ON SPECIFICATION AND DESIGN OF DIGITAL SYSTEMS USING AN APPLICATIVE HARDWARE DESCRIPTION LANGUAGE

Farshad Meshkinpour

November 1984
Report No. CSD-840046

UNIVERSITY OF CALIFORNIA
Los Angeles

On Specification and Design of Digital Systems Using an Applicative Hardware Description Languague

A thesis submitted in partial statisfaction of the requirement for the degree Master of Science in Computer Science

by

Farshad Meshkinpour

1984

To My Father and Mother

Table of Contents

List of Figures

page

ACKNOWLEDGEMENTS

I am grateful to my advisor, Dr. Milos D. Ercegovac, for his guidance, patience, invaluable support, and many stimulating discussions during the course of this work.

I would like to thank Dorab Patel for providing the basic FP interpreter and for his help at many decisive moments of this work. I am in debt to Shih-Lien Lu for design of the graphic software system and for his patient help.

Thanks go to all members of the FP group in the Computer Science Department at UCLA.

I would also like to express my appreciation to my family and friends for their moral support during this work.

This research is supported in part by a grant from MICRO-Rockwell.

ABSTRACT OF THE THESIS

On Specification and Design of
Digital Systems Using an
Applicative Hardware Description Language

by
Farshad Meshkinpour
Master of Science in Computer Science
University of California, Los Angeles, 1983
Professor Milos D. Ercegovac, Chair

An applicative (functional) hardware description language (FHDL) is an enhancement of previous work using applicative languages as hardware description languages. FHDL supports the specification of both combinational and sequential digital systems. The FHDL interpreter implements the specification of hardware algorithms at the gate level. The functions in FHDL operate on attributes as well as objects. The attributes provide a mechanism for computing various parameters associated with the implementation.

FHDL supports the optimization phase of digital system design. As an example, FHDL is used to reorganize a given system in a pipelined fashion to improve its throughput.

CHAPTER 1

Introduction and Objectives

1.1 Introduction

With advances in the field of Very Large Scale Integration(VLSI), the hardware implementation of complex systems become feasible. The availability of more than half-a-million devices per chip is expected by 1985 [Quey79]. These advances provided enormous amount of resources to the digital system designers. At the same time the cost of specification, design, and testing of VLSI systems is growing. VLSI is more than "a lot of LSI". The 256-K-bit RAM's of 1982 are quite different from 1-K-bit RAM's of 1970's. As the scale of the system increases, different design issues gain importance. The internal structuring starts to appear, specialized subparts emerge, communication between the different parts gains importance, and at same level, the complexity of the system becomes an explicitly stated concern [Sequ83]. The main technique to deal with the complexity is the use of high-level languages to specify and synthesize digital systems.

Many high-level languages have been suggested to be used as a Hardware Description Language (HDL). High-level HDLs should provide the following mechanisms for handling the design complexity:

1. Abstraction methods in order to hide the unnecessary details of design.

2. Partitioning in order to use "divide-and-conquer" methods for designing large systems.

3. Structured and modular design methodologies to manage the complexity of communication among sub-parts.

Note, that various HDLs based on the high-level language capabilities provide some of the above features. The above capabilities of HDLs reduce the design time, make the design more error-free, and ease the debugging and modification [Sequ83]. Another advantage of high-level HDLs is that they can be used as a simulation tool for design verification at various levels of abstraction supported by HDLs. Simulation becomes a very important aspect of VLSI design because of the lengthy manufacturing process [Fran81].

These advantages of high-level language approach to design of digital systems motivated the development of automatic synthesis tools [Shiv83, Thom79]. Automatic synthesis is the process of creation of a detailed design from an abstract specification. In other words, a synthesis tool transforms the high-level description of a system into an implementation. Although the results of current synthesis tools are relatively inefficient, when the design cost becomes comparable to the fabrication cost, a faster development cycle at the price of a less efficient implementation becomes an attractive alternative [Sequ83].

Considering the current complexity of digital systems, the design of a system without the use of a HDL (to specify the design) and the automatic synthesis tools (to handle various details of the design) becomes almost impossible. The objectives of this thesis is considered in the next section.

1.2 Objectives

The objectives of this work are two fold. First, we try to demonstrate that functional (applicative) languages in general, with certain enhancements can be used in various phases of design and that they are capable of handling the general class of systems and algorithms. Conceptually, the design process contains the following activities: [Sequ83]

a. functional design: guaranteeing proper behavior;

b. implementation: finding a suitable structure;

c. optimization: fine tuning of the physical arrangement.

The previous work in using a functional language for specification and simulation of hardware-oriented algorithms [Laht81] considered combinational algorithms only. The enhancements suggested and implemented in this work allow us to specify both sequential and combinational systems. Second, the main objective is to recognize the key features and characteristics of functional languages in design of high performance systems. There are many different approaches in organization of high performance digital systems. We consider specification, design and implementation of a class of pipelined system using a functional language as HDL.

Since the evolution of HDLs has been following the evolution of programming languages, both imperative and applicative languages have been used as HDLs. In the following two sections we introduce the differences between these two classes of languages to illustrate our motivation in use of functional languages as a HDL.

1.3 Imperative Languages

Imperative languages have been used extensively as HDLs. These languages are based on sequential execution of statements or instructions and the concept of updatable storage(variable). This concept has migrated to HDLs based on imperative languages. From a HDL point of view the problem is two fold:

First, almost all of HDLs (based on von Neumann model of computation) describe hardware activities among hardware resources [Marc79]. Typically, digital systems are modeled into resources like ALUs, MUXs, registers and others (these are equivalent to declaration of variable in the language). Then, a complex relation and control among these resources is described. This results in an unstructured design with very few mathematical properties in order to perform optimization. In many cases, algorithms are modified to meet the restrictions of the language [Laht81].

Second, in most cases the nature of the language does not allow expressing parallelism. If the language allows that, it becomes computationally unattractive to detect parallelism from the description. Also, because of existence of complex control relations in the language the HDL forces an unstructured design methodology, and mapping of high-level description or constructs to low-level implementation becomes non-trivial. Note, the general critique of the von Neumann model of computation can be found in [Back78, Laht81], and others.

Functional languages offer alternatives to conventional procedural languages and solutions to some of the problems inherent in conventional languages.

1.4 Functional Languages

Functional Programming languages (FP) are attractive as HDLs. Baden and Patel [Bade83] summarize the attractive features of the functional languages in the following:

1. They (functional languages) promote program brevity and encourage hierarchical program structure.

2. They express a model of execution that supports multiple, active centers of computations.

3. Their programs can be optimized through algebraic transformations.

From a HDL point of view, FP can be used to describe a hardware system organization as well as its behavior. That is, the organization of the digital system is known from its FP specification.

Optimization of a digital system for higher performance requires the "non-transparency" of the structure of the system. In most imperative HDLs the system specification is transformed to an explicit implementation before any optimization can be applied [Thom83].

FP provides a structured description of digital systems because the model of computation and control of system is based on the simple data-driven or demand-driven model. The generality of the combining forms in FP encourages hierarchical description which is a powerful abstraction mechanism. The emphasis of the language is on developing larger systems from smaller ones. Thus, both bottom-up and top-down design methodologies are supported.

The attractive feature of FP as a HDL is that FP describes the *implementation* of algorithms (we are concerned more about computation schemes rather than algorithms [Davi83]) explicitly. In other words, a FP program that describes the implementation of a particular computation scheme explicitly describes the precedence graph of precedence relations associated with the computation scheme. This allows an almost one-to-one relation between the high level description and the low-level implementation.

The mathematical nature of the functional language provides a promising base for design verification. The use of mathematical transformations allows optimization of the system based on cost criteria [Pate82].

Functional languages do not allow use of storage and history sensitivity. This disadvantage previously limited the use of functional languages to describe combinational systems only, although the sequential systems can be specified by modeling them with their iterative network equivalents.

Functional languages encourage hierarchical system design. This capability leads to design of systems in a purely hierarchical fashion where the problem of optimum design becomes critical [vanC79].

1.5 Outline

This work consist of two major parts. First, in chapter 2, a functional hardware description language (FHDL) based on Backus's FP [Back78] and Lahti's FP [Laht81] is introduced. The language is enhanced by adding the constructs to support the specification of sequential systems. This language can be used to specify the behavior of digital systems. The same specification is used to generate an implementation in the gate level automatically. Few examples are used to illustrate the capabilities of the approach.

Second, the above language and a synthesis tool are used to implement digital systems in pipelined fashion. This synthesis step, discussed in chapter 3, requires timing analysis of the digital system and automatic insertion of buffer stages (latches). The advantages and disadvantages of FHDL for generating a pipelined system is discussed. Some examples demonstrate the improvement of the pipelined system throughput by factor of two or three. The work is summarized in chapter 4.

In this work, UCLA FP interpreter [Pate84] has been modified to support sequential constructs. Based on this interpreter, a symbolic interpreter is developed to automatically generate an implementation associated with the specification. A graphic software system [Lu83] is used to display the result of the symbolic interpreter. This interpreter also supports the use of attributes in conjunction with functions. The time-delay, logic-level, and pipe-stage attributes are implemented and used to generate a pipelined implementation of the given specification.

CHAPTER 2

An Applicative (Functional) Hardware Description Language

2.1 Introduction

The design process of a digital system consists of three steps: specification of the system; implementation of the specification; and optimization of the implementation. These three steps more or less follow the three levels of hierarchy as discussed by van Cleemput [vanC79]. These three levels of hierarchy are: behavioral hierarchy, structural hierarchy, and physical hierarchy.

At the behavioral level, the user specifies the design or problem in a HDL. Then, a functional simulator is used to verify the correctness of the specification and design. The functional correctness achieved at the behavioral level is totally independent of the implementation, performance, or technology. The next stage of design is to implement the specification. The high-level constructs are transformed into low-level structures with the use of various synthesis steps and tools. Finally, the optimization is performed based on cost criteria of the implementation. This step is highly dependent on the idiosyncrasies of the technology and other design limitations.

The main objective of this chapter is to define a Functional Hardware Description Language (FHDL) based on the concept of functional languages. This language meets the need of the designers in two phases of the design.

First, the language provides an environment for specifying algorithms (hardware oriented algorithms) and a simulation tool for functional verification of the design. Second, a synthesis tool is provided to use the same specification to generate an implementation of the system at the gate level because the basic primitives currently supported by FHDL are logic gates.

The functional language under consideration is based on Backus's FP [Back78]. The use of functional language as a HDL is not new. Part of this approach has been explored previously by [Laht81, Fran81, Fran79, Card81, Gord81, Pate82] and others. The language extensions are logically following Lahti's work [Laht81]. Lahti used FP to describe hardware-oriented algorithms and structures of combinational systems and interconnection networks (which are combinational in nature). The main enhancements provide the functional primitives to describe sequential systems and allow the use of sequential systems with combinational ones.

This chapter is organized into three parts. First, the language itself is described, at the functional level. Second, the mapping of the language constructs to the implementation or structural level is introduced. This section serves as a formal specification of the symbolic interpreter. Finally, some programming examples illustrate the FHDL environment.

2.2 The Language

A digital system in FHDL is simply an expression representing a function that maps objects to objects. This can be simply be visualized as a function like adder that maps two numbers into one. The result is the sum of the two input numbers. Since the functions can only map objects to objects, the functions are memoryless and do not have side effects.

The main difference of FHDL from previous works (using functional languages as a HDL) is the capability to specify sequential machines or systems. In the combinational system the outputs are functions of only the present inputs. In the sequential system the outputs at any given time are functions of the inputs as well as of the stored information (or state) at that time. Sequential machines can be either synchronous or asynchronous. In synchronous sequential systems, the change of state takes place at discrete instants of time defined by clock pulses. Asynchronous systems do not force any restriction on changes of state. Synchronous sequential machines are particularly interesting because every finite output sequence that can be produced sequentially by a synchronous sequential machine can also be produced spatially by a combinational iterative network [Koha78, Erce82, Davi83].

Thus, every synchronous sequential system can be specified in FHDL using its combinational iterative network counter part. Figure 2.1 shows a general case of a sequential machine and its iterative network implementation. The behavior of the two systems(sequential machine and iterative network) is the same, but they differ with each other in implementation characteristics. The sequential machine is slower than the iterative network implementation, while the iterative network implementation is more expensive than its sequential counter part. The iterative network implements functions in space domain, since it generates the results spatially. The sequential machine implements functions in time domain, because it requires number of steps to generate the results.

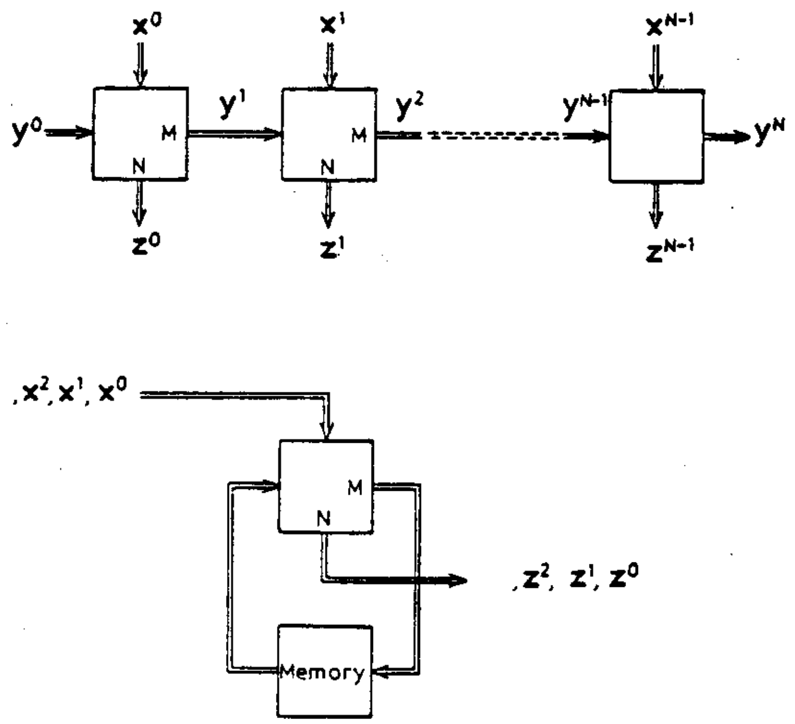


Figure 2.1 Iterative Network and Sequential System

At the behavioral domain we model a sequential system or machine specified in FHDL using sequential construct by an equivalent iterative network. In the behavioral level, FHDL only considers the space domain implementation of systems, while in the structural domain both space and time implementations are used. The formal specification of FHDL sequential constructs is provided in sections 2.2.3.1 and 2.2.3.2.

FHDL consists of five elements:

1. A set of objects, O.

2. A set of functions, F.

3. A set of functional forms, FF.

4. A set of definitions, D.

5. A single operation, application.

2.2.1 Objects

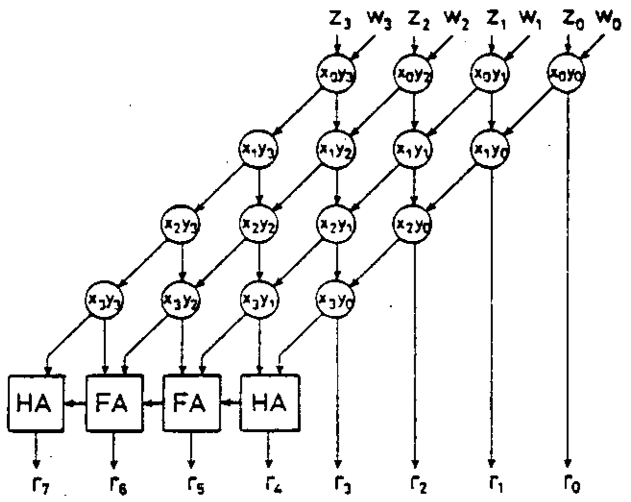
Objects may be atoms, or the "bottom" error symbol (?), or a sequence of objects. Sequence of objects are represented by enclosing the objects in a pair of parenthesis. The following are examples of objects.

() (1 0 1 0 0) 1 A HELLO 4 ? (1 0 (1 1 0) 0)

1 denotes true, and 0 is used as false. The designers can distinguish two types of objects or sequences. First, time objects, this is a sequence of objects that are inputed serially into the system or function (input to the sequential machine of Figure 2.1 is of this type). Second, space objects, the inputs to the carry (z's and w's) to the carry save multipler shown in figure 2.2 are space objects. Since, at the behavioral level, FHDL concerns with the space domain only, the time objects and space objects are treated similarly. In considering space objects no consideration of temporal order is required.

2.2.2 Functions

Functions are applied to objects to generate new objects. Most of the primitive functions described in FHDL are defined previously by Backus [Back78], Lahti [Laht81], and Baden [Bade83a]. The following is the summary of the primitives in FHDL; these definitions closely follow Berkeley's FP definitions [Bade83a].



z_3 w_3 z_2 w_2 z_1 w_1 z_0 w_0
x_0y_3 x_0y_2 x_0y_1 x_0y_0
x_1y_3 x_1y_2 x_1y_1 x_1y_0
x_2y_3 x_2y_2 x_2y_1 x_2y_0
x_3y_3 x_3y_2 x_3y_1 x_3y_0
HA
FA
FA
HA
r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0

Figure 2.2 Carry Save Multipler

n:x	Selects the n'th element of object x.
pick:(n,x)	Selects the n'th element of object x.
ext:(n,x)	Selects the first n elements of object x.
last:x	Returns the last object of x.
first:x	Returns the first object of x.
tl:x	Returns the tail of object x.
tlr:x	Returns the front of object x.
distl:(y,x)	Distributes y to all objects of x.
distr:(x,y)	Distributes y to all objects of x.
id:x	Returns the object x.
apndl:(y,x)	Appends y to left of x.
apndr:(x,y)	Appends y to right of x.
trans:x	Returns the transpose of x.

reverse:x	Returns the reverse of x.
rotl:x	Rotates x left by one element.
rotr:x	Rotates x right by one element.
concat:x	Concatenates all elements of x into one element.
pair:x	Pairs up the adjacent elements of object x.
split:x	Splits x into to two equal halves.
iota:x	Returns the first permutation of object x.
length:x	Returns the length of object x.
atom:x	Checks if the object x is an atom.
null:x	Checks if x is a null object.
+,-,*,/:x	Adds, subtracts, multiplies, and divides elements of x (x has two elements, see 2.2.6).
=,>,<, >=,<=:x	Returns true or false based on the predicate (x has two elements, see 2.2.6).
and,or,xor, nand,nor:x	Returns and, or, xor, nand, and nor of the elements of x (x has two elements, see 2.2.6).
~ :x	Returns the complement of x.

2.2.3 Functional Forms

Functional forms are applied to functions and return a new function. Functional forms can also be considered as combining forms of functions. That is, a number of functions are combined together in a particular way and result in a new function. Three classes of functional form are considered. First, a general class of functional forms is used describe algorithms in general. Second, a number of functional forms are provided for proper interfacing between combinational and sequential systems. The third class are the sequential functional forms. The following general functional forms, given informally, are available in FHDL:

a) Composition

```
(f@g):x  ==  (f:(g:x))
```

Note, "==" symbol means equivalent.

b) Construction

```
[f1,f2,...,fn]:x  ==  (f1:x,f2:x,...,fn:x)
```

c) Constant

```
%c:x  ==  c.
```

d) Conditional

```
(if   f
 then g
 else h
 fi):x  ==
```

If f:x is true then g:x is returned else h:x is returned.

Note: "if", "then", "else", and "fi" are FHDL keywords.

e) Right Insert

```
!f:x  ==  f:(x1,!f:(x2,...,xn))
where !f:x1  ==  x1 and !f:()  ==  ()
```

f) Left Insert

\f:x == f:(\f:(x1,...,xn-1),xn)

where \f:x1 == x1 and \f:() == ()

g) Tree Insert

|f:x == f:(|f:(x1,...,x[n/2]),

|f:(x[n/2+1],...,xn))

where |f:x1 == x1

2.2.3.1 Apply to All Functional Forms

Two apply to all functional forms are provided for interfacing the sequential functions to combinational ones and specifying algorithms in general. These forms are the same at the behavioral level, while they behave differently in the structural domain. This distinction is caused by the modeling the sequential systems with iterative networks in the behavioral domain. That corresponds to the earlier definition of time and space objects. The formal definition of the two apply to all functional forms is as follows.

a) Space Apply to All

&f:x == (f:x1,f:x2,...,f:xn),

where x is a space object.

Note: Space apply to all at the structural level will be mapped to n copies of function f. One can assume that x is a space object because the function "&f" operates on the input simultaneously. This form is the same as "apply to all" functional form defined by Backus [Back78] and Lahti [Laht81].

b) Time Apply to All

$f:x == (f:x1,f:x2,...,f:xn),

where x is a time object.

Note: Time apply to all at the structural level will be mapped to one copy of function f. Input x is a time object because the implementation of "$f" assume that elements of input are applied to f one at a time sequentially. In other words, "x1" is the input at time "t1", x2 is the input at time "t2", and "xn" is the input at time "tn".

2.2.3.2 Sequential Functional Forms

The abstract model of a synchronous sequential system is a finite state machine shown in Figure 2.3. As illustrated in figure, "x" is the input and "z" is the output of the machine, while "y" is the present state and "Y" is the next state. There are many variations in specifying finite state machines. The following three functional forms are provided as the well known implementation of finite state machine. These systems are studied in detail by Davio et. al. [Davi83], Ercegovac and Lang [Erce82], and others. The previous argument used to distinguish between time and space domain implementation applied directly to these functional forms. All the following functional forms are implemented in time domain and applied to time objects.

a) Sequence

(**seq** f **seqfunc** g **seqend**):x ==

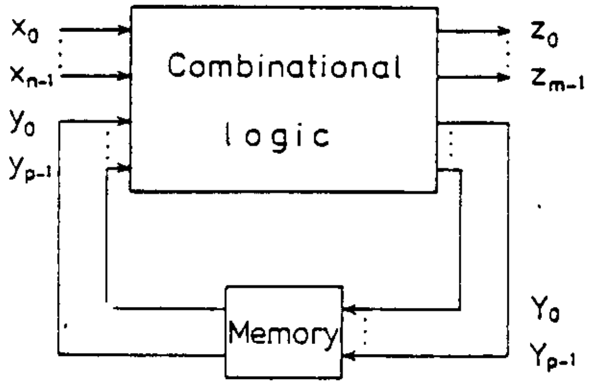


Figure 2.3 Finite State Machine

(g:(f:x,x1),g:(g:(f:x,x1),x2),...),

where x is a time object.

Note: "**seq**", "**seqfunc**", and "**seqend**" are language keywords and act as delimiters. "f" and "g" can be any function. Sequence is a variant of finite state machine(shown in Figure 2.3) where the output vector z is the same as the next state "Y"(see Figure 2.4). Function "f" provides an initial value for state register.

b) Mealy

(**mealy** f **meout** h **menext** g **meend**):x ==

(h:(f:x,x1),h:(g:(f:x,x1),x2),

h:(g:(g:(f:x,x1),x2),x3),...)

where x is a time object.

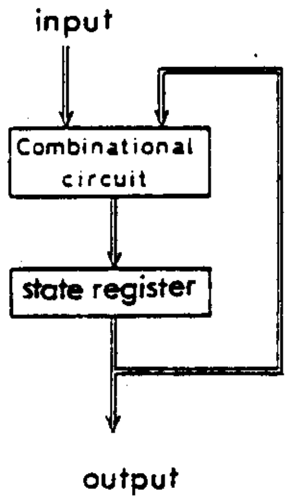


Figure 2.4 Sequence Functional Block Diagram

Note: "**mealy**", "**meout**", "**menext**", and "**meend**" are FHDL keywords and act as delimiters. "f", "h", "g" can be any function. In Mealy machine, the output depends on the present state and input. Function "f" provides an initial value for state register. Function "h" generates output. Function "g" generates next state.

c) Moore

(**moore** f **moout** h **monext** g **moend**):x ==

(h:(f:x),h:(g:(f:x,x1)),h:(g:(g:(f:x,x1),x2)),...)

where x is a time object.

Note: "**moore**", "**moout**", "**monext**", and "**moend**" are FHDL keywords and act as delimiters. "f", "h", and "g" can be any function. In Moore machine the output depends on the present state only. Function "f" provides an initial value for state register. Function "h" generates output. Function "g" generates next state.

The use of the above functional forms is demonstrated in section 2.4.2 and 2.4.3.

2.2.4 Definitions

Construction of larger functions or programs can be facilitated by the use of definitions. A definition is a naming of a function which is constructed out of other functions or functional forms. Whenever the name is encountered the equivalent functions are expanded before the execution proceeds. This process is similar to macro definition and expansion. The syntax of definition is as follows:

```
defun function-name (parameter list)
        FHDL expression
enddef
```

For example a simple carry save cell function(program) of Figure 2.2 can be defined as following:

```
defun CarrySaveCell ()      # functional description
                        # of a cell of figure 2.2
    if (length = %2)        # check the length to see
                        # to generate what kind of module
    then [id,id] @      # and the two inputs and
         (1 and 2)      # make to copies of result
    else id             # generate single wire to
                        # the next level
```

```
        fi
    enddef
```

A null parameter list means unnamed parameters are passed to the function. "#" designates the beginning of comment.

2.2.5 Application

The application ":" is used to apply a function to an object. For example, CarrySaveCell:(1 0) will return (0 0).

2.2.6 Syntax Variation

Original FP defined by Backus uses prefix syntax notation. For example, "and" function that operates on two inputs must be expressed as following:

```
and @ [1,2]
```

The prefix notation makes functional programs hard to read. FHDL allows both prefix and infix notations. Thus, the following two expressions have exactly the same effect.

```
(1 and 2)  ==  and @ [1,2]
```

2.3 Implementation of Hardware Algorithms in FHDL

The language described in the previous section is used to specify, design and simulate hardware algorithms at the functional level. Once an algorithm is verified functionally, the designer has to implement the algorithm. The objective of this section is to specify the mapping from the behavioral domain to structural domain according to the FHDL specification.

One of the motivations for using a functional language as a HDL is that the transformation from the behavioral level to structural level is one-to-one. Since, we are considering the implementation of digital systems many new issues will surface. The designer may be interested to know about various physical parameters of the implemented system such as propagation delay and power dissipation. These physical parameters or system attributes solely depend on the characteristics of functions. That is, one can execute FHDL functions and, as the objects are being transformed, the parameters or attributes of the system can be updated. This directly impacts how the functions behave in structural domain and how the information about the system attributes is handled.

The functions described in section 2.2.2 can be divided into four categories. First, there are the basic functions such as "and", "or", and "not". Second category contains basic interconnection functions such as "select" and "distribute left". Third category consists of predicates such as "equal", "greater than", and conditional functional form. Fourth category contains functions that are used to simplify description of algorithms. Examples are "length" or "iota". The first two classes of functions and functional forms are very important because by performing symbolic execution or interpretation of the specification, a logic diagram or computation graph is obtained. Logic diagram is a graph or diagram where the nodes represent gates and interconnection between the gates are shown by the arcs. Functions in the symbolic interpreter operate on symbols rather than values (the operation of functions on system attributes is considered later). The implementation of FHDL expressions cannot be done directly from the specification, because the structure organization of objects cannot be extracted from the specification. For example, the description of a multiplexer can be used for any size multiplexer. That is, a 16 bit or 32 bit multiplexer has the same description, and for the actual implementation the object must be known.

The third class functions (predicates) usually have dual purpose. That is, they are used sometimes as simple language construct to manage the flow of control and ease the description of algorithms, while in other instances a predicate is mapped directly to low level implementation. For example, the conditional functional form used in description of carry save multiplier cell in section 2.2.4 is used to control the type of structure is generated by the function. Considering different uses of conditional constructs the following strategy is adopted in the symbolic interpreter. Conditional constructs and predicates are used to control the flow of data and ease of algorithm description solely. Since the existence of conditional constructs in the programming languages is unnecessary [Fran79], the system specifications that uses conditional constructs to be mapped to low level implementation can use other functions to achieve the same goal.

The fourth class of functions are not mapped to lower level, and there exists no simple mapping either.

The above classification is basically used to perform symbolic interpretation or execution of FHDL. Also, the impact of design methodology proposed for handling predicates becomes obvious in the implementation phase. Next few sections describe the transformation of FHDL specification from behavioral domain to structural domain.

2.3.1 Objects

Objects in behavioral domain carry values only and functions operate on values to generate new values. In the symbolic domain, objects carry both symbols and values. In this domain, functions operate on symbols to generate new symbols, except in the case of predicates. Since predicates are used to control the flow of data, they operate on values rather than symbols. Thus, each atom must contain a symbol and a value. In order to provide useful design statistics, various attributes is needed to be passed along with each object and each function must update these attributes depending on the characteristic of the function [Erce83].

In general, an atom in symbolic environment has the following form:

(symbol-or-name value optional-list-of-attribute-values)

Currently, the symbolic interpreter supports three types of attributes: the propagation delay(D), the number of logic levels(L), and the number of pipeline stages(S). For example, atom "(MUXIN1 1 25 3 0)" can be interpreted as a wire or connection called "MUXIN1" with value of 1. The delay of the corresponding signals is 25 units of time and the signal has passed through 3 levels of logic. Signal "MUXIN1" has been through 0 stage(s) of the pipe(Chapter 3 provides the details). It is important to note that a predicate like "atom" returns true token "(DUMMY 1 0 0 0)" when applied to an object like "(MUXIN1 1 10 2 0)", since this object is not an atom in the behavioral domain.

2.3.2 Functions

Functions in the behavioral domain operate on values. In structural domain, they must operate on symbols, values and attributes. For example, "or" function applied to "((IN1 1 5 1 0)(IN2 0 9 2 0))" will result an atom "(WIRE.01 1 18 3 0)". Following the previous discussion on the different categories of functions, we now consider three classes of functions. First, the functions that perform basic boolean operations are as follows: "and", "or", "xor", "nand", "nor", and "not(˜)". These are mapped directly to the corresponding logic gate. Second, the class of functions that are for ease of describing algorithms consist of: "iota", "length", "atom", "null", "+", "-", "*", "/" and predicates. The rest of the functions defined in section 2.2.2 belong to the category of interconnection functions.

The boolean functions operate on time delay and logic level attributes. The pipe-stage attribute is updated by latches and this operation in illustrated in Chapter 3. The following algorithm is used for updating time delay and logic level attributes.

$$\text{Dout} = \max(\text{D1},\text{D2}) + \text{Df} \qquad (2.1)$$

$$\text{Lout} = \max(\text{L1},\text{L2}) + 1 \qquad (2.2)$$

Where "D" stands for time delay and "L" for logic level. The equations are interpreted in the worst-case sense: the time delay attribute of the output signal is equal to maximum time delay of inputs plus the time delay of the function or gate. The logic level attribute of the output signal is equal to maximum logic level attributes of inputs plus one. Note, for inverter gate of "not(˜)" function the attribute of input is used instead of finding the maximum, since it has only one input. The following is the output of the symbolic interpreter for a half-adder which illustrates how the boolean functions operate on attributes.

```
> 1 HALFADDER.841
> 1 ((A0 1 0 0 0) (B1 1 0 0 0))
> 2 AND.842
> 2 (A0 1 0 0 0)(B1 1 0 0 0)
> 2 (WIRE.843 1 9 1 0)
> 2 XOR.844
> 2 (A0 1 0 0 0)(B1 1 0 0 0)
> 2 (WIRE.845 0 16 1 0)
> 1 ((WIRE.843 1 9 1 0) (WIRE.845 0 16 1 0))
```

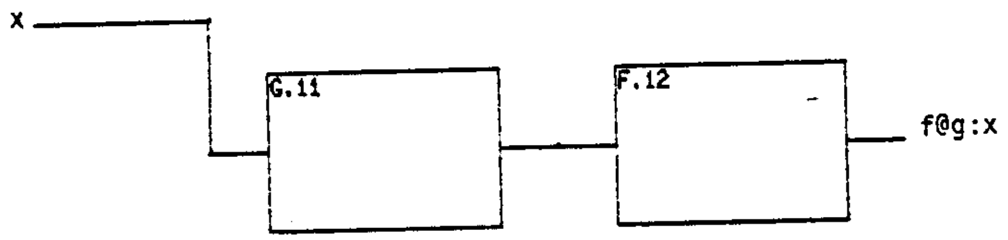
The number following ">" is the level of nested function calls. Each function occupies three lines. The first line has the function name with a unique number appended at the end, to make the particular function unique in the whole output. The second line list the inputs, and the third line has the list of outputs. The function "halfadder.841" has a worst case propagation delay of 16 units of time and has one logic level. The gate delay of "and" primitive is 9 units of time and the gate delay of "xor" gate is 16 units of delay (Appendix A has the details of modifying these values).

2.3.3 Functional Forms

The following is the implementation of functional forms of FHDL in structural domain. The structure shown for each functional form is generated by the symbolic interpreter and a graphic package using the output of the interpreter to provide a visual result, (Appendix A provides further details). The label in each box is the unique label generated by the interpreter for each function.

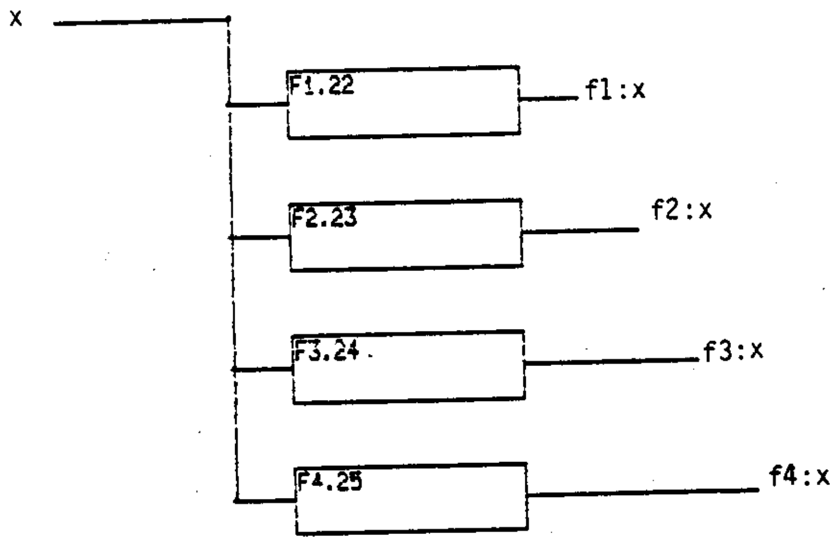
a) Composition

f @ g : x



b) Construction

[f1,f2,f3,f4] : x

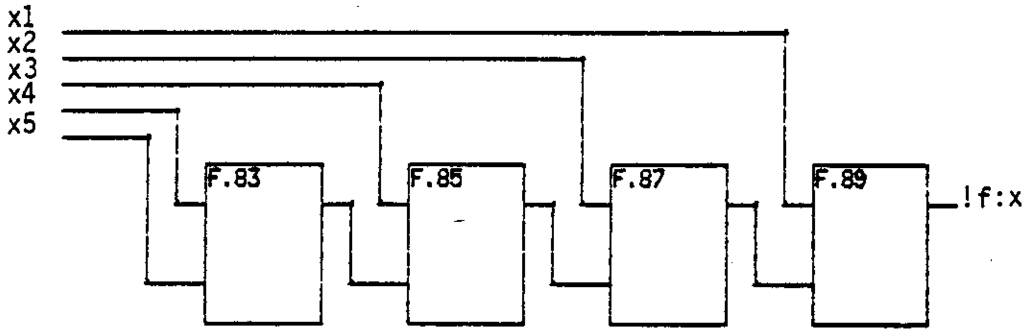


c) Constant

$$\%c{:}x \;==\; (\text{name } x\ 0\ 0\ 0)$$

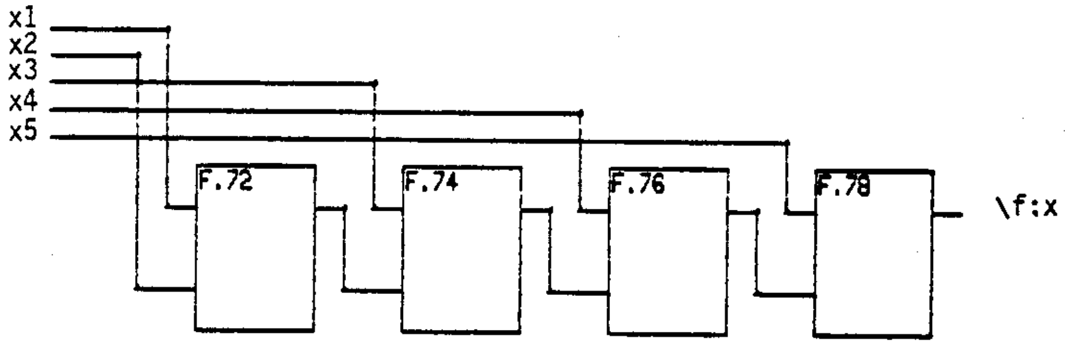
d) Right Insert

!f :x where x = (x1 x2 x3 x4 x5)



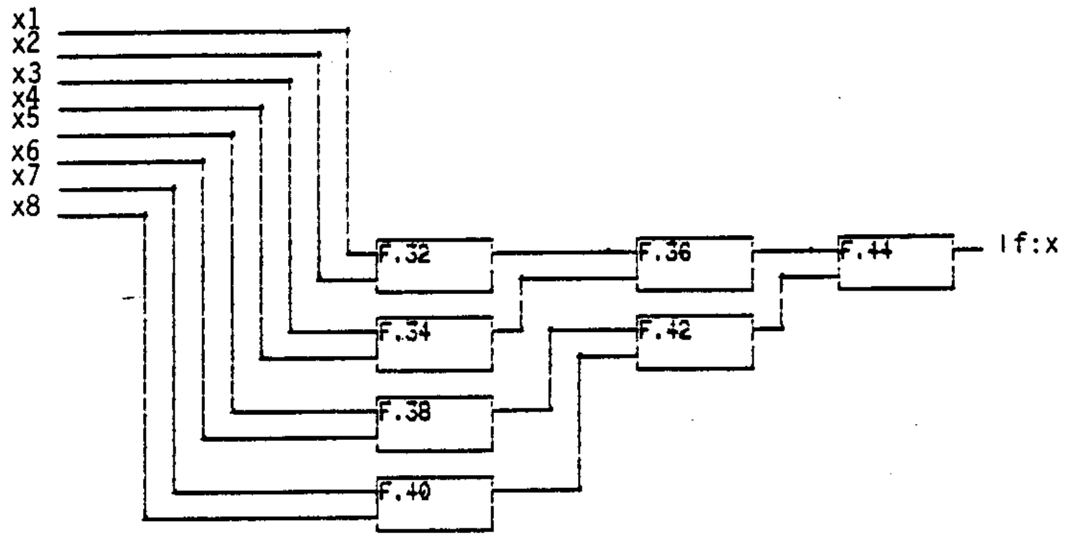
e) Left Insert

\f : x where x = (x1 x2 x3 x4 x5)



f) Tree Insert

|f : x where x = (x1 x2 x3 x4 x5 x6 x7 x8)

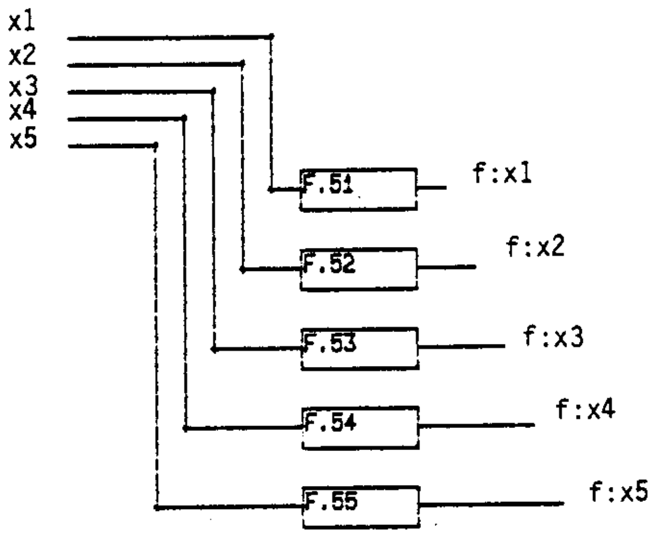


2.3.3.1 Apply to All Functional Forms

a) Space Apply to All

The behavior of "space apply to all" in structural domain is the same as in the behavioral domain. As shown below, the "space apply to all" provides simultaneous operation on space objects and instantaneously generate the result. This functional form is exactly the same as "apply to all" defined by Lahti [Laht81] and Backus [Back78], so it can be used to specify algorithms in general.

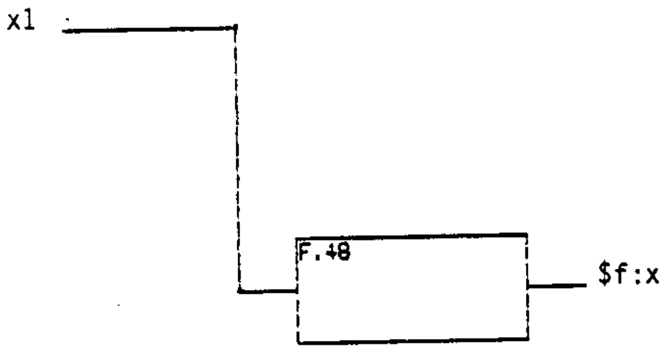
&f : x where x = (x1 x2 x3 x4 x5)



b) Time Apply to All

The "time apply to all" functional form only operates on one element of input (the symbolic interpreter uses the first element of input) because the inputs are processed one-at-a-time in the structural domain. "Time apply to all" assumes that its input is generated by a sequential systems or its output is used by a sequential system. Thus in the actual implementation the input is generated one at a time.

$f : x where x = (x1 x2 x3 x4 x5)

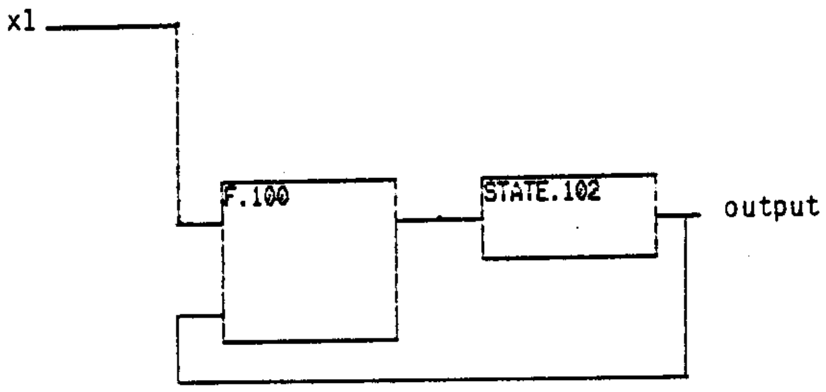


2.3.3.2 Sequential Functional Forms

Considering the previous introduction of sequential functional form in section 2.2.3.2, the following is the implementation of these in structural domain. "finit", "f", "g" can be any function.

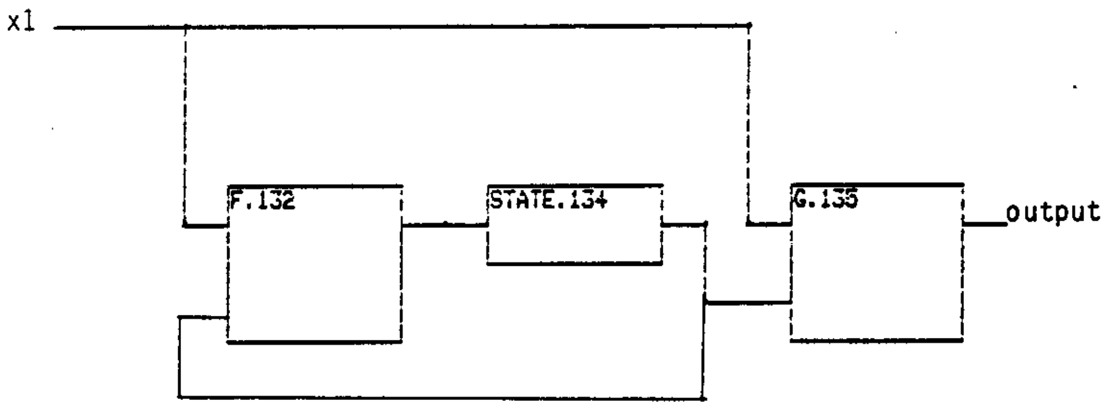
a) Sequence

(**seq** finit **seqfunc** f **seqend**) : x



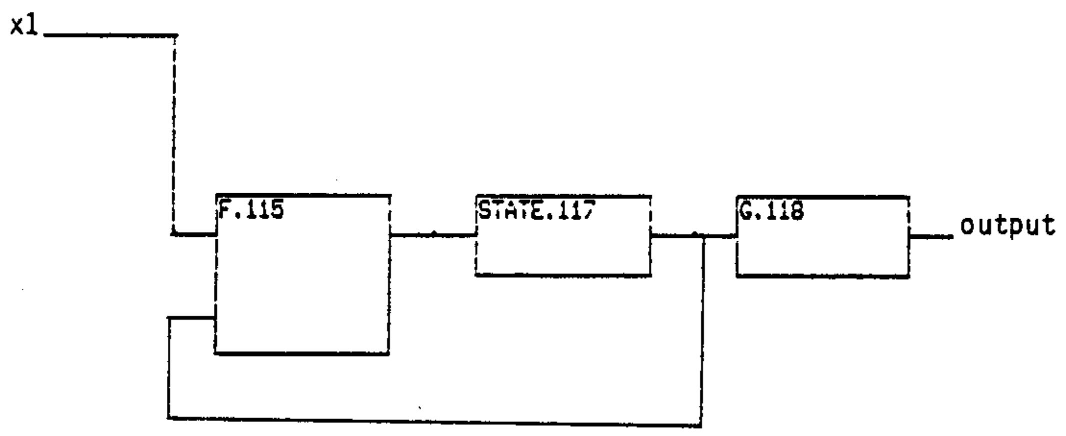
b) Mealy

(**mealy** finit **meout** g **menext** f **meend**) : x



c) Moore

(**moore** finit **moout** g **monext** f **moend**) : x



The use of the above functional forms and apply to all functional forms is illustrated by mean of some examples in the next section.

2.4 Examples

Many combinational systems as well as interconnection networks have been described by Lahti using FP [Laht81]. A number of examples of hardware systems and algorithms are provided in this part. The purpose of these examples is four fold. First, these examples introduce the variation of FHDL from the previous work by Lahti [Laht81] or Baden [Bade83a]. Second, a number of examples demonstrate the new enhancements of FHDL. That is, the use of sequential constructs to describe sequential systems, illustrating the implementation of an algorithm in both the time and space domains. Third, these examples can illustrate the capabilities of the symbolic interpreter for generating gate level diagram and computing system attributes. Forth, a number of these examples are optimized for higher performance in chapter 3 using a synthesis tool.

In the following sections: First, we use an n-input conditional-sum adder [Skla60, Hwan79] as a typical combinational system. A one-bit serial adder is considered next. Four different specifications associated with this function is provided. This example illustrates the use of various sequential constructs. In sequential domain, the implementation of a multiple operand addition using carry-save adder and carry propagation adder complex is also studied. Note, in all the following examples, the pipe-stage attribute is not used because the system is not pipelined. Sections 3.4 and 3.5 describe the use of pipe-stage attribute.

2.4.1 Conditional Sum Adder

FHDL description of a conditional-sum adder [Skla60] is given below.

```
defun hadder()             # (Ci+1 Si)
        [and,xor]
enddef                # (Ai Bi)

defun chadder ()      # ( Ci+1[1] Si[1] Ci+1[0] Si[0] )
                      # conditional half-adder
        [[or,~ @xor],[and,xor]]
enddef                # (Ai Bi)


defun mux2tolv ()     # ((X0) ... (Xn)) if cond
                      # ((Y0) ... (Yn)) if not cond
        &or @ trans @ &&and @ &distl @
       trans @ [[id,~ ]@1,2]
enddef                # (cond ((X0 ... Xn)(Y0 ... Yn))

defun picknext ()     # (X0 ... Xn) if cond
                      # (Y0 ... Yn) if not cond
       concat @[mux2tolv@[1@2,1],tl@2]
enddef                # ((X0...Xn)(Y0...Yn)(cond dummy))

defun formboth ()     # n/2 partial result feed to last
                      # MUX
       if (length = %1)
       then chadder @ 1
```

```
        else &picknext @
             [[1,1@2],[1,2@2]]@
             &formboth@split
        fi
enddef                  # ((An Bn)(An-1 Bn-1)...(An/2 Bn/2))

defun csumadd ()    # (Cout Sn Sn-1 ... S0)
        if (length = %1)
        then hadder @ 1
         else picknext @        # last MUX in the adder
            [formboth@1,csumadd@2] @
             split
        fi
enddef                  #((An Bn)(An-1 Bn-1)...(A0 B0))

defun condsumadder ()       # (Cout Sn Sn-1 ... S0)
        csumadd @ trans
enddef                  # ((An An-1 ... A0)(Bn Bn-1 ... B0))
```

The above description is generic in a sense that it describes any size conditional-sum adder. This example demonstrates the basic style of specifying hardware algorithms in FHDL. Figure 2.5 shows logic schematic of a 16 bit adder in order to make the reader familiar with the algorithm. Figure 2.6 illustrates logic schematic generated by the symbolic interpreter using the graphic interface software for a four-bit conditional sum adder (Appendix A provides further details). Each box in the graph is associated with a logic gate; "X" stands for "xor" gate, "O" stands for "OR" gate, "A"

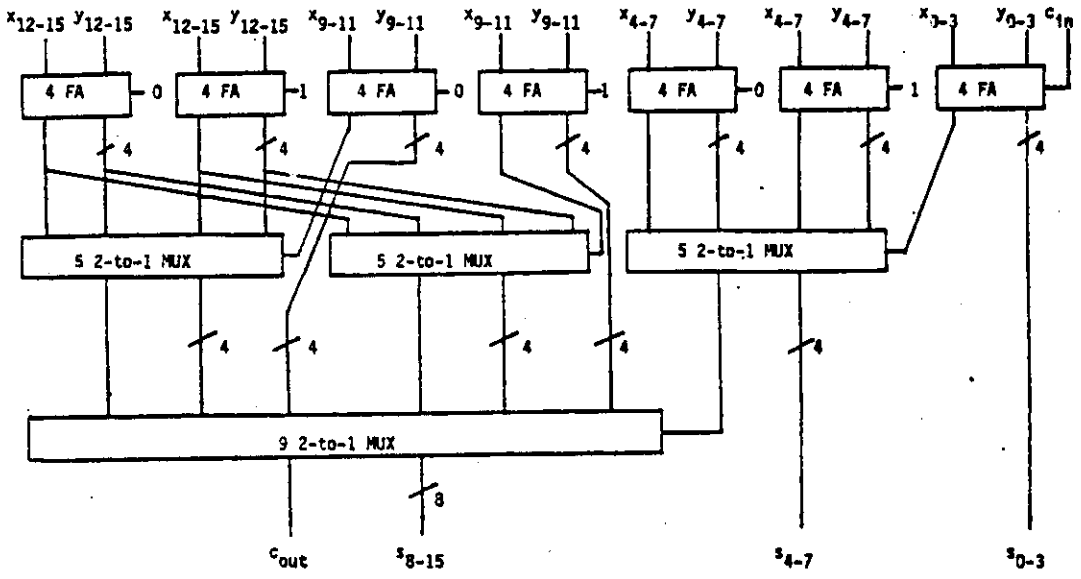


Figure 2.5 A 16-bit Conditional-Sum Adder

denotes "and" gate, and "N" stands for inverter. The result of a sample simulation of the four-bit adder is given below.

```
->> condsumadder:((1 0 0 1)(0 1 1 0))

(0 1 1 1 1)
```

The following is a part of the symbolic interpreter output for four-bit conditional sum adder.

```
> 1 CONDSUMADDER.228
> 1 (((IN11 1 0 0 0) (IN12 1 0 0 0) (IN13 1 0 0 0) (IN14 1 0 0 0))
```

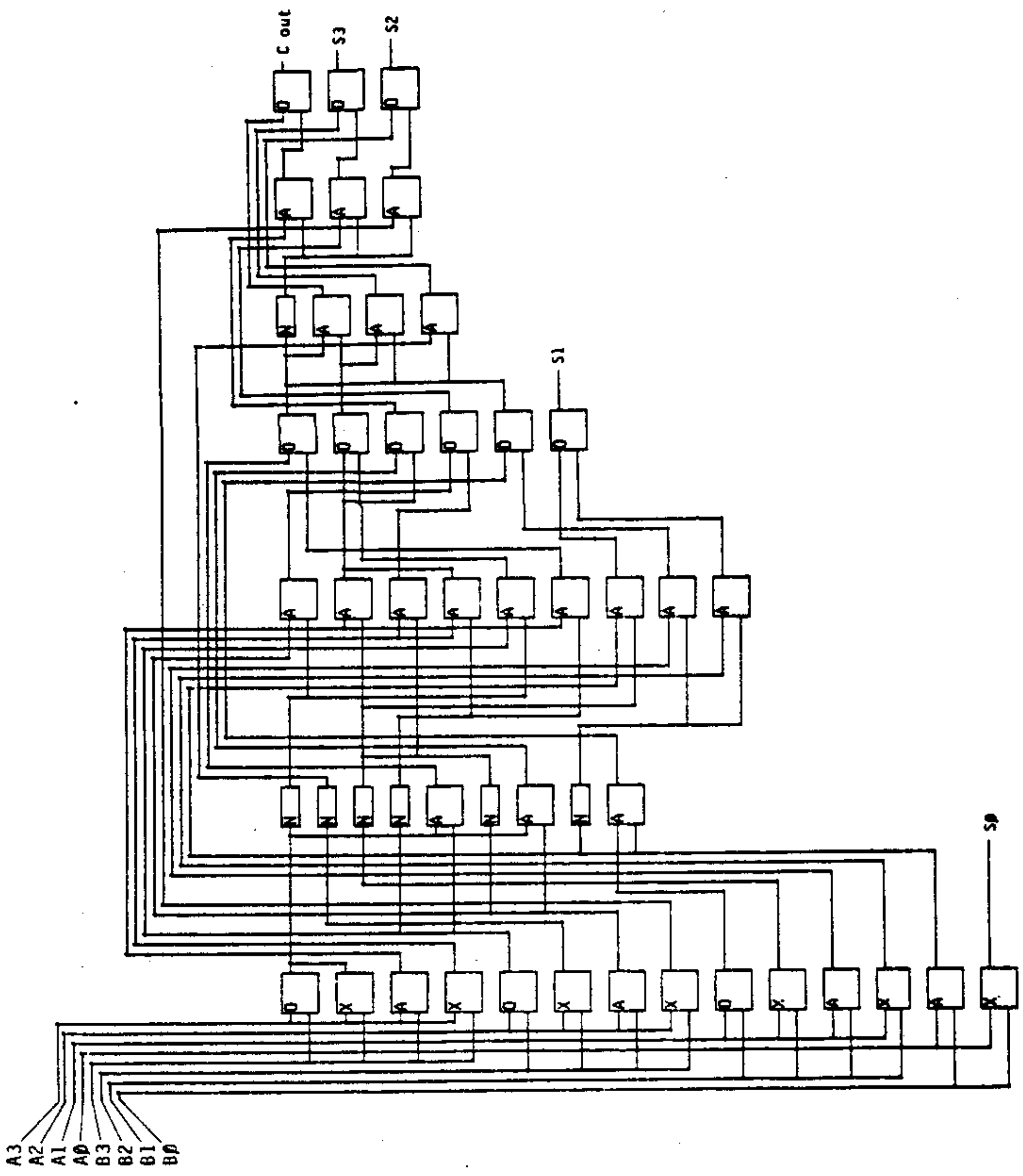


Figure 2.6 Graph of a 4-bit Conditional-Sum Adder

```
((IN21 1 0 0 0) (IN22 1 0 0 0) (IN23 1 0 0 0) (IN24 1 0 0 0)))
> 1 ((WIRE.341 1 53 7 0) (WIRE.343 1 56 7 0) (WIRE.345 1 53 7 0)
(WIRE.325 1 38 4 0) (WIRE.311 0 16 1 0))
```

The four-bit conditional-sum adder has the worst case propagation delay of 53 units of time and seven gate levels. Figure 2.7 shows a 6 bit conditional sum adder that is generated for the same specification. The result of functional and symbolic simulation of six-bit conditional sum adder is given below.

```
->> condsumadder:((1 1 1 1 1 1)(1 1 1 1 1 1))

(1 1 1 1 1 1 0)


> 1 CONDSUMADDER.348
> 1 (((IN11 1 0 0 0) (IN12 1 0 0 0) (IN13 1 0 0 0) (IN14 1 0 0 0)
(IN15 1 0 0 0) (IN16 1 0 0 0)) ((IN21 1 0 0 0) (IN22 1 0 0 0)
(IN23 1 0 0 0) (IN24 1 0 0 0) (IN25 1 0 0 0) (IN26 1 0 0 0)))
> 1 ((WIRE.571 1 71 9 0) (WIRE.573 1 74 9 0) (WIRE.575 1 71 9 0)
(WIRE.577 1 71 9 0) (WIRE.549 1 56 6 0) (WIRE.551 1 38 4 0)
(WIRE.531 0 16 1 0))
```

The propagation delay of 6 input conditional-sum adder is 71 units of time with nine gate levels. In the next section an example of a sequential and combinational system is studied.

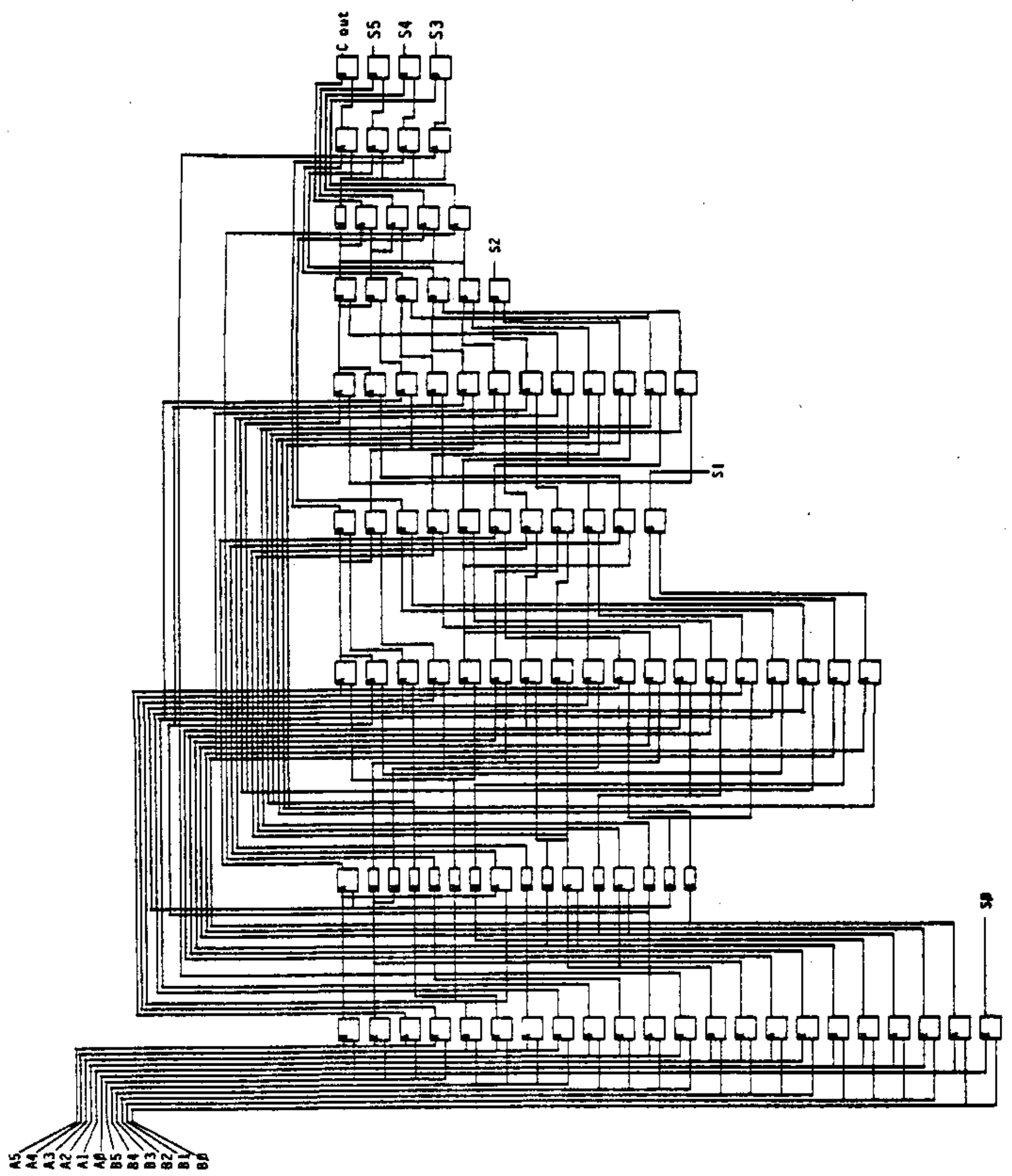


Figure 2.7 Graph of a 6-bit Conditional Sum Adder

2.4.2 One-Bit Sequential Adder

Consider first the design of a carry-propagation adder (CPA) as an iterative network. Figure 2.8 shows the block diagram of an n-bit CPA.

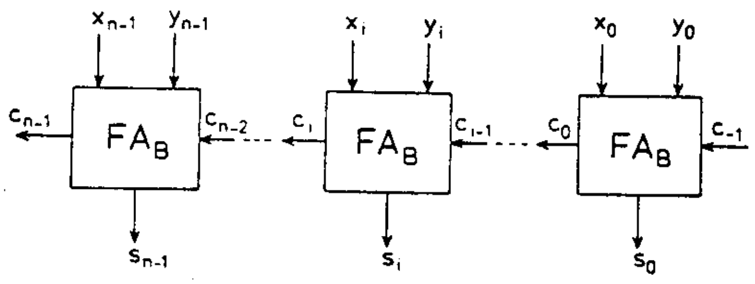


Figure 2.8 n-bit CPA

CPA can be implemented in many different ways. The iterative (space domain) implementation that corresponds to the block diagram of Figure 2.8 is as follows:

```
# Ripple Carry Adder or Carry Propagate Adder (CPA)

defun cpa0 ()         # ((S0 ... Sn-1) Cn-1)
    [&2,1@last] @     # ((S0 ... Sn-1) Cn-1)
    tl @              # ((C0 S0)...(Cn-1 Sn-1))
    \ (apndr @        # ((C-1 0)(C0 S0)...(Cn-1 Sn-1))
       [1,fulladder @
          apndl @ [1@
                last@
                 1,2]]
      ) @
    apndl @           # (((C-1 0))(A0 B0)...(An-1 Bn-1))
```

```
        [[[1,%0]],tl]
enddef                 # (C-1 (A0 B0)...(An-1 Bn-1))
```

The full-adder has the following specification.

```
defun fulladder ()           # OUTPUT (C1 S1)
        [(1 or 2),3] @
       apndl @ [1,halfadder@[2,3]] @
        apndr @ [halfadder@[2,3],1]
enddef                        # INPUT (C0 A1 B1)

defun halfadder ()   # OUTPUT (Ci Si)
        [and,xor]
enddef                 # INPUT  (Ai Bi)
```

Figure 2.9 shows the logic diagram generated from the output of the symbolic interpreter.

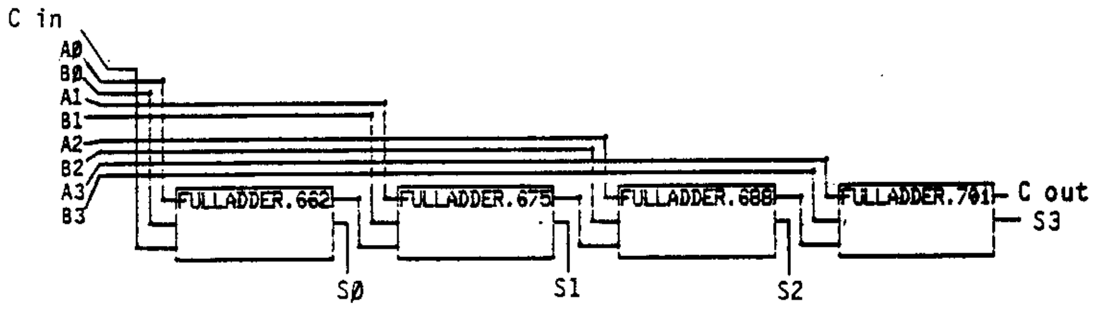


Figure 2.9 CPA Implementation Using Iterative Networks

In the structural domain, the above specification of CPA is implemented using an iterative network, since it does not use any sequential constructs. The result of functional and symbolic execution of "cpa0" specification is given below.

```
->> cpa0:(0 (1 1)(1 1)(1 1)(1 1)(1 1)(1 1))

((0 1 1 1 1 1) 1)


> 1 CPA0.660
> 1 ((CIN 0 0 0 0) ((A0 1 0 0 0) (B1 1 0 0 0))
((A1 1 0 0 0) (B1 1 0 0 0)) ((A2 1 0 0 0) (B2 1 0 0 0))
((A3 1 0 0 0) (B3 1 0 0 0)))

> 1 (((WIRE.672 0 32 2 0) (WIRE.685 1 50 4 0)
(WIRE.698 1 68 6 0) (WIRE.711 1 86 8 0)) (WIRE.713 1 88 9 0))
```

The propagation delay of "cpa0" is 88 units of time with nine gate levels.

Now, let us use "sequence" functional form to implement the same function. The specification is as follows:

```
# 1 bit at a time adder
# Sequence Functional Form

defun seqadder ()      # OUTPUT ((C0 S0)...(Cn-1 Sn-1))
        seq [%0,%0]    # C-1 = 0
        seqfunc fulladder @
                apndl @ [1@1,2]
        seqend
enddef                 # INPUT ((A0 B0)...(An-1 Bn-1))

defun cpa1 ()          # OUTPUT ((S0 S1 ... Sn-1) Cn-1)
```

```
        [$2,          # select Sums
         1@last] @    # select Cn-1
        seqadder
enddef                # INPUT ((A0 B0)...(An-1 Bn-1))
```

Notice, the use of "time apply to all" in function "cpal" for selecting the sum bits generated by the sequential adder. The counter part of this is the use of "space apply to all" in function "cpa0". Figure 2.10 is the logic diagram of function "cpal".

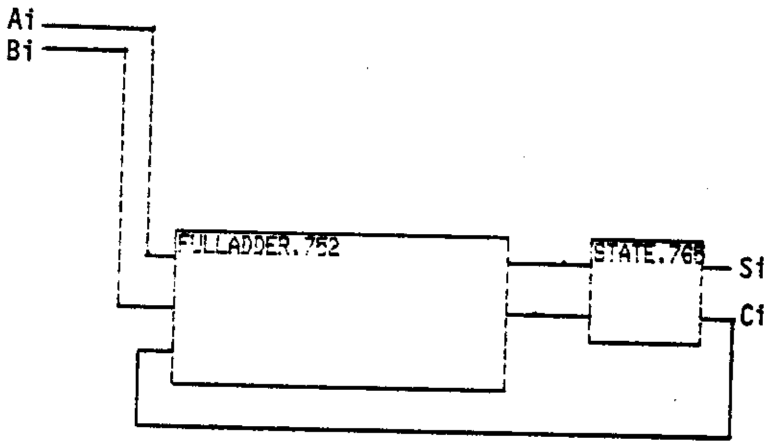


Figure 2.10 CPA Implementation Using Sequence Construct

The following is the result of functional and symbolic simulation of function "cpal".

```
->> cpal:((1 1)(1 1)(1 1)(1 1)(1 1)(1 1))
```

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


> 1 CPA1.748
> 1 (((A0 1 0 0 0) (B1 1 0 0 0)) ((A1 1 10 0 0) (B1 1 10 0 0))
((A2 1 20 0 0) (B2 1 20 0 0)) ((A3 1 30 0 0) (B3 1 30 0 0)))
> 2 SEQADDER.749
> 2 (((A0 1 0 0 0) (B1 1 0 0 0)) ((A1 1 10 0 0) (B1 1 10 0 0))
((A2 1 20 0 0) (B2 1 20 0 0)) ((A3 1 30 0 0) (B3 1 30 0 0)))
> 3 FULLADDER.752
> 3 ((CONST.750 0 0 0 0) (A0 1 0 0 0) (B1 1 0 0 0))
> 3 ((WIRE.764 1 34 3 0) (WIRE.762 0 32 2 0))
> 3 STATE.765
> 3 ((WIRE.764 1 34 3 0) (WIRE.762 0 32 2 0))
> 3 ((CONST.750 0 0 0 0) (CONST.751 0 0 0 0))
> 2 ((CONST.750 0 0 0 0) (CONST.751 0 0 0 0))
> 1 ((CONST.750 0 0 0 0) (CONST.751 0 0 0 0))
```

The worst case propagation delay of the full adder is 34 units of time. The clock period of the above adder is 34 units of time plus whatever time is required for the register or latch (used as state register) to stabilize itself.

The following FHDL program performs the same function using a Moore finite state machine.

```
# 1 bit at a time adder using
# Moore Functional Form
```

```
defun mooreadder ()         # OUTPUT ((C0 S0)...(Cn-1 Sn-1))
    moore [%0,%0] # C-1 = 0
    moout id        # pass Carry and Sum
    monext fulladder @
          apndl @
          [1@1,2]
    moend
enddef                  # ((A0 B0)(A1 B1)...(An-1 Bn-1))

defun cpa2 ()           # OUTPUT ((S0 S1 ... Sn-1) Cn-1)
      [$2,              # select Sums
       1@last] @        # select Cn-1
      mooreadder
enddef                  # INPUT ((A0 B0)...(An-1 Bn-1))
```

"Time apply to all" is used for interfacing the output of a sequential system to a combinational one ("select" in function "cpa2"). The logic diagram of function "cpa2" is shown in Figure 2.11. A sample execution of function "cpa2" in the behavioral and structural domain is shown below.

```
->> cpa2:((1 1)(1 1)(1 1)(1 1)(1 1)(1 1))

((0 1 1 1 1) 1)


> 1 CPA2.792
> 1 (((A0 1 0 0 0) (B1 1 0 0 0)) ((A1 1 10 0 0) (B1 1 10 0 0))
```

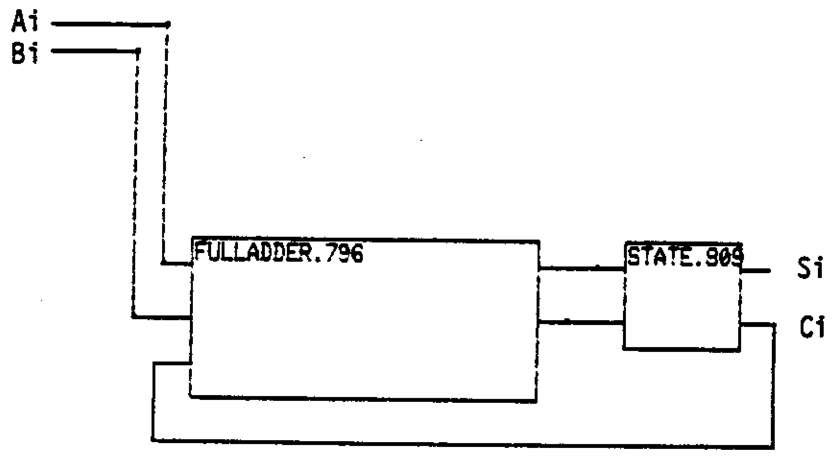


Figure 2.11 CPA Implementation Using Moore Construct

```
((A2 1 20 0 0) (B2 1 20 0 0)) ((A3 1 30 0 0) (B3 1 30 0 0)))
> 2 MOOREADDER.793
> 2 (((A0 1 0 0 0) (B1 1 0 0 0)) ((A1 1 10 0 0) (B1 1 10 0 0))
((A2 1 20 0 0) (B2 1 20 0 0)) ((A3 1 30 0 0) (B3 1 30 0 0)))
> 3 FULLADDER.796
> 3 ((CONST.794 0 0 0 0) (A0 1 0 0 0) (B1 1 0 0 0))
> 3 ((WIRE.808 1 34 3 0) (WIRE.806 0 32 2 0))
> 3 STATE.809
> 3 ((WIRE.808 1 34 3 0) (WIRE.806 0 32 2 0))
> 3 ((CONST.794 0 0 0 0) (CONST.795 0 0 0 0))
> 2 ((CONST.794 0 0 0 0) (CONST.795 0 0 0 0))
> 1 ((CONST.794 0 0 0 0) (CONST.795 0 0 0 0))
```

The system throughput (clock rate) as for function "cpa1" is achieved.

The following program describes the same function with use of the mealy functional form.

```
# 1 bit at a time adder using
# Mealy Functional Form

defun mealyadder ()          # OUTPUT ((C0 S0)...(Cn-1 Sn-1))
    mealy [%0,%0] # C-1 = 0
    meout 1          # pass Carry and Sum
    menext fulladder @
           apndl @
           [1@1,2]
    meend
enddef                   # ((A0 B0)(A1 B1) ... (An-1 Bn-1))

defun cpa3 ()            # OUTPUT ((S0 S1 ... Sn-1) Cn-1)
      [$2,               # select Sums
       1@last] @         # select Cn-1
      mealyadder
enddef                   # INPUT ((A0 B0)...(An-1 Bn-1))
```

Figure 2.12 is the logic diagram of function "cpa3". The functional and symbolic execution of function "cpa3" is given below.

```
->> cpa3:((1 1)(1 1)(1 1)(1 1)(1 1)(1 1))

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

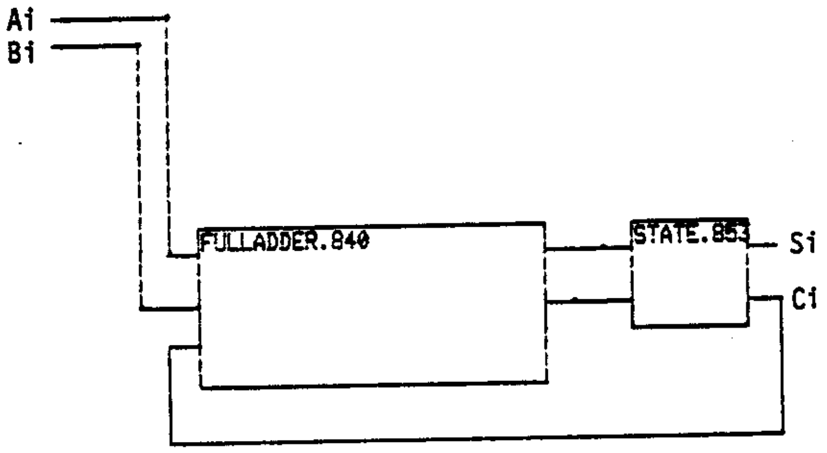


Figure 2.12 CPA Implementation Using Mealy Construct

```
> 1 CPA3.836
> 1 (((A0 1 0 0 0) (B1 1 0 0 0)) ((A1 1 10 0 0) (B1 1 10 0 0))
((A2 1 20 0 0) (B2 1 20 0 0)) ((A3 1 30 0 0) (B3 1 30 0 0)))
> 2 MEALYADDER.837
> 2 (((A0 1 0 0 0) (B1 1 0 0 0)) ((A1 1 10 0 0) (B1 1 10 0 0))
((A2 1 20 0 0) (B2 1 20 0 0)) ((A3 1 30 0 0) (B3 1 30 0 0)))
> 3 FULLADDER.840
> 3 ((CONST.838 0 0 0 0) (A0 1 0 0 0) (B1 1 0 0 0))
> 3 ((WIRE.852 1 34 3 0) (WIRE.850 0 32 2 0))
> 3 STATE.853
> 3 ((WIRE.852 1 34 3 0) (WIRE.850 0 32 2 0))
> 3 ((CONST.838 0 0 0 0) (CONST.839 0 0 0 0))
> 2 ((CONST.838 0 0 0 0) (CONST.839 0 0 0 0))
> 1 ((CONST.838 0 0 0 0) (CONST.839 0 0 0 0))
```

2.4.3 Multi-Operand Carry Save Adder

Multi-operand carry-save adder uses a carry-save logic with feedback in order to perform additions. The final partial-sum and the carries are passed to a carry-propagation adder, described before, to generate the final result. The following is FHDL description of the multi-operand carry-save adder.

```
defun fulladder ()     # (Ci+1 Si)
        [(1 or 2),3] @
        apndl @ [1,halfadder@[2,3]] @
        apndr @ [halfadder@[1,2],3]
enddef                 # (Ai Bi Ci)

defun halfadder ()     # (Ci+1 Si)
        [and,xor]
enddef                 # (Ai Bi)

defun initcsa ()       # initial value of state
        [(& [%0,%0]),%0] @ 1
enddef

defun addall ()                # ((X0 S0ti-1 0)...(Xn-1 Sn-1ti-1 Rn-1ti-1))
        &apndr @ trans @ [1@tlr@1,2]
enddef                 # (((S0ti-1 0)...(Sn-1ti-1 Rn-1ti-1) Cnti-1)
                       #  (X0 ... Xn-1))

defun rearrangeoutput ()
                       # ((S0 0)(S1 0)(S2 C1)...(Sn-1 Cn-2))
        pair @ apndl @ [%0,id] @ concat
        @ & reverse
enddef                  # ((0 S0)(C1 S1)...(Cn-1 Sn-1))

defun csa ()            # ((S0ti 0)...(Sn-1ti Rn-1ti) Cnti)
        [tlr,1@last] @
        rearrangeoutput @
        &fulladder
       @ addall
enddef                  # (((S0ti-1 0)...(Sn-1ti-1 Rn-1ti-1) Cnti-1)
                        #  (X0 ... Xn-1))

defun multiopcsa () # (((S0t1 0)...(Sn-1t1 Rn-1t1))Cnt1)
                    # ...
                    # ((S0tm 0)...(Sn-1tm Rn-1tm))Cntm))
        seq initcsa seqfunc csa seqend
enddef                      # ((A0t1...An-1t1)...(A0tm...An-1tm))

defun multiopadder ()       # ((Cnt1 Pn-1t1...P0t1)...
                             (Cntm Pn-1tm...P0tm))
        $(apndr @ [cpa@1,2])
       @ $([apndl @ [[[%0,%0]],1],2])
       @ multiopcsa
enddef                  # ((A0t1...An-1t1)...(A0tm...An-1tm))
```

Figure 2.13 illustrates a high level logic schematic of the adder [Hwan79]. The important aspect of this example is the use of the sequential element in "csa" function and the use of "time apply to all" to interface between sequential part and combinational one. "Time apply to all" functional

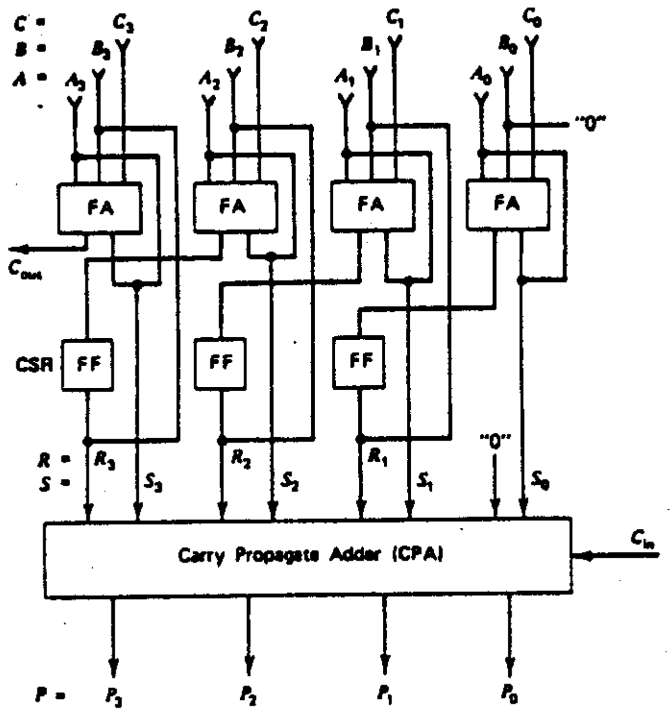


Figure 2.13 A Multi-operand Carry-save Adder

generates multiple copies of carry-propagation adder in the behavioral domain, in order to match the time object generated by the sequential element "multiopcsa". In structural domain only single carry-propagation adder is generated by the symbolic interpreter, because the outputs of "multiopcsa" are folded in time.

Figure 2.14, 2.15, and 2.16 show the logic diagram of functions "multiopadder", "multiopcsa", and "csa" respectively. The following is some sample functional simulation of "multiopadder" and "multiopcsa".

```
->> multiopadder:((1 1 1 1 1)(1 1 1 1 1))
```

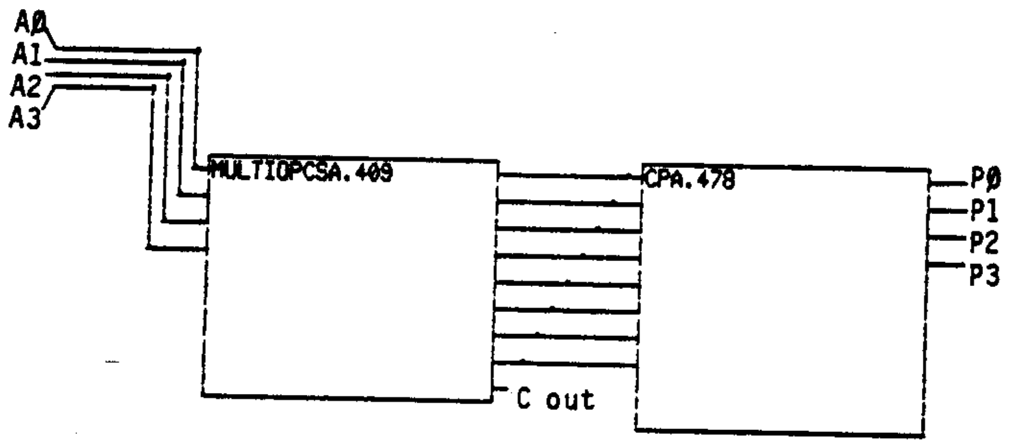


Figure 2.14 Logic Diagram of Multi-Operand-Adder

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

multiopcsa:((0 0 0 0)(0 0 0 1)(0 1 1 0))

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

multiopadder:((0 0 0 0)(0 0 0 1)(0 1 1 0))

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


multiopadder:((0 0 0 0)(0 0 0 1)(0 1 1 0)(1 0 0 0))

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

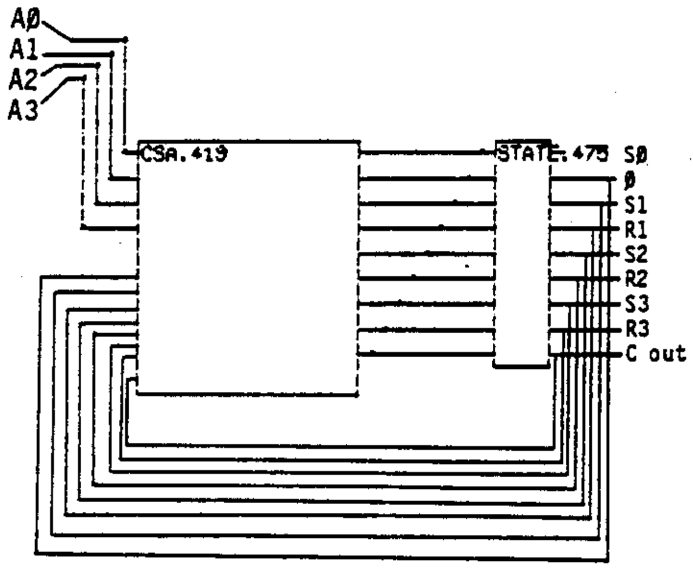


Figure 2.15 Logic Diagram of Sequential CSA

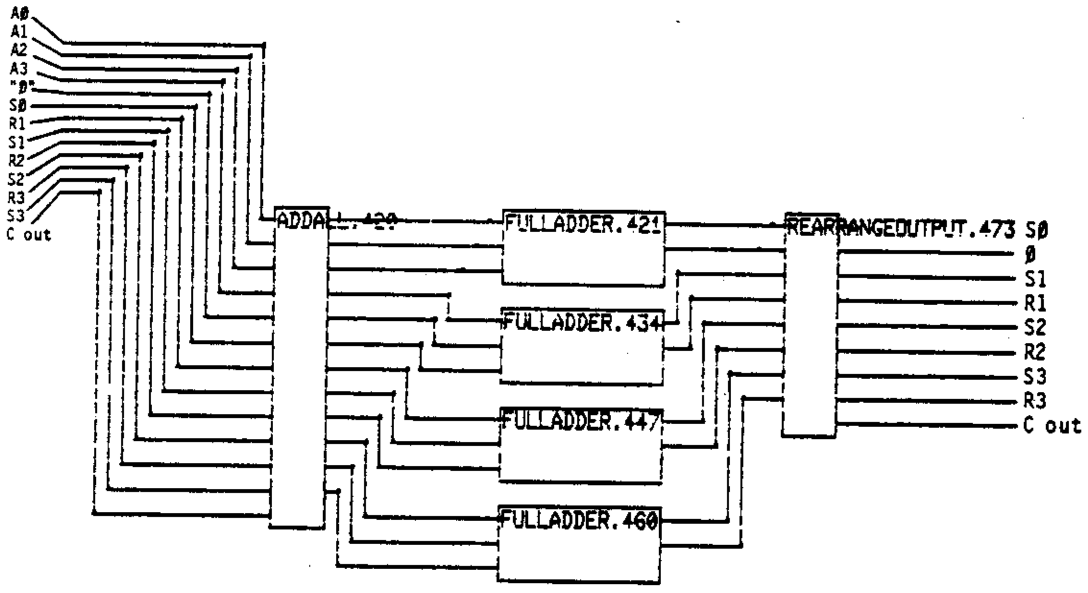


Figure 2.16 Logic Diagram of a CSA

```
->> multiopadder:((1 1 1 1)(1 1 1 1)(0 0 0 1))

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

The simulation of "multiopadder" in the structural domain is given below.

```
> 1 MULTIOPADDER.408
> 1 (((IN11 1 0 0 0) (IN12 1 0 0 0) (IN13 1 0 0 0) (IN14 1 0 0 0))
((IN21 1 0 0 0) (IN22 1 0 0 0) (IN23 1 0 0 0) (IN24 1 0 0 0)))
> 2 MULTIOPCSA.409
> 2 (((IN11 1 0 0 0) (IN12 1 0 0 0) (IN13 1 0 0 0) (IN14 1 0 0 0))
((IN21 1 0 0 0) (IN22 1 0 0 0) (IN23 1 0 0 0) (IN24 1 0 0 0)))
> 3 CSA.419
> 3 (((((CONST.410 0 0 0 0)(CONST.411 0 0 0 0))((CONST.412 0 0 0 0)
(CONST.413 0 0 0 0)) ((CONST.414 0 0 0 0) (CONST.415 0 0 0 0))
 ((CONST.416 0 0 0 0) (CONST.417 0 0 0 0))) (CONST.418 0 0 0 0))
((IN11 1 0 0 0) (IN12 1 0 0 0) (IN13 1 0 0 0) (IN14 1 0 0 0)))
> 3 ((((CONST.474 0 0 0 0)(WIRE.431 1 32 2 0))((WIRE.433 0 34 3 0)
(WIRE.444 1 32 2 0)) ((WIRE.446 0 34 3 0) (WIRE.457 1 32 2 0))
((WIRE.459 0 34 3 0) (WIRE.470 1 32 2 0))) (WIRE.472 0 34 3 0))
> 3 STATE.475
> 3 ((((CONST.474 0 0 0 0) (WIRE.431 1 32 2 0))
 ((WIRE.433 0 34 3 0) (WIRE.444 1 32 2 0)) ((WIRE.446 0 34 3 0)
(WIRE.457 1 32 2 0)) ((WIRE.459 0 34 3 0) (WIRE.470 1 32 2 0)))
(WIRE.472 0 34 3 0))
> 3 ((((CONST.410 0 0 0 0) (CONST.411 0 0 0 0))
((CONST.412 0 0 0 0) (CONST.413 0 0 0 0)) ((CONST.414 0 0 0 0)
(CONST.415 0 0 0 0)) ((CONST.416 0 0 0 0) (CONST.417 0 0 0 0)))
(CONST.418 0 0 0 0))
> 2 ((((CONST.410 0 0 0 0)(CONST.411 0 0 0 0))((CONST.412 0 0 0 0)
(CONST.413 0 0 0 0)) ((CONST.414 0 0 0 0) (CONST.415 0 0 0 0))
((CONST.416 0 0 0 0) (CONST.417 0 0 0 0))) (CONST.418 0 0 0 0))
> 2 CPA.478
> 2 ((((CONST.476 0 0 0 0)(CONST.477 0 0 0 0)))((CONST.410 0 0 0 0)
(CONST.411 0 0 0 0)) ((CONST.412 0 0 0 0) (CONST.413 0 0 0 0))
((CONST.414 0 0 0 0) (CONST.415 0 0 0 0)) ((CONST.416 0 0 0 0)
(CONST.417 0 0 0 0)))
> 2 ((WIRE.490 0 32 2 0) (WIRE.504 0 50 4 0) (WIRE.518 0 68 6 0)
(WIRE.532 0 86 8 0))
> 1 ((WIRE.490 0 32 2 0) (WIRE.504 0 50 4 0) (WIRE.518 0 68 6 0)
(WIRE.532 0 86 8 0) (CONST.418 0 0 0 0))
```

The maximum clock rate of "multiopadder" is 34 units of time plus the time delay of the register. The delay of "cpa" function is 86 units of time. Thus, for adding 20 numbers about 20*34+86 (766) units of time is required.

CHAPTER 3

Design of Pipelined Systems using FHDL

3.1 Introduction

Various methods have been considered for improving the speed of a computing system. First, the improvements in implementation technology could easily decrease the switching speed of components and increase the level of integration. Second, the exploitation of parallelism and utilization of many computing resources concurrently may improve the system performance. Third, the decomposition of an algorithm into a fixed number of steps that are to be executed in an overlapped manner for different inputs improves the throughput. This method is called pipelining. Among the mentioned approaches we consider pipelined systems.

The major advantage of pipelining over other parallel design techniques is that *frequently* the same improvement in performance can be obtained for less cost. This happens because an n-stage pipelined is obtained by partitioning a nonpipelined system into n smaller subsystems and then adding registers between the stages. Thus, the dominant additional cost of a pipelined system is due to the added registers and their cost is ***usually*** much smaller than the cost of the stages.

The objective here is to consider the specification and design of a class of pipelined systems using FHDL. In our approach, a synthesis tool (an enhanced version of the symbolic interpreter introduced in section 2.3) is used to obtain a pipelined implementation of FHDL specification.

In the next section, the general concept of pipelining and associated notation is introduced and the requirements for automatic design of pipelined systems is specified. Then, FHDL advantages and disadvantages for synthesizing a system into a pipelined one are considered. Next, the approaches and criteria used by the synthesis tool is discussed. Finally, some examples illustrate the result of the tool and the performance of synthesized systems.

3.2 Pipelining of Digital Systems

Pipelining of a hardware algorithm is the process of dividing or partitioning the algorithm into number of sub-functions or stages such that : [Kogg81]

1. Evaluation of the basic function is equivalent to some sequential evaluation of the subfunctions.

2. The inputs for one subfunction come totally from outputs of previous sub-functions in the evaluation sequence.

3. Other than the exchange of inputs and outputs, there are no interrelationships between subfunctions.

4. The times to compute different subfunctions are approximately

equal.

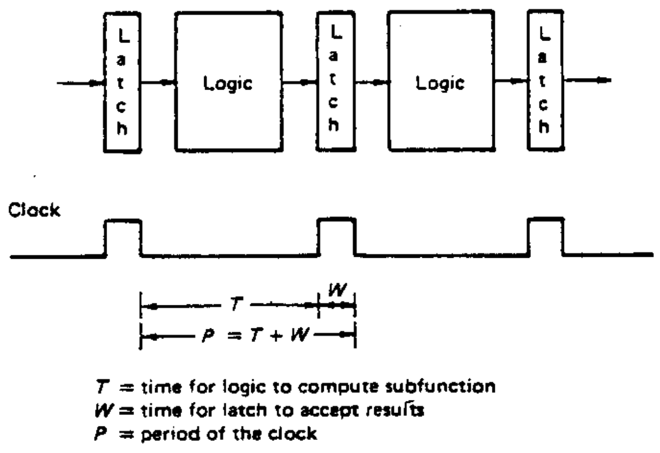
Then each stage is separated from the previous one by a register or latch that holds the intermediate results.

For certain type of algorithms (meeting the above requirements) the use of pipelining can result in a significant increase in performance with only modest increase in cost. For others a gain in performance may not be possible or may only be realized at considerable cost because each evaluation of the basic function or hardware algorithm is relatively dependent of the previous one. This is the case in sequential or recurrence systems which require pipeline with feedback. Currently, we consider only the algorithms suitable for pipelining without feedbacks.

Consider a basic static pipelined system (Figure 3.1), the throughput of the system is calculated using the following relation:

$$\text{Throughput} = \frac{1}{T + W} \qquad (3.1)$$

Where, T is the delay of slowest stage in the pipe, and W is the delay associated with the latch. Note, that the above relation is achieved only when the pipe is completely filled [Jump78, Rama77]. The latch design, is highly technology dependent, can utilize the idiosyncrasies of the target technology to improve the throughput of the system drastically [Earl65]. Also, the control scheme used and packaging can force a lower bound on the value of T and impact the throughput [Cott65]. We did not consider cost as a parameter, because we assumed the algorithm is suitable for pipelining and only a modest cost of latches must be added to the system cost.



Latch
Logic
Latch
Logic
Latch
Clock
T
W
P = T + W
T = time for logic to compute subfunction
W = time for latch to accept results
P = period of the clock

Figure 3.1 A Basic Pipelined System

Our main concern is to partition the high-level language specification into about equal delay stages. In depth analysis of pipelined systems and associated problems is given by Kogge [Kogg81], Ramamoorthy and Li [Rama77], and Cotten [Cott65].

In summary, the basic synthesis step must be able to partition the specification into sub-parts of equal delay and define locations where latches should be inserted. Also, relevant performance data must be collected for evaluation of the result.

3.3 FHDL Support for Pipelining

Two requirements specified in the previous section for partitioning of an algorithm to sub-functions are: (i) the inputs for one sub-function come totally from outputs of previous sub-function in the evaluation sequence, (ii) other than the exchange of inputs and outputs, there are no inter-relationships between sub-functions. These two requirements basically lead to detection of data dependency or parallelism associated with the algorithm. Another requirement for pipelined systems is that the time to compute different sub-functions should be approximately equal. This requires to perform timing analysis on the FHDL specification. Although, timing analysis at the gate level (supported by FHDL) is highly technology dependent, but we use a general timing model to perform this task.

In the following sections, the advantages of FHDL in detection of parallelism, composition capabilities, and partitioning of algorithms (the one specified in FHDL) are discussed and comparison is made with conventional type HDLs. Finally, the disadvantages of FHDL for performing timing analysis of FHDL specification are pointed out.

3.3.1 Detecting Parallelism and Dependencies

In FP one can easily obtain the precedence graph of the computation. That is, the data dependencies are explicit. The symbolic interpreter, introduced in section 2.3, exploits this property of FHDL and the precedence graph of the computation can be obtained easily at a desired level. Also, some constructs such as "construction" functional form or "apply-to-all" functional form provide an easy mean of expressing parallelism. To contrast FP-based approach with a conventional one, let us consider the case of ISPS. ISPS [Barb81, Barb80] provides the following construct to describe parallel operation:

```
condition --> action 1; action 2; ... ; action n
```

All the above "actions" are performed in parallel. ISPS behavioral description is compiled and translated into an internal data flow representation called the Value Trace. The data flow graph is a form of precedence graph of computation. Then, the Value Trace is used by other synthesis tools to perform various transformation on the design [Thom83]. In most cases, obtaining the data flow graph of an imperative language is a computationally intensive task. Thus, FHDL provides a computationally more attractive environment than imperative language based HDLs in detecting parallelism.

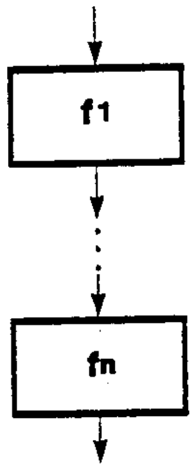
3.3.2 Composition Capability and Partitioning

The method of "divide and conquer" has been used extensively in order to handle the complexity of today's design problems [Sequ83]. FHDL provides several features for decomposing and abstracting the design. First, the generality of the combining forms in FP encourages hierarchical program development. Second, the use of "composition" functional form provides an easy mean of decomposition of a problem into logical subparts. Consider the following two examples. The first description is written in a conventional language, while the second one is in FP.

(1) s1; s2; s3; ... ; sn

(2) fn @ ... @ f3 @ f2 @ f1

The first construct implies a control sequence or dependency between the statements. It is difficult to detect data dependency for decomposition in the first construct due to access of global variables and operations on data structures. Dependency detection becomes more complicated as the programmer tries to minimize storage by re-using the same global variable over and over again [Arvi80]. In other hand, the FP construct in the above enforces the data dependency between the functions and actually reveal the following structure:



If "f3" needs some of results generated by "f1", then "f2" has to pass those explicitly by use of functions such as "id", or "select", otherwise "f3" has access to "f2" results only.

FP composition functional form provides a mechanism for logical decomposition of a program. The above structure revealed by FP closely resembles a pipelined system. Note that for actual implementation, number of functions must be combined together into a single stage in order to meet the partitioning criteria. The synthesis tool implemented in this work uses the above property of composition functional form for partitioning of systems.

3.3.3 Purely Hierarchical Approach

When a function is used in the behavioral domain to specify an operation the concern is the transformation performed on objects. A single function that is used to specify a single operation in two different places may need to have two different interpretations at the lower level of abstraction (i.e. structural domain). Let us elaborate on this. Consider an "and" function that performs logical "and" operation on a pair of inputs. The "and" function may be used at one point to drive three other similar gates (fanout of three) while in another place "and" is used to drive only one more gate. These two "and" functions, at the lower level of abstraction, must be implemented differently and one cannot simply use the same mapping (from one level of abstraction to lower one) for both instances. In case of generating layout or artwork, one cannot use the same cell for both "and" gates. The use of purely hierarchical approach, mapping of high-level primitives to low level constructs one-to-one, leads to a non-optimal design [vanC79]. Van Cleemput studied the problem associated with the use of purely hierarchical approach in the layout domain, in the gate level, a function (i.e. "and") with different fanout has different timing behavior and could result to a faulty implementation if gate fanout is not considered.

The timing analysis of FHDL specification could suffer from the same problem. There exists two solutions: First, the introduction of a set of primitives that perform the same function but have different interpretation at the lower level. Elias and Wetzel [Elia83] used this technique. Second, the primitives at the lower level of abstraction are identified with attributes as introduced in section 2.3.4. The problem with the first solution is that the designer in the behavioral domain must consider various aspects of implementation rather than focusing on design of a functionally correct system. The second solution is used, because it conforms better with hierarchical program design encouraged by FP and the generality of the combining forms in functional languages. The disadvantages of this method are illustrated in the next section.

With the understanding of advantages and disadvantages of FHDL to meet the partitioning requirements of pipelined systems, a synthesis tool(symbolic interpreter) is introduced next. This tool is used to partition a system into stages. The interpreter also provides some statistics in order to evaluate the throughput of the system.

3.4 Partitioning of Digital Systems Using FHDL

A synthesis method has been integrated into the symbolic interpreter described in section 2.3 to partition the FHDL specification into stages. This section describes the algorithms used by this method to perform timing analysis and decomposition of the specification. In the next section, the capabilities of the interpreter are illustrated by several examples.

3.4.1 Timing Analysis

The timing analysis is performed through the use of attributes. The two attributes introduced in section 2.3.1 (time delay and number of logic level) are used. During the symbolic interpretation, each logic gate operates on these attributes. The following is a part of output generated by the symbolic interpreter.

```
> 1 AND.442
> 1 (WIRE.421 1 6 1 0)(WIRE.407 1 15 2 0)
> 1 (WIRE.443 1 24 3 0)
```

The "and" gate "AND.442" has two inputs called "WIRE.421" and "WIRE.407" and both are at logic level "1". "WIRE.421" so far had the delay of 6 unit of time and passed through one stage of delay, while "WIRE.407" passed through two levels of gate and has delay of 15 unit of time. The output of "AND.442" is a signal called "WIRE.443" with logic value of "1", and time delay of twenty four, because the worst case delay of "and" gate is 9 unit of time (this parameter is set by the user and depends on the technology). The time delay of the output signal is calculated by adding the delay associated with the gate to the maximum delay of the inputs. For "WIRE.443" the number of logic levels is three because one of the inputs ("WIRE.407") has been generated by two levels of logic and "AND.442" is the third level. The values used for worst case time delay of gates are based on tau-model for NMOS technology described in detail by Mead and Conway [Mead80], and Hon and Sequin [Hon80]. These values can be defined by the user (see Appendix A for details).

The major disadvantage of using attributes for timing analysis is that in some technologies the fan-out of a gate affects significantly the time delay of the gate and during the interpretation of a gate the information about the fan-out is not known. A two-step interpretation which uses the output of interpreter in the second step is possible and can use more complex timing model. This case is not considered in this thesis.

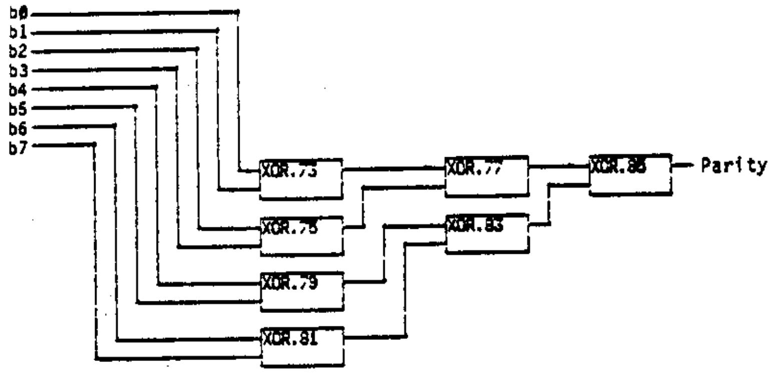
3.4.2 Partitioning Criteria

Now, with some understanding of the timing analysis of FHDL specification, we can partition a system into equally timed stages. The use of "composition" functional form for decomposing a functional program was introduced in section 3.3.2. The symbolic interpreter uses the following method to partition the FHDL specification. During the interpretation, when a "composition" functional form is encountered, the maximum value of time delay and number of logic level of all the atoms that are passed to the second function is calculated. If either of these values exceed the user specified value for the maximum time delay or number of logic level per stage then a latch is required. In this case, a register or latch is generated and all attributes of all those atoms except pipe-stage are set to zero. The pipe-stage attribute is incremented by one since this is the start of a new stage. Also, when the boolean functions operate on attributes they check the pipe-stage attribute of all inputs. If they do not match, the signals are buffered, so that all inputs have the same pipe-stage attributes.

The above method suffers from one problem. Consider the following FHDL construct used to generate odd parity.

```
defun oddparity ()
        | xor
enddef
```

The implementation is shown below.



Since the above specification generates a large system without the use of "composition" functional form, the partitioning algorithm is unable to identify stages in the middle of the structure. This problem can be overcome with the use of the following equivalent specification.

```
defun oddparity ()
        | (xor @ id)
enddef
```

3.5 Pipelining Example

The conditional sum adder (CSA) example, introduced in section 2.4.1, is now used to demonstrate the pipelining capabilities of the symbolic interpreter. Figure 3.2 shows a six-bit pipelined CSA. The pipelining (partitioning) requirement is that each stage of the pipe should have no more than four logic levels. The statistics gathered by the interpreter are as follows:

Register	Time Delay	Logic Levels	Stage
1	38	4	1
2	38	4	1
3	9	1	1
4	9	1	1
5	9	1	1
6	13	2	1
7	13	2	1
8	13	2	1
9	9	1	1
10	9	1	1
11	9	1	1
12	13	2	1
13	13	2	1
14	13	2	1
15	38	4	1
16	38	4	1
17	9	1	1
18	9	1	1
19	9	1	1
20	13	2	1

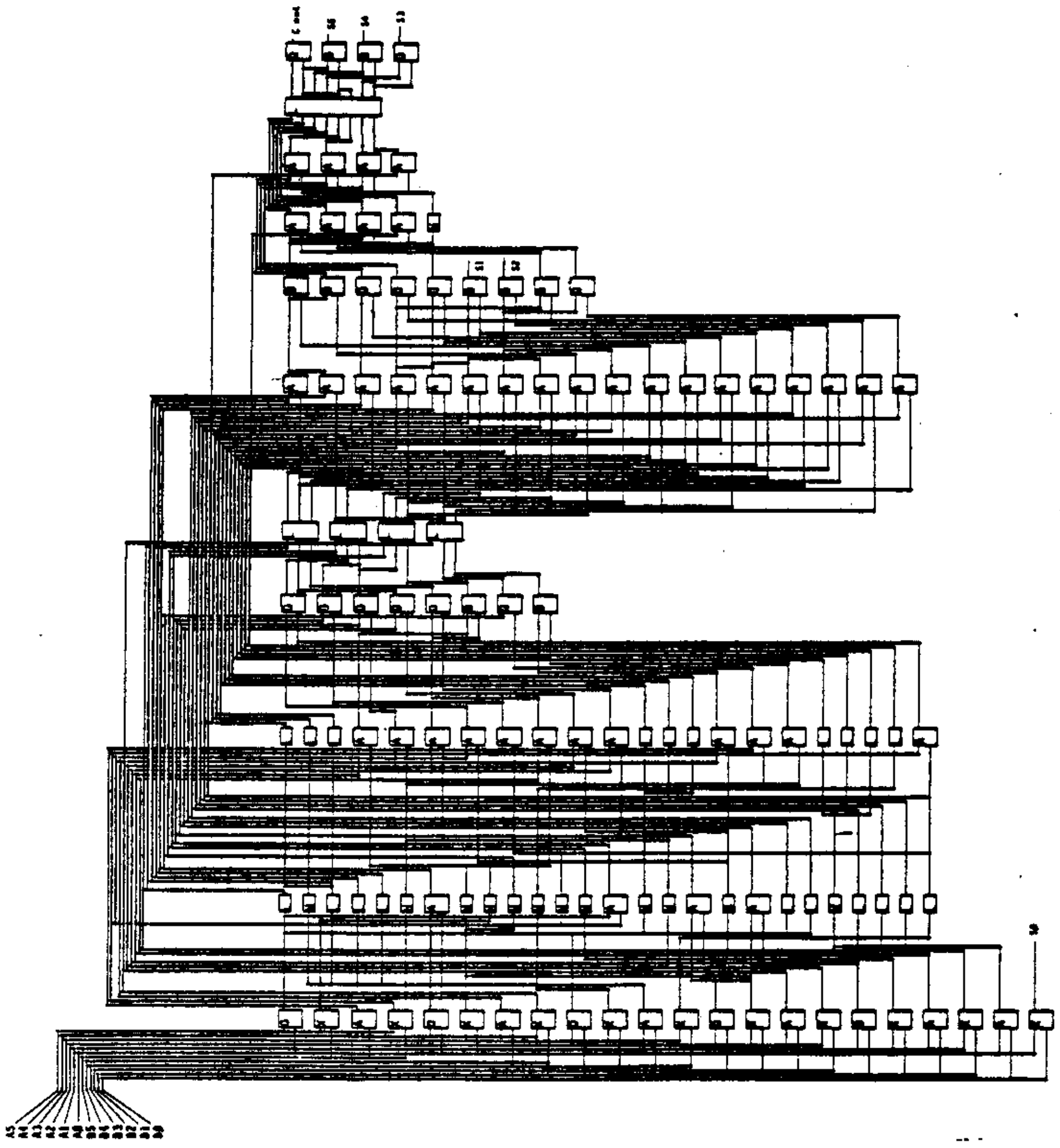


Figure 3.2 6 Bit Pipelined CSA -- Version 1

21	13	2	1
22	13	2	1
23	20	2	1
24	16	1	1
25	31	4	2

As shown in Figure 3.2, there are 25 registers and three stages in the pipe. "L", "A", "O", "X", and "N" stand for "latch" or register, "and" gate, "or" gate, "xor" gate, and inverter, respectively. Registers 1 through 24 separate stages 1 and 2. Register 25 separates stage 2 and 3. The time delay and the number of logic levels shown above are the values of the worst case time delays and number of logic levels. The latches (registers) are numbered from upper left corner of the Figure in column major order. The throughput of the pipe is 1/(38+11) or 1/49 (assuming that the time delay of a latch in NMOS technology is 11 units). That is, every 49 units of time a result is produced by the pipe.

Figure 3.3 shows another pipelined six-bit CSA. Each stage has time delay of about 20 time units. The data collected by the interpreter is as follows:

Register	Time Delay	Logic Levels	Stage
1	20	2	1
2	22	3	2
3	22	3	2
4	20	2	3
5	0	0	2
6	0	0	2
7	0	0	2
8	4	1	2
9	4	1	2
10	4	1	2
11	0	0	2
12	0	0	2
13	0	0	2
14	4	1	2
15	4	1	2
16	4	1	2
17	20	2	1

18	22	3	2
19	22	3	2
20	9	1	1
21	9	1	1
22	9	1	1
23	13	2	1
24	13	2	1
25	13	2	1
26	22	3	3
27	0	0	3
28	0	0	3
29	0	0	3
30	0	0	3
31	0	0	3
32	0	0	3

The throughput is 1/(22+11) which is 1/33 with four stages in the pipe. Registers 1, 17, and 20 through 25 separate stage 1 and 2. Registers 4 and 26 through 32 separate stage 3 and 4. Stage 2 and 3 are separated by the remaining of registers.

Figure 3.4 shows the same CSA, but the partitioning criteria requires that each stage has about 30 time units of delay. The following statistics are gathered:

Register	Time Delay	Logic Levels	Stage
1	38	4	1
2	38	4	1
3	9	1	1
4	9	1	1
5	9	1	1
6	13	2	1
7	13	2	1
8	13	2	1
9	9	1	1
10	9	1	1
11	9	1	1
12	13	2	1
13	13	2	1
14	13	2	1
15	38	4	1
16	38	4	1
17	9	1	1
18	9	1	1
19	9	1	1

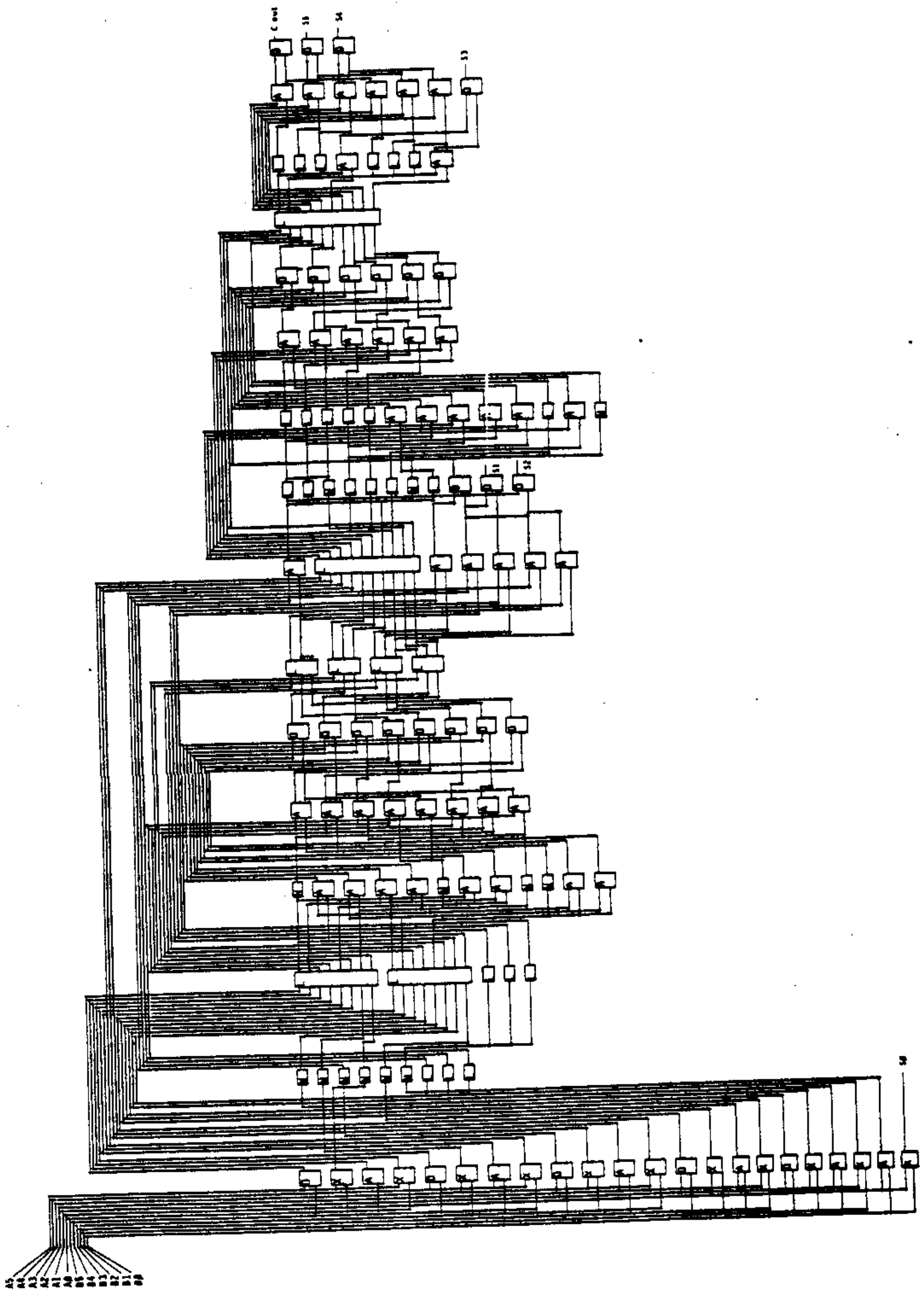


Figure 3.3 6 Bit Pipelined CSA -- Version 2

20	13	2	1
21	13	2	1
22	13	2	1
23	20	2	1
24	16	1	1
25	31	4	2

This partitioning criteria resulted in the same system as the one shown in Figure 3.2.

The results of the symbolic interpreter illustrated in this part provides at least a first order optimization in throughput of the system. As demonstrated in the examples, the resulted throughput is improved by a factor of two or three.

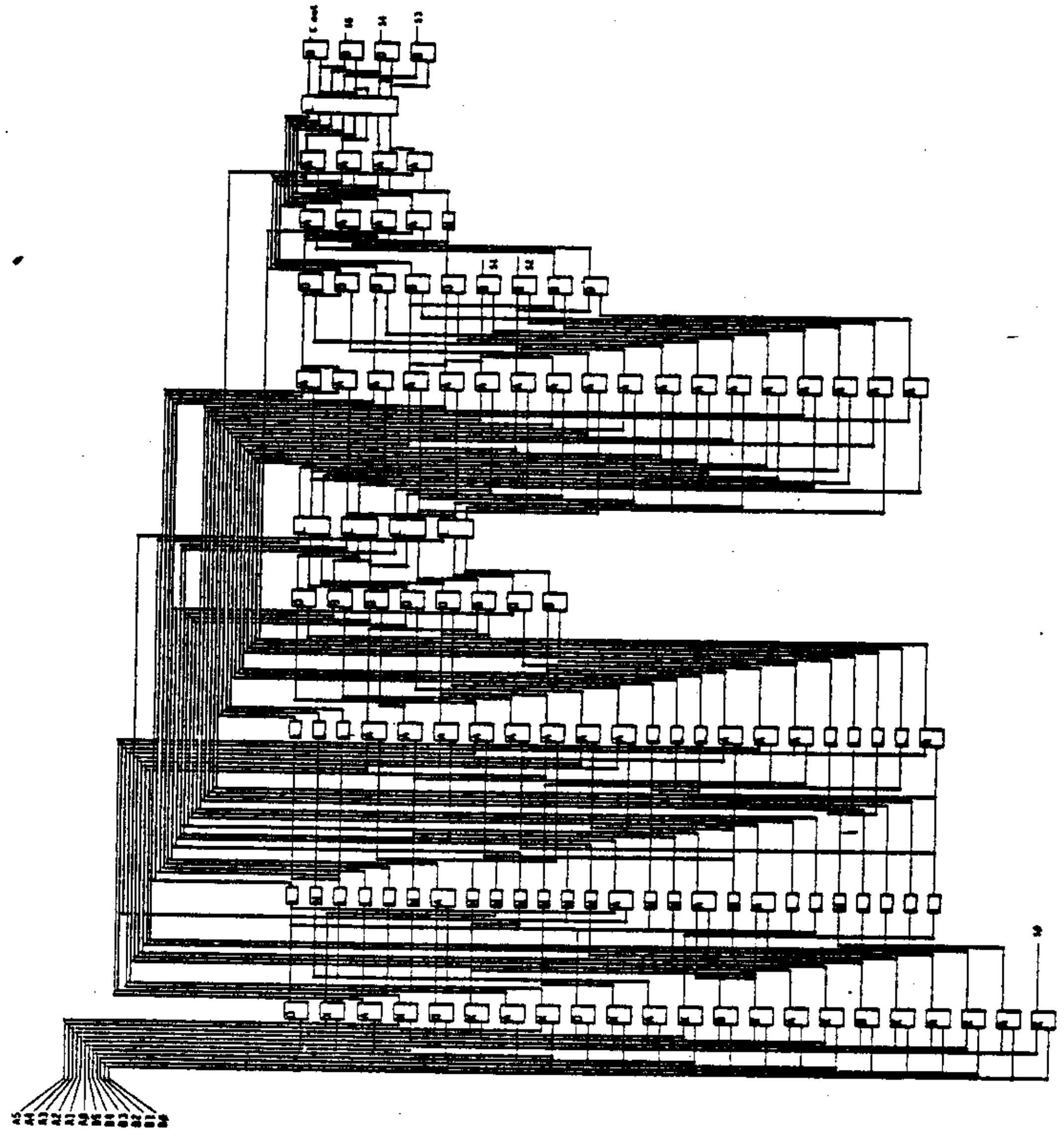


Figure 3.4 6 Bit Pipelined CSA -- Version 3

CHAPTER 4

Conclusion and Further Research

A functional HDL (FHDL) based on the Backus's FP was introduced extending the previous work in this field which used FP to describe combinational systems only. FHDL supports the specification of sequential systems as well. The sequential systems are modeled by an equivalent iterative network at the behavioral level. Then, a symbolic interpreter is used to implement the specification automatically at the gate level. System attributes are introduced so that the information about system performance and design parameters can be extracted. The three attributes used are the time delay, the number of gate delays, and the pipeline stage number. Also, the definition of functions is extended so they operate on attributes as well as objects.

The above three attributes are used to optimize the throughput of the systems specified in FHDL. The higher throughput is achieved by organizing the system in a pipelined fashion. A synthesis method is provided to partition automatically the FHDL specification into equally timed stages. Composition functional form in FHDL is used for decomposing the specification. Finally, a number of examples demonstrate the pipelining capability of the existing tools.

Further research is required to specify a general class of pipelined systems (pipes with feedback) using the sequential constructs introduced in this work. Kogge uses recurrence systems as a general class of pipes with feedback [Kogg81]. "Sequence" functional form can be used to specify recurrence equations. This leads to the use of mathematical properties of functional languages to pipeline *m*th order recurrence system. Also, various control schemes must be studied for implementation of pipelined systems with feedback. Automatic control signal generation for pipelined systems should be part of future research.

APPENDIX A

FHDL Interpreters

The functional and symbolic interpreter for FHDL is written in T [Rees83] which is a dialect of LISP [McCa60]. The functional interpreter resides under "~ farshad/tfp" directory. To run the interpreter type "t", when the system responds with ">" then type "(load "fp.t")". The system will respond with "> " then type "(fp)". Now, the FHDL interpreter is running, the interpreter prompt is "->> ". The latest information about this software is in a file called "~ farshad/tfp/README". Further information on this interpreter is available in "~ micro/dorab/doc".

The symbolic interpreter resides under "~ farshad/tsfp". The user must run T first. Then, inside T, type "(load "sfp.t")" followed by "(sfp)", similar to the functional interpreter. Read file "~ farshad/tsfp/README" for latest updates on the system.

The symbolic interpreter is controlled by a set of parameters. Parameters are described in detail in file "~ farshad/tsfp/param.t". In order to change the value of a parameter during the interpretation, type ")break" to go back to T. Type "(set <param> <value>)", where "<param>" is the name of the parameter and "<value>" is the new value of the parameter. Then, type "(ret)" to go back to interpreter. To get the collected data on partitioning results, go to T and type "*stagedata*", similar data as the one shown in section 3.5 will be provided.

The output generated by the symbolic interpreter is the precedence graph of the computation and is placed in a file, currently called "simoutput", see "param.t" to change the name. This graph can be used to draw logic diagram at the gate level. The graphic software is under "~ farshad/graph" and is called "graph". The output of symbolic interpreter must be copied or moved to a file called "dataf" before the graphic program can be executed. Detail information on the use of graphic software is in "~ micro/sllu/doc".

References

[Arvi80] Arvind, "Decomposing a Program for Multiple Processor Systems," *Proceedings of the 1980 International Conference on Parallel Processing,* August 1980, pp. 7-14.

[Back78] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communication ACM,* Vol. 21, No. 8, Aug. 1978, pp. 613-641.

[Bade83a] S. Baden, *Berkeley FP User's Manual,* U. C. Berkeley, Mar. 1983.

[Bade83] S. B. Baden and D. R. Patel, "Berkeley FP -- Experences with a Functional Programming Language," *Proc. of COMPCON,* Spring 1983, pp. 274-277.

[Barb80] M. R. Barbacci and J. D. Morthcutt, "Applications of ISPS, an Architecture Description Language," *Journal of Digital Systems,* Vol. 4, No. 3, Fall 1980, pp. 221-240.

[Barb81] M. R. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Applications," *IEEE Trans. on Computers,* Vol. C-30, Jan. 1981, pp. 22-40.

[Card81] L. Cardelli and G. Plotkin, "An Algebraic Approach to VLSI Design," in *VLSI 81,* 1981, pp. 173-182.

[Cott65] L. W. Cotten, "Circuit Implementation of High-Speed Pipeline Systems," *AFIPS Proc. FJCC,* 1965, pp. 489-504.

[Davi83] M. Davio, J. P. Deschamps, and A. Thayse, in *Digital Systems with Algorithm Implementation* , John Wiley & Sons, 1983.

[Earl65] J. Earle, "Latched Carry-Save Adder," *IBM Tech. Disclosure Bulletin,* Vol. 7, No. 10, Mar. 1965, pp. 909-910.

[Elia83] N. J. Elias and A. W. Wetzel, "The IC Module Compiler, A VLSI System Design Aid," *Proc. of 20th Design Automation Conference,* 1983, pp. 46-49.

[Erce82] M. D. Ercegovac and T. Lang, in *Digital Systems: Hardware/Firmware Algorithms.*, University of California, Los Angeles, Dec. 1982.

[Erce83] M. D. Ercegovac, *Private Communications*, 1983.

[Fran79] R. E. Frankel and S. W. Smoliar, "Beyond Register Transfer: An Algebraic Approach For Architectural Description," *Proc. of 4th International Conf. on Computer Hardware Description Languages,* Oct. 1979, pp. 1-5.

[Fran81] R. E. Frankel and S. W. Smoliar, "Digital Systems as Mathematical Expressions," *Proc. of COMPCON,* Spring 1981, pp. 414-416.

[Gord81] M. Gordon, "A Model of Register Transfer System with Applications to Microcode and VLSI Correctness," Tech. Rep. Unpublished, 1981.

[Hon80] Robert W. Hon and Carlo H. Sequin, "A Guide to LSI Implementation," Xerox PARC, Palo Alto, CA, Tech. Rep. Report No. SSL-79-7, January 1980.

[Hwan79] K. Hwang, in *Computer Arithmetic,* New York, N.Y., 1979, p. John Wiley.

[Jump78] J. R. Jump and S. R. Ahuja, "Effective Pipelining of Digital Systems," *IEEE Trans. on Computers,* Vol. C-27, No. 9, Sept. 1978, pp. 845-865.

[Kogg81] P. M. Kogge, in *The Architecture of Pipelined Computers,* McGraw Hills, 1981.

[Koha78] Z. Kohavi, in *Switching and Finite Automata Theory,* McGraw Hill, 1978.

[Laht81] D. O. Lahti, "Application of a Functional Programming Language," UCLA Dept. of Computer Science, LA, CA, Tech. Rep. Report No. CSD-810403, 1981.

[Lu83] S. Lu, *Graphic Interface for FP,* UCLA Computer Science Department, FP Group Internal Memo., Oct. 1983.

[Marc79] R. W. Marczynski and P. Bakowski, "What Do the Computer Hardware Description Languages Describe?," *Proc. of 4th International Conf. on Computer Hardware Description Languages,* Oct. 1979, pp. 178-183.

[McCa60] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1," *Communication of ACM,* Vol. 3, No. 4, Apr. 1960, pp. 184-195.

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems,* Reading, Massachusetts, Addison-Wesley, 1980.

[Pate82] D. Patel, "Applicative Languages in the Specification and Implementation of VLSI-oriented Hardware Algorithms," *Ph.D. Prospectus,* UCLA Department of Computer Scince, Dec. 1982.

[Pate84] D. Patel, *UCLA FP User Manual,* UCLA Departement of Computer Science, 1984.

[Quey79] D. Queyssac, "Projecting VLSI's on Microprocessor," *IEEE Sepctrum,* May 1979.

[Rama77] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *Computing Surveys,* Vol. 9, No. 1, Mar. 1977, pp. 61-102.

[Rees83] J. A. Rees, N. I. Adams, and J. R. Meehan, in *The T Manual,* Computer Science Department, Yale University, 1983.

[Sequ83] C. H. Sequin, "Managing VLSI Complexity: An Outlook," *Proceedings of IEEE,* Vol. 71, No. 1, Jan. 1983, pp. 149-166.

[Shiv83] S. G. Shiva, "Automatic Hardware Synthesis," *Proceedings of IEEE,* Vol. 71, No. 1, Jan. 1983, pp. 76-87.

[Skla60] J. Sklansky, "Conditional Sum Addition Logic," *IRE Trans.,* Vol. 9, 1960, pp. 226-231.

[Thom79] D. E. Thomas, "The Automatic Synthesis of Digital Systems," *Proc. IEEE,* Vol. 69, No. 10, Dec. 1979, pp. 1605-1615.

[Thom83] D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan, and R. Walker, "Automatic Data Path Synthesis," *IEEE Computer,* Vol. 16, No. 12, Dec. 1983, pp. 59-73.

[vanC79] W. M. vanCleemput, "Hierarchical Design for VLSI: Problems and Advantages," *Proc. of Caltech Conf. on VLSI,* Jan. 1979, pp. 259-274.