

**A FUNCTIONAL STYLE DESCRIPTION OF DIGITAL SYSTEMS**

**John Shelby Worley**

**February 1986  
CSD-860054**



UNIVERSITY OF CALIFORNIA

Los Angeles

A Functional Style Description  
of Digital Systems

A thesis submitted in partial satisfaction of the  
requirements for the degree of Master of Science  
in Computer Science

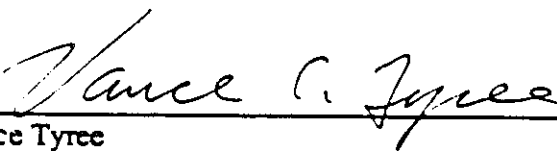
by


John Shelby Worley

1986

© Copyright by  
John Shelby Worley  
1986

The thesis of John Shelby Worley is approved.

  
\_\_\_\_\_  
Vance Tyree

  
\_\_\_\_\_  
David Rennels

  
\_\_\_\_\_  
Milos Ercegovac, Committee Chair

University of California, Los Angeles

1986



## TABLE OF CONTENTS

1.	INTRODUCTION	
1.1	Digital System Specification .....	1
1.2	Networks and Net Lists .....	2
1.3	High Level Network Description .....	5
1.4	Previous Work .....	6
1.5	Objective and Scope of the Thesis .....	8
2.	NET LIST DESCRIPTION AND FUNCTIONAL PROGRAMMING	
2.1	Overview .....	10
2.2	Procedural Description .....	10
2.3	Limitations of Procedural Description .....	12
2.3.1	Data Representation .....	13
2.3.2	Module Combination .....	15
2.4	Functional Style Description .....	17
3.	THE UCLA FUNCTIONAL PROGRAMMING LANGUAGE	
3.1	Overview .....	20
3.2	Objects .....	20

The functional style approach to describing digital systems has also been used. Lahti [Lahti81] showed that a functional style language could be used to specify combinational circuits and investigate their behavior. This was extended to sequential systems by Sheeran in [Sheeran84]. Functional style languages have also been used to extract layout information [Schlag84], examine timing in combinational and sequential systems [Meshkinpour85], and as the basis for a design environment [Patel85].

### 1.5. Objective and Scope of the Thesis

The objective of this thesis is to develop a high level method for describing and managing a net list representation of digital systems. Both the description language and the method for net list extraction will be examined.

In Chapter 2, a high-level computer language (Pascal) is used to describe a simple net list; its problems and limitations are discussed. The motivation for a *functional style* description is developed. Chapter 3 presents the syntax, semantics, and features of the UCLA Functional Programming Language, which is used subsequently as the net list description language.

The use of functional style net list description is covered in Chapter 4. Two styles are examined and compared: the picture style, which reflects the structure of the system; the generator style, based on the synthesis of the system. The "flow of information extraction" technique of extracting net lists is explored.



4.2	“Picture” Style Net List Description .....	38
4.3	“Generator” Style Net List Description .....	42
4.4	Tracing and Net List Extraction .....	44
5.	FUNCTIONAL NET LIST DESCRIPTION IN VLSI DESIGN	
5.1	Overview .....	49
5.2	The Need for a Multilevel Systems Language .....	49
5.3	Functional Description in the Design Cycle .....	52
5.4	A Functional Design Environment .....	53
5.4.1	The BDL System Description Language .....	55
5.4.2	Translation Considerations .....	56
5.4.3	Sample Translation: A Full Adder .....	59
5.4.4	A Ripple Carry Adder Layout .....	63
5.4.5	The Effect of Hierarchy .....	68
6.	CONCLUSION	
6.1	Summary .....	73
6.2	Conclusions .....	74
6.3	Future Work .....	75
A.	A COLLECTION OF HARDWARE DESCRIPTIONS	
A.1	Binary Decoders .....	78

3.3	Application .....	21
3.4	Primitives .....	21
3.4.1	Selector Primitives .....	22
3.4.2	Structural Primitives .....	23
3.4.3	Predicate and Length Primitives .....	25
3.4.4	Arithmetic and Logical Operations .....	26
3.4.5	Constants .....	27
3.5	Functional Forms .....	28
3.5.1	Composition .....	28
3.5.2	Construction .....	29
3.5.3	Mapping .....	29
3.5.4	Apply to All .....	30
3.5.5	Inserts .....	31
3.5.6	Conditional .....	33
3.6	User Defined Macros .....	33
3.6.1	Macro Definition .....	33
3.6.2	Macro Execution Tracing Facility .....	35
3.7	Symbolic Execution .....	36
4.	FUNCTIONAL NET LIST DESCRIPTION AND GENERATION	
4.1	Overview .....	38

## LIST OF FIGURES

3.1	Symbolic Logical Operation Result Matrix .....	37
4.1	A half adder circuit and FP description .....	39
4.2	A full adder circuit and FP description .....	40
5.1	An FP Design Environment .....	53
5.2	Full Adder FP Description .....	59
5.3	BDL Definitions of Basic Logic Gates .....	60
5.4	BDL Description of Binary Half Adder .....	62
5.5	BDL Description of Binary Full Adder .....	63
5.6	Ripple Carry Adder FP Description .....	64
5.7	Four-Bit Ripple Carry, Top Level View .....	66
5.8	Four-Bit Ripple Carry, Logic Level .....	67
5.9	Routing Data for 4-Bit Ripple Carry Adder .....	68
5.10	Four-Bit Ripple Carry, No Hierarchy, Top Level View .....	69
5.11	Four-Bit Ripple Carry, No Hierarchy, Logic Level .....	70
5.12	Routing Data for 4-Bit Ripple Carry Adder - Flat .....	71
A.1	A Single Bit Decoder .....	79
A.2	A Four-Bit Coincident Decoder .....	81
A.3	A Four-Bit Tree Decoder .....	84

A.1.1	Coincident Decoders .....	79
A.1.2	Tree Decoders .....	80
A.2	Encoder .....	83
A.3	Multiplexer .....	86
A.4	Adders ( <i>vipera sumopus</i> ) .....	87
A.4.1	Basic Adders .....	88
A.4.2	Ripple Carry Adder .....	89
A.4.3	Conditional Sum Adder .....	90
A.4.4	A BCD Adder Module .....	95
	References .....	98

## ACKNOWLEDGEMENTS

Thanks to the supporting cast: Genesis, Queen, Pink Floyd, and Rush. Thanks go out to the FP group for so many things; the figures in Appendix A would not have been possible without the efforts of Martine Schlag. A special thank you to the VLSI design group at Hewlett-Packard, especially Dan Conway, who helped immeasurably with BDL. A special award for Dorab Patel, whose ideas formed the basis for much of this work. Above all, the author wishes to thank his parents, his advisor - the one and only Miloš Ercegovic - and Alison, who never lost faith.

This work was supported in part by ONR Contract N00024-83-K-0493, and by Rockwell/UC Micro Grant 157.

A.4	An Eight-Bit Encoder .....	85
A.5	A 4 x 2-to-1 Multiplexer .....	87
A.6	A Four-Bit Ripple Carry Adder - Block Diagram .....	91
A.7	A Four-Bit Conditional Sum Adder .....	94
A.8	A BCD Adder Module .....	97

## CHAPTER 1

### INTRODUCTION

#### 1.1. Digital System Specification

The design of a digital system is aided greatly by the ability to clearly describe its structure and behavior in a high level language [Shiva79]. Besides providing a system specification, a system representation can also serve as a computer simulation language. This permits rapid testing and preliminary debugging of designs; tuning and analysis can be done without repeated physical realizations. The specification might also serve as input to automated design processes, such as placement and routing on a printed circuit board.

A behavioral/structural specification can be given in both the top-down and bottom-up design methodologies. In the top-down approach, a description specifies the behavior and interface for each subsystem at each level of the design hierarchy. Because the design is on paper, it is easy to experiment with different partitions of functionality. The precise definition of each subsystem allows them to be developed and tested in parallel. Further, behavioral simulation at higher levels can uncover possible problems and pinpoint considerations in implementing lower levels *before* they are designed.

Specification assists the bottom-up strategy of putting simpler modules together to achieve the desired operation. The designer can use simulation to

ABSTRACT OF THE THESIS

A Functional Style Description  
of Digital Systems

by

John Shelby Worley

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor Miloš Ercegovac, Chair

The advantages of a high level digital system description language are realized in rapid prototyping, decreased testing and debugging time, increased flexibility to evaluate and experiment and a smoother transition from system design to implementation. The *procedural* style of description is limited by an implicit sequential machine model. A *functional* style description, however, can express the functionality and structure of a digital system clearly and concisely. The hierarchal nature of a functional description makes it readily usable as a multilevel specification language in the VLSI design environment.



and one or more outputs. Some inputs are the output of another element; others, the *system inputs*, are generated externally. A subset of all outputs are the *system outputs*, most often the final result(s) of a computation or a sequence of control signals.

Modeling a system as a network provides a good basis for specification:

- (1) The model is general. There is no restriction on the type of components or the structure of the system.
- (2) The operation of the network is easily understood. The output of each element can be determined by examining its operation and the operation of all elements generating its inputs.
- (3) The model is hierarchic: each element in the network can, itself, be a network of yet simpler components. For example, a multiplexer may be one component in a larger system. The multiplexer, in turn, is a system consisting of a decoder, an array of and gates and an or gate tree.
- (4) Parallelism and dependencies are explicitly expressed by the input-output relationship of the components.

A simple method of describing a network is by listing each element with its inputs and outputs. Each element is labeled with a mnemonic for its operation; all inputs and outputs must be uniquely labeled. This linear, textual representation of a network is called a *net list*. This thesis will use the following format for a single entry, or element, of a net list:

**ELEM** *input*<sub>1</sub>, *input*<sub>2</sub> ··· *input*<sub>*n*</sub> :: *output*<sub>1</sub>, *output*<sub>2</sub> ··· *output*<sub>*m*</sub>

experiment with different combinations, evaluating functionality and performance and deriving system parameters. Once designed and tested, specifications for new subsystems can be stored away in libraries for future reference.

To be useful in describing digital systems, the specification language must be general enough not to restrict the overall structure of the system. This requires that there are no implicit or explicit assumptions on how systems are put together. In addition, the specification language should:

- (1) have clear semantics, so that a description can be easily understood by either a person or a computer program,
- (2) be hierarchical, to be able to describe both primitive and higher level parts of the system clearly, and
- (3) be able to explicitly express parallel and sequential operation within the system.

The structure, strengths, and limitations of a specification language arise from its abstract model of a digital system. If the underlying model cannot express an operation or a structure, the language will hinder problem solutions rather than help. In the next section, we will examine the *network* model of digital systems that serves as the basis for system specification.

## 1.2. Networks and Net Lists

A digital system can be viewed as an interconnection of simpler subsystems. This *network* model represents a system in terms of the operation and input-output relationship of its components. Each element in the network has zero or more inputs

<b>full_add</b>	x, y, Cin	::	Cout, s
<b>half_add</b>	x, y	::	C <sub>1</sub> , s <sub>1</sub>
<b>and</b>	x, y	::	C <sub>1</sub>
<b>xor</b>	x, y	::	s <sub>1</sub>
<b>half_add</b>	s <sub>1</sub> , Cin	::	C <sub>2</sub> , s
<b>and</b>	s <sub>1</sub> , Cin	::	C <sub>2</sub>
<b>xor</b>	s <sub>1</sub> , Cin	::	s
<b>or</b>	C <sub>1</sub> , C <sub>2</sub>	::	Cout

### 1.3. High Level Network Description

A net list specification of a digital system presents the complete connectivity of the system. Being a textual representation of the network model in a simple, regular format, it is usable by both humans and computer programs. Unfortunately, a human user would rapidly become lost in the sheer volume of information. System specifications can easily become long and bulky; even net lists for simple systems may be tedious to generate by hand or to read and understand. Use of hierarchy in the net list can alleviate this difficulty, but the user would still be forced to think in terms of module interconnection.

In translating the system to a net list description, the *intent*, i.e., the higher level meaning, of the structure and behavior of the system cannot be captured. The equation

$$P = Ax^2 + Bx + C$$

is more readily recognizable as a second degree polynomial than the equivalent net list:

If a set of elements are subsystems of a more complex element, this hierarchy will be denoted by indentation, i.e., in the net list

```
ELEM  x, y, z  ::  a, b
      SUB1    x, y    ::  a, t
      SUB2    t, z    ::  b
```

the elements SUB1 and SUB2 make up a network contained within the higher level element ELEM; the connection labeled t is internal to ELEM, hidden from higher levels of the system.

Consider the binary full adder, a system with three inputs (two summand bits and an initial carry) and two outputs (a sum bit and a final carry). Using a top down approach, this can be implemented as a network of two half adders (adds two bits to produce a sum and a carry) and an or gate as follows:

```
full_add  x, y, Cin  ::  Cout, s
half_add  x, y       ::  C1, s1
half_add  s1, Cin   ::  C2, s
or        C1, C2  ::  Cout
```

Each half adder, in turn, consists of an exclusive-or (xor) gate to produce the sum and an and gate to generate the carry. Thus, the specification of a full adder down to the gate level would be

defined once in the description. Connections can be logically organized into bundles having more meaning than the individual connections taken together, e.g., a single byte width data path instead of eight individual wires.

These languages provide a more human readable form of a connectivity description than a simple net list; however, they still force the designer or reader to think in terms of module interconnection instead of the overall operation on the system. *Procedural* description languages represent a digital system in terms of high level operations, which are later compiled into lower level, implementation specific details. The operation and sequencing can directly reflect the hardware behavior, as found in CDL [Chu65], DDL [Duley68] or any of many register transfer languages. Higher level languages, such as HARPA [Viega84], ZEUS [Lieberherr82],  $\Xi$  (Xi) [Feldman83] and the proposed VHSIC description language VHDL [Shahdad85], use idioms found in software languages, such as assignment, general arithmetic expressions, and procedure calls. The use and drawbacks of procedural languages will be examined in Chapter 2.

One compromise between the two previous methodologies is to describe both the functionality and the structure of the system. Robinson and Dion [Robinson82] use the Modula-2 language to describe each module twice: once for the functionality, once for the structure. The language ELLA [Morison82] uses a function oriented syntax to express these two facets in a single description.

MULTIPLY	B, x	::	M1
ADD	C, M1	::	S1
MULTIPLY	x, x	::	M2
MULTIPLY	A, M2	::	M3
ADD	S1, M3	::	P

though they contain exactly the same information about the computation. Even with good use of hierarchy, the form of the net list can only express the *what* characteristics of a system, not the *how* or *why*.

Difficulties of net list specification can be overcome by using a high level description from which the net list can be derived. This is similar to the use of high level computer programming languages, translated to machine instructions by a compiler; the net list can be produced rapidly and without translation errors. By expressing the operation, structure, and intent of a digital system while hiding other details (e.g., the intermediate sum S1 in the polynomial net list), the high level description is semantically denser, more readable and easier to manage than the equivalent net list.

#### 1.4. Previous Work

The advantages of specification languages in the design of digital systems have long been recognized [Dasgupta84]. Many languages, with different assumptions, intents, and features, have been proposed and implemented. Connection oriented languages, such as HIDSLS [Lim65] and BDL [Slutz84], model a system as interconnected structures or blocks. The advantage of these languages over a simple net list description is the degree of abstraction that can be achieved. System components can be hierarchically combined into more complex units, which need only be

Finally, in Chapter 5, the FP description of VLSI circuits is discussed; the advantages of an FP description as a base for a design environment are examined. The translation and layout of a ripple carry adder, starting from the FP description, is examined as a practical application of functional programming to VLSI design.

## CHAPTER 2

### NET LIST DESCRIPTION AND FUNCTIONAL PROGRAMMING

#### 2.1. Overview

A net list description language (NLDL) provides a high level representation of the network model of a digital system. The language should capture the operation, components, connectivity and high level semantics of the system without constraining the high level solution or forcing the designer to manage too many details.

In this chapter, two styles of languages are discussed. The *procedural* style language, though useful for describing algorithms, is limited by the underlying model of computation. This leads to consideration of a *functional* style language; a functional language is presented in the next chapter.

#### 2.2. Procedural Description

A digital system can be considered to implement an algorithm, such that each step from system inputs to system outputs is realized by one or more elements in the system. As noted in the previous chapter, some existing hardware description languages have been modeled after computer programming languages, which describe software algorithms: HARPA and ZEUS are based on Pascal,  $\Xi$  on C, AHPL [Hill75] on APL. and VHDL on Ada. Given a algorithmic, or *procedural*, view of a digital system, the Pascal language, as an example of a procedural language, will be used to



examine how a binary full adder might be described.

Digital circuits operate on binary values. This can be reflected in Pascal by defining a type *bit* equivalent to the Pascal type *boolean*. The first step is to define the logic gates used in a full adder: **and**, **or** and **exclusive or (xor)**.

```
types
  bit = boolean;

function andgate(x, y : bit) : bit;
begin
  andgate := x and y
end;

function orgate(x, y : bit) : bit;
begin
  orgate := x or y
end;

function xorgate(x, y : bit) : bit;
begin
  xorgate := x <> y { * Simulate exclusive or *}
end;
```

Once the basic gates are defined, the next step is to describe a half adder circuit. Unlike a logic gate, however, a half adder has two outputs: the sum and carry bits. Since Pascal function can only return a single value, the two half adder outputs will be returned by *side effect*, i.e., changing a value outside the function. Since there is no explicit value returned, the binary half adder takes the form of a *procedure* as follows:

```

procedure half_add(x, y : bit; var Sum, Cout : bit) : bit;
begin
    Sum := xorgate(x, y);
    Cout := andgate(x, y)
end;

```

Now, the full adder can be constructed from two half adders and an or gate.

Like the half adder, the full adder is a procedure, returning its two outputs by side effect.

```

procedure full_add(x, y, Cin : bit; var Sum, Cout : bit) : bit;
var
    Stmp, Ctmp, Ctmp2 : bit;
begin
    half_add(x, y, Stmp, Ctmp);
    half_add(Stmp, Cin, Sum, Ctmp2);
    Cout := orgate(Ctmp, Ctmp2)
end;

```

### 2.3. Limitations of Procedural Description

Though the *functionality* of a digital system can be adequately described in an algorithmic style, the usefulness of procedural languages for specification is limited by the implicit underlying machine model. Almost all procedural languages mimic, albeit at a higher level, a *Von Neumann machine*. The classic Von Neumann machine, is a single central processing unit (CPU) connected to a linear, addressable memory by a fixed width data path. The CPU sends addresses and data to the memory; the memory sends instructions and data to the CPU.

Most procedural languages provide only a small degree of abstraction from this model of computation. Variables are simply convenient symbolic names for

memory addresses; data types are just an indirect means of grouping and organizing memory cells. Even the most elaborate flow control structures can be decomposed into test-and-branch instructions. A procedural style language, while hiding many of the details of machine and assembly level programming, maintains the sequential, one-step-at-a-time nature of the Von Neumann machine.

The implicit, sequential machine model produces two fundamental weaknesses in the procedural style of system description: the representation of system information (data) is limited and inflexible; constructs for building larger systems from the descriptions of smaller ones are few, and cannot clearly express the structure of the system. The cause of these weaknesses and the problems they introduce are discussed in detail below.

### **2.3.1. Data Representation**

Data representation in most high level languages is closely tied to the organization of memory in a Von Neumann machine. Basic data types normally correspond directly either to a single memory storage cell (e.g., a byte), or a small integral number of cells which can be directly manipulated by machine instructions (e.g., words, addresses).

Mechanisms to build other data types, such as arrays or structures, only form combinations of the basic types; further, these new types are static templates, unable to adapt and change as the computation progresses. Dynamic storage management, e.g., flexible arrays or linked lists, still only builds on the basic data type, and makes

the designer painfully aware of how information in the digital system is represented.

One difficulty this memory oriented representation introduces is in the description of modules with a variable number of inputs and outputs. An N-bit adder, for example, must map the input and output bit vectors into some organization of memory, usually an array. Operating on arrays normally requires some form of repetitive looping, involving language constructs (e.g., a *for* loop) and information (e.g., an index variable) extraneous to the structure and function of the system. This difficulty even appears in languages with array operators if the inputs or outputs cannot be forced into an array organization.

A more important difficulty introduced by the data representation is the limitation on the utility of functions. Almost all high level system description languages use functions, commonly for low level primitives, such as logic gates. The description of more complex modules as functions, however, and their use in expressions, is constrained by limits the language may place on the *data type* a function may return.

One solution, adopted by some description languages, is to return *all* values by side effect, as in the Pascal descriptions above. This use of side effects, however, introduces an artificial distinction between modules that return a single value, modeling them as functions, and those with more than one output, modeling them as procedures.

Modules with more than one output can be described as functions by choosing one output value to return as the explicit result. If the carry-out bit of the half and full

adders is returned as the result of the function, the modules can be described as

```
function half_add(x, y : bit; var Sum : bit) : bit;  
begin  
    Sum := xorgate(x, y);  
    half_add := andgate(x, y)  
end;
```

and

```
function full_add(x, y, Cin : bit; var Sum : bit) : bit;  
var  
    Stmp, Ctmp, Ctmp2 : bit;  
begin  
    Ctmp := half_add(x, y, Stmp);  
    Ctmp2 := half_add(Stmp, Cin, Sum);  
    full_add := orgate(Ctmp, Ctmp2)  
end;
```

Unfortunately, this method only reduces the number of outputs returned through side effect by one, while introducing the additional complexity of choosing and remembering which value is explicitly returned.

### 2.3.2. Module Combination

A top-down design methodology decomposes the functionality of a system into progressively simpler units, whether they be arithmetic expressions or integrated circuits. A bottom-up approach starts with simpler units and builds toward the desired functionality. Both are based on the combination of smaller units into more complex systems.

In a procedural style language, the basic units are the steps in the computation. The combination of two steps requires that the first step modify some portion of the

global state, which can then be referenced by the next step. Language features, such as arithmetic expressions, can combine functions without explicit storage of results, but their use is restricted by the limited utility of functions, discussed above.

In general, storing and fetching data from global storage is the only means of combining units in a procedural style language. This splits a description into two domains: *expressions*, which return values, and *statements*, which modify the state of global storage through *assignment*.

While any combination of modules *can* be described using global storage to represent connections, it lacks the ability to clearly and explicitly represent structure of the system. The connection of modules is implicit in the description, so the structure must be extracted to be seen or analyzed. Instead of expressing how modules are combined, a procedural style description hides the structure in a clutter of temporary variables and side effects.

Constrained by an inflexible data representation and weak, implicit combining forms, a procedural description style forces a designer to compress the two dimensional structure of a net list into a one dimensional, sequential stream. The task of specification becomes concerned with how best to map the system into a sequence of steps, instead of concentrating on how modules are combined. Using a procedural style description complicates the design of a digital system, obscures the intent and structure of the system, limits the ability to use existing systems as new building blocks, and makes the net list description of the system difficult to extract.

## 2.4. Functional Style Description

The ability of procedural languages to describe digital systems is restricted by the underlying, sequential model of computation. A language designer, of course, is not bound by the restrictions of the Von Neumann machine. Language features can be implemented to circumvent or remove specific difficulties. The difficulties introduced by the memory oriented data representation can be overcome by using a more flexible and dynamic model of data. The list notation used in LISP languages presents an attractive alternative:

- (1) The basic data units, *symbols*, are simple, yet general enough to include both numeric and alphabetic data.
- (2) The single data combination mechanism, *lists*, can express an arbitrarily complex grouping of data.
- (3) A list can change dynamically.
- (4) The data representation is completely independent of the organization of underlying memory.

Even with a flexible data representation, however, a procedural style language still models a system as performing a sequence of steps, relying on global storage for combining modules. As the difficulty with data representation was solved by choosing a different high level model for data, the difficulty with combining modules can be solved by choosing a different high level model of digital systems.

Instead of describing a sequence of steps in an algorithm, each component in a digital system can be considered to be a (*mathematical*) *function*, i.e., a one-to-one mapping on the inputs to the outputs. A binary half adder, for example, performs a function on bits, which can be clearly defined by the equations

$$\begin{aligned}\text{sum} &= x \text{ xor } y \\ \text{carry} &= x \text{ and } y\end{aligned}$$

or by the table:

x	y	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

A microprocessor system also implements a function: its input and output are the contents of memory and internal registers. Unlike the half adder, the mapping from inputs to outputs has no closed form or tabular description. Instead, its operation is defined by a set of smaller functions, called machine instructions, combined sequentially to produce the full function.

With a list representation of data, each module can be represented as a function with exactly one argument, a list, as its input, producing exactly one output, also a list. To eschew using the global state for combining modules, the description language must provide explicit combining forms, so that modules (functions) can be manipulated as well as data. With sufficient combining forms, it is possible to remove the global state entirely for combinational digital systems.



A specification language that models a computation as the application of functions to a data object is a *functional style* language. The emphasis in functional style programming is on the combination of functions, much as the design of digital systems focuses on the combination of modules. In the next chapter, the outline of a functional style language, the UCLA Functional Programming language, is presented. It will be subsequently used as a specification language for digital systems.

## CHAPTER 3

### THE UCLA FUNCTIONAL PROGRAMMING LANGUAGE

#### 3.1. Overview

The UCLA Functional Programming Language follows the language proposed by Backus in [Backus78] as implemented by Baden [Baden82]. This implementation simplifies the original mathematical notation, and expands the set of primitives and functional forms. A functional language is characterized by its objects, syntax and semantics of application, primitive functions, and functional forms.

#### 3.2. Objects

An object is an atom, or a parenthesized sequence of objects. In general, an atom may be a number, quoted ASCII string, or alphanumeric string beginning with a alphabetic character. A sequence consists of *zero* or more objects separated by a blank, surrounded with parenthesis. LISP users will note the similarity of functional objects to LISP objects. The predefined atom `?`, the error atom, is returned by all functions when an error occurs. The following are examples of valid objects:

```
12      "FP string"    xyz
()      (1 2 3)       (abc () (1 2 (3)))
```

Logical values have two representations: *boolean* and *bit*. Boolean values use `()` for *false*; any non-null object is considered to be *true*, but the value will be

represented by the symbol **T**. Bit values, used with functions describing digital algorithms, use the constant values **1** and **0** for true and false, respectively.

### 3.3. Application

The application operator is the colon, ':'. The syntax of application is

$$\textit{expression} : \textit{object}$$

For example, if the function *average* calculates the average of a sequence of numbers, then

$$\textit{average} : (4\ 6\ 2\ 15\ 8)$$

will return the numeric atom 7.

Functional languages are *error preserving*, i.e., for all functions *f*,

$$f : ? \equiv ?,$$

and

$$f : ( \dots ? \dots ) \equiv ?,$$

### 3.4. Primitives

Each primitive is defined in the form

$$\begin{aligned} \textit{primitive} : x \equiv & \text{if } x = \textit{obj}_1 \text{ then } \textit{result}_1 \\ & \text{if } x = \textit{obj}_2 \text{ then } \textit{result}_2 \\ & \dots \\ & \text{if } x = \textit{obj}_n \text{ then } \textit{result}_n \\ & \text{else } ? \end{aligned}$$

where *x* is an arbitrary object, and *obj*<sub>1</sub>, *obj*<sub>2</sub>, ... *obj*<sub>*n*</sub> are expected structures. The first **if** clause to yield **T** will determine the value returned, as described by the **then**

part. If none of the if conditions yield true, then the error atom, `?`, will be returned.

### 3.4.1. Selector Primitives

These primitives are used to access parts of a sequence. The functions *id* and *out* both return the input object *x*, but *out* also prints it on the terminal. The element selector  $\mu$  represents any unsigned integer; the other selector names are intended to be a mnemonic for the function performed.

```
id : x ≡ x  
out : x ≡ x
```

```
 $\mu$  : x ≡ if x = (x1 x2 ... xn) and 1 ≤  $\mu$  ≤ n  
      then x $\mu$   
      else ?
```

```
pick : x ≡ if x = (y (x1 x2 ... xn)) and 1 ≤ y ≤ n  
      then xy  
      else ?
```

```
last : x ≡ if x = () then ()  
      if x = (x1 x2 ... xn) and n ≥ 1  
      then xn  
      else ?
```

```
first : x ≡ if x = () then ()  
      if x = (x1 x2 ... xn) and n ≥ 1  
      then x1  
      else ?
```

```

tail : x ≡ if x = () then ()
          if x = (x1) then ()
          if x = (x1 x2 ⋯ xn) and n ≥ 2
          then (x2 x3 ⋯ xn)
          else ?

```

```

head : x ≡ if x = () then ()
           if x = (x1) then ()
           if x = (x1 x2 ⋯ xn) and n ≥ 2
           then (x1 x2 ⋯ xn-1)
           else ?

```

If it was desired to remove the first and last elements from a sequence, for example, the *head* primitive would be applied to the result of applying the *tail* primitive to the sequence, i.e.

$$\text{head} : (\text{tail} : (1\ 2\ 3\ 4\ 5\ 6)) \equiv \text{head} : (2\ 3\ 4\ 5\ 6) \\ \equiv (2\ 3\ 4\ 5)$$

### 3.4.2. Structural Primitives

The structural primitives change the form of objects. This list does not represent all possibilities, but is chosen as base from which desired manipulations can be constructed. *distl* and *distr* distribute an object from the left or right, respectively, to each element in a sequence. *apndl* and *apndr* append an object on the left or right end of a sequence.

*trans* produces the transpose of a matrix represented as a sequence of one or more equal length sequences. *reverse* reverses the order of objects in a sequence. *rotl* and *rotr* circularly rotate elements of a sequence to the left and right, respectively. *concat* concatenates sequences together to form one sequence; any occurrences of the

null sequence  $()$  disappear. *pair* groups objects of a sequence into subsequences of length 2; *split* separates a sequence into two equal length subsequences.

*distl* :  $x \equiv$  if  $x = (y ())$  then  $()$   
 if  $x = (y (x_1 x_2 \cdots x_n))$  and  $n \geq 1$   
 then  $((y x_1) (y x_2) \cdots (y x_n))$   
 else ?

*distr* :  $x \equiv$  if  $x = (() y)$  then  $()$   
 if  $x = ((x_1 x_2 \cdots x_n) y)$  and  $n \geq 1$   
 then  $((x_1 y) (x_2 y) \cdots (x_n y))$   
 else ?

*apndl* :  $x \equiv$  if  $x = (y ())$  then  $(y)$   
 if  $x = (y (x_1 x_2 \cdots x_n))$  and  $n \geq 1$   
 then  $(y x_1 x_2 \cdots x_n)$   
 else ?

*apndr* :  $x \equiv$  if  $x = (() y)$  then  $(y)$   
 if  $x = ((x_1 x_2 \cdots x_n) y)$  and  $n \geq 1$   
 then  $(x_1 x_2 \cdots x_n y)$   
 else ?

*trans* :  $x \equiv$  if  $x = ((x_1 \cdots x_n)(y_1 \cdots y_n) \cdots (z_1 \cdots z_n))$   
 then  $((x_1 y_1 \cdots z_1) \cdots (x_n y_n \cdots z_n))$   
 else ?

*reverse* :  $x \equiv$  if  $x = ()$  then  $()$   
 if  $x = (x_1 x_2 \cdots x_n)$  and  $n \geq 1$   
 then  $(x_n x_{n-1} \cdots x_1)$   
 else ?

*rotl* :  $x \equiv$  if  $x = ()$  then  $()$   
 if  $x = (x_1)$  then  $(x_1)$   
 if  $x = (x_1 x_2 \cdots x_n)$  and  $n \geq 2$   
 then  $(x_2 \cdots x_n x_1)$   
 else ?

```

rotr :  $x \equiv$  if  $x = ()$  then  $()$ 
           if  $x = (x_1)$  then  $(x_1)$ 
           if  $x = (x_1 x_2 \cdots x_n)$  and  $n \geq 2$ 
           then  $(x_n x_1 \cdots x_{n-1})$ 
           else ?

```

```

concat :  $x \equiv$  if  $x = ((x_1 \cdots x_k) \cdots (y_1 \cdots y_m))$   $k, \cdots, m \geq 0$ 
           then  $(x_1 \cdots x_k \cdots y_1 \cdots y_m)$ 
           else ?

```

```

pair :  $x \equiv$  if  $x = (x_1 x_2 \cdots x_n)$  and  $n$  is even
           then  $((x_1 x_2) \cdots (x_{n-1} x_n))$ 
           if  $x = (x_1 x_2 \cdots x_n)$  and  $n$  is odd
           then  $((x_1 x_2) \cdots (x_n))$ 
           else ?

```

```

split :  $x \equiv$  if  $x = (x_1)$  then  $((x_1) ())$ 
           if  $x = (x_1 x_2 \cdots x_n)$  and  $n \geq 2$ 
           then  $((x_1 \cdots x_{\lfloor n/2 \rfloor}) (x_{\lfloor n/2 \rfloor + 1} \cdots x_n))$ 
           else ?

```

In a "perfect" shuffle of  $k$  equal length sequences, the  $n^{\text{th}}$  element of the  $j^{\text{th}}$  sequence becomes the  $(k(n-1)+j)^{\text{th}}$  element of the final sequence. This can be accomplished by combining (*concatenating*) the sequences of all the first elements, all the second elements, etc, formed by *transposing* the input object.

```

concat : (trans : ((1 3 5 7)(2 4 6 8)))  $\equiv$ 
           concat : ((1 2)(3 4)(5 6)(7 8))  $\equiv$ 
           (1 2 3 4 5 6 7 8)

```

### 3.4.3. Predicate and Length Primitives

Predicates return a boolean value (T or ()) based on the input object. There are currently three predicates implemented: *null*, *atom*, and *eq*. *null* and *atom* are

described below; *eq* compares two objects, returning T if they are identical, () otherwise. *length* returns the integer value of the length of the input sequence.

*atom* :  $x \equiv$  if  $x$  is an atom then T  
           if  $x \neq ?$  then ()  
           else ?

*null* :  $x \equiv$  if  $x = ()$  then T  
           if  $x \neq ?$  then ()  
           else ?

*length* :  $x \equiv$  if  $x = ()$  then 0  
           if  $x = (x_1 x_2 \dots x_n)$  then  $n$   
           else ?

#### 3.4.4. Arithmetic and Logical Operations

UCLA FP uses a full complement of arithmetic and logical primitives, written in either *prefix* or *infix* notation. In prefix, the operation is applied directly to a sequence, e.g.

+ : (1 1)

will yield 2 as a result. For addition, the sequence must consist of exactly two numbers, or an error will result. When equations are complex, however, the prefix format becomes almost unreadable and difficult to debug. The use of infix notation improves the readability and simplifies the formatting of lengthy expressions. For example, the expression

-@ [+@ [1,2], 3] : (2 3 5)

would be rewritten as



$$1 + 2 - 3 : (2 3 5)$$

producing the result 0, as expected.

Using infix notation permits an interpreter or compiler to check for the correct number of arguments when the expression is parsed, saving on run time checking and debugging in general. The infix functions have a normal priority hierarchy among themselves (multiplication and division before addition and subtraction, etc.), and are higher priority than prefix operations. Parenthesis are used to alter the normal evaluation order.

The arithmetic and logical primitives available are summarized below. Arithmetic functions require a sequence of two numbers if prefix, a left and a right expression if infix. Logical functions, except for `~` (not), require a sequence of two logical values, or a left and right logical expression.

Mathematical	+	-	*	/		
Numeric Comparison	<	>	<=	>=	=	≠
Logical	and	or	~ (not)	xor	nand	nor

### 3.4.5. Constants

A constant can be viewed as a function whose output is the same regardless of the input. In UCLA FP, constants are denoted by a percent sign, '`%`', followed by an FP object. If  $x$  and  $y$  are any objects other than `?`, then

$$\%x : y \equiv x$$

### 3.5. Functional Forms

Functional forms serve as the glue for combining functions into more complex expressions. Formally, a functional form is a *metafunction*: it takes one or more functions as its arguments and returns a function whose operation is defined as some combination of the argument functions.

*Composition* combines two functions sequentially. *Construction* and *mapping* apply one or more functions in parallel to the same object or to each object in a sequence, respectively. *Apply-to-all* applies one function to all objects in a sequence in parallel. The four *insert* functional forms, left, right, associative and tree, each reduce a sequence of one or more objects to a single object by repeated application of a function. A *conditional* applies one of two functions, determined by the result of a decision function.

#### 3.5.1. Composition

The composition functional form, '@', is the mathematical function composition operator. Composing two functions combines them sequentially, so that

$$function_1 @ function_2 : x$$

means

$$function_1 : ( function_2 : x )$$

The previous examples of removing the first and last elements from a sequence and the perfect shuffle both represent the sequential combination of two functions. Thus, the applications would be written using the composition functional

form as follows:

$$\text{head @ tail} : (1\ 2\ 3\ 4\ 5\ 6) \equiv (2\ 3\ 4\ 5)$$

and

$$\text{concat @ trans} : ((1\ 3\ 5\ 7)(2\ 4\ 6\ 8)) \equiv (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8)$$

### 3.5.2. Construction

Construction is the parallel application of *one* or more FP expressions to the same object. The null construction  $[]$  is equivalent to the null atom  $()$ , but should not be used, since the constant  $\%()$  is much clearer. The syntax of construction is

$$[\text{expr}_1, \text{expr}_2, \dots, \text{expr}_n] : x$$

yielding the object

$$(\text{expr}_1 : x, \text{expr}_2 : x, \dots, \text{expr}_n : x)$$

so, the result of a construction will be a sequence of N objects. Each FP expression in a construction is separated from any surrounding expressions by a comma ','.

The sum, difference, product and quotient of two numbers can be found by applying the construction of addition, subtraction, multiplication and division to the sequence:

$$[+, -, *, /] : (4\ 2) \equiv (6\ 2\ 8\ 2)$$

### 3.5.3. Mapping

A possible usage for construction is the distribution of a different FP expression to each element of a sequence, i.e.,

$[expr_1@1, expr_2@2, \dots expr_n@n] : (x_1 x_2 \dots x_n)$

This is common enough that UCLA FP uses the functional form *mapping* as a cleaner shorthand. The syntax of a mapping is

$\{expr_1, expr_2, \dots expr_n\} : (x_1 x_2 \dots x_n)$

yielding the object

$(expr_1 : x_1, expr_2 : x_2, \dots expr_n : x_n)$

As with construction, each FP expression is separated from any surrounding expressions by a comma ','. The number of *exprs* in the mapping must match the length of the sequence; if not, the result is ?. The identity function *id* can be used as a place holder.

#### 3.5.4. Apply to All

Where construction and mapping distribute one object to several functions, the apply-to-all functional form '&' distributes one function to several objects. Apply to all is defined as:

$\&expr : x \equiv$  if  $x = ()$  then  $()$   
if  $x = (x_1 x_2 \dots x_n)$  and  $n \geq 1$   
then  $(expr : x_1, expr : x_2, \dots expr : x_n)$   
else ?

The *expr* can be a single FP function, a parenthesized FP expression, a construction or mapping, or an applied-to-all or inserted expression.

To produce a truth table of the form

$((bit\_pattern_0 result_0) \dots (bit\_pattern_n result_n))$

for a boolean function  $f$ , the construction of the identity function  $id$  and  $f$  would be applied to all bit patterns. The truth table for **and**, for example, would be produced by the application

**&** [*id*, **and**] : ((0 0)(0 1)(1 0)(1 1))

giving

(((0 0) 0) ((0 1) 0) ((1 0) 0) ((1 1) 1))

### 3.5.5. Inserts

When it is desired to operate on all the elements of a sequence, such as taking the sum of a sequence of numbers, one of the inserts is used. There are four types of inserts: right ('!'), left ('\'), associative ('|'), and tree ('^').

The associative insert repeatedly splits a sequence and does a recursive associative insert on both halves until the length of a sequence is two, then applies the function. A tree insert repeatedly performs an operation pairwise on a list until a single result is obtained; this is distinct from the associative insert for all sequences of length greater than five.

```
!expr : x ≡ if x = (x1) then x1
           if x = (x1 x2 ··· xn) and n ≥ 2
           then expr : (x1 !expr : (x2 ··· xn))
           else ?
```

```
\expr : x ≡ if x = (x1) then x1
           if x = (x1 x2 ··· xn) and n ≥ 2
           then expr : (\expr : (x1 ··· xn-1) xn)
           else ?
```

$|expr : x \equiv$  if  $x = (x_1)$  then  $x_1$   
 if  $x = (x_1 x_2 \cdots x_n)$  and  $n \geq 2$   
 then  $expr : (|expr : (x_1 \cdots x_{\lfloor \frac{n}{2} \rfloor}) |expr : (x_{\lfloor \frac{n}{2} \rfloor + 1} \cdots x_n))$   
 else ?

$\hat{expr} : x \equiv$  if  $x = (x_1)$  then  $x_1$   
 if  $x = (x_1 x_2)$  then  $expr : x$   
 if  $x = (x_1 x_2 \cdots x_n)$  and  $n \geq 2$   
 then  $\hat{expr} @ \&expr @ pair : (x_1 x_2 \cdots x_n)$   
 else ?

The factorial of an integer  $n$  can be computed by inserting multiplication into the sequence of integers from 1 to  $n$ . Since multiplication is both associative and commutative, the result will be the same regardless of the particular insert used. The right insert works from the left to the right of a sequence until one element remains, then performs the multiplication from right to left:

$!* : (1 2 3 4) \equiv * : (1 !* : (2 3 4))$   
 $\equiv * : (1 * : (2 !* : (3 4)))$   
 $\equiv * : (1 * : (2 * : (3 !* : (4))))$   
 $\equiv * : (1 * : (2 * : (3 4)))$   
 $\equiv * : (1 * : (2 12))$   
 $\equiv * : (1 24)$   
 $\equiv 24$

The tree insert, in contrast, performs the operation on pairs of elements until a single result remains:

$\hat{*} : (1 2 3 4) \equiv \hat{*} @ \&* @ pair : (1 2 3 4)$   
 $\equiv \hat{*} @ \&* : ((1 2)(3 4))$   
 $\equiv \hat{*} : (2 12)$   
 $\equiv 24$

### 3.5.6. Conditional

The conditional functional form in UCLA FP is syntactically similar to the Algol68 if statement. This format improves the readability of FP programs. To preserve functionality, there must always be an else expression. The conditional functional form is defined as

$$\text{if } expr_C \text{ then } expr_T \text{ else } expr_F \text{ fi } : x \equiv \begin{cases} expr_T : x & expr_C : x \neq () \\ expr_F : x & expr_C : x = () \\ ? & otherwise \end{cases}$$

### 3.6. User Defined Macros

The functional style of programming is the combination of primitive functions to perform larger, more powerful computations. There are no function programs in the sense of a sequence of statements operating on a underlying storage, as in FORTRAN or Pascal. It is convenient, however, to keep a record of previous combinations around for use as building blocks. These combinations are called *user defined function macros* (macros for short), equating a given user name with a functional expression.

#### 3.6.1. Macro Definition

The syntax for defining a function macro is

**define macro name (object equates) FP expression end**

Like the primitive functions, user defined macros can appear in prefix or infix form. Macros have the lowest precedence, and group left to right.

The *object equates* is a list of zero or more alphanumeric strings separated by commas. Each string must begin with an alphabetic character and should not be the name of a primitive or user defined macro. Certain macros expect to operate on a sequence of a specific length; using  $\mu$  (selector) primitives to access the individual objects quickly becomes unreadable and difficult to understand.

The object equates allow the user to give a mnemonic name to each object in an input sequence. This not only improves readability, but also gives extra error checking by requiring a proper length sequence. If there are no equates, the macro will accept any object; but, the parenthesis must appear anyway.

Equates are *not* parameters or variables, and should not be treated as such. They are only a means of replacing  $\mu$  selectors with more meaningful symbols. Generally, equates are not even valid after the first level of composition, since the structure of the object will likely have been altered.

A function to find the product of all integers between a lower bound and an upper bound can be recursively stated as follows:

```
if      lower bound  $\equiv$  upper bound
then    the result is the lower (or upper) bound
else    multiply the upper bound by the product of integers from
        the lower bound to one less the upper bound
```



The macro for this function can be easily defined using the conditional functional form:

```
define  product(lower, upper)
  if    lower = upper
  then  upper
  else  upper * (lower product (upper - %1))
  fi
end
```

### 3.6.2. Macro Execution Tracing Facility

The execution of user defined macros may be traced to debug functional expressions or to examine the structure of the computation dynamically. Each time a specified macro is entered, i.e, evaluated, the interpreter prints out a message stating that a macro has been entered, the name of the macro, and the input object. When the macro is exited, i.e, evaluation is complete, the macro name is printed again, along with the output object.

A function is set to be traced using the special interpreter command `)trace`. If the macro *product*, defined above, was traced, then the application

```
product : (1 3)
```

would display

ENTER product : (1 3)  
ENTER product : (1 2)  
ENTER product : (1 1)  
EXIT product : 1  
EXIT product : 2  
EXIT product : 6

6

### 3.7. Symbolic Execution

In addition to performing arithmetic and boolean operations on numeric and logical values, the UCLA FP interpreter will also operate on *symbolic* values, or a combination of symbolic and numeric/logical operands. The result of a function application is determined as follows:

- (1) If both operands are numeric (logical), perform the function:

$$+ : (1 1) \equiv 2$$

- (2) If both operands are symbolic, generate a new, unique symbol based on the function name:

$$\text{and} : (a b) \equiv \text{AND1}$$

In this example, the new symbol is the name of the function concatenated with a number to make it unique.

- (3) If one operand is symbolic and the other constant (numeric or logical), generate a new symbol unless there is an identity rule for the function involving the constant operand. That is,

$$* : (a 3) \equiv \text{MUL5}$$

but,

$$* : (a \ 1) \equiv a$$

and

$$* : (a \ 0) \equiv 0$$

since  $a * 1 \equiv a$  and  $a * 0 \equiv 0$  for all  $a$ , except the error atom ?.

Logical functions are further extended to work on a combination of bit, boolean, and symbolic values. The input-output results are summarized in figure 3.1 below. Functional forms are unaffected by the use of symbolic values except the conditional functional form: the conditional expression must generate a boolean or bit value.

Input Types	Boolean	Bit	Symbol
Boolean	Boolean	Bit	Symbol*
Bit	Bit	Bit	Symbol*
Symbol	Symbol*	Symbol*	Symbol

Symbol\* = Symbol based on identity for function  
(e.g.  $x \text{ and } 1 \equiv x$ ,  $y \text{ xor } 1 \equiv \bar{y}$ )

Figure 3.1 - Symbolic Logical Operation Result Matrix

## CHAPTER 4

### FUNCTIONAL NET LIST DESCRIPTION AND GENERATION

#### 4.1. Overview

A functional style programming language fulfills many of the goals for net list specification. The data model does not restrict either the types of data or what combination of values a module can produce. Functional forms provide a general, explicit means of putting modules together into a network, and can clearly express the two dimensional structure of a system. Since the emphasis of functional programming is the combination of simpler functions, it encourages hierarchic design and development. The language itself is flexible: a solution can be built up from powerful primitives, instead of trying to fit it into the language framework.

In this chapter, two complementary techniques for describing net lists are examined. The first approach uses FP to describe a system by capturing its structure using functional forms. The second uses the ability of FP to manipulate functions to describe the structure of a group of net lists. By tracing the symbolic execution of an FP description, the specified net list can be extracted for both methods.

#### 4.2. "Picture" Style Net List Description

One approach to net list description is to use functional forms to reflect the structure of the system in a functional expression. Composition denotes consecutive

levels of the system; modules that execute in parallel are grouped by the construction or mapping functional forms. Structural primitives, such as selectors, serve to make connections between levels where routing is necessary. This method is called *picture style description*, since the functional specification captures a picture of the placement and relationship of modules in the system.

In Chapter 2, a procedural description of a full adder and its components was developed; the equivalent functional description will be developed here. The basic logic gates (*and*, *or*, and *xor*) each operate on a sequence of two logical values. Though these functions exist as FP primitives, macros will be defined for each to distinguish them as logic gates.

```
define andgate() and end
```

```
define orgate() or end
```

```
define xorgate() xor end
```

The half adder is easily described as an *and* gate and an *exclusive or* gate applied to both inputs in parallel, i.e., constructed. The macro definition is written with two parameter equates to convey that the macro expects exactly two inputs and to indicate more clearly that the gates operate on the two inputs.

```
define   half_add(x, y)  
          [ x andgate y, x xorgate y ]  
end
```

The correspondence between the macro definition and the actual circuit is represented in figure 4.1 below.

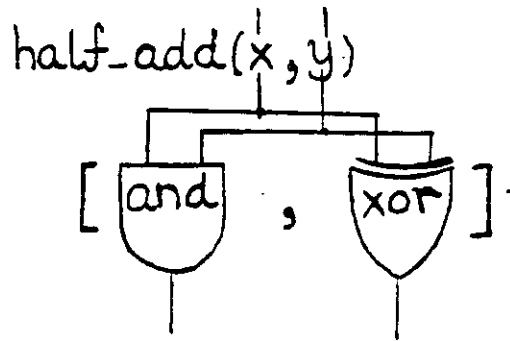


Figure 4.1 - A half adder circuit and FP description

The full adder description is more complex than the previous modules. While the logic gates and the half adder each have one level between input and output, the full adder has three levels, and must route the outputs of one level to the proper inputs of the next. The macro definition uses parameter equates: this tells a designer what input is expected (a sequence of three objects) and gives more meaningful names to each object. Using the half adder and the or gate, a binary full adder can be described as

```

define  full_add(x, y, Cin)
        [1 orgate 2, 3]      @
        apnd                 @
        [1, 2 half_add 3]   @
        apndr                @
        [x half_add y, Cin]
end

```

At first glance, this description seems more complicated than the Pascal equivalent. The functional description, however, explicitly shows the structure of the system, as depicted in figure 4.2 below. Modules, their inputs and their outputs are

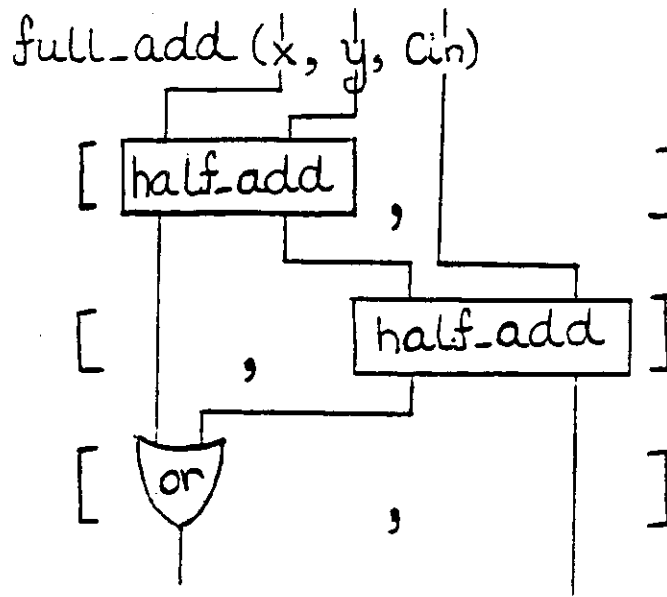


Figure 4.2 - A full adder circuit and FP description

clearly specified. Each level of the computation is distinctly marked by the composition functional form. Though the selectors are initially confusing, they provide an unambiguous map of how data moves from one level to the next. Since the elements of a full adder (half adders and the orgate) the description can use the macro names in infix format. Most important, there are no side effects to interfere with understanding the description.

*apndr* and *apndl* are used to combine the sequence of two bits produced by the half adder with the single bit carried through that stage of the system. This permits the use of single selector primitives at each level. The function could be rewritten without them as follows:

```

define  full_add(x, y, Cin)
        [1 orgate (1 @ 2), 2 @ 2]      @
        [1 @ 1, (2 @ 1) half_add 2]   @
        [x half_add y, Cin]
end

```

but with a slight loss in clarity.

Picture style net list descriptions combine modules by functional forms to reflect the structure of the system. Parallelism, computational levels, and input-output relationships are all clearly mapped out. Macro definitions make hierarchical specification simple and natural, and can serve as the basis for libraries of systems.

#### 4.3. "Generator" Style Net List Description

Because a picture style description accurately captures the structure of a digital system, the specification represents a fixed pattern of net list elements. For certain systems, this is acceptable: a full adder, for example, only operates on three bits and always generates two bits. For other types of systems, the ability of FP to manipulate functions *and* data can be employed to create a generic system specification. In contrast to the picture style, the *generator* style description specifies *how* a network is to be constructed given parameters of the system, rather than the structure of the system itself.

One general type of system is binary adders; the simplest of the adders is the n-bit ripple carry adder, made up of n full adders. Each full adder takes one bit pair and the carry from the previous full adder (or the initial carry in the first stage), producing a sum bit and the next carry. The algorithm for building an n-bit ripple carry



adder can be stated as a simple recursion:

```
if      n = 1
then   apply a full adder to the bit pair and the initial carry,
else   build a ripple carry adder for the n-1 previous bit pairs,
       then apply a full adder to the current bit pair and the carry
       from the previous n-1 bits.
```

This readily translates into a functional description using the **if-then-else** functional form. The macro will take two arguments: a non-empty vector of bit pairs *bitpr* and an initial carry *Cin*. The output will be a vector made up of the carry bit and the sum bits.

```
define  rc_adder(bitpr, Cin)
if      (length @ bitpr) = %0
then   full_add @
       apndr @ [1 @ bitpr, Cin]
else   concat @
       [full_add @ apndr @ [1, 1 @ 2], tl @ 2] @
       [1 @ bitpr, (rc_adder @ [tl @ bitpr, Cin])]
fi
end
```

The description of the ripple carry adder is completed by a macro to take two vectors of bits and an initial carry and transform it into a vector of bit pairs and a carry, then apply *rc\_adder*:

```
define  ripple_carry_add(x, y, Cin)
       rc_adder @
       [trans @ {x, y}, Cin]
end
```

Unlike the picture style, a generator style description does not reflect the explicit structure of the system. What is lost in clarity, however, is gained in the flexibility to describe a general class of net lists. Any system with a repetitive structure, such as adders, multiplexers, decoders, or ALU's, can use one description regardless of how many times the basic modules are repeated.

Both styles maintain the advantage of the functional style: freedom from side effects, flexible data representation, and powerful combining forms. Since the two approaches address different specification requirements, a system description, such as the ripple carry adder, would use both, each where appropriate to the solution of the problem.

#### **4.4. Tracing and Net List Extraction**

Many types of digital systems can be described using the picture and generator styles. Besides providing a system specification, the functional representation can be executed by an interpreter to verify its correctness. It is still necessary, however, to extract the net list from the specification. A net list is a more readily usable input form for design and analysis tools, since the FP description would require each program to act as an interpreter.

Further, there are many applications requiring a two pass interpretation in order to be performed in the functional domain. Consider a system where the output of an and gate is distributed to three other modules:

$[mod_1, mod_2, mod_3]$  @ andgate

If this system were implemented in NMOS technology, the delay of the and gate would be approximately three times greater than if its output only went to one module [Mead80]. The fan-out of an output, however, *cannot* be determined when it is generated during interpretation; therefore, a preliminary pass must extract this information before accurate timing analysis can be done. Since this information is already present in the net list, it is more general to orient analysis tools to work on net lists instead of the functional description that produced them.

Since the net list specified by both picture and generator style descriptions is determined by which functions are applied to what objects, it can readily be extracted by watching when each function is applied, and what are the input and output objects. This can be accomplished through the macro *tracing* facility of FP. If the execution is *symbolic*, as discussed in the previous chapter, the interpreter will automatically generate unique labels for the input-output relationships of the system.

Using the functional definitions for a full adder circuit, tracing the *full\_add* macro applied to the symbolic values a, b, and C will yield

```
full_add : (a b C)
```

```
ENTER full_add : (a b C)
```

```
EXIT full_add : (OR_1 XOR_2)
```

```
(OR_1 XOR_2)
```

Hierarchy levels in the net list can either be hidden or displayed by the choice of which macros are traced. The connectivity of logic gates in the full adder would be derived by tracing the macros for the and, xor and or gates, so that the same application would yield

```
full_add : (a b C)

ENTER full_add : (a b C)
ENTER andgate : (a b)
EXIT andgate : AND_1
ENTER xorgate : (a b)
EXIT xorgate : XOR_1
ENTER andgate : (XOR_2 C)
EXIT andgate : AND_2
ENTER xorgate : (XOR_2 C)
EXIT xorgate : XOR_2
ENTER orgate : (AND_1 AND_2)
EXIT orgate : OR_1
EXIT full_add : (OR_1 XOR_2)

(OR_1 XOR_2)
```

The transformation from a symbolic trace to a net list is a simple, mechanical process. Each 'ENTER'-'EXIT' pair corresponds to one element in the node list; any pair appearing in-between another 'ENTER'-'EXIT' represents an element of a lower level network. Each pair is rewritten in the trace with the module name followed by the input and output object, without parenthesis. If all macros are traced, the net list derived from the full adder description is

```

full_add    a, b, C      :: OR_1, XOR_2
half_add    a, b        :: AND_1, XOR_1
andgate     a, b        :: AND_1
xorgate     a, b        :: XOR_1
half_add    XOR_1, C    :: AND_2, XOR_2
andgate     XOR_1, C    :: AND_2
xorgate     XOR_1, C    :: XOR_2
or          AND_1, AND_2 :: OR_1

```

which is the form of the net list for a full adder, as shown in Chapter 1.

When a net list is extracted from a trace, structural primitives and functional forms disappear. The information they represent is captured in the connectivity of the system. For a generator style description, however, the *if-then-else* functional form disappears completely. It is not part of the static structure because it is used to dynamically build the net list. For example, the arrangement of full adders in a two bit binary ripple adder can be extracted by tracing the *full\_add* macro and applying *ripple\_carry\_add* to two vectors of two bits and the initial carry.

```

ripple_carry_add : ((a1 a0) (b1 b0) Cin)

ENTER ripple_carry_add : ((a1 a0) (b1 b0) Cin)
ENTER full_add       : (a0 b0 Cin)
EXIT full_add        : (OR_1 XOR_2)
ENTER full_add       : (a1 b1 OR_1)
EXIT full_add        : (OR_2 XOR_4)
EXIT ripple_carry_add : (OR_2 XOR_4 XOR_2)

(OR_2 XOR_4 XOR_2)

```

which yields the net list.

```
ripple_carry_add  a1, a0, b1, b0, Cin  ::  OR_2, XOR_4, XOR_2
full_add          a0, b0, Cin          ::  OR_1, XOR_2
full_add          a1, b1, OR_1         ::  OR_2, XOR_4
```

The symbolic trace, or the extracted net list, can serve as a common input for a large collection of analysis tools. Since net lists make no assumptions about physical realization, each analysis program can be written to solve a specific problem. A functional specification, then, can meet the requirements of both high level and low level design of digital systems.

## CHAPTER 5

### FUNCTIONAL NET LIST DESCRIPTION IN VLSI DESIGN

#### 5.1. Overview

A VLSI design passes through four phases: algorithmic, logical, electrical, and physical. In general, the amount of information it takes to describe the design increases as the design moves toward realization in silicon. Further, differing requirements of each step lead to vastly different description techniques. Translating from one step to the next is difficult; skipping steps and/or going backwards even more so.

In this chapter, functional style description is examined to see how well it can serve as a multilevel specification. In order to serve all levels, technology specific *translation programs* must be introduced into the cycle. These can be hidden from the designer, however, so that a functional description can close the feedback loop in the design cycle. Finally, a design environment organized around a functional style description is presented and examined through the generation of VLSI circuits for the ripple carry adder described in the previous chapter.

#### 5.2. The Need for a Multilevel Systems Language

From conception to production, the design of a VLSI digital system passes through four (4) phases:

- (1) Development of one or more digital algorithms to solve the problem, often using high level data representations and operations.
- (2) Translation of an algorithm into small, medium and large scale logic blocks, e.g., gates, decoders and programmed logic arrays, respectively.
- (3) Implementation of logic building blocks in technology dependent electrical components.
- (4) Realization of electrical components in the physical features of an integrated circuit.

At each step, the *design information* represents the basic components for the particular level of design. As the design moves toward fabrication, the volume of information swells dramatically as details, hidden from or irrelevant to higher levels, are exposed.

Design information is managed both within and between steps. During the design process of each step, the components of the design are coded in an understood format, which is then used to test and evaluate the design, as well as serving as a medium for exchange of information. A digital algorithm, for example, might be described in a register transfer language; the electrical realization could be described using SPICE for simulation purposes.

Between phases, however, the information changes: the components and interconnections differ, as well as the format suitable for the phase. While the register transfer description and a SPICE description may represent the same circuit, neither is



useful as a specification in another step.

The effect that is the overall design process becomes a Tower of Babel. The specialization of description languages for the four phases of design necessitates a series of complex translations when the design moves from one step to the next. Translations are possible, but often difficult, since the information content is increasing. Skipping a step is even more difficult; it can be impossible if the lower step depends on information generated in the missing step.

It is most difficult, however, to go back up the chain, since a reverse translation would be a one-to-many mapping. Further, such changes are normally to details that cannot be directly reflected in the higher levels. Changing a logic specification because it cannot be realized in a limited area of silicon, for example, would effectively require a complete re-design of the system.

A more practical method to handle design information would be to use a system specification that can be used in all phases of the design. By changing a single, central specification, each step in the design process can run in parallel instead of sequentially. This allows rapid design evaluation at each level, rather than waiting for changes in the higher levels to propagate through the translation process. Working from a single specification insures that errors in one phase do not propagate to others. Each design step is relatively independent, but can still coordinate changes to the system when necessary.

### 5.3. Functional Description in the Design Cycle

A functional language is a good candidate for single multilevel specification. Since the functional style naturally builds from primitives to more complex systems, it can equally well represent both high and low level views of a system. Hierarchy in the description can be used to manage which level(s) are visible at each step.

The usefulness of a functional description, however, is limited where the system evaluation involves more than the pure functionality of the system. Electrical parameters, such as voltages and parasitic capacitances, or layout characteristics, such as area and system topology, can only be reflected in the functional description if some macro has the information explicitly coded; each set of parameters would require a different macro. This forces a designer to make a low level, technology dependent decision at an inappropriate level of design.

A functional language specification, then, best serves to describe the high level view of the system. The gap to the lower levels of design is bridged by translating a hierarchical net list, representing the application of a functional description to a specific object, into a technology and application dependent form. Each translation program makes decisions based on its specific needs: selecting transistor models, approximating circuit parasitics, choosing from a standard cell library, and so on.

With the "smarts" of translation embedded in separate programs, the functional description remains technology independent. The translation is more rapid than hand design; given that the translation program is correct, the result is guaranteed to

conform to the functional specification. Further, the translation program can consider the system as a whole and possibly employ design “tricks” a human designer would miss or simply not look for.

A functional description can, then, serve as the multilevel specification language for each step in the design cycle. The functional description itself is used directly in algorithm development and logical testing. Using net lists extracted from the functional specification, translation programs can generate the information needed to evaluate the design at the electrical and physical levels. The path from description to net list to translator can be hidden by appropriate software, so that a designer only sees the functional description at each phase.

Changes and optimizations to the overall design can either be reflected directly to the functional description, or to the net list, which can be reverse compiled into FP. By serving each step in the design process, a functional description closes the feedback loop in the design process.

#### **5.4. A Functional Design Environment**

A design environment for digital systems based on a functional style description might appear as in figure 5.1. Using an interpreter for the functional language, the description can be tested for proper functionality; other parameters of the system description, such as component count or parallelism, can also be evaluated here. Other analysis tools can use the algebraic properties of the functional language to inspect the description directly and perform expression optimization, generate circuit

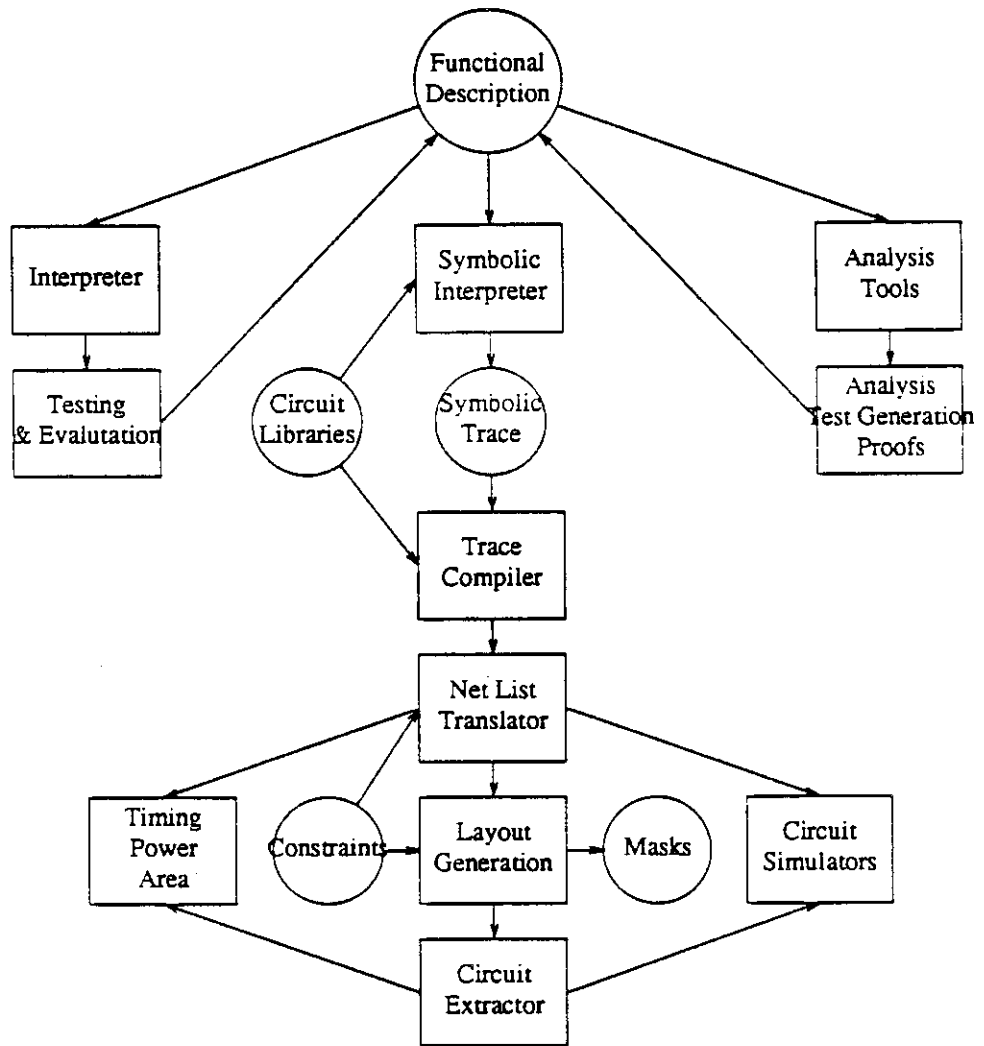


Figure 5.1 - An FP Design Environment

test patterns, or automatically prove characteristics of the description.

The focus of this thesis, however, is the operation of the design environment in the realization of the circuit. In this section, the design environment will be inspected while generating the layout for a four-bit ripple carry adder, using the functional description developed above. The translation program used will translate from the output of symbolic execution into BDL [Slutz84], a connection oriented, hierarchical digital system description language. A quick summary of BDL is presented below.

#### 5.4.1. The BDL System Description Language

BDL (for Block Description Language) is a structural description language for capturing the structure of a digital system, and serves as input to VLSI design tools, e.g., placement and routing or timing analysis. In the BDL model, a circuit is made up of one or more interconnected blocks; each block, in turn, can be composed of one or more sub-blocks. Blocks are interconnected by *signals*, which can be a *net* (one signal), or grouped into *buses* (analogous to Pascal arrays) or *bundles* (analogous to Pascal records).

A block description is composed of up to six sections:

- (1) A block declaration of the form **BLOCK** *name*. This declaration can also list a hierarchical **LEVEL**, or a list of *properties*. Properties are extra information, usually specific to the end use of the BDL description.

- (2) A list of signals called **PORTS**, which are visible outside the block and are used to make connections to other blocks. Ports may also have properties.
- (3) A list of signals called **SIGNALS**, which are all signals internal to this block. Signals may also have properties.
- (4) A list of all sub-block **INSTANCES** that appear in this block. Any block in the **INSTANCES** list must have been declared before the current block. Instances may also have properties.
- (5) A **SIGNALLIST**, listing each signal along with the current block (referenced by the name **THISBLOCK**) and/or sub-block **PORTS** the signal connects.
- (6) An **INSTANCELIST**, listing **THISBLOCK** and each sub-block along with the signals their **PORTS** connect to. Since the **SIGNALLIST** and **INSTANCELIST** contain exactly the same information, only one need appear in a block description.

If the block is primitive, i.e. contains no sub-blocks, only the **PORTS** declaration need appear; otherwise, **PORTS**, **SIGNALS**, and **INSTANCES** must appear along with at least one of **SIGNALLIST** or **INSTANCELIST**.

#### 5.4.2. Translation Considerations

Depending on the characteristics of the target language, there are several issues that a net list translation program must manage. Foremost is the mapping from a hierarchical net list to the final desired output. Translations fall into roughly three categories:

- (1) Only the primitive, or lowest level, modules in the hierarchy are used by the translator. Each primitive is mapped into one or more description components, so that the net list primitives act as macros for the target language. A primary example are switch level and component level circuit simulators, where a basic logic gate may be mapped into a combination of transistors, resistors, capacitors and wires.
- (2) Each element in the net list maps into one description component. If the target language does not have hierarchical constructs, only the primitive elements are used by the translator. This is the case for most connection oriented languages.
- (3) Several elements in the net list may be combined into one description component. A target language that uses arithmetic expressions, for example, might combine logic gates from several levels of the net list into a single expression. This type of translation "reverse interprets" the net list, and is therefore more difficult than the first two types. The target is probably an algorithm oriented, high level description language.

Since BDL is a connection oriented, hierarchical language, the translation from a net list is reasonably straightforward. Elements in the net list map neatly into blocks, with inputs and outputs becoming the ports and any subelements becoming sub-blocks. Predefined blocks can be stored in libraries to handle primitive modules. This has the advantage that the same net list can be translated towards several applications merely by specifying different libraries.

Symbolic execution guarantees that output symbols are unique, so that signal names are provided by the net list. Though BDL provides compound signal types (*busses* and *bundles*), assigning these types to a group of signals would require that the entire net list be analyzed to find which signals appear together everywhere. It is therefore simpler and more general to make each signal of type *net*.

To exploit existing libraries and the hierarchy available in BDL the translation occurs in two phases:

- (1) Each element of the net list is examined to find modules that exist in any libraries in use. If an element is found, the block that element represents is considered defined, and will not be defined again.
- (2) The net list is searched in a post-order traversal (i.e., subcomponents are examined first) to find undefined elements. The first time an element is encountered, a block is defined for that element using the input and output objects for ports, any sub-elements to define instances and signals. As with library blocks, each element is defined exactly once.

The effect of hierarchy in the BDL description on the resultant layout will be examined in the next section. For this purpose, the original BDL translator was slightly modified to be able to produce a flat description, i.e., one top level block consisting entirely of library cells, as well as the hierarchical block description.



### 5.4.3. Sample Translation: A Full Adder

The full adder description developed in the previous chapter will now be used to generate the BDL description for a full adder. The functions used in the full adder description are summarized in figure 5.2. Notice, however, that the definition of *xorgate* has been changed to reflect the absence of a primitive exclusive-or gate in the BDL library, and is now defined as the 4 nand gate realization. The net list used below was derived by tracing the *full\_add*, *half\_add*, *andgate*, *orgate* and *nandgate* functions.

```
define andgate() and end

define orgate() or end

define nandgate() nand end

define xorgate(x, y)
    nandgate                                     @
    [1 nandgate 2, 2 nandgate 3]               @
    [x, x nandgate y, y]
end

define half_add(x, y)
    [ x andgate y, x xorgate y ]
end

define full_add(x, y, Cin)
    [1 orgate 2, 3]                               @
    apndl                                         @
    [1, 2 half_add 3]                             @
    apndr                                         @
    [x half_add y, Cin]
end
```

Figure 5.2 - Full Adder FP Description

During the first pass of the translator, the basic logic gates, defined in a BDL library, will be defined. These descriptions contain extra information about the size of the cell and the location of inputs and outputs used by the place and route program. Three BDL library cells are found in a full adder: a two input and gate, a two input or gate and a two input nand gate. The BDL library definitions are shown below (figure 5.3) without the application specific information. The input port names are A and B, indicating a two input cell. The output port is named T for True, or positive logic gates; the N output port denotes an inverted logic gate.

In traversing the net list, the half adder description will be the first new block defined. The block has four ports: two inputs, two outputs. The port names for a

```
BLOCK ANDGATE  
LEVEL LIBRARY;  
PORTS  
NET: A, B, T;  
ENDPORTS  
ENDBLOCK
```

```
BLOCK ORGATE;  
LEVEL LIBRARY;  
PORTS  
NET: A, B, T;  
ENDPORTS  
ENDBLOCK
```

```
BLOCK NANDGATE  
LEVEL LIBRARY;  
PORTS  
NET: A, B, N;  
ENDPORTS  
ENDBLOCK
```

Figure 5.3 - BDL Definitions of Basic Logic Gates

block are derived by prepending the string "P\_" to each of the atoms in the input and output objects when the element is first encountered. Since the first half adder element encountered is

*half\_add* a0, b0 :: AND4, NAND12

the input and output port names will be P\_a0, P\_b0 and P\_AND4, P\_NAND12, respectively.

Signal names are collected for the **SIGNALS** declarations by scanning the element input object and the output object for each subelement. Each time a new symbol name is encountered, the name is added to the list of signals, printed out in declaration form when the scanning is complete. There are seven signals in a half adder: two inputs, two outputs, and three signals in the intermediate stages of the exclusive-or logic.

The **INSTANCES** list is easily derived from the subelement list. An integer, generated by a sequentially incremented counter, appended to the element name identifies a particular instance of an element in the circuit. A half adder contains five instances: four **nand** gates and one **and** gate.

The actual connectivity of elements in the block is defined in the **SIGNALLIST** and/or the **INSTANCELIST**. The **SIGNALLIST** uses the list generated for the **SIGNALS** declarations, looking for an occurrence(s) of the signal in the input and/or output objects of the block and its subelements. Since the **INSTANCELIST** represents the same information, it is omitted from the descriptions used in this thesis.

The complete BDL block description for a binary half adder, as translated from an input net list, is shown in figure 5.4 below. Once the block *half\_add* is defined, the translator will generate the BDL description for the full adder, which appears in figure 5.5. The full adder contains two instances of the block *half\_add*, and one or gate.

```
BLOCK half_add;  
LEVEL MACRO;  
  
PORTS  
  NET : P_a0, P_b0;  
  NET : P_AND4, P_NAND12;  
ENDPORTS  
  
SIGNALS  
  NET : a0, b0, AND4, NAND6, NAND8, NAND10, NAND12;  
ENDSIGNALS  
  
INSTANCES  
  ANDGATE : andgate0;  
  NANDGATE : nandgate1, nandgate2, nandgate3, nandgate4;  
ENDINSTANCES  
  
SIGNALLIST  
  a0  : THISBLOCK*P_a0, andgate0*A, nandgate1*A, nandgate2*A;  
  b0  : THISBLOCK*P_b0, andgate0*B, nandgate1*B, nandgate3*B;  
  AND4 : THISBLOCK*P_AND4, andgate0*T;  
  NAND6 : nandgate1*N, nandgate2*B, nandgate3*A;  
  NAND8 : nandgate2*N, nandgate4*A;  
  NAND10 : nandgate3*N, nandgate4*B;  
  NAND12 : THISBLOCK*P_NAND12, nandgate4*N;  
ENDSIGNALLIST  
ENDBLOCK
```

Figure 5.4 - BDL Description of Binary Half Adder

```

BLOCK full_add;
LEVEL MACRO;

PORTS
NET : P_a0, P_b0, P_Cin;
NET : P_OR22, P_NAND21;
ENDPORTS

SIGNALS
NET : a0, b0, Cin, AND4, NAND12, OR13, NAND21, OR22;
ENDSIGNALS

INSTANCES
half_add : half_add5, half_add6;
ORGATE : orgate7;
ENDINSTANCES

SIGNALLIST
a0 : THISBLOCK*P_a0, half_add5*P_a0;
b0 : THISBLOCK*P_b0, half_add5*P_b0;
Cin : THISBLOCK*P_Cin, half_add6*P_b0;
AND4 : half_add5*P_AND4, orgate7*A;
NAND12 : half_add5*P_NAND12, half_add6*P_a0;
OR13 : half_add6*P_AND4, orgate7*B;
NAND21 : THISBLOCK*P_NAND21, half_add6*P_NAND12;
OR22 : THISBLOCK*P_OR22, orgate7*T;
ENDSIGNALLIST
ENDBLOCK

```

Figure 5.5 - BDL Description of Binary Full Adder

#### 5.4.4. A Ripple Carry Adder Layout

One of the tools which uses BDL as input is a standard cell place and route program for integrated circuits. Using a library of cells defined with the necessary properties (e.g., size, port locations), it is possible to derive actual circuit layouts starting from a net list description.

The net list for a ripple carry adder can be extracted from the FP description (figure 5.6) by tracing the basic logic gates (*andgate*, *orgate* and *nandgate*), and the top level macro, *ripple\_carry\_adder*. This would produce a single level trace; the hierarchy in the adder can be extracted by tracing the half and full adder macros as well.

In BDL, a circuit is a block; every BDL description, then, must consist of exactly one block directly or indirectly containing all other block instances. For the ripple carry adder, *ripple\_carry\_adder* is this special *top* level block. The *ripple\_carry\_adder* block description contains, in addition to the normal block information, application specific global information which affects the circuit as a whole. When used to generate a circuit layout, the ports of the top level block are associated with input and output pads. The location of these pads, plus pads for power and

```

define  rc_adder(bitpr, Cin)
  if    (length @ bitpr) = %0
  then  full_add @
        apndr @ [1 @ bitpr, Cin]
  else  concat @
        [full_add @ apndr @ [1, 1 @ 2], tl @ 2] @
        [1 @ bitpr, (rc_adder @ [tl @ bitpr, Cin])
  fi
end

define  ripple_carry_add(x, y, Cin)
  rc_adder @
  [trans @ [x, y], Cin]
end

```

Figure 5.6 - Ripple Carry Adder FP Description

ground, is also specified in the top level block.

For the layouts that follow, the input net list was produced by the symbolic execution of a four-bit ripple carry add

*ripple\_carry\_adder* : ((a3 a2 a1 a0)(b3 b2 b1 b0) Cin)

tracing the *andgate*, *orgate*, *nandgate*, *half\_add*, *full\_add*, and *ripple\_carry\_adder* macros.

The first layout (figure 5.7) shows the top level view of the circuit. The pads labeled P\_a3 . . . P\_a0, P\_b3 . . . P\_b0 and P\_ are the inputs; pads labeled P\_OR79, P\_NAND78, P\_NAND59, P\_NAND40 and P\_NAND21 are the outputs (P\_OR79 is the carry out, P\_NAND21 is the least significant bit). The VDP and VSP are the power and ground pads, respectively. The box labeled *ripple\_carry\_adder* is the actual adder circuitry, shown in detail in figure 5.8.

The logic for the four-bit ripple carry adder is 344.4 by 426.0 microns, the first dimension measured across the row of cells. Logic gates are realized in 3 micron CMOS technology, arranged in four rows. There are 32 nand gates, used to realize the eight exclusive-or gates in the half adders, eight and gates and four or gates. In addition, there are five *feed through* cells, used to make connections through a row.

Connections are routed in the three channels between the rows. Some statistics for the routing are summarized in figure 5.9 below. *Columns* are the number of distinct vertical (across a row of cells) positions occupied by wires. *Tracks* indicate the width of the routing channel, and is the number of horizontal positions used. The

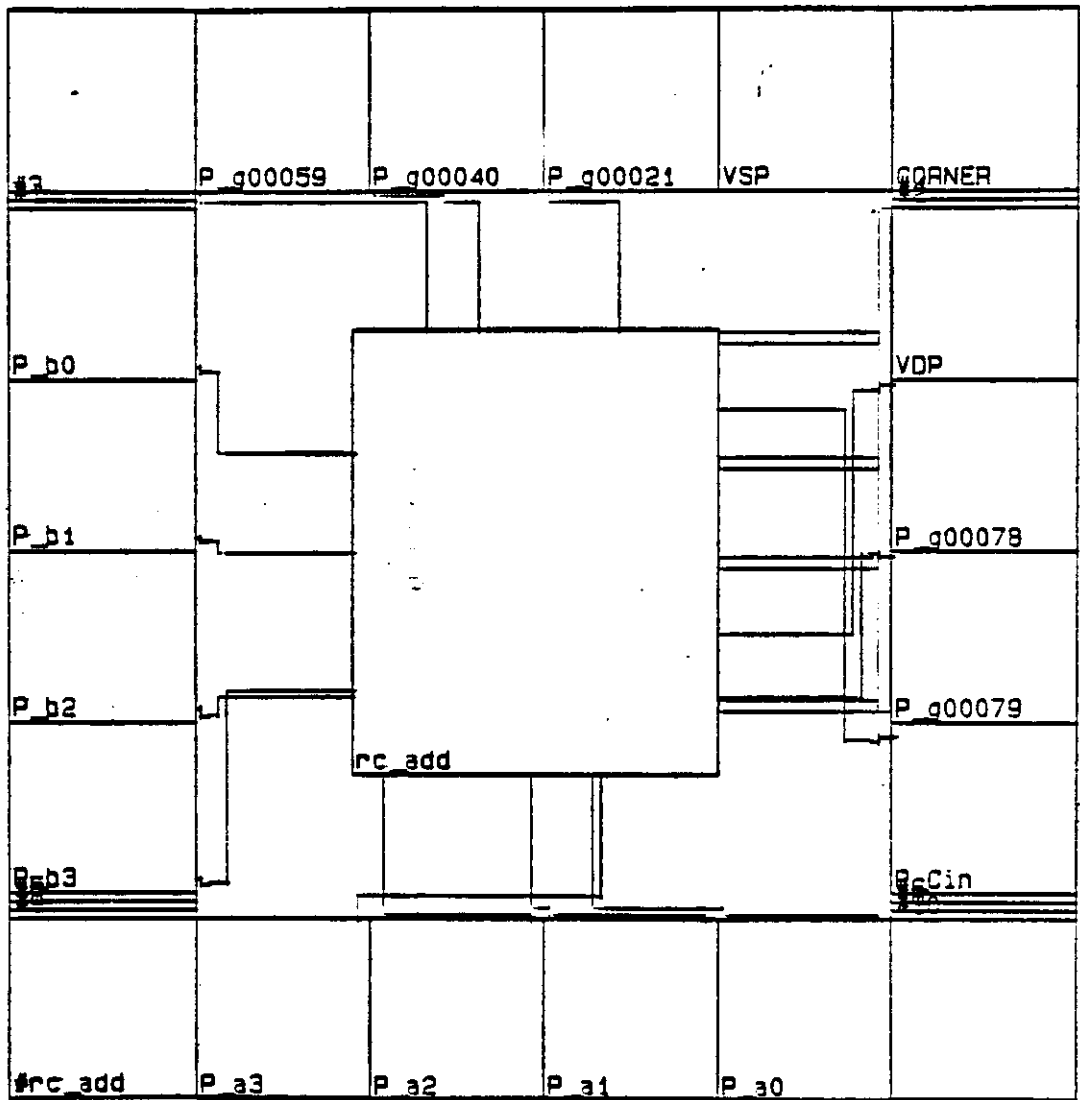


Figure 5.7 - Four-Bit Ripple Carry, Top Level View



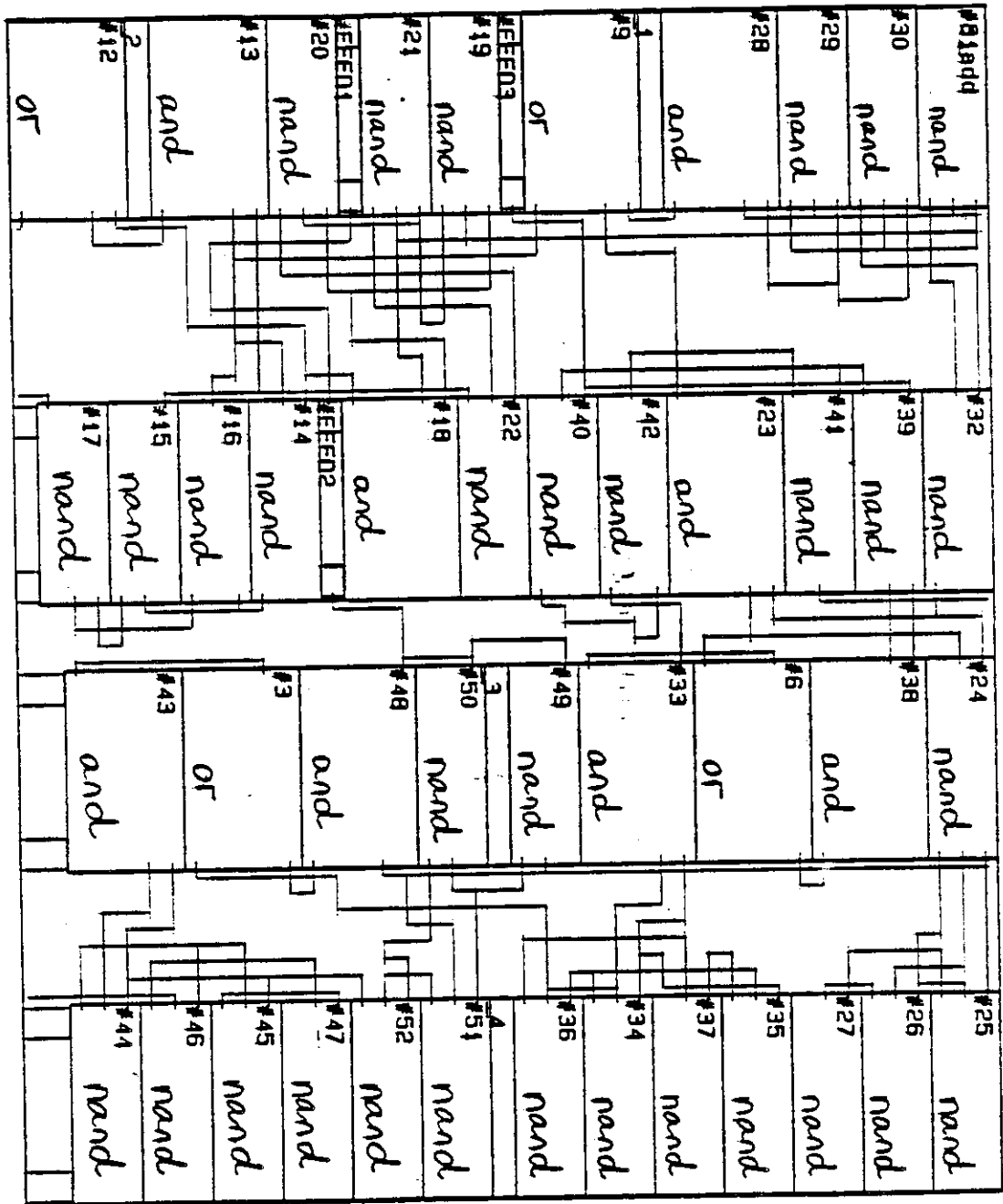


Figure 5.8 - Four-Bit Ripple Carry, Logic Level

Routing for 4-bit Ripple Carry Adder			
344.4 × 426.0 microns			
Channel	Columns	Tracks	Wire/Area
1	43	12	1.000
2	42	5	0.417
3	41	9	0.750
Wire/Cell area ratio = 0.540			

Figure 5.9

entry labeled *wire/area* refers to the ratio of the area of the wires to the total area available in the channel. As an overall metric for the layout, the *wire/cell* area ratio represents the area occupied by wires compared to the total area of the cells.

The numbers in the corner of each cell are assigned by the layout program as it traverses each successive level of the hierarchy. To clarify the layout, the logic function of the cells has been written in the corresponding box.

#### 5.4.5. The Effect of Hierarchy

The preceding layout of the four-bit ripple carry adder used all the hierarchy available in the net list extracted from the functional description. With a slight modification to the BDL translation program, the net list can be used to generate a single level description of the full adder. Each step in the translation is the same as when the hierarchy was used except that the signals, instances and connections are all derived from the bottom level of the hierarchy, rather than the next lower level.

*ripple\_carry\_adder*, as before, is the top level block. A top level layout of the circuit derived from this flat BDL description (figure 5.10) appears almost identical to

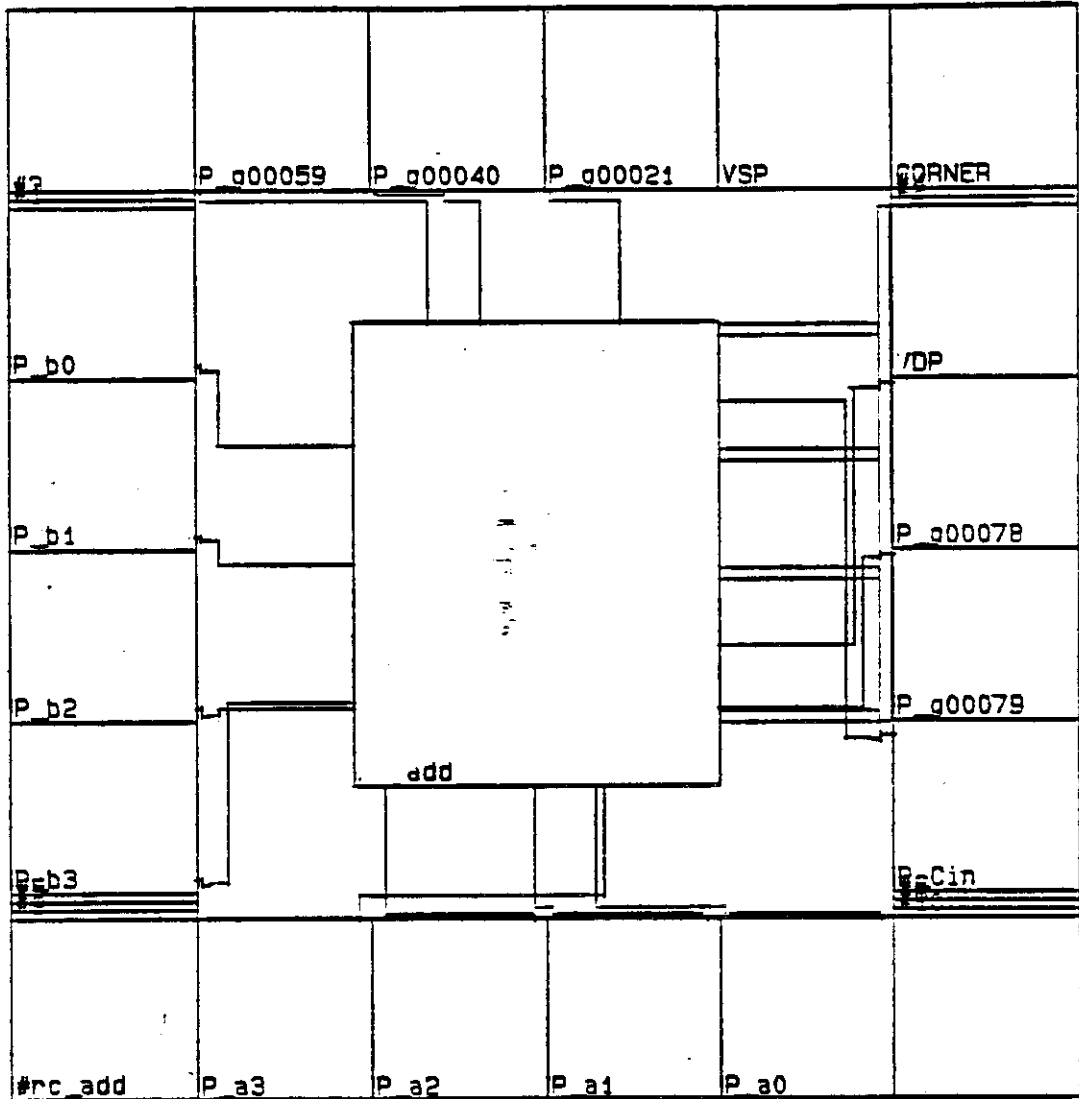


Figure 5.10 - Four-Bit Ripple Carry, No Hierarchy, Top Level View

the corresponding previous layout. The layout of the logic (figure 5.11), however, shows some small differences. The logic occupies an area 344.4 by 444.0 microns, 4.23% larger than the previous layout.

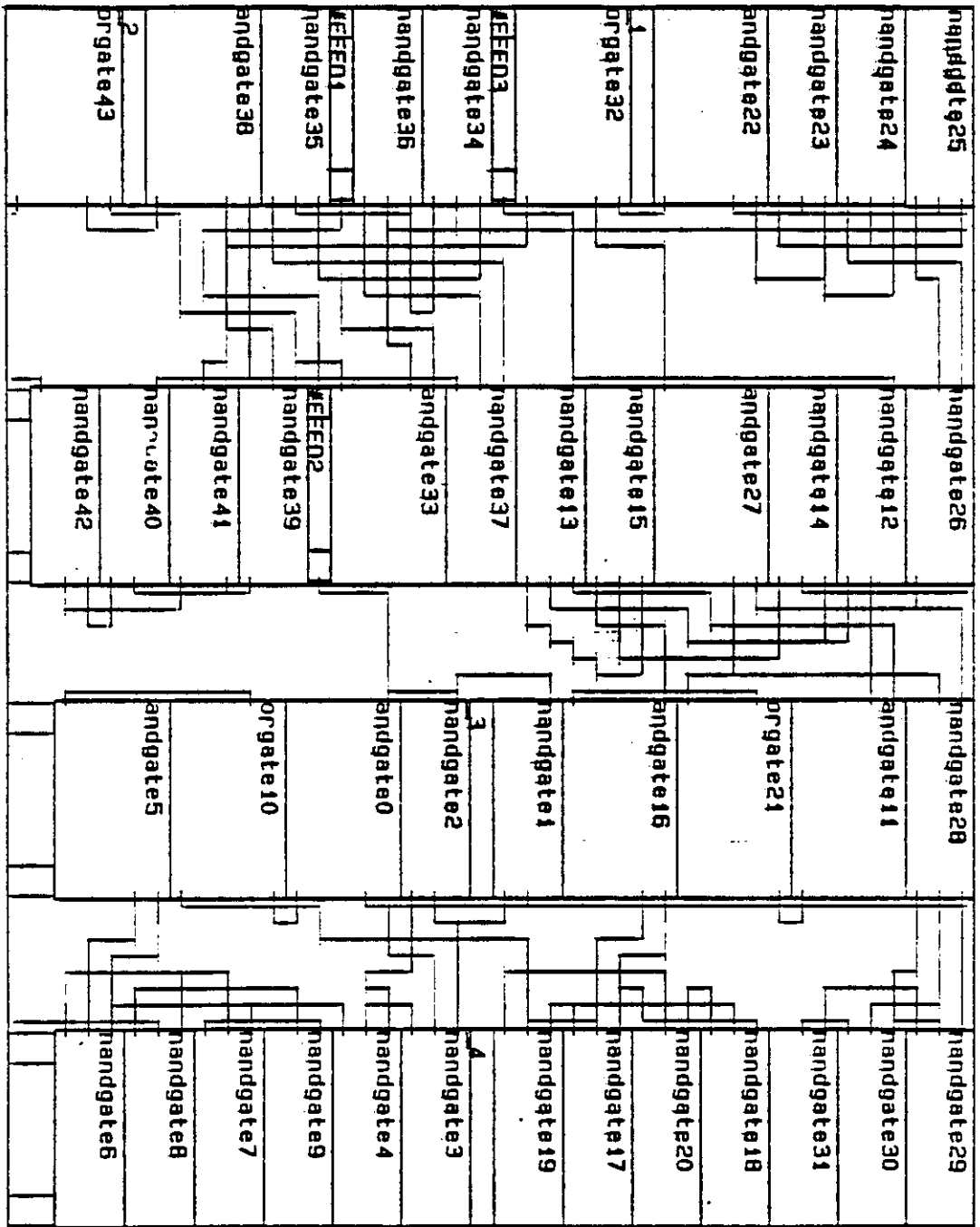


Figure 5.11 - Four-Bit Ripple Carry, No Hierarchy, Logic Level

While the placement of the gates is identical, the routing, summarized in figure 5.12, shows a 60% increase in the number of tracks in the second channel, and a 11.48% increase in the total wire area. Two of the connections previously made in the first channel have been moved to the second channel, possible since the logic cells have input and output ports on both sides. These two extra connections force the additional tracks in the second channel to accommodate them.

Since the design program flattens the hierarchy of the BDL blocks before the place and route begins, the expectation is that there would be no difference between the layouts generated from a hierarchal and a flat circuit description. There is a slight difference, however, in the order the design program encounters the primitive cells. The hierarchy flattening is a breadth-first process: one level of the hierarchy is flattened completely before proceeding to the next lower level. A flat description, owing to the translation algorithm, represents a depth-first flattening, where the lowest level in the hierarchy is reached before examining the next element in the net list.

<b>Routing for 4-bit Ripple Carry Adder - No Hierarchy</b>			
<b>344.4 × 444.0 microns</b>			
<b>Channel</b>	<b>Columns</b>	<b>Tracks</b>	<b>Wire/Area</b>
1	43	12	1.000
2	42	8	0.667
3	41	9	0.750
<b>Wire/Cell area ratio = 0.602</b>			

Figure 5.12

This difference is indirectly reflected by the labels on the gates. In the flat description, the number on the end of the name shows that the gates are encountered in the sequence **and gate, four nand gates, and gate, four nand gates** and an **or gate**. In the hierarchal description, the four **or gates** are encountered first, then the sequence of an **and gate** followed by four **nand gates**, repeated four times.

Another interesting difference between the two layouts is the computation time involved. Measured with the system clock, the layout using hierarchal description took 217 seconds of real time to generate. From a flat description, the design program required 227 seconds, a 4.6% increase. While the hierarchal layout required extra time for design flattening and routing, the first phase of the design program, which builds the initial connectivity lists, ran in 40% of the time required for the flat description. Experiments with other circuits (see Appendix A for additional hardware descriptions) show that these percentages are approximately proportional to the degree of hierarchy in the circuit.

## CHAPTER 6

### CONCLUSION

#### 6.1. Summary

The advantages of a high level digital system description language are realized in rapid prototyping, decreased testing and debugging time, increased flexibility to evaluate and experiment and a smoother transition from system design to implementation. The simple, yet powerful, *net list* model for digital systems provides a sound basis for describing systems. Net lists themselves, however, are tedious for human users, owing to their connection oriented nature and the large volume of information present in a complete description.

In looking for a high level, algorithm oriented means to describe net lists, the *procedural* style of description is found wanting. The memory oriented data representation, weak combining forms, and inherent sequential nature present more of a hindrance to describing digital systems than a help.

A *functional* style description, which uses a flexible data representation and explicit, useful combining forms, can express the functionality *and* structure of a digital system clearly and concisely. A functional style description can describe either a specific circuit, such as a full adder, or a general class of circuits, such as an N-bit ripple carry adder. These styles are complementary; net lists can be extracted by tracing the flow of information during symbolic execution.

The hierarchal nature of a functional description makes it readily usable as a multilevel specification language in the VLSI design environment. The description can serve directly for algorithm testing or formal analysis. Where the ability of a strictly functional description does not suffice, translation programs can bridge the gap. Extracting and translating net lists also allows existing design tools to be included in the functional design environment.

## 6.2. Conclusions

A functional style of digital system description solves many of the difficulties of the procedural style. The data representation is general and dynamic, so that functions can have arbitrary inputs and outputs, and change the data freely as needed. Powerful combining forms provide clear and explicit connections between pieces of a function. Most important, the inherent sequential, word-at-a-time nature of the procedural style is removed from the language.

Since a functional style is based on the combination of functions, putting a system description together is a natural part of the style. A bottom-up design is enhanced by the ability to combine and test modules; the top-down approach benefits from the ability to specify the functionality and interface to lower level modules before they are defined.<sup>1</sup> The hierarchal nature of functional programming and the well defined behavior and interface of functions, free from side effects and global

---

<sup>1</sup>See the conditional sum adder discussion in Appendix A for an example of top down design using functional description.



state, simplifies building and using libraries of functions.

Extracting net lists from the execution of a functional description and the use of translation programs provides a simple method of employing the high level description in lower levels (i.e., electrical and physical) of digital design. The separation of the functional and physical aspects of a system simplifies the design of both the FP interpreter and translation programs. By keeping the functional description largely independent of the underlying technology and implementation details, the same description can be used for any translation program. This allows the power of functional description to be combined with existing design tools, while guiding and driving the development of new ones.

### **6.3. Future Work**

Many important aspects of the use of functional style descriptions still need to be developed to provide a useful and coherent design environment. These break down into roughly three areas: the direct use of the high level description, the low level interface and design tools, and modifications and extensions to the functional language itself.

The analysis of functional hardware descriptions has been partially explored. Lahti [Lahti81] showed how to obtain component counts and an estimate of parallelism from functional descriptions. Schlag [Schlag84], using a special interpreter, was able to extract system topology directly from an FP specification. Meshkinpour [Meshkinpour85] was able to extract rough timing information from functional

descriptions. Sheeran [Sheeran84] discusses the algebraic properties of a functional language when extended to sequential circuits. Patel, Schlag, and Ercegovac [Patel85] demonstrated the use of the algebraic properties of FP to optimize a system design at the description level. Possible areas to explore next include automated algorithm analysis, program proofs, circuit test generation, and expert systems for analyzing or even synthesizing digital designs.

The design and construction of translation programs and lower level design tool interfaces is almost a completely new area. In addition to the translator to BDL, there is also a translator to a component level circuit simulator and a switch level logic simulator. In addition to the mapping of the net list to the target language, additional translation considerations become relevant depending on the type of translation and the target language:

- The use of standard cells and/or libraries for generating output.
- Signal naming and typing
- Management of hierarchy in the net list if it can be expressed in the target language.
- Peephole or global optimization, exploiting design "tricks" during translation

Finally, the functional description language itself may need extensions or improvements as further experience is gained in the design environment. One obvious concern is the inability of a functional language to reflect *state*. Sequential machines, shift registers, or any digital system with memory or feedback loops, can-

not be described using the language presented here. One approach, as found in [Meshkinpour85], is to define a new functional form, which the interpreter understands as having an implicit state. A more general method would be to introduce a *single assignment* state, passing the task of implementation to translation programs.

Another area for improvement is in the symbolic execution of programs. By exploiting function identities, the interpreter can reduce the components of a circuit when some of the values are constants or the same value. Ideally, the application

$$\text{ripple\_carry\_adder} : \langle \langle a_n \cdots a_0 \rangle \langle a_n \cdots a_0 \rangle C_{in} \rangle$$

would generate

$$\langle a_n \cdots a_0 C_{in} \rangle$$

and there would be no gates in the net list.

The functional style of digital system description provides not only a new method for describing systems, but a new model for what a digital system is. This simple, consistent model removes the restrictions and limitations of the step-oriented, procedural view. Though the syntax and semantics can be initially confusing, functional style description quickly becomes a powerful tool in the design and development of digital systems.

## APPENDIX A

### A COLLECTION OF HARDWARE DESCRIPTIONS

#### A.1. Binary Decoders

Decoders take  $N$  binary inputs and generate  $2^N$  binary outputs. For each bit pattern appearing on the inputs, exactly one of the  $2^N$  output lines will be *true*. The outputs of a decoder are usually numbered  $0, 1, \dots, 2^N$  such that output  $X$  is true if and only if the input bit pattern has the decimal value  $X$  interpreted as a binary number. Decoders have many uses: finite state machines, device control circuitry, memory circuits and in multiplexers, as shown later in the appendix.

The simplest decoder has one input and two outputs, consisting of a single inverter. The FP description (below) and the block diagram (figure A.1) are correspondingly simple:

```
define  decode_1(x)
        [id, inverter]
end
```

This simple circuit, however, presents a difficult problem for a translation program: when this decoder appears in the net list, the input and one of the outputs have the same name. The solution, unfortunately, is most often specific to the target language. In BDL, the signal must either be renamed or the output “version” of the signal must be ignored.

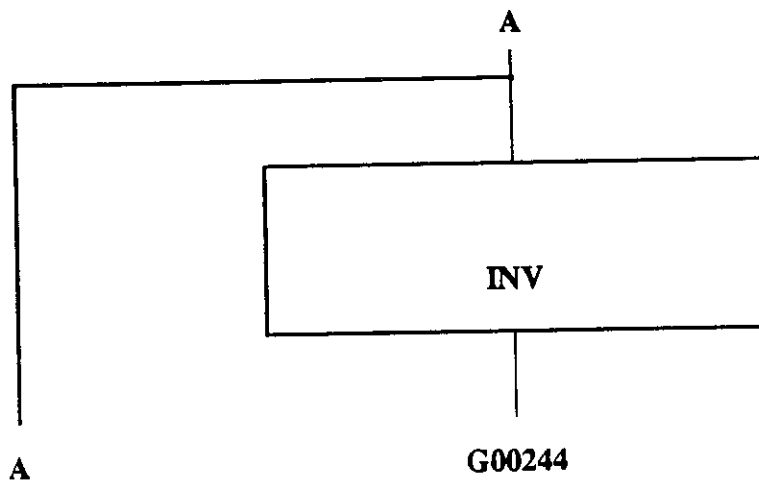


Figure A.1 - A Single Bit Decoder

There are several types of decoders, falling roughly into two categories: *coincident* decoders and *tree* decoders.

#### A.1.1. Coincident Decoders

A coincident decoder decodes  $N$  bits by first decoding the most and least significant  $\frac{N}{2}$  bits in parallel, then anding each of the possible pairs of bits from the two smaller decoders to get the full decoding. This scheme is applied recursively until the number of inputs to be decoded is small enough to be realized by an existing primitive.

Before defining the coincident decoder, it is useful to define a function to cross-match the bits from the two smaller decodings. This function will be called *cross\_match*, and is defined as follows:

```

define  cross_match(x, y)
        concat    @
        &distl    @
        distr
end

```

The actual cross-matching can be formed by distributing either from the left or right first, but the order above preserves the proper bit position for the results.

For simplicity, the coincident decoder described below continues to divide the input bits until it encounters a single bit. This can then be decoded using the *decode\_1* circuit. Since the input is a sequence of bits, the single bit must be selected from the sequence to be decoded. Otherwise, the sequence is *split*, decoded, and cross-anded:

```

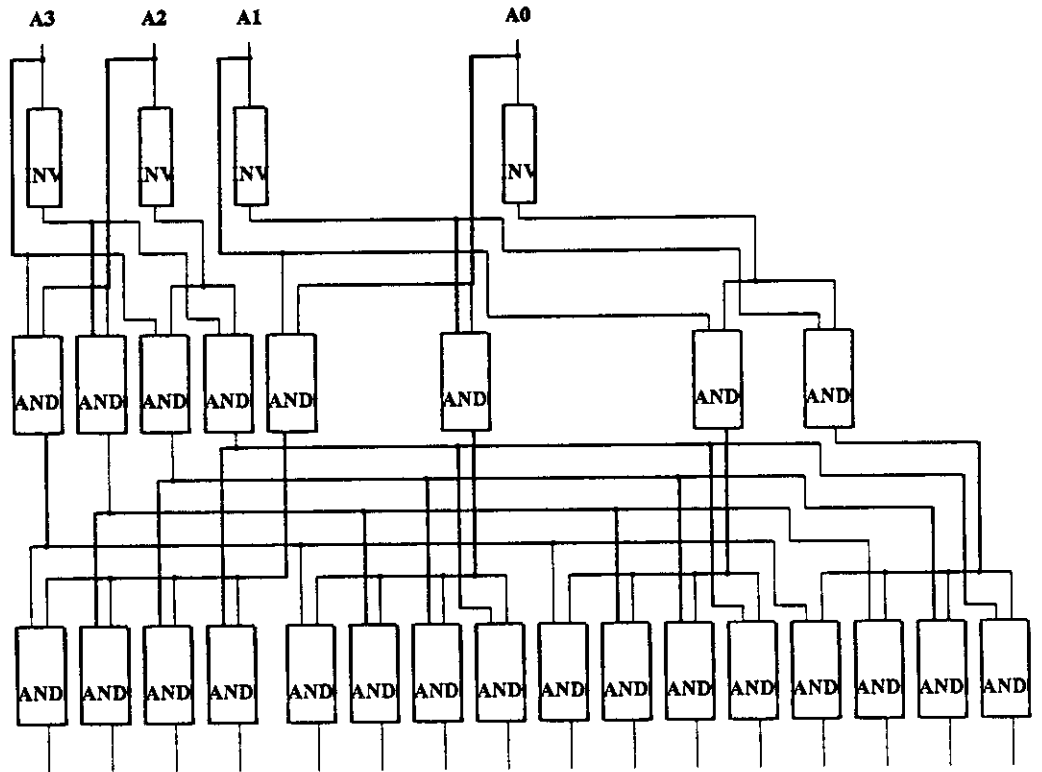
define  C_decode()
        if      length = %1
        then    decode_1 @ 1
        else    &andgate          @
                cross_match      @
                &C_decode @ split
        fi
end

```

The block diagram below (figure A.2) shows a four bit coincident decoder at the logic gate level.

### A.1.2. Tree Decoders

Like the coincident decoder, a tree decoder reduces the number of inputs until a primitive module can do the decoding, then combines the results to generate the full decoding. Instead of recursively splitting, however, the tree strategy divides the  $N$



G0033G0033G0033G0034 G0034G0034G0034G0034G0035G0035G0035G0035G0035G0035G0035G0035G0036G0036G0036G0036

Figure A.2 - A Four-Bit Coincident Decoder

inputs into  $G$  groups of the primitive size  $P$ . At the top of the tree, the most significant  $P$  bits are decoded by the primitive module. At the next level, the next most significant group of  $P$  bits is decoded  $2^P$  times in parallel; the output from the previous level is used to enable one decoder so that the remaining  $2^P - 1$  decoders' outputs are all *false*. The enabling is done by **anding** the  $n^{th}$  bit from the previous level with each bit of the output of the  $n^{th}$  decoder in the current level.

At the  $k_{th}$  level of the tree, there are  $2^k$  bits from the higher level of the tree and  $G-k$  groups of  $P$  bits left to decode. The current level can then be built by decoding the next group  $2^k$  times in parallel, then anding each bit from the previous level with each bit in the corresponding decoder, i.e., enabling the outputs. This scheme continues until there are no more groups to decode and the input is fully decoded.

The main function to describe a tree decoding scheme, then, is one which takes two vectors of bits, the previous decoding and the remaining groups, performs the necessary decoding and enabling, then continues the process with the new decoded bits and the remaining groups. When the function is first applied, the vector of decoded bits will be *null*; in this case, the first group is simply decoded and passed to the next stage as the decoded bits. The building terminates when the vector of input bits is *null*. As with the coincident decoder, the primitive decoder in this example is *decode\_1*, that is,  $G$  is equal to  $N$  and the size of  $G$  is one bit.

```

define  T_decode_next(prev, groups)
  if    null @ groups
  then  prev
  else  T_decode_next @
        if    null @ prev
        then  [decode_1 @ 1 @ groups, tl @ groups]
        else  [concat @
                &(&andgate @ distl @ [1, decode_1 @ 2]) @
                distr @ [prev, 1 @ groups]
                tl @ groups]
        fi
  fi
end

```



The main function for a tree decoder sets up the initial input for the actual building function.

```
define  T_decode()  
        T_decode_next @ [%(), id]  
end
```

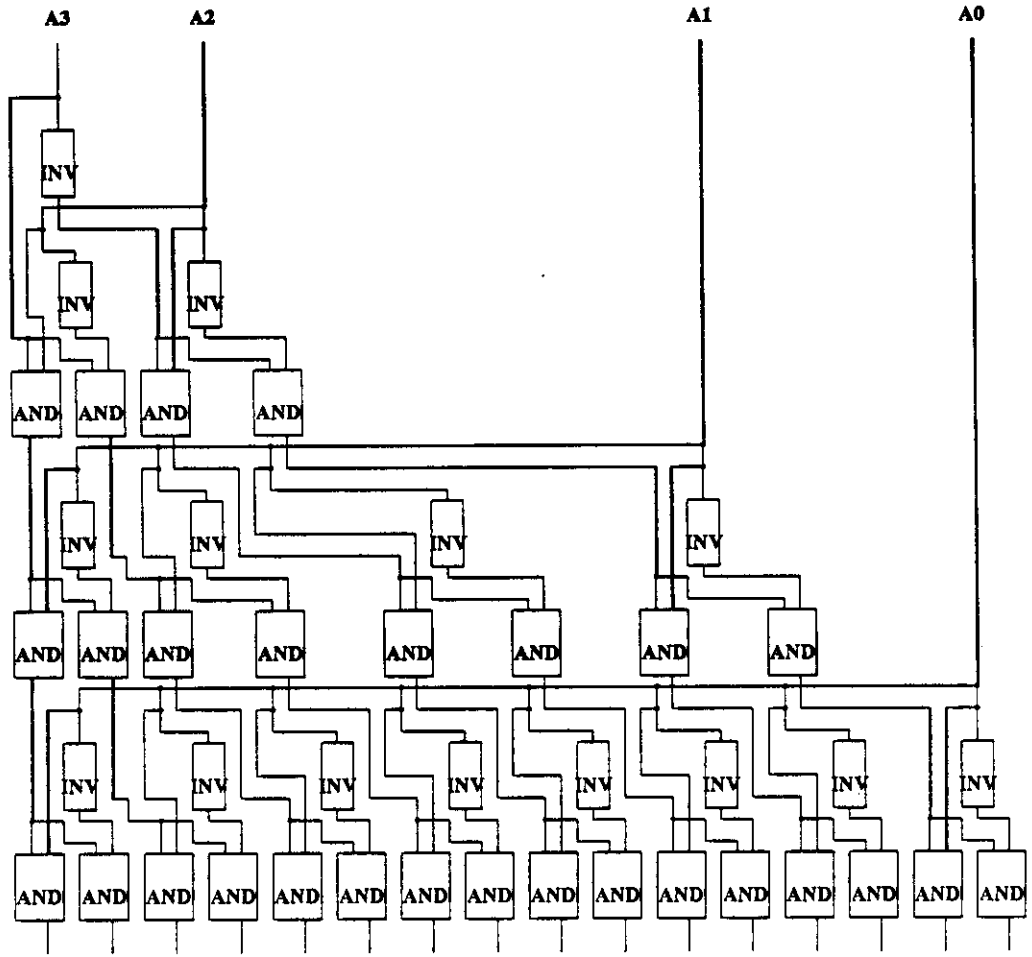
The block level diagram (figure A.3) clearly shows the tree structure of the decoding scheme in a four-bit decoder.

## A.2. Encoder

An encoder performs the inverse operation of a decoder. There are  $2^N$  inputs, exactly one of which will be *true* at any time. The  $N$  outputs form a binary number corresponding to the *true* input bit.

A particular bit of the output is *true* if any of  $2^{N-1}$  of the inputs is true; which set of  $2^{N-1}$  varies for each bit position. It can be observed that the  $k^{\text{th}}$  output bit, where the least significant bit is numbered 0, is generated by dividing the inputs into groups of  $2^k$  bits, then oring together the first, third, etc., groups in the sequence. Further, the input grouping for the  $(k+1)^{\text{th}}$  output bit is formed by *pairing* the current groupings, then *concatenating* the groups in each pair.

The proper groups for oring are selected by *pairing* the input groups, then selecting the first element in the pairs. *Concatenation* combines the groups into a single bit vector, which is then *ored* by an associative insert of *orgate*.



G00058G00064G00064G00064G00070G00070G00070G00070G00078G00082G00082G00082G00082G00088G00088G00090G00090G00090G00090G00100G00102

Figure A.3 - A Four-Bit Tree Decoder

```

define  encode_bits()
  if    length = %2
  then  |orgate @ 1
  else  apndr @
        [encode_bits @ &concat @ pair,
         |orgate @ concat @ &1 @ pair]
  fi
end

```

Since the description above assumes that the inputs start as  $2^N$  groups of one bit, there must be a preliminary description to do the grouping, then call *encode\_bits*:

```

define  encode()
        encode_bits @
        &(if atom then [id] else id fi)
end

```

The gate level diagram for an eight-bit encoder (figure A.4) shows that an encoder merely chooses which bits to or together; an interesting point is that the least significant bit of the input, A0, is not used in the encoder circuit.

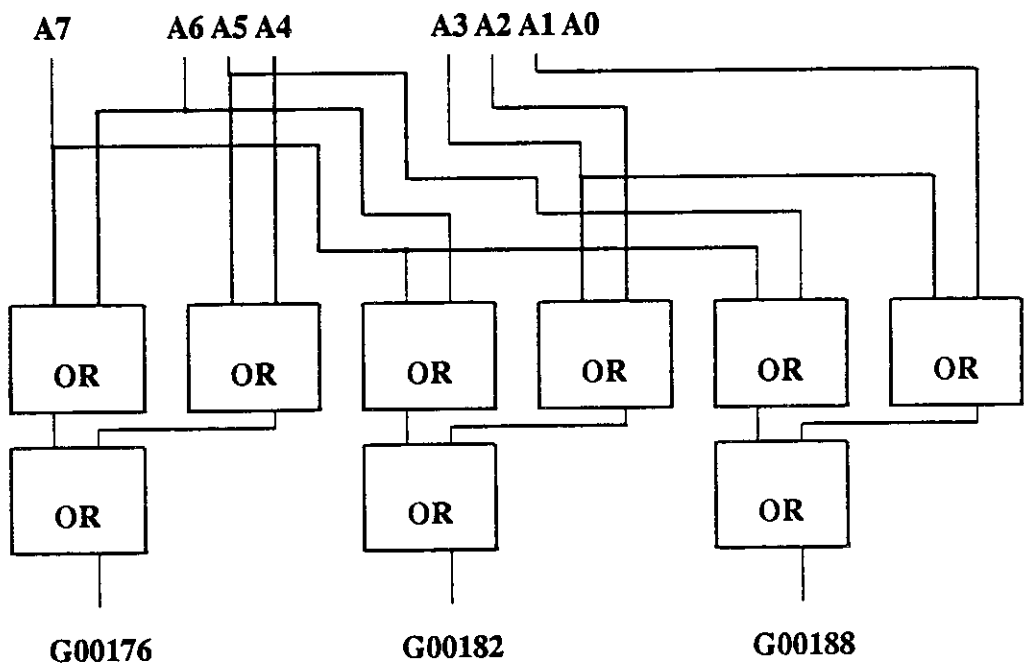


Figure A.4 - An Eight-Bit Encoder

### A.3. Multiplexer

A multiplexer circuit chooses one of  $2^N$  input values using  $N$  input control signals. Since the choices may consist of single bits or bit vectors, it would seem that two multiplexer descriptions are needed. One description can suffice, however, if an input of  $2^N$  bits is viewed as choosing one of  $2^N$  vectors, each of length one.

First, a multiplexer decodes the  $N$  control signals, then **ands** each bit of the decoder output with each bit in the corresponding input bit vector. Finally, all the  $k^{\text{th}}$  bits from each vector generated by the anding are **ored** together to form the  $k^{\text{th}}$  output. In the description below, the coincident decoder circuit described earlier is used to decode the control inputs; however, any decoder circuit can be used. The initial **if** ... **then** ... **else** statement converts an input of  $2^N$  bits into a sequence of  $2^N$  groups of one bit each, as discussed above.

```
define    multiplex(control, choices)
          &|orgate
          trans @
          &(&andgate @ dist1) @
          trans @
          [C_decode @ control,
           &(if atom then [id] else id fi) @ choices]
end
```

The gate level diagram in figure A.5 is a multiplexer that chooses between one of two four-bit vectors.

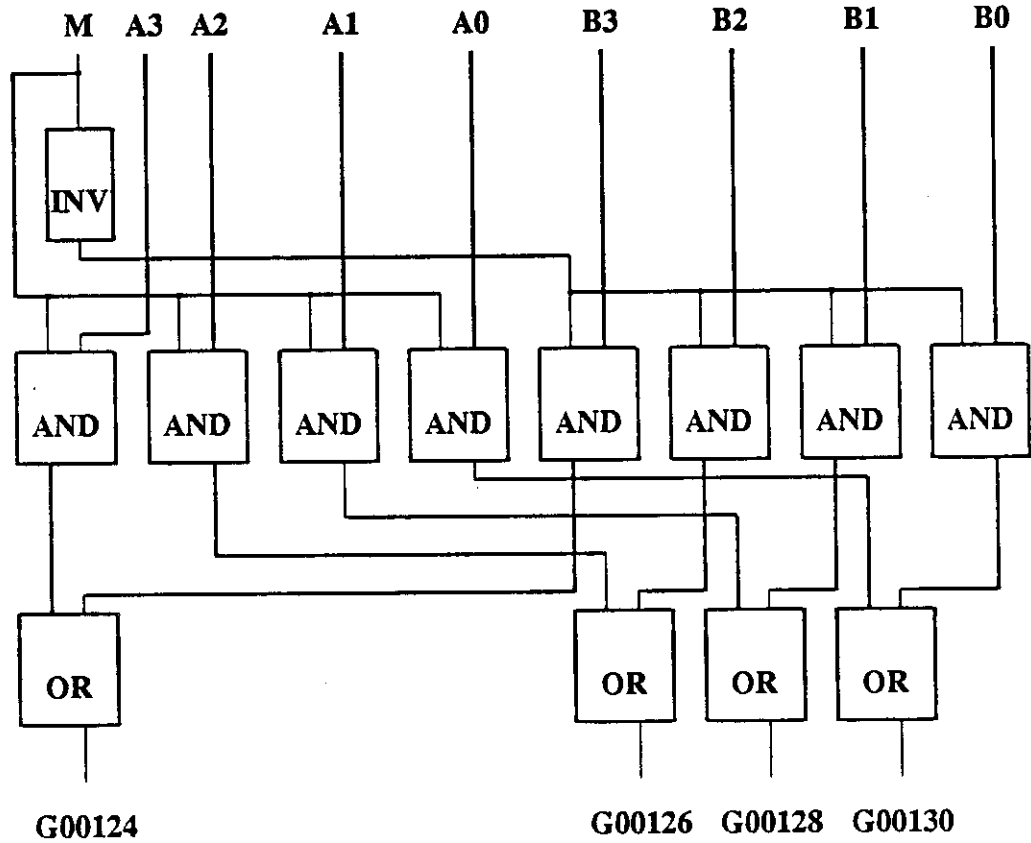


Figure A.5 - A 4 x 2-to-1 Multiplexer

#### A.4. Adders (*vipera sumupus*)

The relatively simple function of addition has received great attention in computer design. There are many types of adders: ripple carry, carry look-ahead, carry skip, carry save, and conditional sum, to name a few. Each different design makes some tradeoff between the complexity of the circuit and the speed of operation. Below are the functional descriptions for the basic adder circuits, a ripple carry, a conditional sum and a binary coded decimal (BCD) adder, along with some

explanation of their derivation.

#### A.4.1. Basic Adders

The basic adders are the half and full adders. The half adder consists of an xor and and gate in parallel, easily described as:

```
define  half_add(x, y)
        [ x andgate y, x xorgate y ]
end
```

Given the description of a half adder, we can now describe the full adder:

```
define  full_add(x, y, Cin)
        [1 orgate 2, 3]      @
        apndl                @
        [1, 2 half_add 3]   @
        apndr                @
        [x half_add y, Cin]
end
```

The full adder description, unlike the half adder, is more complex than the actual circuit. The apndr and apndl steps are used to simplify the selection. Using the proper selectors, the full adder description could have been:

```
define  full_add(x, y, Cin)
        [1 orgate (1 @ 2), 2 @ 2]  @
        [1 @ 1, (2 @ 1) half_add 2]  @
        [x half_add y, Cin]
end
```

This bears a closer resemblance to the actual circuit, but is less readable; both descriptions specify exactly the same circuit.

#### A.4.2. Ripple Carry Adder

The simplest adder is the ripple carry adder, performing addition much as a human being would: forming the sum in the  $n^{\text{th}}$  position, adding the carry (if any) to the  $(n+1)^{\text{th}}$  position. The macro will be specified to add two vectors of bits using an initial carry, so the input object will be of the form:

$$((x_{n-1} \cdots x_0)(y_{n-1} \cdots y_0) C_{in})$$

with the output object being a single vector

$$(C_{out} s_{n-1} \cdots s_0)$$

Each step in the addition will be applied to an object consisting of the unadded pairs (waiting for the carry to propagate), the computed sum, and the current carry:

$$((x_{n-1} y_{n-1}) \cdots (x_k y_k) c_k s_{k-1} \cdots s_0)$$

Each step will add the next pair and the carry together using a full adder:

```
define rc_adder(bitpr, Cin)
  if (length @ bitpr) = %0
  then full_add @
      apndr @ [1 @ bitpr, Cin]
  else concat @
      [full_add @ apndr @ [1, 1 @ 2], tl @ 2] @
      [1 @ bitpr, (rc_adder @ [tl @ bitpr, Cin])]
  fi
end
```

The main function, *ripple\_carry\_add*, will set up the object for *rc\_adder* to operate on:

```

define   ripple_carry_add(x, y, Cin)
           rc_adder @
           [trans @ [x, y], Cin]
end

```

The gate level diagram for a four-bit ripple carry adder is shown in figure A.6.

### A.4.3. Conditional Sum Adder

The conditional sum adder [Sklansky60] avoids carry propagation by forming both possible sums ( $C_{in} = 0$  and  $C_{in} = 1$ ), then choosing the proper result when the carry becomes available. The formation of the two sums can be stated recursively:

```

if      there is one pair of bits to add

then   use a conditional half adder (which generates both possible
           sums).

else   divide the pairs to add in half; form both sums for each
           half. For each carry of the least significant half sums,
           choose the correct sum and carry from the most significant
           half, and concatenate it.

```

In practice, the initial carry is known, so  $\frac{1}{4}$  of the sums need not be formed; the description below, however, makes no assumptions about the initial carry value.

The functional solution can now be designed in a top-down manner. The top level function, *cond\_sum\_adder*, will cross match the bits, then apply the adder recursion:





**end**

The function *c\_sum\_add* will perform the conditional summing algorithm.

From *cond\_sum\_adder*, the object passed will be of the form:

$$((x_{n-1} y_{n-1}) \cdots (x_0 y_0) C_{in})$$

where  $(x_{n-1} y_{n-1})$  are pairs to add, and  $C_{in}$  is the initial carry. At each step in the recursion, the pairs to be added are *split*. *c\_sum\_add* will be applied recursively to the right half of the vector, until a full addition is performed upon two bits and the initial carry. When the pairs are split, the carry will be grouped with the right half.

```
define c_sum_add (pairs, carry)
  if length@pairs = %1
  then full_adder @
      apndr @
      [1 @ pairs, carry]
  else picksum @
      [form_both @ 1, c_sum_add @ 2] @
      [1 @ 1, [1 @ 2, 2]] @
      [split @ pairs, carry]
  fi
end
```

The algorithm for *form\_both*, generating both possible sums, was given above.

The functional version is easily derived to be:

```

define form_both()
  if length = %1
  then c_half_add
  else &pick_next @
      distl @
      &form_both @
      split
  fi
end

```

The conditional half adder, *c\_half\_add*, is composed of a regular binary half adder and the logic to generate the half adder output plus one:

```

define c_half_add(x, y)
  [[ x orgate y, x xnorgate y ], x half_add y]
end

```

The remaining function to define is *pick\_next*, which takes an object of the form

$$((c_1 s_{n-1,1} \cdots s_{k,1})(c_0 s_{n-1,0} \cdots s_{k-1,0})) (c_k s_{k-1} \cdots s_0)$$

and produces

$$\begin{aligned} &(c_1 s_{n-1,1} \cdots s_{k,1} s_{k-1} \cdots s_0) \text{ if } c_k \equiv 1 \\ &(c_0 s_{n-1,0} \cdots s_{k,0} s_{k-1} \cdots s_0) \text{ if } c_k \equiv 0 \end{aligned}$$

The function will use *multiplexer*, defined previously, to choose between the two sum vectors. The code for *pick\_next* is:

```

define pick_next(choices, sum)
  concat @
  [multiplexer @ [1 @ sum, choices], tl @ sum]
end

```

The gate level diagram for a four-bit conditional sum adder is shown in figure

A.7.

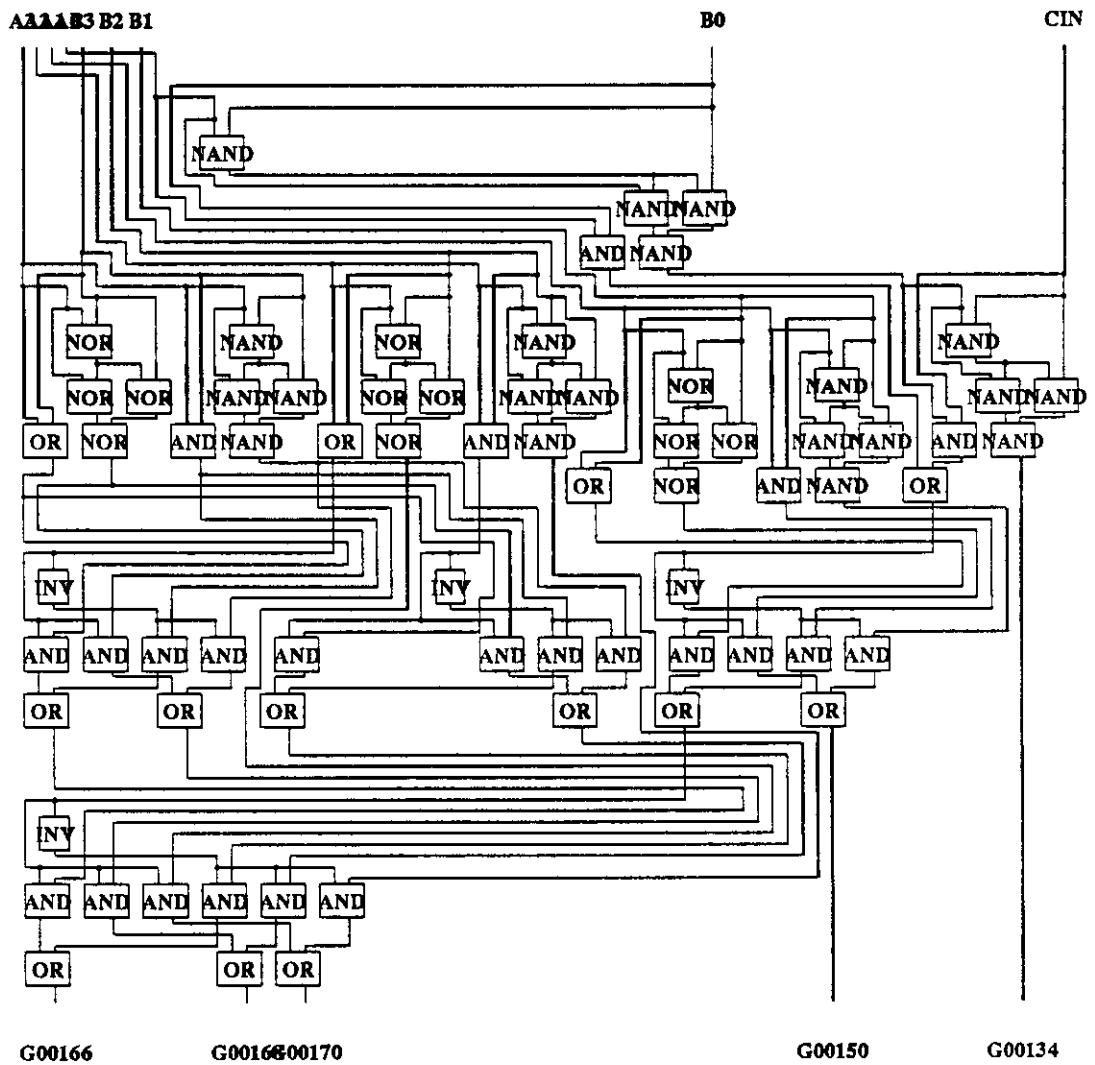


Figure A.7 - A Four-Bit Conditional Sum Adder

#### A.4.4. A BCD Adder Module

In a binary coded decimal (BCD) representation of a number, each four bits encodes one digit of decimal number. For example, the integer 123 would appear as 0001 0010 0011 in BCD notation.

The algorithm for adding two BCD digits consists of two steps:

- (1) Add the BCD digits as if they were four bit binary numbers.
- (2) If the result of the addition (including the carry bit) is greater than nine (01001), add six (0110) to the low order four bits, i.e., exclude the carry. The carry out is the oring of the decimal overflow and the carry (if any) from the adjustment stage.

The addition of four bit numbers can use any addition algorithm; the ripple carry adder described above will be used here. Using a top-down approach, the BCD adder block description is:

```
define   BCD_add(x, y, Cin)  
          bcd_adjust @  
          ripple_carry_adder  
end
```

Adjusting the result requires a circuit to add six (0110) to a four bit number if the BCD digit overflows. Since the logical result of the overflow is also a numeric binary value, the adjustment simply consists of adding the overflow result to the second and third bit positions of the result. This can be realized by two half adders and a full adder circuit as follows:

```

define   bcd_adjust(Cout, S3, S2, S1, S0)
           apndl                                     @
           [orgate @ [1, 1 @ 2], [2 @ 2, 3, 4, 5]]   @
           [1, half_add @ [2, 1 @ 3], 2 @ 3, 4, 5]   @
           [1, 2, full_add @ [3, 1 @ 4, 1], 2 @ 4, 5] @
           [1, 2, 3, half_add @ [1, 4], 5]           @
           [bcd_overflow, S3, S2, S1, S0]
end

```

*bcd\_overflow* returns *true* if the decimal value of the five bit binary input is greater than nine (01001). For the value to be greater than nine, either the carry out bit is *true*, or the most significant sum bit is *true* and one of the other two three most significant bits is on. The translates easily in the functional description:

```

define   bcd_overflow(Cout, S3, S2, S1, S0)
           Cout orgate (S3 andgate (S2 orgate S1))
end

```

The block level diagram for the BCD module (figure A.8) uses the four bit ripple carry adder from figure A.6, and the half and full adders outlined in Chapter 4.

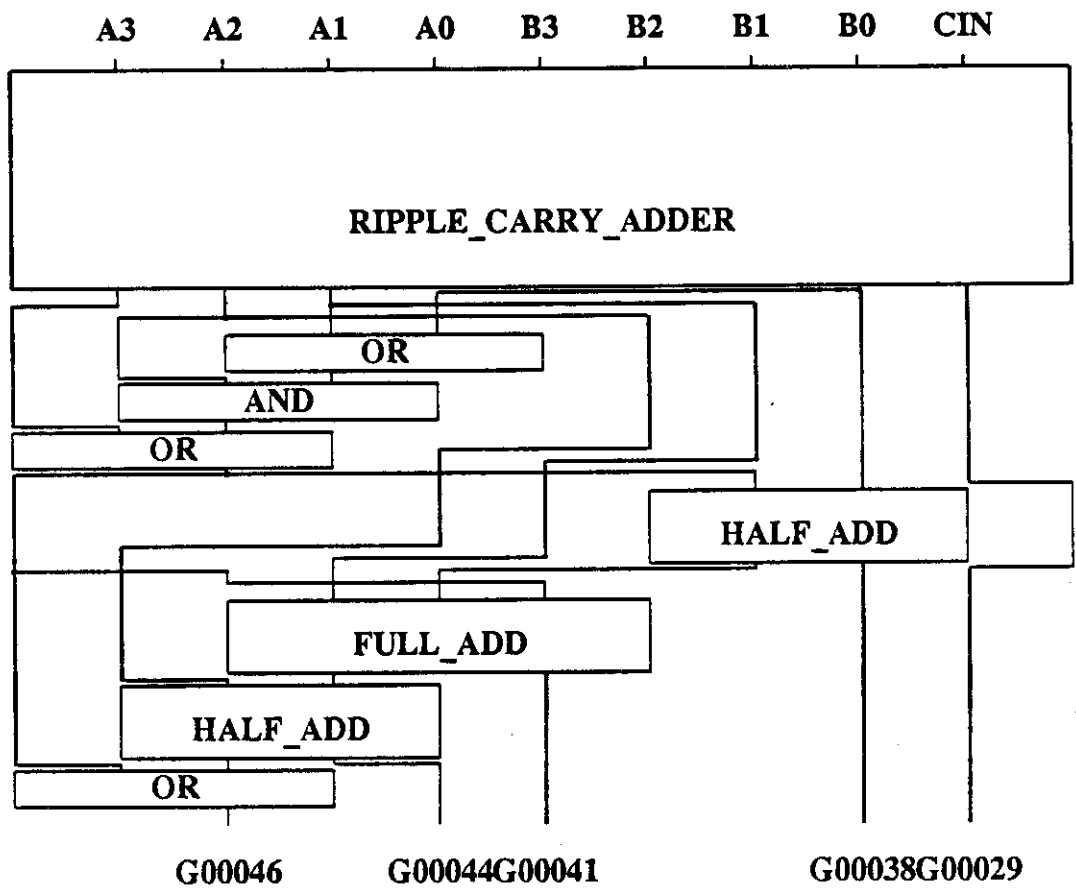


Figure A.8 - A BCD Adder Module

## REFERENCES

- [Backus78] 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) pp. 613-641 (Aug 1978).
- [Baden82] Scott Baden, *FP Users' Manual Version 4.0*, University of California, Berkeley (1982).
- [Chu65] Yaohan Chu, "An ALGOL-like Computer Design Language," *Communications of the ACM* 8(10) pp. 607-615 (October 1965).
- [Dasgupta84] S. Dasgupta, *The Design and Description of Computer Architecture*, Wiley-Interscience, New York, NY (1984).
- [Duley68] J. R. Duley and D. L. Dietmeyer, "A Digital System Design Language (DDL)," *IEEE Transactions on Computing C-18* pp. 850-861 (September, 1968).
- [Feldman83] Stuart Feldman, "The Circuit Design Language Xi ( $\Xi$ )," *IEEE Conference on Computer Design VLSI in Computers*, pp. 652-655 (October 31 - November 1, 1983).
- [Hill75] F. J. Hill, "Updating AHPL," *Proceedings of the International Symposium on CHDL Applications*, pp. 22-29 (1975).
- [Lahti81] David O. Lahti, "Applications of a Functional Programming Language," Report No. CSD-810403,, Computer Science Department, University of California, Los Angeles, CA (April 1981).



- [Lieberherr82] Karl J. Lieberherr and Svend E. Knudsen, "ZEUS: A Hardware Description Language for VLSI," *Proceedings of the 20th IEEE Conference on Design Automation*, pp. 17-20 (June 27-29, 1982).
- [Lim65] Willie Y-P. Lim, "HISDL - A Structural Design Language," *Communications of the ACM* 25(11) pp. 823-830 (November 1965).
- [Mead80] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980).
- [Meshkinpour85] Farshad Meshkinpour and Milos Ercegovac, "A Functional Language for Description and Design of Digital Systems: Sequential Constructs," *Proceedings of the 22nd Conference on Design Automation*, pp. 238-244 (June 23-26 1985).
- [Morison82] J.D. Morison, N.E. Peeling, and T.L. Thorp, "ELLA: A Hardware Description Language," *Proceedings of ICC-82*, pp. 604-607 (September 28 - October 1, 1982).
- [Patel85] Dorab Patel, Martine Schlag, and Milos Ercegovac, "vFP: An Environment for Multilevel Specification, Analysis and Synthesis," *Functional Programming Language and Computer Architecture*, pp. 233-255 (September 16-19, 1985).
- [Robinson82] P. Robinson and J. Dion, "Programming Languages for Hardware Description," *Proceedings of the 20th IEEE Conference on Design Automation*, pp. 12-16 (June 27-29, 1982).
- [Schlag84] Martine Schlag, "Extracting Geometry from FP for VLSI Layout," Report No. CSD-840043, Computer Science Department, University of California, Los Angeles, CA (October 1984).

- [Shahdad85] Moe Shahdad, Roger Lipsett, Erich Marschner, Kellye Sheehan, Howard Cohen, Ron Waxman, and Dave Ackley, "VHSIC Hardware Description Language," *IEEE Computer* 18(2) pp. 94-103 (February 1985).
- [Sheeran84] Mary Sheeran, "mFP, A Language for VLSI Design," *Proceeding of the 1984 ACM Conference on LISP and Functional Programming*, pp. 104-112 (August 6-8, 1984).
- [Shiva79] S. G. Shiva, "Computer Hardware Description Languages - A Tutorial," *Proceedings of the IEEE* 67(12) pp. 1605-1615 (December 1979).
- [Sklansky60] J. Sklansky, "Conditional Sum Adder Logic," *IRE Transactions on Electronic Computers* C-10 pp. 226-231 (June 1960).
- [Slutz84] Eric Slutz, Glenn Okita, and Jeanne Wiseman, "BDL: A Hierarchical Block Description Language," *Proceedings of the 21st IEEE Conference on Design Automation*, p. 81 (June 25-27, 1984).
- [Viega84] P. M. B. Viega and M. J. A. Lanca, "HARPA: A Hierarchical Multi-Level Hardware Description Language," *Proceedings of the 21st IEEE Conference on Design Automation*, pp. 59-65 (June 25-27, 1984).