level of description consists of the programmer-defined functions, similar to procedures defined in the programming language Algol 68. The levels of description result from applying the concept of modularity, which simplifies the specification of complicated programs. Modules are specified as function declarations. These facilitate the encapsulation of behavioral descriptions into units that are used as primitive functions in other functions, making these new functions easier for the designer to understand and use. This modularity, mapped into modules at the hardware level, is useful for designers and some analysis tools, but is not essential for the fabrication of the circuits.

The following sections discuss properties of the algorithmic language and of the target circuit architecture. A hardware designer might want to know about both of these if effective designs are to be produced. Knowledge of the algorithmic language is essential because the designs will be expressed in this language. A designer's knowledge of the target language is not as important as is proficiency with the algorithmic language, but, nonetheless, it is useful for the designer to know which types of circuitry are produced.

2.3 Algorithmic Language Features

The algorithmic language is the vehicle for specifying an algorithm that will be implemented by an integrated circuit. Before a designer can design an integrated circuit effectively, the capabilities of ALICS and the translator needs to be understood. The designer needs to know

- which kinds of algorithms can be expressed in the language,
- how to re-use previously-designed algorithms in the language,
- how good the resulting designs are, and
- the kinds of errors the system will catch.

The following features of a language determine which kinds of algorithms can be expressed in the language: data objects, data structuring constructs, operators, flow of control, flow of data, communication, and storage.

The major features of ALICS are presented next.

2.3.1 Data and Data Types

An important part of a hardware specification language is the provision for specifying the datatypes of the entities in a design. In ISP [Barb79], the primitive datatype is the bit. The structuring constructs provide a way of forming an array of bits (called words) and then forming an array of words. Unfortunately for designers, it is not possible to treat these structures as data types, and so an ISP parser cannot detect type mismatches, except for the ones in which the number of bits differ. Thus, assigning a 32-bit stack pointer to an 8-bit character register would be detected as an error. However, assigning a 32-bit stack pointer to the 32-bit program counter would not be flagged as an error, even though it is unlikely to be a meaningful operation. Loading the program counter with a 32-bit program address *addressed* by the stack pointer *is* a sensible operation.

These three examples *can* be checked by declaring the stack pointer and program counter to be of the appropriate types in Algol 68. Algol 68 [Wijn75] has four basic data types (or *modes*, as they are called in the Algol 68 Revised Report)—**integer**, **real**, **boolean**, and **character**. The Algol 68 Revised Report makes the assumption that Algol 68 programs will be run on machines that support these types. When custom hardware is designed, it is often advantageous to use various numeric representations rather than the standard two's complement **integer**. These will affect non-behavioral characteristics such as speed, area, power, testability, reliability, and fault tolerance. Therefore, it is assumed in ALICS that the supporting hardware does not provide these datatypes since custom hardware is designed for which there are no standard representations or sets of operators. Instead, a facility for the definition of new datatypes is provided.

Algol 68's provision for describing any datatype and the operators that can operate on that datatype has been incorporated into ALICS. Once a datatype has been defined and operators designed to manipulate the datatype, the operators can be placed in a library for use in other designs. So while the integer, character, and real datatypes are not built into ALICS, there is the ability to define them when needed, or recall them from a library if they have already been designed.

The types of all functions' actual and formal parameters are checked for consistency to ensure that the proper types of values are passed to all functions. For digital circuits, it is important not only that outputs are connected only to inputs, but that the output types match the input types. The first consistency check is specified in the syntax of ALICS and checked by its parser. Type-checking is the second semantic consistency check performed by the ALICS parser. For example, type-checking ensures that integer values are not passed to floating-point operators without proper conversion.

The Algol 68 Revised Report avoids discussing the representations for values. At a low-level of algorithm design, representation is of great importance. As mentioned before, a change in representation can result in operations that run faster, require less power or area, are more testable, reliable, or fault tolerant than a given implementation in two's complement (not all at once, of course). ALICS has two aspects to representing values, or constants. The first is how values are denoted. The second is how a value's representation is denoted.

Values are denoted as numbers in ASCII. The default radix is decimal, but can be changed by prepending the number with desired radix (expressed in decimal) and the character r. For example, the value 42 can be represented as 2r101010, 4r222, 8r52, 13r33, or 16r2a.

A value's representation is denoted as a list of the values that make up the representation. The target representation is in the domain of digital integrated circuits, where all values are groups of zeros and ones. This is represented in ALICS by the **BIT** datatype and the structuring mechanisms that enable more complex datatypes to be built (section 2.3.1). For example, the value 41 can be represented in radix arithmetic as **MODULUS**(1,2,1,6) with respect to the moduli (2,3,5,7), where the datatype called **MODULUS** can be defined to be a structure of four arrays of bits. Each of the residues is represented in binary. The first array is defined with one bit; the second, two bits; and the third and fourth, four bits each. Another possibility for a representation of the value 41 in radix arithmetic is MOD_OON(1, 2, 1, 6) with respect to the moduli (2,3,5,7), where the datatype called MOD_OON is defined to be a structure of four arrays of bits. Each of the residues can be represented in one-of-n code. The first array is defined with two bits; the second, three bits; the third, five bits; and the fourth, seven bits.

Defining New Datatypes

One of the most important features of this language is the support of extensible data-types. The designer can define new data types from existing data types through the use of one of four data type constructors—*arrays* (collections of homogeneous types, indexed by a numerical position) (also called *rows*), *sequences* (arrays whose elements are accessed monotonically sequentially), *structures* (collections of non-homogeneous types), *unions* (alternative types), and *functions* (executable code that take values from a set of types and returns a value of a given type).

2.3.2 Arrays

Elements in arrays are numbered both to provide access to the elements and to provide an ordering for the elements. The numbering provides access to the elements in the array in a logical fashion, whether numbering of the elements is to be from the right or the left. What is considered a *logical fashion* depends on what type of data the array is modeling and how the denotations for values are to look like. Array indices increasing from left-to-right are more logical when the arrays contain time or western reading-order dependent data. In arrays containing weighted numeric representations such as radix number systems, the same order is useful for fractional numbers, while the opposite ordering is more logical for integers. A specification method and associated semantics for both of these specification styles are presented in Appendix A.

2.3.2.1 Array Declarations

The syntax for an array declaration is shown in Figure 2.1. Where *variable* is

[*boundpair-list*]*MODE variable*

Figure 2.1: Syntax of an Array Declaration

the variable name; *MODE* is the mode (data type) from which this array is built; and *boundpair-list* is a comma-separated list of boundpairs, one for each dimension in the array. A *boundpair* is a pair of bounds for an array consisting of a left and a right bound. There is no restriction on which bound should be the greater of the two. If one of the bounds is omitted from a declaration, it is assumed to be 1. For instance, in the declaration [4,-3:5,2:-4]**BITS** x the first bound is from 1 to 4, the second from −3 to +5, and the third bound is from +2 to −4. Thus x is a three-dimensional array of 252 BITs ($4 \times 9 \times 7$).

2.3.2.2 Array Denotations

Arrays of values are denoted as a parenthesized list of values that are separated by commas, e.g., (a, **SKIP**, a **OR** b, **TRUE**) is a one-dimensional four-element array of bits. The second element in this array is non-existent. Multi-dimensional arrays of bits can be described by specifying a list of bound pairs. An example of a two-dimensional value is

$$((b, 4+3, 8), (3, 5, -7), (-4, 17, -2), (0, 0, a+1)).$$

2.3.2.3 Array Operators

The operators for array data types are now discussed. These consist of operators which extract attributes of an array (such as its bounds), operators for extracting elements from an array, and operators for constructing larger arrays from two arrays. The construction operators maintain the precision of the operands, as specified by the array bounds.

2.3.2.4 LEB Operator:

The left-bound operator is both a monadic and a dyadic operator. The dyadic version takes two operands. The first operand is the boundpair number (from the left, beginning at one), and the second operand is an array. The monadic version takes one operand, the array, and returns all the left bounds for the array. For the declaration [4,-3:5,2:-4]**BITS** a, the following relationships hold:

$$\textbf{LEB}\ a \;=\; (1,-3,2)$$

$$\begin{aligned} 1\ \textbf{LEB}\ \mathtt{a} &= 1 \\ 2\ \textbf{LEB}\ \mathtt{a} &= -3 \\ 3\ \textbf{LEB}\ \mathtt{a} &= 2. \end{aligned}$$

2.3.2.5 RIB Operator:

The **RIB** operator is similar to the **LEB** operator except that it returns the right-bound. For the declaration [4,-3:5,2:-4]**BITS** a, the following relationships hold:

$$\begin{aligned} \textbf{RIB}\ \mathtt{a} &= (4,5,-4) \\ 1\ \textbf{RIB}\ \mathtt{a} &= 4 \\ 2\ \textbf{RIB}\ \mathtt{a} &= 5 \\ 3\ \textbf{RIB}\ \mathtt{a} &= -4. \end{aligned}$$

2.3.2.6 LWB Operator:

The **LWB** operator returns the lower bound of an array. It is similar to the **LEB** operator except that it returns the smaller of the left and right bounds. For the declaration [4,-3:5,2:-4]**BITS** a, the following relationships hold:

$$\begin{aligned} \textbf{LWB}\ \mathtt{a} &= (1,-3,-4) \\ 1\ \textbf{LWB}\ \mathtt{a} &= 1 \\ 2\ \textbf{LWB}\ \mathtt{a} &= -3 \\ 3\ \textbf{LWB}\ \mathtt{a} &= -4. \end{aligned}$$

2.3.2.7 UPB Operator:

The **UPB** operator returns the upper bound of an array. It is similar to the **LWB** operator except that it returns the larger of the left and right bounds. For the declaration [4,-3:5,2:-4]**BITS** a, the following relationships hold:

$$\begin{aligned} \textbf{UPB}\ a &= (4,5,2) \\ 1\ \textbf{UPB}\ a &= 4 \\ 2\ \textbf{UPB}\ a &= 5 \\ 3\ \textbf{UPB}\ a &= 2. \end{aligned}$$

2.3.2.8 @ Operator:

The @ operator is called the revised lower-bound operator, and is a dyadic operator that creates a new copy of an array, changing only the lower-bound (not necessarily the left-bound). The corresponding upper-bound is also modified so that that there are still the same number of elements in the array. The first operand is the array whose bounds will be copied and modified, and the second operand is a one-dimensional array with as many entries as the first array has dimensions. If one of the revised lower-bounds is **SKIP**, the corresponding bounds is not modified. For the declaration [4,−3:5,2:-4]**BITS** a, a @(0,**SKIP**,0) has the bounds [0:3,−3:5,6:0].

2.3.2.9 Array Element Access:

Elements of arrays can be extracted by subscripting the array. Subscripts are enclosed in square brackets following an array and select single items from the array. If the integer array variable c has the value

((2, 4, 8), (3, 9, 27), (5, 25, 125), (7, 49, 343)),

then c[1,1] = 2, and c[4,1]= 7.

2.3.2.10 **REV Operator:**

The **REV** operator reverses the left and right bounds of an array. **REV** is both a dyadic and monadic operator. The dyadic version takes two operands. The first operand is the boundpair number, and the second is the array. The monadic version of the operator reverses all the boundpairs. For the declaration [4,-3:5,2:-4]**BITS** a, the following relationships hold.

REV a	has bounds	$[4:1, 5:-3, -4:2]$
1 **REV** a	has bounds	$[4:1, -3:5, 2:-4]$
2 **REV** a	has bounds	$[1:4, 5:-3, 2:-4]$
3 **REV** a	has bounds	$[1:4, -3:5, -4:2]$

2.3.2.11 **APPEND Operator:**

This operator provides an intuitive method for concatenating two arrays that represent numbers with weighted digits. The **APPEND** operator creates a new array by copying the second operand to the right of a copy of the first operand. The resulting array has a lower bound identical to the lower bound of the first operand. The new right bound is computed from the length of the new array extending in the direction of the boundpair of the first argument. If the direction of the second argument's boundpair is different from that of the first argument, the elements of the second array argument are copied in reverse order.

For example, two arrays can be appended as follows.

(1,0,2,3) **APPEND** (0,0,4)@89 = (1,0,2,3,0,0,4)

The array (0,0,4) (with bounds [89:91]) is appended to the right of array (1,0,2,3) (which has bounds [1:3]). The resulting array has the value $(1,0,2,3,0,0,4)$, which has bounds [1:7].

Two arrays representing numbers in a weighted digit number system can be appended as follows.

REV(1,0,2,3)@0 APPEND REV(0,0,4)@89 = REV(1,0,2,3,0,0,4)@−3

If the array represents a radix five number[1], the append operation represents appending 004×5^{89} to the right of 1023_5. 004×5^{89} is effectively scaled so that the high-order digit (0) will be in the 5^{-1} position and then added to 1023_5 to yield 1023.004_5.

The arrays in this example are (1,0,2,3) which has bounds $[3:0]$ and (0,0,4) which has bounds $[91:89]$. The value returned from **APPEND**, is $(1,0,2,3,0,0,4)$ which has bounds $[3:-3]$.

2.3.2.12 PREPEND Operator:

This operator provides an intuitive method for concatenating two arrays that might represent radix numbers. This is similar to the **APPEND** operator, except that the bounds of the second operand are preserved after the first operand is prepended to the left of the second operand.

The **PREPEND** operator creates a new array by copying the first argument to the left of a copy of the second argument. The resulting array has a right bound

[1]The particular radix is irrelevant to the **APPEND** operator.

identical to the right bound of the second argument. The new left bound is computed from the length of the new array extending in the direction of the boundpair of the second argument. If the direction of the first argument's boundpair is different from that of the second argument, the elements of the first argument array are copied in reverse order.

For example, two arrays, not necessarily representing radix numbers, can be prepended as follows:

$$(1,0,1,1)@0 \textbf{ PREPEND } (0,0,1)@89 = (1,0,1,1,0,0,1)@0.$$

The array (1,0,1,1), (which has bounds [0 : 3]), is prepended to the left of array (0,0,1), which has bounds [89 : 91]. The resulting array has the value $(1, 0, 1, 1, 0, 0, 1)$, which has bounds [96 : 102].

Two arrays representing radix numbers can be prepended as follows:

$$\textbf{REV}(1,0,1,1)@0 \textbf{ PREPEND REV}(0,0,1)@89 = \textbf{REV}(1,0,1,1,0,0,1)@{-3}.$$

If the array represents a radix five number[2], this represents prepending 1023_5 (131_{10}) to the left of 001×5^{89}. 1023_5 is effectively scaled by 5^{92} and then added to 001×5^{89} to yield $1023001_5 \times 5^{89}$.

The arrays in this example are (1,0,2,3), which has bounds [3 : 0] and (1,3,1), which has bounds [92 : 89]. The value returned from **PREPEND** is $(1, 0, 2, 3, 1, 3, 1)$, which has bounds [95 : 89].

[2]The particular radix is irrelevant to the **PREPEND** operator.

2.3.2.13 slicing:

Slicing is the process of extracting a contiguous portion of an array. (This is known as a *subrange* in some languages.) Slicing is indicated by a list of ranges within square brackets. A range consists of two optional integer expressions separated by a colon. If one (or both) of these expressions is missing, it is replaced by the corresponding left or right bound. The new left and right bounds will be in the same order that is specified in the slice. The bounds cannot be reversed if one of the bounds is missing in the slice range.

When a slice is taken from an array, the boundpair is not adjusted automatically so that it has a revised lower bound of 1 if a revised lower bound is not specified[3].

A slice accesses a contiguous portion of an array. The middle three elements of the array (2,3,5,7,11) can be selected with the operation (2,3,5,7,11)[2:4], yielding the value $(3,5,7)$. This value is an array which has bounds $[2:4]$, so that the operation (2,3,5,7,11)[2:4][3] will yield the value 5. The operation (2,3,5,7,11)[4:2] yields the value (3,5,7), which has the bounds $[4:2]$. If a bound in a slice is missing, the corresponding left or right (or both) bound is (are) used in its place. Thus, (2,3,5,7,11)[4:] yields the value $(7,11)$, which has bounds $[4:5]$.

[3]The lower bound of a slice is automatically changed to 1 in Algol 68, but seems arbitrary. This has the effect of extracting the slice and performing a numeric shift on the resulting value. This malformation can be accomplished in ALICS with the revised lower-bound operator @, defined here, if desired. In Algol 68, maintaining the array indices in a slice is awkward for the programmer because the indices need to be specified twice—once in the slice and again in the revised lower-bound.

2.3.3 Sequences

Sequences provide a way to access elements of an array sequentially. The addition of time sequences to the functional language FP was discussed in [Mesh85].

Eight operators convert between arrays and sequences. The first four are necessary to access the two types of array organization. The remaining four operators can be derived from the first four, but are provided as a convenience for the designer. Four operators convert between arrays and sequences. Their names reflect the four ways to choose the first element of a sequence—leftmost, rightmost, lowest index, or highest index. The four sequence-to-array operators reflect the four ways to map the first element of a sequence into an array. This element can be either the lowest numbered or highest numbered element of the array, *and* the element can be in the leftmost or the rightmost position of the array.

PISOLE This monadic operator converts an array to a sequence, beginning with the lowest numbered element. (Parallel in, serial out, leftmost element first.)

PISORI This monadic operator converts an array to a sequence, beginning with the rightmost element. (Parallel in, serial out, rightmost element first.)

SIPOLOLE This monadic operator converts a sequence to an array. The first element of the sequence is the first in the sequence and becomes the leftmost array element. The left bound of the array is 1. (Serial in, parallel out, lowest index first becomes leftmost element.)

SIPOHILE This monadic operator converts a sequence to an array. The elements of the sequence are assumed to arrive in reverse order, that is, last element (highest index) first. This last element of the sequence becomes the leftmost array element. The right bound of the array is 1. (Serial in, parallel out, highest index first becomes leftmost element.)

PISOLO This monadic operator converts an array to a sequence, beginning with the lowest numbered element. (Parallel in, serial out, lowest numbered element first.) This operator is equivalent to

```
IF LEB a < RIB a
THEN PISOLE a
ELSE PISORI REV a
FI
```

PISOHI This monadic operator converts an array to a sequence, beginning with the highest numbered element. (Parallel in, serial out, highest numbered element first.) This operator is equivalent to

```
IF LEB a < RIB a
THEN PISORI REV a
ELSE PISOLE a
FI
```

SIPOLORI This monadic operator converts a sequence to an array. The first element of the sequence is the first in the sequence and becomes the rightmost array element. The right bound of the array is 1. (Serial in, parallel out, lowest index first becomes rightmost element.) This operator is equivalent to

REV SIPOHILE a.

SIPOHIRI This monadic operator converts a sequence to an array. The elements of the sequence are assumed to arrive in reverse order, that is, last element (highest index) first. This last element of the sequence becomes the rightmost array element. The right bound of the array is 1. (Serial in, parallel out, highest index first becomes rightmost element.) This operator is equivalent to

REV SIPOLOLE a.

Figure 2.2 shows the bit-by-bit application of the **OR** function on two arrays. The first line checks (at compile time) that the bounds of a and b are the same.

```
IF LWB a = LWB b AND UPB a = UPB b
THEN PAR i FROM LWB a TO UPB a
        DO a[i] OR b[i] OD
FI
```

Figure 2.2: Parallel Application of **OR** to Two Bit Arrays

If there are n elements in each array, n **OR** operators will be instantiated for execution in parallel.

The parallel to serial and serial to parallel operators just introduced can be used to reduce the number of operator instantiations at the expense of longer execution time as is shown in Figure 2.3. The **PISOLE** operator converts a and

```
SIPOLOLE ((PISOLE a) OR (PISOLE b))
```

Figure 2.3: Serial Application of **OR** to Two Bit Arrays

b from parallel arrays to sequences. Each pair of elements in these sequences is input to the **OR** operator, and a new sequence is produced. This sequence is input to the **SIPOLOLE** operator which converts it to an array of the same size of a and b.

2.3.4 Structures

The second data structuring technique names and collects non-homogeneous types so that they can be manipulated as a single entity. Access to an individual field in a structure is through the name of the field. For example, a floating point number could be represented by the structure definition

STRUCT (XS exponent; FRAC mantissa),

where XS is the type for the exponent and FRAC is the type for the mantissa (presumably, XS and FRAC are defined elsewhere). A structured value is denoted by the name of the structure type followed by a parenthesized list of comma separated fields. For example, the data type (mode) definition,

MODE FLOAT = **STRUCT** (XS exponent; FRAC mantissa)

treats the value **FLOAT** (ex1,frac) as a structure of type **FLOAT**. The **OF** operator takes a field name and a structure and yields the value of that field. So

exponent **OF FLOAT** (ex1,frac)

yields the value of ex1.

2.3.5 Unions

A union specifies that a formal parameter's value may be one of several data types. The declaration

FUNCTION foo = (**UNION** (**FLOAT**, **INTEGER**)parm)

states that the actual parameter to the function foo may be either **FLOAT** or **INTEGER**.

The type of a union is determined through the application of a *conformity clause.*

```
CASE parm IN
        (FLOAT float_parm): f1(float_parm),
        (INTEGER integer_parm): f2(integer_parm)
ESAC
```

This construction examines the data type of parm. If it is a **FLOAT**, then parm's value is associated with float_parm and the function f1 is invoked with this value. If it is an **INTEGER**, then parm's value is associated with integer_parm and the function f2 is invoked with this value.

2.3.6 Operators and Functions

There are three types of functions in ALICS—prefix monadic operators which take one argument, dyadic (properly called infix dyadic) operators which take two arguments, and functions which take zero or more arguments. The difference between operators and functions is entirely syntactic. Operators are used in infix formulae in which the relative priorities and associativities of the operators determine the structure of the computation. Operators are syntactically less powerful than functions, since operators can have only one or two operands. Operators are provided as a convenience to designers who prefer to read and write infix formulae. The parentheses in a functional expression determine the computational order. Thus an infix formula such as

a **NAND NOT** b **NOR** c,

is equivalent to the function description nor2(nand2(a,not(b)),c)

Functions and operators provide several capabilities in ALICS—most notably, modularity and replication of values. Modularity is important because it allows a reduction of the amount of code that needs to be written and the resulting programs are more easily understood. Commonly used code can be written once and then invoked as a function (module) in many places.

The second use for a function or operator is to replicate values. Functional languages do not allow simultaneous access to components of a composed value. For example, functional languages do not allow the value (a,c) to be created from a composed value such as (a,b,c) unless the composed value is passed to a function. Inside the function the value is referenced by its formal parameter name (number in FP) multiple times and the desired value computed and returned as the value of this function.

The syntax of a function is shown in Figure 2.4. In this figure *name* is the

FUNCTION *name* (*parameter-list*) *function-type/mode*:
BEGIN *function-body*
END

Figure 2.4: Syntax of a Function Definition

name of the function; *parameter-list* is the list of formal parameter declarations; *function-type/mode* is the mode, or type, of the value of the function; and *function-body* is the body of the function whose value is yielded as the value of the function. A variation on this syntax allows **BEGIN** to be written as (and **END** as).

The syntax of an operator is shown in Figure 2.5. In this figure *name* is the

OP *name* = (*parameter-list*) *operator-type/mode*:
BEGIN *operator-body*
END
associativity name-list;
PRIO *name-list* = *priority*;

Figure 2.5: Syntax of a Operator Definition

name of the function—which is formed from uppercase alphabetic or a symbol formed from the characters `+-*/<>%&!~?=:|`. *parameter-list* is the list of one or two formal parameter declarations; *operator-type/mode* is the mode, or type, of the value of the operator; and *operator-body* is the body of the operator whose value is yielded as the value of the operator. A variation on this syntax allows **BEGIN** to be written as (and **END** as). The associativity of operators is declared as **LEFTASSOC**, **RIGHTASSOC**, or **NONASSOC**. The **PRIO** declaration is used for assignment of a *priority* (also known as precedence) of one to nine to a of list of operators. The higher the number, the higher the priority.

Operators have an additional property that functions do not have. An operator symbol can be *overloaded*, as in Algol 68 or Ada [Ada83], to provide *multiple entendres* for the symbol. This means that, for instance, the operator symbol plus-sign can be simultaneously associated with an operator that performs integer addition, and with one that performs floating-point addition, and with one that performs some other function on some other type of arguments. Which of these operators is actually invoked depends on the types of the arguments used in conjunction with the operator symbol.

2.3.7 Flow of Control

Several types of flow of control (sequencing of operations) are available in ALICS: sequential, parallel, conditional, and limited forms of iteration and recursion.

Control flows sequentially through composed functions. In the expression f(g(x)), g(x) executes first. The value returned by **g** is then passed to the function f. In the expression a+b*c, execution is sequential, beginning with the multiplication and concluding with the addition, according to the relative priorities of these two operators.

Parallel execution occurs in parallel clauses. Parallel clauses yield a vector of values, which may appear as an input to a function. For example, in the function call f(a+b, c+d), both additions are performed in parallel. The two resultant values are passed to the function f.

Functions are expanded at compile-time. A separate function is created on the chip for each instance in the functional specification. This is not always an optimum approach since some hardware might be idle part of the time.

There are two types of iteration—parallel and sequential. Parallel iteration occurs at compile time, replicating the iteration body. The bounds of a parallel iteration must be determined at compile time. Sequential iteration specifies repeated execution of the iteration body at run time. The sequential iteration construct specifies a state machine. User-defined variables comprise the state, which can be changed during initialization of the state and at the end of each iteration. The state changes and iteration output are computed from user-defined functions.

Only a limited form of recursion is handled by this method. Recursion is performed by interpreting functions at compile-time. Since chips have bounded area, recursions require fixed points that are calculable at compile time. This is not powerful enough for general recursive computations, but as is shown in Chapter 5, is useful in generic functions (functions that return varying-length results).

Storage is available only as part of the state in the ALICS sequential construct.

2.3.7.1 Conditionals

There are two types of conditionals—one is based on a Boolean value, the other on an integer value. The first is the **IF** expression, whose syntax is shown in Figure 2.6, in which zero or more **ELIF–THEN** parts may be used. The semantics

```
IF a
THEN b
ELIF c
THEN d
ELSE e
FI
```

Figure 2.6: Syntax of an **ELIF** Clause in an **IF**-Expression

of this expression call for the Boolean expression a to be evaluated. If it evaluates to TRUE, then expression b is evaluated and yielded as the value of the **IF**. If it evaluates to FALSE, then the Boolean expression c is evaluated. If it evaluates to TRUE, then expression d is evaluated and yielded as the value of the **IF**. If expressions a and c are both FALSE, then expression e is evaluated and yielded as the value of the **IF**.

The second type of conditional statement is the **CASE** expression. Figure 2.7

shows its syntax. The **CASE**'s selector expression, sel, is evaluated to yield a value

```
CASE sel IN
[1]: a,
[2]: b,
[4]: c,
[8]: d
OUSE e
ESAC
```

Figure 2.7: General Form of a **CASE** Expression

which is used as an index into the list of expressions that follow. Each expression in the list has an associated constant expression label. If there is an expression label matching the selector expression, then the associated expression is evaluated and yielded as the value of the **CASE** expression. If there is no expression label that matches the selector expression, then the expression following the **OUSE** keyword is evaluated and yielded as the value of the **CASE** expression.

2.3.7.2 Iteration

There are two types of iteration in ALICS—one for iteration in space and one for iteration in time. Iteration in space is specified with a parallel expression. This construct specifies parallel evaluation of an expression, that is, the expression is replicated and simultaneously evaluated. The syntax for a parallel expression is shown in Figure 2.8. This expression expands to $abs(\mathsf{a} - \mathsf{b}) + 1$ parallel calls to

```
PAR i FROM a TO b BY ±1
DO f(i) OD
```

Figure 2.8: General Form of a Parallel Expression

the function f, as shown in Figure 2.9. The yield is one of two possible arrays,

```
IF a < b
THEN (f(a), f(a+1), ..., f(b))@a
ELSE  REV(REV(f(b), f(b+1), ..., f(a))@a)
FI
```

Figure 2.9: Expansion of a Parallel Iteration

depending on whether the left bound is greater than or less than the right bound. The left bound of both arrays is a, and the right bound of both arrays is b, corresponding to the **FROM** and **TO** parts of the **PAR**.

In Figure 2.10 a parallel iteration is used to form a permutation of an array. The subscript in the body of this loop is calculated by the function f. The bounds

```
PAR i FROM LWB x TO UPB x
DO x[f(i)] OD
```

Figure 2.10: Parallel Iteration for a Permutation

of the iteration are **LWB** x (the lower-bound of x) and **UPB** x (the upper-bound of x). Assuming that f(i) returns integers within the bounds of the array x, x[f(i)] will permute element $x_{f(i)}$ to element x_i. For example, if the bounds of x are zero through four then the iteration in Figure 2.10 expands to the clause shown in Figure 2.11. This requires five copies of the function f and five copies of the decoding logic. If

```
(x[f(0)], x[f(1)], x[f(2)], x[f(3)], x[f(4)])
```

Figure 2.11: Expanded Parallel Iteration for a Permutation

the function f(i) is defined as $(3\times i) \bmod 5$, then the the function can be evaluated at compile time. Then the iteration in Figure 2.10 expands to the clause shown in Figure 2.12. This requires no copies of the function f, and the permutation is

```
(x[0], x[3], x[1], x[4], x[2])
```

Figure 2.12: Expanded Parallel Iteration for a Permutation

performed as a permutation route at compile time, requiring no decoding logic.

2.3.7.3 Sequential Iteration

Meshkinpour and Ercegovac discuss additional forms for FP that allow sequential systems to be designed [Mesh85]. A generalization of these sequencing forms allows the specification of sequential systems in the imperative language ALICS.

The general form for a sequential iteration is shown in Figure 2.13. The number

```
FOR var SEQ seq
DO state-initialization;
    state ← g(var, state)
OUT h(var, state)
OD
```

Figure 2.13: General Form of a Sequential Iteration

of iterations in this construct is the same as the number of elements in the sequence *seq*. During the first iteration, the *state-initialization* code is executed, and the variable *var* takes on the value of the first element from the sequence, *seq*. In the second and successive iterations, a state-transition function (**g** in the example) of *var* and the state is executed. After each iteration the output of this function is assigned to state variables, and the function h of *var* and the state variables is output. Variations of this form are now given to demonstrate this construct's ability to specify a state machine, a Moore machine, and a Mealy machine.

A sequential iteration for a state machine is shown in Figure 2.14. This differs

```
FOR var SEQ seq
DO state-initialization;
    state ← g(var, state)
OUT state
OD
```

Figure 2.14: Specification of a State Machine

from the general form in that the output is the current state (or part of it). The number of iterations in this construct is one plus the number of elements in the sequence *seq*. Before the first iteration the *state-initialization* code is executed. Before each iteration, the variable *var* takes on successive values from the sequence, *seq*. After each iteration, assignment to state variables is performed and the state is output.

A Moore machine is a sequential finite-state automaton (machine) whose output depends only on the current state [Moor55]. The general form of a Moore machine is shown in Figure 2.15.

```
FOR var SEQ sequence
DO state-initialization;
    state ← g(state, var)
OUT h(state)
OD
```

Figure 2.15: General Specification of a Moore Machine

A Mealy machine is a sequential finite-state automaton (machine) whose output depends on the current state and the current input to the automaton [Meal55]. The general form of a Mealy machine is illustrated in Figure 2.16.

A least-significant-bit first binary adder can be defined with the Moore machine

```
FOR var SEQ sequence
DO state-initialization;
    state ← g(state, var)
OUT h(state, var)
OD
```

Figure 2.16: General Specification of a Mealy Machine

described in Figure 2.17. First a new data type consisting of a sum and a carry

```
MODE CARRY_SAVE = STRUCT (BIT carry, sum);
SIPOLORI
FOR pair SEQ []BIT(PISOLE a, PISOLE b)
DO CARRY_SAVE psum ← (0,0);
    psum ← full_add(pair[0], pair[1], carry OF psum)
OUT sum OF psum
OD
```

Figure 2.17: Specification of a Moore Machine Ripple-Carry Adder

is defined with the **MODE** statement. The first function, **SIPOLORI**, converts the output of the **FOR** from a sequence of *sum* bits into an array that represents the sum (except for the carry). The arrays a and b are converted to sequences (least-indexed element first) with **PISOLE** operators and then combined into a two-element array of sequences. The **FOR** loop assigns each element in succession to the variable named pair. This variable (there could be more) is the *state* required for a Moore machine. The body of the loop declares the variable psum. The declaration includes the initialization of the variable. This assignment occurs only once, before the first iteration of the loop body. In each iteration the result from the full_add function is assigned to this variable. If there is more than one assignment in the loop body, all the assignments take effect simultaneously at the end of the current iteration. After the assignment(s), the state is available for output. In this case, only a portion of the state (the sum) is output.

An adder for binary numbers implemented as a Mealy machine is illustrated in Figure 2.18. This adder operates almost exactly the same way as the Moore

```
SIPOLORI
FOR pair SEQ []BIT(PISOLE a, PISOLE b)
DO BIT carry ← 0;
    carry ← nand3(pair[0] NAND pair[1],
                  pair[0] NAND carry,
                  pair[1] NAND carry)
OUT xor3(pair[0], pair[1], carry)
OD
```

Figure 2.18: Specification of a Mealy Machine Ripple-Carry Adder

machine adder just described. The major difference (from the standpoint of the difference between Mealy and Moore machines) is that the output consists of a function of the state as well as the current element from the loop's sequence (pair). The minor differences are that the state consists of only one bit and that the full_add function has been expanded.

2.3.8 Flow of Data

The structures of programs written in ALICS correspond almost directly to the topological structures that implement the programs on chips. Data flows vertically through a program and the chip that implements it. In the non-self-timed implementation, the circuits are clocked with a master clock. The clock rate is a function of the delays between the inputs, the latches in the sequential iterations, and the outputs.

2.3.9 Communication

Communication with all functions is through their inputs and outputs. The inputs and outputs of the outermost function are connected to the I/O pins of the chip implementing the function. The outermost function is written as a function without an actual parameter list. The actual parameters will be the inputs to the chip. Since an output can be of any data type, structures can be defined to represent complex outputs from a function/chip.

2.3.10 Semantics

The language attributes discussed determine the types of algorithms that can be expressed in ALICS. A good language and recognizer for that language not only needs expressive features, but also well-defined, straightforward semantics [Schw78]. These are essential if a programmer is to be able to write and understand algorithms in ALICS. The ALICS language has several features that aid this. The language has units that are functional and interchangeable. In the same position that a constant can occur, so can a function parameter, a conditional, a function application, or any unit that yields the proper type of value for that position. To avoid repetitive descriptions, ALICS encourages hierarchical function definitions.

2.4 Target Language Features

This section lists some features of the target language.

2.4.1 Operators

Operators are implemented by predefined cells that are stored in a library. The library contains the mask data and the ports locations for all the cells. An innovative tree-structured method distributes power and ground to the cells when they are laid out in a hierarchical method. Power and ground run vertically along the left and right sides of each cell. The output and input ports are available on the top and bottom of each cell.

2.4.2 Flow of Control

The method for flow of control is independent of the synthesizer as it depends on the type of cells in the synthesizer's library By changing libraries, it is possible to change the method of flow of control. The current library is made from combinational logic. It is possible to make another from self-timed combinational logic.

The current version of the cells is implemented in combinational logic. Control and data flow through operators is as fast as the logic executes. The flow of control operators **IF** and **CASE** are also implemented in combinational logic as multiplexors and decoders. The **IF** and **CASE** operators select one of their operands based on the value of a clause. The delays through these operators need to be as long as the longest delay in each of the sub-clauses. The delay for a parallel clause must be as long as the longest of its subclauses. Each of the clauses in a parallel clause must complete for the parallel clause to complete. Since these are implemented in combinational logic, the maximum delay of all the clauses needs to be determined for this to operate properly inside a clocked system.

If a self-timed library is constructed, the flow of data and the particular data values will control the execution of the cells. The time delay through conditional circuits will depend on which paths are activated by the particular data.

2.4.3 Flow of Data

In both a combinational and self-timed implementation, the flow of data is direct, from operator inputs through the operator body directly to the operator output. The flow of data might be inhibited by conditional operators that do not select all of their sub-clauses.

2.4.4 Data Objects and Structuring

Data objects are either primitive ones built into the synthesizer or defined from the primitive ones and other defined data types. The primitive data type in ALICS is that of Boolean values. New data types are formed from existing ones in one of four ways—arrays, sequences, structures, and unions. An array is a numerically-indexed collection of identical data types. A sixteen-bit integer can be represented as an array of Boolean values. A sequence is an array whose elements are available sequentially rather than simultaneously. A structure is a name-indexed collection of not necessarily identical data types. A sixteen-bit sign-and-magnitude integer can be represented as a structure that contains a Boolean value for the sign and an integer for the value. A union is a data type that may be one of several types, but only one at a given time. For example, a union may specify that a certain value can be of type integer or real, but nothing else.

2.4.5 Communication

Communication with a chip is through its input and output ports. In the combinational implementation, the delay through the circuit is computation-dependent. In the self-timed implementation, inputs are presented to the input ports. When all the data has been presented to the chip, the external *input-data-ready* signal is strobed. The data then flows through the chip. When the outputs have been computed, a *output-data-ready* signal is made active. Data-acknowledgment signals in and out of the chip indicate when the chip is ready for inputs.

2.4.6 Storage

Storage is available in a limited manner in sequential loops External storage can be connected between the inputs and the output of a function, but the description of the storage is external to ALICS.

2.4.7 Technology-Dependent features

The synthesizer knows about many technology-dependent features. These are necessary so that the synthesizer can produce layouts in a given technology. Chisel [Karp83] operates in a fabrication technology independent manner by providing to the designer functions that are independent of the given technology. This is done by providing technology files that describe the particular technology. These files describe the design rules for the particular technology. These rules describe the minimum widths and separations for the layers in the technology. Figure 2.19 shows a representative set of the rules for the scalable CMOS technology (scmos). Lines that begin with percent signs are comments. The first non-comment line declares the basic unit of measurement, lambda, as 150 centi-microns. Next, fifteen layers are defined. Each layer definition contains a descriptive name for the layer and the CIF name of the layer. (CIF names are restricted to four characters in version 2.0.) The next four lines declare the minimum widths (in lambdas) of the individual layers. After this are the specifications of minimum widths for various combinations of layers. Following this are the minimum separations (in lambdas) between different layers. If there is no stated separation between any given pair of layers, they can abut without interacting. Finally, the lines that begin with Ext specify how far a layer has to extend past some other combination of layers. These rules include the polysilicon and diffusion extensions past transistors and extensions around cuts between layers.

More information is needed by the synthesizer to perform its task. The following information has been added to the technology files and additional subroutines written to access the information. The example lines shown are for the scalable CMOS technology (scmos).

- To perform routing, the synthesizer needs to know which layers are available for routing. Figure 2.20 shows the entries in the scmos file that specify the

```
% process "scmos"
Lambda 150

Layer DCUT   CCA   Layer DIFF   CAA   Layer GLASS  COG
Layer METAL1 CMF   Layer METAL2 CMS   Layer NDIFF  CND
Layer NSEL   CSN   Layer NWELL  CWN   Layer PAD    XP
Layer PCUT   CCP   Layer PDIFF  CPD   Layer POLY   CPG
Layer PSEL   CSP   Layer PWELL  CWP   Layer VIA    CVA

Width DCUT   2     Width GLASS  40    Width METAL1  3
Width METAL2 3     Width NDIFF   2    Width PAD    50
Width PCUT   2     Width PDIFF   2    Width POLY    2
Width VIA    2

Width METAL1+PCUT 2       Width METAL1+DCUT 2
Width METAL1+POLY+PCUT 2  Width METAL1+NDIFF+DCUT 2
Width METAL1+PDIFF+DCUT 2 Width METAL1+METAL2+VIA 2

Sep POLY POLY        2   Sep PDIFF NDIFF     8
Sep POLY NDIFF       1   Sep POLY PDIFF      1
Sep NDIFF NDIFF      3   Sep PDIFF PDIFF     3
Sep METAL1 METAL1    3   Sep METAL2 METAL2   4
Sep POLY NDIFF+DCUT 2   Sep POLY PDIFF+PCUT 2
Sep NDIFF POLY+DCUT 2   Sep PDIFF POLY+PCUT 2

Ext POLY+NDIFF           POLY  end 2 side 0
Ext POLY+PDIFF           POLY  end 2 side 0
Ext POLY+NDIFF           NDIFF end 0 side 2
Ext POLY+PDIFF           PDIFF end 0 side 2
Ext METAL1+NDIFF+DCUT    METAL1 1
Ext METAL1+PDIFF+DCUT    METAL1 1
Ext METAL1+NDIFF+DCUT    NDIFF  1
Ext METAL1+PDIFF+DCUT    PDIFF  1
Ext METAL1+POLY+PCUT     METAL1 1
Ext METAL1+POLY+PCUT     POLY   1
Ext METAL1+GLASS         METAL1 4
```

Figure 2.19: Chisel SCMOS Technology File

routable layers.

```
Route NDIFF
Route PDIFF
Route POLY
Route METAL1
Route METAL2
```

Figure 2.20: Additional Chisel Parameters for Routable Layers Specification

- To avoid metal migration in too-small wires, the maximum current per cross-sectional area, thickness, and minimum width must be available for routable layers. Figure 2.21 shows the chisel statements needed for the metal migration calculations.

```
% microamps per square centimicron (floating point)
MaxCurrent METAL1 .1
MaxCurrent METAL2 .1

% thickness of the layers in centimicrons
Thickness METAL1 100
Thickness METAL2 100
```

Figure 2.21: Additional Chisel Parameters for Metal Migration Specification

- For placement, the minimum layer width and separation between layers must be known. This information is already present in the Chisel technology file.

- The per-layer boundary for each cell needs to be known. This information is not currently available. It is approximated by the maximum bounding box for all the layers in the cell.

- The location, layer, width, and connection direction for all cells' ports must be known. This information is present in the Chisel cell library.

CHAPTER 3

Synthesis

3.1 Background

The ALICS Synthesizer converts ALICS programs into the integrated circuit mask description language CIF (Caltech Intermediate Format) [Hon80,Mead80]. The main steps for this process are shown in Figure 3.1. The large dashed box contains the parts of the ALICS Synthesizer. The solid boxes in the figure represent executable programs or subroutines. The ellipses represent data or data structures. The dashed arrows represent flow of control and the solid arrows represent the flow of data. We now discuss in detail each block of the ALICS Synthesizer.

3.2 Source Language Analysis

The analysis of the ALICS source language is performed in three steps—lexical analysis, the grouping together of characters into the vocabulary of the language; and then parsing, the grouping of this vocabulary into expressions in the language; and the transformation of these expressions into a structural representation of the program. This structure is mapped onto a tree of interconnected cells that implement the functions in the program.

This chapter discusses a method for traversing this structure only once in order

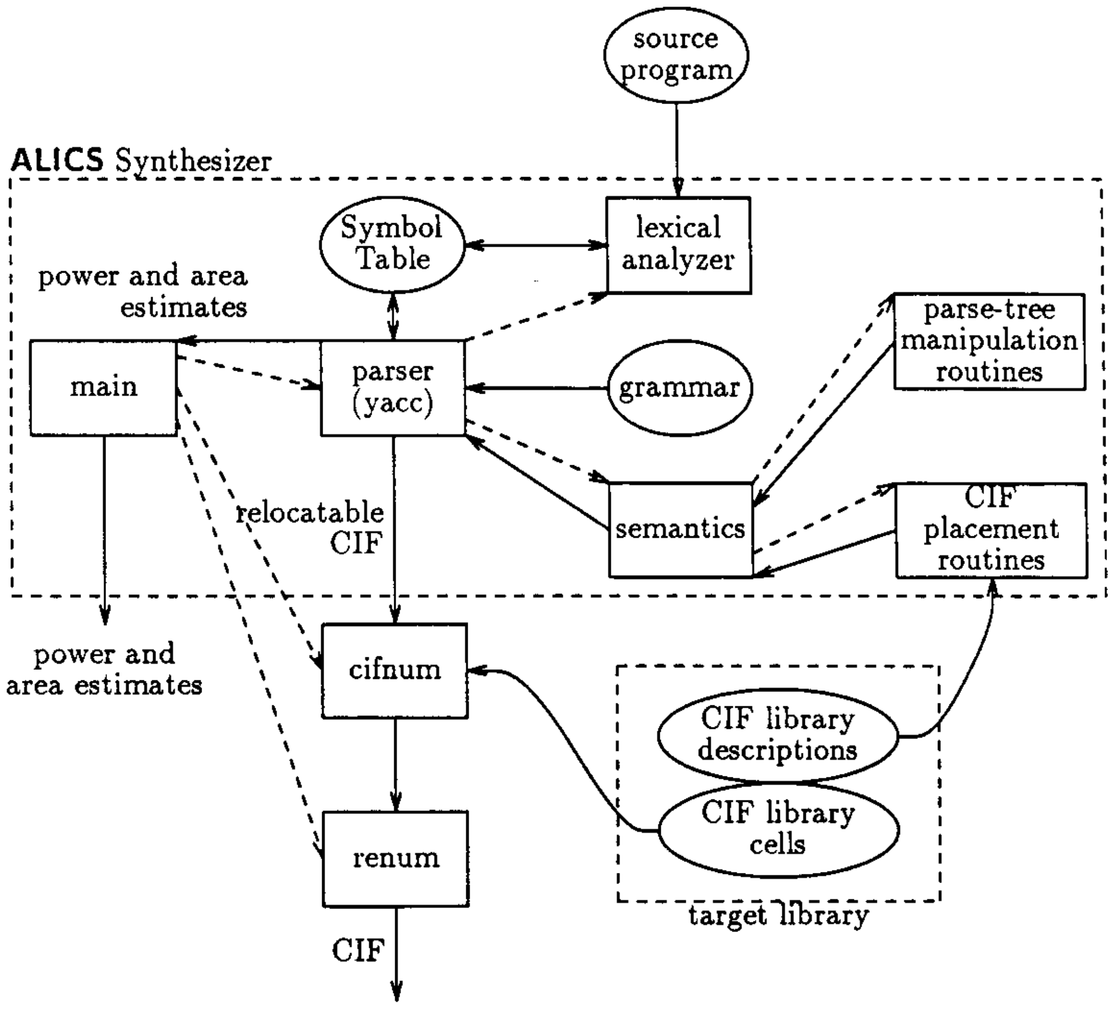


Figure 3.1: Flow of Data and Control in the ALICS Synthesizer

to determine the placement of cells, their interconnections, and the distribution and sizing of power and ground wires to those cells. This algorithm makes use of local information as the structure is traversed, reducing the amount of information that the compiler needs to consider at any given time.

3.2.1 Lexical Analysis

The characters in the ALICS source program are read by a lexical analyzer [Lesk75]. Groups of characters are recognized by the lexer and represented by a much shorter sequence of tokens. Each "key word" in the language is represented by a unique token. Each "identifier" or "tag" in a program is represented by a **tag token**, which contains a pointer to an entry in a symbol table. An example program fragment is shown in Figure 3.2. The stream of tokens produced by the

```
a * b + (c*d +e)
```

Figure 3.2: ALICS Program Fragment

lexer for this program is shown in Figure 3.3. There are fifty-nine characters in

```
tag token (a)
left operator 3 (*)
tag token (b)
left operator 2 (+)
open token ('(')
tag token (c)
left operator 3 (*)
tag token (d)
left operator 2 (+)
tag token (e)
close token (')')
```

Figure 3.3: Tokens Representing the Program Fragment in Figure 3.2

```
tag token (a)
left operator 3 (*)
tag token (b)
left operator 2 (+)
open token ('(')
tag token (c)
left operator 3 (*)
tag token (d)
left operator 2 (+)
tag token (e)
close token (')')
```

Figure 3.3: Tokens Representing the Program Fragment in Figure 3.2

rules. Figure 3.4 shows the parse tree for the sample program. This began as a simple program and is now a complicated graph. The graph shows the structure of the formula as a tertiary tree. The left and middle parts of the the graph are easy to understand. The right sub-formula is more complicated because of the parenthesized sub-expression. Each level in the graph represents the recognition of a grammar rule and its application to the rules that have been parsed below it. The linear sequence of grammar rules appears unnecessary in the parse of the program fragment, but each level indicates that other expressions would be possible. For instance, the line connecting collateral clause to enclosed clause indicates that this enclosed clause is made up of a collateral clause. The existence of this line also indicates that (in general) an enclosed clause might be made up of other entities. The grammar rule for an enclosed clause is shown in Figure 3.5. This states that an enclosed clause is either a choice clause, a collateral clause, or a loop clause. A choice clause is either an **IF**-expression or a **CASE**-expression. A collateral clause is a list of one or more units which execute collaterally. A loop clause is a loop

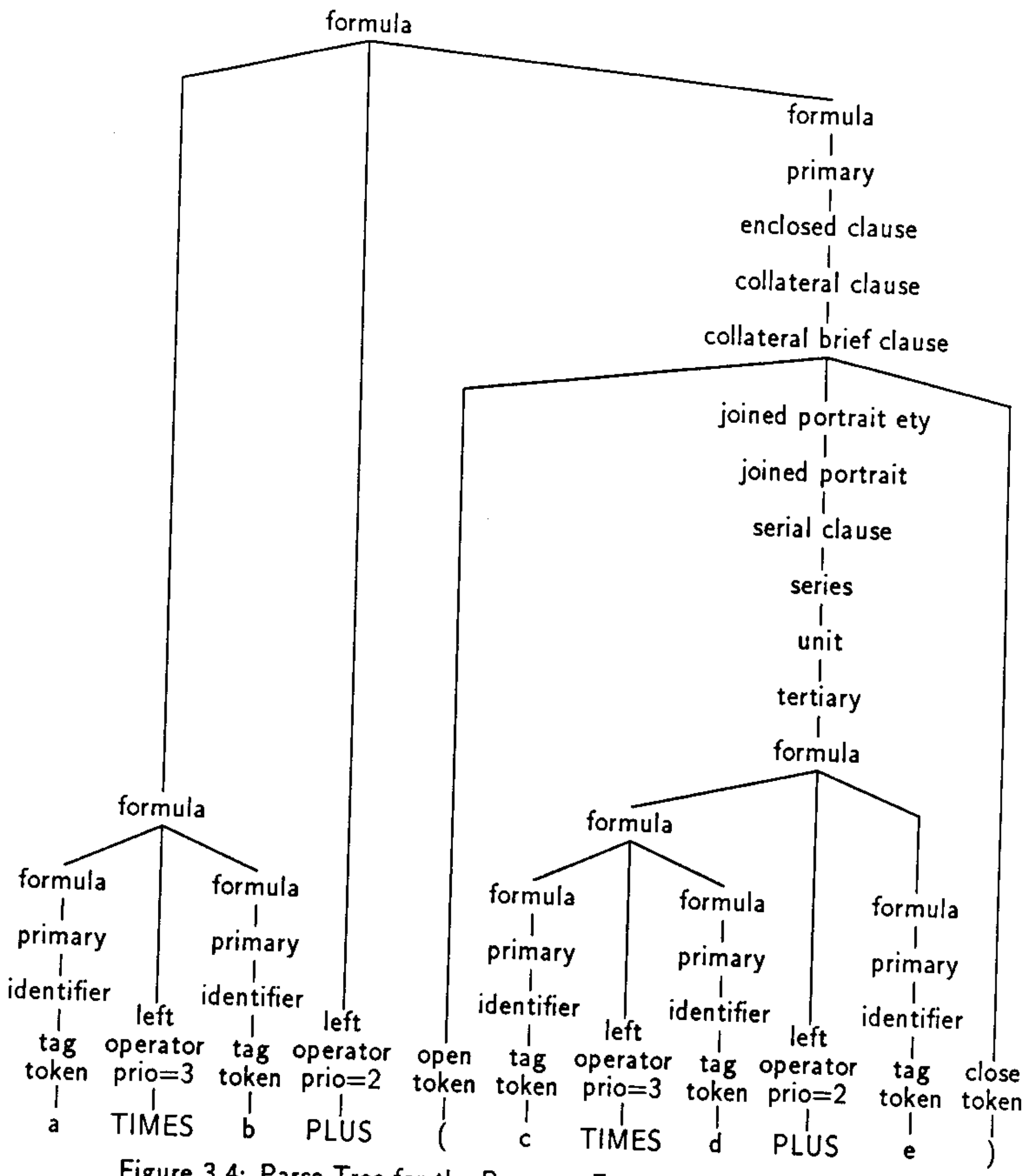


Figure 3.4: Parse Tree for the Program Fragment in Figure 3.2

```
enclosed clause ::= choice clause;
                    collateral clause;
                    loop clause.
```

Figure 3.5: Grammar Rule for an enclosed clause

whose bounds are known at compile-time and whose loop body instantiations are executed collaterally. This program fragment corresponds to the grammar rule for a collateral clause.

3.2.3 Determining Program Structure

The parser in the ALICS Synthesizer does not need to produce an actual parse tree. Instead it produces a simpler form, a *structure tree*, which is related to a parse tree. This tree represents the hierarchical (tree) structure of that program. The grammar rules serve as templates for the recognition of ALICS programs. Each time a rule is used in the parse of a particular program it is placed in a directed graph to indicate where it is used in relation to the other grammar rules. In syntax-directed parsing, when a grammar rule is recognized, the semantic routine for that rule is executed. Each semantic rule applies only to the elements of the corresponding grammar rule and so the effects of the semantics are localized. In this case the semantics are designed to produce the structure tree, usually with one semantic rule for each node in the tree.

For instance, the grammar rule for the top node in the parse tree is

formula ::= formula, left operator 2, formula.

The semantics for this rule create the node shown in Figure 3.6. The node contains fields for each of the parts of this grammar rule. There is one field that identifies the node as a formula, pointers to the node for the operator, and the two for the formulae. The tokens in the rule do not need to be represented in the node. They

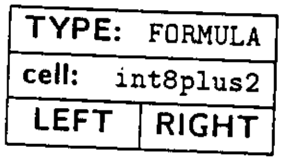


Figure 3.6: Structure Tree Node for a Formula

serve to separate the parts in the source program and to aid in parsing. A node for the collateral brief clause does not contain entries for the open token or the close token.

Each node in the structure tree is created by the semantics for a grammar rule for each sub-node. Each semantic rule is designed to return a pointer to the node that it creates. So the semantics for an enclosed clause needs to be capable of representing a choice clause, a collateral clause, or a loop clause.

The structure tree for the sample program is shown in Figure 3.7. Note that there are no nodes for the parentheses.

This structure is mapped onto a tree of interconnected cells that implement the functions in the program.

For example, an expression a*b + (c*d +e) is represented by the structure shown in Figure 3.8. A postfix traversal of the structure yields the nodes $*_1$, $*_2$, $+_2$, and $+_1$. The cells for these nodes are constructed in this order as the structure is traversed. A cell (call it $cell_1$) that contains the cells $*_2$ and $+_2$ is created first. Then a cell that contains the cells $*_1$, $cell_1$, and $+_1$ is created. The resulting structure is shown in Figure 3.9. The dashed lines show the subcell hierarchy. Extra space is used to make the diagram easier to read..

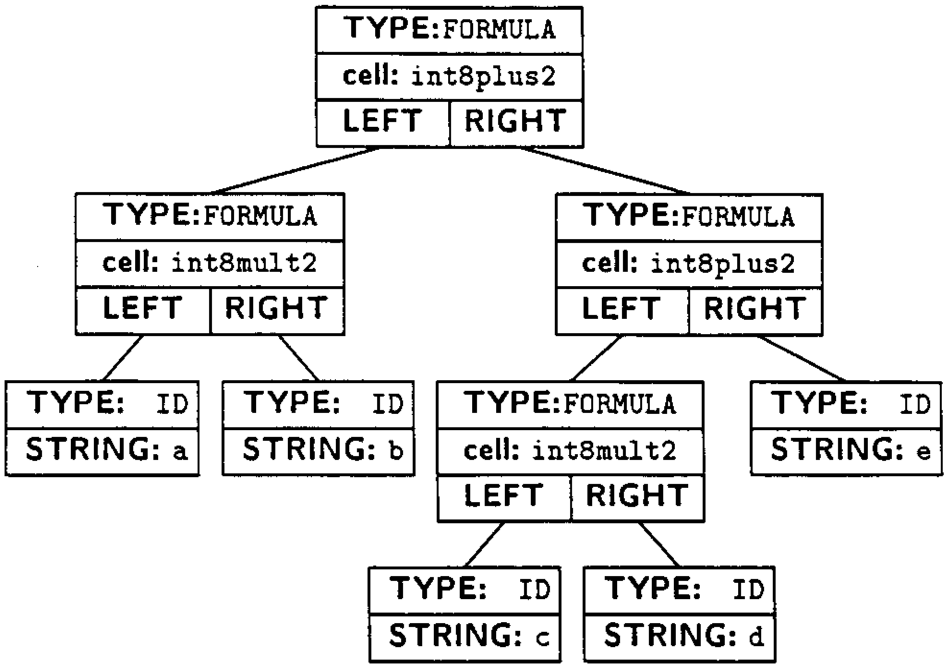


Figure 3.7: Structure Tree for the ALICS Program Fragment in Figure 3.2

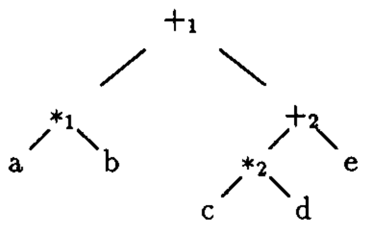


Figure 3.8: Structure Representing a Simple Computation

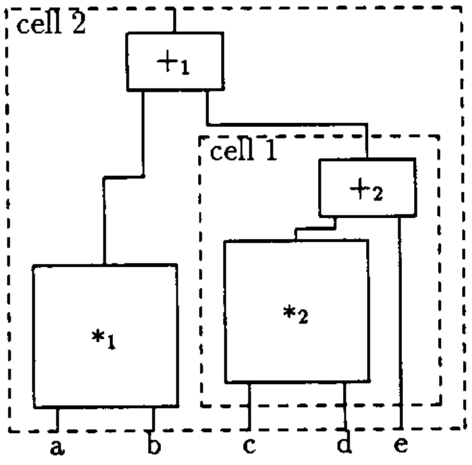


Figure 3.9: Hierarchical Structure for a Simple Computation

3.3 Target Architecture

Once the structure of the program has been determined by the parser's semantic routines, the structure is mapped into custom integrated circuitry. A hierarchical layout (topological structure) is described that facilitates this mapping. This layout is novel and exposes some problems that do not occur in traditional layout methods. A new method for cell layout is developed here to produce the layout, and route power, ground, and signal wires.

This section describes the target layout for the ALICS synthesizer. The overall layout method consists of methods for composing and interconnecting circuits in a hierarchical manner. An important feature of the hierarchical construction method is that only a small portion of the entire circuit need be considered at each level in the hierarchy. The topology of the cells used with this method is discussed along with the types of routing used to interconnect these cells. These methods are then related to the implementation of the source language.

3.3.1 Hierarchical Layout

One of the problems with hierarchical layout is the unsuitability of existing layout techniques for automated hierarchical layout. In this hierarchical layout method it is useful to treat primitive cells and cells created from primitive cells identically. This increases software reliability by reducing the number of special cases that the software needs to accommodate. Imposing a hierarchy also reduces the amount of information that the synthesizer has to consider at any given time.

The hierarchical layout method includes algorithms for routing signals and power and ground buses. The routing of power and ground has an important constraint that effectively restricts the routing of each of these buses to a single metal routing layer. While it is possible for each of these buses to be routed on more than one metal routing layer, there is a capacitive, resistive, and area penalty in each place where a wire changes layers.

The predominant method for cell layout, standard cells, is described next, with a brief discussion of its shortcomings for hierarchical layout. A method for designing cells that overcomes those shortcomings is proposed.

3.3.2 Standard Cells

Standard cell is a term for integrated circuitry that has a fixed height that is identical for all the cells in the library. All functional (not power or ground) inputs and outputs to the cells are on the tops or bottoms of the cells. Figure 3.10 shows two standard cells, a two-input XOR cell, and a two-input NAND cell, modified from cells provided in the UW/NW Consortium cell library [UWN84]. (The figure also contains a legend that shows the patterns that represent each mask layer.) The power buses are 7λ wide and separated by 43λ. A power bus runs horizontally across the tops of all the cells and ground runs horizontally across the bottoms of all the cells. The power and ground buses are connected by abutting adjacent cells. The power buses for several rows of cells are all connected on the left and ground buses are connected on the right so that power is distributed in an interleaved,

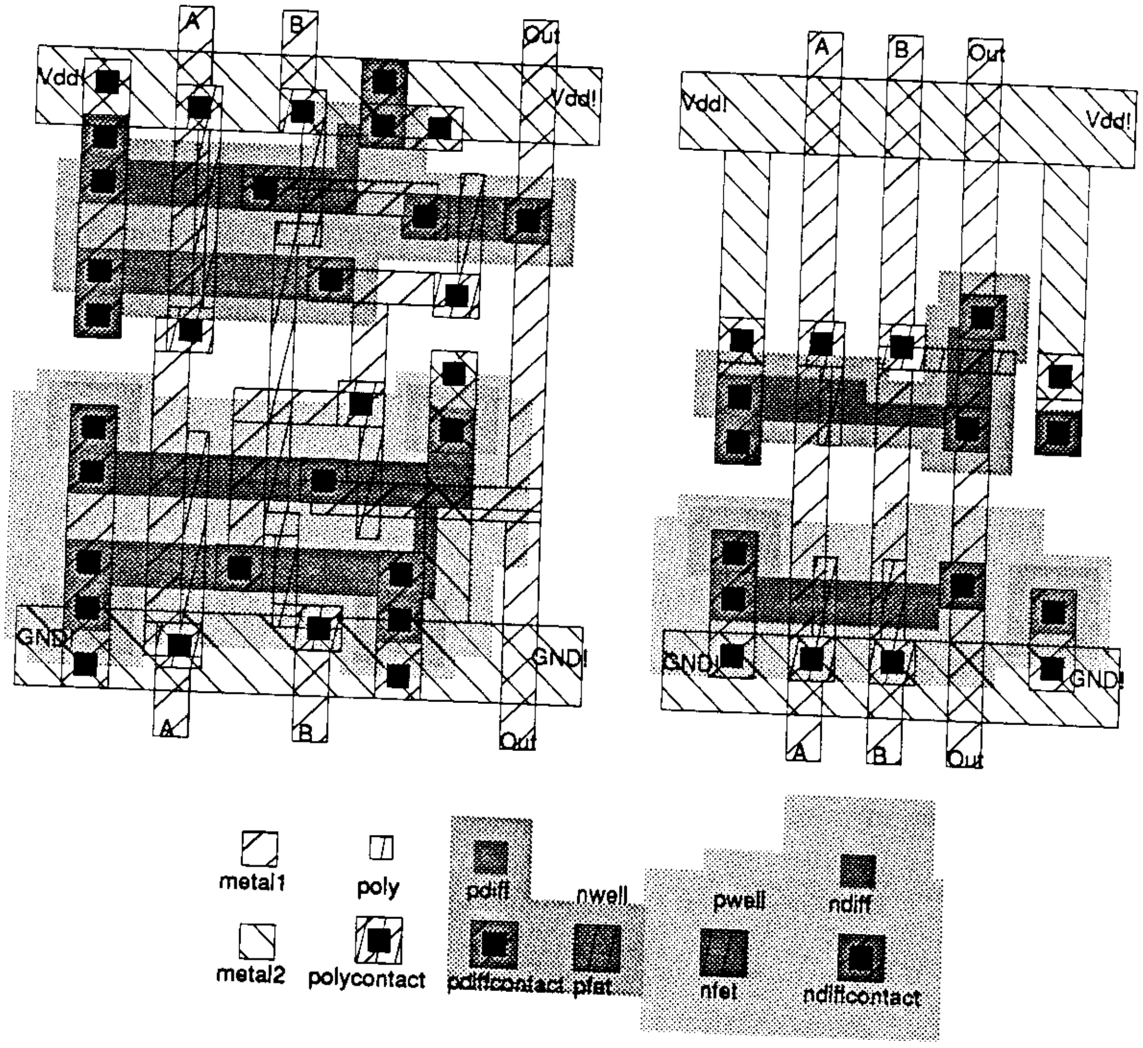


Figure 3.10: Standard Cells

non-overlapping manner. Signal interconnections are routed between, not over, the cell rows. Figure 3.11 shows an implementation of an eight-bit ripple-carry adder in five standard cell rows with their interconnections.

If standard cells were used by a synthesizer, a pass through the structure that represents the computation would be needed to flatten any hierarchy that may be present in the design. The standard-cell method does not allow composite cells to be created since this would violate the constant-height basis for the standard cells. Once the standard cell rows have been determined, the amount of current required by each cell is used in the calculation of the cumulative current across each cell row at each cell in the row. So if three cells in a row all require one milliamp of current, then the V_{DD} bus entering on the left of the row would have to be wide enough to supply three milliamps of current to the row. Similarly the ground bus (entering the row on the right) would have to be just as wide.

3.3.3 Hierarchical Standard Cells

To overcome the problems with "standard cells", a hierarchical variation on standard cells has been developed. The *hierarchical standard cells* all have the same structure, but not necessarily the same height or width. The power wires, V_{DD} and ground, are on the left and right sides, respectively, of the cells on the second level of metal. Inputs to cells are on the bottom sides of the cells, outputs are on the top sides, all on the first level of metal. Figure 3.12 shows several hierarchical cells. Note that the power buses are not separated by the same distance as was

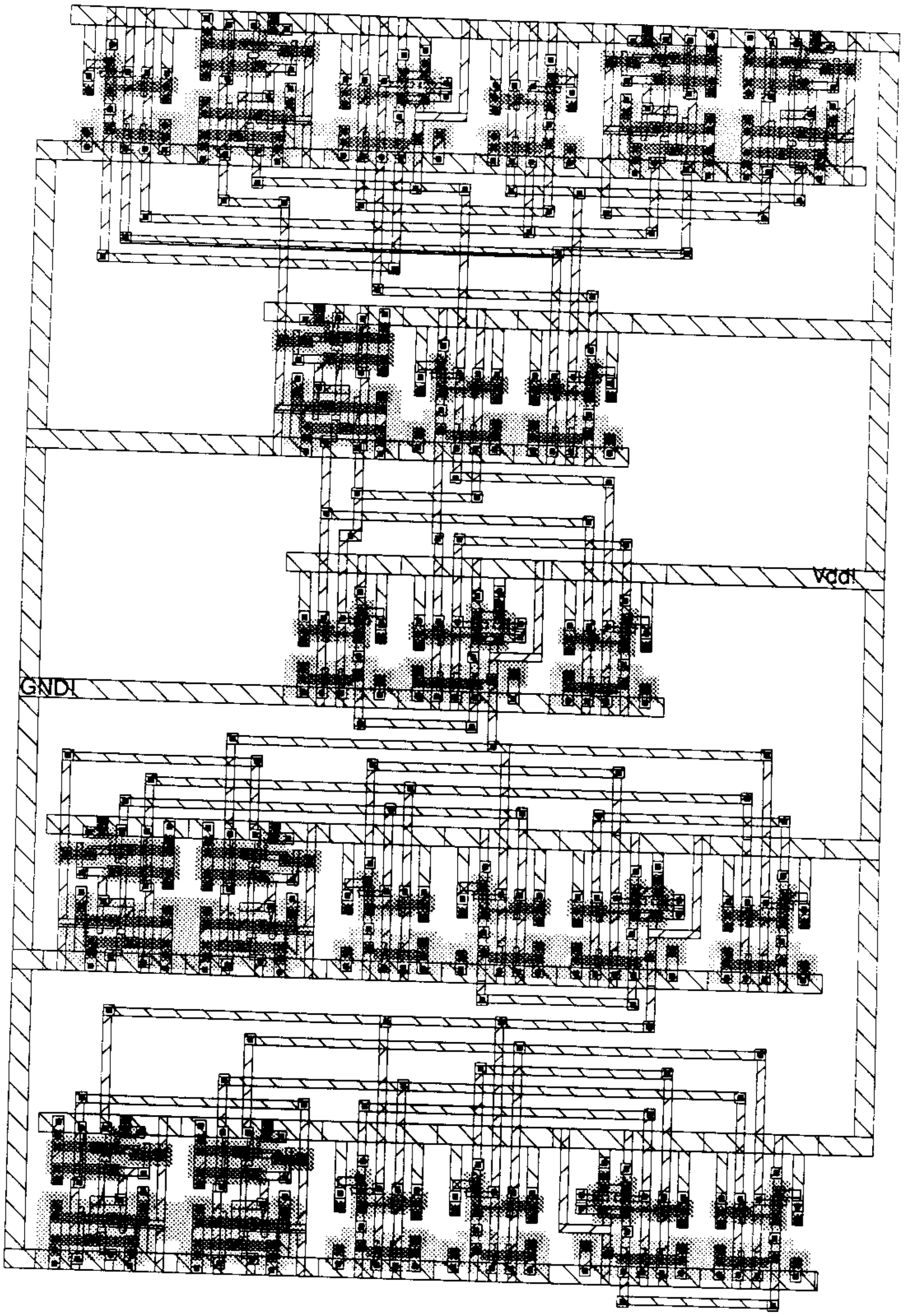


Figure 3.11: Standard Cell Rows

required for the standard cells. The two-input **XOR** has 4λ wide power buses which are separated by 45λ. The two-input **NOR** has 4λ wide power buses which are separated by only 35λ. This is a distinct advantage over "standard cells", because cells can be made as small or as large as is necessary. This is important so that new cells can be added to a cell library without having to conform to a pre-existing height restriction.

When the hierarchical standard cells are combined to form more complex circuits, a new cell is created for each subexpression in the circuit. The power and ground lines are connected on the second level of metal (metal-2) without ever crossing each other. The V_{DD} and ground wires are brought to the left and right sides of the bounding box of each cell so that the power layout of the composite cell corresponds to power layout of the primitive cells. If wide power wires are needed on the left or right side of this new cell, a wide power wire is laid on top of the existing power wires in the subcells, with the wider part of the power wire extending away from the center of the composite cell. The signal wires, which are on the first level of metal (metal-1), cross the power wires without shorting with power wires. Figure 3.13 shows several cells that have been laid out hierarchically.

3.3.4 Power and Ground Distribution

Power (V_{DD}) and ground must be distributed to all on-chip circuitry. One consideration for wiring these two signals is that they be wide enough to accommodate

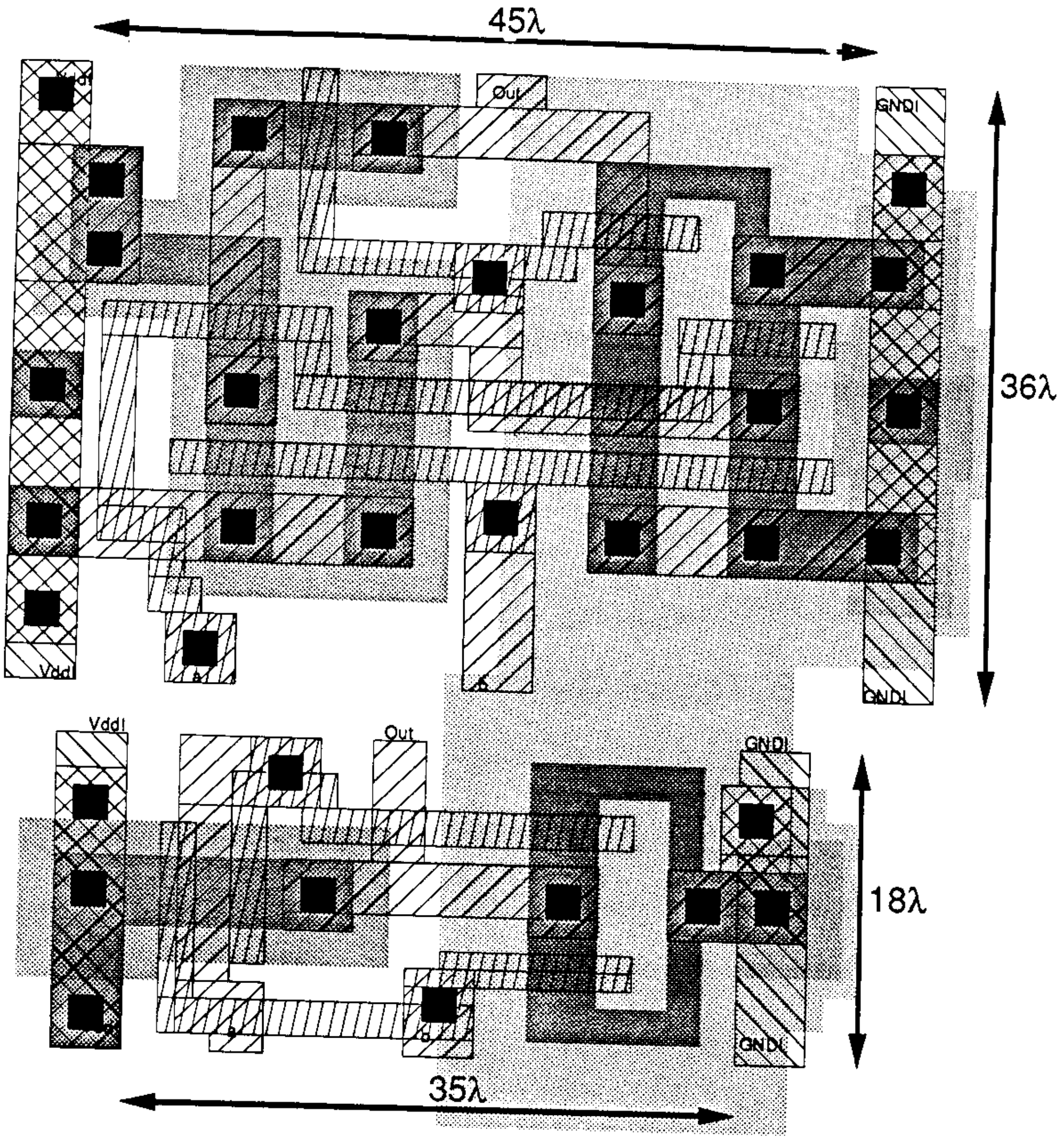


Figure 3.12: Primitive Hierarchical Standard Cells (Two-Input XOR and Two-Input NOR)

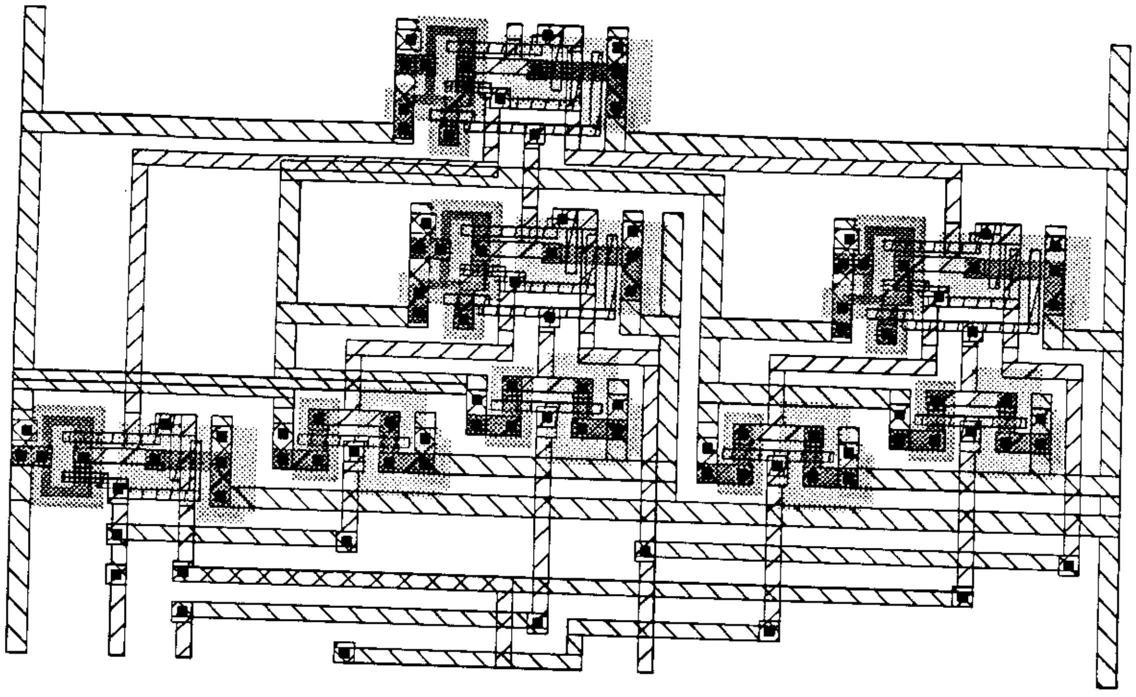


Figure 3.13: Composite Hierarchical Standard Cells

the current that passes through those wires. If the current is too large, a phenomenon known as *metal migration* occurs. Metal migration causes the aluminum atoms in a wires to move with the electric current [West85,Mead80,Chow86]. If wires are made larger than necessary, area on the chip is wasted. As a wire's atoms are moved out of the high current density regions, the current density (per cross-sectional area) increases until the wire burns out (as a fuse does). By carefully calculating the amount of current required, wires are made wide enough so that this problem does not occur.

Figure 3.14 shows the power and ground requirements for a row of four cells that are combined into a parent cell. Each box in the figure contains the electrical current required for that cell. Internal to each cell (not shown) are vertical V_{DD}

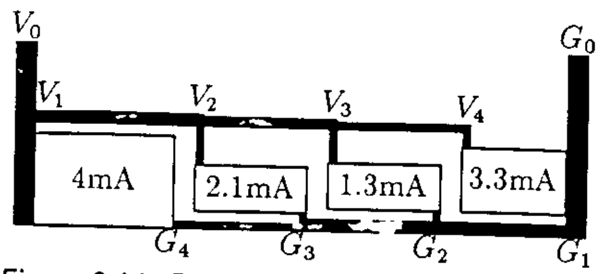


Figure 3.14: Power and Ground Requirements

and ground wires on the left and right (respectively). The cell created from these cells will also have the same V_{DD} and ground configuration. Internal to this parent cell are horizontal wires that bring V_{DD} and ground to each cell from the power buses on the left and right of the cell. Four points are marked along each horizontal V_{DD} and ground wire. These mark places where the width of the power wire may change. The V_{DD} wire segments between each of these points need to carry the current for the cell beneath the right point of this segment plus the current for all the cells to the right of this point. So V_{DD} wire at point V_0 needs to be wide enough to supply current to all four cells (10.7mA); the V_1–V_2 segment needs to carry the current for the three cells on the right (6.7mA); V_2–V_3 carries the current for the two rightmost cells (4.6mA); and V_3–V_4 carries the current for the rightmost cell (3.3mA). Similarly the ground wire at point G_0 needs to be wide enough to supply current to all four cells (10.7mA); the G_1–G_2 segment needs to carry the current for the three cells on the left (7.4mA); G_2–G_3 carries the current for the two leftmost cells (6.1mA); and G_3–G_4 carries the current for the leftmost cell (4mA). The minimum width for each segment is calculated from the following formula and are summarized in Table 3.1.

$$width = \min(min\_width(layer), \frac{current}{max\_current\_density \times thickness(layer) \times \lambda})$$

Table 3.1: Minimum Widths for Power and Ground Wires for Figure 3.14

Segment	Current	Width
V_0- V_1	10.7mA	8
V_1- V_2	6.7mA	5
V_2- V_3	4.6mA	4
V_3- V_4	3.3mA	3
G_0- G_1	10.7mA	8
G_1- G_2	7.4mA	5
G_2- G_3	6.1mA	5
G_3- G_4	4.0mA	3

3.3.5 Hierarchical Placement

In hierarchical placement, cells are created as the structure tree is traversed. A tree-structured computation is mapped into a tree-structured layout. For each node with children, a cell is created containing the parent and its children. At the top of this hierarchy is a cell containing all the other cells in the layout. At each level in the hierarchy, the children are placed in a row with the their tops at the same y coordinate. The parent cell is centered above the children, with enough space to accommodate the signal wiring between the children and the parent. A V_{DD} bus is then created that connects the tops of all the V_{DD} ports of the child cells and is connected to a new bus that runs vertically at the left side of the new cell. This bus is created wide enough to accommodate all the current that passes through it. Thus the V_{DD} bus is made narrower on the right, since it supplies only the rightmost cell. The V_{DD} bus for the parent cell is connected to this bus, and the width of this bus accounts for the current consumed by the parent cell also. A similar ground bus is then created below all the children cells. The ground bus runs the height of the new cell on the right side.

This method is similar to the standard cell row routing because the two buses never cross and are interleaved. The signals wires still have to cross the power buses as was necessary for the standard cells. The advantage is that the widths of the buses are calculated from only the current of each parent cell and its immediate children. This information is retained as new cells are created so that the contents of already created cells (grandchildren and further generations) do not have to be re-examined. The resulting process requires time proportional to the number of nodes. Coupled with the fact that the per-node calculation is not very complicated, the entire process is fast. Another advantage is that adjacent cells do not have to have the same height since their power buses do not have to abut. This means that library cells can be of differing sizes, as needed.

3.3.6 *n*-ary Trees

Whether written as a function with arguments or as an equation with infix operators, the main component of arithmetic expressions is represented by an n-ary tree. In general, the function application

$$f(a_1, a_2, a_3, \ldots, a_n)$$

is represented by the n-ary tree shown in Figure 3.15. Each node represents a

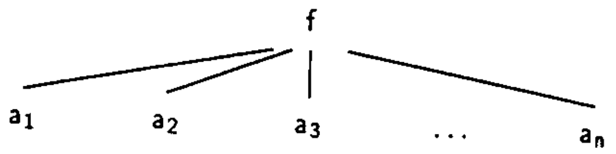


Figure 3.15: n-ary Tree

computational unit and each arc represents a data path. This tree can then be mapped to the layout shown in Figure 3.16. In this picture, each box contains

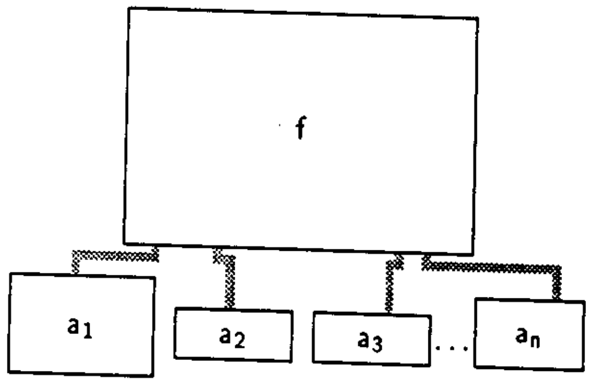


Figure 3.16: n-ary Tree Layout

circuitry that implements the corresponding computational unit in the tree. The shaded lines represent wires that connect the outputs of the circuitry implementing the functions that are the main function's actual parameters to the inputs of the circuitry for the main function. If the leftmost subcell is taller than the other subcells, the other cells are placed high enough so that a ground wire whose bottom edge is flush with the bottom of the leftmost cell can be run underneath the cells.

3.3.7 Conditional Expressions

Figure 3.17 shows the floorplan for an **IF** expression. The circuitry on the bot-

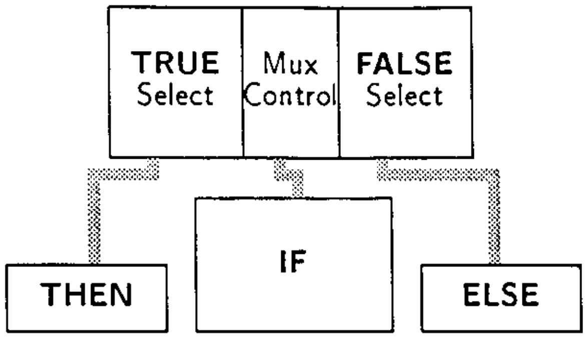


Figure 3.17: Block Diagram for an **IF**-expression

tom is partitioned into three parts, corresponding to the source language **THEN**, **IF**, and **ELSE** parts. The multiplexor at the top of the figure is controlled by the output of the **IF**-part. If the output of the **IF**-part is **TRUE**, the multiplexor selects the output of the **THEN**-part; otherwise the output of the **ELSE**-part is selected. A CMOS schematic of the multiplexor is shown in Figure 3.18.

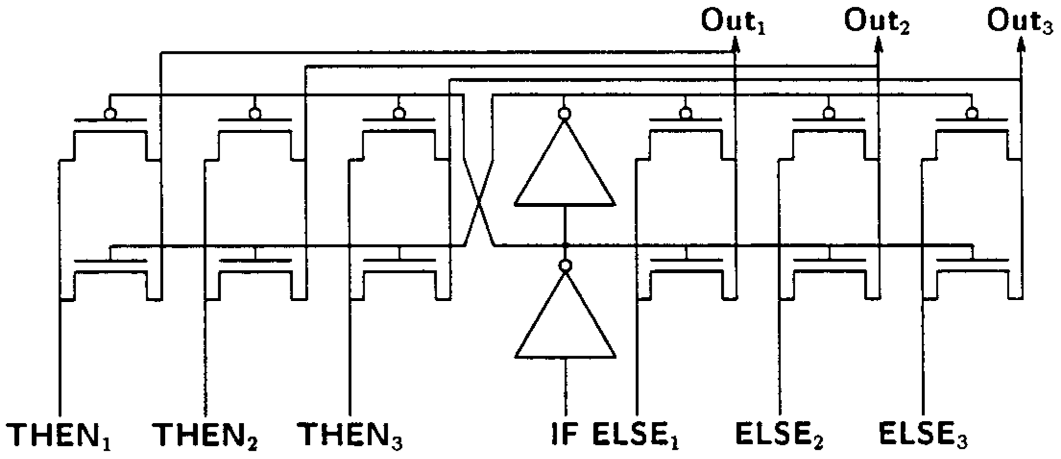


Figure 3.18: Schematic for a CMOS **IF**-expression

The source language allows an **IF**-expression with the syntax shown in Figure 2.6, where the **ELIF** (a contraction of **ELSE** and **IF**) can be repeated as many times as desired. This structure is hierarchical in that the **ELIF-THEN-ELSE** parts are mapped onto the **IF**, **THEN**, and **ELSE** parts, respectively, of the structure in Figure 3.17. This structure is then used as the **ELSE** part in an identical structure as is shown in Figure 3.19. The **IF** and **THEN** parts are mapped into

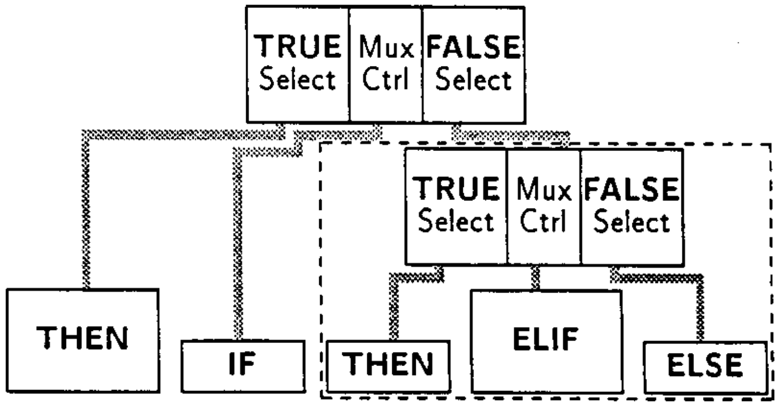


Figure 3.19: Structure for an **IF** Expression Containing an **ELIF** Clause

the corresponding **IF** and **THEN** parts of this structure.

3.3.8 Converting Arrays to Sequences

All the elements in an array are accessible in parallel; elements in sequences are accessible sequentially. A parallel-load shift register performs the conversion of an array to a sequence. The cell at the end of the register contains the current value that will be output. The conversion operation begins by loading an array into the shift register and outputting the first element during the first clock cycle. The remaining elements are shifted into (and output from) the end register cell during each successive clock cycle.

There are four operators that convert from arrays to sequences, differing according to which of the array bounds is used to select the first element in the sequence. Two configurations of shift registers are needed to implement all four of these operators. Block diagrams for these are shown in Figure 3.20. Which of

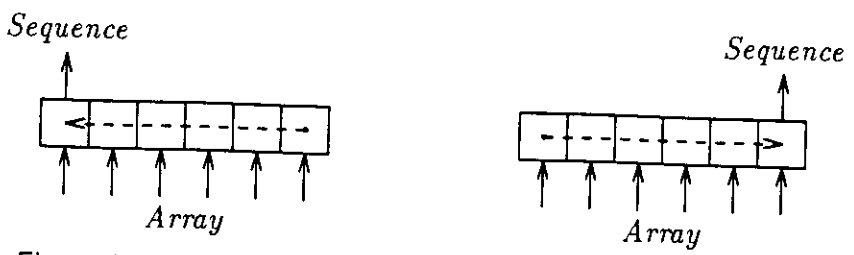


Figure 3.20: Block Diagrams for Array to Sequence Operators

these two cells is chosen depends on whether the first element of the sequence is on the left or on the right of the array. This information is known at compile time and so the proper configuration is chosen.

A CMOS schematic for two bits of a cell in the shift register used for a rightmost-element-first or highest-index-first parallel to serial converter is shown in Figure 3.21. Two-phase static flip-flops are used in the design. These latch the input on the high state of the clock (ϕ) and transfer the data from the master flip-flop to the slave flip-flop during the clock's low state. Static flip-flops are used so that the clock is allowed to be slowed to make testing easier. The timing diagram for an extension of this circuit to four bits is shown in Figure 3.22. The D_3 signal is the input to the third bit of the register and needs to be valid during the time indicated. During the first clock pulse, the array values are loaded into the register. The output of the high-order position is available during the clock's low state. The next three clock pulses transfer the low-order elements into the output position.

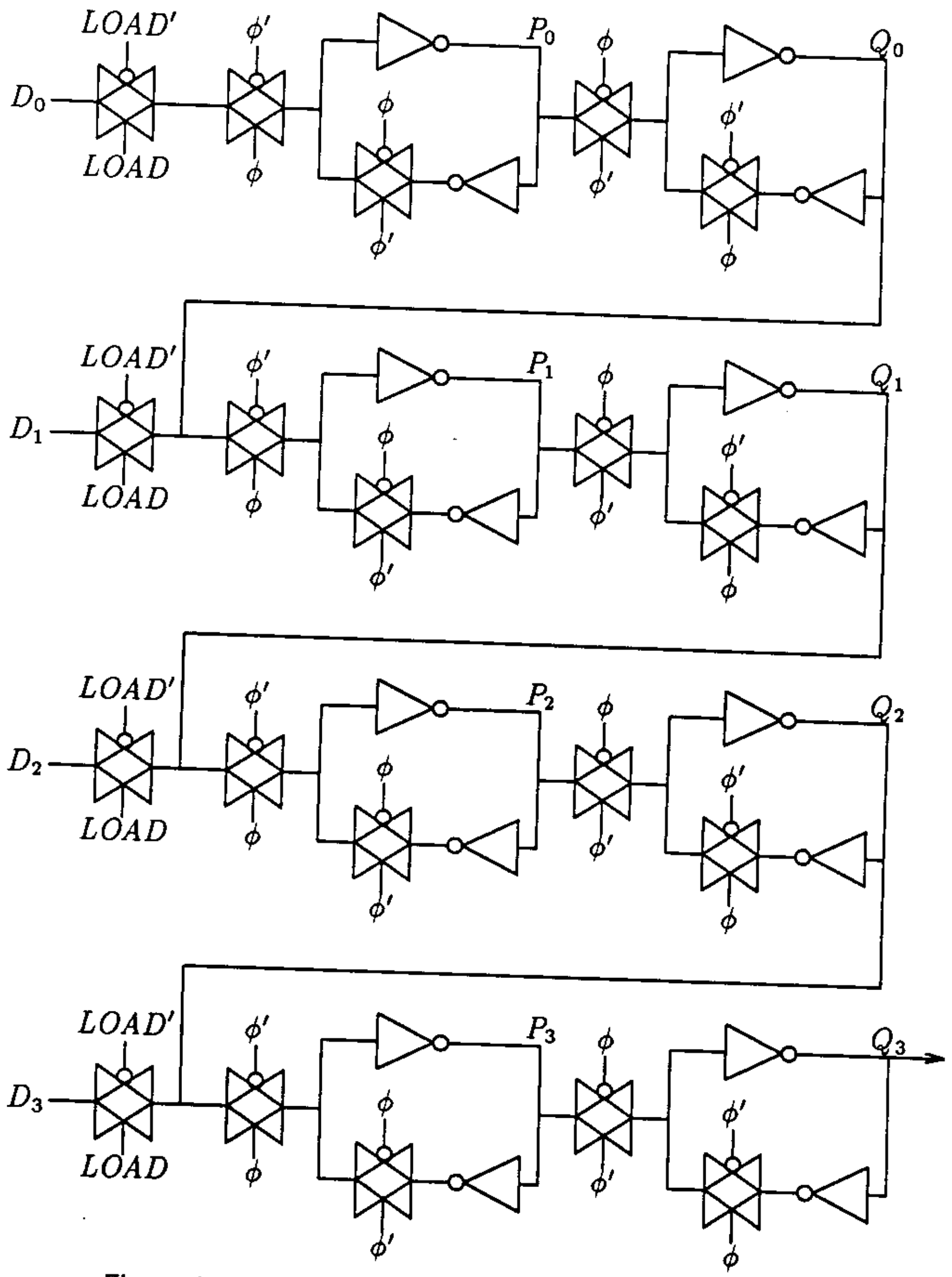


Figure 3.21: Schematic for Four Bits of a Shift Register

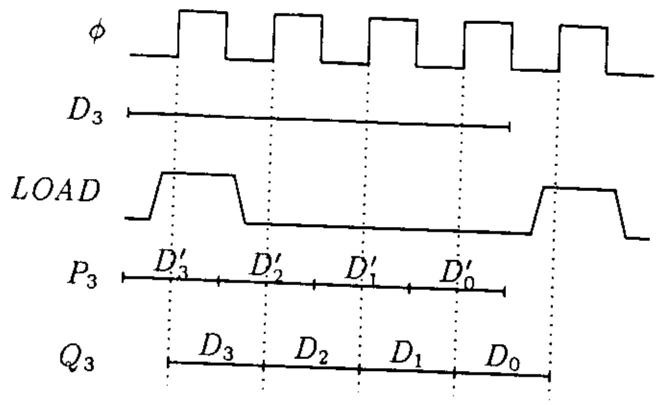


Figure 3.22: Timing Diagram for a 4-Bit Shift Register in Figure 3.21

3.3.9 Converting Sequences to Arrays

Sequences are converted to arrays using shift registers very similar to those used for the array to sequence converters. These shift registers are serial-in, parallel-out. Block diagrams for the two basic configurations are shown in Figure 3.23. Which

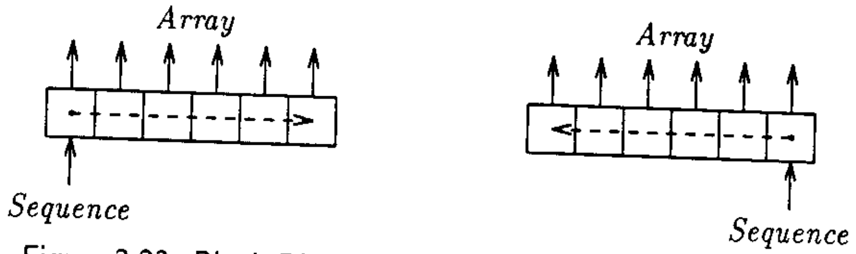


Figure 3.23: Block Diagrams for Sequence to Array Operators

of these two cells is chosen depends on whether the first element of the sequence will be on the left or on the right of the array.

Figure 3.24 shows the schematic for a shift register that implements a sequence-to-array converter. The circuitry for the serial-in, parallel-out shift register is similar to the parallel-in, serial-out shift register. Figure 3.25 shows the timing diagram for this circuit. The elements of the sequence are input one per clock cycle. The shift register in the example is four bits wide, so all the values will be

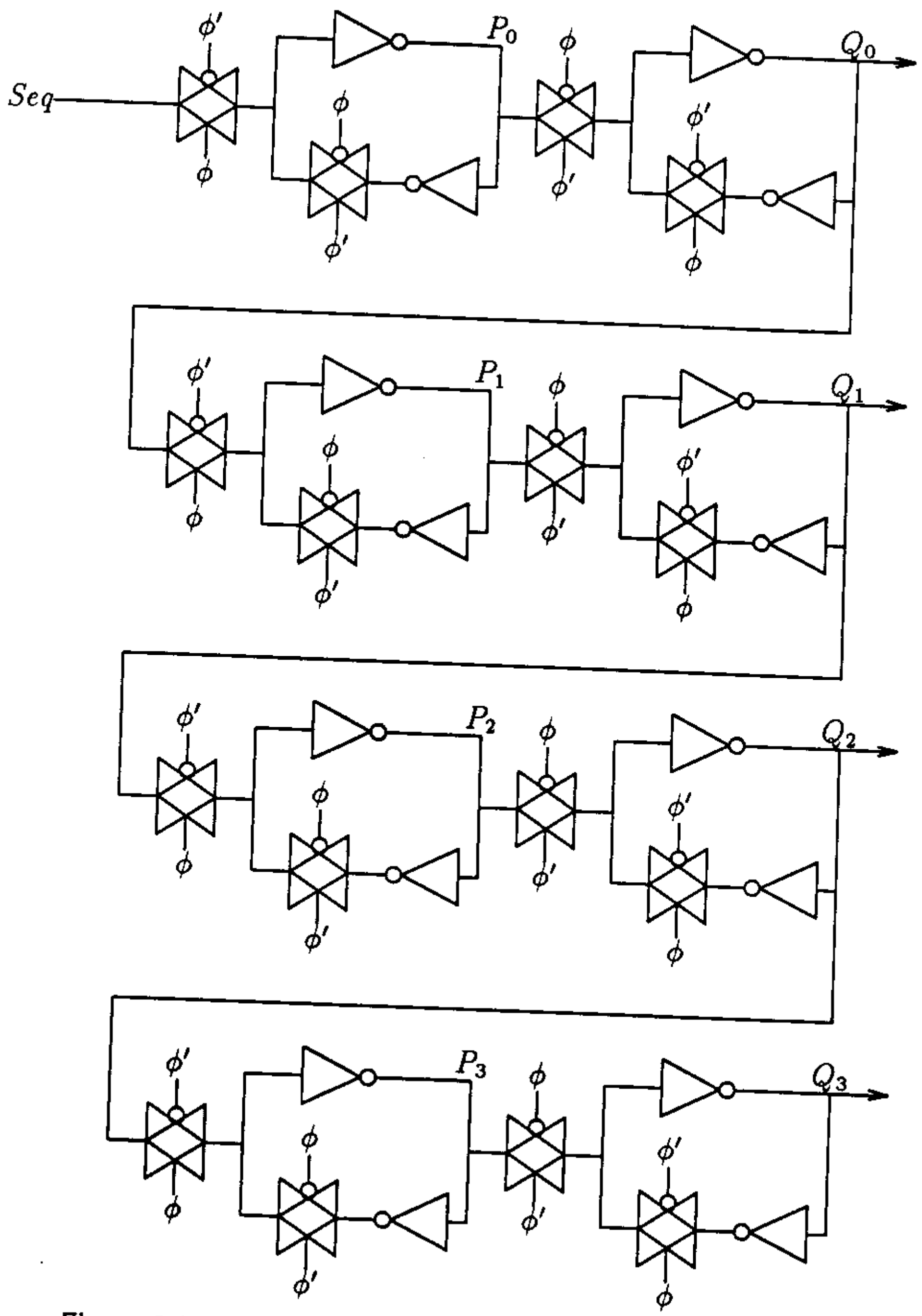


Figure 3.24: Schematic for Four Bits of a Shift Register

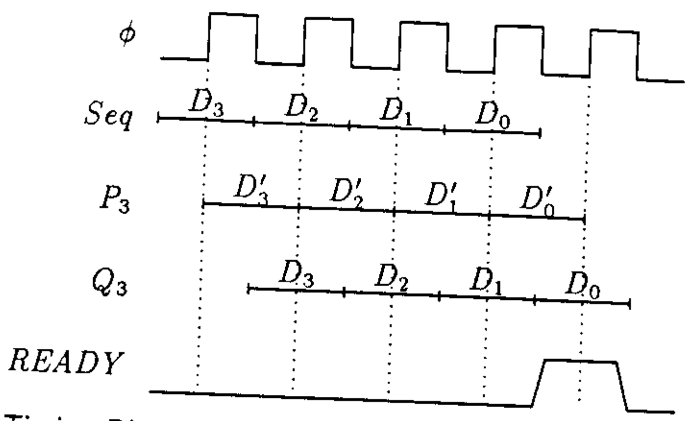


Figure 3.25: Timing Diagram for a 4-Bit Serial-In, Parallel Out Shift Register in Figure 3.24

loaded after the fourth clock cycle.

3.3.10 Sequential Iterations

The general form of a sequential iteration is shown in Figure 3.26. This is

```
FOR var SEQ seq
DO state-initialization;
    state ← g(var, state)
OUT h(var, state)
OD
```

Figure 3.26: General Form of a Sequential Iteration

mapped onto the structure shown in Figure 3.27. The clock (ϕ) signal controls the iterations. On each iteration the clock strobes the current element of the sequence into the *var* register and the output of the multiplexor into the *state* register. In the first iteration the first element of the sequence is latched into the *var* register and the *first* indicator directs multiplexor to select the output of *init* into the *state* register. The g module then takes as input the first element of the sequence (from *var*) and the output of *init*. The output of the g module is stored in the *state*

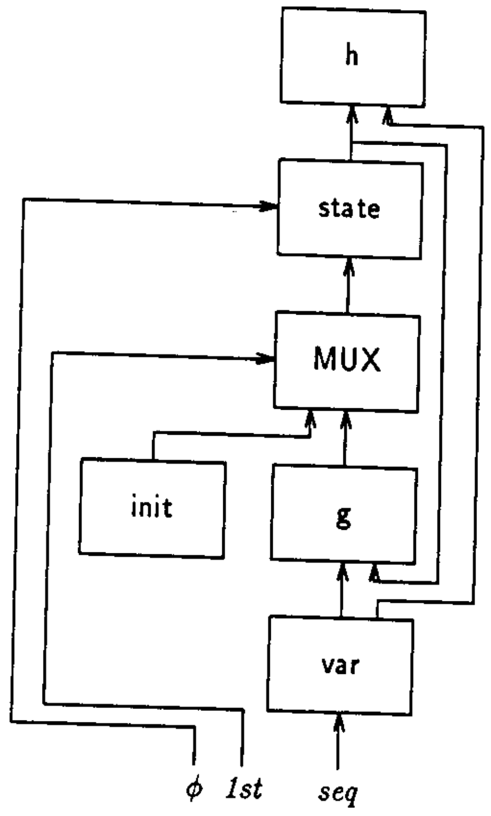


Figure 3.27: Layout for a Sequential Iteration

register by the next clock pulse. The output function, *h*, then takes the output of the *init* module (stored in *state*) and the first element of the sequence (from *var*). At the same time the second element of the sequence is latched into the *var* register for the next iteration. During all the remaining iterations the multiplexor selects the output of the *state* register, which contains the output of the *g* module from the previous iteration. At the end of each iteration the state is available to the output function h.

3.3.11 Routing

The previous section described how the power and ground routing is performed hierarchically. This section discusses methods to route the signal wires with this layout method. The three methods needed are river routing, permutation routing, and a variation on channel routing, called here one-to-many channel routing. The next sections discuss where these types of routing are needed and how each type of routing is performed. Each type is discussed in turn and is compared with existing methods.

In all types of routing, wires need to be of a minimum width for a given technology and may be larger than the minimum in order to conduct the required electrical current. All points in all wires need to be separated from all unrelated wires on the same layer by a minimum separation. In the cases we consider, wires are run vertically across a rectangular region called a channel. Circuitry lies above and below the channel and wires are placed in the channel connecting the appropriate ports on this circuitry. All I/O ports under consideration lie on the top and bottom borders of the channel. The channel height is constructed to be large enough to contain all the necessary wires. One of the goals in wiring the channel is to minimize the channel height, given a fixed placement of cells on the top and bottom of the channel. Figure 3.28 shows a valid routing of three signals in a channel. If more bends are introduced into the wire between ports B_2 and T_2, the channel height is reduced, as shown in Figure 3.29.

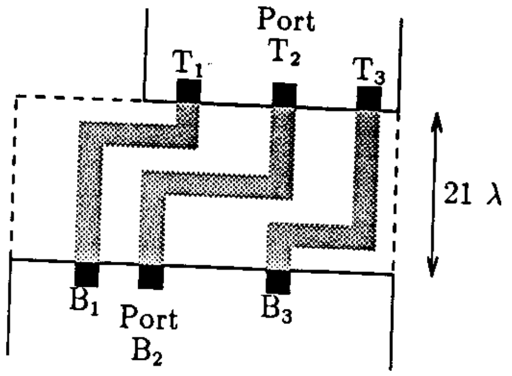


Figure 3.28: A River Route with Two Bends per Wire

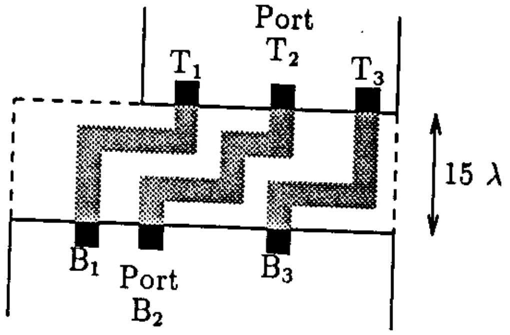


Figure 3.29: A Minimum-Height River Route

3.3.11.1 River Routing

River routing is a special form of routing involving wires that are on a single layer, adjacent, and never cross adjacent wires [Dole81]. Figures 3.28 and 3.29 are examples of river routing. This approach is useful in routing the n sub-nodes to the parent in a n-ary tree. Figure 3.30 shows the routing for an n-ary tree ($n = 4$). The parent cell is above the four sub-cells. Because the four cells are placed, left

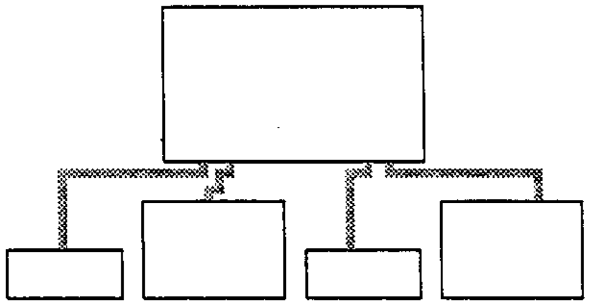


Figure 3.30: River Routing for an n-ary Tree

to right, in the order that the values are needed by the parent cell the routing from the subcells to the parent do not have to cross one another. Therefore river-routing is used to connect each of the outputs of the four subcells to the parent's inputs. Additionally, since the datatypes of the subcells' outputs are not restricted to **BIT**'s, the wire bundles representing the outputs' datatypes may consist of many individual wires. All these wires can be routed using river routing, as is shown in Figure 3.31.

IF sub-expressions are routed to the multiplexors using river-routes. Figure 3.17 shows the floorplan for an **IF** expression. The circuitry on the bottom is partitioned into three parts, corresponding to the source language **THEN**, **IF**, and **ELSE** parts.

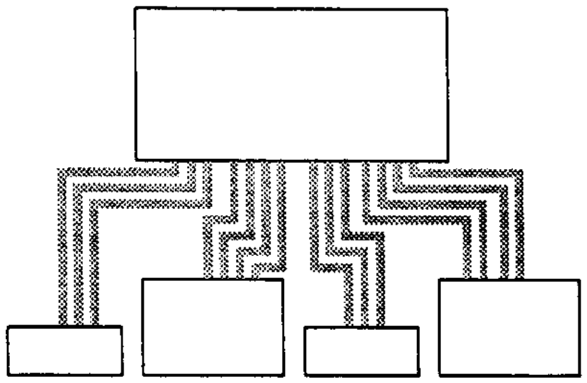


Figure 3.31: Wide Datapath River Routing for an n-ary Tree

The multiplexor at the top selects the value from either the **THEN** part or the **ELSE** part, depending on the value of the **IF** part. Although the figure shows single wires connecting each of the sub-expressions to the multiplexor, these wires may be wire bundles consisting of many individual wires that all travel in a similar path.

3.3.11.2 Permutation Routing

In the previous sections it was assumed that the signals were available in the order needed. This, however, is not necessarily the case. If the formal parameters to a function are not declared in the same order in which they are referenced in the function body, permutation routing is used to reorder the parameters from the declared order into the referenced order. In this section it is assumed that no parameter is referenced more than once. This restriction is removed in the following section.

Extant literature labeled *permutation channel routing* [Leon85] simplifies channel routing by permuting the ports on the channel boundary. This reduces the number of wire-crossings, but it does not address the routing problem. General channel-routing methods, such as the "greedy" channel router [Rive82] or Baker *et al* [Bake83], reserve one routing layer for horizontal wires and the other for vertical wires and thus cannot route a reversing permutation-route without having to widen the channel. Joobani's artificially intelligent router [Joob86] does not reserve one direction per layer, and so can route permutations. However, the analysis required for his method is more complex (at least n^3) than is needed.

Some definitions from [Kele87] are useful:

Definition 1 (Permutation) *The term permutation refers to a function that is one-to-one and onto.*

Definition 2 (Port) *A port is a point on the top or bottom or the channel where a wire terminates. A port is associated with a unique signal. The main goal in routing is to create wires within the channel that independently connect all the ports associated with each signal.*

Definition 3 (Wiring permutation) *A permutation applied to a set of ports on one side of a channel is called a wiring permutation. The order of the ports on one side of the channel is said to be permuted to yield the order of the ports on the top of the channel.*

Definition 4 (Permutation route) *A permutation route is a valid routing, or placement, of wires connecting the bottom and top ports of a channel. The channels* *under consideration have a fixed width and may vary in height to accommodate the necessary wires.*

Definition 5 (Permutation routing algorithm) *The permutation routing algorithm performs any permutation-route on* n *signals in two layers, denoted L1 and L2.*

Without loss of generality, the signals are assumed to start on L1 at the top of the channel[1] and end on L1 or L2 at the bottom of the channel. The ports at the top and bottom of the channel are assumed to be properly positioned in columns wide enough for a wire and a contact. However the ports at the bottom of the channel do not have to line up with the ports at the top of the channel.

A formula for the height of a channel is difficult to determine in general because it depends on the particular permutation. The height is determined by performing the first two steps of the algorithm and then noting the column that is the tallest.

All the wires in the permutation-route are either completely vertical or consist of three wire parts—vertical wire that brings the signal down from a port into the channel, a wire that brings that wire to the destination column, and another vertical wire that completes the connection to the port at the bottom of the channel.

Definition 6 (Route-dexter) *A route whose lower port is to the right of its upper port is referred to as a route-*dexter.

Similarly,

[1]With the addition of an extra row of L1–L2 contacts at the top of the channel, the restriction of beginning with layer L1 can be removed.

Definition 7 (Route-sinister) *is defined as a route whose lower port is to the left of its upper port.*

Definition 8 (Route-vertical) *Routes-vertical are routes whose top ports lie directly above the corresponding bottom port.*

Routes-vertical are handled the same way as routes-sinister. Figure 3.32 shows a route-sinister and a route dexter. *Shading-sinister* represents one routing layer

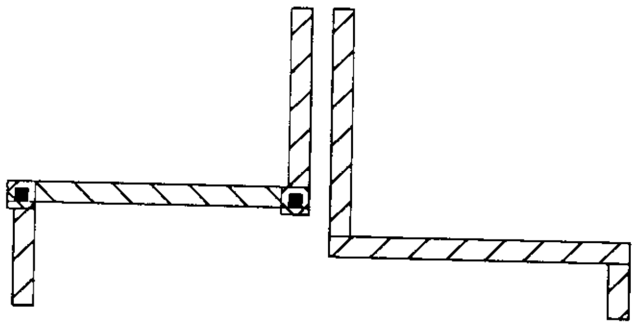


Figure 3.32: A Route-Sinister and a Route-Dexter

(L1, which is metal-1) and *shading-dexter* represents the other layer (L2, which is metal-2). A black square represents a via, or contact, between the two layers. As this figure shows, each wire consists of three parts—a vertical top and bottom portions and a horizontal middle portion.

The choice of routing layers for the different signals is important. A badly placed signal route could become a barrier for remaining unrouted signals. For instance, consider a route of five signals that contains a complete reversal. The ports on the bottom of the channel are numbered 0 through 4 from left-to-right. The ports on the top of the channel are numbered from left to right by the permutation function $(i \times 4) \bmod 5$, where i is the number of the corresponding port on the bottom of the channel.

We might try to route this permutation as follows. First, we create a route-sinister between top-port 4 and bottom-port 1 on layer L1 (Figure 3.33). Next

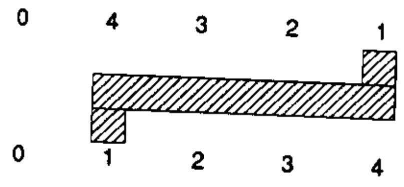


Figure 3.33: Simple-Minded Routing Strategy

we try to create the route-dexter between top-port 1 and bottom-port 4. Because the first route blocks top-port 1 on layer L1, we have to route the second wire on layer L2 (Figure 3.34). The remaining ports are unroutable because both routing

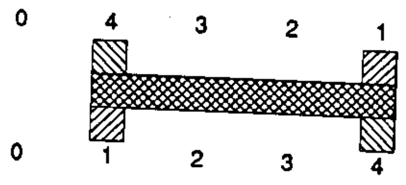


Figure 3.34: Simple-Minded Routing Strategy Blocks Routes 3 and 2

layers are occupied for the width of the channel by the first two wires.

An efficient algorithm for performing the routing without backtracking is explained in two ways. A simplified version of the permutation routing algorithm is explained first. The simplified version is easy to understand, but it wastes routing area. Next, the details of a complete permutation routing algorithm are given. The detailed version contains optimization techniques that obfuscate the underlying algorithm.

The Simplified Permutation Routing Algorithm

The permutation routing algorithm always generates routes without any backtracking. Routes-sinister and routes-dexter are routed from the top of the channel down to a free horizontal track (row), then over to the column of the bottom port for the route and then down to the bottom of the channel.

The algorithm consists of three steps. In the first step the top two portions (top vertical wire on layer L1, middle on layer L2) of each sinister wire are routed. This choice of layers insures that the top vertical segments of routes can enter the channel and not be blocked by the horizontal segments. In the second step, the routes-dexter are brought down from the top port on layer L1 to a free row (also called a *horizontal track*) and then routed rightwards, still on layer L1, to the destination column. (To insure that a route-dexter does not block another route-dexter, routes-dexter are selected for routing from right to left by their top ports.) In the third step, the lower vertical segments of all the wires are routed down to the destination ports. This last wire segment is routed on layer L1 if the destination layer is L1 and no routes-dexter (routed on layer L1) pass through this segment. The height of the channel is determined in this step by noting the height of the channel if the necessary contacts were placed. After the height is determined, the contacts are actually placed and wires drawn to connect the routes to the bottom of the channel.

For example, consider a function whose formal parameters are (a, b, c, d, e, f, g) and the order of usage inside the function is (a, d, g, c, f, b, e). The ports are assumed to be in L1 in both the top and bottom of the channel and legally positioned. The routes-sinister along the bottom of the channel are examined from left to right. In this example, the routes-sinister along the bottom of the channel are in columns 0 (a), 1 (b), 2 (c), and 4 (d) and are routed, one at a time, in that order.

1. Route a begins and ends in column 0. It does not have to cross any existing routes, so a minimum-height L1 stub is brought down from the top of the channel.

2. Route b begins in column 5 at the top of the channel and ends in column 1 at the bottom of the channel. The L2 horizontal portion of this wire can occupy the top (first) horizontal channel between these columns.

3. Route c begins in column 3 at the top of the channel and ends in column 2 at the bottom of the channel. The L2 horizontal portion of this wire must be placed below the first horizontal channel.

4. Route e begins at the top in column 6 and ends at the bottom in column 4. The horizontal portion of this wire can occupy the next available horizontal track. (For the sake of the discussion, it is placed in the third track instead of sharing the second track.

Figure 3.35 shows the channel after the first step has been completed.

In the second step the routes-dexter are routed. A list of top-ports routes-dexter is scanned from right to left, and the routes-dexter are determined to be

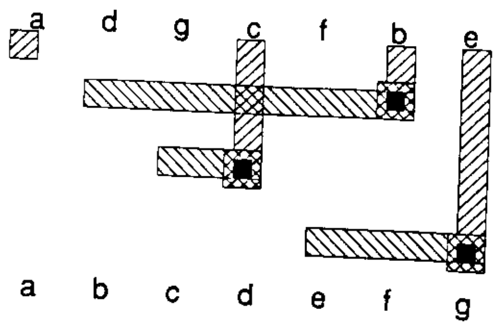


Figure 3.35: Simplified Routing Step 1 for × 5 in Residue 1-of-7

routes f, g, and d. These are routed as follows:

1. Route f begins at the top of column 4 and ends at the bottom of column 5. All the previously-placed horizontal wires are on layer L2, so an L1 wire can cross all these to reach the next available horizontal track (the fourth from the top). The horizontal portion of this route is placed on layer L1.

2. Route g begins at the top of column 2 and ends at the bottom of column 6. Again, an L1 wire is brought down from the top of the channel and crosses two horizontal L2 wires (horizontal portions of routes-sinister). Route f has to cross the horizontal L1 wire of route g, so an L1–L2 contact is needed at the end of the partially routed route f.

3. Route d begins at the top of column 1 and ends at the bottom of column 3. Again, an L1 wire is brought down from the top of the channel and crosses only horizontal L2 wires (horizontal portions of routes-sinister).

Figure 3.36 shows the channel after the second step has been completed.

In the third step of the algorithm, the height of the channel is determined by scanning all the columns and noting the maximum of their heights.

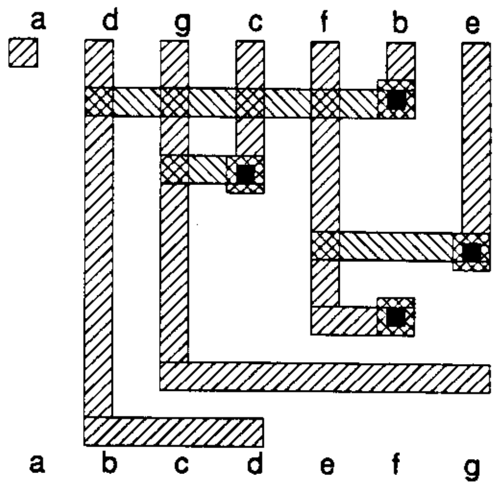


Figure 3.36: Simplified Routing Step 2 for × 5 in Residue 1-of-7

- The height of column 0 is the height of the L1–stub.
- Columns 1 and 2 require space for L1–L2 contacts below the horizontal L1 portion of route **d**.
- Column 3 does not require any more height than is taken by route d.
- Columns 4 and 5 require space for L1–L2 contacts below the horizontal L1 portion of route **g**.
- Column 6 does not require any more height than is taken by route **g**.

In the final step of the algorithm the horizontal portions of each wire are brought down to the bottom of the channel.

- Routes that had ended on layer L2, (routes **b**, c, and **e**) are brought on layer L2 below the lowest L1 wire in each column. An L1–L2 contact is placed below the lowest L1 wire or at the end of the horizontal L2 wire if there is no L1 wire below it. Finally an L1 wire connects the contact to the bottom of the channel if the contact is not already at the bottom of the channel.

- Routes that end on layer L1 are connected to the bottom of the channel in layer L1 if the horizontal portion is not already at the bottom of the channel.

Figure 3.37 shows the channel after the final step has been completed.

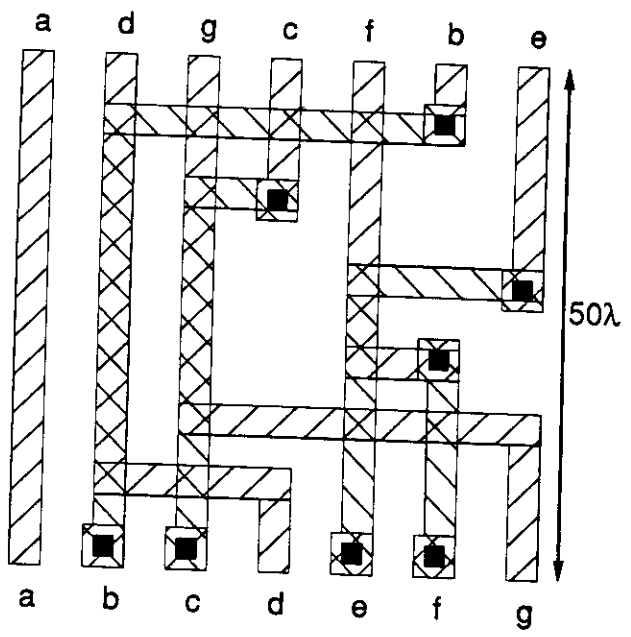


Figure 3.37: Simplified Routing Step 3 for $\times$ 5 in Residue 1-of-7

The Detailed Permutation Routing Algorithm

If this algorithm were executed exactly as described above, it would run in time proportional to the number of routes, but there would be empty space in the channel that could be used more effectively. One could perform the simplified version of the algorithm and then reduce the height in each column by squeezing out all the extra space. However, this would require more complicated data structures. Instead, while the full version of the algorithm is placing the wires, it checks for the placement of existing wires in each column and then places new wires at the highest possible point in each column. Figure 3.38 shows how the routes in Figure 3.32

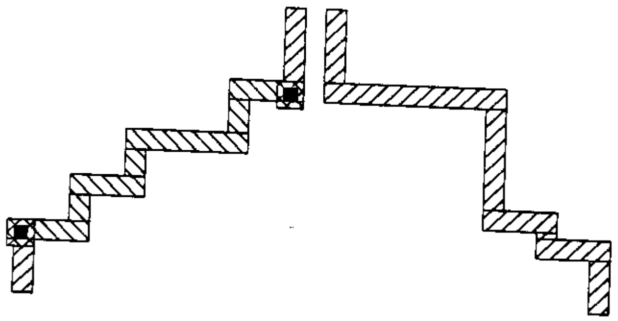


Figure 3.38: Compressed Route-Sinister and Route-Dexter

might look like after the optimization. As the figure illustrates, this leaves more room below the routes (y-coordinate, not layer) for more wires in the same space.

The example in the previous section is now examined in detail. The routes-sinister along the bottom of the channel are examined from left to right. In this example, the routes-sinister along the bottom of the channel are in columns 0, 1, 2, and 4. These routes are partially routed by beginning at the top of the channel. Figure 3.39 shows the channel after the first step has been completed. Route a is in column 0 at the top and bottom of the channel. There is no L2 in this column, so an L1 wire is brought down from the top of the channel far enough to avoid any other L2 in the circuitry that might lie above the top of the channel. (There is no L2 in this column, so a minimum length L1 wire is created.)

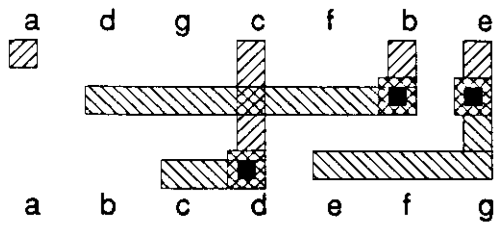


Figure 3.39: Routing Step 1 for × 5 in Residue 1-of-7

After the stub for route zero is placed, route b is considered. The port in column one at the bottom of the channel has to be connected to the port in column five at the top. There is no L2 wire in the third column, so a stub in column five is drawn down far enough to avoid L2 in the circuitry above it. Because this route spans several columns, a contact is placed at the end of the stub and an L2 wire is placed between the fifth and first column.

The route c (which is a route-sinister) is the next one to be considered since its bottom port lies to the right of the ones already routed. Here, the horizontal L2 wire for route b is the lowest L2 in column two, so another horizontal L2 wire (for route c) has to lie below it (route one already occupies the highest possible row). An L1 wire is drawn from the port at the top of column three down below the lowest L2. A contact is placed at the end of this wire and an L2 wire is drawn to column two.

The final route-sinister is route e. This route begins in the sixth column on the top of the channel and does not have any L2 below it. The contact is placed at the same height as for route b. Since the L2 wire cannot run left in this row, it is brought down to the next available row and then brought left to column four.

In the second step, the routes-dexter are brought down from the top port in L1 to a free row and then routed rightwards to the column for the bottom port. Figure 3.40 shows the channel after this step is performed. The routes-dexter are

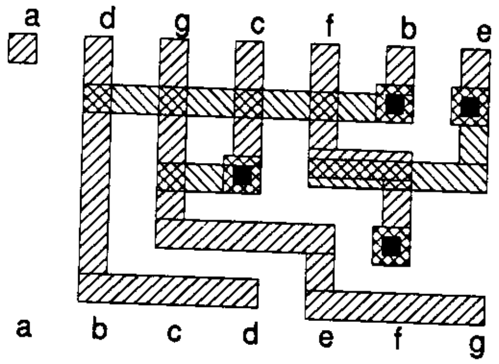


Figure 3.40: Routing Step 2

scanned from right to left, in the order of their top ports. In this example these routes are examined in the order **f**, **g**, and finally, **d**. Route **g** has to run from the top of column four in L1, to the bottom of column five. Therefore the L1 wire in column four must be brought down low enough to avoid the L1 in the contact in column five. The wire is not brought down any farther at this time, nor is a contact placed because it is not yet known that a contact is needed to allow route f to cross route **g**.

Route **g** similarly avoids the contact in column three by running below it.

Before a route-dexter crosses a column, there is a check to see if any incomplete routes-dexter end in that column. The route in column five does, so a contact has to be placed at the end of route five. This contact must be placed below the L2 in that column from route **e**, and an L1 wire must connect the end of route five to this contact. Route g can then avoid this contact by running below it and rightwards to column six. Route d needs to avoid only the L1 in route g.

In the third step, the lower vertical segments of all the wires are routed down to the source ports. The height of the channel is determined in this step by noting the height of the channel if the necessary contacts were placed. After the height is determined, the contacts are actually placed and wires drawn to connect the routes to the bottom of the channel. Figure 3.41 shows the channel after this step is performed. The layer last used in each route and the destination layer

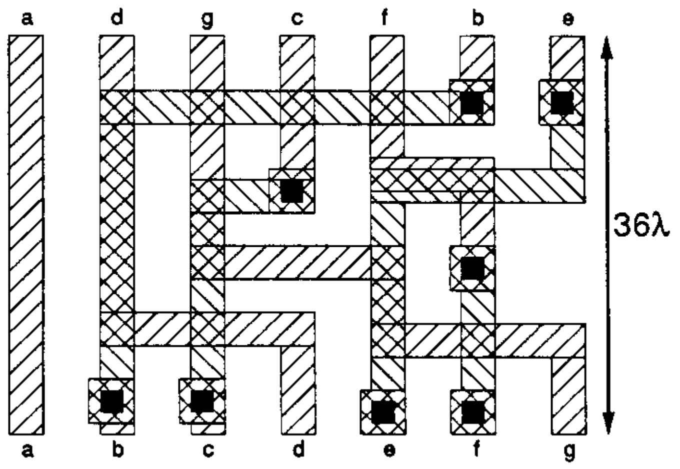


Figure 3.41: Routing Step 3

at the bottom of the channel determine the layer for the wire that connects the partial route to the bottom of the channel. In this example, it is assumed that the destination layer is L1. If the route ends in L1, there is no lower L1 in that column. (Step two places a contact when a route-dexter would block an L1 wire.) Routes a, d, and g are examples of routes that end in L1.

The remaining routes end in L2 or an L1–L2 contact. These routes require an

L2 wire down below the lowest L1 in that column so that a contact to L1 can be placed. If necessary, the contact and the bottom of the channel are connected with an L1 wire.

The height of this channel is 36λ versus 48λ for the simplified approach, illustrating that the area is reduced significantly (25%) without a large time penalty for doing the compression.

3.3.11.3 One-to-Many Channel Routing

Permutation routing is necessary and sufficient when there is a one-to-one mapping between the ports on the top and bottom of a channel. The functions defined in the synthesis language do not necessarily have this property. Each of the formal parameters in a function is unique. If each formal parameter is referenced only once in the function body, a one-to-one mapping exists between the formal parameters and the points of reference. If, instead, parameters are used more than once, there is a one-to-many correspondence between the function's formal parameters and the actual use of the parameters in the function body.

Definition 9 (Multi-ported signal) *A signal associated with more than one port on one side of a channel is called a multi-ported signal.*

The one-to-many mapping is routed in two steps. First the multi-ported signals are connected on the top side of the channel. New ports are created on these wires, and then a permutation route is used to connect these new ports to the ports on the bottom of the channel[2].

An algorithm to connect the multi-ported signals that runs in n^2 time, where n is the number of ports on the top of the channel, is now presented. As an example, the ports (0–12) at the top of the channel are

d a d g a b e c c b e f e

and the formal parameter order (implemented by the ports 0–6 on the bottom of the channel) is

a b c d e f g.

The channel routing algorithm incorporates the permutation routing algorithm given in the previous section, and so we also assume that the ports on the top and bottom of the channel make their signals available on layer L1. It is also assumed that the permutation routing algorithm uses layer L1 to route the vertical wire segments that go to the top of the channel.

1. First, a list is made of the ports associated with each signal. The first element in each list is the leftmost port for that signal. The succeeding elements in each list are the remaining ports for that signal in the order that they appear at the top of the channel. The lists for the example are shown in Table 3.2.

2. A second list is made of the lists of multi-port signal in the relative order of the positions of the first elements in each multi-port list. Table 3.3 shows this list for the example.

[2]A many-to-many mapping is routed by repeating the first step on the bottom of the channel. A permutation route can then be used to connect the signals that have to cross the channel.

Table 3.2: Lists of Ports

d:	0	2	
a:	1	4	
g:	3		
b:	5	9	12
e:	6	10	
c:	7	8	
f:	11		

Table 3.3: Lists of Ports

→	d:	0	2	
	a:	1	4	
	b:	5	9	
	e:	6	10	12
	c:	7	8	

3. The height of the highest possible occurrence of L1 and L2 in each column is noted. It is assumed that the y-coordinate of the top of the channel is 0. For the purpose of this example, the worst case is assumed—that there is unrelated L2 at the top of the channel in each column. This means that the next occurrence of L2 in each column cannot be any higher than the minimum separation between unrelated L2 wires below the top of the channel. For scalable CMOS, metal-2–metal-2 separation is 4λ. The L1 and L2 heights are shown in Table 3.4.

Table 3.4: Initial Heights of L1 and L2 Layers in Each Column

	0	1	2	3	4	5	6	7	8	9	10	11	12
L1:	0	0	0	0	0	0	0	0	0	0	0	0	0
L2:	−4	−4	−4	−4	−4	−4	−4	−4	−4	−4	−4	−4	−4

4. Multi-ported signals are routed by scanning this list of lists of ports as long as it is not empty. Lists of ports are removed from this list of lists when they

have been routed. The first list of ports in the list is routed first. This is route d in the example. A wire has to be routed between columns 0 and 2. The horizontal portion of this wire must be routed on layer L2 so that the vertical L1 wire in column 1 can pass under it. The vertical portions of this wire should be routed on layer L1 so that it can connect to the ports at the top of the channel and so that it can cross other, unrelated, horizontal wires that are on layer L2. Since the highest occurrence of new L2 is at $y = -4$, the horizontal L2 wire is placed with its uppermost edge at $y = -4$. L1–L2 contacts are placed with their uppermost edges at this same height. The minimum separations are then subtracted from the y-coordinates of the lower edges in each column. The heights of each layer in each column after this step is shown in Table 3.5.

Table 3.5: Heights of L1 and L2 Layers in Each Column After Step 4

	0	1	2	3	4	5	6	7	8	9	10	11	12
L1:	−11	0	−11	0	0	0	0	0	0	0	0	0	0
L2:	−12	−11	−12	−4	−4	−4	−4	−4	−4	−4	−4	−4	−4

5. In an attempt to reduce the height of the multi-port routing, as many signals are routed in each horizontal track as possible. After a signal has been routed, the next signal chosen to be routed is the first one that begins in a column to the right of the one just routed. After signal d is routed, it is removed from the list of lists of ports. The list of lists then begins with signal a, which begins in column 1, which is to the left of the rightmost port of signal d. The first signal in the list which is to the right of the rightmost port of signal d is

signal b, which begins in column 5. The ports for this signal are independent of all previously routed signals, and is routed similarly as for signal d. The heights of each layer in each column after this step is shown in Table 3.6.

Table 3.6: Heights of L1 and L2 Layers in Each Column After Step 5

	0	1	2	3	4	5	6	7	8	9	10	11	12
L1:	−11	0	−11	0	0	−11	0	0	0	−11	0	0	0
L2:	−12	−11	−12	−4	−4	−12	−11	−11	−11	−12	−4	−4	−4

6. After signal b is routed, it is removed from the list of lists of ports as is shown in Table 3.7. None of the remaining ports in the remainder of the list (after

Table 3.7: Lists of Ports after Step 5

	a:	1	4	
→	e:	6	10	12
	c:	7	8	

the arrow) begins to the right of the rightmost port for signal b. (Because the signals in the list of lists are ordered by their leftmost ports, none of the signals above the arrow need to be considered.) As many signals have been routed on the first horizontal track as could be, so the list of lists is re-examined from its beginning. The first entry in the list is signal a, beginning in column 1. L1–L2 contacts need to be placed in columns 1 and 4, so the arrays of L1 and L2 heights are examined The lower of the two values in each column are −11 and −4, for columns 1 and 4, respectively. An L2 wire is drawn from this new contact in column 1 to column 3, and then up to the minimum of the minimum heights in column 3 and the bottom of the new contact in column 4. Figure 3.42 shows the channel after this signal has been routed. The heights of each layer in each column after this step are shown

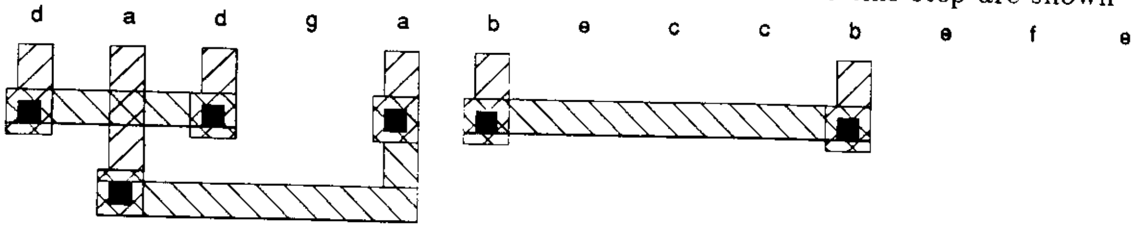


Figure 3.42: Channel After Signals d, b, and a Have Been Routed

in Table 3.8.

Table 3.8: Heights of L1 and L2 Layers in Each Column After Step 6

	0	1	2	3	4	5	6	7	8	9	10	11	12
L1:	−11	−18	−11	0	−11	−11	0	0	0	−11	0	0	0
L2:	−12	−19	−19	−19	−12	−12	−11	−11	−11	−12	−4	−4	−4

7. Signal a is removed from the list of lists, making signal e the next signal available for routing. It is routed in the same way that signal a was, placing the wires and contacts as high as possible. Figure 3.43 shows the channel after signal e has been routed. The heights of each layer in each column after

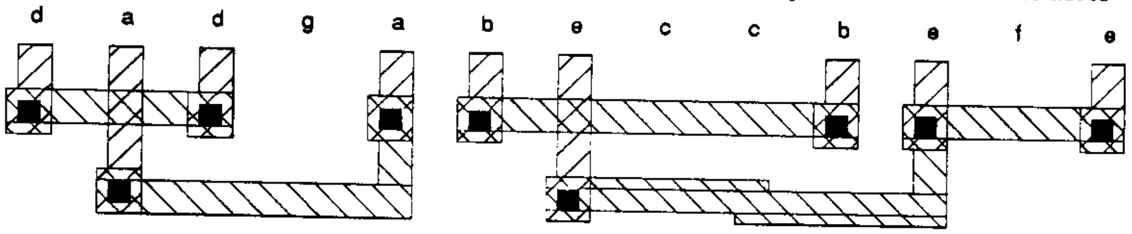


Figure 3.43: Channel Routing After Step 7

this step are shown in Table 3.9.

Table 3.9: Heights of L1 and L2 Layers in Each Column After Step 7

	0	1	2	3	4	5	6	7	8	9	10	11	12
L1:	−11	−18	−11	0	−11	−11	−18	0	0	−11	−11	0	−11
L2:	−12	−19	−19	−19	−12	−12	−19	−18	−19	−19	−19	−11	−12

8. After signal **e** is routed, its entry is removed from the list of lists, leaving only signal **c**. Signal **e** ended in column 12. Signal **c** begins in column 7, and so cannot be routed on the same level as signal **e**. As many signals have been routed on this horizontal track as could be, so the list of lists is re-examined from its beginning. Signal **c** is the next (and only) signal available for routing. Normally, this signal would be routed in the third horizontal track in columns 7 and 8; however a special condition exists. These two ports are adjacent and so a horizontal wire can connect these two ports on layer L1 without blocking any ports that might otherwise have lain between them. The heights of each layer in each column after this final step are shown in Table 3.10. The routing of the multi-port signals is now complete, as is

Table 3.10: Heights of L1 and L2 Layers in Each Column After Step 8

	0	1	2	3	4	5	6	7	8	9	10	11	12
L1:	−11	−18	−11	0	−11	−11	−18	−9	−9	−11	−11	0	−11
L2:	−12	−19	−19	−19	−12	−12	−19	−18	−19	−19	−19	−11	−12

shown in Figure 3.44.

9. All the ports in the multi-ported signals have now been connected together. Any of the columns in which a multi-ported signal lies can be chosen as the top port for that signal. Table 3.3 shows the signals that have multi-

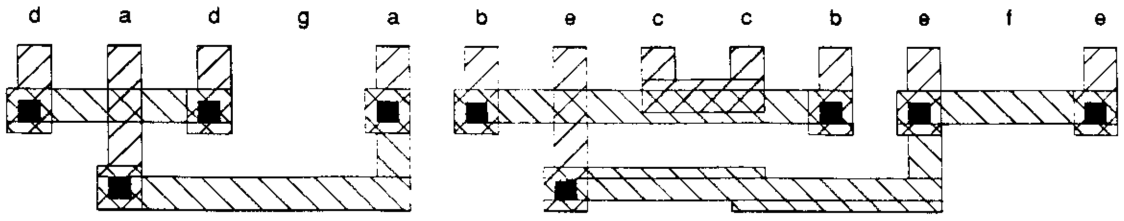


Figure 3.44: Completed Multi-Port Signal Route

ple top ports. The top ports chosen should be the closest ones (horizontal distance) to the ports on the bottom of the channel. Since the positioning of the bottom ports affects how long the wires are, some tests have to be made to find these positions. This involves sliding the wires at the bottom of the channel to minimize the horizontal portions of the wires. The total number of positions for the wires at the bottom of the channel is given by the binomial coefficient $\binom{t}{b}$, where t and b are the number of ports at the top and bottom of the channel, respectively. This term grows exponentially as the $t - b$ increases. To reduce the number of cases that need be considered, a simplification is introduced. It is assumed that the ports at the bottom are grouped together and can be slid from left to right. While this may not yield the best results in all cases, this requires only $t - w$ tests and gives good results quickly.

A good metric appears to be given by summing the differences between the x-coordinates of the ports on the bottom and the x-coordinates of the closest corresponding ports on the top of the channel. Table 3.11 shows the results of shifting the bottom ports varying distances. The left part represents the

Table 3.11: Top and Bottom Port Alignment Tests

dadgabeccbefe	a	b	c	d	e	f	g	sum
abcdefg......	1	4	5	−1	2	6	−3	22
.abcdefg.....	0	3	4	−2	1	5	−4	19
..abcdefg....	−1	2	3	−3	0	4	−5	18
...abcdefg...	1	1	2	−4	−1	3	−6	18
....abcdefg..	0	0	1	−5	±2	2	−7	17
.....abcdefg.	−1	−1	0	−6	1	1	−8	18
......abcdefg	−2	−2	0	−7	0	0	−9	20

ports on the top and bottom of the channel. The first line shows the ports at the top of the channel. The remaining lines show the bottom ports shifted right by one port on each succeeding line. The next seven columns show the horizontal separations for each of the ports. The next column is the sum of the absolute values of these separations of the seven ports. The goal is to minimize this number, which is the sum of the lengths of the horizontal components. This table shows that shifting the bottom ports four positions to the right gives the minimum total horizontal separations.

10. Now the bottom ports have been positioned, and there are unique to ports at the top of the channel. The permutation routing algorithm from Section 3.3.11.2 can now be applied. This algorithm requires that each signal be classified as *sinister* (definition 7 on page 83) or *dexter* (definition 6). This information is obtained from the calculation in the previous step. The sign of the horizontal separation calculated for each port indicates whether the corresponding wire is sinister (non-negative) or dexter (negative). The resulting route is shown in Figure 3.45.

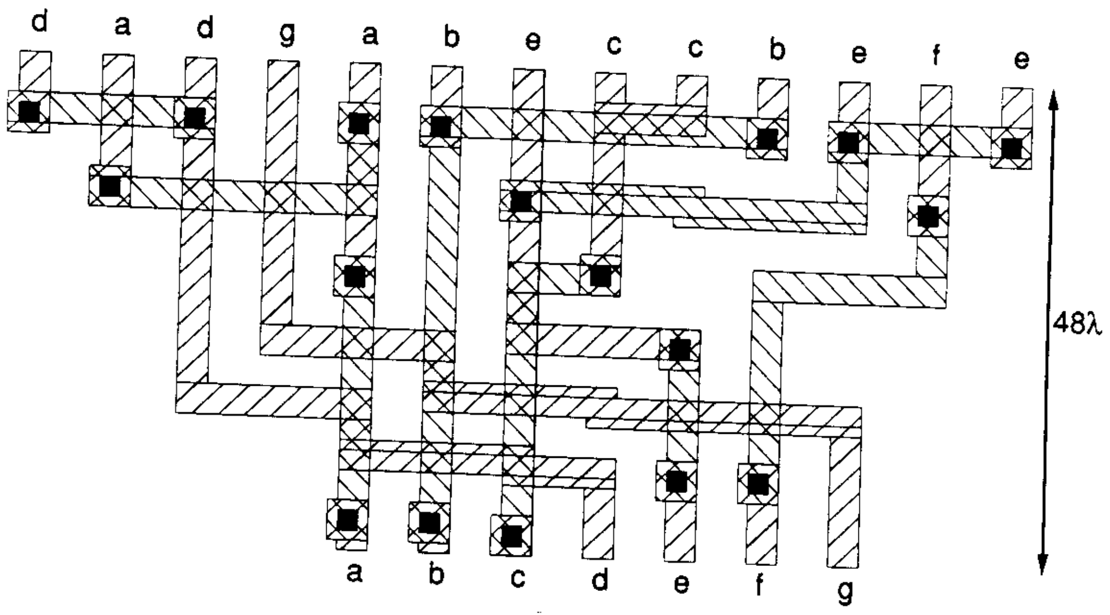


Figure 3.45: A Channel Route for a One-to-Many Function

3.3.11.4 Routing for *n*-ary Trees

The routing methods just discussed are now applied to the placement methods shown in Sections 3.3.6 and 3.3.7. For example, consider the function body in Section 3.2.3 as part of the function definition shown in Figure 3.46. For the

```
FUNCTION f = (INT a, b, c, d, e) INT:
BEGIN a*b + (c*d +e)
END
```

Figure 3.46: A Function Definition

purpose of illustration, the formal parameters for the function f are declared in the reference order.

Figure 3.47 shows the general layout for the hierarchical layout of a *n*-ary tree. The parent cell corresponds to the parent node. The sub-cells corresponding to the sub-nodes are placed below the parent node with enough room for the power

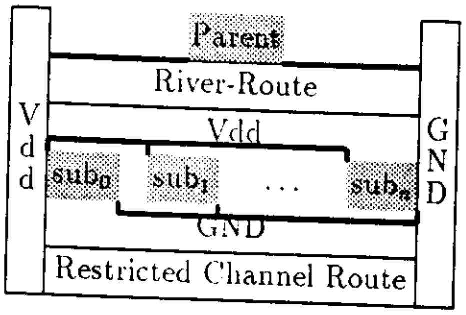


Figure 3.47: Floor Plan for an n-ary Tree

routing (V_{DD}) and the signal wires between the parent and the sub-cells. Below the sub-cells lies the ground wiring. Below that lies wiring that permutes the inputs to the sub-cells and the parent so that they are in the same order as the formal parameters to the containing function. This entire complex is formed into a cell that has V_{DD} on the left and ground on the right.

All the levels in the hierarchy are formed in the same manner, always creating a rectangular cell that has V_{DD} on the left, ground on the right, inputs on the bottom, and outputs on the top.

To place the cells for the function given in Figure 3.46, the subexpression (c*d+e) is laid out first. The multiplier $*_2$ corresponds to sub_0 and the adder $+_2$ is the parent cell. In the next level up in the hierarchy, (a*b+$cell_1$), this cell is sub_1. The multiplier $*_1$ corresponds to sub_0 and the adder $+_1$ is the parent cell. The layout for the complete tree is shown in Figure 3.48.

In Figure 3.46 the formal parameters to the function were declared in the same order as they were referenced in the function body. This allowed the routing to be simple (no wires crossing, no change in wire order), but the formal parameters are not always referenced in the declaration order. If, the function is altered slightly

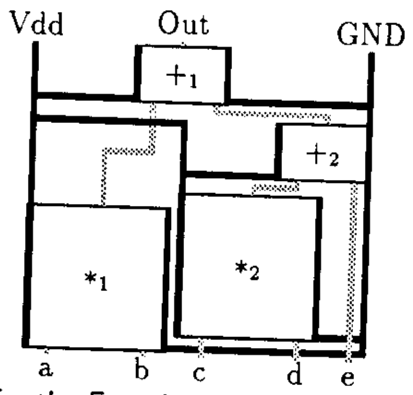


Figure 3.48: Floor Plan for the Function in Figure 3.46 with Power and Ground

and the order of the formal parameters is reversed, as in Figure 3.49, the same layout becomes more difficult to route.

```
FUNCTION f = (INT d, c, b, a) INT:
BEGIN a*b + (c*d +c)
END
```

Figure 3.49: Another Function Definition

A possible solution to this routing problem is to perform a channel route of the ports on the top of the channel that effectively puts these ports in the same order as that of the formal parameters.[3] Figure 3.50 shows an input permutation for the inputs to the cell that corresponds to $*_2$. This cell needs inputs in the order c, d

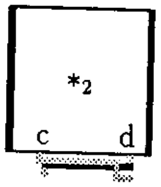


Figure 3.50: Input Wire Permutation for $*_2$

and they are made available as d, c. A permutation route is needed to put the connections in the right order. In the hierarchical routing, it is not known which

[3]This discussion assumes that the cells cannot be mirrored about their y-axes, and that the inputs to the cells are not commutative. Given enough inputs to a cell, it is possible that mirroring a cell may make routing worse.

wiring constraints will be imposed later from higher levels in the hierarchy, so the wires are brought out in the same positions as in the inputs to the cell.

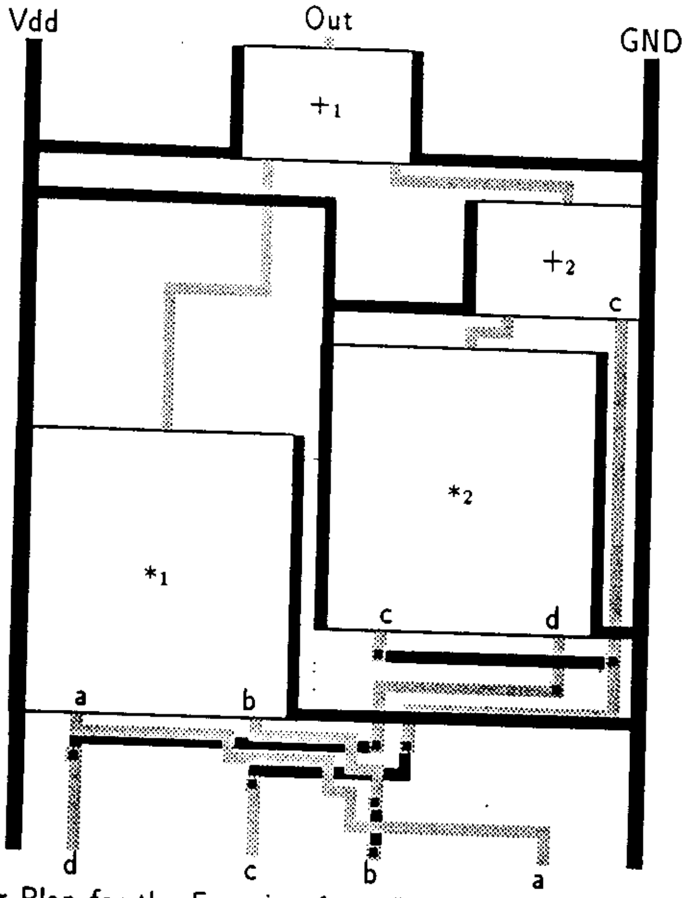


Figure 3.51: Floor Plan for the Function from Figure 3.46 with Power, Ground, and Permuted Inputs

Figure 3.51 shows the placement and routing of cells implementing the function shown in Figure 3.49. The routing of inputs to $*_2$, c and d had to be reversed to match their order in the formal parameter list. The next level up in the hierarchy (c*d+c) requires inputs that are available in the order d, c. The input c has to be merged with the inputs to the previously routed cell, d, c to yield the wires in the order d, c. This is done with the one-to-many channel-routing algorithm from

Section 3.3.11.3. The next sub-expression, a+b, has the inputs reversed so that at the next level up in the hierarchy the inputs require inputs in the order b,a,d,c. In the top level of the hierarchy, these wires are permuted to d,c,b,a, matching the formal parameter order. These last two steps are also performed with the one-to-many channel routing algorithm. Since there are no multi-ports in the last two steps, only the permutation routing portion of the channel-routing algorithm is performed.

The above discussion does not take into account the advantage of commutativity of the operators. Without exploiting commutativity, local routing permutations of wires to match the order of the formal parameters require no more horizontal wiring channels than if the routing permutations were done for the entire complex of placed cells. The vertical area required for these horizontal channels can be absorbed in the space taken by the associated subcell if the adjacent subcell is taller than the subcell plus the wiring permutation.

3.3.11.5 Routing for Conditional Expressions

There are two kinds of conditional expressions. These are the **IF** and the **CASE** expressions. The former selects one of two expressions based on the value of a Boolean expression. The **CASE** expression selects one of several expressions based on an integer value.

Figure 3.52 shows the general form of an **IF** expression. In this expression, a is an arbitrary Boolean expression and b and c are arbitrary expressions that are

```
IF a
THEN b
ELSE c
FI
```

Figure 3.52: General Form of an **IF** Expression

of the same datatype. Since the **IF** expression yields either the value from the **THEN** part or from the **ELSE** part, the **IF** expression has the same datatype of the **THEN** and **ELSE** parts of the expression. A floor plan for this **IF** expression is shown in Figure 3.53.

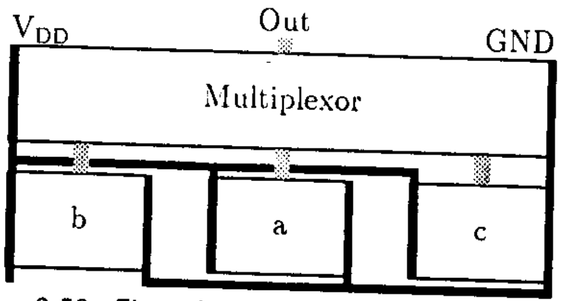


Figure 3.53: Floor Plan for a General **IF** Expression

The **CASE** expression selects one of several expressions based on the value of an arbitrary integer expression.

Figure 2.7 on page 36 shows the general form of an **CASE** expression. In this expression, sel is an integer expression and a, b, c, d, and e are arbitrary expressions that are of the same datatype. If the selection value (sel) matches one of the cases (in this example, an integer between zero and three), then the value of the associated expression is yielded as the value of the **CASE**. If none of the values match, the value of the expression associated with the **OUSE** is yielded as the value of the **CASE**. The **CASE** expression has the same datatype of all of its parts (including the **OUSE** part). A floor plan for this **CASE** expression is shown in Figure 3.54.

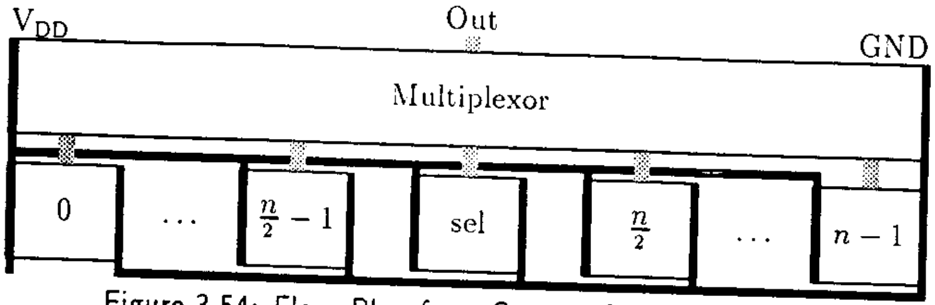


Figure 3.54: Floor Plan for a General **CASE** Expression

3.4 Estimating Non-Behavioral Attributes

The ALICS Synthesizer creates circuitry that implements a given algorithm and data types. In order to be useful as a prototyping tool, the synthesizer must be able to provide measurements or estimates that faciliate the comparison of different designs or provide a design quickly enough that existing estimators can be used. Non-behavioral attributes provide a basis for comparisons [Clin84]. Generally, designs are better if they require less area, less power, and have lower delays. Unfortunately, changing a design so that one or more of the attributes are lowered may raise others. The ALICS Synthesizer enables a designer to change a design and then see the effect of the changes on the non-behavioral characteristics.

The hierarchical structure of the language and the target circuitry makes it easy to estimate these non-behavioral characteristics. The ones that will be examined are *area*, *power*, and *delay*.

3.4.1 Area

The computation of the area of a circuit that implements an Algol algorithm is done hierarchically by the routines that place and route the circuitry. These routines are part of the Chisel routines [Karp83]. These routines create the CIF descriptions of the masks and also maintain a bounding box for the circuitry created so far. As mask entity or an already defined cell is added to a current cell, the bounding box for the current cell is updated.

Thus the area of a circuit is estimated by creating the CIF description of the circuit and noting the size of the bounding box.

3.4.2 Power

The power is determined by summing up the V_{DD} current required for each cell in the design and multiplying by the operating voltage. Associated with each port in a chisel cell is the maximum current needed by (driven by) the port. The current in CMOS circuits is determined by the charge required to charge capacitive gates and wires. The current is given by

$$I = fC\Delta V,$$

where f is the operating frequency, C is the capacitance, and ΔV the change in the voltage [Mukh86]. The current increases proportionally with the capacitance of gates and wires.

3.4.3 Delay

Data flows through the function inputs to the function outputs. If self-timed circuitry implements the functions, then data-dependent delays will be seen. If combinational circuitry implements the functions, then clocking data through the circuitry will have to be done at the worst-case delay through the circuit.

The delay through the circuit can be computed through standard techniques. A CIF file is produced by the Algol synthesizer. The features in the CIF file are easily converted into input for a timing simulator like *crystal* [Oust83,Oust85] or an electrical simulator like *spice* [Vlad81,Quar86], and the timing results quickly obtained.

3.5 Validity of the Approach

The layout methods described in this thesis have been applied to several designs. The ALICS descriptions and their layouts are shown in Chapter 5. The translation process is shown in the next chapter to be rapid and scale well for large designs. Most of the analysis requires time proportional to the size of the design, but routing requires time proportional to the square of the number of wires that need to be routed at any given time. While n^2 seems lengthy for large n, The hierarchical nature of the design process reduces the number of wires that need to be considered at any given time, so that n^2 is for a small n.

It is useful to find out how a manually laid out design compares with the ones produced by the ALICS Synthesizer. To compare the two layout methods fairly, some rules have to be imposed on the manual technique. This is because many techniques that are done manually can be performed on the cells used by the synthesizer. For example, ALICS Synthesizer's cells use complementary logic. Using pre-charged logic in the manual designs would not give a valid comparison with the automatic technique, because the use of cells with pre-charged logic is not prohibited in the automatic method.

The rules for converting an automatically laid out design to a manual design are as follows.

1. The transistor schematics for the primitive cells must be the same in both layout techniques.

2. The corresponding transistor sizes must be the same in both techniques.

3. Corresponding wire lengths may differ in the two techniques.

4. The hierarchy may be flattened in the manual technique.

5. Substrate contacts may be added or removed in the manual technique, but there must be enough contacts for the circuit to function properly.

6. The power and ground wiring may have different topography in the two techniques. This includes sharing the power buses in adjacent cells.

7. The manual technique need not restrict the placement of a cell's inputs and outputs to the bottom and top of a cell.

These rules were followed to manually compact the ripple-carry adder described in Section 5.5. This design was chosen because the automatic layout appears to waste a lot of area. Rather than lay the cells side-to-side horizontally, the automatic method attempts to reduce lengths of the wires that go from cell to cell. The layout of the manual design of an eight-bit ripple-carry adder is shown in Figure 3.55. The areas and power requirements of the manually laid-out design are shown in

Figure 3.55: Manually Laid Out Ripple-Carry Adder

Table 3.12. The corresponding areas and delays for the automatic method are

Table 3.12: Areas and Delays for Manually Laid-Out Ripple-Carry Adders

# of Bits	x (λ)	y (λ)	Area (λ^2)	Delay (nSec)
1	147	82	12054	10.14
2	302	88	26576	14.19
3	457	98	44786	22.54
4	612	98	59976	30.89
5	767	104	79768	39.25
6	922	110	101420	47.60
7	1077	110	118470	55.95
8	1232	116	142912	64.30

given in Table 5.3 on page 145. Table 3.13 shows a comparison of these features. The numbers are the percent increase for the automatic design over the manual design. For example, the automatic layout of the eight-bit ripple-carry adder is 2.58% slower (by 1.66 nanoseconds) than the manually laid-out one, but is larger by 260% (3.6 times as large). So the automatic layout produces circuits that are

Table 3.13: Comparison of Areas and Delays for Automatically Laid-Out and Manually Laid-Out Ripple-Carry Adders

# of Bits	Delay % Increase	Area % Increase
1	−8.38	100
2	−4.58	129
3	−1.15	144
4	0.39	182
5	1.27	200
6	1.85	216
7	2.27	247
8	2.58	260

less area-efficient than the manually laid-out counterparts, but the delay estimates are comparable.

3.6 Summary

A method has been given for efficiently analyzing an ALICS source program. This consists of lexical analysis, parsing, and the transformation of these expressions into a structural representation of the program. This structure is mapped onto a tree of interconnected cells that implement the functions in the program. A method has been demonstrated for placing cells hierarchically and for routing functions and conditional expressions. Efficient methods have been defined for routing of permutations and one-to-many mappings. These mappings are needed for the routing of signals from a function's formal parameters to the points where the parameters are referenced. Because the methods are fast, it is easy for the designer to experiment with changes to an algorithm and data types and then measure the non-behavioral attributes of the resulting designs. These measurements can be used to select the best of several designs for more thorough examination.

The automatic layout is less area-efficient than a comparable manual layout. However, the delay estimates obtained from the automatic designs are within a few percent of the manual designs. This is significant because it means that this technique is worthwhile for the design of prototypes. Once a design with the desired delays is obtained, the area of the design can be reduced by manual editing or other lengthy techniques.

CHAPTER 4

Time Complexity of the Synthesis Method

The time complexity of the major portions of the Algol synthesizer, important for creating prototypes, are discussed in this chapter. To be useful, a prototyping system must work quickly. The time complexity gives a good indication of how long it will take for a chip design to be synthesized from an algorithmic description. The complexity gives a more important indication of how much longer a larger design will take to be synthesized. A synthesizer might operate quickly for small designs, but if its complexity is exponential it may be infeasible to compare design alternatives if the circuits are large. Less complex synthesis algorithms allow large designs to be synthesized faster and so more design alternatives can considered by a designer.

The complexity of the synthesizer is obtained by determining the complexity of the different parts of the synthesizer. A picture of the flow-of-control in these parts is shown in Figure 3.1 on page 50. The major modules in the synthesizer are the lexer, parser, structure-tree traversal, layout, and wiring.

4.1 Lexer

The lexer, written in the lexical analysis language, *lex*[Lesk75], reads characters from the input Algol program and partitions the characters into indivisible words called tokens. The time taken by a *lex* program is proportional to the number of characters in the input.

The symbol table created by the lexer and the parser is implemented with a hash table. The time taken for hash table entry and retrieval is a constant if the table is not full. If the table is full, the overflow buckets are organized in a balanced binary tree. Entry into and retrieval from balanced binary trees is $\log_2$ of the number of items in the overflow bucket.

4.2 Parser

The parser, written in the compiler-compiler language *yacc*[John78], reads tokens from the lexical analyzer and activates semantic routines when grammar rules have been recognized. The time required to parse the input (classify tokens according to the rules of the grammar) is bounded by the number of input tokens times the number of productions in the grammar. Since the number of grammar productions is constant, parsing is proportional to the number of input tokens.

4.3 Structure Tree Construction

Construction of the structure tree is formed as the input is parsed. The tree is a condensed form of a parse tree. The formation of each node in the tree requires a constant amount of time. Entries that are made in the symbol table for function and variable definitions require constant amount of time before the hash table fills, and require $\log_2$ of the number of items in the overflow bucket when a hash table entry fills.

4.4 Structure Tree Traversal

After the complete program has been parsed and the structure tree has been built, the structure tree is traversed to perform the layout and wiring. The time required to traverse the tree is proportional to the number of nodes in the tree, since each node is visited only once.

During the traversal of the structure tree, the nodes are interpreted. The interpretation of a node might involve iteration, function expansion, or recursion expansion. The number of iterations and levels of recursion is program-dependent, must be finite, and has to be analyzed on a program-by-program basis. Interpretation of each of the nodes requires constant time.

4.5 Layout

Layout is performed while examining each node in the structure tree. Each type of node has a corresponding layout template. The layout of each template is computed in time proportional to the number of subnodes plus one for the node itself.

4.6 Wiring

4.6.1 Terminology and Assumptions:

W The total number of wires in the river route.

wiring order The wiring order is assumed to be from bottom-to-top and left-to-right, for the purpose of discussion.

wire bundles Bundles are groups of adjacent wires in a river route that can be routed in a similar manner, either from lower-left to upper-right (bundles-sinister) or from lower-right to upper-left (bundles-dexter). Wires that run straight across the channel without any bends are degenerate cases of both types. All bundles consist of adjacent bottom ports that need to be routed to adjacent top ports in the same order. Bundles-sinister consist of pairs of connected ports where the top port of each pair lies above and to the right of the bottom port. Adjacent bundles alternate between bundles-sinister and bundles-dexter and can be routed independently. Adjacent bundles of the same type are merged to form a single bundle. Each wire in a bundle-sinister consists of a list of points whose x and y coordinates increase monotonically.

w The number of wires in a bundle. $w \leq W$

b_i The number of bends in wire i in a bundle. This counts bends to the right and bends up (see Figure 3.29).

$$\begin{aligned} & b_1 && = 2 \\ 2 \leq\ & b_2 && \leq b_1 + 2 = 4 \\ & \vdots \\ 2 \leq\ & b_i && \leq b_{i-1} + 2 \leq 2i \\ & \vdots \end{aligned}$$

4.6.2 Determining Channel Height–Previous Work

Dolev, et al[Dole81], discuss a method for determining the height of a channel in $\mathcal{O}(w)$ time. This method, and its shortcomings, is summarized in the following paragraph.

Wires and ports are represented by lines that lie on a fixed grid (see Figure 4.1), where the grid size is the sum of the minimum wire width

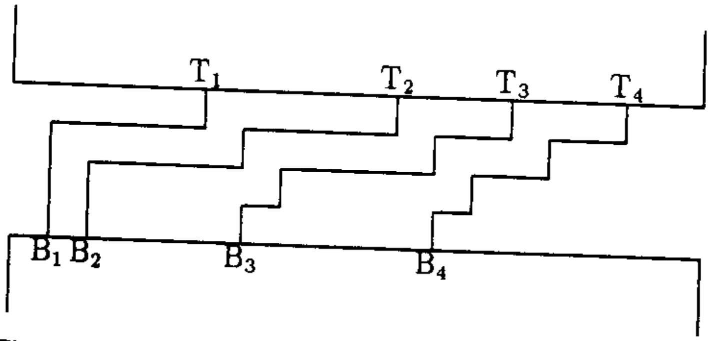


Figure 4.1: River Routing Expressed as Lines on a Grid

and minimum wire separation. The channel width is determined by calculating a *conflict number* $W(i,j)$ for all pairs of ports B_j and T_i. (B_j and T_i indicate the x coordinates of port B_j and T_i, respectively.)

The conflict number indicates how many wires pass between the pair of ports and contribute to the height of the channel. If the ports are far enough apart, or directly above one another, then they do not contribute anything to the height of the channel. If they are close enough, $|j - i| + 1$ wires must pass horizontally between the ports. $W(i, j) = 0$ in the following cases:

$$i = j \quad \text{and} \quad T_i = B_j. \tag{4.1}$$

$$i < j \quad \text{and} \quad T_i - i \leq B_j - j. \tag{4.2}$$

$$i > j \quad \text{and} \quad T_i - i \geq B_j - j. \tag{4.3}$$

In all other cases the conflict number $W(i, j) = |i - j| + 1$.

To find the minimum channel height, they let c_j be the smallest $i \leq j$ such that $B_j - T_i \leq j - i$ (or j if no such i). j is then incremented from 1 to n, searching c_j (starting at $c_j - 1$) and computing $W(c_j, j)$. This process requires $\mathcal{O}(w)$ time.

4.6.3 Problems

The preceding method does not work if some of the wire widths are different. In practical cases, some of the wires will have to be created wider than others in order to accommodate higher electrical currents or lessen voltage drops [Chow86] (see Figure 4.2). If the grid size in a channel is enlarged to accommodate the largest wire in that channel, the method will still require $\mathcal{O}(w)$ time, but will require an unnecessarily large channel.

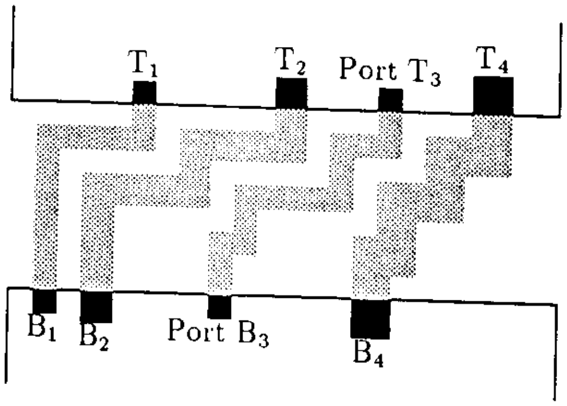


Figure 4.2: River Routing with Differing-Width Wires

To produce a minimum-height channel, it is necessary to do additional work. We will denote the x coordinate of the left and right sides of a port B_j by B_{jl} and B_{jr}. The width of the jth wire is W_j. The wire separation is s. The above equations become:

$$i = j \quad \text{and} \quad B_{jl} + W_j \leq Q_{jr}$$
$$\text{and} \quad T_{jl} + W_j \leq B_{jr}. \tag{4.1'}$$
$$i < j \quad \text{and} \quad T_{il} + W_i - (B_{jr} - W_j) \leq \left(\sum_{k=i+1}^{j-1} W_k \right) + (i - j + 1)s. \tag{4.2'}$$
$$i > j \quad \text{and} \quad T_{il} + W_i - (B_{jr} - W_j) \geq \left(\sum_{k=i+1}^{j-1} W_k \right) + (i - j + 1)s. \tag{4.3'}$$

In all other cases the conflict number $W(i,j)$ is given by

$$W(i,j) = \left(\sum_{k=i+1}^{j-1} W_k \right) + (i - j + 1)s.$$

The search for the largest c_j proceeds as above.

Because of the summations in these equations, the time to determine the channel height is actually $\mathcal{O}(w^2)$.

4.6.4 Creation of the Wires

The creation of the wires in a river-route involves determining the channel height (the wires run from the bottom to the top of the channel). This is accomplished in time proportional in $\mathcal{O}(w^2)$. The number of bends in each wire is determined by the number of bends in the wire to its immediate left and above it. The maximum number of coordinates created is proportional to w^2, and therefore, so is the time required.

4.6.5 Signal Routing

The binding of formal parameters to actual references inside function bodies is performed by routing wires from a cell's inputs to the input ports on the subcells that need the parameters. Routing is performed in a channel that lies below the subcells. Inputs to the function are at the bottom of this channel and correspond to the function's formal input parameters, each of which must be unique. The mapping of input parameters to reference order is a one-to-many mapping. The ports at the top and bottom of the channel are on the metal-1 layer so that the wires connected to these ports can pass underneath the power and ground wires which are on the second metal layer, metal-2.

This routing is done in two steps. First the signals with multiple ports at the top of the channel are connected together. For each of the signals with multiple top ports, one of these ports is chosen as a destination port. Then permutation routing is used to route what is now a one-to-one mapping.

The complexity of connecting the signals with multiple ports at the top of the channel will be examined first. To perform this routing requires the construction of the signal name to port list mapping, ordered by leftmost port for each signal, which is implemented as a list of port lists. This list is built in time proportional to the number of ports. The ports at the top of the channel are ordered by their positions along the x-axis. Associated with each port is a signal name which is an index into what will become the list of port lists. This list is initially empty and each listhead (one for each signal) is empty. Each listhead is associated with a signal and contains a pointer (which is initially NIL) to the first port in the list and a pointer (also initially NIL) to the last port in the list for that signal. There are pointers to the first and last listheads for the signal port lists, which are initially empty.

The construction of the list of port lists is performed by scanning the ports at the top of the channel from left to right. This requires time proportional to the number of ports. The signal name associated with each port is used as an index into the listheads for the lists of ports for each signal. If the listhead is empty, a new listhead is appended to the end of the list of listheads. An entry for the port is then appended to the list pointed to by the listhead. Since pointers to the end of the list of listheads and to the end of the lists of signal ports are maintained, this operation requires constant time. Thus the creation of these lists is performed in time proportional to the number of ports.

The next step is the determination of how far to shift the bottom ports so that the total length of the horizontal portions of the wires is minimized (section 3.3.11.3, step 9). The simple approach taken involves shifting the collection of bottom ports as a unit and using the position with the smallest total horizontal wire length. The number of comparisons that have to be made is between none (if there is a one-to-one correspondence between ports on the top and bottom of the channel) and the difference between the number of ports on the top and bottom of the channel times the number of ports on the top of the channel, which goes to $\mathcal{O}(n^2)$ in the limit.

The final step is the permutation route between the one-to-one mapping of ports created in the previous step to the ports at the bottom of the channel. The permutation routing algorithm shown in section 3.3.11.2 divides the routes into two classes–routes-sinister and routes-dexter. Routes-sinister are routed in a manner similar to river-routing (section 3.3.11.1), except that the vertical portions of the wires are routed on the first routing layer and the horizontal portions on the second layer. Routes-dexter are then routed on the first routing layer, passing under the horizontal portions of the routes-sinister. In the process of routing the wires, each successive wire has to be routed so that it avoids the corners of the preceding wires. River-routing requires time proportional to the square of the number of routes.

Thus the channel routing can be performed in the worst case in time proportional to the square of the number of ports on the top of the channel.

Permutation routing requires a channel no wider than that required for n wires (with contacts) and no higher than that required for n wires (with contacts).

One-to-many mappings contain a permutation route and a many-to-one route. At most, half of the ports can be multi-ported. Therefore a many-to-one route requires at most $\frac{n}{2}$ horizontal wires, since one horizontal wire is required for connect each multi-ported wire. The actual number might be less than this because multi-ported wires may share the same horizontal track when they do not interfere with each other. The permutation route is placed south of the many-to-one route, so that it does not interfere with it, and therefore the maximum total area is the sum of the areas of these two routes. For a maximum height ($\frac{n}{2}$) many-to-one route a permutation route of $\frac{n}{2}$ wires, which requires height $\frac{n}{2}$, is needed. Thus the total height is $\mathcal{O}(n)$.

General channel routing is performed with a many-to-one mapping at the top and at the bottom of the channel with a permutation route in between. This produces a channel no wider than that required for n wires (with contacts) and no higher than $\mathcal{O}(\frac{3n}{2})$, since the maximum height of each many-to-one channel route is $\frac{n}{2}$ and the maximum height of the channel route is also $\frac{n}{2}$.

4.7 Summary

The time complexities of the synthesis algorithms are summarized in Table 4.1. The symbols are defined as follows.

c The number of characters in the source program.

t The number of tokens in the source program ($< c$).

Table 4.1: Complexity of Synthesis Algorithms Used

Lexing	$\mathcal{O}(c + \log_2 h)$
Parsing	$\mathcal{O}(t)$
Structure Tree Construction	$\mathcal{O}(t \times \log_2 t + \log_2 h)$
Structure Tree Traversal	$\mathcal{O}(t)$
Layout	$\mathcal{O}(t)$
River Routing	$\mathcal{O}(w^2)$
Channel Routing	$\mathcal{O}(w^2)$

H The number of hash buckets in the symbol table.

h The number of overflow buckets in the symbol table $(t - H)$.

w The number of wires to route.

It is difficult to relate these numbers to one another. For example, the number of wires in a channel, w, is not directly related to the size of the structure tree or the number of tokens or characters in the source program. While it is difficult to characterize this synthesis process, the squared terms probably do not dominate the complexity. This is because these terms depend on the number of wires in individual channels, not the number in the entire circuit.

CHAPTER 5

Examples

This chapter contains examples of several designs using ALICS. These examples illustrate the ability to express algorithms in ALICS. These algorithms have been laid out using the methods in Chapter 3. The area and speed are shown for each of these so that variations on the designs can be compared.

First some basic switching expressions are presented. Then, some designs for some equality function are presented. This illustrates the use of compile-time conditionals and compile-time recursion in the specification of a function of varying length operands. Two methods of performing addition are given and the area, power, and speed compared.

5.1 Switching Expressions

The basic component of an ALICS program is a switching expression. This involves Boolean operators on a function's inputs. A sample function is shown in Figure 5.1 This example is a complete one, in that it contains everything needed to compile it. The entire program is enclosed in a **BEGIN-END** block. The next three lines define the operators **NOR**, **NAND**, and **XOR**. They are functions, each requiring two **BIT**s and yielding a **BIT** value. The function body is defined as an

```
BEGIN
      OP NOR = (BIT a, b) BIT : CODE "nor2";
      OP NAND = (BIT a, b) BIT : CODE "nand2";
      OP XOR = (BIT a, b) BIT : CODE "xor2";

      LEFTASSOCIATIVE NOR, NAND, XOR;
      PRIO NOR = 2;
      PRIO XOR = 2;
      PRIO NAND = 3;
      FUNC switch = (BIT c, d, e) BIT :
            (c XOR e) NOR d NAND e;

      switch
END
```

Figure 5.1: A Function Definition Containing Switching Expressions

external cell by using the token **CODE**. That is, **NOR** is defined as an external cell named nor2. Presumably this name is descriptive, implying that it computes the *nor* function of two inputs (although this is not necessarily so). The three operators are defined as being left associative. The priorities of the operators are defined in the next three lines. The higher the number, the higher the priority, or precedence, of operator. Next the function named switch is defined as requiring three **BIT** inputs and yields a **BIT** value. The function body is on the next line. Because **NAND** has a higher priority than **NOR**, the function is equivalent to

(c **XOR** e) **NOR** (d **NAND** e).

Finally, the name of the function switch without any arguments implies that this function is the yield of the block (not the value of the function, but the function itself). This means that the inputs and output of the block are synonymous with the inputs and outputs of the overall function, which will become the inputs and outputs of the chip. The layout of this function after the specification has been run through the synthesizer is shown in Figure 5.2

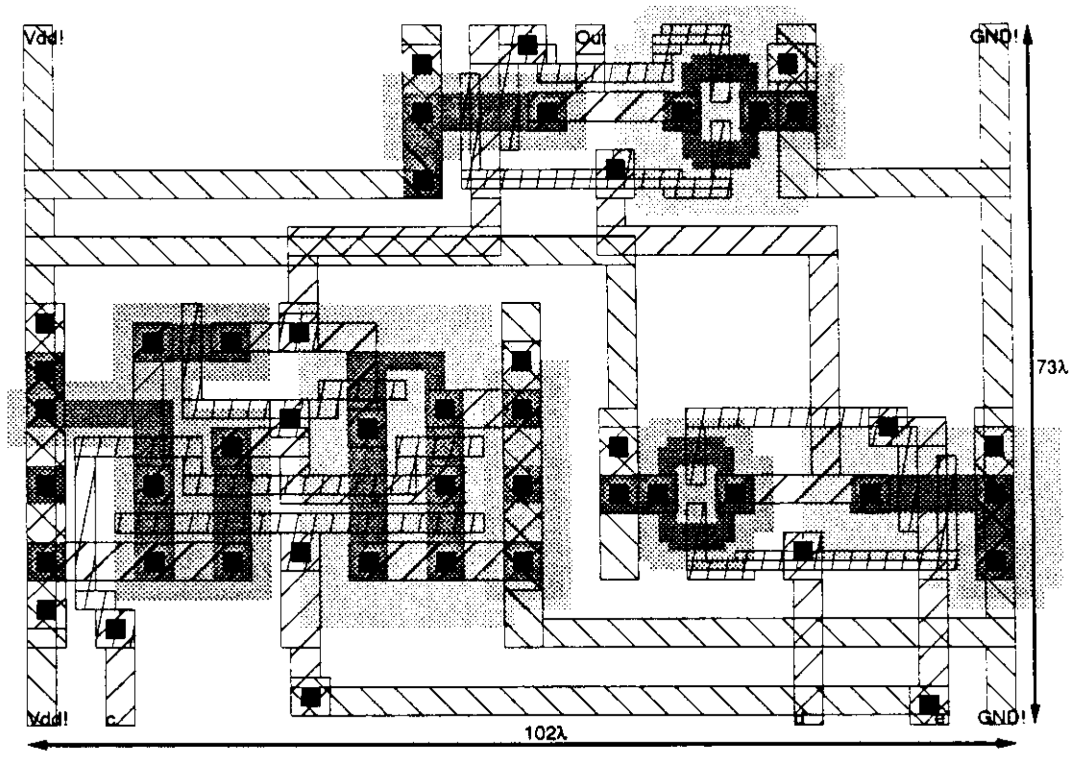


Figure 5.2: Layout for the Function Definition in Figure 5.1

5.2 Conditional Expressions

Figure 5.3 shows the specification of a conditional expression. The corresponding layout is shown in Figure 5.4. Note that there is permutation routing at the bottom of the channel that puts the input parameters in the proper order for their use within the circuitry.

```
BEGIN
        OP NOR = (BIT a, b) BIT : CODE "nor2";
        OP NAND = (BIT a, b) BIT : CODE "nand2";
        OP XOR = (BIT a, b) BIT : CODE "xor2";

        LEFTASSOCIATIVE NOR, NAND, XOR;
        PRIO NOR = 2;
        PRIO XOR = 2;
        PRIO NAND = 3;
        FUNC ifchip = (BIT j, a, f, d, b, i, h, g, c, e) BIT :
                IF a NOR b
                THEN c NAND d
                ELIF g NOR h
                THEN i NAND j
                ELSE e XOR f
                FI;

        ifchip
END
```

Figure 5.3: Specification of a Conditional Expression

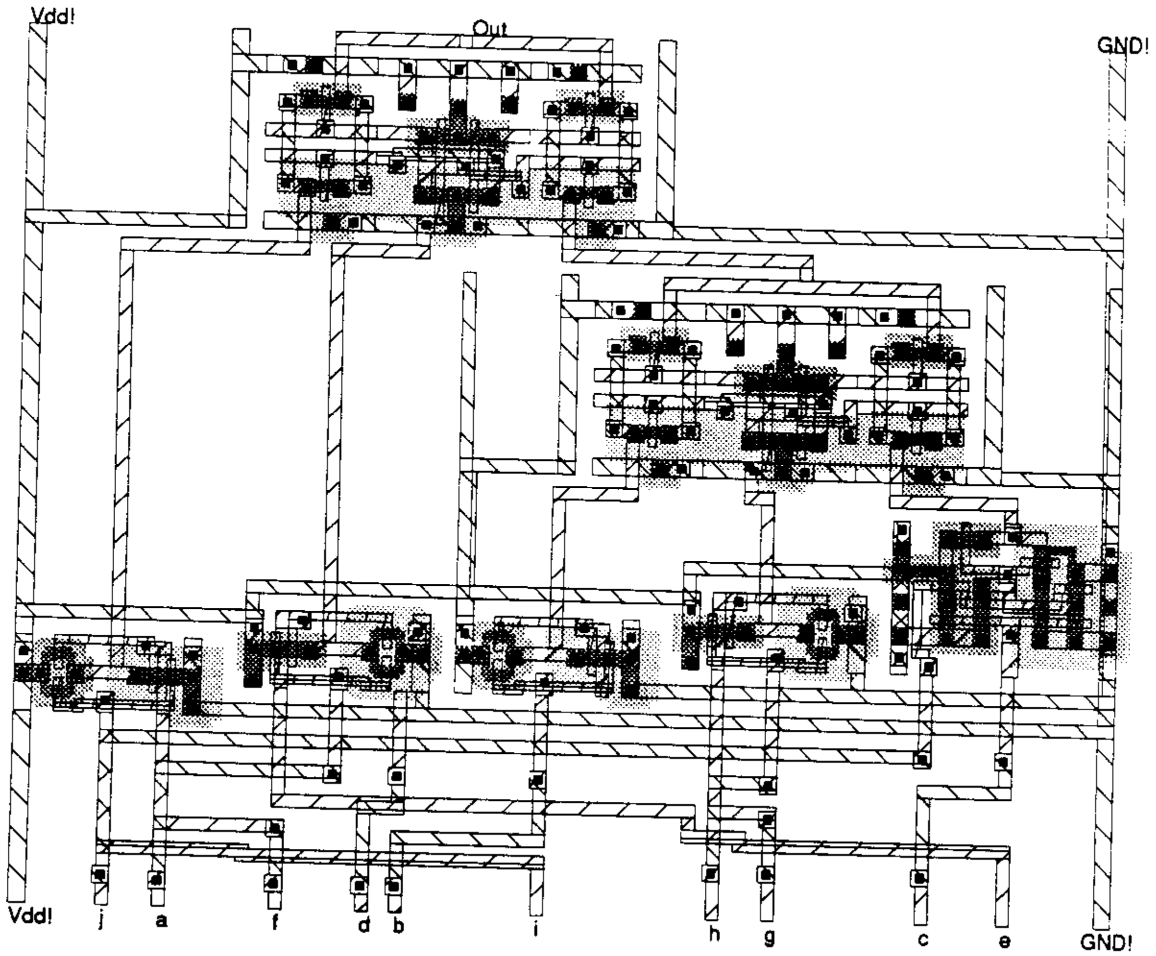


Figure 5.4: Layout for the Function in Figure 5.3

5.3 Equals

While ALICS contains an equality operator (=) and the ALICS Synthesizer will produce circuitry for this function, it is sometimes useful for the designer to redefine (or *overload*) this operator for certain data types. This is necessary when equality is not defined as bit-by-bit equality, as in interval arithmetic, or comparisons of stacks. This, however, is not the purpose of this example, but instead this shows the definition of an operator that works on operands of varying length. This is done by the function defined in Figure 5.5. For single-bit operands, a primitive comparison function (inverted exclusive OR) is called. For an array of operands the array is divided in half, and the equality routine is called twice in parallel with these half-sized arrays. The values returned by these two invocations are ANDed together. These invocations might involve further recursions and AND operators.

The question "Which operations are performed at compile time and which produce circuitry?" might come up. The answer is that the operations that *can* be performed at compile time *are*, and the rest are translated into circuitry. As it stands, the = operator in Figure 5.5 cannot be implemented because the sizes of the operands are not known. (If a size were chosen arbitrarily, it might be exceeded in some application.) The function in Figure 5.6 can be implemented because the sizes of the operands are known. They are both arrays of **BIT**s of size eight. To produce a circuit with its inputs corresponding to the function's inputs, this function is invoked without an argument list. This implies that the

```
/* Define AND as the inversion of NAND. */
OP AND = (BIT a,b) BOOL:
BEGIN
NOT (a NAND b)
END

/* Define = for BITs. */
OP = = (BIT a,b) BOOL:
BEGIN
NOT (a XOR b)
END

OP = = ([] BIT c,d) BOOL:
BEGIN
        IF LWB c = LWB d
        ANDIF UPB c = UPB d
        THEN /* The precisions of the two arrays are the same. */
                IF LWB c = UPB d
                THEN /* Use the BIT comparator function. */
                        c[LWB c] = d[LWB d]
                ELSE /* Else divide the work recursively
                       and AND the results. */
                       (      c[UPB c : ceil((UPB c + LWB c)/2)]
                       = d[UPB d : ceil((UPB d + LWB d)/2)])
                       AND (  c[ceil((UPB c + LWB c)/2) -1 : LWB c]
                       = d[ceil((UPB d + LWB d)/2) -1 : LWB d])
                FI
        ELSE /* The precisions of the two arrays are different. */
        FI
END
```

Figure 5.5: Definition of a Generic-Length Equality Operator

```
FUNC equals8 ([7:0]BIT a,b) BIT:
BEGIN
        a = b
END
```

Figure 5.6: An Implementable Function

inputs and the outputs of the function are bound to the inputs and outputs of the circuit. An outline of the complete definition of the chip is given in Figure 5.7 (these definitions have already be given).

```
BEGIN
        Definition for AND.
        Definition for =.
        Definition for equals8.
        equals8
END
```

Figure 5.7: A Specification for an Circuit for Computing Equals on Two Arrays of **BIT**s

The translation of this specification is performed as follows. During the parsing of this program the three definitions are saved. The fourth line of the program is the last functional unit of the **BEGIN-END** block, and so it is the value yielded by the block. The function body for equals becomes the body of the block. This is the expression

$$a = b.$$

Since this block is the outermost one in the specification, the inputs (a and b) and outputs of equals are the inputs and outputs of the circuit/chip. Continuing the expansion of this function, = needs to be expanded. equals' values a and b are bound to the operator ='s formal parameters c and d, respectively. The operator = is not implementable in isolation, but in this context the sizes of its operands are known. This means that the following substitutions can be made.

$$\textbf{LWB}\ c = 0$$

```
LWB d = 0
UPB c = 7
UPB d = 7
```

These substitutions cause the lines

```
IF LWB c = LWB d
ANDIF UPB c = UPB d
```

to become

```
IF 0 = 0 ANDIF 7 = 7
```

Since 0 and 7 are integers, the = operator for integers is invoked. This does not correspond to the = operator defined in Figure 5.5, which operates only on arrays of **BIT**s, so the built-in = operator is invoked. The operands (0 and 7) are compile-time constants, so this function is evaluated at compile time and replaced with the following.

```
IF TRUE ANDIF TRUE.
```

Similarly, the operands of **ANDIF** are known at compile time, and evaluate to **TRUE**. Thus the **THEN** part is evaluated and the **ELSE** part is ignored.

The inner **IF** is then evaluated. Since **LWB** c=0 is not equal to **UPB** d, the inner **ELSE** expression is evaluated. The **ELSE** expression simplifies to the following.

```
a[7:4] = b[7:4] AND a[3:0] = b[3:0]
```

The operands for the = operators in this expression are arrays of **BIT**s, and so the same **equals** operator is called recursively causing the **ELSE** expression to become

```
(a[7:6] = b[7:6] AND a[5:4] = b[5:4]) AND
(a[3:2] = b[3:2] AND a[1:0] = b[1:0])
```

and on the next iteration,

```
((a[7:7] = b[7:7] AND a[6:6] = b[6:6]) AND
 (a[5:5] = b[5:5] AND a[4:4] = b[4:4])) AND
((a[3:3] = b[3:3] AND a[2:2] = b[2:2]) AND
 (a[1:1] = b[1:1] AND a[0:0] = b[0:0])).
```

This expression involves the = operator on one-element arrays of **BIT**s so the = operator is invoked eight more times at compile time. At this point the inner **IF** expression in the = operator evaluates to **TRUE**. This causes the above expression to become

```
((a[7] = b[7] AND a[6] = b[6]) AND
 (a[5] = b[5] AND a[4] = b[4])) AND
((a[3] = b[3] AND a[2] = b[2]) AND
 (a[1] = b[1] AND a[0] = b[0])).
```

This expression involves the = operator on **BIT**s. This expression is expanded one more time to yield

```
((NOT(a[7] XOR b[7]) AND NOT(a[6] XOR b[6])) AND
 (NOT(a[5] XOR b[5]) AND NOT(a[4] XOR b[4]))) AND
((NOT(a[3] XOR b[3]) AND NOT(a[2] XOR b[2])) AND
 (NOT(a[1] XOR b[1]) AND NOT(a[0] XOR b[0]))).
```

No further substitutions can be made on this expression because none of the values of the **BIT**s of **a** and **b** are known at compile time. The layout for an eight-bit equals operator from this algorithm is shown in Figure 5.8. The areas and delays for several sizes of **BIT**-array equals operators are shown in Table 5.1.

5.4 Another Equals

The equals operator in the previous section incorporates only 2-input nand gates in the design. In this section an algorithm that also incorporates 3-input,

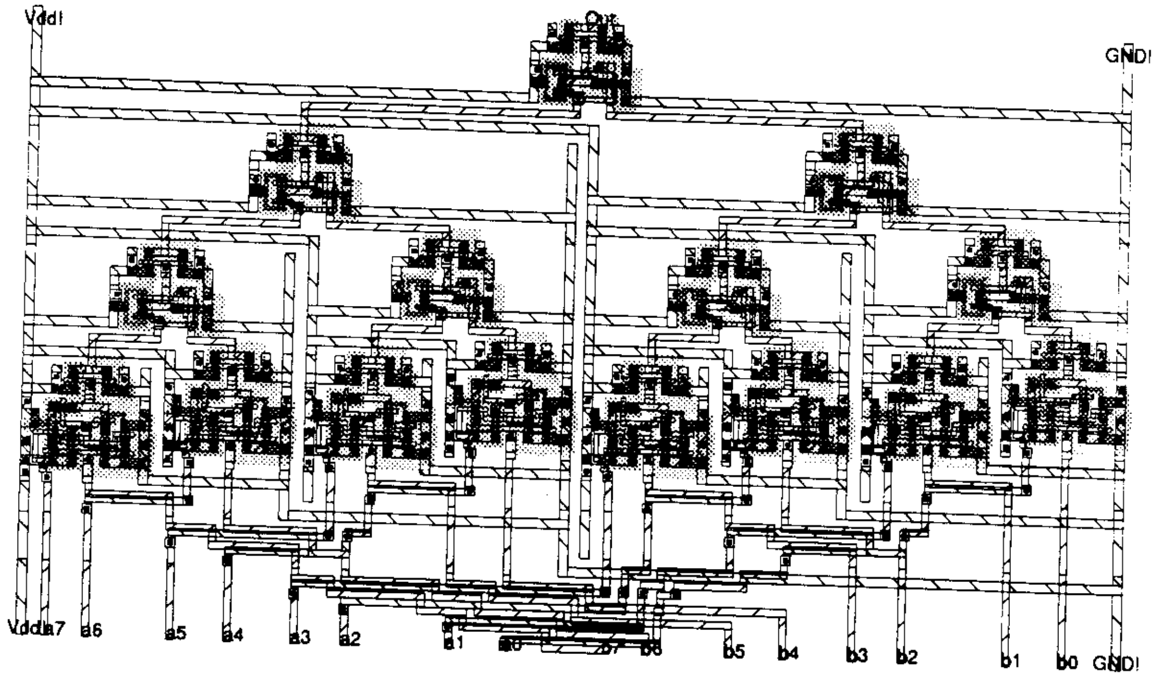


Figure 5.8: Eight-Bit Equals Using 2-Input Nands

Table 5.1: Areas and Delays for BIT-Array Equals Operators

# of Bits	x (λ)	y (λ)	Area (λ^2)	Delay (nSec)
1	53	50	2,650	6.59
2	112	114	12,768	10.05
3	230	185	42,550	13.55
4	230	185	42,550	13.55
5	465	268	124,620	17.13
6	465	268	124,620	17.13
7	465	268	124,620	17.13
8	465	268	124,620	17.13

and 4-input nands is presented. This not only produces a more compact layout, but also illustrates a compile-time looping operator. The algorithm is shown in Figures 5.9 and 5.10. This algorithm creates a tree of AND gates in order

```
FUNC and4 (BIT a,b,c,d) BIT:
NOT nand4(a,b,c,d);
FUNC and3 (BIT a,b,c) BIT:
NOT nand3(a,b,c);
FUNC and2 (BIT a,b) BIT:
NOT nand2(a,b);
/* Define = for BITs. */
OP = = (BIT a,b) BOOL:
NOT (a XOR b);
```

Figure 5.9: Definition of a Generic-Length Equality Operator Using 2, 3, and 4-Input Nands

to compute the AND of all the comparison operators at the leaf nodes. This algorithm incorporates as many large (four-input) gates at each level of the tree, but will incorporates smaller gates (three or two-input) when this will equalize the sizes of the gates at each level. By using larger gates close to the leaf nodes, the design will have fewer levels than if smaller gates were used.

This algorithm is best understood by considering a few examples. The length calculated in the **FOR** loop is actually one less than the number of elements in the arrays. If the number of elements is 2, 3, or 4, then length is 1, 2, or 3 and the expression corresponding to [1], [2], or [3] in the **CASE** expression is evaluated. These expressions are two-input, three-input, and four-input and gates.

If there are more than four elements in the arrays, then the number of elements minus 1 is repeatedly divided by four until the integer portion is one, two, or three.

```
/* General-purpose equals using nand2,3,4. */
OP = = ([] BIT a,b) BOOL:
BEGINIF LWB a = LWB b
       ANDIF UPB a = UPB b
       THEN/* The precisions of the two arrays are the same. */
             IF UPB a - LWB a = 0
             THEN a[LWB a] = b[LWB b] /* Use the BIT = function. */
             ELSEFOR length := UPB a - LWB a
                        WHILE length != 0 EXEC length /:= 4
                  DO CASE length IN
                  /* Divide the work recursively and AND the results. */
                  [1]: ( a[UPB a : ceil((UPB a + LWB a)/2)]
                      = b[UPB b : ceil((UPB b + LWB b)/2)])
                      AND ( a[ceil((UPB a + LWB a)/2) -1 : LWB a]
                      = b[ceil((UPB b + LWB b)/2) -1 : LWB b])
                  [2]: and3( a[LWB a:ceil((LWB a + UPB a)/3)-1]
                      = b[LWB b:ceil((LWB b + UPB b)/3)-1],
                      a[ceil((LWB a + UPB a)/3)
                       :ceil(2*(LWB a + UPB a)/3)-1]
                      = b[ceil((LWB b + UPB b)/3)
                       :ceil(2*(LWB b + UPB b)/3)-1],
                      a[ceil(2*(LWB a + UPB a)/3):UPB a]
                      = b[ceil(2*(LWB b + UPB b)/3):UPB b])
                  [3]: and4( a[LWB a:ceil((LWB a + UPB a)/4)-1]
                      = b[LWB b:ceil((LWB b + UPB b)/4)-1],
                      a[ceil((LWB a + UPB a)/4)
                       :ceil(2*(LWB a + UPB a)/4)-1]
                      = b[ceil((LWB b + UPB b)/4)
                       :ceil(2*(LWB b + UPB b)/4)-1],
                      a[ceil(2*(LWB a + UPB a)/4)
                       :ceil(3*(LWB a + UPB a)/4)-1]
                      = b[ceil(2*(LWB b + UPB b)/4)
                       :ceil(3*(LWB b + UPB b)/4)-1],
                      a[ceil(3*(LWB a + UPB a)/4):UPB a]
                      = b[ceil(3*(LWB b + UPB b)/4):UPB b])
                      ESAC
                  OD
             FI
       ELSE /* The precisions of the two arrays are different. */
       FI
 END
```

Figure 5.10: Definition of a Generic-Length Equality Operator Using 2, 3, and 4-Input Nands (cont.)

Then the corresponding expression in the **CASE** expression is evaluated. Each of these expressions divides the input arrays into 2, 3, or 4 arrays of more or less equal number of elements. These arrays are recursively partitioned and passed to 2, 3, or 4 invocations of the = operator until arrays of 2, 3, or 4 **BIT**s are obtained. Then 2, 3, or 4-input AND gates are used to AND together the yields of these = operators. The layout for an eight-bit equals operator from this algorithm is shown in Figure 5.11.

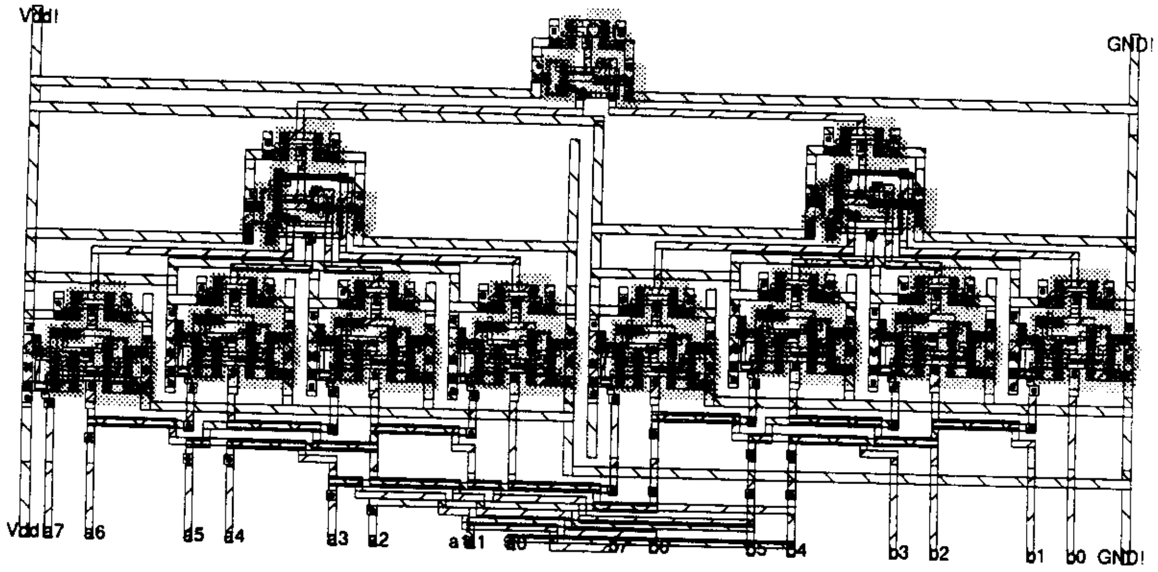


Figure 5.11: Eight-Bit Equals Using 2, 3, and 4-Input Nands

The areas and delays for several sizes of **BIT**-array equals operators using this method are shown in Table 5.2.

Table 5.2: Areas and Delays for BIT-Array Equals Operators

# of Bits	x (λ)	y (λ)	Area (λ^2)	Delay (nSec)
1	53	50	2,650	6.59
2	112	114	12,768	10.05
3	171	129	22,059	12.05
4	230	142	32,660	14.89
5	289	205	59,245	15.55
6	348	206	71,688	15.59
7	407	223	90,761	18.42
8	466	225	104,850	18.46

5.5 Two's Complement Ripple-Carry Adder

A ripple-carry adder is formed from a linear array of full-adders. Block diagrams for a full-adder and a ripple-carry adder made from full-adders are shown in Figure 5.12. As is seen from the figure, the full-adder function is not used as a pure

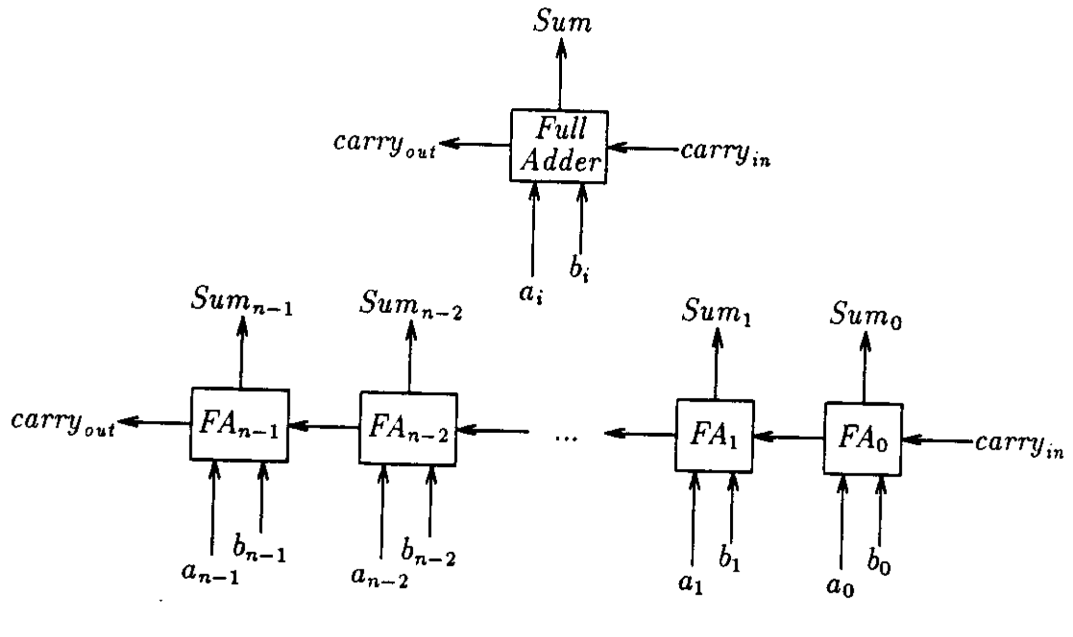


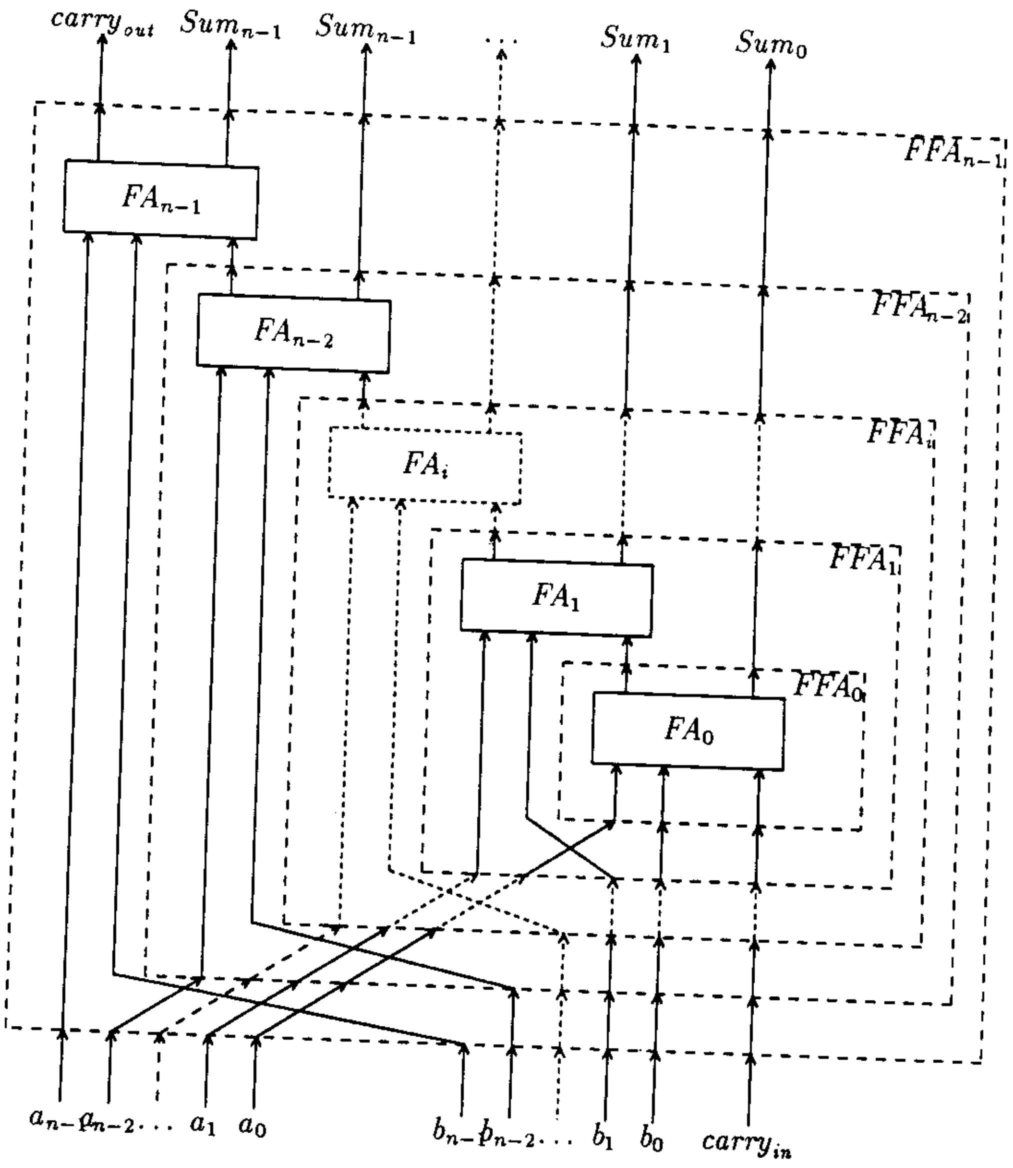
Figure 5.12: Two's Complement Ripple-Carry Adder

function. This is because the Sum and $\text{carry}_{\text{out}}$ outputs are not inputs to a single function. One of the outputs ($\text{carry}_{\text{out}}$) is an input to another full-adder, and the other output (Sum) forms part of the output sum vector. However, if the definition for the full-adder is modified slightly so that it takes all of the lower-significant bits as input, a purely functional representation can be made. This functional program is shown in Figure 5.13. A layout for this form is shown in Figure 5.14. In this

```
MODE CARRY_SUM STRUCT (BIT carry, [ ]BIT sum);
MODE INT_SIXTEEN [15:0]BIT;

FUNC full_add (BIT a, b; CARRY_SUM c) CARRY_SUM:
      (nand3(a NAND b, a NAND carry OF c, b NAND carry OF c),
       xor3(a, b, carry OF c) PREPEND sum OF c);

FUNC ripple_carry([ ]BIT a, b; BIT c) CARRY_SUM:
      /* check for terminating condition of nil BIT arrays */
      IF (UPB a ≠ LWB a) ANDIF (UPB b ≠ LWB b)
      THEN full_add(a[UPB a], b[UPB b],
       ripple_carry(a[UPB a:LWB a −1], b[UPB b:LWB b −1], c))
      ELSE (c, SKIP)
      FI

OP + = (INT_SIXTEEN a, b) INT_SIXTEEN:
        ripple_carry(a, b, 0)
```

Figure 5.13: Functional Description of a Ripple-Carry Adder

figure, the dashed boxes represent instantiations of the new full adder function. Instead of having three one-bit inputs (corresponding to the carry-in, the two inputs) and two one-bit outputs (the sum, and carry-out outputs), the functional full-adder has two one-bit inputs representing the summands and a **CARRY_SUM** input. The **CARRY_SUM** input contains the carry-in and all of the less-significant sum bits. This representation of the adder is useful because its structure can be synthesized automatically from a recursive functional description of the behavior of the adder.



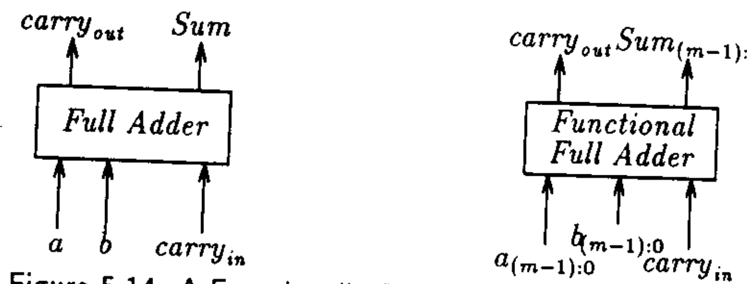

$carry_{out}$
Sum
Full Adder
a b $carry_{in}$
$carry_{out}$ $Sum_{(m-1):0}$
Functional Full Adder
$a_{(m-1):0}$ $b_{(m-1):0}$ $carry_{in}$

Figure 5.14: A Functionally-Defined Ripple-Carry Adder

Figure 5.15 shows the mask layout for an eight-bit two's complement ripple-carry adder. The inputs to the adder are on the bottom, and the sum outputs are on the top. Adders from size one to eight were designed. The sizes and delays of these adders are shown in Table 5.3. The delays are computed by *crystal.*

Table 5.3: Areas and Delays for Ripple-Carry Adders

# of Bits	x (λ)	y (λ)	Area (λ^2)	Delay (nSec)
1	221	109	24,089	9.29
2	260	234	60,840	13.54
3	299	366	109,434	22.28
4	338	500	169,000	31.01
5	377	634	239,018	39.75
6	416	770	320,320	48.48
7	455	904	411,320	57.22
8	494	1041	514,254	65.96

5.6 Redundant-Digit Adder

Another form of addition makes use of a redundant digit set. This example illustrates the use of functions which replicate values so that the designer can avoid replicating functional units.

If, instead of using digits from the set {0,1}, the digit set {−1,0,1} is used, adders can be designed that have limited carry propagation [Aviz61,Boro68,Chow78]. In particular, Chow [Chow78] gives the designs for a class of redundant binary adders. These will be the basis of the example in this section.

Figure 5.16 shows a block diagram from [Chow78] for the redundant digit adder. The subscripts have been changed from the original so that the ith column has the

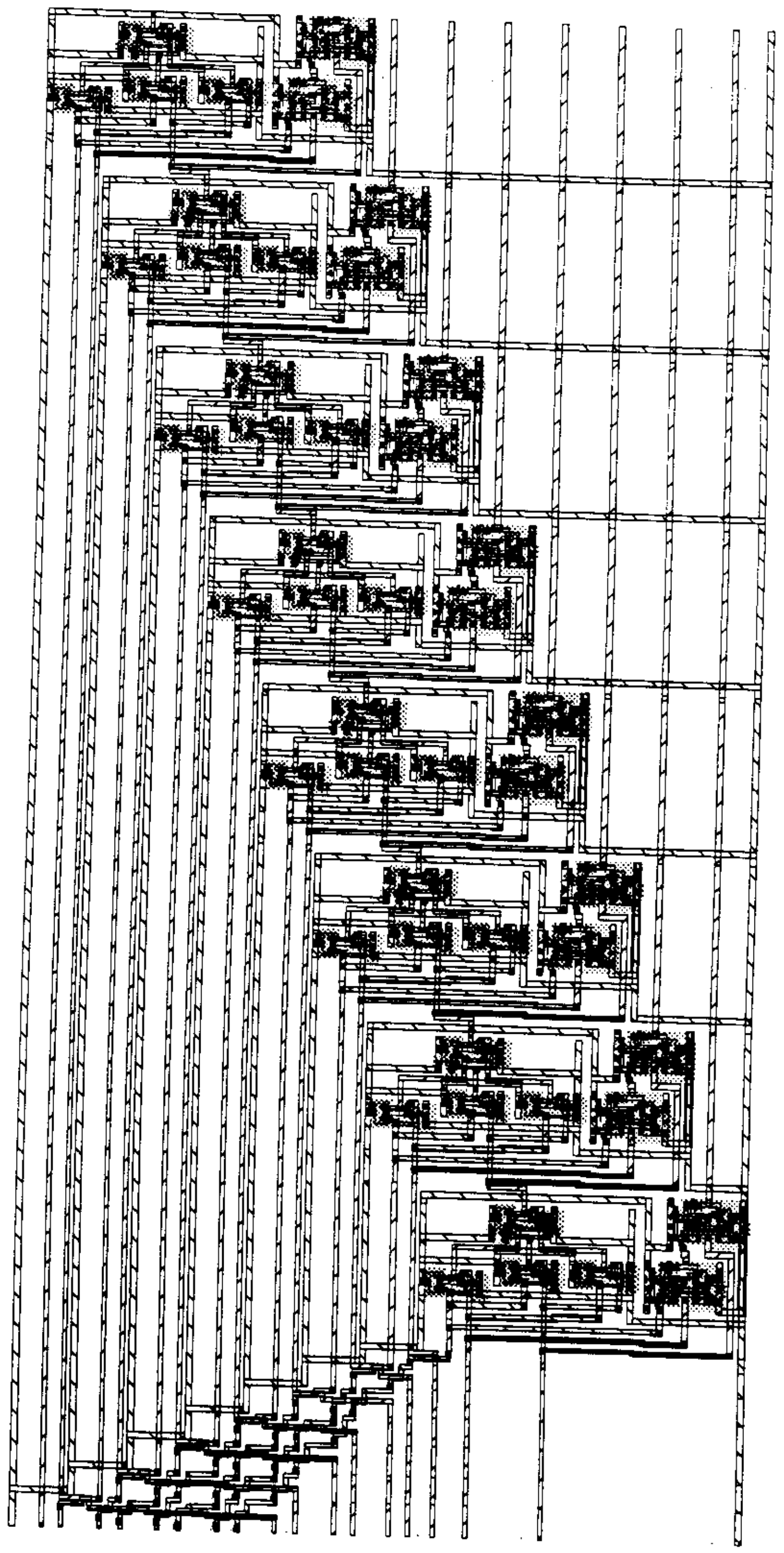


Figure 5.15: Layout For an 8-Bit Two's Complement Ripple-Carry Adder

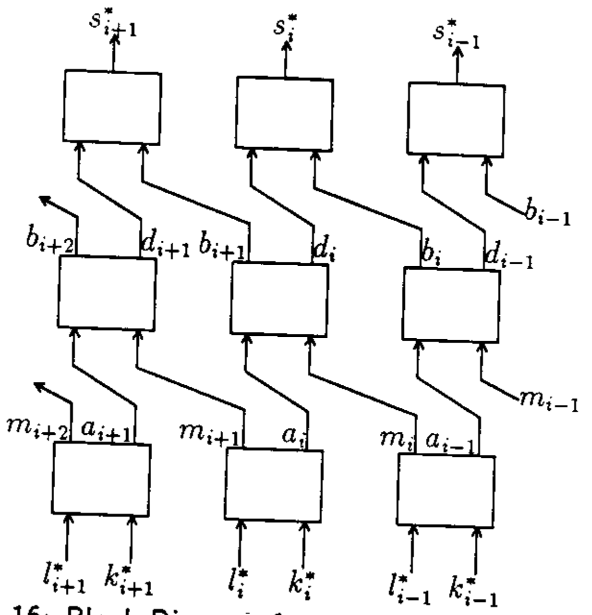


Figure 5.16: Block Diagram for a Redundant-Digit Adder

weight 2^i. The inputs at the bottom (l^* and k^*) and the sum sum^* have values from the set $\{-1,0,1\}$. The inputs at the bottom l^*_i and k^*_i are represented here as the two-tuples $(l,\ L)$, $(k,\ K)$ and may take on values from the set $\{-1, 0, 1\}$. The sum sum^*_i is represented at the two-tuple $(s,\ S)$. There is a carry *carry* and a borrow *borrow* from the less-significant stages. The carry has values from the set $\{0,1\}$ and the borrow values from the set $\{-1,0\}$. These values can be represented by sets of Boolean values in many ways. One of these ways is shown by the encodings in Table 5.4. The left two columns show encodings and the right column show the

sum^*_i		*Value*
σ_i	s_i	
0	0	0
0	1	1
1	0	*d.c.*
1	1	−1

$carry_i$	*Value*
0	0
1	+1

$borrow_i$	*Value*
0	0
1	−1

Table 5.4: Encodings for Signed Digits

value being represented. The *d.c.* represents a "don't care" encoding, an encoding that is not used. "Don't care" values are not generated or input, so their use can simplify logic.

The adder/subtracter implements the equations:

$$d_i + borrow_i = sum_i^*$$

$$l_i^* + k_i^* + carry_i = 2 \times carry_{i+1} + 2 \times borrow_{i+1} + d_i$$

Combining these equations with the encodings from Table 5.4, yields the truth table in Table 5.5. This table contains entries labeled C, which represent *coupled-don't cares*. These are entries that can be zero or one, but all the C's must have the same value within a row. To make the $carry_{i+1}$ values independent of $carry_i$, the entries in the top half of the table must be the same as those in the bottom

Table 5.5: Truth Table for Redundant Digit Adder

$carry_i$		l_i	L_i	l_i^*	k_i	K_i	k_i^*	sum	$carry_{i+1}$		$borrow_{i+1}$		d_i	
bin	*val*	*bin*		*val*	*bin*		*val*	*val*	*bin*	*val*	*bin*	*val*	*bin*	*val*
0	0	0	0	0	0	0	0	0	C	C	C	$-C$	0	0
0	0	0	0	0	0	1	1	1	C	C	C	$-C$	1	1
0	0	0	0	0	1	1	−1	−1	0	0	1	−1	1	1
0	0	0	1	1	0	0	0	1	C	C	C	$-C$	1	1
0	0	0	1	1	0	1	1	2	1	1	0	0	0	0
0	0	0	1	1	1	1	−1	0	C	C	C	$-C$	0	0
0	0	1	1	−1	0	0	0	−1	0	0	1	−1	1	1
0	0	1	1	−1	0	1	1	0	C	C	C	$-C$	0	0
0	0	1	1	−1	1	1	−1	−2	0	0	1	−1	0	0
1	1	0	0	0	0	0	0	1	C	C	C	$-C$	1	1
1	1	0	0	0	0	1	1	2	1	1	0	0	0	0
1	1	0	0	0	1	1	−1	0	C	C	C	C	0	0
1	1	0	1	1	0	0	0	2	1	1	0	0	0	0
1	1	0	1	1	0	1	1	3	1	1	0	0	1	1
1	1	0	1	1	1	1	−1	1	C	C	C	$-C$	1	1
1	1	1	1	−1	0	0	0	0	C	C	C	C	0	0
1	1	1	1	−1	0	1	1	1	C	C	C	$-C$	1	1
1	1	1	1	−1	1	1	−1	−1	0	0	1	−1	1	1

half of the table. This still leaves some coupled don't care entries, whose values are chosen to simplify the implementing logic. Table 5.6 shows the truth table after these decisions have been made. Logic equations derived from this table using

Table 5.6: Complete Truth Table for Redundant Digit Adder

$carry_i$		l_i	L_i	l_i^*	k_i	K_i	k_i^*	sum	$carry_{i+1}$		$borrow_{i+1}$		d_i	
bin	*val*	*bin*		*val*	*bin*		*val*	*val*	*bin*	*val*	*bin*	*val*	*bin*	*val*
0	0	0	0	0	0	0	0	0	1	1	1	−1	0	0
0	0	0	0	0	0	1	1	1	1	1	1	−1	1	1
0	0	0	0	0	1	1	−1	−1	0	0	1	−1	1	1
0	0	0	1	1	0	0	0	1	1	1	1	−1	1	1
0	0	0	1	1	0	1	1	2	1	1	0	0	0	0
0	0	0	1	1	1	1	−1	0	0	0	0	0	0	0
0	0	1	1	−1	0	0	0	−1	0	0	1	−1	1	1
0	0	1	1	−1	0	1	1	0	0	0	0	0	0	0
0	0	1	1	−1	1	1	−1	−2	0	0	1	−1	0	0
1	1	0	0	0	0	0	0	1	1	1	1	−1	1	1
1	1	0	0	0	0	1	1	2	1	1	0	0	0	0
1	1	0	0	0	1	1	−1	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	2	1	1	0	0	0	0
1	1	0	1	1	0	1	1	3	1	1	0	0	1	1
1	1	0	1	1	1	1	−1	1	0	0	0	0	1	1
1	1	1	1	−1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	−1	0	1	1	1	0	0	0	0	1	1
1	1	1	1	−1	1	1	−1	−1	0	0	1	−1	1	1

Karnaugh maps are shown in Figure 5.17.

This can not be expressed directly in the functional synthesis language, as these equations are not strict functions. The block diagram in Figure 5.16 can be reorganized as a completely functional circuit. A block diagram for a functional three-bit redundant-digit adder is shown in Figure 5.18. An n-bit redundant digit adder (the outermost dashed box) contains a one-bit redundant-digit adder and an $n-1$-bit redundant-digit adder.

$$
\begin{aligned}
carry_{i+1} &= \overline{l_i + k_i} \\
borrow_{i+1} &= \overline{\overline{l_i \cdot K_i} \cdot \overline{\overline{carry_i} \cdot \overline{k_i} \cdot K_i} \cdot \overline{\overline{carry_i} \cdot \overline{l_i} \cdot \overline{L_i}}} \\
d_i &= carry_i \oplus L_i \oplus K_i \\
\sigma_i &= \overline{\overline{borrow_i} + d_i} \\
sum_i &= borrow_i \oplus d_i
\end{aligned}
$$

Figure 5.17: Logic Equations for the Redundant Digit Adder

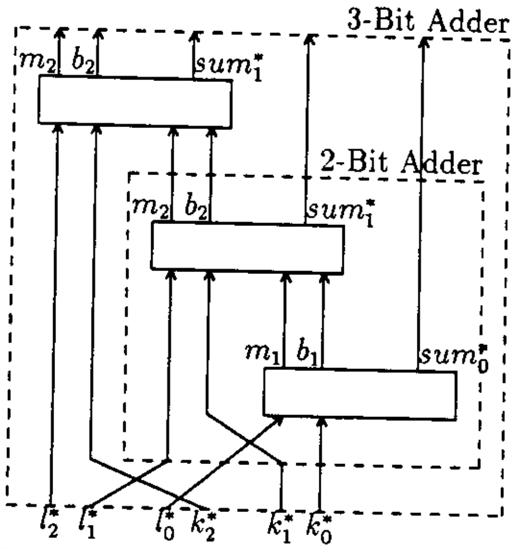


Figure 5.18: Block Diagram for a Functional Three-Bit Redundant-Digit Adder

A functional description of this adder is shown in Figure 5.19. The program

```
MODE SIGNED_DIGIT_TWO = STRUCT (BIT sign, digit);
MODE PARTIAL_SUM =
        STRUCT (BIT carry, borrow; SIGNED_DIGIT [ ] sum);
FUNC red adder sum(BIT d, borrow) SIGNED_DIGIT:
      (d NOR NOT borrow, d XOR borrow);
FUNC red adder borrow(SIGNED_DIGIT_TWO l, k; BIT carry bar) BIT:
      nand3(l.sign NAND k.digit,
            nand3(NOT l.sign, NOT l.digit, carry bar),
            nand3(NOT k.sign, NOT k.digit, carry bar));
FUNC red adder a(SIGNED_DIGIT_TWO l, k; PARTIAL_SUM sum)
                PARTIAL_SUM:
      (l.sign NOR k.sign,
       red adder borrow(l, k, NOT sum.carry),
       red adder sum(xor3(l.digit, k.digit, sum.carry), sum.borrow)
                  PREPEND sum.sum);
FUNC red adder([ ]SIGNED_DIGIT_TWO l, k) PARTIAL_SUM:
      IF (UPB a ≠ LWB a) ANDIF (UPB borrow ≠ LWB borrow)
      THEN red adder a(l[UPB l], k[UPB k],
                       red adder(l[UPB l−1:LWB l],
                                 k[UPB k−1:LWB k]))
      ELSE red adder a(l[UPB l], k[UPB k], (0, 0, SKIP))
      FI;
FUNCTION red adder high order(PARTIAL_SUM partial)
      [ ]SIGNED_DIGIT_TWO:
          (red adder sum(partial.carry, partial.borrow) PREPEND partial.sum);
OP + = ([ ]SIGNED_DIGIT_TWO a, b) [ ]SIGNED_DIGIT_TWO:
      red adder high order(red adder(a, b))
```

Figure 5.19: Functional Description of a Redundant-Digit Adder

first declares two new modes (data-types) that will represent the redundant digits. The mode **SIGNED_DIGIT** represents signed binary digits, and contains the **BIT** fields **sign** and **digit**. The other mode, **PARTIAL_SUM**, represents the output of the redundant-digit adder. It contains a vector (with unfixed bounds) of signed digits (sum), a borrow, and a carry. The latter two fields can be combined to represent the high-order digit of the sum, but this algorithm defers this process.

An operator and five functions are defined. (Operators are functions whose names can appear in arithmetic expressions as infix dyadic or monadic operators.) The **+** operator is defined to add two vectors of signed digits. This is the top-level operator that an algorithm designer would design. The five functions form part of the semantics of this operator. The operator calls two functions, red adder high order and red adder. The latter function is a recursive function that adds two signed-digit vectors and returns a **PARTIAL_SUM**. This function will add any size vector of signed digits. The red adder high order function converts a **PARTIAL_SUM** to a vector of signed digits. The other three functions, red adder a, red adder borrow, and red adder sum are invoked whenever the output of a function needs to be replicated. For example, the function red adder sum has two inputs, each of which are input to two functions. It is only with the use of the named function inputs that a value can be replicated.

The layout for an eight-bit redundant-digit adder is shown in Figure 5.20. The areas and delays for adders of one through eight bits are shown in Table 5.7.

Table 5.7: Areas and Delays for Redundant Digit Adders

# of Bits	x (λ)	y (λ)	Area (λ^2)	Delay (nSec)
1	301	155	46,655	6.68
2	416	391	162,656	29.17
3	533	636	338,988	29.97
4	650	903	586,950	30.89
5	767	1169	896,623	31.81
6	884	1447	1,279,148	32.77
7	1001	1740	1,741,740	33.77
8	1118	2039	2,279,602	34.81

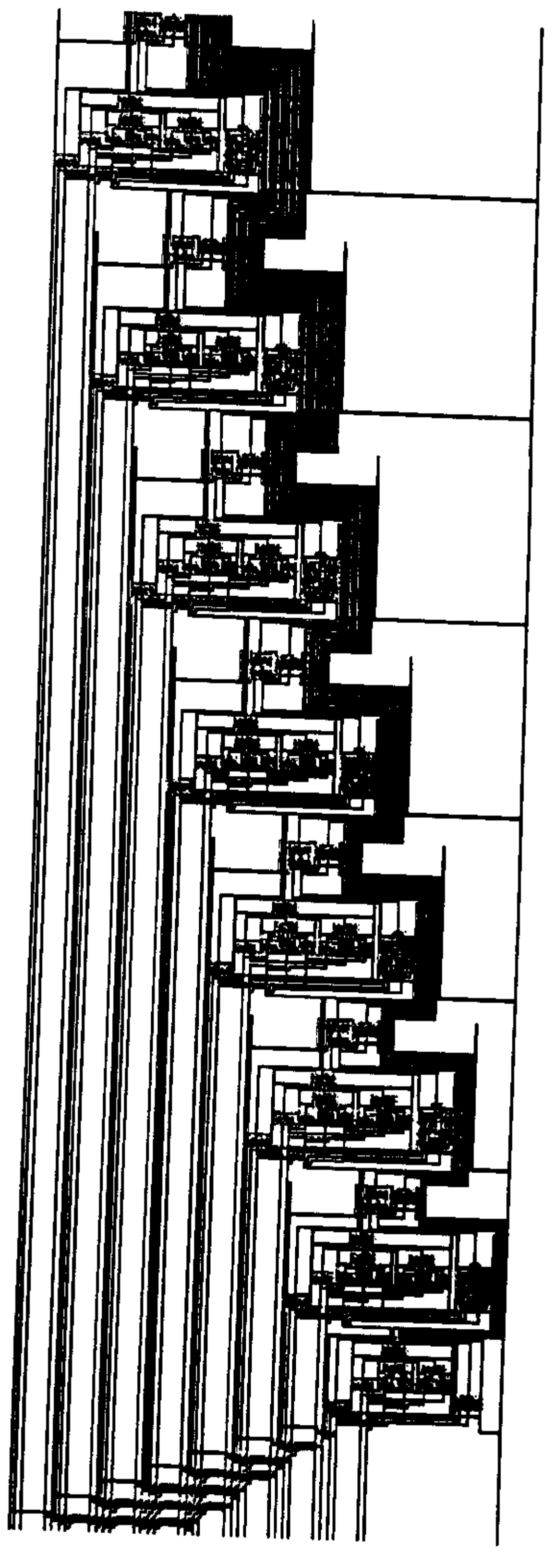


Figure 5.20: Layout for an Eight-Bit Redundant Digit Adder

CHAPTER 6

Conclusions

6.1 Contributions

The following are the contributions of the thesis research:

1. A method was developed for specifying the behavior of an integrated circuit at an algorithmic level and quickly translating the behavior into the topographical layout of custom integrated circuit masks. The translation process is shown to be rapid, scales well for large circuits, but is area-inefficient. The area-inefficiency is shown not to degrade the performance of the circuit so that the derived performance metrics of the resulting design can be used as a basis for comparing designs. Special-purpose circuits are often of non-standard design and require many iterations in the design process before cost-effective designs are obtained. The methods shown here are effective for rapid prototyping.

2. Consistent syntax and semantics for representing values by arrays and the manipulation of these values have been defined. The semantics is independent of the data type of the array elements. This provides a consistent definition for built-in datatypes and all user-defined datatypes.

3. Features from Algol 68 and FP have been combined to form a language that is functional in nature, but contains domains of sequentiality. These domains are treated as functional, as changes to localized storage in each domain do not affect the global state of a circuit. The data typing features from Algol 68 have been retained so that the types of all values can be checked at compile time. The Algol 68 operator definition facility has been extended to allow the specification of associativity and commutativity of operators.

4. A layout method has been designed for hierarchically composing non-uniformly sized cells (circuits). All these cells have a common structure—power on the left side, ground on the right, inputs on the bottom, and outputs at the top. The composition method does not have the common-height restriction that is present with "standard" cells. The layout method is shown to require time proportional to the number of cells. The hierarchical nature of this procedure enables ALICS Synthesizer to consider only information local to each cell, rather than a large portion of a circuit or details internal to portions of the circuit already laid out, and thereby is done quickly. A side-effect of this layout procedure is that the bounding box of the circuit is updated as each new element is added to it. Thus the area of the circuit is quickly determined.

   Because the circuit is quickly generated, existing tools can be used to obtain the power, speed, and other non-behavioral attributes of the circuit.

5. A time-efficient method has been designed for hierarchically routing power and ground wires for these circuits. The sizing of these wires is performed as cells are hierarchically constructed so that the phenomenon of *metal migration* is avoided. The overall routing of the power and ground wires is an interdigitated net on a single metal layer with no crossovers. This ensures that the voltage drops on these wires are as small as possible.

6. An extension has been made to existing river-routing methods that handles wires of differing sizes. Existing methods assume identically-sized wires. This extension to differing-sized wires is shown to increase the time complexity of channel-height determination from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$.

7. An area- and time-efficient method for two-layer routing of wire permutations across a channel has been demonstrated. This method has time complexity of $\mathcal{O}(n^2)$ and produces a channel route of area no greater than $\mathcal{O}(n^2)$.

8. An area- and time-efficient method for a two-layer routing of one-to-many mappings across a channel has been demonstrated. This method has time complexity of $\mathcal{O}(n^2)$ and produces a channel no wider than that required for n wires (with contacts) and no higher than that required for n wires (with contacts). One-to-many mappings are necessary in routing formal parameters (cell inputs) to parameter usage in a function's body.

9. A time-efficient method for a two-layer routing of many-to-many mappings across a channel (general channel routing) has been demonstrated. This

method has time complexity of $\mathcal{O}(n^2)$ and produces a channel no wider than that required for n wires (with contacts) and no higher than $\mathcal{O}(\frac{3n}{2})$. This method guarantees a channel route within the given channel width and can route *any* many-to-many mapping in this width.

10. Subroutines from the *Chisel* software have been incorporated into the ALICS Synthesizer. Extensions had to be made to the chisel IC technology descriptions to support the decisions needed by the synthesizer. This includes the specification of:

    - the per-port capacitance,
    - the per-port electrical current,
    - the fabrication layers for routing,
    - the fabrication layers that form transistors,
    - the sheet resistivity of the fabrication layers,
    - the thickness of the fabrication layers,
    - the maximum current density for each routing layer,
    - the area capacitance to substrate (ground),
    - the sidewall capacitance to substrate (ground), and
    - the overlap coupling capacitance.

    These values are used to calculate the minimum widths of wires needed to support a given electrical current to avoid metal migration.

6.2 Limitations

The hierarchical layout method presented here produces circuits quickly. However, the limitations of this method should be noted.

- The area of these circuits is often larger than would be required if the circuit were compacted.
- The computational elements are assumed to have inputs on the south side and outputs on the north side to facilitate the functional nature of the algorithmic language and the corresponding hierarchical layout and routing method. This precludes more compact placement and routing that is possible when inputs and outputs are not restricted in this manner.
- Sizing of transistors is not performed by the synthesizer. This could result in sub-optimal performance.

6.3 Future Research

The following tasks need to be performed:

- The synthesizer currently (as of May 31, 1987) can place and route switching expressions consisting of Boolean operators and conditional statements. The implementation of the data type facility in the synthesizer has not been completed and should be if this is to be a prototyping tool.

- The sizing of transistors is not performed by the synthesizer. This requires flattening the hierarchy of the circuit to the transistor level, computing critical paths and then modifying transistor sizes. It is not obvious whether flattening the hierarchy will destroy the $\mathcal{O}(n)$ time complexity of the layout and the $\mathcal{O}(n^2)$ of the routing procedures.

- Space compaction is possible if the circuit hierarchy is flattened, but this could destroy the fast properties of the layout and routing procedures. In general, these procedures are *NP*-complete.

6.4 Conclusions

The traditional placement and routing techniques have been discarded and re-invented here in a new context. This context eschews the notion carried over from printed circuit board layout that circuits must be of fixed sizes and separated by fixed distances so that they can be placed and routed efficiently. This context facilitates design at a higher level in the design process, enabling the designer to make decisions that have much greater impact on a design than any made at the low levels of design. The methods described in this thesis make it possible to explore the behavioral design space quickly. The designer expresses algorithms and data types without having to specify a layout for their implementation or having to rely on time-consuming methods to obtain an implementation. Because the translation process is fast, experiments can be performed in which algorithms and data types are changed and then the non-behavioral attributes of the resulting circuits compared. Once a few implementations with the desired non-behavioral attributes have been obtained, the circuits can be fabricated via a "silicon foundry". If the area of the circuits is too large, a few options are possible. For the most compact layouts, the traditional lengthy process of compaction can be performed on the designs so that the circuits will fit within the bounds of a chip. For less demanding area restrictions, the CIF file produced by the ALICS Synthesizer synthesizer is hierarchical and can be edited using a mask layout editor such as Magic.

Bibliography

[Ada83] *Reference Manual for the Ada Programming Language* (1983).

[Aviz61] Algirdas Avizienis, Signed-Digit Number Representations for Fast Parallel Arithmetic, *IRE Transactions on Electronic Computers* (1961).

[Ayre79] Ron Ayres, Silicon Compilation—A Hierarchical Use of PLAs, pp. 311–326, in *Proceedings of the Caltech Conference on VLSI*, Caltech, Pasadena, California (January 1979).

[Ayre83] Ron Ayres, *VLSI Silicon Compilation and the Art of Automatic Microchip Design*, Prentice-Hill, Inc., Englewood Cliffs, New Jersey 07632 (1983).

[Back63] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, and M. Woodger, Revised Report on the Algorithmic Languge Algol 60, *Communications of the ACM*, 6(1):1–17 (January 1963).

[Back78] John Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM*, 21(8):613–641 (August 1978).

[Bake83] Brenda Baker, Sandeep Bhatt, and Frank Leighton, An Approximation Algorithm for Manhattan Routing, pp. 477–486, in *15th Annual ACM Symposium on Theory of Computing*, Boston, Massachusetts (April 25–27 1983).

[Barb79] Mario R. Barbacci, Gary E. Barnes, Roderic G. Cattell, and Daniel P. Siewiorek, *The ISPS Computer Description Language*, Technical Report CMU-CS-79-137, Carnegie-Mellon University Department of Computer Science (August 16 1979).

[Boro68] R. T. Borovec, *The Logical Design of a Class of Limited Carry-Borrow Propagation Adders*, Technical Report 275, University of Illinois Department of Computer Science (1968).

[Char82] Philippe Charles and Gerald Fisher, A LALR(1) Grammar for '82 Ada, *ACM Ada Sigplan Ada Letters*, II(2) (September, October 1982).

[Chow78] Catherine Chow, Logical Design of a Redundant Binary Adder, in *Proceedings of the 4th Symposium on Computer Arithmetic*, IEEE Computer Society (October 1978).

[Chow86] Salim Chowdhury, *Power and Ground Routing for Semi-Custom VLSI Circuits*, Technical Report CRI-86-19, University of Southern California, Computer Research Institute, Los Angeles, California 90089-0781 (March 1986), Ph.D. Thesis.

[Clin84] Ken Cline, Mel Cutler, Carl Kesselman, and Gary York, Automated Attribute Optimization for VLSI Systems, pp. 106–113, in *Third Annual International Phoenix Conference on Computers and Communications*, Phoenix, Arizona (March 1984), IEEE.

[Cohe81] Danny Cohen, On Holy Wars and a Plea for Peace, *IEEE Computer*, 14(10):48–54 (October 1981).

[Cuyk82] Robert Cuykendall, Anton Domic, William H. Joyner, Steve C. Johnson, Steve Kelem, Dennis McBride, Jack Mostow, John E. Savage, and Gabriele Saucier, Design Synthesis and Measurement, *VLSI ∩ Software Engineering Workshop Report* (October 1982).

[Cuyk84] Robert Cuykendall, Anton Domic, William H. Joyner, Steve C. Johnson, Steve Kelem, Dennis McBride, Jack Mostow, John E. Savage, and Gabriele Saucier, Design Synthesis in VLSI and Software Engineering, *The Journal of Systems and Software*, 4(1):7–12 (April 1984), This is a reprint of the workshop report.

[Dole81] Danny Dolev, Kevin Karplus, Alan Siegel, Alex Strong, and Jeffrey Ullman, Optimal Wiring Between Rectangles, pp. 312–317, in *13th Annual ACM Symposium on Theory of Computing*, Milwaukee, Wisconsin (May 11–13 1981).

[Hama85] Gordon Hamachi, *Designing Finite State Machines with PEG*, 1986 VLSI Tools: Still More Works by the Original Artists Report No. UCB/CSD 86/272, Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley (December 1985).

[Hon80] Robert W. Hon and Carlo H. Séquin, *A Guide to LSI Implementation, Second Edition*, Technical Report, Xerox Palo Alto Research Center (January 1980).

[Joha78] David L. Johannsen, *Bristle Blocks: A Silicon Compiler*, Technical Report, Caltech Dept. of Computer Science (January 1978), reprinted January 1979, Caltech Conference on VLSI, Pasadena, California.

[John78] Stephen C. Johnson, *Yacc—Yet Another Compiler-Compiler*, Technical Report, Bell Laboratories, Murray Hill, New Jersey (July 31 1978).

[John83] Stephen C. Johnson, Code Generation for Silicon, pp. 14–19, in *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas (January 24–26 1983).

[Joob86] Rostam Joobbani, *An Artifical Intelligence Approach to VLSI Routing*, Kluwer Academic Publishers (1986).

[Karp83] Kevin Karplus, *CHISEL—An Extension to the Programming Language C for VLSI Layout*, PhD dissertation, Stanford University (January 1983).

[Kele87] Steven H. Kelem, *A Method for Compact Two-Layer Routing of Permutations in Less Than n^2 Time*, Technical Report TR-0086A (2920-03)-1, The Aerospace Corporation (February 1987).

[Kern78] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey 07632 (1978).

[Kirr83] Hubert Kirrmann, Data Format and Bus Compatibility, *IEEE Micro*, 3(4):32–47 (August 1983).

[Leon85] H.W. Leong and C.L. Liu, Permutation Channel Routing, pp. 579–584, in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, Rye Town Hilton, Port Chester, N.Y. (October 7–10 1985).

[Lesk75] M. E. Lesk, *Lex—A Lexical Analyzer Generator*, Technical Report, Bell Laboratories, Murray Hill, New Jersey (October 1975).

[Mead80] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison Wesley Publishing Company, Reading, Mass. (1980).

[Meal55] G.H. Mealy, A Method for Synthesizing Sequential Circuits, *Bell System Technical Journal*, 34(5):1045–1079 (September 1955).

[Meer81a] L.G.L.T. Meertens and J.C. van Vliet, *Algol 68+, A Superlanguage of Algol 68 for Processing the Standard-Prelude*, Technical Report, Mathematisch Centrum, Amsterdam (Juni 1981).

[Meer81b] L.G.L.T. Meertens and J.C. van Vliet, *An Underlying Context-Free Grammar of Algol 68+*, Technical Report IW 171/81, Mathematisch Centrum, Amsterdam (Juli 1981).

[Mesh85] Farshad Meshkinpour and Miloš Ercegovac, A Functional Language for Description and Design of DIgital Systems: Sequential Constructs, in *IEEE Proceedings of the 22nd ACM/IEEE Design Automation Conference* (June 23–26 1985).

[Moor55] E.F. Moore, A Method for Synthesizing Sequential Circuits, *Bell System Technical Journal*, 34(5):1045–1079 (September 1955).

[Mukh86] Amar Mukherjee, *Introduction to nMOS and CMOS VLSI Systems Design*, Prentice-Hall, Englewood Cliffs, New Jersey (1986).

[Oust83] John Ousterhout, A Timing Analyzer for nMOS VLSI Circuits, pp. 57–69, in Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley (1983).

[Oust85] John Ousterhout, *Using Crystal for Timing Analysis*, 1986 VLSI Tools: Still More Works by the Original Artists Report No. UCB/CSD 86/272, Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley (December 1985).

[Pang86] Barry M. Pangrle and Daniel D. Gajski, State Synthesis and Connectivity Binding for Microarchitecture Compilation, pp. 210–213, in *International Conference on Computer-Aided Design* (1986).

[Pate85] Dorab Patel, Martine Schlag, and Miloš Ercegovac, *νFP*: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms, in G. Goos and J. Hartmanis, editors, *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science*, Springer-Verlag, Nancy, France (September 1985).

[Quar86] T. Quarles, A.R. Newton, D.O. Pederson, and A. Sangiovanni-Vincentelli, *SPICE 3A7 User's Guide*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley (April 1 1986).

[Rive82] Ronald L. Rivest and Charlse M. Fiduccia, A "Greedy" Channel Router, pp. 418–424, in *ACM IEEE Nineteenth Design Automation Conference Proceedings*, IEEE Computer Society and the ACM, Caesar's Palace, Las Vegas, Nevada (June 14–16 1982), IEEE Computer Society Order No. 416 or ACM Order No. 477820.

[Schl84] Martine Schlag, *Extracting Geometry from FP for VLSI Layout*, Technical Report CSD-840043, UCLA Computer Science Department (October 1984).

[Schl86] Martine Denise Francoise Schlag, *Layout From a Topological Description*, PhD dissertation, UCLA Computer Science Department (July 1986), CSD-860039.

[Schw78] Richard Schwartz, *An Axiomatic Semantic Definition of Algol 68*, PhD dissertation, Computer Science Department, University of California, Los Angeles (July 1978), UCLA-ENG-7838.

[Sequ82] Carlo H. Sequin, *Generalized IC Layout*, Technical Report, Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, California (January 6 1982).

[Sisk81] Jeffrey Mark Siskind, Ray Roger Southard, and Kenneth Walter Crouch, *Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions*, Technical Report, MIT Lincoln Laboratory, Lexington, Massachusetts (December 21 1981).

[Swif26] Jonathan Swift, *Gulliver's Travels*, Benjamin Mott, London (1726), (original title: *Travels into several Remote Nations of the World. By Lemuel Gulliver)*.

[UWN84] *VLSI Design Tools Reference Manual*, UW/NW Consortium, University of Washington, Seattle, Washington 98195, release 2.1 edition (October 1 1984), TR-84-08-07.

[Vlad81] A. Vladimirescu, Kaihe Zhang, A.R. Newton, D.O. Pederson, and A. Sangiovanni-Vincentelli, *SPICE Version 2G User's Guide*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley (August 10 1981).

[West85] Neil Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, *VLSI Systems Series*, Addison Wesley, Reading, Massachusetts (October 1985).

[Wijn65] A. van Wijngaarden, *Orthogonal Design and Description of a Formal Language*, Technical Report, Mathematisch Centrum, Amsterdam (October 1965).

[Wijn75] A. van Wijngaarden, B.J Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisher, Revised Report on the Algorithmic Language Algol 68, *Acta Informatica*, 5(1–3) (1975).

[Will77] John Douglas Williams, *STICKS—A New Approach To LSI Design*, PhD dissertation, MIT Computer Science Dept. (June 1977).

APPENDIX A

Semantics for the Row Data Type Constructor and Primitive Operators

A.1 Introduction

This appendix provides the semantics for a flexible way to implement arrays. An extension to the semantics defined for Algol 68 is presented that uniformly models weighted number systems and character strings. It is a modest proposal for a truce in the war between the Big-Endians and Little-Endians [Cohe81,Kirr83,Swif26].

If an array holds (western) characters, it is logical that the characters be numbered beginning with zero or one and that the numbering of the characters increase from left to right, modeling the direction of western reading. Thus in an array of characters denoted by (one-origin) "`this is a character array`", the first character is `t` and the 25th character is `y`.

If a linear array contains numbers that are periodic data, the indexing of elements usually increases from left to right, corresponding to left-to-right reading and increasing time.

If a linear array holds bits (or other numbers), the *logical* numbering scheme depends on which number system is being modeled. If a linear array of bits represents radix-two integers, then it is logical that the bits be numbered beginning with zero and that the numbering increases from right to left. This numbering scheme models the powers of the radix at each bit position in the array. Thus if an array of bits is denoted by (1,0,1,0,1,0), then the 1st bit (leftmost) is a 1, and the 6th bit (rightmost) is a 0. A radix-two integer can be denoted by **REV**(1,0,1,0,1,0)@0, where @0 changes the lower bound from 1 to 0. **REV** reverses the left and right bounds so that the 0th bit (rightmost) is a 0, and the 5th bit (leftmost) is a 1. If a linear array of bits represents radix-two fractions, then it is logical that the bits be numbered beginning with -1 and that it decreases from left to right. This numbering scheme models the powers of the radix at each bit position in the array. Thus in an array of bits denoted by **REV**((1,0,1,0,1,0)@-6), (denoting the value $\frac{42}{64}$), the -1th bit (2^{-1}) is a 1 and the -6th bit (2^{-6}) is a 0.

A.2 Background

Algol 68 has a data-typing mechanism called *rows* which is the Algol term for an array. Rows may have multiple subscripts, each with a pair of bounds. Algol 68 bounds are numbered from left to right in a multiply-dimensioned array. Each boundpair is declared as *lower-bound, colon, upper-bound.* If the lower-bound is missing, it defaults to one. For example,

[-5:3,0:7]**INT** a,b;	Declares two two-dimensional integer arrays called a and b. Each array contains seventy-two integers. The first lower-bound of a is −5, the second lower-bound is 0.

In Algol 68, values are represented by *denotations*. For example,

42	Is the integer denotation for the number forty-two.
4r222	Is the integer denotation for the number forty-two in radix four.
(0, 2, 0, 9)	Is a row-of-integer denotation. The bounds are [1:4], and the nine is the fourth integer in the row.
(0, 2, 0, 9)@4	Is a row-of-integer denotation with a revised-lower-bound. The bounds are [4:7] and the nine is the seventh integer in the row. (There are no elements in this array corresponding to positions one through three.)

A.3 Representations

In its present form, Algol 68 can represent only the left-to-right indexing scheme. One can easily modify the lower bounds of rows to achieve zero-based or any arbitrary base for indexing. [1:8]**BIT** and [8]**BIT** represent rows of **BIT**s with the same lower bound of one. [0:7]**BIT** represents a row of **BIT**s with a lower bound of zero. Algol 68 can represent only low-to-high subscripts as left-to-right denotations.

This is unfortunate, because it cannot accommodate an important representation scheme. The high-to-low subscripts, represented by left-to-right denotations are natural for representing radix number systems. In this representation, the high-order-digit is on the left. This representation also makes it possible for the subscript to be the same as the power of the radix at each digit position (E.g. the 0 subscript corresponds to the weight 2^0, three to the weight 2^3.)

A.4 Extensions

This section describes extensions to Algol 68 that allow both representation schemes and are easy to implement. The left-to-right or right-to-left ordering allows stylistic differences in numeric representations and introduces minor semantic differences (improvements?). This is useful because it enables a programmer to express data in whichever style is more readable.

The extensions are as follows:

- The lower and upper bounds in a boundpair are renamed. Instead of calling the two components of a boundpair the lower and upper bounds, they are called the left and right bounds. E.g. [7:0] represents subscripts increasing from right-to-left and [0:7] represents subscripts increasing from left-to-right.
- A row with a boundpair of [7:0] is represented internally the same way as is a row with a boundpair of [0:7]. The subscripting procedure is identical for both types of rows, and assignments between these two types are simple.
- The operators **LEB** and **RIB** are introduced and refer to the left or right bound, respectively.

- The lower bound **LWB** of a boundpair becomes min(**LEB**, **RIB**), and the upper bound **UPB** becomes max(**LEB**, **RIB**).

- An empty row is denoted by a boundpair of **SKIP**. (**SKIP** is a special token that represents a **void** value for all modes.)

- The slice specified by the boundpair [a,b] is almost the same as the slice [b,a]. The row elements that are selected are the same; however the slice boundpair becomes the new boundpair unless explicitly modified by a revised-lower-bound.

- Missing slice bounds are replaced by the corresponding **LEB** or **RIB**.

- Row denotations begin with the 1 index on the left, increasing to the right. The operator **REV** reverses the items in a row. For example, to denote the number 42 in radix 2 as an array of bits, one could write: **REV** (1,0,1,0,1,0)@0. This is stored internally at (0,1,0,1,0,1)@0. The **LEB** is 5 and the **RIB** is 0. Thus the 0th bit is a 0, and the 5th bit is a 1.

A.5 Semantics

This section defines the semantics of the row operators. The semantics are given for a single-dimension row. The multi-dimension cases follow similarly.

row of **mode**: A row of a given mode M with integer left-bound lb and right-bound rb is a value from the domain $INT_1 \times INT_2 \times M^*$. INT is a value from the domain of integers. M^* denotes zero or more values of mode M. A **row** of mode M is written only in this section as (lb, rb, M^*).

row denotation: A **row** denotation, written as $(v_1,\ v_2,\ \ldots,\ v_n)$, becomes the value $(1, n, (v_1,\ v_2,\ \ldots,\ v_n))$. The left-bound is one; the right-bound is n, the number of values in the **row** denotation.

LEB operator: The left-bound operator is both a monadic and a dyadic operator. The dyadic version takes two operands. The first operand is the boundpair number (from the left, beginning at one), and the second operand is a **row**. The monadic version takes one operand, the **row**, and yields an array value consisting of all the left-bounds for the array. For a single **row** r, (lb, rb, M^*), **LEB** r returns lb.

RIB operator: The right-bound operator is similar to the **LEB** operator except that for a single **row** r, (lb, rb, M^*), **RIB** r returns rb.

LWB operator: The lower-bound operator returns the lower bound of a row. It is similar to the **LEB** operator except that for a single **row** r, (lb, rb, M^*), **LWB** r returns $\min(lb, rb)$.

UPB operator: The upper-bound operator returns the upper bound of a row. It is similar to the **LEB** operator except that for a single **row**, (lb, rb, M^*), **UPB** r returns $\max(lb, rb)$.

@ operator: The revised lower-bound operator is a dyadic operator that creates a new copy of an array, changing only the lower-bound (not necessarily the left-bound). The corresponding upper-bound is also modified so that that there are still the same number of elements in the array. The first operand is the array whose bounds will be modified, and the second operand is a one-dimensional array with as many entries as the first array has dimensions. If one of the revised-lower-bounds is **SKIP**, the corresponding bounds are not modified.

```
$(lb, rb, M^*)$@ rlb = IF $lb \leq rb$
                       THEN $(rlb, rb - lb + rlb, M^*)$
                       ELSE $(lb - rb + rlb, rlb, M^*)$
                       FI
```

subscript: Subscripts are enclosed in square brackets following a row to select single items from the row. Subscripts are operators from the domain

$$(INT_1 \times INT_2 \times M^*) \times INT \rightarrow M \cup error.$$

For a **row r**, (lb, rb, M^*) and a subscript s, The value of r[s] is:

```
IF lb ≤ rb
THEN IF lb ≤ s ≤ rb
         THEN el(M, s-lb)
         ELSE subscript_out_of_bounds
         FI
ELSE IF rb ≤ s ≤ lb
         THEN el(M, s-rb)
         ELSE subscript_out_of_bounds
         FI
FI
```

where the function **el**(r,i) extracts the ith element of the zero-origin array r. The semantics of **el** are independent of representations for R and implementation methods for the extraction.

REV operator: The **REV** operator reverses the **LEB** and **RIB** of a **row**. The first operand is the boundpair number and the second is the array. The monadic version of the operator reverses all the boundpairs. For a **row** r, (lb, rb, M^*), **REV**r $= (rb, lb, M^*)$.

APPEND operator: This operator provides an intuitive method for concatenating two **rows** that represent radix numbers. The magnitude of the first argument is preserved as the second argument is appended to the right of the first argument.

The **APPEND** operator creates a new **row** by copying the second argument to the right of a copy of the first argument. The resulting **row** has an **LEB** identical to the **LEB** of the first argument. The new **RIB** is computed from the length of the new **row** extending in the direction of the boundpair of the first argument. If the direction of the second argument's boundpair is different from that of the first argument, the elements of the second **row** argument are copied in reverse order.

```
$(lb_1, rb_1, (v_1, \ldots, v_m))$ APPEND $(lb_2, rb_2, (w_1, \ldots, w_n)) =$
    IF $lb_1 \leq rb_1$
    THEN IF $lb_2 \leq rb_2$
            THEN $(lb_1, rb_1 + rb_2 - lb_2 + 1, (v_1, \ldots, v_m, w_1, \ldots, w_n))$
            ELSE  $(lb_1, rb_1 + lb_2 - rb_2 + 1, (v_1, \ldots, v_m, w_n, \ldots, w_1))$
            FI
    ELSE IF $lb_2 \leq rb_2$
            THEN $(rb_1, lb_1 + rb_2 - lb_2 + 1, (v_m, \ldots, v_1, w_n, \ldots, w_1))$
            ELSE  $(rb_1, lb_1 + lb_2 - rb_2 + 1, (v_m, \ldots, v_1, w_n, \ldots, w_1))$
            FI
    FI
```

For example, two **rows**, not necessarily representing radix numbers, can be appended as follows.

(1,0,1,1)@0 **APPEND** (0,0,1)@10] = (1,0,1,1,0,0,1)@0

The **row** (100,102,(0,0,1)) is appended to the right of **row** (0,3,(1,0,1,1)). The resulting **row** has the value (0,6,(1,0,1,1,0,0,1)).

Two **rows** representing radix numbers can be appended as follows.

REV(1,0,1,1)@0 **APPEND** **REV**(0,0,1)@100
= **REV**(1,0,1,1,0,0,1)@−3

If the array represents a radix five number (Note that the particular radix is irrelevant in to the **APPEND** operator.), this represents appending 001×5^{100} to the right of 1011_5. 001×5^{100} is effectively scaled so that the high-order digit (0) will be in the 5^{-1} position and then added to 1011_5 to yield 1011.001_5.

The **rows** in this example are (3,0,(1,0,1,1)) and (102,100,(0,0,1)). The value returned from **APPEND** is (3,-3,(1,0,1,1,0,0,1)).

PREPEND operator: This operator provides an intuitive method for concatenating two **rows** that represent radix numbers. This is similar to the **APPEND** operator except that the magnitude of the second argument is preserved after the first argument is prepended to the left of the second argument.

The **PREPEND** operator creates a new **row** by copying the first argument to the left of a copy of the second argument. The resulting **row** has an **RIB** identical to the **RIB** of the second argument. The new **LEB** is computed from the length of the new **row** extending in the direction of the boundpair of the second argument. If the direction of the first argument's boundpair is different from that of the second argument, the elements of the first argument **row** are copied in reverse order.

```
$(lb_1, rb_1, (v_1, \ldots, v_m))$ PREPEND$(lb_2, rb_2, (w_1, \ldots, w_n)) =$
    IF $lb_2 \leq rb_2$
    THEN IF $lb_1 \leq rb_1$
            THEN $(lb_2 - rb_1 + lb_1 - 1, rb_2, (v_1, \ldots, v_m, w_1, \ldots, w_n))$
            ELSE $(lb_2 - lb_1 + rb_1 - 1, rb_2, (v_1, \ldots, v_m, w_n, \ldots, w_1))$
            FI
    ELSE IF $lb_1 \leq rb_1$
            THEN $(rb_2 - rb_1 + lb_1 - 1, lb_2, (v_m, \ldots, v_1, w_n, \ldots, w_1))$
            ELSE $(rb_2 - lb_1 + rb_1 - 1, lb_2, (v_m, \ldots, v_1, w_n, \ldots, w_1))$
            FI
    FI
```

For example, two **rows**, not necessarily representing radix numbers, can be prepended as follows:

(1,0,1,1)@0 **PREPEND** (0,0,1)@100 = (1,0,1,1,0,0,1)@0.

The **row** (0,3,(1,0,1,1)) is prepended to the left of **row** (100,102,(0,0,1)). The resulting **row** has the value (96,102,(1,0,1,1,0,0,1)).

Two **rows** representing radix numbers can be prepended as follows:

$$\textbf{REV}(1,0,1,1)@0 \ \textbf{PREPEND} \ \textbf{REV}(0,0,1)@100$$
$$= \textbf{REV}(1,0,1,1,0,0,1)@{-3}.$$

If the array represents a radix five number (Note that the particular radix is irrelevant in to the **PREPEND** operator.), this example represents prepending 1011_5 (131_{10}) to the left of 001×5^{100}. 1011_5 is effectively scaled by 5^{103} and then added to 001×5^{100} to yield $1011001_5 \times 5^{100}$.

The **rows** in this example are (3,0,(1,0,1,1)) and (102,100,(0,0,1)). The value returned from **PREPEND**, is (106,100,(1,0,1,1,0,0,1)).

slicing: Slicing is the process of extracting a contiguous portion of a **row**. (This is known as a *subrange* in some languages.) Slicing is indicated by a range within square brackets. A range consists of two optional integer expressions separated by a colon. If one (or both) of these expressions is missing, it is replaced by the corresponding left or right bound. The new left and right bounds will be in the same order that are specified in the slice. The bounds cannot be reversed if one of the bounds is missing in the slice range.

When a slice is taken from a row, the boundpair is not adjusted automatically so that it has a revised lower bound of 1 if no revised lower bound is specified. (This is done in Algol 68, but does not seem necessary. This has the effect of extracting the slice and performing a numeric shift on the resulting value. This can be accomplished with the revised-lower-bound operator **@** if desired.)

For a **row** r, $(lb, rb, (v_1, \ldots, v_n))$, a slice is represented by r[l:r]. The resulting value is:

```
IF ($lb \leq$ l $\leq rb$ ANDIF $lb \leq$ r $\leq rb$ )
ORIF ($lb \geq$ l $\geq rb$ ANDIF $lb \geq$ r $\geq rb$)
THEN (l,r,$(v_{\min(l,r)}, \ldots, v_{\max(l,r)})$)
ELSE subscript_out_of_bounds
FI
```

APPENDIX B

Syntax of ALICS

The following is the syntax for ALICS. It is based on two gramars for Algol 68 from the Amsterdam Mathematisch Centrum [Meer81b,Meer81a], modified so that a parser could be produced from it by *yacc*[John78]. It is written in a form that is a combination of BNF[Back63] and a van Wijngaarden grammar[Wijn65]. The syntax is that of BNF, and the style is from vWg, being easier to read than BNF grammars. A preprocessor has been written that converts this grammar with semantics written in the style of *yacc* and converts it to the form for *yacc*.

The productions are in alphabetic order except for the production for the distinguished symbol of the grammar, **compilation input**, which appears first. A grammar rule consists of four parts—a left hand side, a colon, a right hand side, and a period. The left hand side is a grammar symbol, which is denoted by a word that may include spaces. Grammar symbols that end in the work **token**, generally are terminal symbols of the grammar and may not appear on the left hand side of a grammar rule. The right hand side consists of one or more alternatives, each separated by semicolons. Each alternative is a (possibly empty) list of grammar symbols separated by commas.

There are 205 grammar rules (including the alternatives). This is small compared to the 400 or so that are needed for an Ada[1] parser[Char82].

```
# productions #

compilation input:
    initialization, particular program.

assignation:
    identifer, becomes token, unit.

associativity declaration:
     left associative token, operator list;
      non associative token, operator list;
    right associative token, operator list.

bold else part:
    bold else token, serial clause;
    bold else if token, serial clause, bold then part,
        bold else token, serial clause.

bold in part:
    bold in token, case part list.

bold out part:
    bold out token, serial clause;
    bold ouse token, serial clause, bold in part, bold out token,
        serial clause.

bold then part:
    bold then token, serial clause.

brief if or case clause:
    open, serial clause, brief in part, brief out part, close token;
    open, serial clause, brief in part, close token.

brief in part:
    stick token, case part list.

brief out part:
    /* The following should be:
            stick token, serial clause but can't be due to ambiguities.
*/
```

[1]Ada is a registered trademark of the U.S. Government—Ada Joint Program Office.

```
    brief in part;
    stick colon token, serial clause, brief in part, brief out part.

by part:
    by token, unit.

call:
    primary, collateral brief clause.

case clause:
    bold case token, serial clause,
        bold in part, bold out part, bold esac token;
    bold case token, serial clause, bold in part, bold esac token.

case part:
    serial clause;
    specification, serial clause.

case part list:
    case part;
    case part, and also token, case part list.

cast:
    declarer, enclosed clause.

choice clause:
    brief if or case clause;
    case clause;
    if clause.

code:
    code token, denoter.

collateral bold begin:
    bold begin token.

collateral brief clause:
    open, joined portrait ety, close token.

collateral clause:
    collateral bold begin, bold end token;
    collateral bold begin, joined portrait, bold end token;
    collateral brief clause.
```

```
common declaration:
    associativity declaration;
    commutativity declaration;
    identity declaration;
    mode declaration;
    operation declaration;
    priority declaration.

commutativity declaration:
    commutative token, operator list;
    non commutative token, operator list.

declaration:
    ldecety common declaration.

declarative:
    declarer, parameter joined definition;
    declarative, go on token, declarer, parameter joined definition.

    # formal parameter list or structure definition #
declarative pack:
    open, declarative, close token.

declarer:
    bold mode token;
    procedure declarator;
    rows of declarator;
    structured with declarator;
    union declarator.

declarer or code:
    declarer;
    code.

do part:
    do token, serial clause, od token.

enclosed clause:
    choice clause;
    collateral clause;
    loop clause.
```

```
exec part:
    series.

for part:
    for token, series;
    par token, series.

formal procedure plan:
    declarer;
    open, joined declarer pack, close token, declarer.

formula:
    formula, left operator 0, formula;
    formula, left operator 1, formula;
    formula, left operator 2, formula;
    formula, left operator 3, formula;
    formula, left operator 4, formula;
    formula, left operator 5, formula;
    formula, left operator 6, formula;
    formula, left operator 7, formula;
    formula, left operator 8, formula;
    formula, left operator 9, formula;
    formula, left operator 10, formula;
    formula, nonassoc operator 0, formula;
    formula, nonassoc operator 1, formula;
    formula, nonassoc operator 2, formula;
    formula, nonassoc operator 3, formula;
    formula, nonassoc operator 4, formula;
    formula, nonassoc operator 5, formula;
    formula, nonassoc operator 6, formula;
    formula, nonassoc operator 7, formula;
    formula, nonassoc operator 8, formula;
    formula, nonassoc operator 9, formula;
    formula, nonassoc operator 10, formula;
    formula, right operator 0, formula;
    formula, right operator 1, formula;
    formula, right operator 2, formula;
    formula, right operator 3, formula;
    formula, right operator 4, formula;
    formula, right operator 5, formula;
    formula, right operator 6, formula;
    formula, right operator 7, formula;
    formula, right operator 8, formula;
    formula, right operator 9, formula;
    formula, right operator 10, formula;
    right operator 0, formula;
    left operator 0, formula;
    nonassoc operator 0, formula;
    right operator 1, formula;
    left operator 1, formula;
    nonassoc operator 1, formula;
    right operator 2, formula;
    left operator 2, formula;
    nonassoc operator 2, formula;
    right operator 3, formula;
    left operator 3, formula;
    nonassoc operator 3, formula;
    right operator 4, formula;
    left operator 4, formula;
    nonassoc operator 4, formula;
    right operator 5, formula;
    left operator 5, formula;
    nonassoc operator 5, formula;
    right operator 6, formula;
    left operator 6, formula;
    nonassoc operator 6, formula;
    right operator 7, formula;
    left operator 7, formula;
    nonassoc operator 7, formula;
    right operator 8, formula;
    left operator 8, formula;
    nonassoc operator 8, formula;
    right operator 9, formula;
    left operator 9, formula;
    nonassoc operator 9, formula;
    right operator 10, formula;
    left operator 10, formula;
    nonassoc operator 10, formula;
    formula, of token, formula;
    primary.

from part:
    from token, unit.

identifier:
    tag token.
```

```
identity declaration:
    declarer, identity joined definition;
    procedure token, identity joined definition.

identity definition:
    identifier, operator, ldecety source.

identity joined definition:
    identity definition;
    identity joined definition, and also token, identity definition.

if clause:
    bold if token, serial clause, bold then part, bold else part,
        bold fi token;
    bold if token, serial clause, bold then part, bold fi token.

indexer:
    trimscript;
    indexer, and also token, trimscript.

initialization:
    /* empty */.

joined declarer pack:
    declarer;
    joined declarer pack, and also token, declarer.

joined portrait:
    serial clause;
    joined portrait, and also token, serial clause.

joined portrait ety:
        /* empty */;
    joined portrait.

ldec source choice:
    relational, length denoter, colon token, unit or code.

ldec source choice list:
    ldec source choice;
    ldec source choice list, and also token, ldec source choice.
```

```
ldecety common declaration:
    common declaration;
    ldec token, common declaration.

ldecety source:
    unit or code;
    choice token, open, ldec source choice list, close token.

length denoter:
    denoter;
    other tao token, denoter.

longs:
    long token;
    longs, long token.

loop clause:

    to part,                                     repeating part;
    by part,                                     repeating part;
    by part,                           to part,  repeating part;
    from part,                                   repeating part;
    from part,                         to part,  repeating part;
    from part, by part,                          repeating part;
    from part, by part,                to part,  repeating part;
    for part,                                    repeating part;
    for part,                          to part,  repeating part;
    for part,          by part,                  repeating part;
    for part,          by part,        to part,  repeating part;
    for part, from part,                         repeating part;
    for part, from part,               to part,  repeating part;
    for part, from part, by part,                repeating part;
    for part, from part, by part,to part, repeating part;
    for part, seq part,                          repeating part;
    for part, seq part, bold out part,           repeating part;

mode declaration:
    mode token, mode joined definition.

mode definition:
    bold tag token, operator, declarer or code;
    sizes, bold tag token, operator, declarer or code.
```

```
mode joined definition:
    mode definition;
    mode joined definition, and also token, mode definition.

open:
    open token.

operation declaration:
    operator token, operation joined definition;
    operator token, formal procedure plan,
        operation joined definition.

operation definition:
    operator display, operator, ldecety source.

operation joined definition:
    operation definition;
    operation joined definition, and also token,
        operation definition.

operator:
    bold tag token;
    other tao token;
    left operator 0;
    left operator 1;
    left operator 2;
    left operator 3;
    left operator 4;
    left operator 5;
    left operator 6;
    left operator 7;
    left operator 8;
    left operator 9;
    left operator 10;
    nonassoc operator 0;
    nonassoc operator 1;
    nonassoc operator 2;
    nonassoc operator 3;
    nonassoc operator 4;
    nonassoc operator 5;
    nonassoc operator 6;
    nonassoc operator 7;
    nonassoc operator 8;
```

```
        nonassoc operator 9;
        nonassoc operator 10;
        right operator 0;
        right operator 1;
        right operator 2;
        right operator 3;
        right operator 4;
        right operator 5;
        right operator 6;
        right operator 7;
        right operator 8;
        right operator 9;
        right operator 10.

operator display:
    operator;
    undefined operator;
    choice token, open, operator list, close token.

operator list:
    operator;
    undefined operator;
    operator list, and also token, operator;
    operator list, and also token, undefined operator.

parameter joined definition:
    identifier;
    parameter joined definition, and also token, identifier.

particular program:
    enclosed clause.

prelude:
    declaration;
    prelude, go on token, declaration.

primary:
    call;
    cast;
    denoter;
    enclosed clause;
    identifier;
    primary, brief sub token, indexer, brief bus token.
```

```
priority declaration:
    priority token, priority joined definition.

priority definition:
    /* operator, equals token, digit token. */
    operator display, operator, denoter.

priority joined definition:
    priority definition;
    priority joined definition, and also token, priority definition.

procedure declarator:
    procedure token, formal procedure plan.

relational:
    operator.

repeating part:
    do part;
    while part, exec part, do part.

revised lower bound:
    at token, unit.

routine text:
    declarer, colon token, unit or code;
    declarative pack, declarer, colon token, unit or code.

row rower:
    /* empty */ ;
    unit, colon token, unit;
    unit;
    colon token.

rower.
    row rower;
    rower, and also token, row rower.

rower bracket:
    brief sub token, rower, brief bus token.

rows of declarator:
        rower bracket, declarer.

    seq part:
        unit.

    serial clause:
        series;
        prelude, go on token, series.

    series:
        unit.
    /*     series, go on token, unit. */

    shorts:
        short token;
        shorts, short token.

    sizes:
        shorts;
        longs.

    skip token:
        bold skip token;
        tilde token.

    specification:
        /* The syntax of this is changed slightly to:
            1. make it easier to parse.
            2. allow for specifications other than modes
                (in place of choice using integral and choice using boolean).
        */
        rower bracket, colon token;
        brief sub token, declarer, brief bus token, colon token;
        brief sub token, declarer, identifier, brief bus token,
colon token.

    structured with declarator:
        structure token, declarative pack.

    tertiary:
        formula;
        nil token.
```

```
to part:
    downto token, unit;
    to token, unit;
    upto token, unit.

trimscript:
    /* empty */;
    unit;
    colon token;
    colon token, revised lower bound;
    colon token, unit;
    colon token, unit, revised lower bound;
    unit, colon token;
    unit, colon token,    revised lower bound;
    unit, colon token, unit;
    unit, colon token, unit, revised lower bound;
    revised lower bound.

union declarator:
    union token, open, joined declarer pack, close token.

unit:
    assignation;
    routine text;
    skip token;
    tertiary;
    tertiary, is not token, tertiary;
    tertiary, is token, tertiary.

unit or code:
    unit;
    code.

while part:
    until token, series;
    while token, series.
```